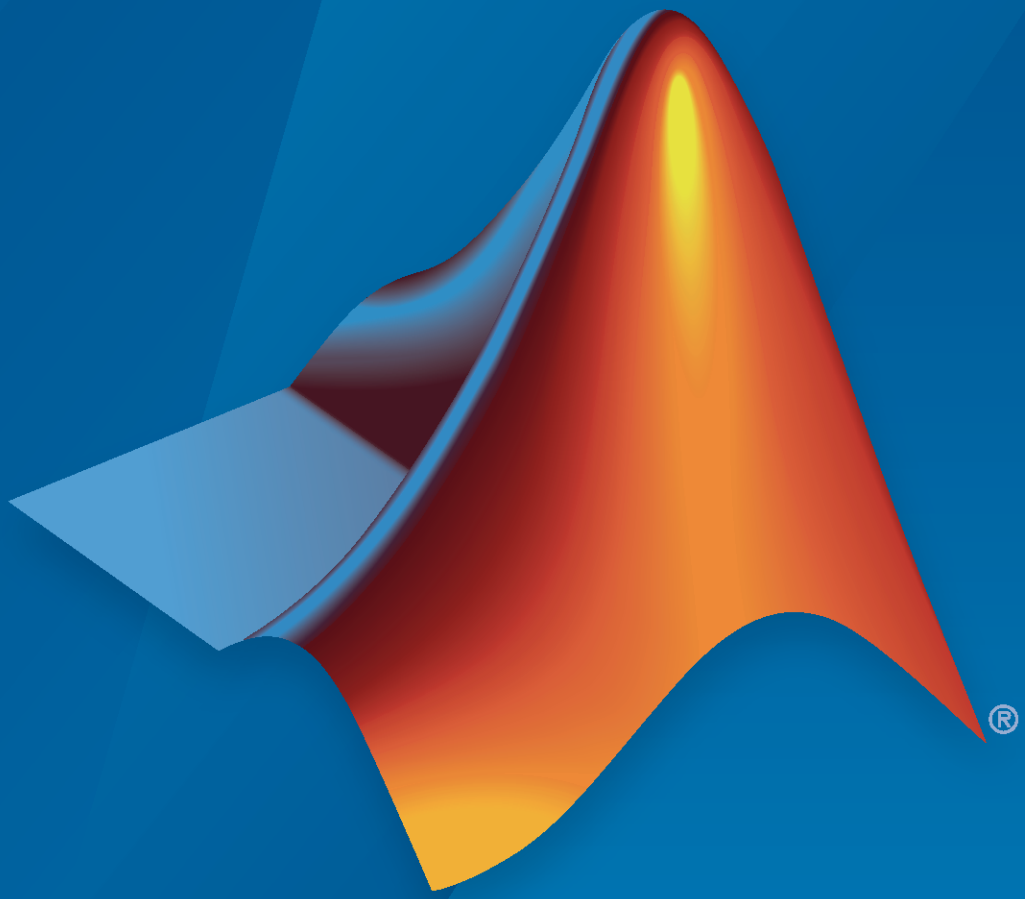
Deep Learning Toolbox™

Reference

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth



MATLAB®

R2021b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Deep Learning Toolbox™ Reference

Revision History

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

Contents

Deep Learning Functions

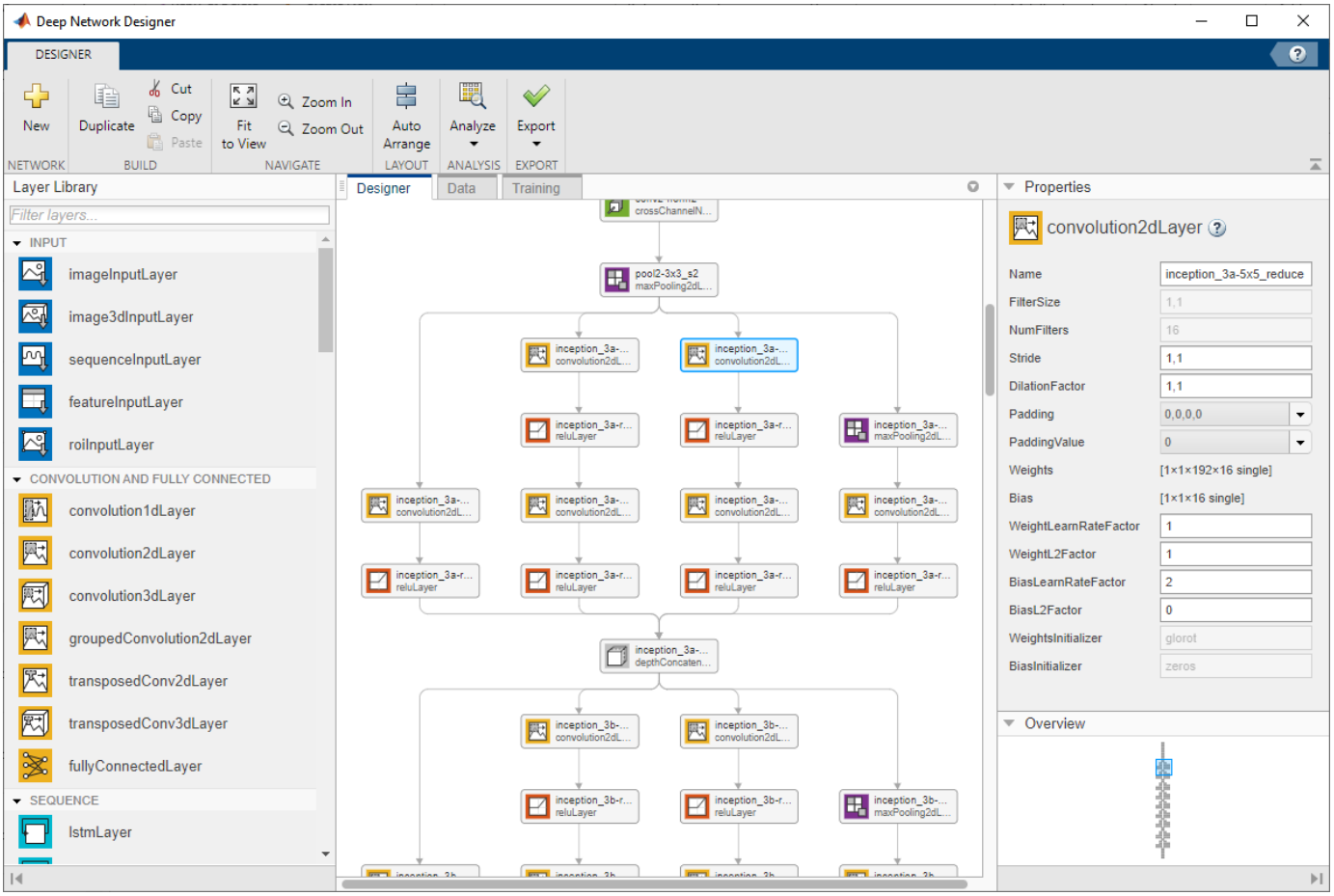
Deep Network Designer

Design, visualize, and train deep learning networks

Description

The **Deep Network Designer** app lets you build, visualize, edit, and train deep learning networks. Using this app, you can:

- Build, import, edit, and combine networks.
- Load pretrained networks and edit them for transfer learning.
- View and edit layer properties and add new layers and connections.
- Analyze the network to ensure that the network architecture is defined correctly, and detect problems before training.
- Import and visualize datastores and image data for training and validation.
- Apply augmentations to image classification training data and visualize the distribution of the class labels.
- Train networks and monitor training with plots of accuracy, loss, and validation metrics.
- Export trained networks to the workspace or to Simulink®.
- Generate MATLAB® code for building and training networks.



Open the Deep Network Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `deepNetworkDesigner`.

Examples

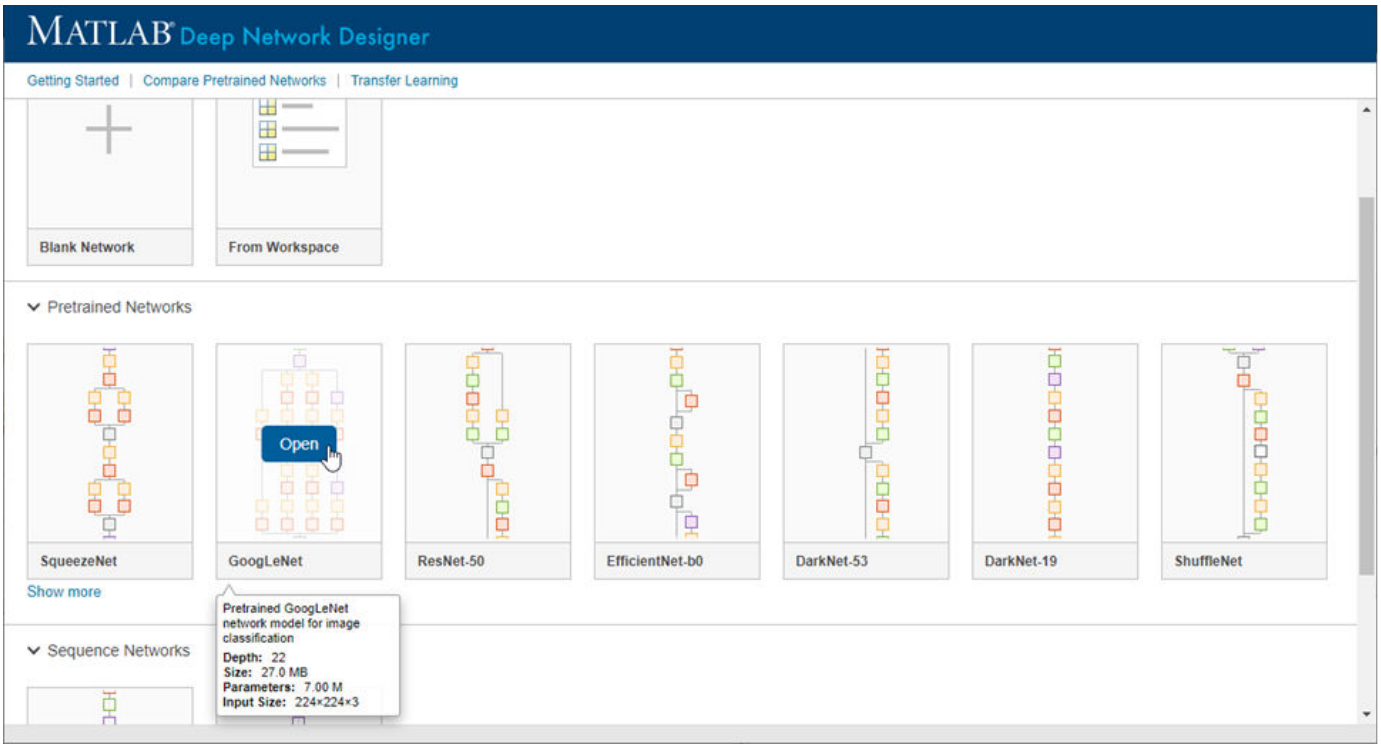
Select Pretrained Image Classification Network

Examine a simple pretrained image classification network in Deep Network Designer.

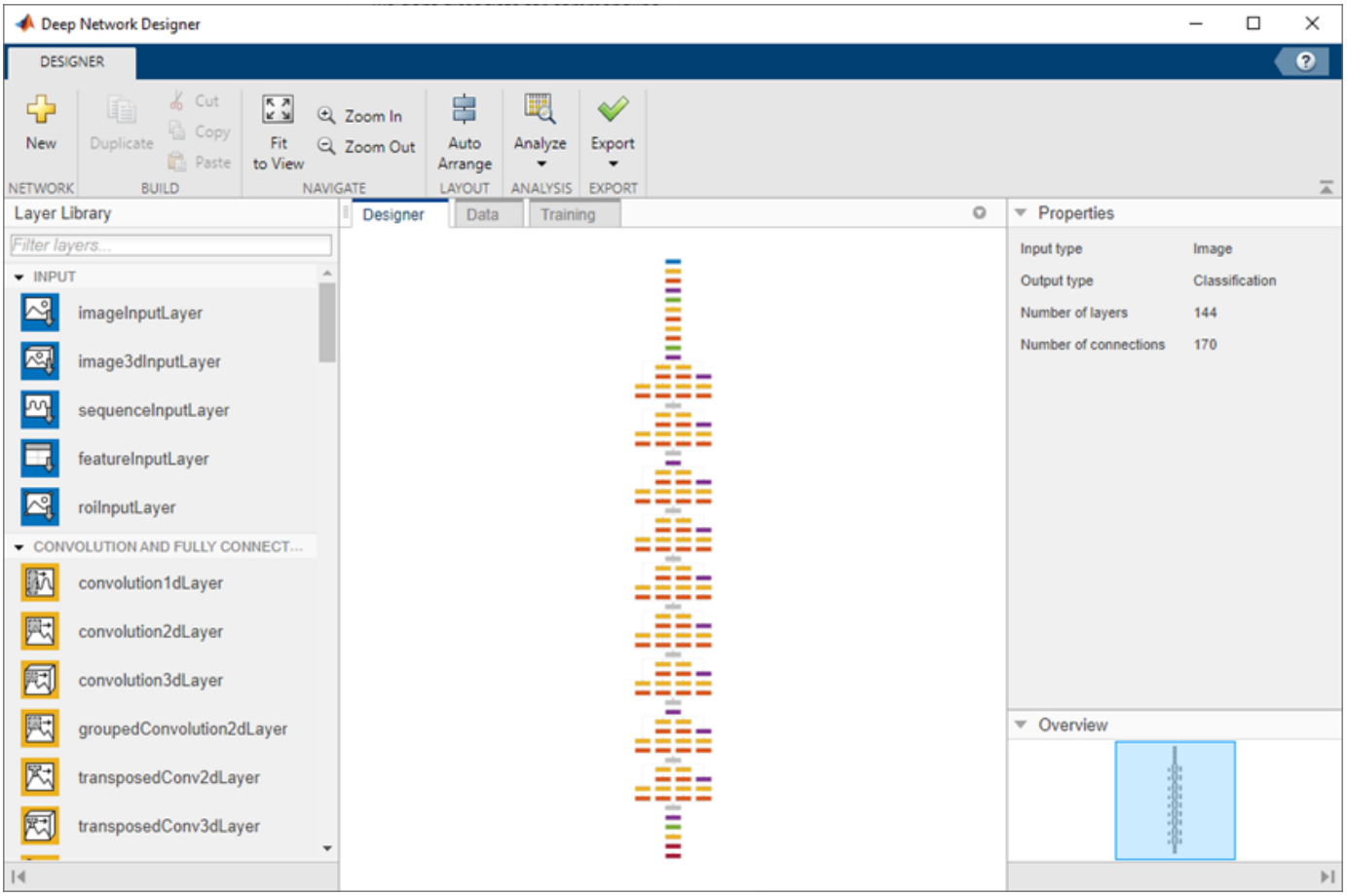
Open the app and select a pretrained network. You can also load a pretrained network by selecting the **Designer** tab and clicking **New**. If you need to download the network, then click **Install** to open the Add-On Explorer.

Tip To get started, try choosing one of the faster networks, such as SqueezeNet or GoogLeNet. Once you gain an understanding of which settings work well, try a more accurate network, such as

Inception-v3 or a ResNet, and see if that improves your results. For more information on selecting a pretrained network, see “Pretrained Deep Neural Networks”.



In the **Designer** pane, visualize and explore the network. For a list of available pretrained networks and how to compare them, see “Pretrained Deep Neural Networks”.



For information on constructing networks using Deep Network Designer, see “Build Networks with Deep Network Designer”.

Edit Pretrained Network for Transfer Learning

Prepare a network for transfer learning by editing it in Deep Network Designer.

Transfer learning is the process of taking a pretrained deep learning network and fine-tuning it to learn a new task. You can quickly transfer learned features to a new task using a smaller number of training images. Transfer learning is therefore often faster and easier than training a network from scratch. To use a pretrained network for transfer learning, you must change the number of classes to match your new data set.

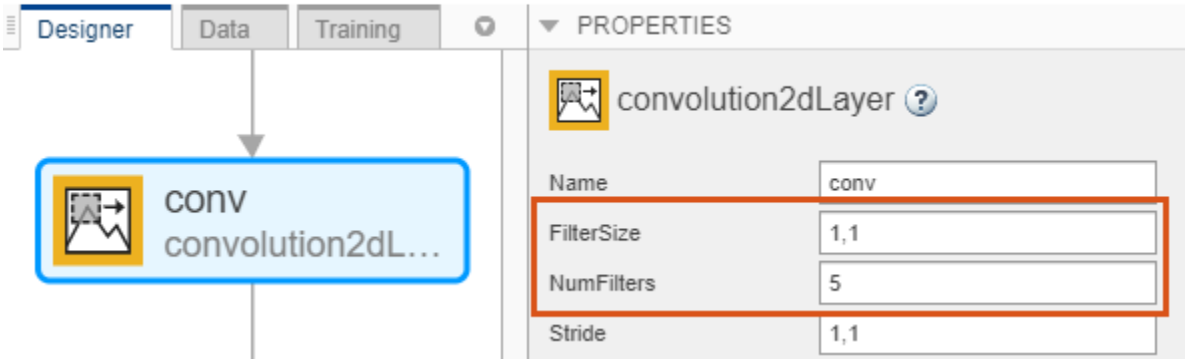
Open Deep Network Designer with SqueezeNet.

```
deepNetworkDesigner(squeezenet)
```

To prepare the network for transfer learning, replace the last learnable layer and the final classification layer. For SqueezeNet, the last learnable layer is a 2-D convolutional layer named `'conv10'`.

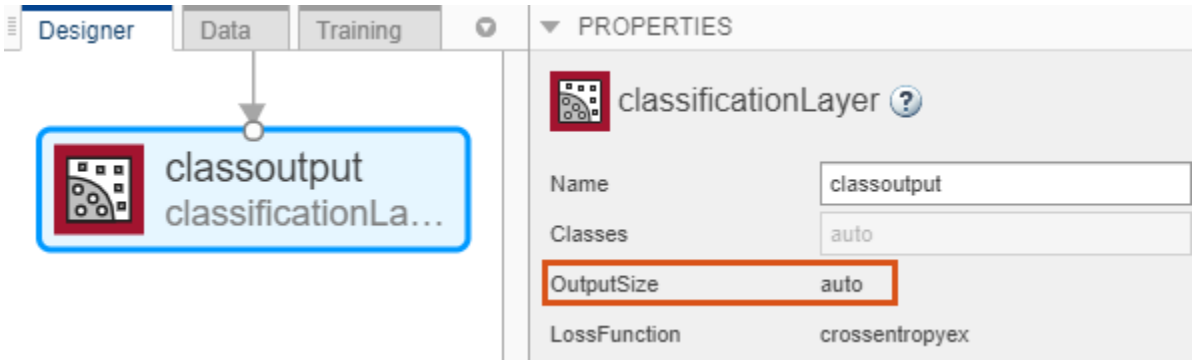
- Drag a new **convolution2dLayer** onto the canvas. Set the **FilterSize** property to `1,1` and the **NumFilters** property to the new number of classes.

- Change the learning rates so that learning is faster in the new layer than in the transferred layers by increasing the **WeightLearnRateFactor** and **BiasLearnRateFactor** values.
- Delete the last **convolution2dLayer** and connect your new layer instead.



Tip For most pretrained networks (for example, GoogLeNet) the last learnable layer is the fully connected layer. To prepare the network for transfer learning, replace the fully connected layer with a new fully connected layer and set the **OutputSize** property to the new number of classes. For an example, see “Get Started with Deep Network Designer”.

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

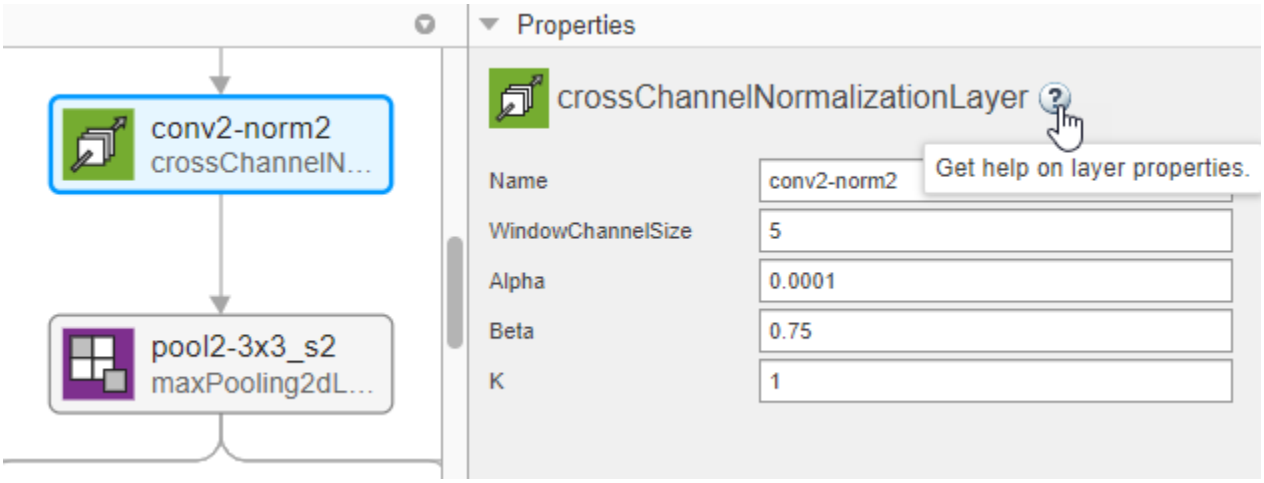


Check your network by clicking **Analyze** in the **Designer** tab. The network is ready for training if **Deep Learning Network Analyzer** reports zero errors. For an example showing how to train a network to classify new images, see “Transfer Learning with Deep Network Designer”.

Get Help on Layer Properties

For help understanding and editing layer properties, click the help icon next to the layer name.

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



For more information about layer properties, see “List of Deep Learning Layers”.

Add Custom Layer to Network

Add layers from the workspace to a network in Deep Network Designer.

In Deep Network Designer, you can build a network by dragging built-in layers from the **Layer Library** to the **Designer** pane and connecting them. You can also add custom layers from the workspace to a network in the **Designer** pane. Suppose that you have a custom layer stored in the variable `myCustomLayer`.

1. Click **New** in the **Designer** tab.
2. Pause on **From Workspace** and click **Import**.
3. Select `myCustomLayer` and click **OK**.
4. Click **Add**.

The app adds the custom layer to the top of the **Designer** pane. To see the new layer, zoom-in using a mouse or click **Zoom in**.

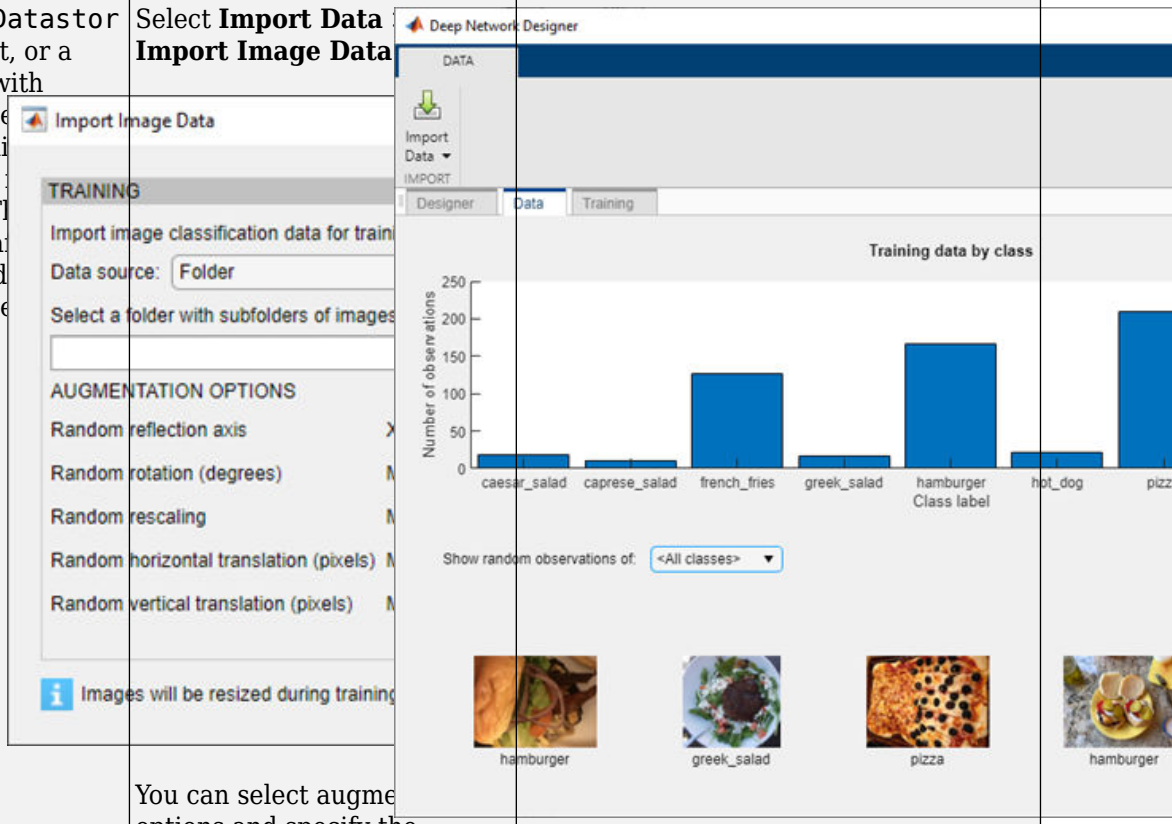
Connect **myCustomLayer** to the network in the **Designer** pane. For an example showing how build a network with a custom layer in Deep Network Designer, see “Import Custom Layer into Deep Network Designer”.

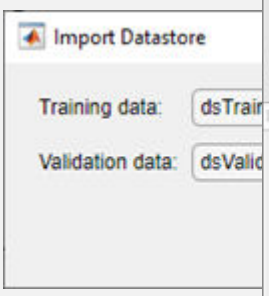
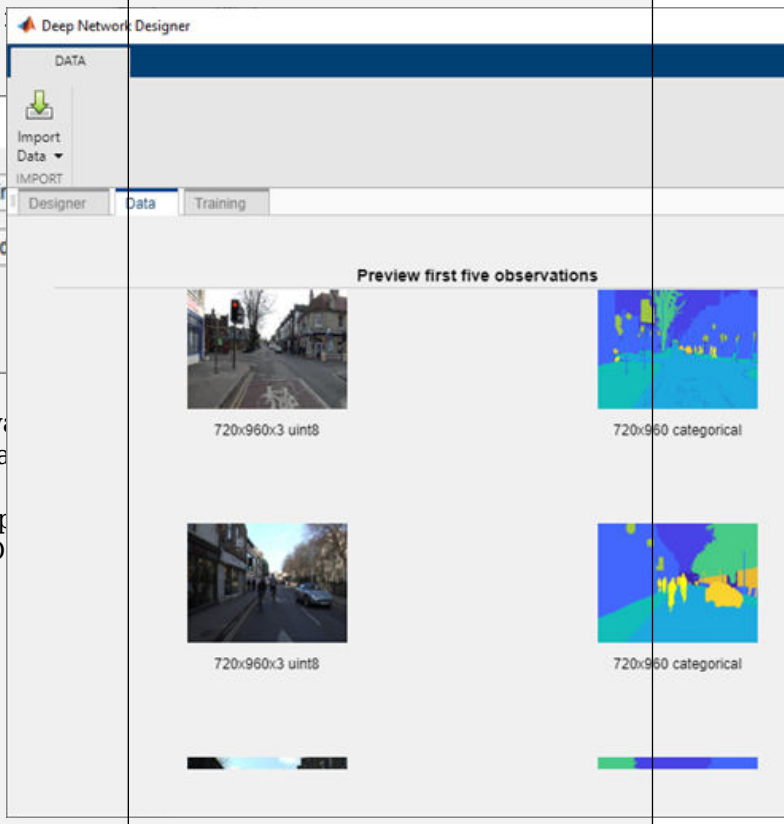
You can also combine networks in Deep Network Designer. For example, you can create a semantic segmentation network by combining a pretrained network with a decoder subnetwork.

Import Data for Training

Import data into Deep Network Designer for training.

You can use the **Data** tab of Deep Network Designer to import training and validation data. Deep Network Designer supports the import of image data and datastore objects. Select an import method based on the type of task.

Task	Data Type	Data Import Method	Example Visualization
Image classification	`ImageDatastore` object, or a folder with subfolde containi images class. Tl labels a sourced subfolde	Select **Import Data** **Import Image Data** You can select augme options and specify the validation data in the Import Image Data dialog box. For more information, see “Import Data into Deep Network Designer”.	

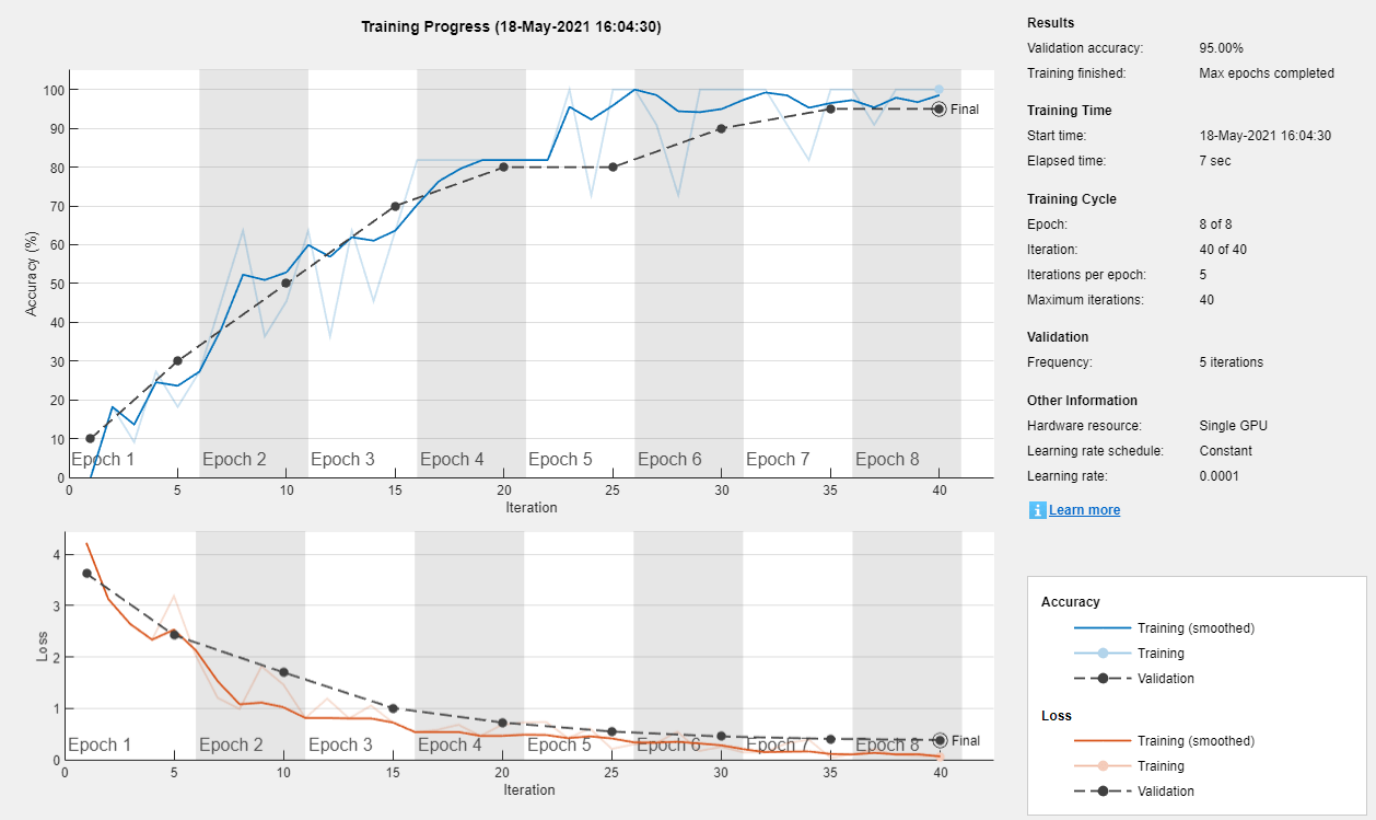
Task	Data Type	Data Import Method	Example Visualization
Other extended workflows (such as numeric feature input, out-of-memory data, image processing, and audio and speech processing)	Datastore. For other extended workflows, use a suitable datastore object. For example, `AugmentedImageDatastore`, `CombinedDatastore`, `pixelLabelImageDatastore`, or custom datastore. You can import and train any datastore object that works with the `trainNetwork` function. For more information about constructing and using datastore objects for deep learning applications, see “Datastores for Deep Learning”.	Select **Import Data** > **Import Datastore**.  You can specify the va data in the Import Da dialog box. For more information, see “Imp into Deep Network D	

Train Network

Train deep neural networks using Deep Network Designer.

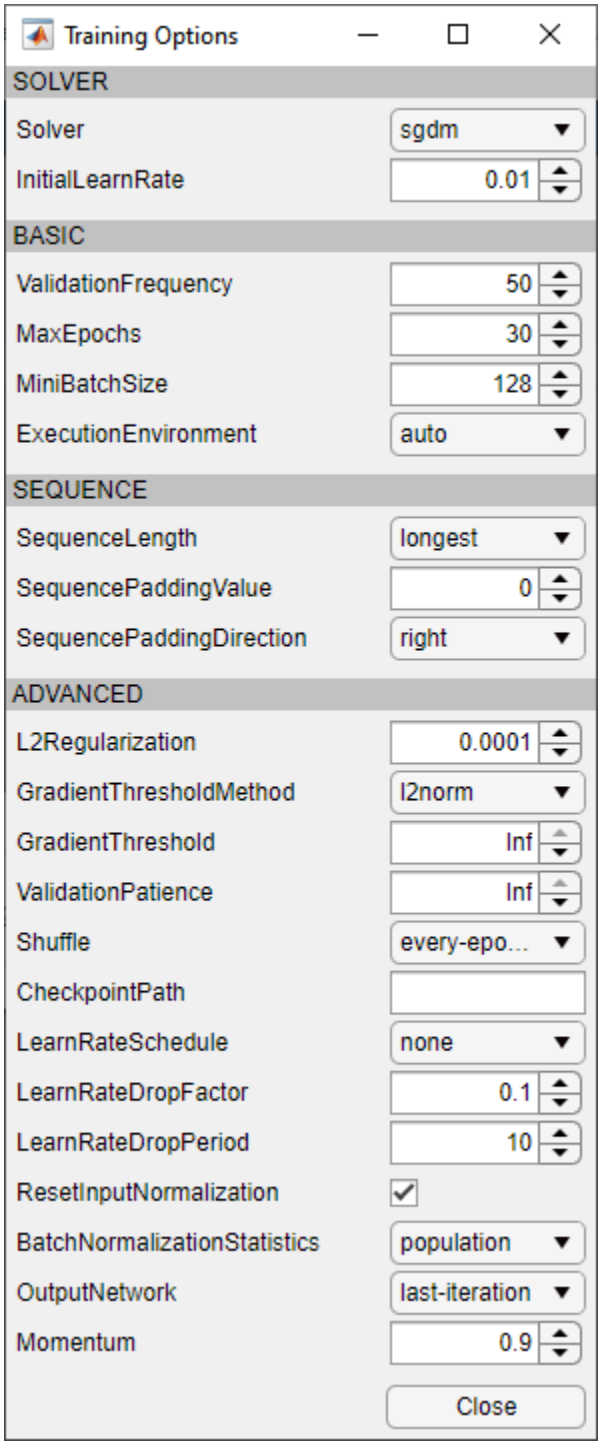
Using Deep Network Designer, you can train a network using image data or any datastore object that works with the `trainNetwork` function. For example, you can train a semantic segmentation network or a multi-input network using a `CombinedDatastore` object. For more information about importing data into Deep Network Designer, see “Import Data into Deep Network Designer”.

To train a network on data imported into Deep Network Designer, on the **Training** tab, click **Train**. The app displays an animated plot of the training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.



For more information, see “Train Networks Using Deep Network Designer”.

If you require greater control over the training, click **Training Options** to select the training settings. For more information about selecting training options, see `trainingOptions`.



For an example showing how to train an image classification network, see “Transfer Learning with Deep Network Designer”. For an example showing how to train a sequence-to-sequence LSTM network, see “Train Network for Time Series Forecasting Using Deep Network Designer”.

To train a network on data not supported by Deep Network Designer, select the **Designer** tab, and click **Export** to export the initial network architecture. You can then programmatically train the network, for example, using a custom training loop.

Export Network and Generate Code

Export the network architecture created in Deep Network Designer to the workspace or Simulink and generate code to recreate the network and training.

- To export the network architecture with the initial weights to the workspace, on the **Designer** tab, click **Export**. Depending on the network architecture, Deep Network Designer exports the network as a `LayerGraph` *lgraph* or as a `Layer` object *layers*.
- To export the network trained in Deep Network Designer to the workspace, on the **Training** tab, click **Export**. Deep Network Designer exports the trained network architecture as a `DAGNetwork` object *trainedNetwork*. Deep Network Designer also exports the results from training, such as training and validation accuracy, as the structure array *trainInfoStruct*.
- To export the network trained in Deep Network Designer to Simulink, on the **Training** tab, click **Export > Export to Simulink**. Deep Network Designer saves the trained network as a MAT-file and generates Simulink blocks representing the trained network. The blocks generated depend on the type of network trained.
  - Image Classifier — Classify data using a trained deep learning neural network.
  - Predict — Predict responses using a trained deep learning neural network.
  - Stateful Classify — Classify data using a trained recurrent neural network.
  - Stateful Predict — Predict responses using a trained recurrent neural network.

To recreate a network that you construct and train in Deep Network Designer, generate MATLAB code.

- To recreate the network layers, on the **Designer** tab, select **Export > Generate Code**.
- To recreate the network layers, including any learnable parameters, on the **Designer** tab, select **Export > Generate Code with Initial Parameters**.
- To recreate the network, data import, and training, on the **Training** tab, select **Export > Generate Code for Training**.

After generating a script, you can perform the following tasks.

- To recreate the network layers created in the app, run the script. If you generated the training script, running the script will also replicate the network training.
- Examine the code to learn how to create and connect layers programmatically, and how to train a deep network.
- To modify the layers, edit the code. You can also run the script and import the network back into the app for editing.

For more information, see “Generate MATLAB Code from Deep Network Designer”.

You can also use the generated script as a starting point to create deep learning experiments which sweep through a range of hyperparameter values or use Bayesian optimization to find optimal training options. For an example showing how to use **Experiment Manager** to tune the

hyperparameters of a network trained in Deep Network Designer, see “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager”.

- “Transfer Learning with Deep Network Designer”
- “Build Networks with Deep Network Designer”
- “Import Data into Deep Network Designer”
- “Train Networks Using Deep Network Designer”
- “Train Network for Time Series Forecasting Using Deep Network Designer”
- “Train Simple Semantic Segmentation Network in Deep Network Designer”
- “Image-to-Image Regression in Deep Network Designer”
- “Import Custom Layer into Deep Network Designer”
- “Generate MATLAB Code from Deep Network Designer”
- “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager”
- “List of Deep Learning Layers”

Programmatic Use

`deepNetworkDesigner` opens the Deep Network Designer app. If Deep Network Designer is already open, `deepNetworkDesigner` brings focus to the app.

`deepNetworkDesigner(net)` opens the Deep Network Designer app and loads the specified network into the app. The network can be a series network, DAG network, layer graph, or an array of layers.

For example, open Deep Network Designer with a pretrained SqueezeNet network.

```
net = squeezenet;
deepNetworkDesigner(net);
```

If Deep Network Designer is already open, `deepNetworkDesigner(net)` brings focus to the app and prompts you to add to or replace any existing network.

Tips

To train multiple networks and compare the results, try **Experiment Manager**.

See Also

Functions

`analyzeNetwork` | `trainNetwork` | `trainingOptions` | **Experiment Manager**

Topics

“Transfer Learning with Deep Network Designer”
“Build Networks with Deep Network Designer”
“Import Data into Deep Network Designer”
“Train Networks Using Deep Network Designer”
“Train Network for Time Series Forecasting Using Deep Network Designer”
“Train Simple Semantic Segmentation Network in Deep Network Designer”
“Image-to-Image Regression in Deep Network Designer”

“Import Custom Layer into Deep Network Designer”
“Generate MATLAB Code from Deep Network Designer”
“Adapt Code Generated in Deep Network Designer for Use in Experiment Manager”
“List of Deep Learning Layers”

Introduced in R2018b

Deep Network Quantizer

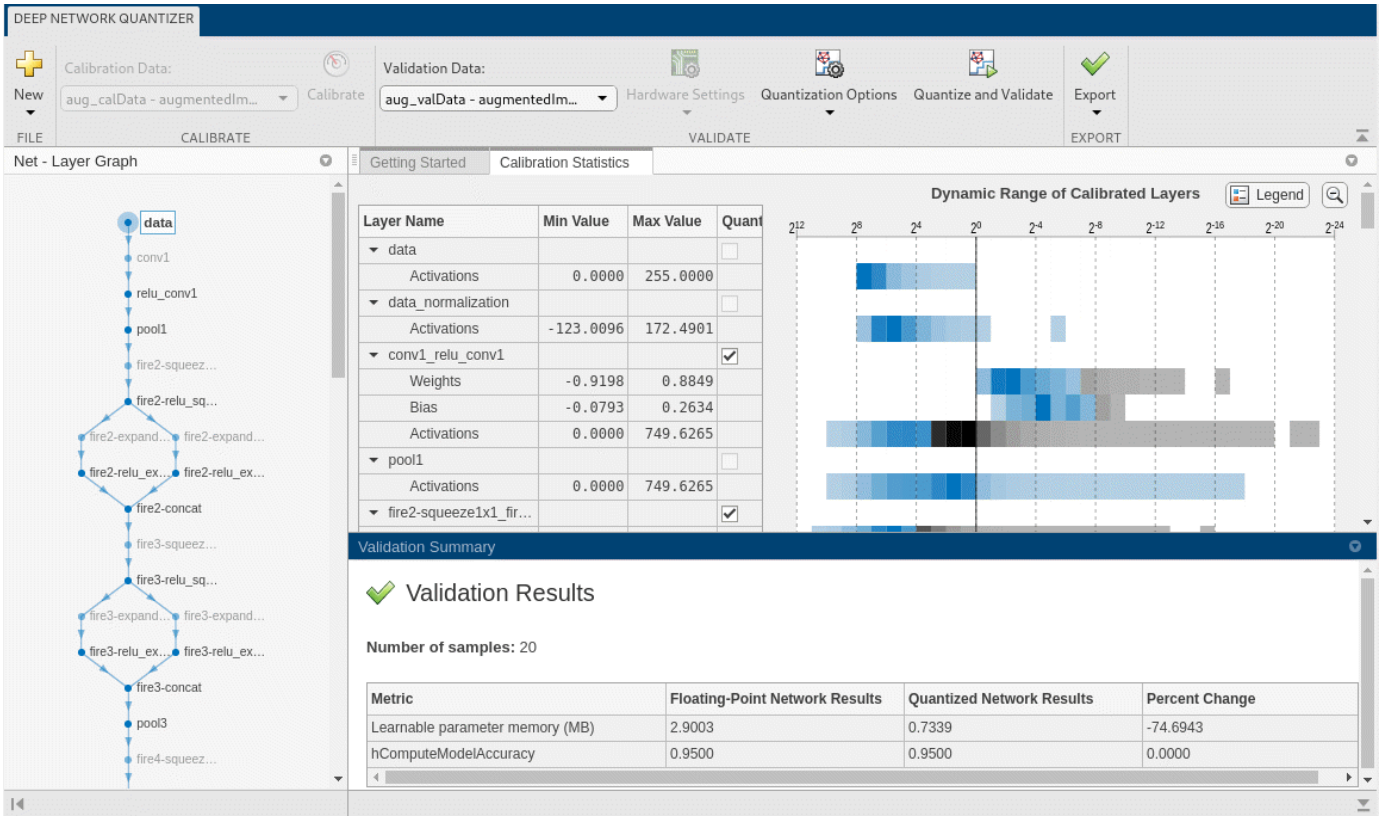
Quantize a deep neural network to 8-bit scaled integer data types

Description

Use the **Deep Network Quantizer** app to reduce the memory requirement of a deep neural network by quantizing weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Using this app you can:

- Visualize the dynamic ranges of convolution layers in a deep neural network.
- Select individual network layers to quantize.
- Assess the performance of a quantized network.
- Generate GPU code to deploy the quantized network using GPU Coder™.
- Generate HDL code to deploy the quantized network to an FPGA using Deep Learning HDL Toolbox™.
- Generate C++ code to deploy the quantized network to an ARM Cortex-A microcontroller using MATLAB Coder™.

The Deep Learning Toolbox Model Quantization Library support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library. To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.



Open the Deep Network Quantizer App

- MATLAB command prompt: Enter `deepNetworkQuantizer`.
- MATLAB toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.

Examples

Quantize a Network for GPU Deployment

To explore the behavior of a neural network with quantized convolution layers, use the **Deep Network Quantizer** app. This example quantizes the learnable parameters of the convolution layers of the `squeezenet` neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example.

This example uses a DAG network with the GPU execution environment.

Load the network to quantize into the base workspace.

```
load net
net
```

```
net =
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data.

The app uses calibration data to exercise the network and collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The app uses the validation data to test the network after quantization to understand the effects of the limited range and precision of the quantized learnable parameters of the convolution layers in the network.

In this example, use the images in the `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

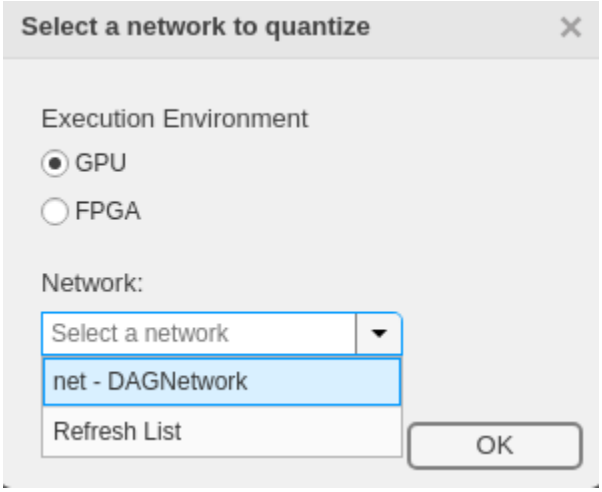
At the MATLAB command prompt, open the app.

```
deepNetworkQuantizer
```

In the app, click **New** and select `Quantize a network`.

The app verifies your execution environment. For more information, see Quantization Workflow Prerequisites.

In the dialog, select the execution environment and the network to quantize from the base workspace. For this example, select a GPU execution environment and the DAG network, `net`.



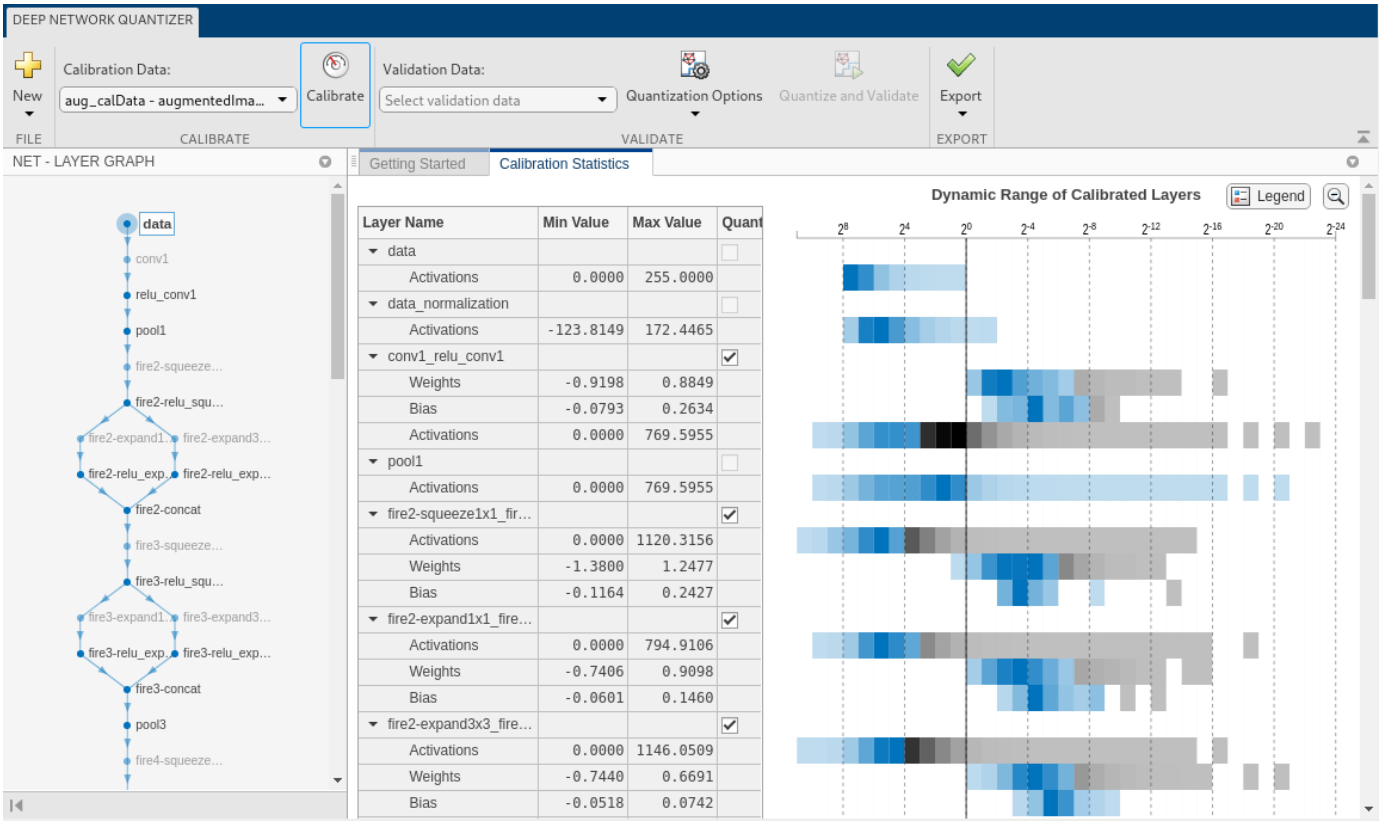
The app displays the layer graph of the selected network.

In the **Calibrate** section of the toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data, `calData`.

Click **Calibrate**.

The **Deep Network Quantizer** uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

When the calibration is complete, the app displays a table containing the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network and their minimum and maximum values during the calibration. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see Quantization of Deep Neural Networks.



In the **Quantize** column of the table, indicate whether to quantize the learnable parameters in the layer. Layers that are not convolution layers cannot be quantized, and therefore cannot be selected. Layers that are not quantized remain in single-precision after quantization.

In the **Validate** section of the toolstrip, under **Validation Data**, select the `augmentedImageDatastore` object from the base workspace containing the validation data, `aug_valData`.

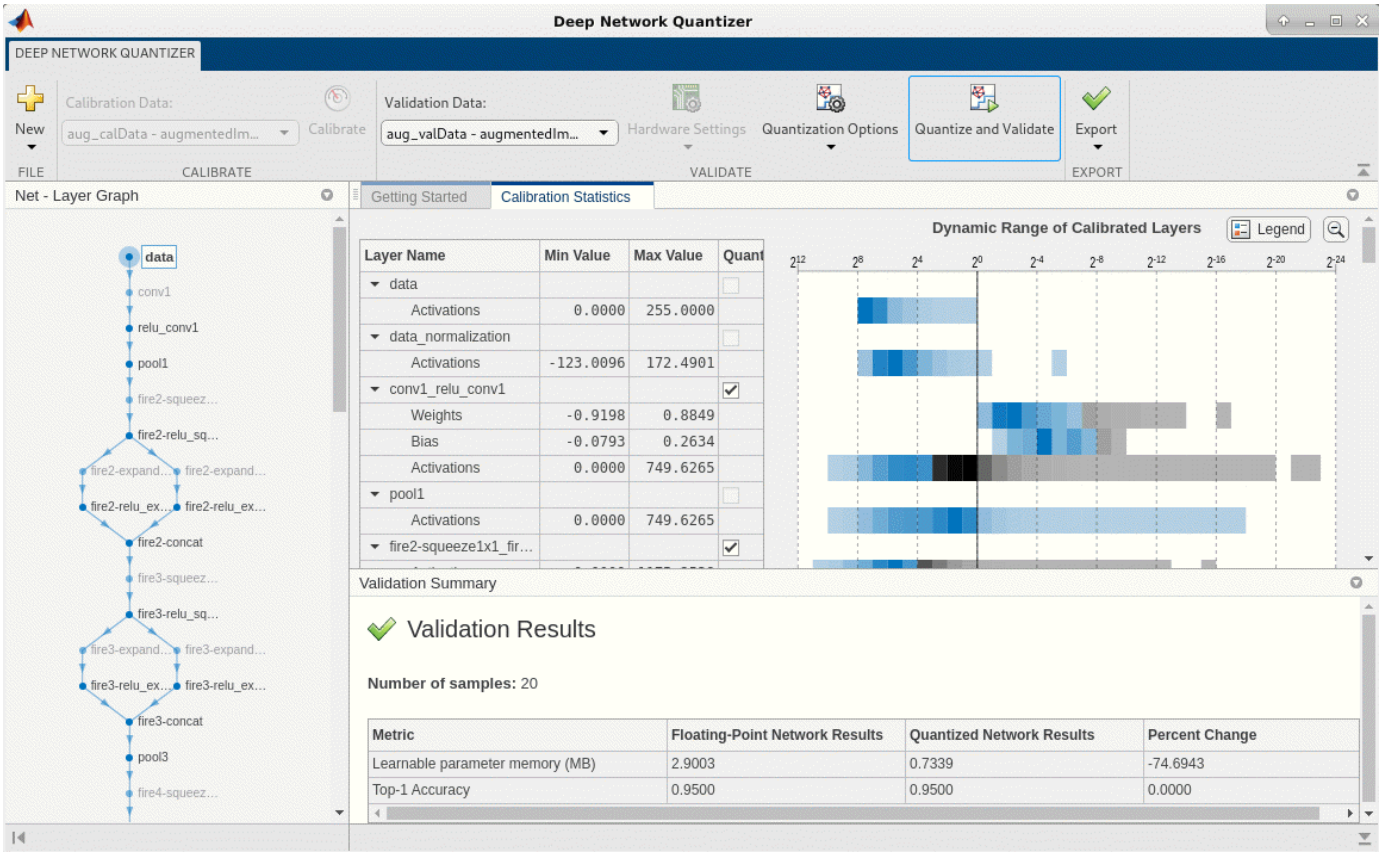
In the **Validate** section of the toolstrip, under **Quantization Options**, select the **Default** metric function.

Click **Quantize and Validate**.

The **Deep Network Quantizer** quantizes the weights, activations, and biases of convolution layers in the network to scaled 8-bit integer data types and uses the validation data to exercise the network. The app determines a default metric function to use for the validation based on the type of network that is being quantized. For a classification network, the app uses Top-1 Accuracy.

When the validation is complete, the app displays the results of the validation, including:

- Metric function used for validation
- Result of the metric function before and after quantization
- Memory requirement of the network before and after quantization (MB)



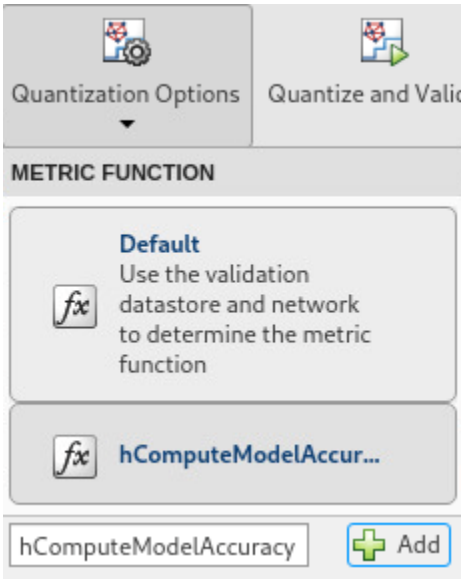
If you want to use a different metric function for validation, for example to use the Top-5 accuracy metric function instead of the default Top-1 accuracy metric function, you can define a custom metric function. Save this function in a local file.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare with predicted label with actual ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

To revalidate the network using this custom metric function, under **Quantization Options**, enter the name of the custom metric function, `hComputeModelAccuracy`. Select **Add** to add

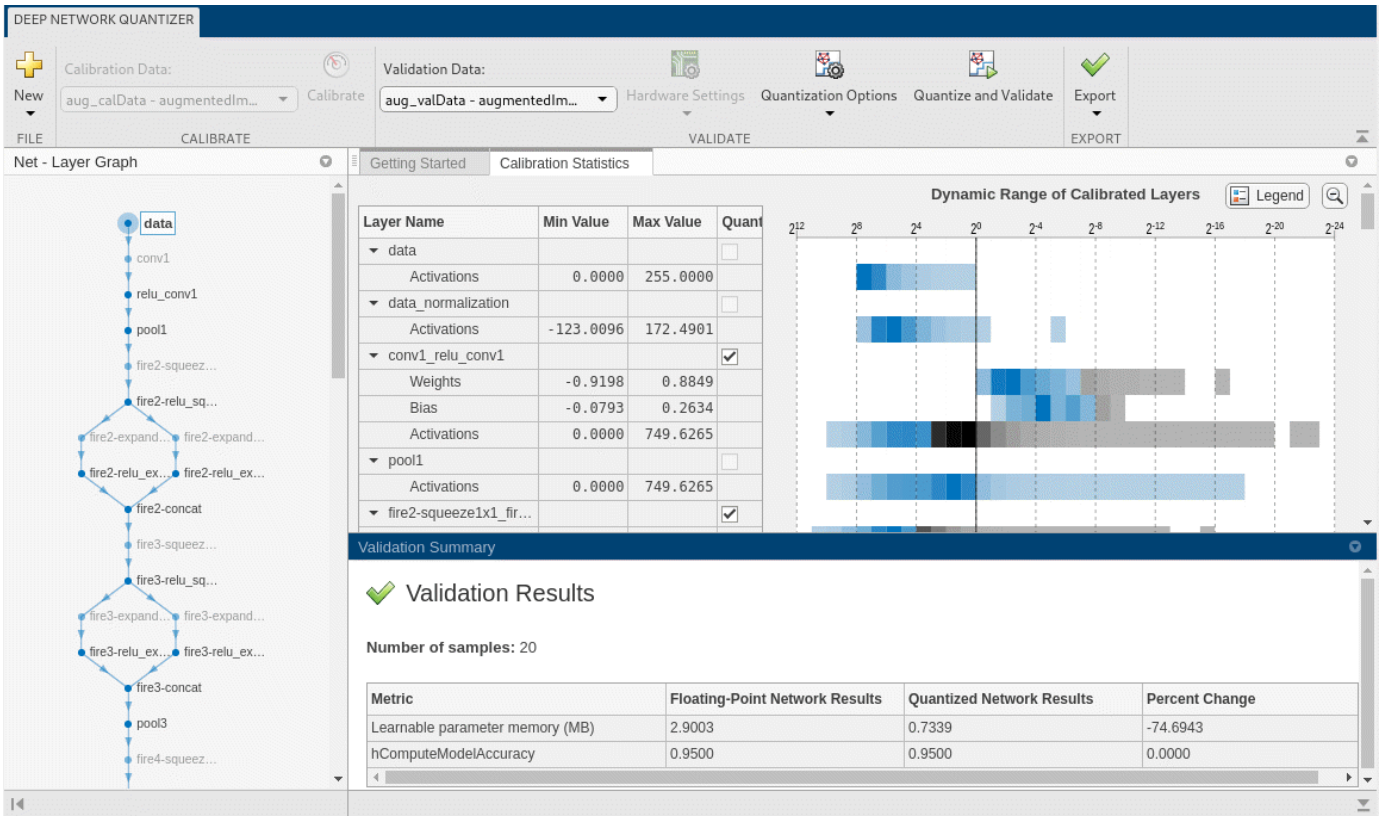
`hComputeModelAccuracy` to the list of metric functions available in the app. Select `hComputeModelAccuracy` as the metric function to use.

The custom metric function must be on the path. If the metric function is not on the path, this step will produce an error.



Click **Quantize and Validate**.

The app quantizes the network and displays the validation results for the custom metric function.



The app displays only scalar values in the validation results table. To view the validation results for a custom metric function with non-scalar output, export the `dlquantizer` object as described below, then validate using the `validate` function at the MATLAB command window.

After quantizing and validating the network, you can choose to export the quantized network.

Click the **Export** button. In the drop down, select `Export Quantizer` to create a `dlquantizer` object in the base workspace. To open the **GPU Coder** app and generate GPU code from the quantized neural network, select `Generate Code`. Generating GPU code requires a GPU Coder license.

If the performance of the quantized network is not satisfactory, you can choose to not quantize some layers by deselecting the layer in the table. To see the effects, click **Quantize and Validate** again.

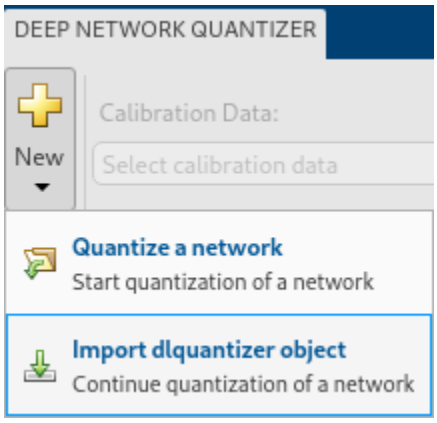
Import a `dlquantizer` Object into the Deep Network Quantizer App

This example shows you how to import a `dlquantizer` object from the base workspace into the **Deep Network Quantizer** app. This allows you to begin quantization of a deep neural network using the command line or the app, and resume your work later in the app.

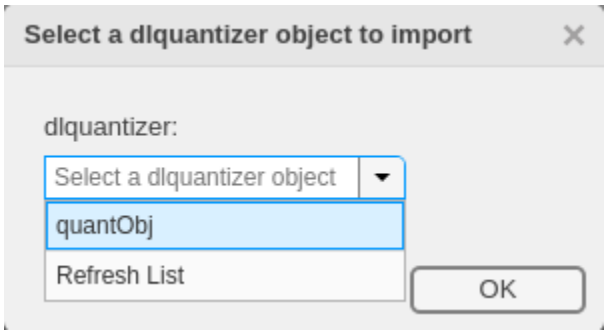
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

In the app, click **New** and select `Import dlquantizer object`.

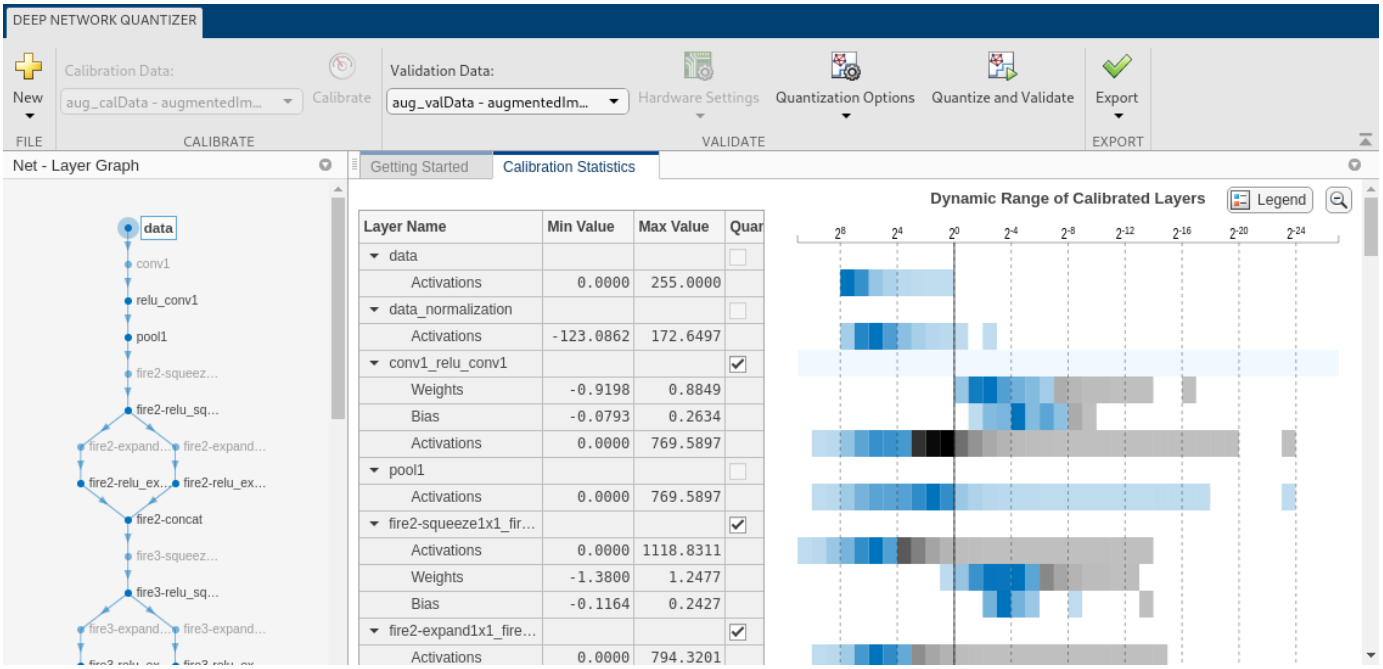


In the dialog, select the `dlquantizer` object to import from the base workspace. For this example, use `quantObj` that you create in the above example Quantize a Neural Network for GPU Target.



The app imports any data contained in the `dlquantizer` object that was collected at the command line. This data can include the network to quantize, calibration data, validation data, and calibration statistics.

The app displays a table containing the calibration data contained in the imported `dlquantizer` object, `quantObj`. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks”.



Quantize a Network for FPGA Deployment

To explore the behavior of a neural network that has quantized convolution layers, use the **Deep Network Quantizer** app. This example quantizes the learnable parameters of the convolution layers of the `LogoNet` neural network.

For this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working folder called `getLogoNetwork.m`. In the file, enter:

```
function net = getLogoNetwork
 if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
    net  = data.convnet;
end
```

Load the pretrained network.

```
snet = getLogoNetwork;
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
    InputNames: {'imageinput'}
   OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The app uses calibration data to exercise the network and collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network. The app also exercises the dynamic ranges of the activations in all layers of the LogoNet network. For the best quantization results, the calibration data must be representative of inputs to the LogoNet network.

After quantization, the app uses the validation data set to test the network to understand the effects of the limited range and precision of the quantized learnable parameters of the convolution layers in the network.

In this example, use the images in the `logos_dataset` data set to calibrate and validate the LogoNet network. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

Expedite the calibration and validation process by using a subset of the `calibrationData` and `validationData`. Store the new reduced calibration data set in `calibrationData_concise` and the new reduced validation data set in `validationData_concise`.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_concise = calibrationData.subset(1:20);
validationData_concise = vaidationData.subset(1:1);
```

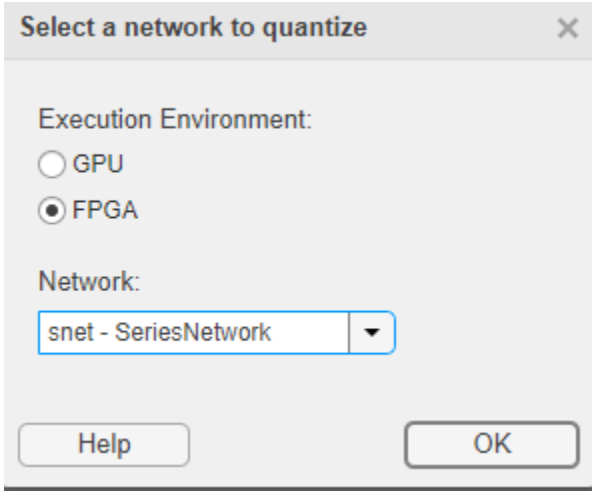
At the MATLAB command prompt, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Click **New** and select `Quantize a network`.

The app verifies your execution environment.

Select the execution environment and the network to quantize from the base workspace. For this example, select a FPGA execution environment and the series network `snet`.



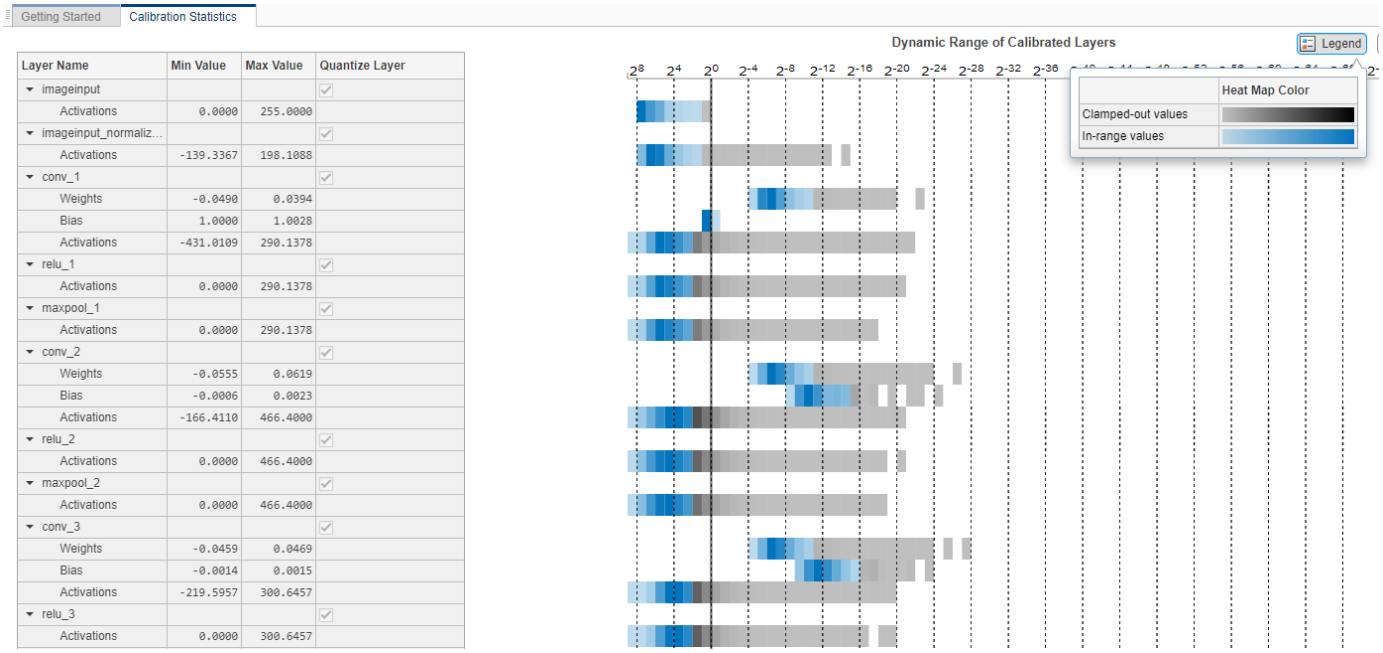
The app displays the layer graph of the selected network.

In the **Calibrate** section of the app toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data `calibrationData_concise`.

Click **Calibrate**.

The **Deep Network Quantizer** app uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

When the calibration is complete, the app displays a table containing the weights and biases in the convolution and fully connected layers of the network. Also displayed are the dynamic ranges of the activations in all layers of the network and their minimum and maximum values during the calibration. The app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks”.



In the **Quantize** column of the table, indicate whether to quantize the learnable parameters in the layer. You cannot quantize layers that are not convolution layers. Layers that are not quantized remain in single-precision.

In the **Validate** section of the app toolstrip, under **Validation Data**, select the `augmentedImageDatastore` object from the base workspace containing the validation data `validationData_concise`.

In the **Hardware Settings** section of the toolstrip, select from the options listed in the table:

Simulation Environment	**Action**
MATLAB (Simulate in MATLAB)	Simulates the quantized network in MATLAB. Validates the quantized network by comparing performance to single-precision version of the network.

Intel Arria 10 SoC (arria10soc_int8)	Deploys the quantized network to an Intel® Arria® 10 SoC board by using the `arria10soc_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.
Xilinx ZCU102 (zcu102_int8)	Deploys the quantized network to a Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 10 SoC board by using the `zcu102_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.
Xilinx ZC706 (zc706_int8)	Deploys the quantized network to a Xilinx Zynq-7000 ZC706 board by using the `zc706_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.

When you select the `Intel Arria 10 SoC (arria10soc_int8)`, `Xilinx ZCU102 (zcu102_int8)`, or `Xilinx ZC706 (zc706_int8)` options, select the interface to use to deploy and validate the quantized network. The **Target** interface options are listed in this table.

Target Option	**Action**
JTAG	Programs the target FPGA board selected in **Simulation Environment** by using a JTAG cable. For more information, see “JTAG Connection” (Deep Learning HDL Toolbox)
Ethernet	Programs the target FPGA board selected in **Simulation Environment** through the Ethernet interface. Specify the IP address for your target board in **IP Address**.

For this example, select `Xilinx ZCU102 (zcu102_int8)`, select **Ethernet**, and enter the board IP address.

In the **Validate** section of the app toolstrip, under **Quantization Options**, select the **Default** metric function.

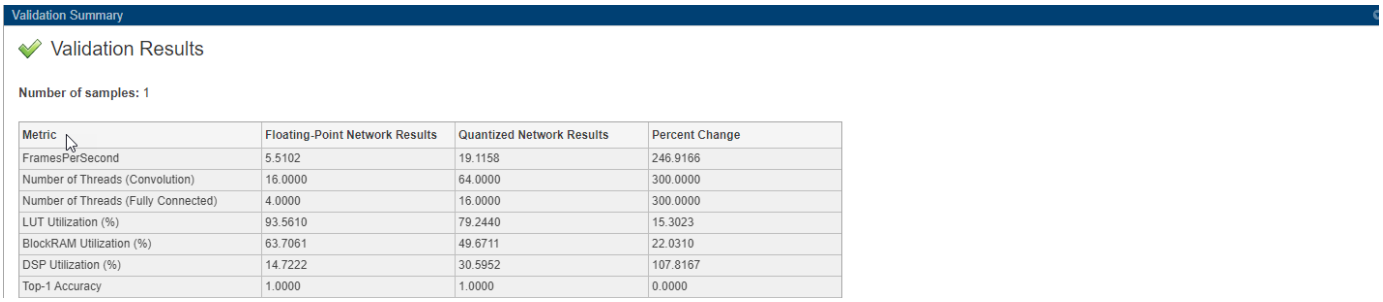
Click **Quantize and Validate**.

The **Deep Network Quantizer** app quantizes the weights, activations, and biases of convolution layers in the network to scaled 8-bit integer data types and uses the validation data to exercise the network. The app determines a metric function to use for the validation based on the type of network that is being quantized.

Type of Network	Metric Function
Classification	**Top-1 Accuracy** – Accuracy of the network
Object Detection	**Average Precision** – Average precision over all detection results. See `evaluateDetectionPrecision`.
Regression	**MSE** – Mean squared error of the network
Semantic Segmentation	`evaluateSemanticSegmentation` – Evaluate semantic segmentation data set against ground truth
Single Shot Detector (SSD)	**WeightedIOU** – Average IoU of each class, weighted by the number of pixels in that class

When the validation is complete, the app displays the results of the validation, including:

- Metric function used for validation
- Result of the metric function before and after quantization



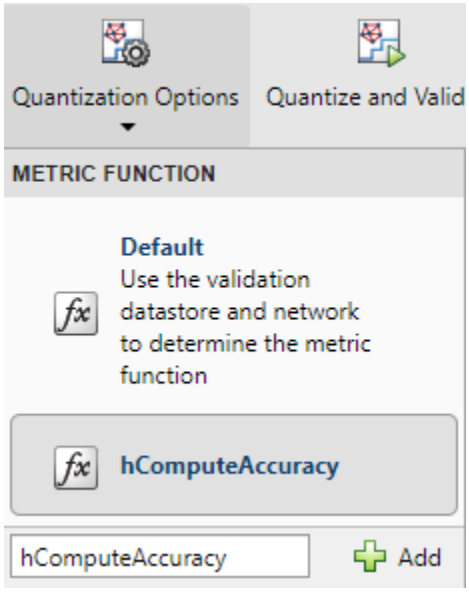
Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
FramesPerSecond	5.5102	19.1158	246.9166
Number of Threads (Convolution)	16.0000	64.0000	300.0000
Number of Threads (Fully Connected)	4.0000	16.0000	300.0000
LUT Utilization (%)	93.5610	79.2440	15.3023
BlockRAM Utilization (%)	63.7061	49.6711	22.0310
DSP Utilization (%)	14.7222	30.5952	107.8167
Top-1 Accuracy	1.0000	1.0000	0.0000

If you want to use a different metric function for validation, for example to use the Top-5 accuracy metric function instead of the default Top-1 accuracy metric function, you can define a custom metric function. Save this function in a local file.

```
function accuracy = hComputeAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

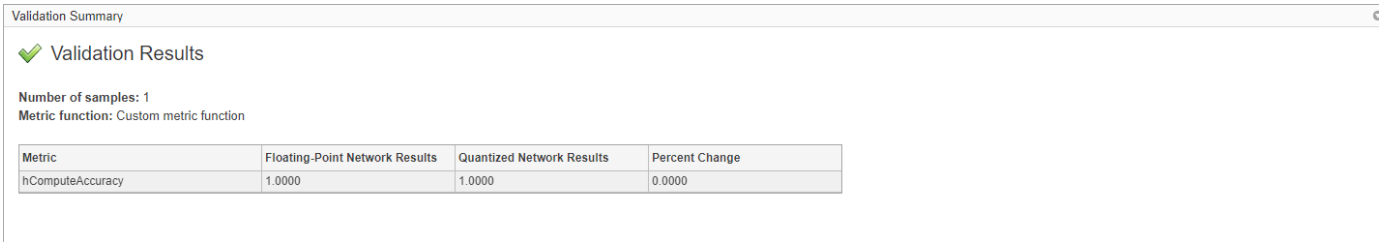
To revalidate the network by using this custom metric function, under **Quantization Options**, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function to use.

The custom metric function must be on the path. If the metric function is not on the path, this step produces an error.



Click **Quantize and Validate**.

The app quantizes the network and displays the validation results for the custom metric function.



The app displays only scalar values in the validation results table. To view the validation results for a custom metric function with nonscalar output, export the `dlquantizer` object, then validate the quantized network by using the `validate` function in the MATLAB command window.

After quantizing and validating the network, you can choose to export the quantized network.

Click the **Export** button. In the drop-down list, select `Export Quantizer` to create a `dlquantizer` object in the base workspace. You can deploy the quantized network to your target FPGA board and retrieve the prediction results by using MATLAB. See, “Deploy Quantized Network Example” (Deep Learning HDL Toolbox).

- “Quantization of Deep Neural Networks”

Parameters

`Quantization Options` — Metric function to use for validation

Default (default) | custom metric function

By default, the **Deep Network Quantizer** app determines a metric function to use for the validation based on the type of network that is being quantized.

Type of Network	Metric Function
Classification	**Top-1 Accuracy** – Accuracy of the network
Object Detection	**Average Precision** – Average precision over all detection results. See `evaluateDetectionPrecision`.
Regression	**MSE** – Mean squared error of the network
Semantic Segmentation	**WeightedIOU** – Average IoU of each class, weighted by the number of pixels in that class. See `evaluateSemanticSegmentation`.

You can also specify a custom metric function to use.

See Also

Functions

`calibrate` | `validate` | `dlquantizer` | `dlquantizationOptions`

Topics

“Quantization of Deep Neural Networks”

Introduced in R2020a

Experiment Manager

Design and run experiments to train and compare deep learning networks

Description

The **Experiment Manager** app enables you to create deep learning experiments to train networks under multiple initial conditions and compare the results. For example, you can use deep learning experiments to:

- Sweep through a range of hyperparameter values or use Bayesian optimization to find optimal training options. Bayesian optimization requires Statistics and Machine Learning Toolbox™.
- Use the built-in function `trainNetwork` or define your own custom training function.
- Compare the results of using different data sets or test different deep network architectures.

To set up your experiment quickly, you can start with a preconfigured template. The experiment templates support workflows that include image classification, image regression, sequence classification, semantic segmentation, and custom training loops.

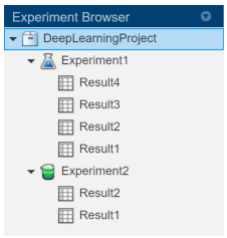
Experiment Manager provides visualization tools such as training plots and confusion matrices, filters to refine your experiment results, and annotations to record your observations. To improve reproducibility, every time that you run an experiment, Experiment Manager stores a copy of the experiment definition. You can access past experiment definitions to keep track of the hyperparameter combinations that produce each of your results.

Experiment Manager organizes your experiments and results in a project.

- You can store several experiments in the same project.
- Each experiment contains a set of results for each time that you run the experiment.
- Each set of results consists of one or more trials corresponding to a different combination of hyperparameters.

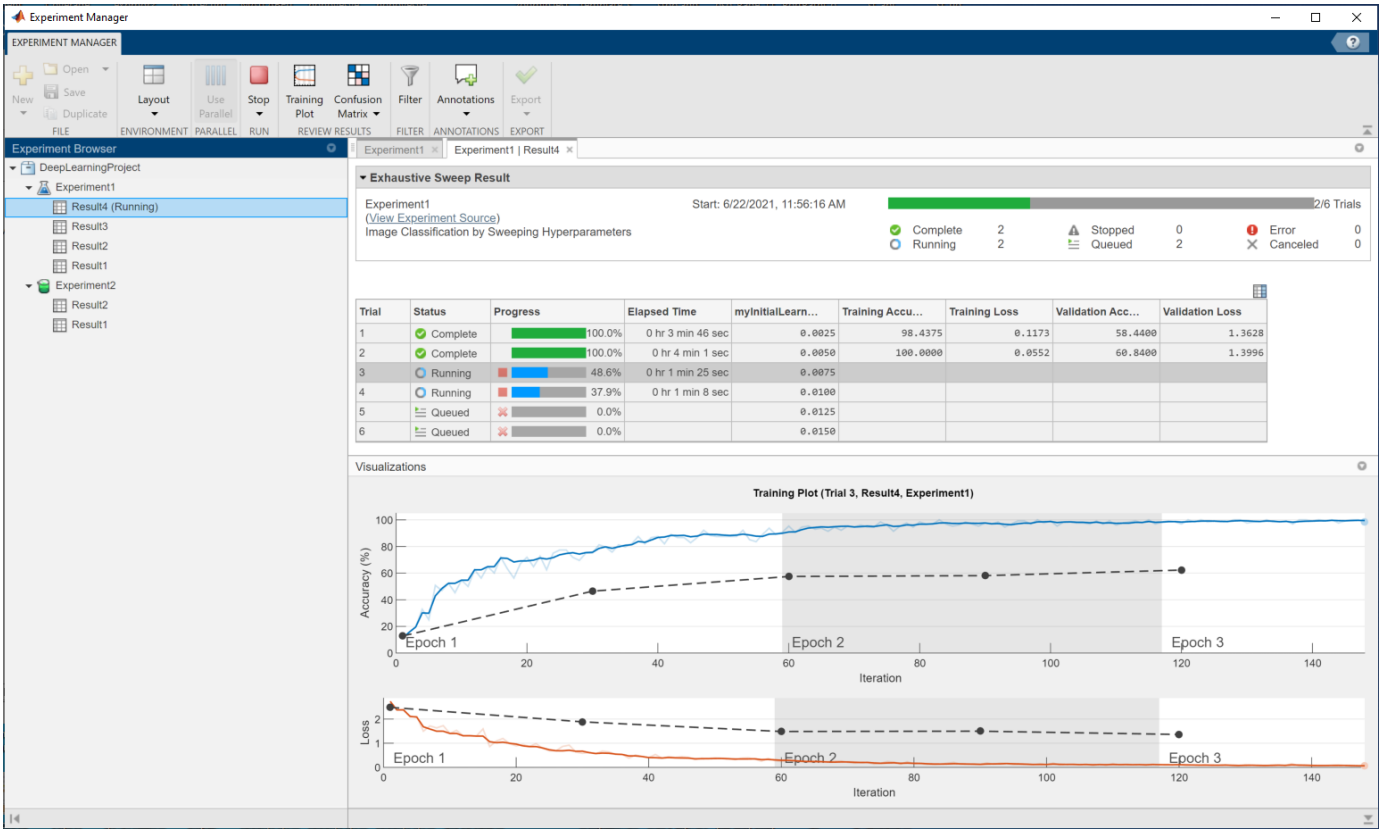
By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can configure your experiment to run multiple trials at the same time or to run a single trial at a time on multiple GPUs, on a cluster, or in the cloud. For more information, see “Use Experiment Manager to Train Networks in Parallel”.

The **Experiment Browser** pane displays the hierarchy of experiments and results in the project. For instance, this project has two experiments, each of which has several sets of results.



The blue flask indicates a built-in training experiment that uses the `trainNetwork` function. The green beaker indicates a custom training experiment that relies on a different training function.

To open the configuration for an experiment and view its results, double-click the name of the experiment or a set of results.



Open the Experiment Manager App

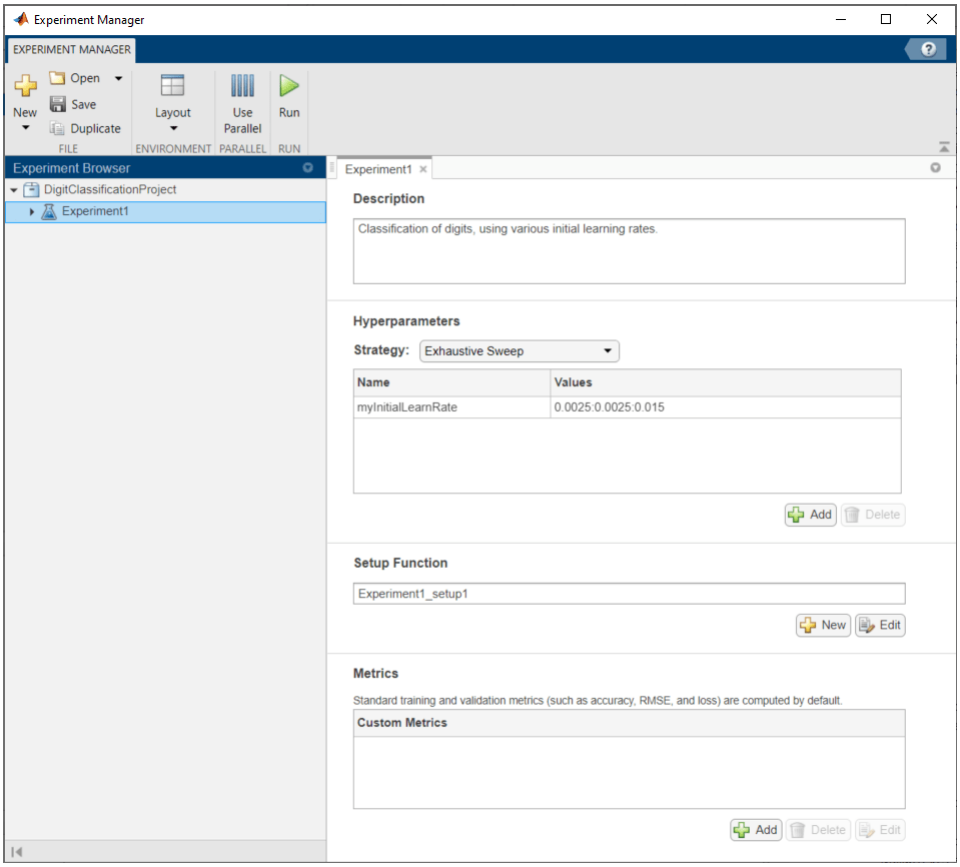
- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `experimentManager`.

Examples

Image Classification by Sweeping Hyperparameters

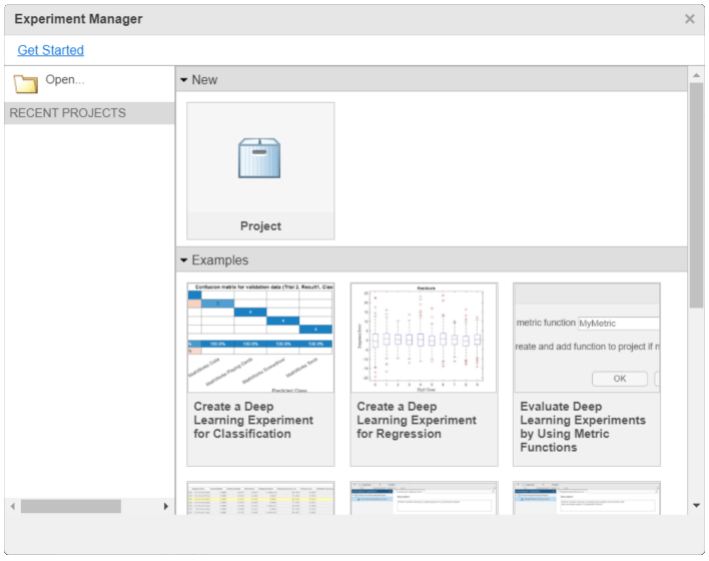
This example shows how to use the experiment template for image classification by sweeping hyperparameters. With this template, you can quickly set up a built-in training experiment that uses the `trainNetwork` function. For more examples of solving image classification problems with Experiment Manager, see “Create a Deep Learning Experiment for Classification” and “Use Experiment Manager to Train Networks in Parallel”. For more information on an alternative strategy to sweeping hyperparameters, see “Tune Experiment Hyperparameters by Using Bayesian Optimization”.

Open the example to load a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`Experiment1`).

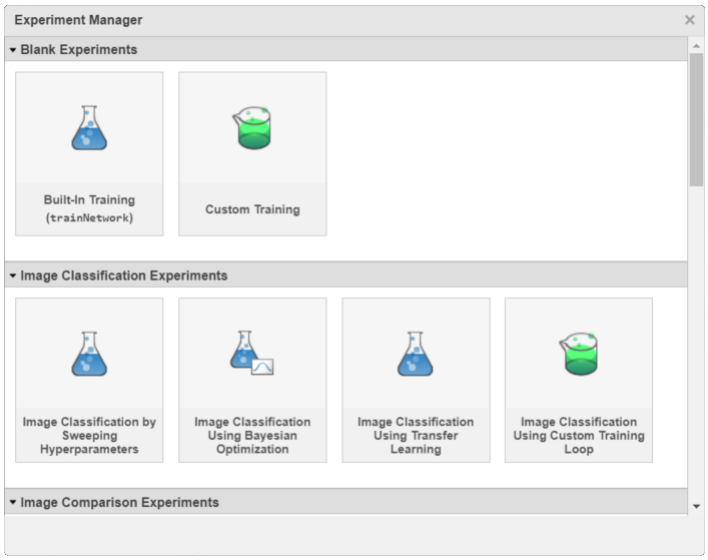


Alternatively, you can configure the experiment yourself by following these steps.

1. Open Experiment Manager. A dialog box provides links to the getting started tutorials and your recent projects, as well as buttons to create a new project or open an example from the documentation.



2. Under **New**, select **Project**. A dialog box lists several templates that support workflows including image classification, image regression, sequence classification, semantic segmentation, and custom training loops.



3. Under **Image Classification Experiments**, select **Image Classification by Sweeping Hyperparameters**.

4. Specify the name and location for the new project. Experiment Manager opens a new experiment in the project. The **Experiment** pane displays the description, hyperparameters, setup function, and metrics that define the experiment.

5. In the **Description** field, enter a description of the experiment:

```
Classification of digits, using various initial learning rates.
```

6. Under **Hyperparameters**, replace the value of `myInitialLearnRate` with `0.0025:0.0025:0.015`. Verify that **Strategy** is set to `Exhaustive Sweep`.

7. Under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. The setup function specifies the training data, network architecture, and training options for the experiment. In this experiment, the setup function has three sections.

- **Load Training Data** defines image datastores containing the training and validation data for the experiment. The experiment uses the Digits data set, which consists of 10,000 28-by-28 pixel grayscale images of digits from 0 to 9, categorized by the digit they represent. For more information on this data set, see “Image Data Sets”.
- **Define Network Architecture** defines the architecture for a simple convolutional neural network for deep learning classification.
- **Specify Training Options** defines a `trainingOptions` object for the experiment. In this experiment, the setup function loads the values for the initial learning rate from the `myInitialLearnRate` entry in the hyperparameter table.

When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses one of the learning rates specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run

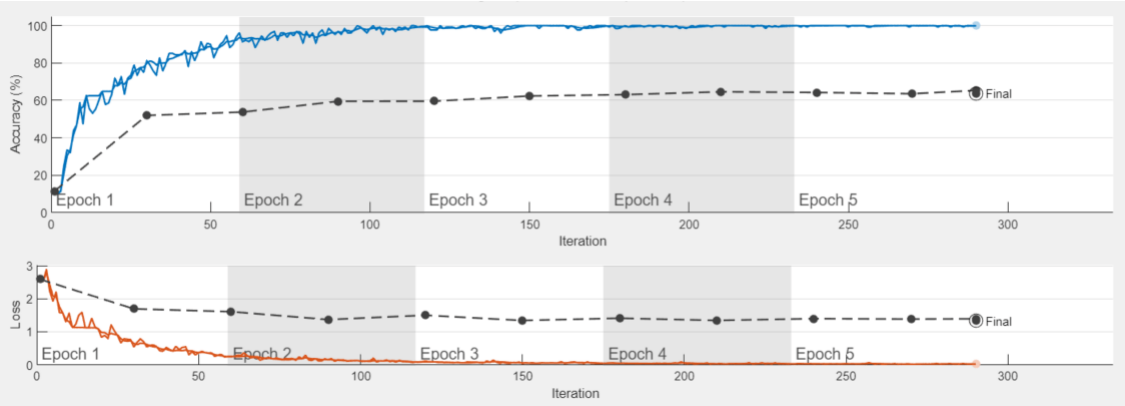
multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

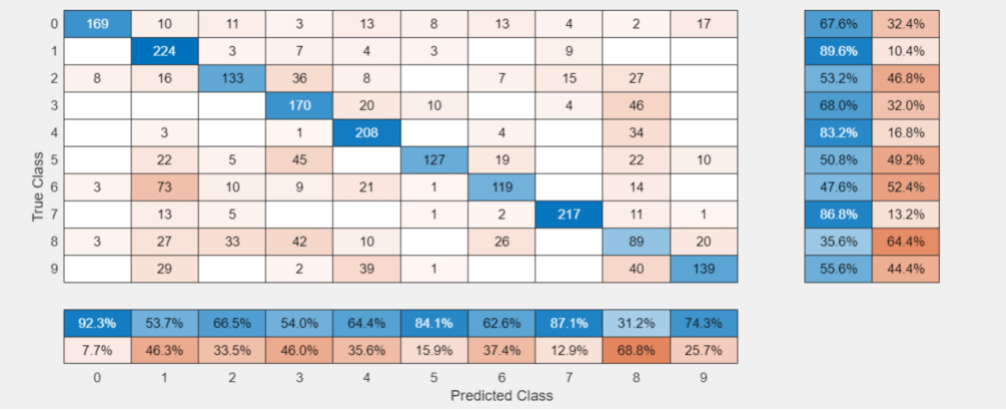
A table of results displays the accuracy and loss for each trial.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete	100.0%	0 hr 3 min 33 sec	0.0025	100.0000	0.0699	58.6400	1.2963
2	Complete	100.0%	0 hr 3 min 30 sec	0.0050	100.0000	0.0318	62.4800	1.3210
3	Running	30.7%	0 hr 0 min 56 sec	0.0075				
4	Running	30.7%	0 hr 0 min 56 sec	0.0100				
5	Queued	0.0%		0.0125				
6	Queued	0.0%		0.0150				

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. You can also monitor the training progress in the MATLAB Command Window.



Click **Confusion Matrix** to display the confusion matrix for the validation data in each completed trial.



When the experiment finishes, you can sort the table by column or filter trials by using the **Filters** pane. You can also record observations by adding annotations to the results table. For more information, see “Sort, Filter, and Annotate Experiment Results” on page 1-47.

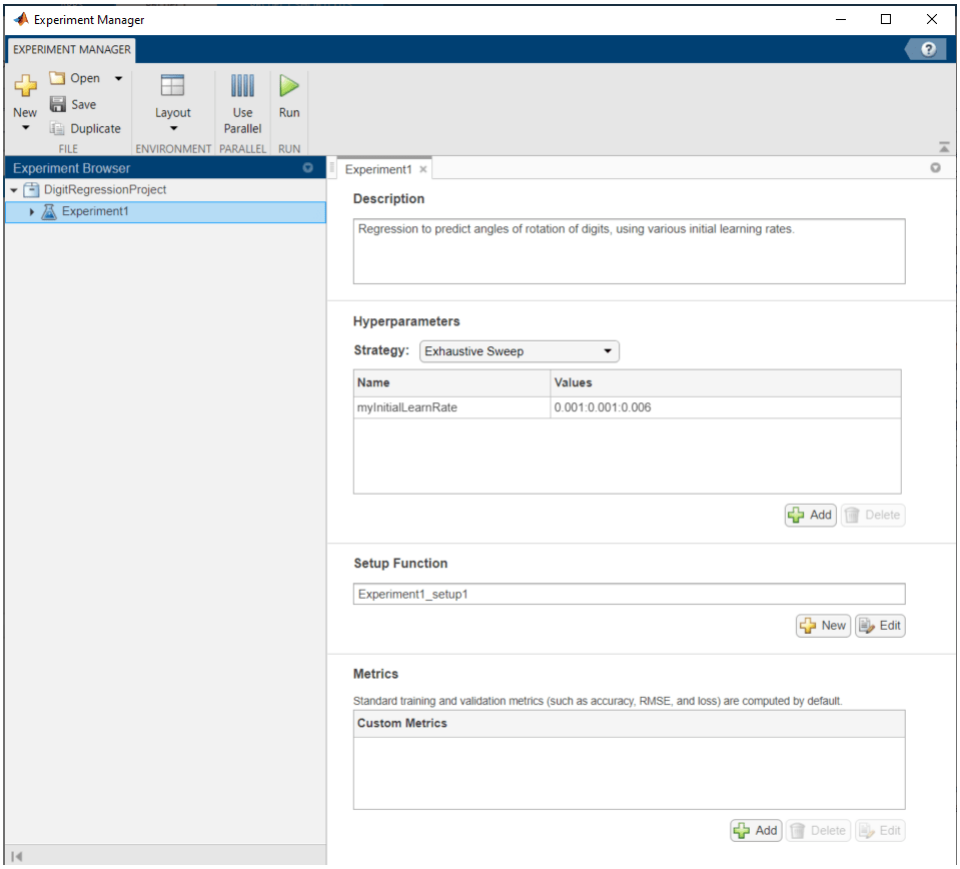
To test the performance of an individual trial, export the trained network or the training information for the trial. On the **Experiment Manager** toolstrip, select **Export** > **Trained Network** or **Export** > **Training Information**, respectively. For more information, see “net” on page 1-0 and “info” on page 1-0 .

To close the experiment, in the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Image Regression by Sweeping Hyperparameters

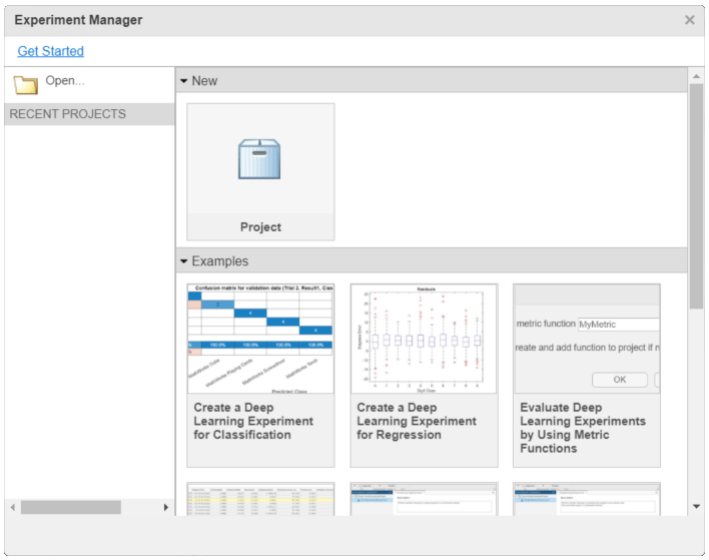
This example shows how to use the experiment template for image regression by sweeping hyperparameters. With this template, you can quickly set up a built-in training experiment that uses the `trainNetwork` function. For another example of solving a regression problem with Experiment Manager, see “Create a Deep Learning Experiment for Regression”. For more information on an alternative strategy to sweeping hyperparameters, see “Tune Experiment Hyperparameters by Using Bayesian Optimization”.

Open the example to load a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`Experiment1`).

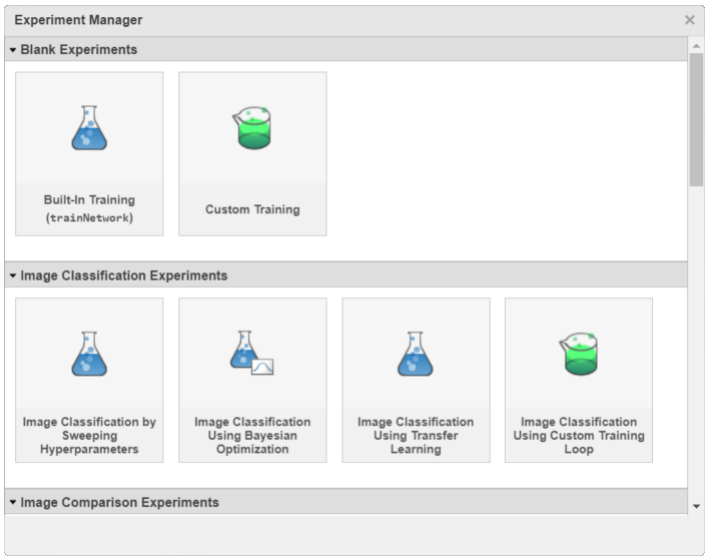


Alternatively, you can configure the experiment yourself by following these steps.

1. Open Experiment Manager. A dialog box provides links to the getting started tutorials and your recent projects, as well as buttons to create a new project or open an example from the documentation.



2. Under **New**, select **Project**. A dialog box lists several templates that support workflows including image classification, image regression, sequence classification, semantic segmentation, and custom training loops.



3. Under **Image Regression Experiments**, select **Image Regression by Sweeping Hyperparameters**.

4. Specify the name and location for the new project. Experiment Manager opens a new experiment in the project. The **Experiment** pane displays the description, hyperparameters, setup function, and metrics that define the experiment.

5. In the **Description** field, enter a description of the experiment:

```
Regression to predict angles of rotation of digits, using various initial learning rates.
```

6. Under **Hyperparameters**, replace the value of `myInitialLearnRate` with `0.001:0.001:0.006`. Verify that **Strategy** is set to `Exhaustive Sweep`.

7. Under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. The setup function specifies the training data, network architecture, and training options for the experiment. In this experiment, the setup function has three sections.

- **Load Training Data** defines the training and validation data for the experiment as 4-D arrays. The training and validation data each consist of 5000 images from the Digits data set. Each image shows a digit from 0 to 9, rotated by a certain angle. The regression values correspond to the angles of rotation. For more information on this data set, see “Image Data Sets”.
- **Define Network Architecture** defines the architecture for a simple convolutional neural network for deep learning regression.
- **Specify Training Options** defines a `trainingOptions` object for the experiment. In this experiment, the setup function loads the values for the initial learning rate from the `myInitialLearnRate` entry in the hyperparameter table.

When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses one of the learning rates specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run

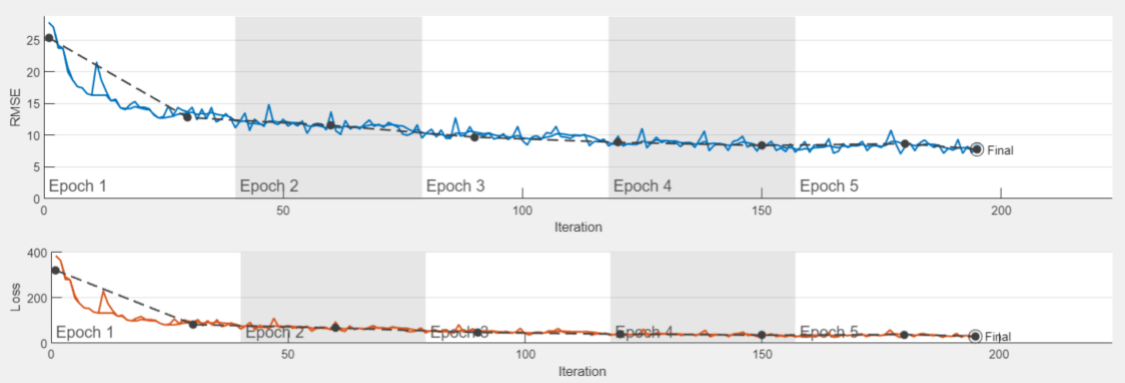
multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the root mean squared error (RMSE) and loss for each trial.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training RMSE	Training Loss	Validation RMSE	Validation Loss
1	Complete	100.0%	0 hr 0 min 47 sec	0.0010	8.1168	32.9413	7.7264	29.8488
2	Complete	100.0%	0 hr 0 min 49 sec	0.0020	8.3831	35.1378	8.0632	32.5073
3	Running	76.4%	0 hr 0 min 49 sec	0.0030				
4	Running	74.4%	0 hr 0 min 48 sec	0.0040				
5	Queued	0.0%		0.0050				
6	Queued	0.0%		0.0060				

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. You can also monitor the training progress in the MATLAB Command Window.



When the experiment finishes, you can sort the table by column or filter trials by using the **Filters** pane. You can also record observations by adding annotations to the results table. For more information, see “Sort, Filter, and Annotate Experiment Results” on page 1-47.

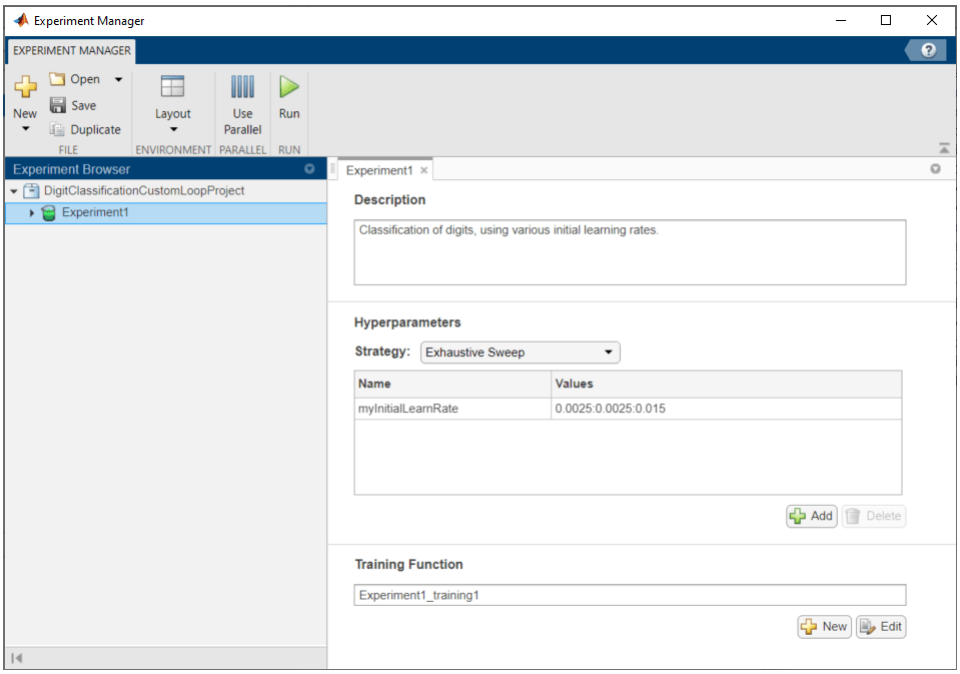
To test the performance of an individual trial, export the trained network or the training information for the trial. On the **Experiment Manager** toolstrip, select **Export** > **Trained Network** or **Export** > **Training Information**, respectively. For more information, see “net” on page 1-0 and “info” on page 1-0.

To close the experiment, in the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Image Classification Using Custom Training Loop

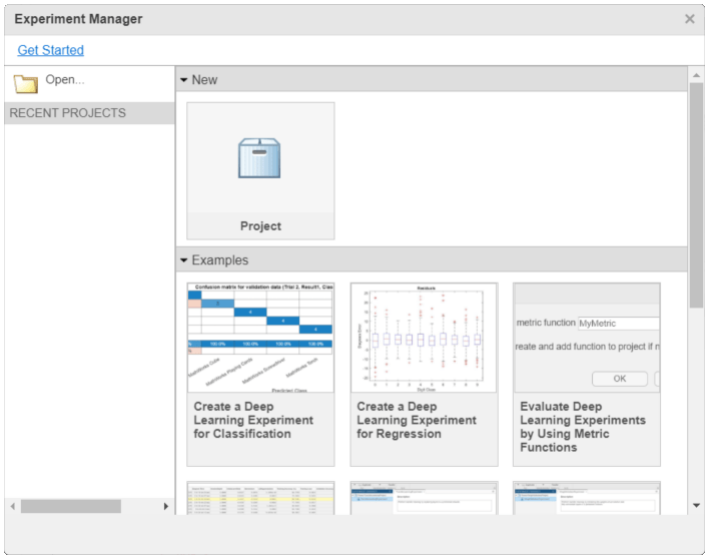
This example shows how to use the training experiment template for image classification using a custom training loop. With this template, you can quickly set up a custom training experiment.

Open the example to load a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`Experiment1`).

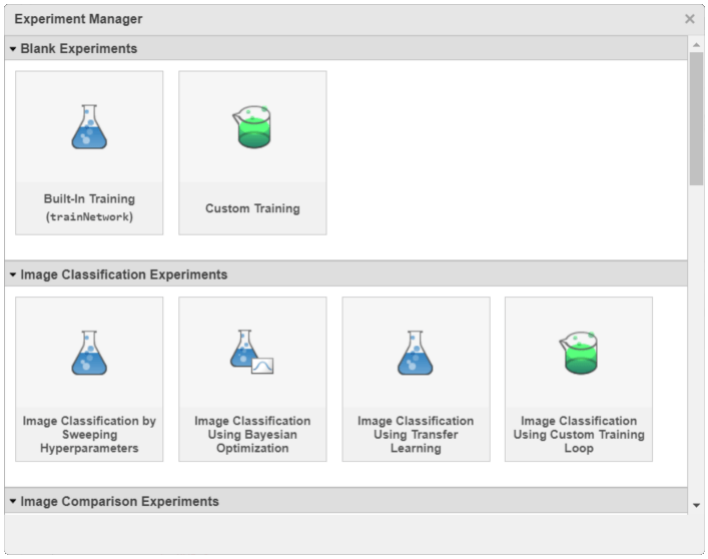


Alternatively, you can configure the experiment yourself by following these steps.

1. Open Experiment Manager. A dialog box provides links to the getting started tutorials and your recent projects, as well as buttons to create a new project or open an example from the documentation.



2. Under **New**, select **Project**. A dialog box lists several templates that support workflows including image classification, image regression, sequence classification, semantic segmentation, and custom training loops.



3. Under **Image Classification Experiments**, select **Image Classification Using Custom Training Loop**.

4. Select the location and name for a new project. Experiment Manager opens a new experiment in the project. The **Experiment** pane displays the description, hyperparameters, and training function that define the experiment.

3. In the **Description** field, enter a description of the experiment:

```
Classification of digits, using various initial learning rates.
```

4. Under **Hyperparameters**, replace the value of `myInitialLearnRate` with `0.0025:0.0025:0.015`. Verify that **Strategy** is set to `Exhaustive Sweep`.

5. Under **Training Function**, click **Edit**. The training function opens in MATLAB Editor. The training function specifies the training data, network architecture, training options, and training procedure used by the experiment. In this experiment, the training function has four sections.

- **Load Training Data** defines the training data for the experiment as 4-D arrays. The experiment uses the Digits data set, which consists of 5,000 28-by-28 pixel grayscale images of digits from 0 to 9, categorized by the digit they represent. For more information on this data set, see “Image Data Sets”.
- **Define Network Architecture** defines the architecture for a simple convolutional neural network for deep learning classification. To train the network with a custom training loop, the training function represents the network as a `dlnetwork` object.
- **Specify Training Options** defines the training options used by the experiment. In this experiment, the training function loads the values for the initial learning rate from the `myInitialLearnRate` entry in the hyperparameter table.
- **Train Model** defines the custom training loop used by the experiment. For each epoch, the custom training loop shuffles the data and iterates over mini-batches of data. For each mini-batch, the custom training loop evaluates the model gradients, state, and loss, determines the learning rate for the time-based decay learning rate schedule, and updates the network parameters. To track the progress of the training and record the value of the training loss, the training function uses the `experiments.Monitor` object `monitor`.

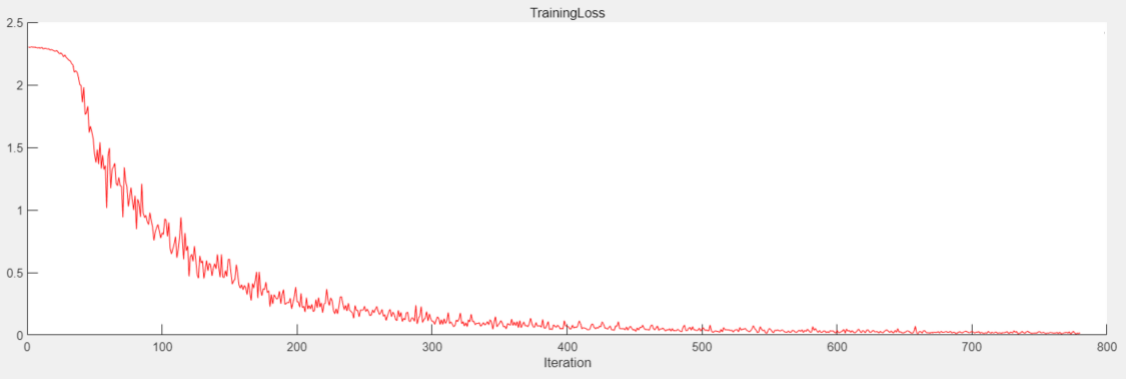
When you run the experiment, Experiment Manager trains the network defined by the training function six times. Each trial uses one of the learning rates specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the training loss for each trial.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	TrainingLoss
1	Complete	100.0%	0 hr 0 min 56 sec	0.0025	1.0177
2	Complete	100.0%	0 hr 1 min 10 sec	0.0050	0.4267
3	Running	90.0%	0 hr 0 min 44 sec	0.0075	0.1350
4	Running	10.0%	0 hr 0 min 24 sec	0.0100	1.1928
5	Queued	0.0%		0.0125	
6	Queued	0.0%		0.0150	

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



When the experiment finishes, you can sort the table by column or filter trials by using the **Filters** pane. You can also record observations by adding annotations to the results table. For more information, see “Sort, Filter, and Annotate Experiment Results” on page 1-47.

To test the performance of an individual trial, export the training output for the trial. On the **Experiment Manager** toolstrip, select **Export**. In this experiment, the training output is a structure that contains the values of the training loss and the trained network.

To close the experiment, in the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Configure Built-In Training Experiment

This example shows how to set up a built-in training experiment using the Experiment Manager app. Built-in training experiments rely on the `trainNetwork` function and support workflows such as image classification, image regression, sequence classification, and semantic segmentation.

Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment.

In the **Description** field, enter a description of the experiment.

Under **Hyperparameters**, select the strategy to use for your experiment.

- To sweep through a range of hyperparameter values, set **Strategy** to `Exhaustive Sweep`. In the hyperparameter table, specify the values of the hyperparameters used in the experiment. You can specify hyperparameter values as scalars or vectors with numeric, logical, or string values. For example, these are valid hyperparameter specifications:

  - `0.01`
  - `0.01:0.01:0.05`
  - `[0.01 0.02 0.04 0.08]`
  - `["sgdm" "rmsprop" "adam"]`

  When you run the experiment, Experiment Manager trains the network using every combination of the hyperparameter values specified in the table.
- To find optimal training options by using Bayesian optimization, set **Strategy** to `Bayesian Optimization`. In the hyperparameter table, specify these properties of the hyperparameters used in the experiment:

  - **Range** — Enter a two-element vector that gives the lower bound and upper bound of a real- or integer-valued hyperparameter, or a string array or cell array that lists the possible values of a categorical hyperparameter.
  - **Type** — Select `real` (real-valued hyperparameter), `integer` (integer-valued hyperparameter), or `categorical` (categorical hyperparameter).
  - **Transform** — Select `none` (no transform) or `log` (logarithmic transform). For `log`, the hyperparameter must be `real` or `integer` and positive. With this option, the hyperparameter is searched and modeled on a logarithmic scale.

  When you run the experiment, Experiment Manager searches for the best combination of hyperparameters. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials.

  To specify the duration of your experiment, under **Bayesian Optimization Options**, enter the maximum time (in seconds) and the maximum number of trials to run. Note that the actual run time and number of trials in your experiment can exceed these settings because Experiment Manager checks these options only when a trial finishes executing.

  Bayesian optimization requires Statistics and Machine Learning Toolbox. For more information, see “Tune Experiment Hyperparameters by Using Bayesian Optimization”.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table.

The output of the setup function must match the input of the `trainNetwork` function. This table lists the supported signatures for the setup function.

Goal of Experiment	Setup Function Signature

Train a network for image classification and regression tasks using the images and responses specified by `images` and the training options defined by `options`.	`function [images,layers,options] = Experiment_setup(p`
	`...`
	`end`

Train a network using the images specified by `images` and responses specified by `responses`.	`function [images,responses,layers,options] = Experime`
	`...`
	`end`

Train a network for sequence or time-series classification and regression tasks (for example, an LSTM or GRU network) using the sequences and responses specified by `sequences`.	`function [sequences,layers,options] = Experiment_setu`
	`...`
	`end`

Train a network using the sequences specified by `sequences` and responses specified by `responses`.	`function [sequences,reponses,layers,options] = Experi`
	`...`
	`end`

Train a network for feature classification or regression tasks (for example, a multilayer perceptron, or MLP, network) using the feature data and responses specified by `features`.	`function [features,layers,options] = Experiment_setup`
	`...`
	`end`

Train a network using the feature data specified by `features` and responses specified by `responses`.	`function [features,responses,layers,options] = Experi`
	`...`
	`end`

Note Experiment Manager does not support the execution of multiple trials in parallel when you set the training option `ExecutionEnvironment` to `"multi-gpu"` or `"parallel"` or when you enable the training option `DispatchInBackground`. Use these options to speed up your training only if you intend to run one trial of your experiment at a time. For more information, see “Use Experiment Manager to Train Networks in Parallel”.

The **Metrics** section specifies functions to evaluate the results of the experiment. The input to a metric function is a structure with three fields:

- `trainedNetwork` is the `SeriesNetwork` object or `DAGNetwork` object returned by the `trainNetwork` function. For more information, see Trained Network on page 1-0 .
- `trainingInfo` is a structure containing the training information returned by the `trainNetwork` function. For more information, see Training Information on page 1-0 .
- `parameters` is a structure with fields from the hyperparameter table.

The output of a metric function must be a scalar number, a logical value, or a string.

If your experiment uses Bayesian optimization, select a metric to optimize from the **Optimize** list. In the **Direction** list, specify that you want to `Maximize` or `Minimize` this metric. Experiment Manager uses this metric to determine the best combination of hyperparameters for your experiment. You can

choose a standard training or validation metric (such as accuracy, RMSE, or loss) or a custom metric from the table.

Configure Custom Training Experiment

This example shows how to set up a custom training experiment using the Experiment Manager app. Custom training experiments support workflows that require a training function other than `trainNetwork`. These workflows include:

- Training a network that is not defined by a layer graph.
- Training a network using a custom learning rate schedule.
- Updating the learnable parameters of a network by using a custom function.
- Training a generative adversarial network (GAN).
- Training a Siamese network.

Custom training experiments consist of a description, a table of hyperparameters, and a training function.

In the **Description** field, enter a description of the experiment.

Under **Hyperparameters**, select the strategy to use for your experiment.

- To sweep through a range of hyperparameter values, set **Strategy** to `Exhaustive Sweep`. In the hyperparameter table, specify the values of the hyperparameters used in the experiment. You can specify hyperparameter values as scalars or vectors with numeric, logical, or string values. For example, these are valid hyperparameter specifications:

  - `0.01`
  - `0.01:0.01:0.05`
  - `[0.01 0.02 0.04 0.08]`
  - `["sgdm" "rmsprop" "adam"]`

  When you run the experiment, Experiment Manager trains the network using every combination of the hyperparameter values specified in the table.
- To find optimal training options by using Bayesian optimization, set **Strategy** to `Bayesian Optimization`. In the hyperparameter table, specify these properties of the hyperparameters used in the experiment:

  - **Range** — Enter a two-element vector that gives the lower bound and upper bound of a real- or integer-valued hyperparameter, or a string array or cell array that lists the possible values of a categorical hyperparameter.
  - **Type** — Select `real` (real-valued hyperparameter), `integer` (integer-valued hyperparameter), or `categorical` (categorical hyperparameter).
  - **Transform** — Select `none` (no transform) or `log` (logarithmic transform). For `log`, the hyperparameter must be `real` or `integer` and positive. With this option, the hyperparameter is searched and modeled on a logarithmic scale.

  When you run the experiment, Experiment Manager searches for the best combination of hyperparameters. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials.

To specify the duration of your experiment, under **Bayesian Optimization Options**, enter the maximum time (in seconds) and the maximum number of trials to run. Note that the actual run time and number of trials in your experiment can exceed these settings because Experiment Manager checks these options only when a trial finishes executing.

Bayesian optimization requires Statistics and Machine Learning Toolbox. For more information, see “Use Bayesian Optimization in Custom Training Experiments”.

The **Training Function** specifies the training data, network architecture, training options, and training procedure used by the experiment. The inputs to the training function are:

- A structure with fields from the hyperparameter table
- An `experiments.Monitor` object that you can use to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots

Experiment Manager saves the output of the training function, so you can export it to the MATLAB workspace when the training is complete.

Note Both information and metric columns display numerical values in the results table for your experiment. Additionally, metric values are recorded in the training plot. Use information columns for values that you want to display in the results table but not in the training plot.

If your experiment uses Bayesian optimization, in the **Metrics** section, under **Optimize**, enter the name of a metric to optimize. In the **Direction** list, specify that you want to `Maximize` or `Minimize` this metric. Experiment Manager uses this metric to determine the best combination of hyperparameters for your experiment. You can choose any metric that you define using the `experiments.Monitor` object for the training function.

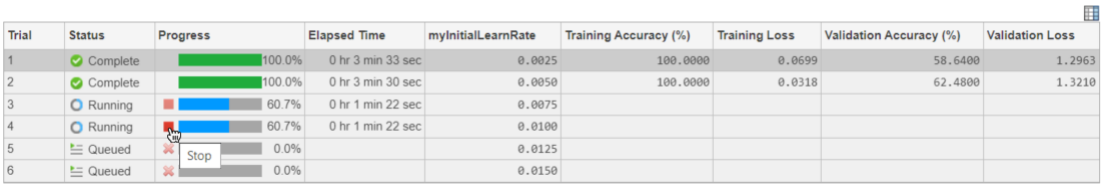
Stop and Restart Training

Experiment Manager provides two options for interrupting experiments:

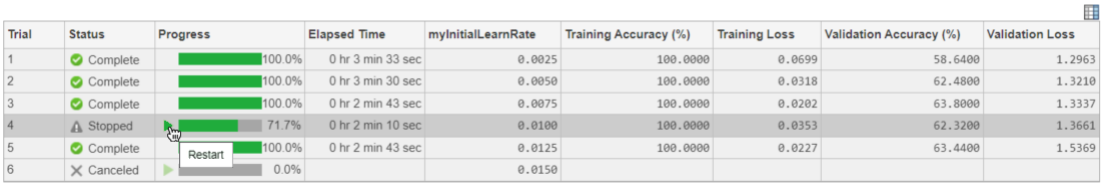
- **Stop** marks any running trials as `Stopped` and saves their results. When the experiment stops, you can display the training plot and export the training output for these trials.
- **Cancel** marks any running trials as `Canceled` and discards their results. When the experiment stops, you cannot display the training plot or export the training output for these trials.

Both options save the results of any completed trials and cancel any queued trials. Typically, **Cancel** is faster than **Stop**.

Instead of stopping an experiment, you can stop an individual trial that is running or cancel an individual queued trial. In the **Progress** column of the results table, click the Stop button or Cancel button for the trial.



When the training is complete, you can restart a trial that you stopped or canceled. In the **Progress** column of the results table, click the Restart button for the trial.



Alternatively, to restart all the trials that you canceled, in the Experiment Manager toolstrip, click **Restart All Canceled** .

Note Experiments that use Bayesian optimization support only the **Cancel** option. In addition, these experiments do not support restarting of canceled trials.

Custom training experiments that use exhaustive sweep support only the **Stop** option.

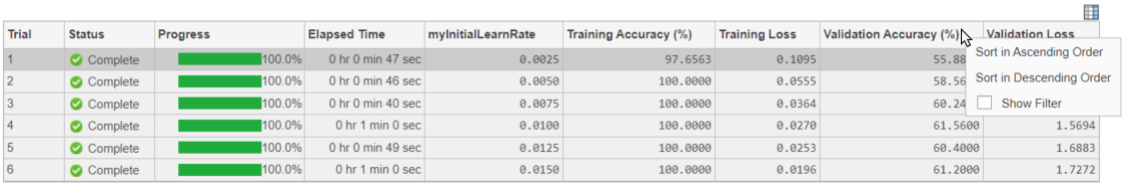
Sort, Filter, and Annotate Experiment Results

This example shows how to compare your results and record your observations after running an experiment.

When you run an experiment, Experiment Manager trains the network defined by the setup function multiple times. Each trial uses a different combination of hyperparameters. When the experiment finishes, a table displays training and validation metrics (such as accuracy, RMSE, and loss) for each trial. To compare the results of an experiment, you can use these metrics to sort the results table and filter trials.

To sort the trials in the results table, use the drop-down menu for the column corresponding to a training or validation metric.

1. Point to the header of a column by which you want to sort.
2. Click the triangle icon.
3. Select **Sort in Ascending Order** or **Sort in Descending Order**.

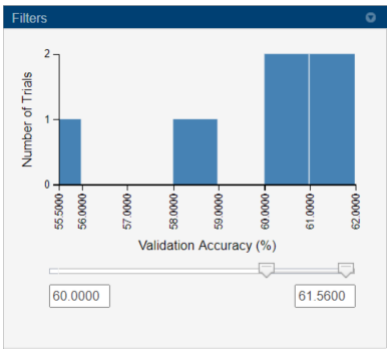


To filter trials from the results table, use the **Filters** pane.

1. On the **Experiment Manager** toolstrip, select **Filters**.

   The **Filters** pane shows histograms for the numeric metrics in the results table. To remove a histogram from the **Filters** pane, in the results table, open the drop-down menu for the corresponding column and clear the **Show Filter** check box.

2. Adjust the sliders under the histogram for the training or validation metric by which you want to filter.



   The results table shows only the trials with a metric value in the selected range.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
4	Complete	100.0%	0 hr 1 min 0 sec	0.0100	100.0000	0.0270	61.5600	1.5694
6	Complete	100.0%	0 hr 1 min 0 sec	0.0150	100.0000	0.0196	61.2000	1.7272
5	Complete	100.0%	0 hr 0 min 49 sec	0.0125	100.0000	0.0253	60.4000	1.6883
3	Complete	100.0%	0 hr 0 min 40 sec	0.0075	100.0000	0.0364	60.2400	1.5254

3. To restore all of the trials in the results table, close the **Experiment Result** pane and reopen the results from the **Experiment Browser** pane.

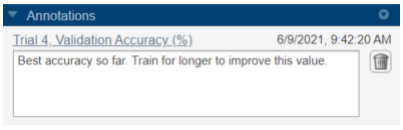
To record observations about the results of your experiment, add an annotation.

1. Right-click a cell in the results table and select **Add Annotation**. Alternatively, select a cell in the results table and, on the Experiment Manager toolstrip, select **Annotations** > **Add Annotation**.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
4	Complete	100.0%	0 hr 1 min 0 sec	0.0100	100.0000	0.0270		1.5694
6	Complete	100.0%	0 hr 1 min 0 sec	0.0150	100.0000	0.0196		1.7272
5	Complete	100.0%	0 hr 0 min 49 sec	0.0125	100.0000	0.0253		1.6883
3	Complete	100.0%	0 hr 0 min 40 sec	0.0075	100.0000	0.0364	60.2400	1.5254

Add Annotation
Highlight Annotation

2. In the **Annotations** pane, enter your observations in the text box.



   You can add multiple annotations for each cell in the results table. Each annotation is marked with a time stamp.

3. To highlight the cell that corresponds to an annotation, click the link above the annotation.

To open the **Annotations** pane and view all of your annotations, on the Experiment Manager toolstrip, select **Annotations > View Annotations**.

View Source of Past Experiment Definitions

This example shows how to inspect the configuration of an experiment that produced a given result.

After you run an experiment, you can open the **Experiment Source** pane to see a read-only copy of the experiment description and hyperparameter table, as well as links to all of the functions used by the experiment. You can use the information in this pane to track the configuration of data, network, and training options that produces each of your results.

For instance, suppose that you run an experiment multiple times. Each time that you run the experiment, you change the contents of the setup function but always use the same function name. The first time that you run the experiment, you use the default network provided by the experiment template for image classification. The second time that you run the experiment, you modify the setup function to load a pretrained GoogLeNet network, replacing the final layers with new layers for transfer learning. For an example that uses these two network architectures, see “Create a Deep Learning Experiment for Classification”.

On the first **Experiment Result** pane, click the **View Experiment Source** link. Experiment Manager opens an **Experiment Source** pane that contains the experiment definition that produced the first set of results. Click the link at the bottom of the pane to open the setup function that you used the first time you ran the experiment. You can copy this setup function to rerun the experiment using the simple classification network.

On the second **Experiment Result** pane, click the **View Experiment Source** link. Experiment Manager opens an **Experiment Source** pane that contains the experiment definition that produced the second set of results. Click the link at the bottom of the pane to open the setup function that you used the second time you ran the experiment. You can copy this setup function to rerun the experiment using transfer learning.

Experiment Manager stores a copy of all the functions that you use, so you do not have to manually rename these functions when you modify and rerun an experiment.

- “Create a Deep Learning Experiment for Classification”
- “Create a Deep Learning Experiment for Regression”
- “Evaluate Deep Learning Experiments by Using Metric Functions”
- “Tune Experiment Hyperparameters by Using Bayesian Optimization”
- “Use Bayesian Optimization in Custom Training Experiments”
- “Try Multiple Pretrained Networks for Transfer Learning”
- “Experiment with Weight Initializers for Transfer Learning”
- “Choose Training Configurations for LSTM Using Bayesian Optimization”
- “Run a Custom Training Experiment for Image Comparison”
- “Use Experiment Manager to Train Generative Adversarial Networks (GANs)”

Tips

- To visualize, build, and train a network without sweeping hyperparameters, you can use the **Deep Network Designer** app. After you train your network, generate a script to use as a starting point for your deep learning experiments. For more information, see “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager”.
- To run an experiment in parallel using MATLAB Online, you must have access to a Cloud Center cluster. For more information, see “Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online” (Parallel Computing Toolbox).
- To navigate Experiment Manager when using a mouse is not an option, use shortcut keyboards. For more information, see “Keyboard Shortcuts for Experiment Manager”.

See Also

Apps
Deep Network Designer

Functions
`dlnetwork` | `trainNetwork` | `trainingOptions`

Objects
`experiments.Monitor`

Topics
“Create a Deep Learning Experiment for Classification”
“Create a Deep Learning Experiment for Regression”
“Evaluate Deep Learning Experiments by Using Metric Functions”
“Tune Experiment Hyperparameters by Using Bayesian Optimization”
“Use Bayesian Optimization in Custom Training Experiments”
“Try Multiple Pretrained Networks for Transfer Learning”
“Experiment with Weight Initializers for Transfer Learning”
“Choose Training Configurations for LSTM Using Bayesian Optimization”
“Run a Custom Training Experiment for Image Comparison”
“Use Experiment Manager to Train Generative Adversarial Networks (GANs)”
“Use Experiment Manager to Train Networks in Parallel”
“Keyboard Shortcuts for Experiment Manager”

Introduced in R2020a

activations

Compute deep learning network layer activations

Syntax

```
act = activations(net,imds,layer)
act = activations(net,ds,layer)

act = activations(net,X,layer)
act = activations(net,X1,...,XN)
act = activations(net,sequences,layer)

act = activations(net,tbl,layer)

act = activations( ___ ,Name,Value)
```

Description

You can compute deep learning network layer activations on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value pair argument.

`act = activations(net,imds,layer)` returns network activations for a specific layer using the trained network `net` and the image data in the image datastore `imds`.

`act = activations(net,ds,layer)` returns network activations using the data in the datastore `ds`.

`act = activations(net,X,layer)` returns network activations using the image or feature data in the numeric array `X`.

`act = activations(net,X1,...,XN)` returns network activations for the data in the numeric arrays `X1`, ..., `XN` for the mutli-input network `net`. The input `Xi` corresponds to the network input `net.InputNames(i)`.

`act = activations(net,sequences,layer)` returns network activations for a recurrent network (for example, an LSTM or GRU network), where `sequences` contains sequence or time series predictors.

`act = activations(net,tbl,layer)` returns network activations using the data in the table `tbl`.

`act = activations( ___ ,Name,Value)` returns network activations with additional options specified by one or more name-value pair arguments. For example, `'OutputAs','rows'` specifies the activation output format as `'rows'`. Specify name-value pair arguments after all other input arguments.

Examples

Feature Extraction Using SqueezeNet

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.

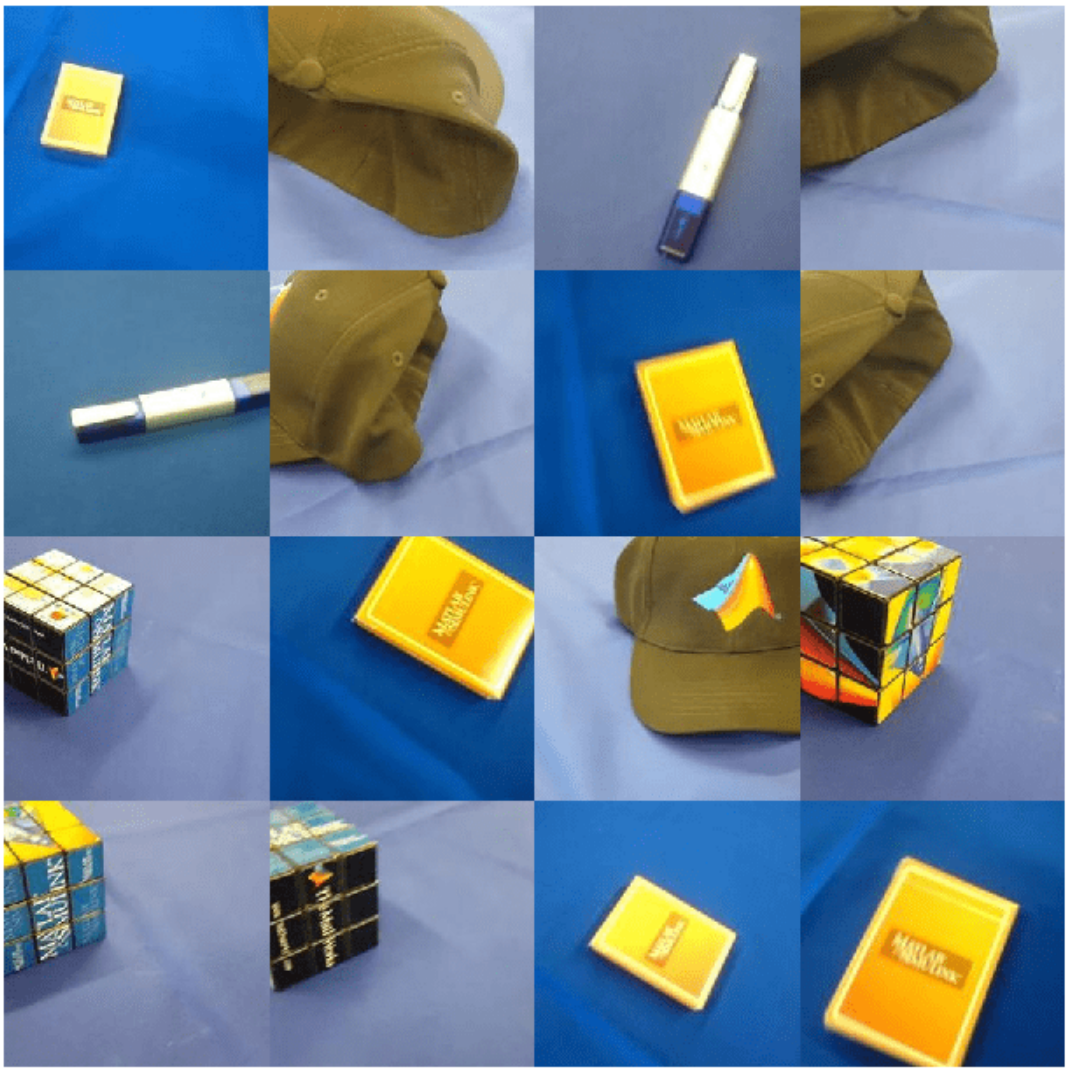
Load Data

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');
```

This very small data set now has 55 training images and 20 validation images. Display some sample images.

```
numImagesTrain = numel(imdsTrain.Labels);
idx = randperm(numImagesTrain,16);

I = imtile(imds, 'Frames', idx);

figure
imshow(I)
```



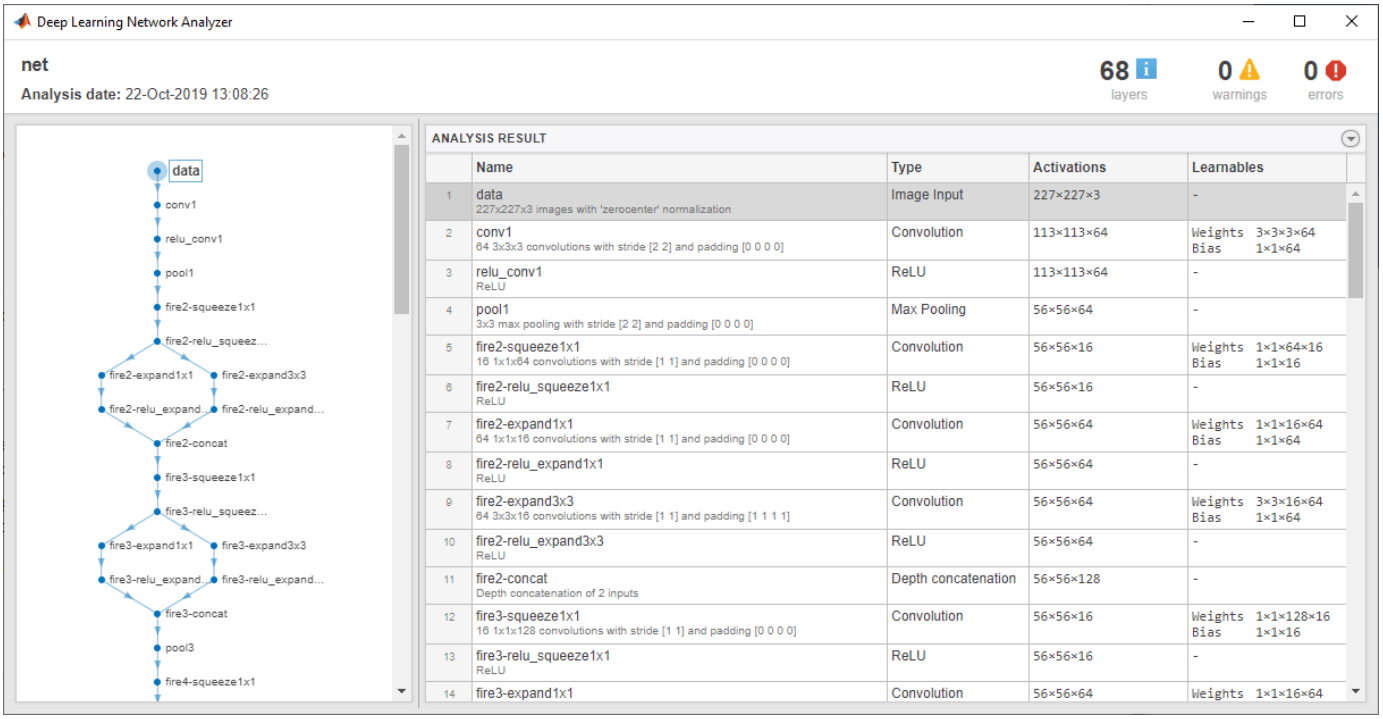
Load Pretrained Network

Load a pretrained SqueezeNet network. SqueezeNet is trained on more than a million images and can classify images into 1000 object categories, for example, keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = squeezenet;
```

Analyze the network architecture.

```
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
```

Extract Image Features

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 average pooling layer `'pool10'`. To get a lower level representation of the images, use an earlier layer in the network.

The network requires input images of size 227-by-227-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);

layer = 'pool10';
featuresTrain = activations(net,augimdsTrain,layer,'OutputAs','rows');
featuresTest = activations(net,augimdsTest,layer,'OutputAs','rows');
```

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).

```
mdl = fitcecoc(featuresTrain,YTrain);
```

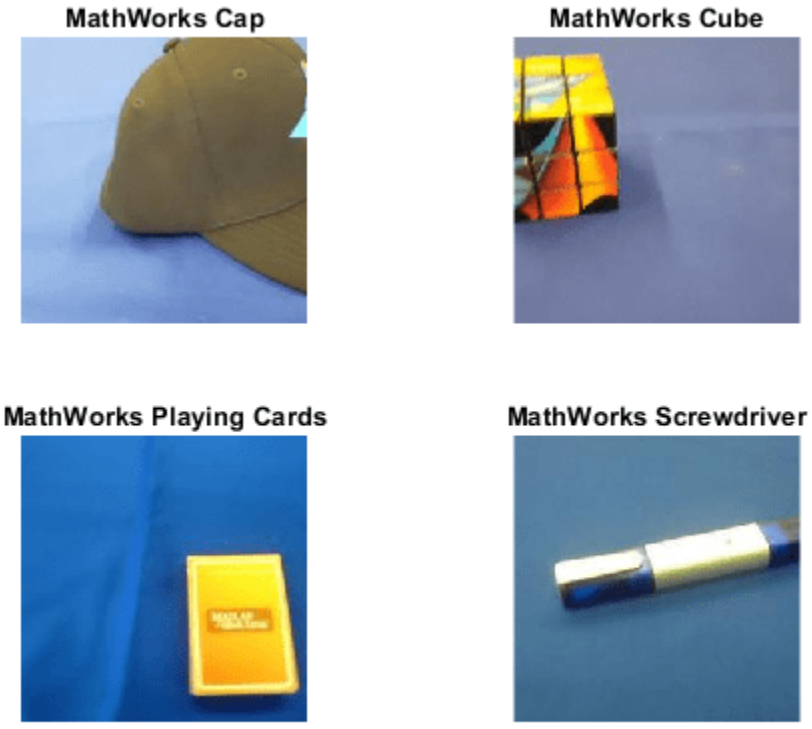
Classify Test Images

Classify the test images using the trained SVM model and the features extracted from the test images.

```
YPred = predict(mdl,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(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
```

This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

Input Arguments

`net` — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

`imds` — Image datastore

`ImageDatastore` object

Image datastore, specified as an `ImageDatastore` object.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

Tip Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.

`ds` — Datastore

datastore

Datastore for out-of-memory data and preprocessing. The datastore must return data in a table or a cell array. The format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output

Single input	Table or cell array, where the first column specifies the predictors.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom datastores must output tables.
		`data = read(ds)`
		
		`data =`
		
		`  4×1 table`
		
		`  Predictors`
		`  ____________________`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

Network Architecture	Datastore Output	Example Output

		`data = read(ds)`
		
		`data =`
		
		`  4×1 cell array`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

Multiple input	Cell array with at least `numInputs` columns, where `numInputs` is the number of network inputs.
	
	The first `numInputs` columns specify the predictors for each input.
	
	The order of inputs is given by the `InputNames` property of the network.
		`data = read(ds)`
		
		`data =`
		
		`  4×2 cell array`
		
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`

The format of the predictors depend on the type of data.

Data	Format of Predictors
2-D image	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the image, respectively.
3-D image	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the image, respectively.
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.
1-D image sequence	*h*-by-*c*-by-*s* array, where *h* and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.

Data	Format of Predictors
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
Features	*c*-by-1 column vector, where *c* is the number of features.

For more information, see “Datastores for Deep Learning”.

X — Image or feature data
numeric array

Image or feature data, specified as a numeric array. The size of the array depends on the type of input:

Input	Description
2-D images	A *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
3-D images	A *h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.
Features	A *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

If the array contains `NaN`s, then they are propagated through the network.

For networks with multiple inputs, you can specify multiple arrays `X1`, ..., `XN`, where `N` is the number of network inputs and the input `Xi` corresponds to the network input `net.InputNames(i)`.

For image input, if the `'OutputAs'` option is `'channels'`, then the images in the input data `X` can be larger than the input size of the image input layer of the network. For other output formats, the images in `X` must have the same size as the input size of the image input layer of the network.

sequences — Sequence or time series data
cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

`tbl` — Table of image or feature data

`table`

Table of image or feature data. Each row in the table corresponds to an observation.

The arrangement of predictors in the table columns depend on the type of input data.

Input	Predictors
Image data	• Absolute or relative file path to an image, specified as a character vector in a single column • Image specified as a 3-D numeric array Specify predictors in a single column.
Feature data	Numeric scalar. Specify predictors in the first `numFeatures` columns of the table, where `numFeatures` is the number of features of the input data.

This argument supports networks with a single input only.

Data Types: `table`

`layer` — Layer to extract activations from

numeric index | character vector

Layer to extract activations from, specified as a numeric index or a character vector.

To compute the activations of a `SeriesNetwork` object, specify the layer using its numeric index, or as a character vector corresponding to the layer name.

To compute the activations of a `DAGNetwork` object, specify the layer as the character vector corresponding to the layer name. If the layer has multiple outputs, specify the layer and output as the layer name, followed by the character "/", followed by the name of the layer output. That is, `layer` is of the form `'layerName/outputName'`.

Example: 3

Example: `'conv1'`

Example: `'mpool/out'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `activations(net,X,layer,'OutputAs','rows')`

`OutputAs` — Format of output activations

`'channels'` (default) | `'rows'` | `'columns'`

Format of output activations, specified as the comma-separated pair consisting of `'OutputAs'` and either `'channels'`, `'rows'`, or `'columns'`. For descriptions of the different output formats, see `act`.

For image input, if the `'OutputAs'` option is `'channels'`, then the images in the input data `X` can be larger than the input size of the image input layer of the network. For other output formats, the images in `X` must have the same size as the input size of the image input layer of the network.

Example: `'OutputAs','rows'`

`MiniBatchSize` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

Example: `'MiniBatchSize',256`

`SequenceLength` — Option to pad, truncate, or split input sequences

`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: `'SequenceLength','shortest'`

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

`Acceleration` — Performance optimization
`'auto'` (default) | `'mex'` | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'mex'` — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'none'` — Disable all acceleration.

The default option is `'auto'`. If `'auto'` is specified, MATLAB will apply a number of compatible optimizations. If you use the `'auto'` option, MATLAB does not ever generate a MEX function.

Using the 'Acceleration' options 'auto' and 'mex' can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The 'mex' option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The 'mex' option is only available when you are using a GPU. MEX acceleration supports single GPU execution using the name-value option 'ExecutionEvironment','gpu' only.

To use the 'mex' option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see "MEX Setup" (GPU Coder). GPU Coder is not required.

The 'mex' option does not support all layers. For a list of supported layers, see "Supported Layers" (GPU Coder). Only networks with an imageInputLayer are supported.

You cannot use MATLAB Compiler™ to deploy your network when using the 'mex' option.

Example: 'Acceleration','mex'

ExecutionEnvironment — Hardware resource
'auto' (default) | 'gpu' | 'cpu' | 'multi-gpu' | 'parallel'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.
- 'multi-gpu' — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- 'parallel' — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform computation. If the pool does not have GPUs, then computation takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see "Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud".

'gpu', 'multi-gpu', and 'parallel' options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

The 'multi-gpu' and 'parallel' options do not support recurrent neural networks (RNNs) containing lstmLayer, bilstmLayer, or gruLayer objects.

Example: 'ExecutionEnvironment','cpu'

Output Arguments

`act` — Activations from the network layer

numeric array | cell array

Activations from the network layer, returned as a numeric array or a cell array of numeric arrays. The format of `act` depends on the type of input data, the type of layer output, and the `'OutputAs'` option.

Image or Folded Sequence Output

If the layer outputs image or folded sequence data, then `act` is a numeric array.

`'OutputAs'`	`act`
`'channels'`	For 2-D image output, `act` is an *h*-by-*w*-by-*c*-by-*n* array, where *h*, *w*, and *c* are the height, width, and number of channels for the output of the chosen layer, respectively, and *n* is the number of images. In this case, `act(:,:,:,i)` contains the activations for the `i`th image. For 3-D image output, `act` is an *h*-by-*w*-by-*d*-by-*c*-by-*n* array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels for the output of the chosen layer, respectively, and *n* is the number of images. In this case, `act(:,:,:,:,i)` contains the activations for the `i`th image. For folded 2-D image sequence output, `act` is an *h*-by-*w*-by-*c*-by-(*n***s*) array, where *h*, *w*, and *c* are the height, width, and number of channels for the output of the chosen layer, respectively, *n* is the number of sequences, and *s* is the sequence length. In this case, `act(:,:,:,(t-1)*n+k)` contains the activations for time step `t` of the `k`th sequence. For folded 3-D image sequence output, `act` is an *h*-by-*w*-by-*d*-by-*c*-by-(*n***s*) array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels for the output of the chosen layer, respectively, *n* is the number of sequences, and *s* is the sequence length. In this case, `act(:,:,:,:,(t-1)*n+k)` contains the activations for time step `t` of the `k`th sequence.
`'rows'`	For 2-D and 3-D image output, `act` is an *n*-by-*m* matrix, where *n* is the number of images and `m` is the number of output elements from the layer. In this case, `act(i,:)` contains the activations for the `i`th image. For folded 2-D and 3-D image sequence output, `act` is an (*n***s*)-by-*m* matrix, where *n* is the number of sequences, *s* is the sequence length, and `m` is the number of output elements from the layer. In this case, `act((t-1)*n+k,:)` contains the activations for time step `t` of the `k`th sequence.

'OutputAs'	act
'columns'	For 2-D and 3-D image output, act is an *m*-by-*n* matrix, where *m* is the number of output elements from the chosen layer, and *n* is the number of images. In this case, act(:,i) contains the activations for the ith image. For folded 2-D and 3-D image sequence output, act is an *m*-by-(*n***s*) matrix, where *m* is the number of output elements from the chosen layer, *n* is the number of sequences, and *s* is the sequence length. In this case, act(:,(t-1)*n+k) contains the activations for time step t of the kth sequence.

Sequence Output

If layer has sequence output (for example, LSTM layers with output mode 'sequence'), then act is a cell array. In this case, the 'OutputAs' option must be 'channels'.

'OutputAs'	act
'channels'	For vector sequence output, act is a *n*-by-1 cell array, of *c*-by-*s* matrices, where *n* is the number of sequences, *c* is the number of features in the sequence, and *s* is the sequence length. For 2-D image sequence output, act is a *n*-by-1 cell array, of *h*-by-*w*-by-*c*-by-*s* matrices, where *n* is the number of sequences, *h*, *w*, and *c* are the height, width, and the number of channels of the images, respectively, and *s* is the sequence length. For 3-D image sequence output, act is a *n*-by-1 cell array, of *h*-by-*w*-by-*c*-by-*d*-by-*s* matrices, where *n* is the number of sequences, *h*, *w*, *d*, and *c* are the height, width, depth, and the number of channels of the images, respectively, and *s* is the sequence length. In these cases, act{i} contains the activations of the ith sequence.

Single Time-Step Output

If layer outputs a single time-step of a sequence (for example, an LSTM layer with output mode 'last'), then act is a numeric array.

'OutputAs'	act
'channels'	For a single time-step containing vector data, act is a *c*-by-*n* matrix, where *n* is the number of sequences and *c* is the number of features in the sequence. For a single time-step containing 2-D image data, act is a *h*-by-*w*-by-*c*-by-*n* array, where *n* is the number of sequences, *h*, *w*, and *c* are the height, width, and the number of channels of the images, respectively. For a single time-step containing 3-D image data, act is a *h*-by-*w*-by-*c*-by-*d*-by-*n* array, where *n* is the number of sequences, *h*, *w*, *d*, and *c* are the height, width, depth, and the number of channels of the images, respectively.

'OutputAs'	act
'rows'	*n*-by-*m* matrix, where *n* is the number of observations, and m is the number of output elements from the chosen layer. In this case, act(i,:) contains the activations for the ith sequence.
'columns'	*m*-by-*n* matrix, where *m* is the number of output elements from the chosen layer, and *n* is the number of observations. In this case, act(:,i) contains the activations for the ith image.

Algorithms

When you train a network using the trainNetwork function, or when you use prediction or validation functions with DAGNetwork and SeriesNetwork objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include trainNetwork, predict, classify, and activations. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - act = activations(net,X,layer)
  - act = activations(net,sequences,layer)
  - act = activations(__,Name,Value)
- The input X must not have variable size. The size of the input must be fixed at code generation time.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- The layer argument must be a constant during code generation.
- Only the 'OutputAs', 'MiniBatchSize', 'SequenceLength', 'SequencePaddingDirection', and 'SequencePaddingValue' name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- The format of the output activations must be 'channels'.

- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

For more information about generating code for deep learning neural networks, see “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:
  - `act = activations(net,X,layer)`
  - `act = activations(net,sequences,layer)`
  - `act = activations(__,Name,Value)`
- The input `X` must not have variable size. The size of the input must be fixed at code generation time.
- GPU code generation does not support `gpuArray` inputs to the `activations` function.
- The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences. The ARM® `Compute Library` for GPU does not support recurrent networks.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- The `layer` argument must be a constant during code generation.
- Only the `'OutputAs'`, `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- The format of the output activations must be `'channels'`.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- GPU code generation for the `activations` function supports inputs that are defined as half-precision floating point data types. For more information, see `half`.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- When input data is a `gpuArray`, a cell array or table containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`predict` | `classify` | `deepDreamImage` | `trainNetwork`

Topics
“Transfer Learning Using Pretrained Network”
“Visualize Activations of a Convolutional Neural Network”
“Visualize Activations of LSTM Network”
“Deep Learning in MATLAB”

Introduced in R2016a

AcceleratedFunction

Accelerated deep learning function

Description

An `AcceleratedFunction` stores traces of the underlying function

Reusing a cached trace depends on the function inputs and outputs:

- For any `dlarray` object or structure of `dlarray` object inputs, the trace depends on the size, format, and underlying datatype of the `dlarray`. That is, the accelerated function triggers a new trace for `dlarray` inputs with size, format, or underlying datatype not contained in the cache. Any `dlarray` inputs differing only by value to a previously cached trace do not trigger a new trace.
- For any `dlnetwork` inputs, the trace depends on the size, format, and underlying datatype of the `dlnetwork` state and learnable parameters. That is, the accelerated function triggers a new trace for `dlnetwork` inputs with learnable parameters or state with size, format, and underlying datatype not contained in the cache. Any `dlnetwork` inputs differing only by the value of the state and learnable parameters to a previously cached trace do not trigger a new trace.
- For other types of input, the trace depends on the values of the input. That is, the accelerated function triggers a new trace for other types of input with value not contained in the cache. Any other inputs that have the same value as a previously cached trace do not trigger a new trace.
- The trace depends on the number of function outputs. That is, the accelerated function triggers a new trace for function calls with previously unseen numbers of output arguments. Any function calls with the same number of output arguments as a previously cached trace do not trigger a new trace.

When necessary, the software caches any new traces by evaluating the underlying function and caching the resulting trace in the `AcceleratedFunction` object.

The returned `AcceleratedFunction` object caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` objects, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

Invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Note When using the `dlfeval` function, the software automatically accelerates the `forward` and `predict` functions for `dlnetwork` input. If you accelerate a deep learning function where the majority of the computation takes place in calls to the `forward` or `predict` functions for `dlnetwork` input, then you might not see an improvement in training time.

Caution An `AcceleratedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the accelerated function, then clear the cache using the `clearCache` object function or alternatively use the command `clear functions`.

Creation

To create an `AcceleratedFunction` object, use the `dlaccelerate` function.

Properties

`Function` — Underlying function

function handle

This property is read-only.

Underlying function, specified as a function handle.

Data Types: `function_handle`

`Enabled` — Flag to enable tracing

`true` (default) | `false`

Flag to enable tracing, specified as `true` or `false`.

Data Types: `logical`

`CacheSize` — Size of cache

50 (default) | positive integer

Size of cache, specified as a positive integer.

The cache size corresponds to the maximum number of input and output combinations to cache.

Data Types: `double`

`HitRate` — Cache hit rate

scalar in the range [0,100]

This property is read-only.

Cache hit rate, specified as a scalar in the range [0,100].

The cache hit rate corresponds to the percentage of reused evaluations.

Data Types: `double`

`Occupancy` — Cache occupancy

scalar in the range [0,100]

This property is read-only.

Cache occupancy, specified as a scalar in the range [0,100].

The cache occupancy corresponds to the percentage of the cache in use.

Data Types: `double`

CheckMode — Check mode
`'none'` (default) | `'tolerance'`

Check mode, specified as one of the following:

- `'none'` – Do not check accelerated results.
- `'tolerance'` – Check that the accelerated results and the results of the underlying function are within the tolerance given by the `CheckTolerance` property. If the values are not within this tolerance, then the function throws a warning.

CheckTolerance — Check tolerance
`1e-4` (default) | positive scalar

Check tolerance, specified as a positive scalar.

If the `CheckMode` property is `'tolerance'`, then the function checks that the accelerated results and the results of the underlying function are within the tolerance given by the `CheckTolerance` property. If the values are not within this tolerance, then the function throws a warning.

Data Types: `double`

Object Functions

clearCache Clear accelerated deep learning function trace cache

Examples

Accelerate Model Gradients Function

Load the `dlnetwork` object and class names from the MAT file `dlnetDigits.mat`.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Accelerate the model gradients function `modelGradients` listed at the end of the example.

```
fun = @modelGradients;
accfun = dlaccelerate(fun);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

View the properties of the accelerated function. Because the cache is empty, the `Occupancy` property is 0.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @modelGradients
           Enabled: 1
         CacheSize: 50
           HitRate: 0
```

```
         Occupancy: 0
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

The returned `AcceleratedFunction` object stores the traces of underlying function calls and reuses the cached result when the same input pattern reoccurs. To use the accelerated function in a custom training loop, replace calls to the model gradients function with calls to the accelerated function. You can invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Evaluate the accelerated model gradients function with random data using the `dlfeval` function.

```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');

T = categorical(classNames(randi(10,[128 1])));
T = onehotencode(T,2)';
dlT = dlarray(T,'CB');

[gradients,state,loss] = dlfeval(accfun,dlnet,dlX,dlT);
```

View the `Occupancy` property of the accelerated function. Because the function has been evaluated, the cache is nonempty.

```
accfun.Occupancy
```

```
ans = 2
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `dlT` 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, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,dlT)

[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,dlT);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Clear Cache of Accelerated Function

Load the `dlnetwork` object and class names from the MAT file `dlnetDigits.mat`.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Accelerate the model gradients function `modelGradients` listed at the end of the example.

```
fun = @modelGradients;
accfun = dlaccelerate(fun);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

View the properties of the accelerated function. Because the cache is empty, the `Occupancy` property is 0.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

         Function: @modelGradients
          Enabled: 1
        CacheSize: 50
          HitRate: 0
        Occupancy: 0
        CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

The returned `AcceleratedFunction` object stores the traces of underlying function calls and reuses the cached result when the same input pattern reoccurs. To use the accelerated function in a custom training loop, replace calls to the model gradients function with calls to the accelerated function. You can invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Evaluate the accelerated model gradients function with random data using the `dlfeval` function.

```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');

T = categorical(classNames(randi(10,[128 1])));
T = onehotencode(T,2)';
dlT = dlarray(T,'CB');

[gradients,state,loss] = dlfeval(accfun,dlnet,dlX,dlT);
```

View the `Occupancy` property of the accelerated function. Because the function has been evaluated, the cache is nonempty.

```
accfun.Occupancy
```

```
ans = 2
```

Clear the cache using the `clearCache` function.

```
clearCache(accfun)
```

View the `Occupancy` property of the accelerated function. Because the cache has been cleared, the cache is empty.

```
accfun.Occupancy
```

```
ans = 0
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `dlT` 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, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,flT)

[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,flT);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Check Accelerated Deep Learning Function Outputs

This example shows how to check that the outputs of accelerated functions match the outputs of the underlying function.

In some cases, the outputs of accelerated functions differ to the outputs of the underlying function. For example, you must take care when accelerating functions that use random number generation, such as a function that generates random noise to add to the network input. When caching the trace of a function that generates random numbers that are not `dlarray` objects, the accelerated function caches resulting random numbers in the trace. When reusing the trace, the accelerated function uses the cached random values. The accelerated function does not generate new random values.

To check that the outputs of the accelerated function match the outputs of the underlying function, use the `CheckMode` property of the accelerated function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ by more than a specified tolerance, the accelerated function throws a warning.

Accelerate the function `myUnsupportedFun`, listed at the end of the example using the `dlaccelerate` function. The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
accfun = dlaccelerate(@myUnsupportedFun)
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @myUnsupportedFun
           Enabled: 1
         CacheSize: 50
           HitRate: 0
         Occupancy: 0
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun.CheckMode = 'tolerance'
```

```
accfun =
  AcceleratedFunction with properties:

         Function: @myUnsupportedFun
          Enabled: 1
        CacheSize: 50
          HitRate: 0
        Occupancy: 0
        CheckMode: 'tolerance'
    CheckTolerance: 1.0000e-04
```

Evaluate the accelerated function with an array of ones as input, specified as a `dlarray` input.

```
dlX = dlarray(ones(3,3));
dlY = accfun(dlX)
```

```
dlY =
  3x3 dlarray

    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Evaluate the accelerated function again with the same input. Because the accelerated function reuses the cached random noise values instead of generating new random values, the outputs of the reused trace differs from the outputs of the underlying function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ, the accelerated function throws a warning.

```
dlY = accfun(dlX)
```

```
Warning: Accelerated outputs differ from underlying function outputs.

dlY =
  3x3 dlarray

    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Random number generation using the `'like'` option of the `rand` function with a `dlarray` object supports acceleration. To use random number generation in an accelerated function, ensure that the function uses the `rand` function with the `'like'` option set to a traced `dlarray` object (a `dlarray` object that depends on an input `dlarray` object).

Accelerate the function `mySupportedFun`, listed at the end of the example. The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
accfun2 = dlaccelerate(@mySupportedFun);
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun2)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun2.CheckMode = 'tolerance';
```

Evaluate the accelerated function twice with the same input as before. Because the outputs of the reused cache match the outputs of the underlying function, the accelerated function does not throw a warning.

```
dlY = accfun2(dlX)
```

```
dlY =
  3x3 dlarray

    1.7922    1.0357    1.6787
    1.9595    1.8491    1.7577
    1.6557    1.9340    1.7431
```

```
dlY = accfun2(dlX)
```

```
dlY =
  3x3 dlarray

    1.3922    1.7060    1.0462
    1.6555    1.0318    1.0971
    1.1712    1.2769    1.8235
```

Checking the outputs match requires extra processing and increases the time required for function evaluation. After checking the outputs, set the `CheckMode` property to `'none'`.

```
accfun1.CheckMode = 'none';
accfun2.CheckMode = 'none';
```

Example Functions

The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
function out = myUnsupportedFun(dlX)

sz = size(dlX);
noise = rand(sz);
out = dlX + noise;

end
```

The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
function out = mySupportedFun(dlX)

sz = size(dlX);
noise = rand(sz,'like',dlX);
out = dlX + noise;
```

```
end
```

See Also

`dlaccelerate` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Topics

“Deep Learning Function Acceleration for Custom Training Loops”
“Accelerate Custom Training Loop Functions”
“Check Accelerated Deep Learning Function Outputs”
“Evaluate Performance of Accelerated Deep Learning Function”

Introduced in R2021a

adamupdate

Update parameters using adaptive moment estimation (Adam)

Syntax

```
[dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,averageGrad,
averageSqGrad,iteration)
[params,averageGrad,averageSqGrad] = adamupdate(params,grad,averageGrad,
averageSqGrad,iteration)
[ ___ ] = adamupdate( ___ learnRate,gradDecay,sqGradDecay,epsilon)
```

Description

Update the network learnable parameters in a custom training loop using the adaptive moment estimation (Adam) algorithm.

Note This function applies the Adam optimization algorithm to update network parameters in custom training loops that use networks defined as `dlnetwork` objects or model functions. If you want to train a network defined as a `Layer` array or as a `LayerGraph`, use the following functions:

- Create a `TrainingOptionsADAM` object using the `trainingOptions` function.
- Use the `TrainingOptionsADAM` object with the `trainNetwork` function.

`[dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,averageGrad, averageSqGrad,iteration)` updates the learnable parameters of the network `dlnet` using the Adam algorithm. Use this syntax in a training loop to iteratively update a network defined as a `dlnetwork` object.

`[params,averageGrad,averageSqGrad] = adamupdate(params,grad,averageGrad, averageSqGrad,iteration)` updates the learnable parameters in `params` using the Adam algorithm. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined using functions.

`[ ___ ] = adamupdate( ___ learnRate,gradDecay,sqGradDecay,epsilon)` also specifies values to use for the global learning rate, gradient decay, square gradient decay, and small constant epsilon, in addition to the input arguments in previous syntaxes.

Examples

Update Learnable Parameters Using `adamupdate`

Perform a single adaptive moment estimation update step with a global learning rate of `0.05`, gradient decay factor of `0.75`, and squared gradient decay factor of `0.95`.

Create the parameters and parameter gradients as numeric arrays.

```
params = rand(3,3,4);
grad = ones(3,3,4);
```

Initialize the iteration counter, average gradient, and average squared gradient for the first iteration.

```
iteration = 1;
averageGrad = [];
averageSqGrad = [];
```

Specify custom values for the global learning rate, gradient decay factor, and squared gradient decay factor.

```
learnRate = 0.05;
gradDecay = 0.75;
sqGradDecay = 0.95;
```

Update the learnable parameters using `adamupdate`.

```
[params,averageGrad,averageSqGrad] = adamupdate(params,grad,averageGrad,averageSqGrad,iteration,
```

Update the iteration counter.

```
iteration = iteration + 1;
```

Train Network Using adamupdate

Use `adamupdate` to train a network using the Adam algorithm.

Load Training Data

Load the digits training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

Define Network

Define the network and specify the average image value 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')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of the example. The function takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```
miniBatchSize = 128;
numEpochs = 20;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

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

Visualize the training progress in a plot.

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

Train Network

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the `adamupdate` function. At the end of each epoch, display the training progress.

Initialize the training progress plot.

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

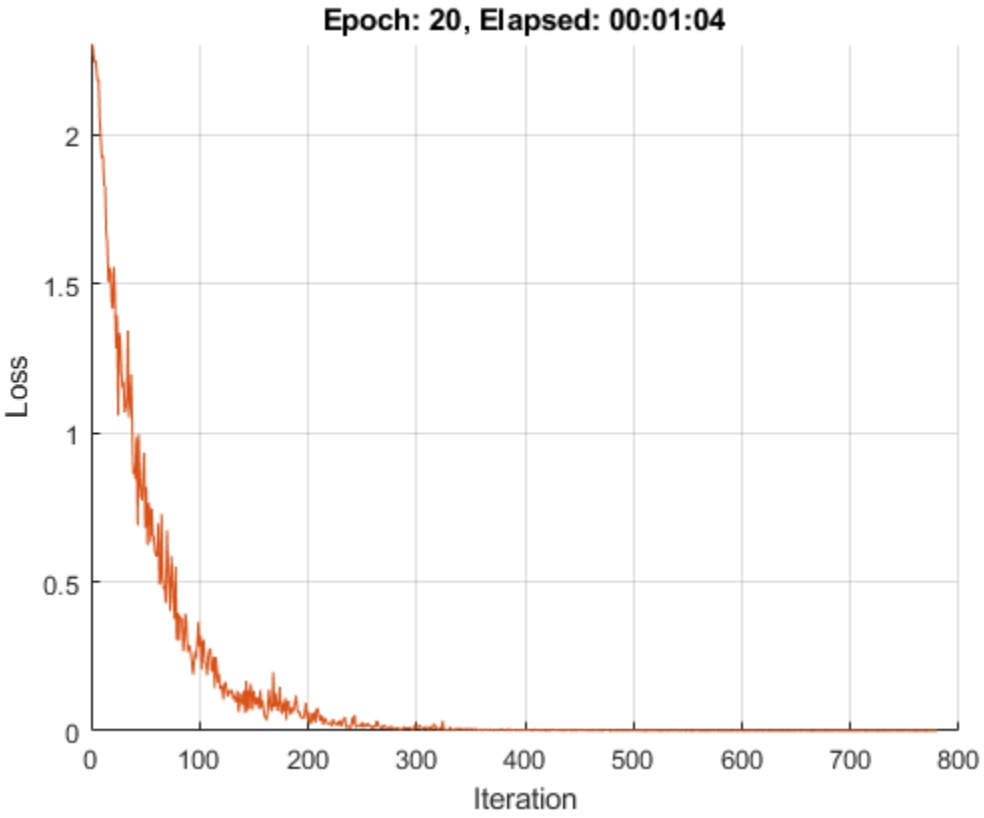
Initialize the average gradients and squared average gradients.

```
averageGrad = [];
averageSqGrad = [];
```

Train the network.

```
iteration = 0;
start = tic;

for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);
```

```
        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 a dlarray.
            dlX = dlarray(single(X),'SSCB');

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

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients helper function.
            [grad,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

            % Update the network parameters using the Adam optimizer.
            [dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,averageGrad,averageSqGrad,itera

            % 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 Network

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

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

To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.9896
```

Model Gradients Function

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

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

    dlYPred = forward(dlnet,dlX);

    loss = crossentropy(dlYPred,Y);

    gradients = dlgradient(loss,dlnet.Learnables);

end
```

Input Arguments

`dlnet` — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object. `dlnet.Learnables` is a table with three variables:

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

The input argument `grad` must be a table of the same form as `dlnet.Learnables`.

`params` — Network learnable parameters

`dlarray` | numeric array | cell array | structure | table

Network learnable parameters, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

If you specify `params` as a table, it must contain the following three variables:

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

You can specify `params` as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `grad` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

Data Types: `single` | `double` | `struct` | `table` | `cell`

grad — Gradients of the loss
`dlarray` | numeric array | cell array | structure | table

Gradients of the loss, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `grad` depends on the input network or learnable parameters. The following table shows the required format for `grad` for possible inputs to `adamupdate`.

Input	Learnable Parameters	Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.

You can obtain `grad` from a call to `dlfeval` that evaluates a function that contains a call to `dlgradient`. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

averageGrad — Moving average of parameter gradients
`[]` | `dlarray` | numeric array | cell array | structure | table

Moving average of parameter gradients, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `averageGrad` depends on the input network or learnable parameters. The following table shows the required format for `averageGrad` for possible inputs to `adamupdate`.

Input	Learnable Parameters	Average Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `averageGrad` must have a `Value` variable consisting of cell arrays that contain the average gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `averageGrad` must have a `Value` variable consisting of cell arrays that contain the average gradient of each learnable parameter.

If you specify `averageGrad` and `averageSqGrad` as empty arrays, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageGrad` output of a previous call to `adamupdate` as the `averageGrad` input.

`averageSqGrad` — Moving average of squared parameter gradients
`[]` | `dlarray` | numeric array | cell array | structure | table

Moving average of squared parameter gradients, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `averageSqGrad` depends on the input network or learnable parameters. The following table shows the required format for `averageSqGrad` for possible inputs to `adamupdate`.

Input	Learnable parameters	Average Squared Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `averageSqGrad` must have a `Value` variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables and ordering as `params`. `averageSqGrad` must have a `Value` variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.

If you specify `averageGrad` and `averageSqGrad` as empty arrays, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageSqGrad` output of a previous call to `adamupdate` as the `averageSqGrad` input.

`iteration` — Iteration number

positive integer

Iteration number, specified as a positive integer. For the first call to `adamupdate`, use a value of `1`. You must increment `iteration` by `1` for each subsequent call in a series of calls to `adamupdate`. The Adam algorithm uses this value to correct for bias in the moving averages at the beginning of a set of iterations.

`learnRate` — Global learning rate

`0.001` (default) | positive scalar

Global learning rate, specified as a positive scalar. The default value of `learnRate` is `0.001`.

If you specify the network parameters as a `dlnetwork`, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

gradDecay — Gradient decay factor
0.9 (default) | positive scalar between 0 and 1

Gradient decay factor, specified as a positive scalar between 0 and 1. The default value of gradDecay is 0.9.

sqGradDecay — Squared gradient decay factor
0.999 (default) | positive scalar between 0 and 1

Squared gradient decay factor, specified as a positive scalar between 0 and 1. The default value of sqGradDecay is 0.999.

epsilon — Small constant
1e-8 (default) | positive scalar

Small constant for preventing divide-by-zero errors, specified as a positive scalar. The default value of epsilon is 1e-8.

Output Arguments

dlnet — Updated network
dlnetwork object

Network, returned as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object.

params — Updated network learnable parameters
dlarray | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a dlarray, a numeric array, a cell array, a structure, or a table with a Value variable containing the updated learnable parameters of the network.

averageGrad — Updated moving average of parameter gradients
dlarray | numeric array | cell array | structure | table

Updated moving average of parameter gradients, returned as a dlarray, a numeric array, a cell array, a structure, or a table.

averageSqGrad — Updated moving average of squared parameter gradients
dlarray | numeric array | cell array | structure | table

Updated moving average of squared parameter gradients, returned as a dlarray, a numeric array, a cell array, a structure, or a table.

More About

Adam

The function uses the adaptive moment estimation (Adam) algorithm to update the learnable parameters. For more information, see the definition of the Adam algorithm under “Stochastic Gradient Descent” on page 1-1368 on the trainingOptions reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.
  - `grad`
  - `averageGrad`
  - `averageSqGrad`
  - `params`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlnetwork` | `dlarray` | `dlupdate` | `rmspropupdate` | `sgdmupdate` | `forward` | `dlgradient` | `dlfeval`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”

Introduced in R2019b

additionLayer

Addition layer

Description

An addition layer adds inputs from multiple neural network layers element-wise.

Specify the number of inputs to the layer when you create it. The inputs to the layer have the names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer by using `connectLayers` or `disconnectLayers`. All inputs to an addition layer must have the same dimension.

Creation

Syntax

```
layer = additionLayer(numInputs)
layer = additionLayer(numInputs,'Name',name)
```

Description

`layer = additionLayer(numInputs)` creates an addition layer that adds `numInputs` inputs element-wise. This function also sets the `NumInputs` property.

`layer = additionLayer(numInputs,'Name',name)` also sets the `Name` property.

Properties

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer greater than or equal to 2.

The inputs have the names `'in1','in2',...,'inN'`, where `N` is `NumInputs`. For example, if `NumInputs` is 3, then the inputs have the names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer using the `connectLayers` or `disconnectLayers` functions.

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`InputNames` — Input Names
`{'in1','in2',…,'inN'}` (default)

Input names, specified as `{'in1','in2',...,'inN'}`, where `N` is the number of inputs of the layer.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create and Connect Addition Layer

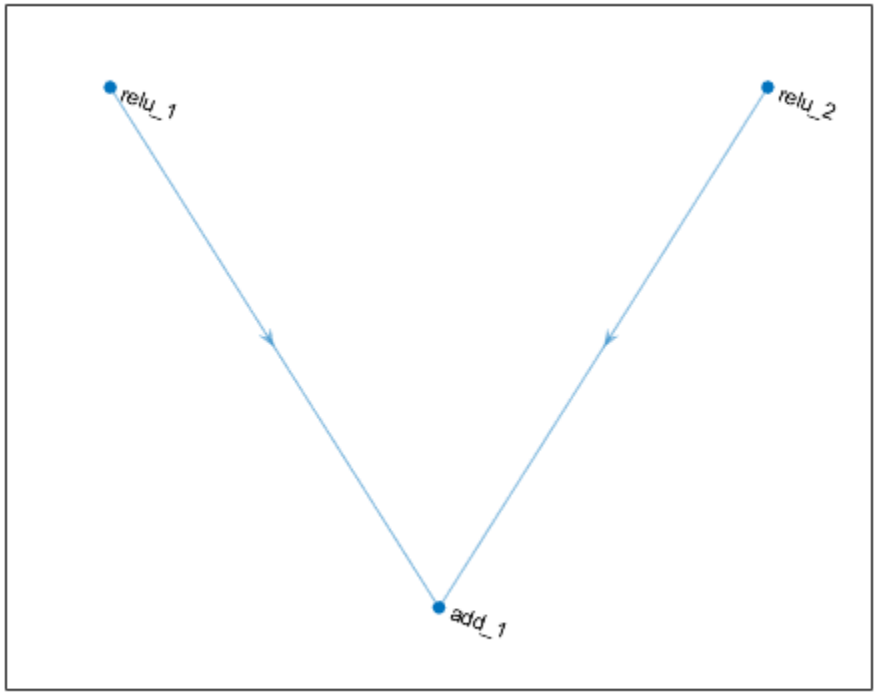
Create an addition layer with two inputs and the name `'add_1'`.

```
add = additionLayer(2,'Name','add_1')
```

```
add =
  AdditionLayer with properties:

         Name: 'add_1'
    NumInputs: 2
   InputNames: {'in1'  'in2'}
```

Create two ReLU layers and connect them to the addition layer. The addition layer sums the outputs from the ReLU layers.

```
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph;
lgraph = addLayers(lgraph,relu_1);
lgraph = addLayers(lgraph,relu_2);
lgraph = addLayers(lgraph,add);

lgraph = connectLayers(lgraph,'relu_1','add_1/in1');
lgraph = connectLayers(lgraph,'relu_2','add_1/in2');

plot(lgraph)
```



Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

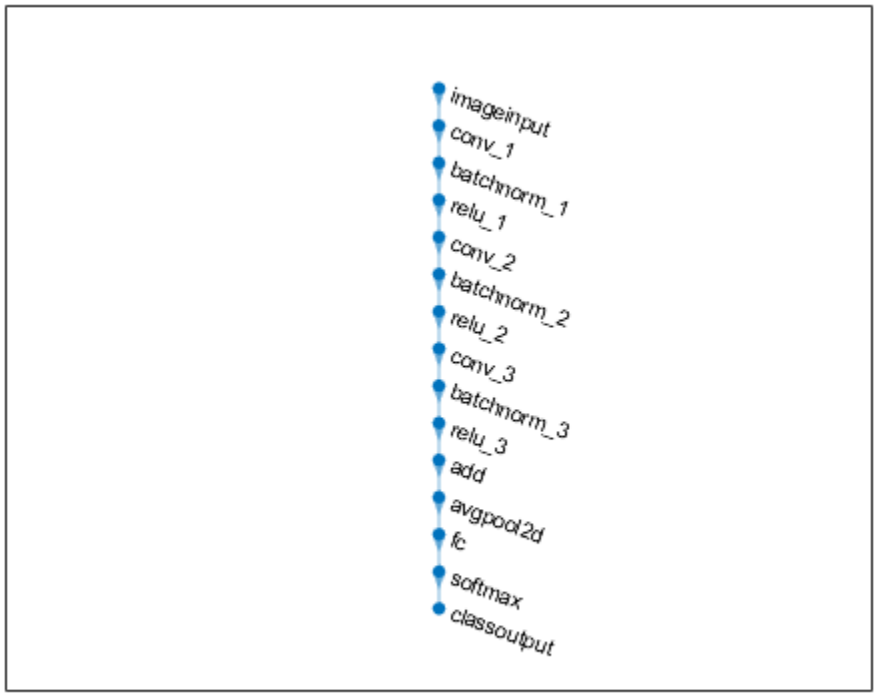
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(5,16,'Padding','same')
    batchNormalizationLayer
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
```

```
    reluLayer

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

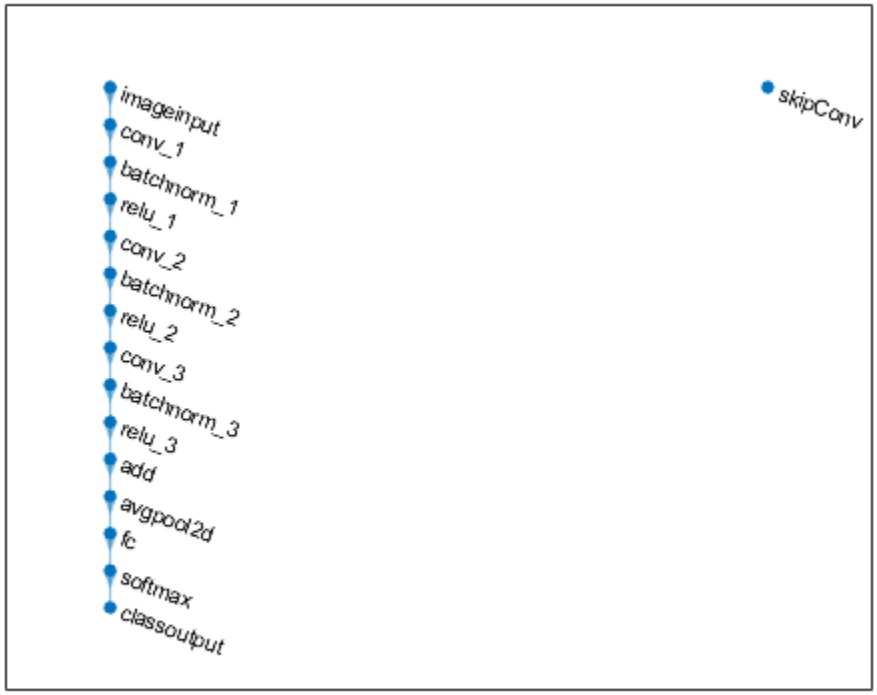
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



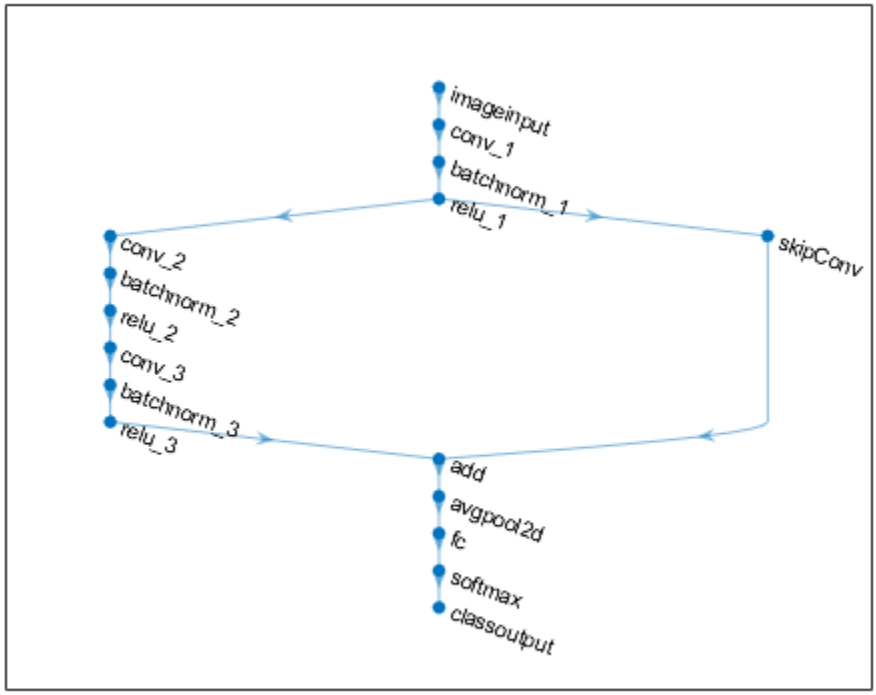
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 1-by-1 convolutional layer. To check that the layer is in the graph, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` layer to the `'add'` layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named `'in1'` and `'in2'`. The third ReLU layer is already connected to the `'in1'` input. Connect the `'relu_1'` layer to the `'skipConv'` layer and the `'skipConv'` layer to the `'in2'` input of the `'add'` layer. The addition layer now sums the outputs of the third ReLU layer and the `'skipConv'` layer. To check that the layers are connected correctly, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

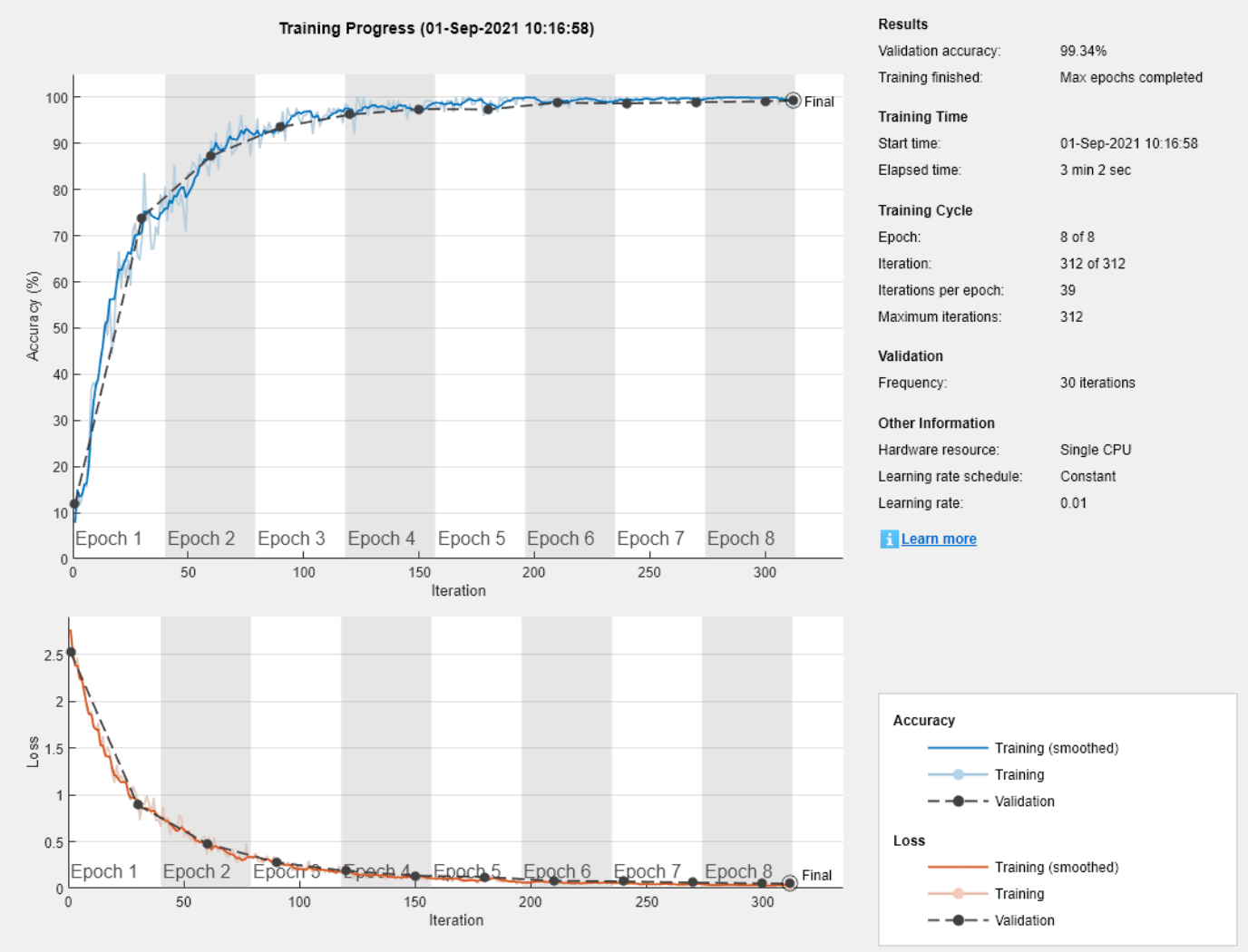


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



Display the properties of the trained network. The network is a `DAGNetwork` object.

```
net
```

```
net =
  DAGNetwork with properties:

         Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9934
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `layerGraph` | `depthConcatenationLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”

Introduced in R2017b

addLayers

Add layers to layer graph

Syntax

```
newlgraph = addLayers(lgraph,larray)
```

Description

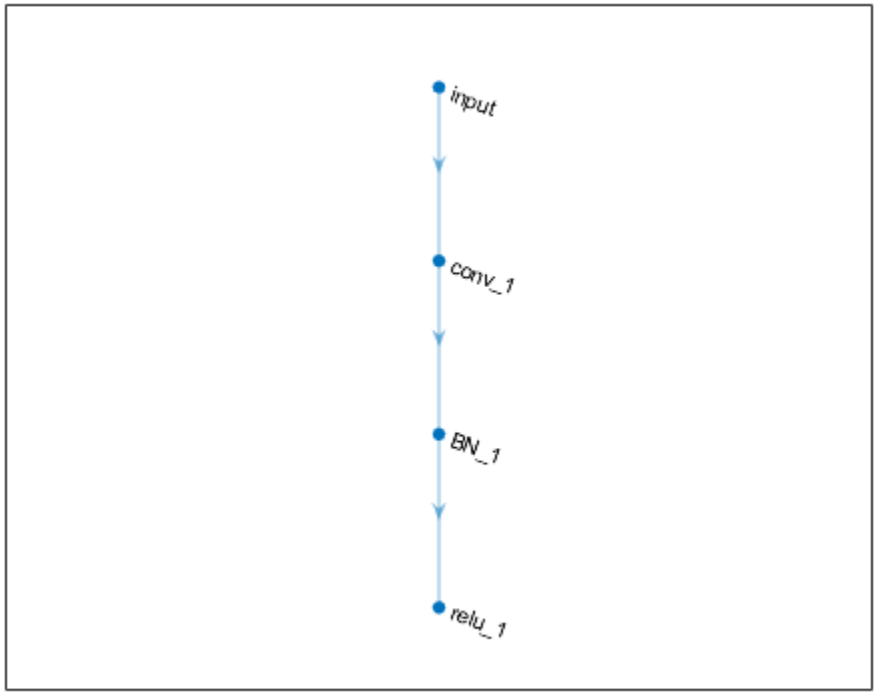
`newlgraph = addLayers(lgraph,larray)` adds the network layers in `larray` to the layer graph `lgraph`. The new layer graph, `newlgraph`, contains the layers and connections of `lgraph` together with the layers in `larray`, connected sequentially. The layer names in `larray` must be unique, nonempty, and different from the names of the layers in `lgraph`.

Examples

Add Layers to Layer Graph

Create an empty layer graph and an array of layers. Add the layers to the layer graph and plot the graph. `addLayers` connects the layers sequentially.

```
lgraph = layerGraph;

layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];

lgraph = addLayers(lgraph,layers);
figure
plot(lgraph)
```



Input Arguments

`lgraph` — Layer graph
`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`larray` — Network layers
`Layer` array

Network layers, specified as a `Layer` array.

For a list of built-in layers, see “List of Deep Learning Layers”.

Output Arguments

`newlgraph` — Output layer graph
`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `removeLayers` | `connectLayers` | `disconnectLayers` | `plot` | `assembleNetwork` | `replaceLayer`

Topics
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b

addParameter

Add parameter to `ONNXParameters` object

Syntax

```
params = addParameter(params,name,value,type)
params = addParameter(params,name,value,type,NumDimensions)
```

Description

`params = addParameter(params,name,value,type)` adds the network parameter specified by `name`, `value`, and `type` to the `ONNXParameters` object `params`. The returned `params` object contains the model parameters of the input argument `params` together with the added parameter, stacked sequentially. The added parameter `name` must be unique, nonempty, and different from the parameter names in `params`.

`params = addParameter(params,name,value,type,NumDimensions)` adds the network parameter specified by `name`, `value`, `type`, and `NumDimensions` to `params`.

Examples

Add Parameters to Imported ONNX Model Function

Import a network saved in the ONNX format as a function and modify the network parameters.

Import the pretrained `simplenet3fc.onnx` network as a function. `simplenet3fc` is a simple convolutional neural network trained on digit image data. For more information on how to create a network similar to `simplenet3fc`, see “Create Simple Image Classification Network”.

Import `simplenet3fc.onnx` using `importONNXFunction`, which returns an `ONNXParameters` object that contains the network parameters. The function also creates a new model function in the current folder that contains the network architecture. Specify the name of the model function as `simplenetFcn`.

```
params = importONNXFunction('simplenet3fc.onnx','simplenetFcn');
```

```
A function containing the imported ONNX network has been saved to the file simplenetFcn.m.
To learn how to use this function, type: help simplenetFcn.
```

Display the parameters that are updated during training (`params.Learnables`) and the parameters that remain unchanged during training (`params.Nonlearnables`).

```
params.Learnables
```

```
ans = struct with fields:
    imageinput_Mean: [1×1 dlarray]
              conv_W: [5×5×1×20 dlarray]
              conv_B: [20×1 dlarray]
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
```

```
            fc_1_W: [24×24×20×20 dlarray]
            fc_1_B: [20×1 dlarray]
            fc_2_W: [1×1×20×20 dlarray]
            fc_2_B: [20×1 dlarray]
            fc_3_W: [1×1×20×10 dlarray]
            fc_3_B: [10×1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
             ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
            ConvStride1007: [2×1 dlarray]
    ConvDilationFactor1008: [2×1 dlarray]
           ConvPadding1009: [4×1 dlarray]
            ConvStride1010: [2×1 dlarray]
    ConvDilationFactor1011: [2×1 dlarray]
           ConvPadding1012: [4×1 dlarray]
            ConvStride1013: [2×1 dlarray]
    ConvDilationFactor1014: [2×1 dlarray]
           ConvPadding1015: [4×1 dlarray]
```

The network has parameters that represent three fully connected layers. You can add a fully connected layer in the original parameters `params` between layers `fc_2` and `fc_3`. The new layer might increase the classification accuracy.

To see the parameters of the convolutional layers `fc_2` and `fc_3`, open the model function `simplenetFcn`.

```
open simplenetFcn
```

Scroll down to the layer definitions in the function `simplenetFcn`. The code below shows the definitions for layers `fc_2` and `fc_3`.

```
% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_2] = prepareConvArgs(Vars
Vars.fc_2 = dlconv(Vars.fc_1, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,

% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_3] = prepareConvArgs(Vars
Vars.fc_3 = dlconv(Vars.fc_2, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,
```

Name the new layer `fc_4`, because each added parameter name must be unique. The `addParameter` function always adds a new parameter sequentially to the `params.Learnables` or `params.Nonlearnables` structure. The order of the layers in the model function `simplenetFcn` determines the order in which the network layers are executed. The names and order of the parameters do not influence the execution order.

Add a new fully connected layer `fc_4` with the same parameters as `fc_2`.

```
params = addParameter(params,'fc_4_W',params.Learnables.fc_2_W,'Learnable');
params = addParameter(params,'fc_4_B',params.Learnables.fc_2_B,'Learnable');
params = addParameter(params,'fc_4_Stride',params.Nonlearnables.ConvStride1010,'Nonlearnable');
params = addParameter(params,'fc_4_DilationFactor',params.Nonlearnables.ConvDilationFactor1011,'N
params = addParameter(params,'fc_4_Padding',params.Nonlearnables.ConvPadding1012,'Nonlearnable')
```

Display the updated learnable and nonlearnable parameters.

```
params.Learnables
```

```
ans = struct with fields:
    imageinput_Mean: [1×1 dlarray]
             conv_W: [5×5×1×20 dlarray]
             conv_B: [20×1 dlarray]
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
             fc_1_W: [24×24×20×20 dlarray]
             fc_1_B: [20×1 dlarray]
             fc_2_W: [1×1×20×20 dlarray]
             fc_2_B: [20×1 dlarray]
             fc_3_W: [1×1×20×10 dlarray]
             fc_3_B: [10×1 dlarray]
             fc_4_W: [1×1×20×20 dlarray]
             fc_4_B: [20×1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
            ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
            ConvStride1007: [2×1 dlarray]
    ConvDilationFactor1008: [2×1 dlarray]
           ConvPadding1009: [4×1 dlarray]
            ConvStride1010: [2×1 dlarray]
    ConvDilationFactor1011: [2×1 dlarray]
           ConvPadding1012: [4×1 dlarray]
            ConvStride1013: [2×1 dlarray]
    ConvDilationFactor1014: [2×1 dlarray]
           ConvPadding1015: [4×1 dlarray]
               fc_4_Stride: [2×1 dlarray]
       fc_4_DilationFactor: [2×1 dlarray]
              fc_4_Padding: [4×1 dlarray]
```

Modify the architecture of the model function to reflect the changes in `params` so you can use the network for prediction with the new parameters or retrain the network. Open the model function `simplenetFcn`. Then, add the fully connected layer `fc_4` between layers `fc_2` and `fc_3`, and change the input data of the convolution operation `dlconv` for layer `fc_3` to `Vars.fc_4`.

```
open simplenetFcn
```

The code below shows the new layer `fc_4` in its position, as well as layers `fc_2` and `fc_3`.

```
% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_2] = prepareConvArgs(Vars
Vars.fc_2 = dlconv(Vars.fc_1, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,

% Conv
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_4] = prepareConvArgs(Vars
Vars.fc_4 = dlconv(Vars.fc_2, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,

% Conv:
```

```
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_3] = prepareConvArgs(Vars
Vars.fc_3 = dlconv(Vars.fc_4, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,
```

Input Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, specified as an `ONNXParameters` object. `params` contains the network parameters of the imported ONNX™ model.

`name` — Name of parameter

character vector | string scalar

Name of the parameter, specified as a character vector or string scalar.

Example: `'conv2_W'`

Example: `'conv2_Padding'`

`value` — Value of parameter

numeric array | character vector | string scalar

Value of the parameter, specified as a numeric array, character vector, or string scalar. To duplicate an existing network layer (stored in `params`), copy the parameter values of the network layer.

Example: `params.Learnables.conv1_W`

Example: `params.Nonlearnables.conv1_Padding`

Data Types: `single` | `double` | `char` | `string`

`type` — Type of parameter

`'Learnable'` | `'Nonlearnable'` | `'State'`

Type of parameter, specified as `'Learnable'`, `'Nonlearnable'`, or `'State'`.

- The value `'Learnable'` specifies a parameter that is updated by the network during training (for example, weights and bias of convolution).
- The value `'Nonlearnable'` specifies a parameter that remains unchanged during network training (for example, padding).
- The value `'State'` specifies a parameter that contains information remembered by the network between iterations and updated across multiple training batches.

Data Types: `char` | `string`

`NumDimensions` — Number of dimensions for every parameter

structure

Number of dimensions for every parameter, specified as a structure. `NumDimensions` includes trailing singleton dimensions.

Example: `params.NumDimensions.conv1_W`

Example: 4

Output Arguments

`params` — Network parameters
`ONNXParameters` object

Network parameters, returned as an `ONNXParameters` object. `params` contains the network parameters updated by `addParameter`.

See Also

`importONNXFunction` | `ONNXParameters` | `removeParameter`

Introduced in R2020b

alexnet

AlexNet convolutional neural network

Syntax

```
net = alexnet
net = alexnet('Weights','imagenet')

layers = alexnet('Weights','none')
```

Description

AlexNet is a convolutional neural network that is 8 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 227-by-227. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the AlexNet network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with AlexNet.

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

`net = alexnet` returns an AlexNet network trained on the ImageNet data set.

This function requires Deep Learning Toolbox Model *for AlexNet Network* support package. If this support package is not installed, the function provides a download link. Alternatively, see Deep Learning Toolbox Model *for AlexNet Network*.

For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

`net = alexnet('Weights','imagenet')` returns an AlexNet network trained on the ImageNet data set. This syntax is equivalent to `net = alexnet`.

`layers = alexnet('Weights','none')` returns the untrained AlexNet network architecture. The untrained model does not require the support package.

Examples

Download AlexNet Support Package

Download and install Deep Learning Toolbox Model *for AlexNet Network* support package.

Type `alexnet` at the command line.

```
alexnet
```

If Deep Learning Toolbox Model *for AlexNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the

support package, click the link, and then click **Install**. Check that the installation is successful by typing `alexnet` at the command line.

```
alexnet
```

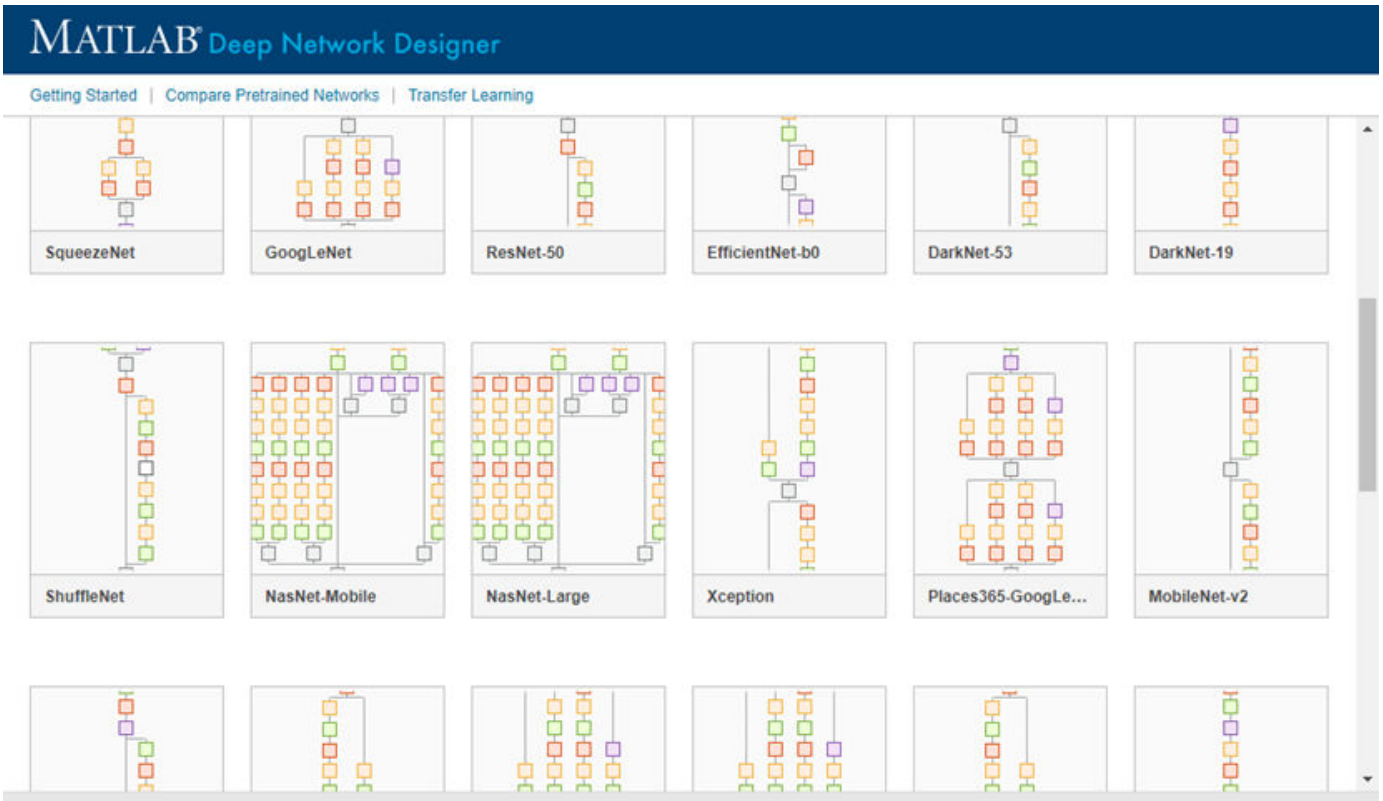
```
ans =

  SeriesNetwork with properties:

    Layers: [25×1 nnet.cnn.layer.Layer]
```

If the required support package is installed, then the function returns a `SeriesNetwork` object.

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(alexnet)
```

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



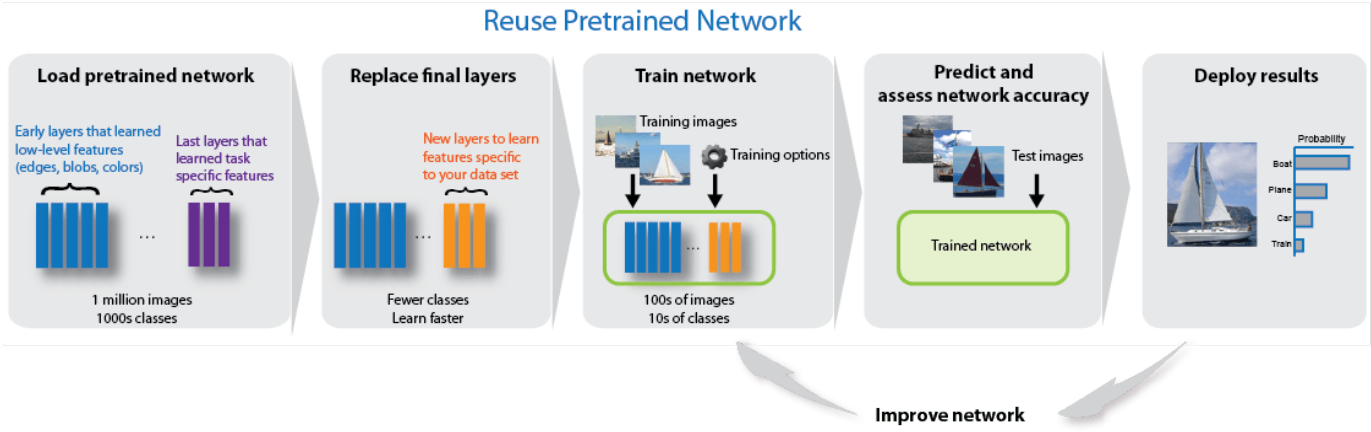
If you need to download a network, pause on the desired network and click **Install** to open the Add-On Explorer.

Transfer Learning Using AlexNet

This example shows how to fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.

AlexNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

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



Load Data

Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

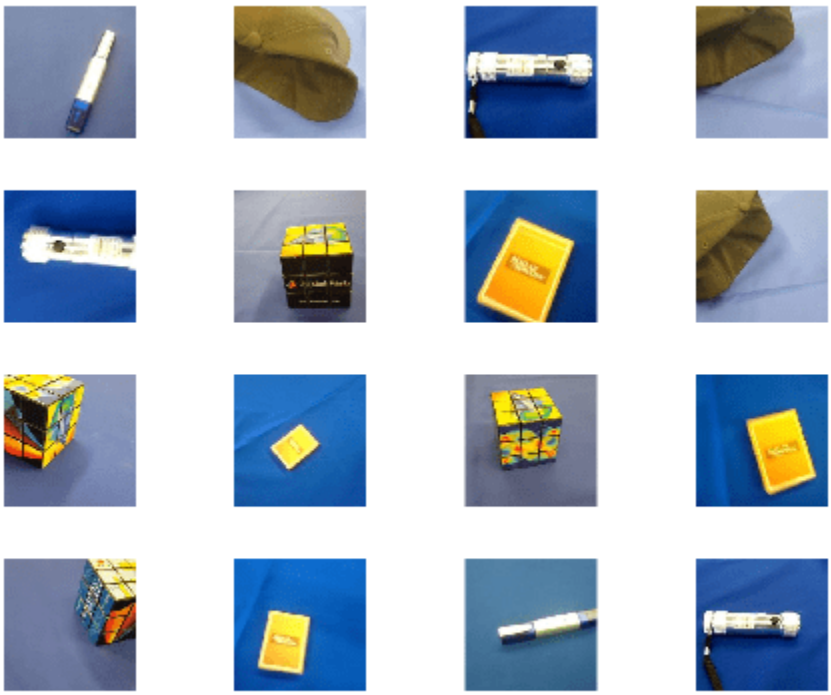
```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

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

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This very small data set now contains 55 training images and 20 validation images. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```



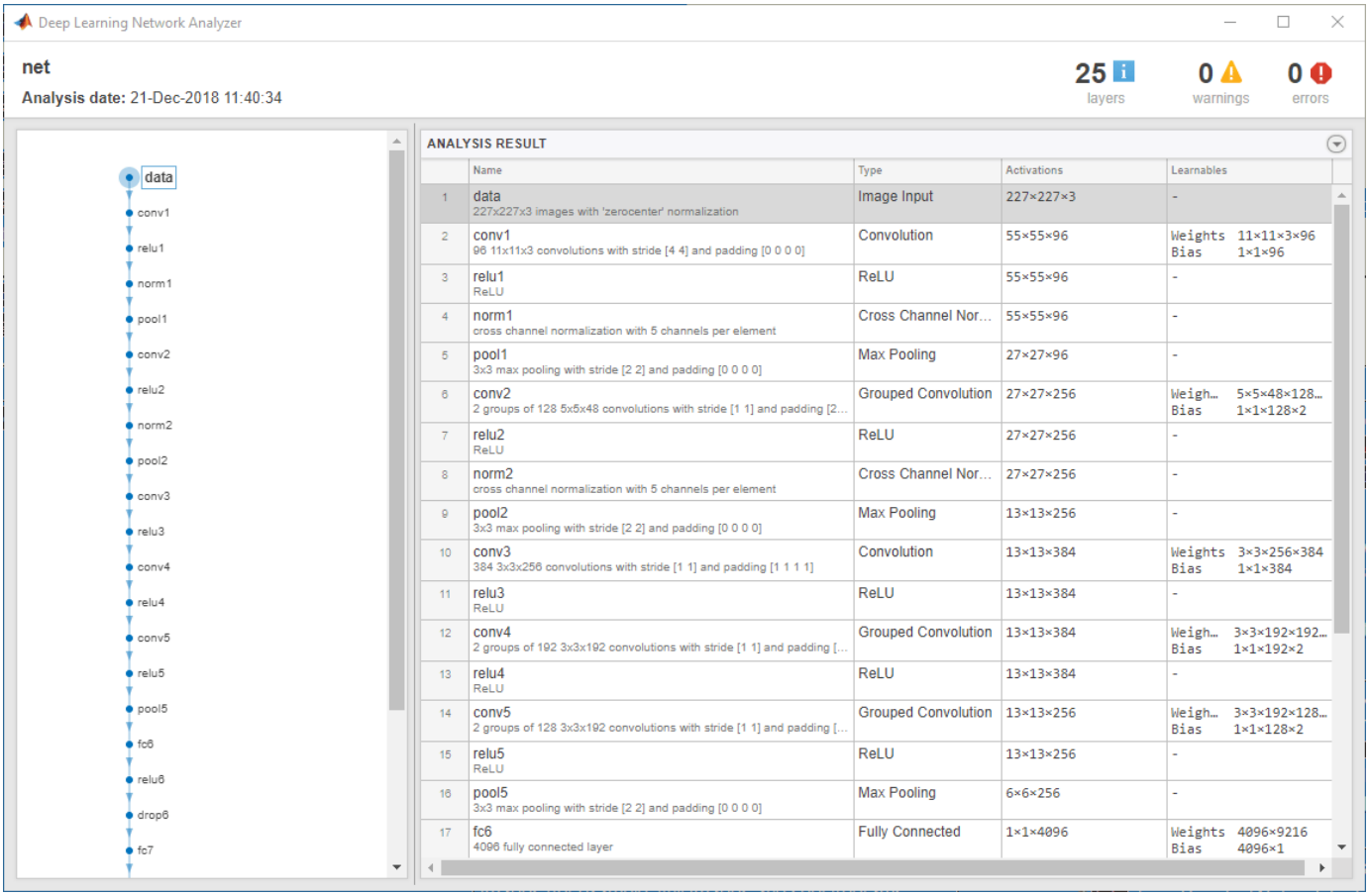
Load Pretrained Network

Load the pretrained AlexNet neural network. If Deep Learning Toolbox™ Model *for AlexNet Network* is not installed, then the software provides a download link. AlexNet is trained on more than one million images and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = alexnet;
```

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
analyzeNetwork(net)
```



The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize = 1×3

   227   227     3
```

Replace Final Layers

The last three layers of the pretrained network `net` are configured for 1000 classes. These three layers must be fine-tuned for the new classification problem. Extract all layers, except the last three, from the pretrained network.

```
layersTransfer = net.Layers(1:end-3);
```

Transfer the layers to the new classification task by replacing the last three layers with a fully connected layer, a softmax layer, and a classification output layer. Specify the options of the new fully connected layer according to the new data. Set the fully connected layer to have the same size as the number of classes in the new data. To learn faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

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

```
numClasses = 5

layers = [
    layersTransfer
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

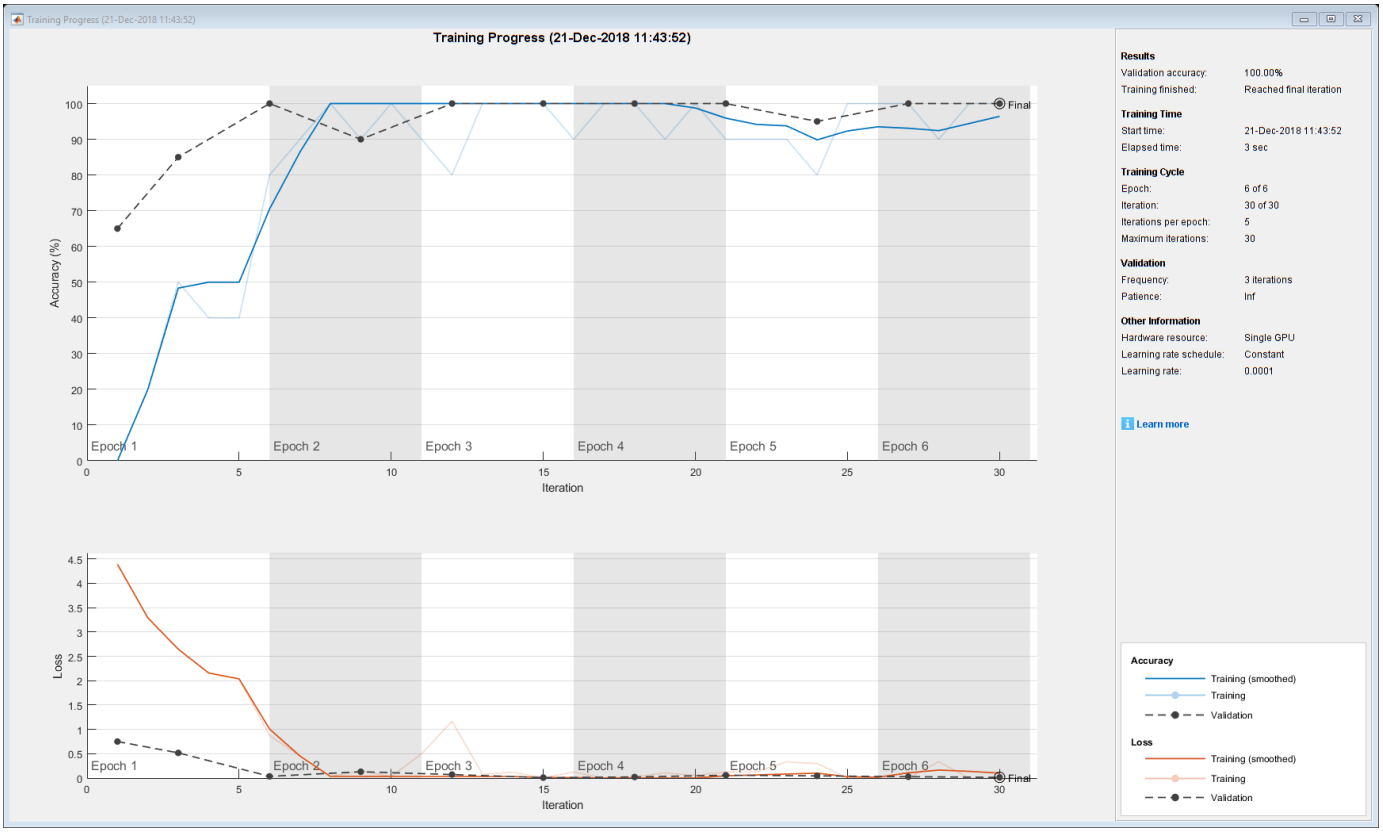
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

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

```
netTransfer = trainNetwork(augimdsTrain,layers,options);
```



Classify Validation Images

Classify the validation images using the fine-tuned network.

```
[YPred,scores] = classify(netTransfer,augimdsValidation);
```

Display four sample validation images with their predicted labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label));
end
```

Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

```
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 1
```

For tips on improving classification accuracy, see “Deep Learning Tips and Tricks”.

Classify an Image Using AlexNet

Read, resize, and classify an image using AlexNet. First, load a pretrained AlexNet model.

```
net = alexnet;
```

Read the image using `imread`.

```
I = imread('peppers.png');
figure
imshow(I)
```

The pretrained model requires the image size to be the same as the input size of the network. Determine the input size of the network using the `InputSize` property of the first layer of the network.

```
sz = net.Layers(1).InputSize
```

```
sz = 1×3

   227   227     3
```

Resize the image to the input size of the network.

```
I = imresize(I,sz(1:2));
figure
imshow(I)
```

Classify the image using `classify`.

```
label = classify(net,I)
```

```
label = categorical
     bell pepper
```

Show the image and classification result together.

```
figure
imshow(I)
title(label)
```

Feature Extraction Using AlexNet

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.

```
numImagesTrain = numel(imdsTrain.Labels);
idx = randperm(numImagesTrain,16);

for i = 1:16
```

```
    I{i} = readimage(imdsTrain,idx(i));
end

figure
imshow(imtile(I))
```

Load Pretrained Network

Load a pretrained AlexNet network. If the Deep Learning Toolbox Model *for AlexNet Network* support package is not installed, then the software provides a download link. AlexNet is trained on more than a million images and can classify images into 1000 object categories. For example,

keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = alexnet;
```

Display the network architecture. The network has five convolutional layers and three fully connected layers.

```
net.Layers
```

```
ans =
  25x1 Layer array with layers:

     1   'data'     Image Input                   227x227x3 images with 'zerocenter' normalizatio
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride [4  4] and
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] and padding
     6   'conv2'    Grouped Convolution           2 groups of 128 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] and padding
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride [1  1] and
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Grouped Convolution           2 groups of 192 3x3x192 convolutions with strid
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Grouped Convolution           2 groups of 128 3x3x192 convolutions with strid
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] and padding
    17   'fc6'      Fully Connected               4096 fully connected layer
    18   'relu6'    ReLU                          ReLU
    19   'drop6'    Dropout                       50% dropout
    20   'fc7'      Fully Connected               4096 fully connected layer
    21   'relu7'    ReLU                          ReLU
    22   'drop7'    Dropout                       50% dropout
    23   'fc8'      Fully Connected               1000 fully connected layer
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 other class
```

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
```

Extract Image Features

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 fully connected layer `'fc7'`. To get a lower-level representation of the images, use an earlier layer in the network.

The network requires input images of size 227-by-227-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);

layer = 'fc7';
featuresTrain = activations(net,augimdsTrain,layer,'OutputAs','rows');
featuresTest = activations(net,augimdsTest,layer,'OutputAs','rows');
```

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).

```
mdl = fitcecoc(featuresTrain,YTrain);
```

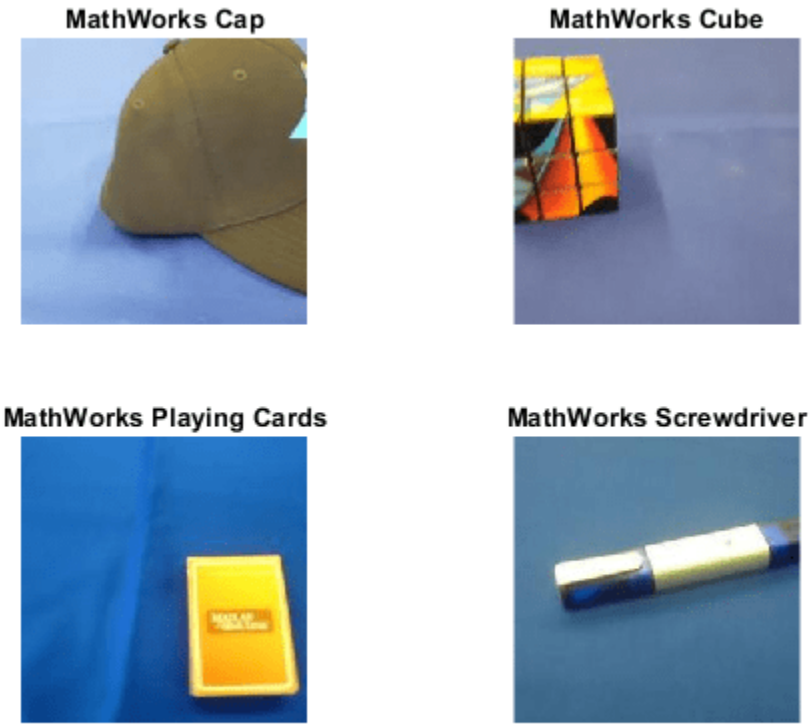
Classify Test Images

Classify the test images using the trained SVM model and the features extracted from the test images.

```
YPred = predict(mdl,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(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
```

This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

Output Arguments

`net` — Pretrained AlexNet convolutional neural network
`SeriesNetwork` object

Pretrained AlexNet convolutional neural network, returned as a `SeriesNetwork` object.

`layers` — Untrained AlexNet convolutional neural network architecture
`Layer` array

Untrained AlexNet convolutional neural network architecture, returned as a `Layer` array.

Tips

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

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] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in neural information processing systems*. 2012.

[4] *BVLC AlexNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = alexnet` or by passing the `alexnet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('alexnet')`.

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `alexnet('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = alexnet` or by passing the `alexnet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('alexnet')`.

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `alexnet('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `resnet18` | `resnet50` | `densenet201` | `googlenet` | `inceptionresnetv2` | `squeezenet` | `importKerasNetwork` | `importCaffeNetwork`

Topics

“Deep Learning in MATLAB”
“Classify Webcam Images Using Deep Learning”
“Pretrained Deep Neural Networks”
“Train Deep Learning Network to Classify New Images”
“Transfer Learning with Deep Network Designer”
“Deep Learning Tips and Tricks”

Introduced in R2017a

analyzeNetwork

Analyze deep learning network architecture

Syntax

```
analyzeNetwork(net)

analyzeNetwork(layers)
analyzeNetwork(layers,'TargetUsage',target)
analyzeNetwork(layers,dlX1,...,dlXn,'TargetUsage','dlnetwork')

analyzeNetwork(dlnet)
analyzeNetwork(dlnet,dlX1,...,dlXn)
```

Description

Use `analyzeNetwork` to visualize and understand the architecture of a network, check that you have defined the architecture correctly, and detect problems before training. Problems that `analyzeNetwork` detects include missing or unconnected layers, incorrectly sized layer inputs, an incorrect number of layer inputs, and invalid graph structures.

Tip To interactively visualize, analyze, and train a network, use `deepNetworkDesigner(net)`. For more information, see **Deep Network Designer**.

Trained Networks

`analyzeNetwork(net)` analyzes the `SeriesNetwork` or `DAGNetwork` object `net`. The function displays an interactive visualization of the network architecture and provides detailed information about the network layers. The layer information includes the number and sizes of layer activations, learnable parameters, and state parameters.

Network Layers

`analyzeNetwork(layers)` analyzes the network layers specified in `layers` and also detects errors and issues for `trainNetwork` workflows. `layers` can be a `Layer` array or a `LayerGraph` object. The function displays an interactive visualization of the network architecture and provides detailed information about the network layers. The layer information includes the number and sizes of layer activations, learnable parameters, and state parameters.

`analyzeNetwork(layers,'TargetUsage',target)` analyzes the network layers specified in `layers` for the specified target workflow. Use this syntax when analyzing a `Layer` array or layer graph for `dlnetwork` workflows.

`analyzeNetwork(layers,dlX1,...,dlXn,'TargetUsage','dlnetwork')` analyzes the network layers using example networks inputs `dlX1,...,dlXn`. The software propagates the example inputs through the network to determine the number and sizes of layer activations, learnable parameters, and state parameters. Use this syntax to analyze a network that has one or more inputs that are not connected to an input layer.

dlnetwork Objects

`analyzeNetwork(dlnet)` analyzes the `dlnetwork` object for custom training loop workflows. The function displays an interactive visualization of the network architecture and provides detailed information about the network layers. The layer information includes the number and sizes of layer activations, learnable parameters, and state parameters.

`analyzeNetwork(dlnet,dlX1,...,dlXn)` analyzes the `dlnetwork` object using example networks inputs `dlX1,...,dlXn`. The software propagates the example inputs through the network to determine the number and sizes of layer activations, learnable parameters, and state parameters. Use this syntax to analyze an uninitialized `dlnetwork` that has one or more inputs that are not connected to an input layer.

Examples

Analyze Trained Network

Load a pretrained GoogLeNet convolutional neural network.

```
net = googlenet
```

```
net =
  DAGNetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Analyze the network. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers.

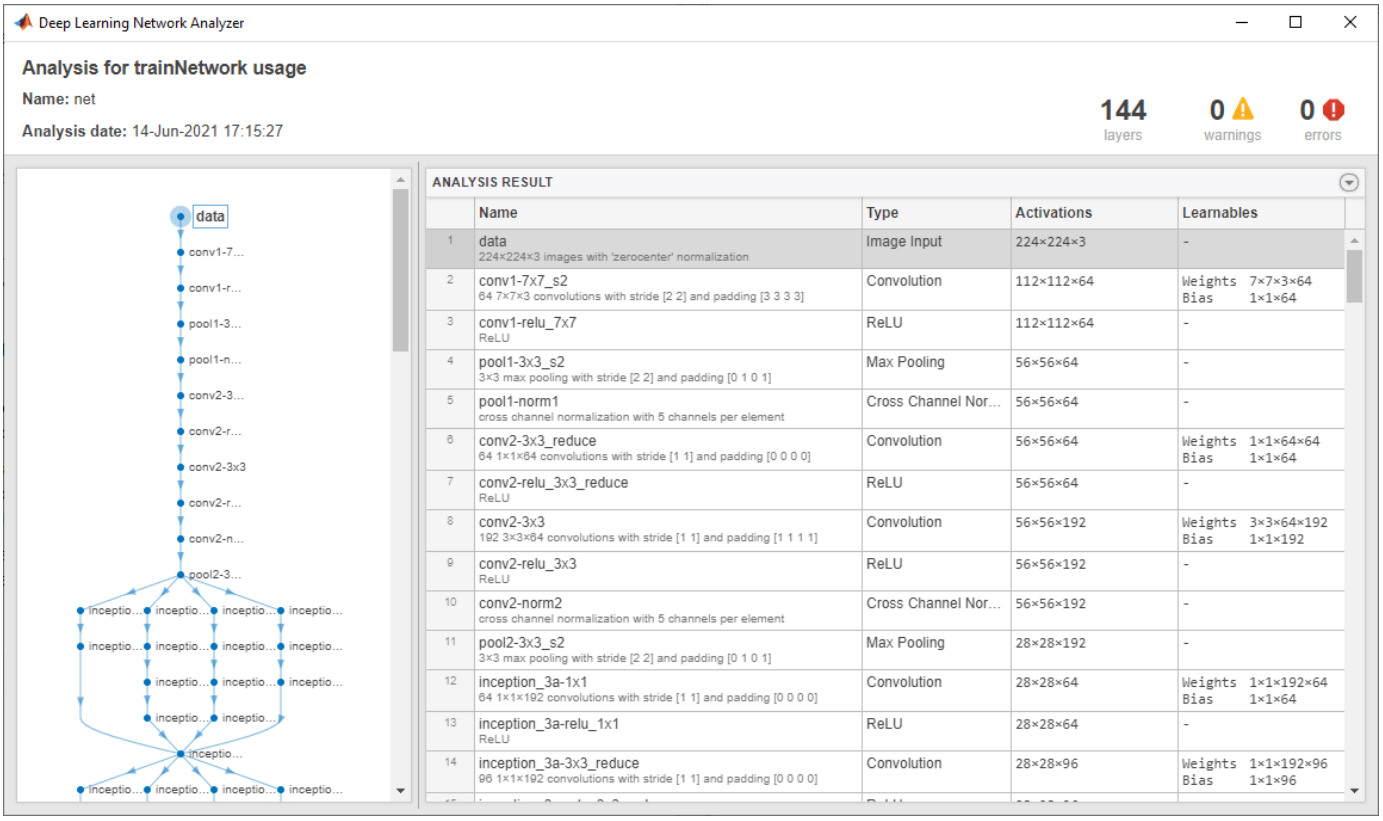
Investigate the network architecture using the plot to the left. Select a layer in the plot. The selected layer is highlighted in the plot and in the layer table.

In the table, view layer information such as layer properties, layer type, and sizes of the layer activations and learnable parameters. The activations of a layer are the outputs of that layer.

Select a deeper layer in the network. Notice that activations in deeper layers are smaller in the spatial dimensions (the first two dimensions) and larger in the channel dimension (the last dimension). Using this structure enables convolutional neural networks to gradually increase the number of extracted image features while decreasing the spatial resolution.

Show the total number of learnable parameters in each layer by clicking the arrow in the top-right corner of the layer table and select **Total Learnables**. To sort the layer table by column value, hover the mouse over the column heading and click the arrow that appears. For example, you can determine which layer contains the most parameters by sorting the layers by the total number of learnable parameters.

```
analyzeNetwork(net)
```



Fix Errors in Network Architecture

Create a simple convolutional network with shortcut connections. Create the main branch of the network as an array of layers and create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially.

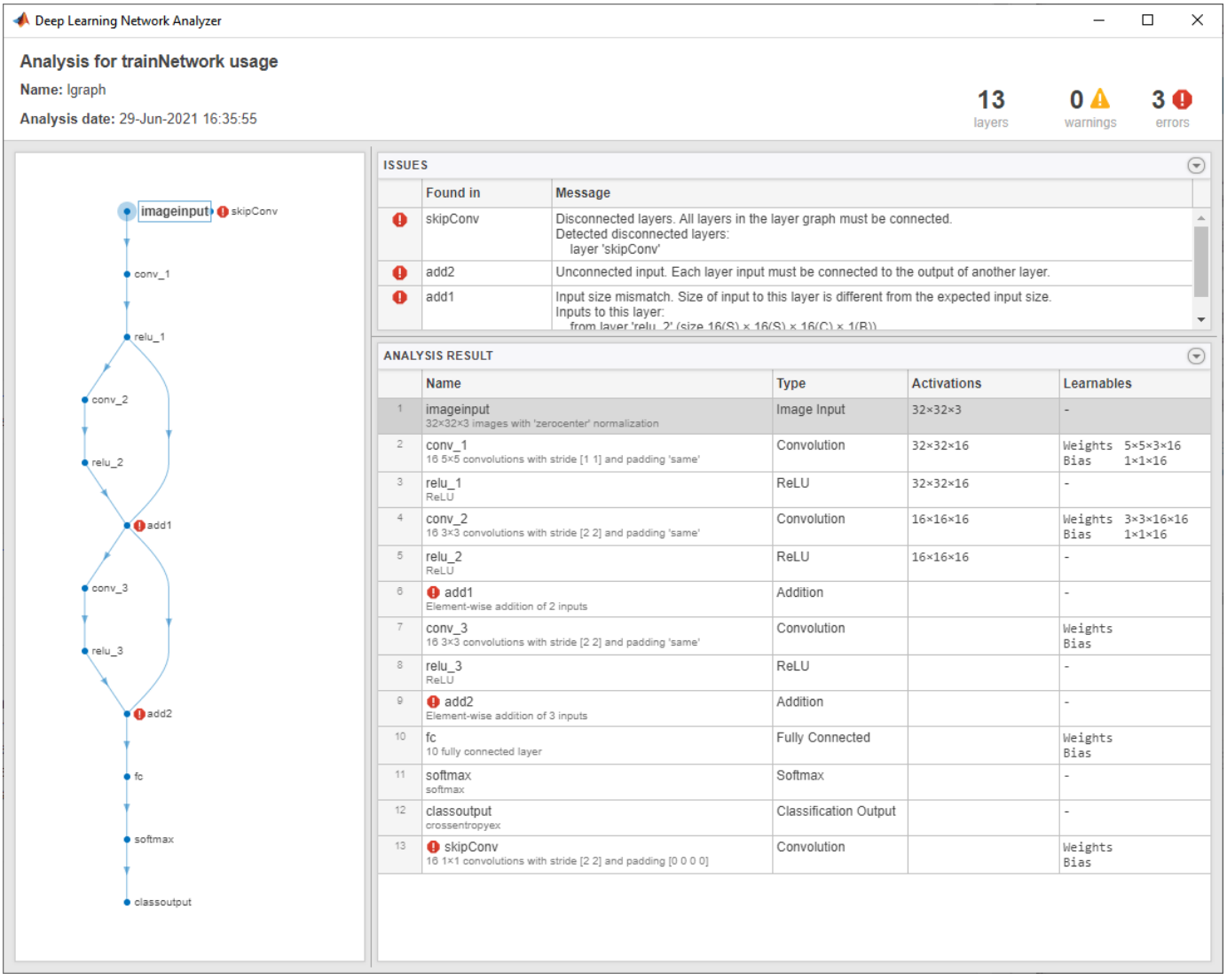
```
layers = [
    imageInputLayer([32 32 3])

    convolution2dLayer(5,16,'Padding','same')
    reluLayer('Name','relu_1')

    convolution2dLayer(3,16,'Padding','same','Stride',2)
    reluLayer
    additionLayer(2,'Name','add1')

    convolution2dLayer(3,16,'Padding','same','Stride',2)
    reluLayer
    additionLayer(3,'Name','add2')

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

lgraph = layerGraph(layers);
```

Create the shortcut connections. One of the shortcut connections contains a single 1-by-1 convolutional layer `skipConv`.

```
skipConv = convolution2dLayer(1,16,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
lgraph = connectLayers(lgraph,'relu_1','add1/in2');
lgraph = connectLayers(lgraph,'add1','add2/in2');
```

Analyze the network architecture. `analyzeNetwork` finds four errors in the network.

```
analyzeNetwork(lgraph)
```



Investigate and fix the errors in the network. In this example, the following issues cause the errors:

- The `skipConv` layer is not connected to the rest of the network. It should be a part of the shortcut connection between the `add1` and `add2` layers. To fix this error, connect `add1` to `skipConv` and `skipConv` to `add2`.
- The `add2` layer is specified to have three inputs, but the layers only has two inputs. To fix the error, specify the number of inputs as `2`.

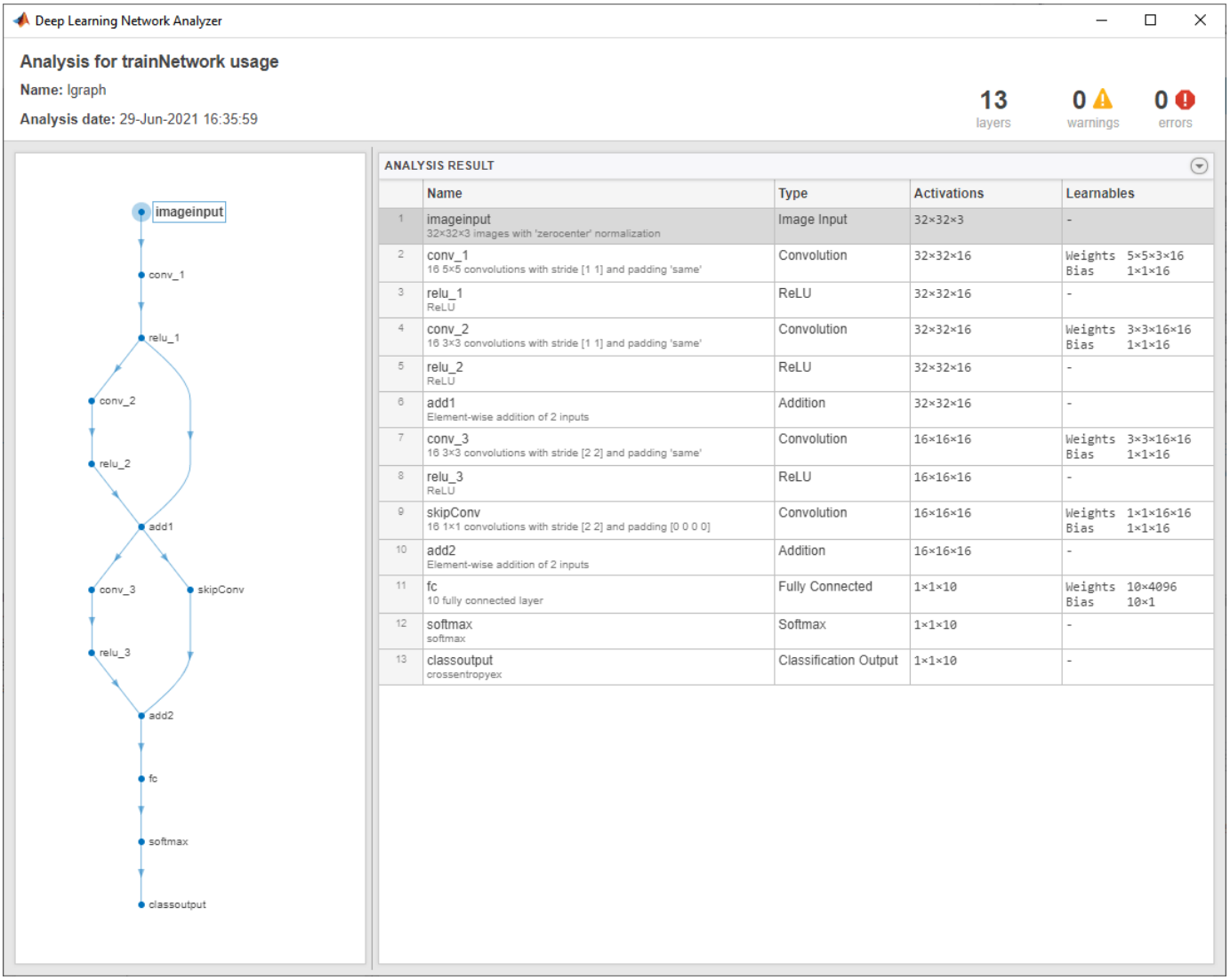
- All the inputs to an addition layer must have the same size, but the `add1` layer has two inputs with different sizes. Because the `conv_2` layer has a `'Stride'` value of 2, this layer downsamples the activations by a factor of two in the first two dimensions (the spatial dimensions). To resize the input from the `relu2` layer so that it has the same size as the input from `relu1`, remove the downsampling by setting the `'Stride'` value of the `conv_2` layer to 1.

Apply these modifications to the layer graph construction from the beginning of this example and create a new layer graph.

```
layers = [
    imageInputLayer([32 32 3])

    convolution2dLayer(5,16,'Padding','same')
    reluLayer('Name','relu_1')

    convolution2dLayer(3,16,'Padding','same','Stride',1)
    reluLayer
    additionLayer(2,'Name','add1')

    convolution2dLayer(3,16,'Padding','same','Stride',2)
    reluLayer
    additionLayer(2,'Name','add2')

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

lgraph = layerGraph(layers);

skipConv = convolution2dLayer(1,16,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
lgraph = connectLayers(lgraph,'relu_1','add1/in2');
lgraph = connectLayers(lgraph,'add1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add2/in2');
```

Analyze the new architecture. The new network does not contain any errors and is ready to be trained.

```
analyzeNetwork(lgraph)
```



Analyze Layer Graph for Custom Training Loop

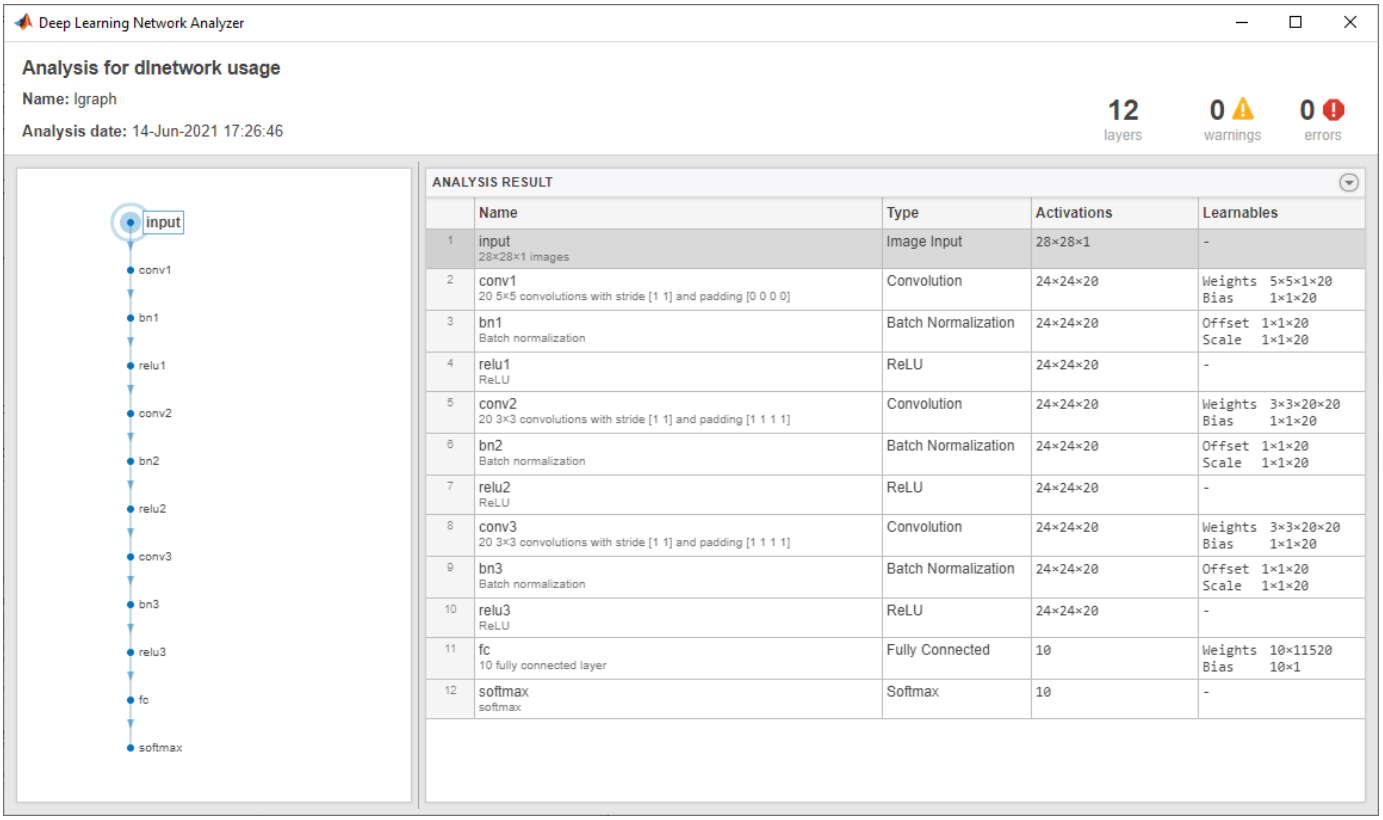
Create a layer graph for a custom training loop. For custom training loop workflows, the layer graph must not have an output layer.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','input')
    convolution2dLayer(5, 20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3, 20,'Padding', 1,'Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
```

```
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')];

lgraph = layerGraph(layers);
```

Analyze the layer graph using the `analyzeNetwork` function and set the `'TargetUsage'` option to `'dlnetwork'`.

```
analyzeNetwork(lgraph,'TargetUsage','dlnetwork')
```



Here, the function does not report any issues with the layer graph.

Analyze Network Using Example Inputs

To analyze a network that has an input that is not connected to an input layer, you can provide example network inputs to the `analyzeNetwork` function. You can provide example inputs when you analyze `dlnetwork` objects, or when you analyze `Layer` arrays or `LayerGraph` objects for custom training workflows using the `'TargetUsage','dlnetwork'` name-value option.

Define the network architecture. Construct a network with two branches. The network takes two inputs, with one input per branch. Connect the branches using an addition layer.

```
numFilters = 24;
inputSize = [64 64 3];
```

```
layersBranch1 = [
    imageInputLayer(inputSize,'Normalization','none','Name','input')
    convolution2dLayer(3,6*numFilters,'Padding','same','Stride',2,'Name','conv1Branch1')
    groupNormalizationLayer('all-channels','Name','gn1Branch1')
    reluLayer('Name','relu1Branch1')
    convolution2dLayer(3,numFilters,'Padding','same','Name','conv2Branch1')
    groupNormalizationLayer('channel-wise','Name','gn2Branch1')
    additionLayer(2,'Name','add')
    reluLayer('Name','reluCombined')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

layersBranch2 = [
    convolution2dLayer(1,numFilters,'Name','convBranch2')
    groupNormalizationLayer('all-channels','Name','gnBranch2')];

lgraph = layerGraph(layersBranch1);
lgraph = addLayers(lgraph,layersBranch2);
lgraph = connectLayers(lgraph,'gnBranch2','add/in2');
```

Create the `dlnetwork`. Because this network contains an unconnected input, create an uninitialized `dlnetwork` object by setting the `'Initialize'` name-value option to `false`.

```
dlnet = dlnetwork(lgraph,'Initialize',false);
```

Create example network inputs of the same size and format as typical inputs for this network. For both inputs, use a batch size of 32. Use an input of size 64-by-64 with three channels for the input to the layer `'input'`. Use an input of size 64-by-64 with 18 channels for the input to the layer `'convBranch2'`.

```
exampleInput = dlarray(rand([inputSize 32]),'SSCB');
exampleConvBranch2 = dlarray(rand([32 32 18 32]),'SSCB');
```

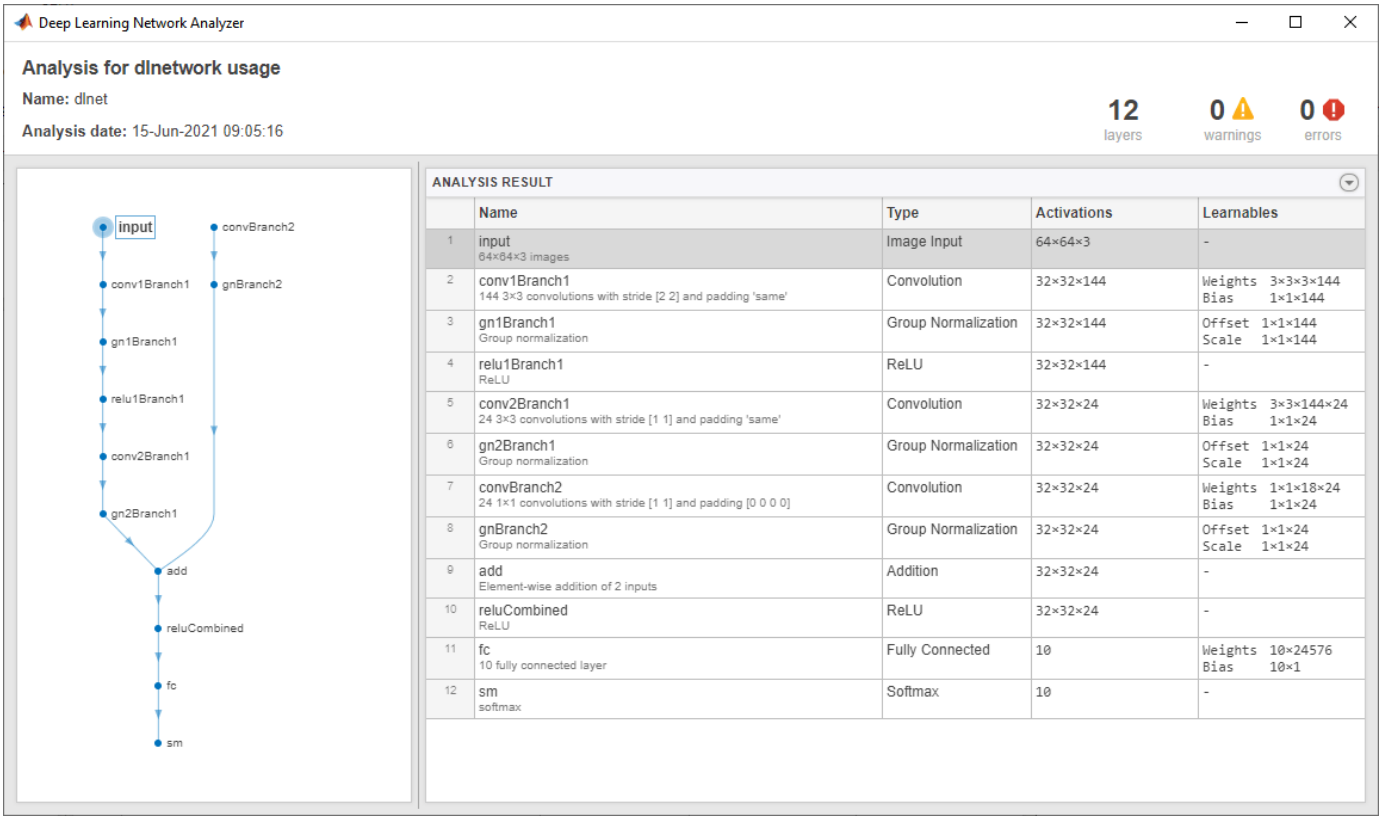
Examine the `Layers` property of the network to determine the order in which to supply the example inputs.

```
dlnet.Layers
```

```
ans =
  12×1 Layer array with layers:

     1   'input'          Image Input           64×64×3 images
     2   'conv1Branch1'   Convolution           144 3×3 convolutions with stride [2  2] and paddi
     3   'gn1Branch1'     Group Normalization   Group normalization
     4   'relu1Branch1'   ReLU                  ReLU
     5   'conv2Branch1'   Convolution           24 3×3 convolutions with stride [1  1] and paddin
     6   'gn2Branch1'     Group Normalization   Group normalization
     7   'add'            Addition              Element-wise addition of 2 inputs
     8   'reluCombined'   ReLU                  ReLU
     9   'fc'             Fully Connected       10 fully connected layer
    10   'sm'             Softmax               softmax
    11   'convBranch2'    Convolution           24 1×1 convolutions with stride [1  1] and paddin
    12   'gnBranch2'      Group Normalization   Group normalization
```

Analyze the network. Provide the example inputs in the same order as the layers that require inputs appear in the `Layers` property of the `dlnetwork`. You must provide an example input for all network inputs, including inputs that are connected to an input layer.

```
analyzeNetwork(dlnet,exampleInput,exampleConvBranch2)
```



Input Arguments

`net` — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

`layers` — Network layers

`Layer` array | `LayerGraph` object

Network layers, specified as a `Layer` array or a `LayerGraph` object.

For a list of built-in layers, see “List of Deep Learning Layers”.

`dlnet` — Network for custom training loops

`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`target` — Target workflow

`'trainNetwork'` (default) | `'dlnetwork'`

Target workflow, specified as one of the following:

- `'trainNetwork'` — Analyze layer graph for usage with the `trainNetwork` function. For example, the function checks that the layer graph has an output layer and no disconnected layer outputs.
- `'dlnetwork'` — Analyze layer graph for usage with `dlnetwork` objects. For example, the function checks that the layer graph does not have any output layers.

`dlX1,...,dlXn` — Example network inputs
`dlarray`

Example network inputs, specified as formatted `dlarray` objects. The software propagates the example inputs through the network to determine the number and sizes of layer activations, learnable parameters, and state parameters.

Use example inputs when you want to analyze a network that has inputs that are unconnected to an input layer.

The order in which you must specify the example inputs depends on the type of network you are analyzing:

- `Layer` array — Provide example inputs in the same order that the layers that require inputs appear in the `Layer` array.
- `LayerGraph` — Provide example inputs in the same order as the layers that require inputs appear in the `Layers` property of the `LayerGraph`.
- `dlnetwork` — Provide example inputs in the same order as the inputs are listed in the `InputNames` property of the `dlnetwork`.

If a layer has multiple unconnected inputs, then example inputs for that layer must be specified separately in the same order as they appear in the layer's `InputNames` property.

You must specify one example input for each input to the network, even if that input is connected to an input layer.

See Also

Deep Network Designer | `SeriesNetwork` | `DAGNetwork` | `LayerGraph` | `trainNetwork` | `plot` | `assembleNetwork`

Topics
"Create Simple Deep Learning Network for Classification"
"Transfer Learning with Deep Network Designer"
"Build Networks with Deep Network Designer"
"Train Deep Learning Network to Classify New Images"
"Pretrained Deep Neural Networks"
"Visualize Activations of a Convolutional Neural Network"
"Deep Learning in MATLAB"

Introduced in R2018a

assembleNetwork

Assemble deep learning network from pretrained layers

Syntax

```
assembledNet = assembleNetwork(layers)
```

Description

`assembleNetwork` creates deep learning networks from layers without training.

Use `assembleNetwork` for the following tasks:

- Convert a layer array or layer graph to a network ready for prediction.
- Assemble networks from imported layers.
- Modify the weights of a trained network.

To train a network from scratch, use `trainNetwork`.

`assembledNet = assembleNetwork(layers)` assembles the layer array or layer graph `layers` into a deep learning network ready to use for prediction.

Examples

Assemble Network from Pretrained Keras Layers

Import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox™. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers` and display their Keras configurations.

```
placeholderLayers = findPlaceholderLayers(lgraph);
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 by saving 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);
```

Specify Class Names

The imported classification layer does not contain the classes, so you must specify these before assembling the network. 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.

The classification layer has the name `'ClassificationLayer_activation_1'`. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer = lgraph.Layers(end);
cLayer.Classes = string(0:9);
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Input Arguments

`layers` — Network layers

`Layer` array | `LayerGraph` object

Network layers, specified as a `Layer` array or a `LayerGraph` object.

To create a network with all layers connected sequentially, you can use a `Layer` array as the input argument. In this case, the returned network is a `SeriesNetwork` object.

A directed acyclic graph (DAG) network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG network, specify the network architecture as a `LayerGraph` object and then use that layer graph as the input argument to `assembleNetwork`.

For a list of built-in layers, see “List of Deep Learning Layers”.

Output Arguments

`assembledNet` — Assembled network
`SeriesNetwork` object | `DAGNetwork` object

Assembled network ready for prediction, returned as a `SeriesNetwork` object or a `DAGNetwork` object. The returned network depends on the `layers` input argument:

- If `layers` is a `Layer` array, then `assembledNet` is a `SeriesNetwork` object.
- If `layers` is a `LayerGraph` object, then `assembledNet` is a `DAGNetwork` object.

See Also

`trainNetwork` | `importKerasNetwork` | `replaceLayer` | `importKerasLayers` | `findPlaceholderLayers` | `functionLayer`

Topics
“Assemble Network from Pretrained Keras Layers”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Define Custom Deep Learning Layers”

Introduced in R2018b

augment

Apply identical random transformations to multiple images

Syntax

```
augI = augment(augmenter,I)
```

Description

`augI = augment(augmenter,I)` augments image `I` using a random transformation from the set of image preprocessing options defined by image data augmenter, `augmenter`. If `I` consists of multiple images, then `augment` applies an identical transformation to all images.

Examples

Augment Image Data with Custom Rotation Range

Create an image augmenter that rotates images by a random angle. To use a custom range of valid rotation angles, you can specify a function handle when you create the augmenter. This example specifies a function called `myrange` (defined at the end of the example) that selects an angle from within two disjoint intervals.

```
imageAugmenter = imageDataAugmenter('RandRotation',@myrange);
```

Read multiple images into the workspace, and display the images.

```
img1 = imread('peppers.png');
img2 = imread('corn.tif',2);
inImg = imtile({img1,img2});
imshow(inImg)
```

Augment the images with identical augmentations. The randomly selected rotation angle is returned in a temporary variable, `angle`.

```
outCellArray = augment(imageAugmenter,{img1,img2});

angle = 8.1158
```

View the augmented images.

```
outImg = imtile(outCellArray);
imshow(outImg);
```

Supporting Function

This example defines the `myrange` function that first randomly selects one of two intervals (-10, 10) and (170, 190) with equal probability. Within the selected interval, the function returns a single random number from a uniform distribution.

```
function angle = myrange()
    if randi([0 1],1)
        a = -10;
        b = 10;
    else
        a = 170;
        b = 190;
    end
    angle = a + (b-a).*rand(1)
end
```

Input Arguments

`augmenter` — Augmentation options

`imageDataAugmenter` object

Augmentation options, specified as an `imageDataAugmenter` object.

`I` — Images to augment

numeric array | cell array of numeric and categorical images

Images to augment, specified as one of the following.

- Numeric array, representing a single grayscale or color image.
- Cell array of numeric and categorical images. Images can be different sizes and types.

Output Arguments

`augI` — Augmented images

numeric array | cell array of numeric and categorical images

Augmented images, returned as a numeric array or cell array of numeric and categorical images, consistent with the format of the input images `I`.

Tips

- You can use the `augment` function to preview the transformations applied to sample images.
- To perform image augmentation during training, create an `augmentedImageDatastore` and specify preprocessing options by using the `'DataAugmentation'` name-value pair with an `imageDataAugmenter`. The augmented image datastore automatically applies random transformations to the training data.

See Also

`augmentedImageDatastore` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2018b

augmentedImageDatastore

Transform batches to augment image data

Description

An augmented image datastore transforms batches of training, validation, test, and prediction data, with optional preprocessing such as resizing, rotation, and reflection. Resize images to make them compatible with the input size of your deep learning network. Augment training image data with randomized preprocessing operations to help prevent the network from overfitting and memorizing the exact details of the training images.

To train a network using augmented images, supply the `augmentedImageDatastore` to `trainNetwork`. For more information, see “Preprocess Images for Deep Learning”.

- 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.
- An `imageInputLayer` normalizes images using the mean of the augmented images, not the mean of the original data set. This mean is calculated once for the first augmented epoch. All other epochs use the same mean, so that the average image does not change during training.

By default, an `augmentedImageDatastore` only resizes images to fit the output size. You can configure options for additional image transformations using an `imageDataAugmenter`.

Creation

Syntax

```
auimds = augmentedImageDatastore(outputSize,imds)
auimds = augmentedImageDatastore(outputSize,X,Y)
auimds = augmentedImageDatastore(outputSize,X)
auimds = augmentedImageDatastore(outputSize,tbl)
auimds = augmentedImageDatastore(outputSize,tbl,responseNames)
auimds = augmentedImageDatastore( ___ ,Name,Value)
```

Description

`auimds = augmentedImageDatastore(outputSize,imds)` creates an augmented image datastore for classification problems using images from image datastore `imds`, and sets the `OutputSize` property.

`auimds = augmentedImageDatastore(outputSize,X,Y)` creates an augmented image datastore for classification and regression problems. The array `X` contains the predictor variables and the array `Y` contains the categorical labels or numeric responses.

`auimds = augmentedImageDatastore(outputSize,X)` creates an augmented image datastore for predicting responses of image data in array `X`.

`auimds = augmentedImageDatastore(outputSize,tbl)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses.

`auimds = augmentedImageDatastore(outputSize,tbl,responseNames)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses. The `responseNames` argument specifies the response variables in `tbl`.

`auimds = augmentedImageDatastore( ___ ,Name,Value)` creates an augmented image datastore, using name-value pairs to set the `ColorPreprocessing`, `DataAugmentation`, `OutputSizeMode`, and `DispatchInBackground` properties. You can specify multiple name-value pairs. Enclose each property name in quotes.

For example,
`augmentedImageDatastore([28,28],myTable,'OutputSizeMode','centercrop')` creates an augmented image datastore that crops images from the center.

Input Arguments

`imds` — Image datastore
`ImageDatastore` object

Image datastore, specified as an `ImageDatastore` object.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

Tip Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.

`X` — Images
4-D numeric array

Images, specified as a 4-D numeric array. The first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

If the array contains `NaN`s, then they are propagated through the training. However, in most cases, the training fails to converge.

Data Types: `single` | `double` | `uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32`

`Y` — Responses for classification or regression
array of categorical responses | numeric matrix | 4-D numeric array

Responses for classification or regression, specified as one of the following:

- For a classification problem, `Y` is a categorical vector containing the image labels.
- For a regression problem, `Y` can be an:
  - *n*-by-*r* numeric matrix. *n* is the number of observations and *r* is the number of responses.

- *h*-by-*w*-by-*c*-by-*n* numeric array. *h*-by-*w*-by-*c* is the size of a single response and *n* is the number of observations.

Responses must not contain `NaN`s.

Data Types: `categorical` | `double`

`tbl` — Input data
`table`

Input data, specified as a table. `tbl` must contain the predictors in the first column as either absolute or relative image paths or images. The type and location of the responses depend on the problem:

- For a classification problem, the response must be a categorical variable containing labels for the images. If the name of the response variable is not specified in the call to `augmentedImageDatastore`, the responses must be in the second column. If the responses are in a different column of `tbl`, then you must specify the response variable name using the `responseNames` argument.
- For a regression problem, the responses must be numerical values in the column or columns after the first one. The responses can be either in multiple columns as scalars or in a single column as numeric vectors or cell arrays containing numeric 3-D arrays. When you do not specify the name of the response variable or variables, `augmentedImageDatastore` accepts the remaining columns of `tbl` as the response variables. You can specify the response variable names using the `responseNames` argument.

Responses must not contain `NaN`s. If there are `NaN`s in the predictor data, they are propagated through the training, however, in most cases the training fails to converge.

Data Types: `table`

`responseNames` — Names of response variables in the input table
character vector | cell array of character vectors | string array

Names of the response variables in the input table, specified as one of the following:

- For classification or regression tasks with a single response, `responseNames` must be a character vector or string scalar containing the response variable in the input table.

  For regression tasks with multiple responses, `responseNames` must be string array or cell array of character vectors containing the response variables in the input table.

Data Types: `char` | `cell` | `string`

Properties

`ColorPreprocessing` — Preprocessing color operations
`'none'` (default) | `'gray2rgb'` | `'rgb2gray'`

Preprocessing color operations performed on input grayscale or RGB images, specified as `'none'`, `'gray2rgb'`, or `'rgb2gray'`. When the image datastore contains a mixture of grayscale and RGB images, use `ColorPreprocessing` to ensure that all output images have the number of channels required by `imageInputLayer`.

No color preprocessing operation is performed when an input image already has the required number of color channels. For example, if you specify the value `'gray2rgb'` and an input image already has three channels, then no color preprocessing occurs.

Note The `augmentedImageDatastore` object converts RGB images to grayscale by using the `rgb2gray` function. If an image has three channels that do not correspond to red, green, and blue channels (such as an image in the L*a*b* color space), then using `ColorPreprocessing` can give poor results.

No color preprocessing operation is performed when the input images do not have 1 or 3 channels, such as for multispectral or hyperspectral images. In this case, all input images must have the same number of channels.

Data Types: `char` | `string`

`DataAugmentation` — Preprocessing applied to input images
`'none'` (default) | `imageDataAugmenter` object

Preprocessing applied to input images, specified as an `imageDataAugmenter` object or `'none'`. When `DataAugmentation` is `'none'`, no preprocessing is applied to input images.

`DispatchInBackground` — Dispatch observations in background
`false` (default) | `true`

Dispatch observations in the background during training, prediction, or classification, specified as `false` or `true`. To use background dispatching, you must have Parallel Computing Toolbox.

Augmented image datastores only perform background dispatching when used with `trainNetwork` and inference functions such as `predict` and `classify`. Background dispatching does not occur when you call the `read` function of the datastore directly.

`MiniBatchSize` — Number of observations in each batch
128 | positive integer

Number of observations that are returned in each batch. You can change the value of `MiniBatchSize` only after you create the datastore. For training, prediction, and classification, the `MiniBatchSize` property is set to the mini-batch size defined in `trainingOptions`.

`NumObservations` — Total number of observations in the datastore
positive integer

This property is read-only.

Total number of observations in the augmented image datastore. The number of observations is the length of one training epoch.

`OutputSize` — Size of output images
vector of two positive integers

Size of output images, specified as a vector of two positive integers. The first element specifies the number of rows in the output images, and the second element specifies the number of columns.

Note If you create an `augmentedImageDatastore` by specifying the image output size as a three-element vector, then the datastore ignores the third element. Instead, the datastore uses the value of

`ColorPreprocessing` to determine the dimensionality of output images. For example, if you specify `OutputSize` as `[28 28 1]` but set `ColorPreprocessing` as `'gray2rgb'`, then the output images have size 28-by-28-by-3.

`OutputSizeMode` — Method used to resize output images

`'resize'` (default) | `'centercrop'` | `'randcrop'`

Method used to resize output images, specified as one of the following.

- `'resize'` — Scale the image using bilinear interpolation to fit the output size.

  Note `augmentedImageDatastore` uses the bilinear interpolation method of `imresize` with antialiasing. Bilinear interpolation enables fast image processing while avoiding distortions such as caused by nearest-neighbor interpolation. In contrast, by default `imresize` uses bicubic interpolation with antialiasing to produce a high-quality resized image at the cost of longer processing time.

- `'centercrop'` — Take a crop from the center of the training image. The crop has the same size as the output size.
- `'randcrop'` — Take a random crop from the training image. The random crop has the same size as the output size.

Data Types: `char` | `string`

Object Functions

combine	Combine data from multiple datastores
hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
partitionByIndex	Partition augmentedImageDatastore according to indices
preview	Preview subset of data in datastore
read	Read data from augmentedImageDatastore
readall	Read all data in datastore
readByIndex	Read data specified by index from augmentedImageDatastore
reset	Reset datastore to initial state
shuffle	Shuffle data in augmentedImageDatastore
subset	Create subset of datastore or file-set
transform	Transform datastore
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Examples

Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

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

`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 width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

```
idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,:,idx);
XTrain(:,:,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Create an `imageDataAugmenter` object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

```
imageAugmenter = imageDataAugmenter( ...
    'RandRotation',[-20,20], ...
    'RandXTranslation',[-3 3], ...
    'RandYTranslation',[-3 3])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-20 20]
           RandScale: [1 1]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [-3 3]
    RandYTranslation: [-3 3]
```

Create an `augmentedImageDatastore` object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

Specify the convolutional neural network architecture.

```
layers = [
    imageInputLayer(imageSize)

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
```

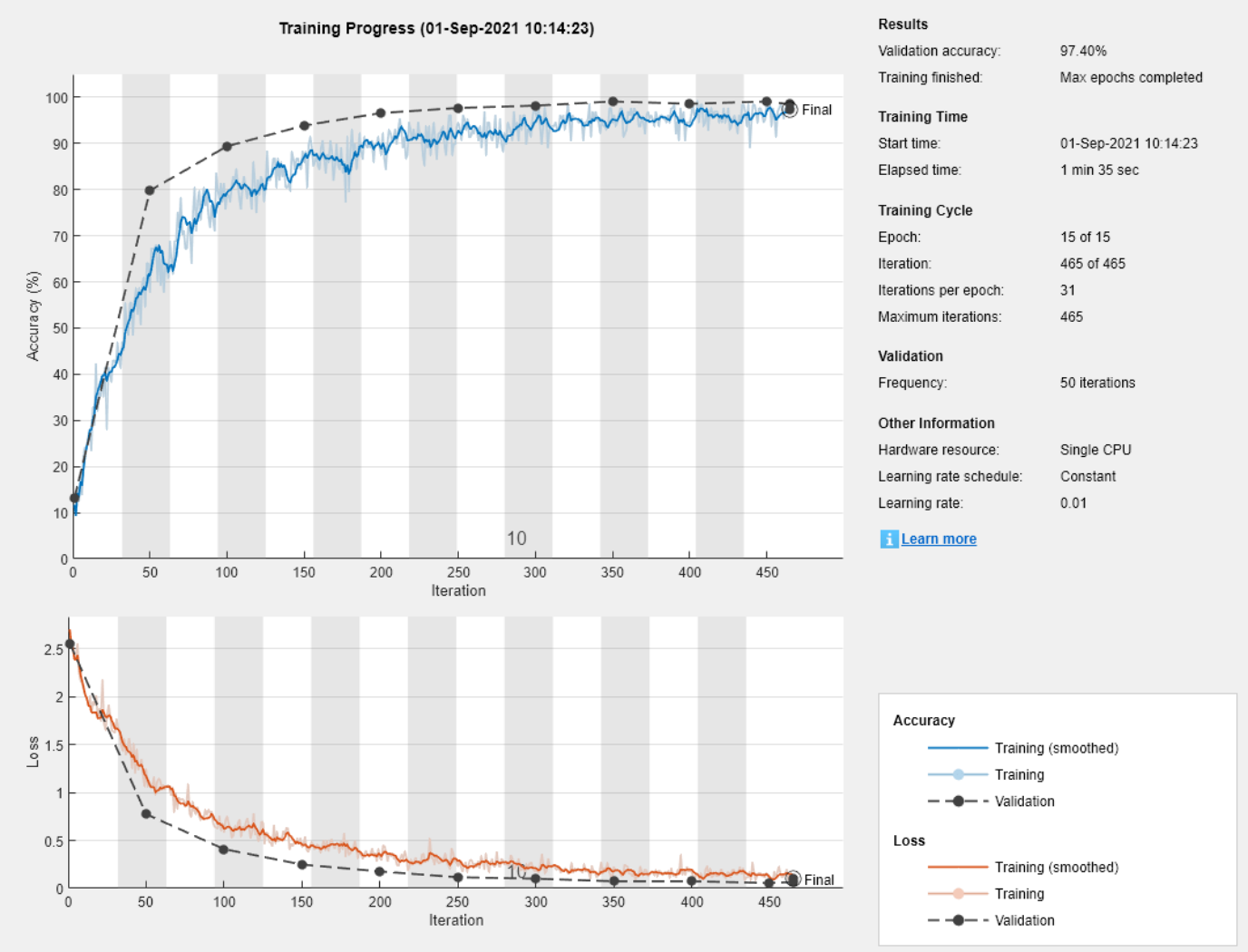
```
    reluLayer

    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 training options for stochastic gradient descent with momentum.

```
opts = trainingOptions('sgdm', ...
    'MaxEpochs',15, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation});
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```
net = trainNetwork(augimds,layers,opts);
```



Tips

- You can visualize many transformed images in the same figure by using the `imtile` function. For example, this code displays one mini-batch of transformed images from an augmented image datastore called `auimds`.

  ```
  minibatch = read(auimds);
  imshow(imtile(minibatch.input))
  ```

- By default, resizing is the only image preprocessing operation performed on images. Enable additional preprocessing operations by using the `DataAugmentation` name-value pair argument with an `imageDataAugmenter` object. Each time images are read from the augmented image datastore, a different random combination of preprocessing operations are applied to each image.

See Also

`imageDataAugmenter` | `imageInputLayer` | `trainNetwork`

Topics
“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2018a

augmentedImageSource

(To be removed) Generate batches of augmented image data

Note `augmentedImageSource` will be removed in a future release. Create an augmented image datastore using the `augmentedImageDatastore` function instead. For more information, see "Compatibility Considerations".

Syntax

```
auimds = augmentedImageSource(outputSize,imds)
auimds = augmentedImageSource(outputSize,X,Y)
auimds = augmentedImageSource(outputSize,tbl)
auimds = augmentedImageSource(outputSize,tbl,responseNames)
auimds = augmentedImageSource( ___ ,Name,Value)
```

Description

`auimds = augmentedImageSource(outputSize,imds)` creates an augmented image datastore, `auimds`, for classification problems using images from image datastore `imds`, with output image size `outputSize`.

`auimds = augmentedImageSource(outputSize,X,Y)` creates an augmented image datastore for classification and regression problems. The array `X` contains the predictor variables and the array `Y` contains the categorical labels or numeric responses.

`auimds = augmentedImageSource(outputSize,tbl)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses.

`auimds = augmentedImageSource(outputSize,tbl,responseNames)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses. The `responseNames` argument specifies the response variable in `tbl`.

`auimds = augmentedImageSource( ___ ,Name,Value)` creates an augmented image datastore, using name-value pairs to configure the image preprocessing done by the augmented image datastore. You can specify multiple name-value pairs.

Examples

Train Network with Rotational Invariance Using `augmentedImageSource`

Preprocess images using random rotation so that the trained convolutional neural network has rotational invariance. This example uses the `augmentedImageSource` function to create an augmented image datastore object. For an example of the recommended workflow that uses the `augmentedImageDatastore` function to create an augmented image datastore object, see "Train Network with Augmented Images" on page 1-140.

Load the sample data, which consists of synthetic images of handwritten numbers.

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

`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 width of the images.
- 1 is the number of channels
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` is a categorical vector containing the labels for each observation.

Create an image augmenter that rotates images during training. This image augmenter rotates each image by a random angle.

```
imageAugmenter = imageDataAugmenter('RandRotation',[-180 180])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-180 180]
           RandScale: [1 1]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [0 0]
    RandYTranslation: [0 0]
```

Use the `augmentedImageSource` function to create an augmented image datastore. Specify the size of augmented images, the training data, and the image augmenter.

```
imageSize = [28 28 1];
auimds = augmentedImageSource(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter)
```

```
auimds =
  augmentedImageDatastore with properties:

         NumObservations: 5000
           MiniBatchSize: 128
        DataAugmentation: [1x1 imageDataAugmenter]
      ColorPreprocessing: 'none'
              OutputSize: [28 28]
          OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

Specify the convolutional neural network architecture.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)
```

```
    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,64,'Padding',1)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the training options for stochastic gradient descent with momentum.

```
opts = trainingOptions('sgdm', ...
    'MaxEpochs',10, ...
    'Shuffle','every-epoch', ...
    'InitialLearnRate',1e-3);
```

Train the network.

```
net = trainNetwork(auimds,layers,opts);
```

```
Training on single CPU.
Initializing image normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:01 |        7.81% |       2.4151 |          0.0010 |
|       2 |          50 |       00:00:23 |       52.34% |       1.4930 |          0.0010 |
|       3 |         100 |       00:00:44 |       74.22% |       1.0148 |          0.0010 |
|       4 |         150 |       00:01:05 |       78.13% |       0.8153 |          0.0010 |
|       6 |         200 |       00:01:26 |       76.56% |       0.6903 |          0.0010 |
|       7 |         250 |       00:01:45 |       87.50% |       0.4891 |          0.0010 |
|       8 |         300 |       00:02:06 |       87.50% |       0.4874 |          0.0010 |
|       9 |         350 |       00:02:30 |       87.50% |       0.4866 |          0.0010 |
|      10 |         390 |       00:02:46 |       89.06% |       0.4021 |          0.0010 |
|=========================================================================================|
```

Input Arguments

`outputSize` — Size of output images

vector of two positive integers

Size of output images, specified as a vector of two positive integers. The first element specifies the number of rows in the output images, and the second element specifies the number of columns. This value sets the `OutputSize on page 1-0` property of the returned augmented image datastore, `auimds`.

`imds` — Image datastore

`ImageDatastore` object

Image datastore, specified as an `ImageDatastore` object.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

Tip Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.

X — Images

4-D numeric array

Images, specified as a 4-D numeric array. The first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

If the array contains `NaN`s, then they are propagated through the training. However, in most cases, the training fails to converge.

Data Types: `single` | `double` | `uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32`

Y — Responses for classification or regression

array of categorical responses | numeric matrix | 4-D numeric array

Responses for classification or regression, specified as one of the following:

- For a classification problem, `Y` is a categorical vector containing the image labels.
- For a regression problem, `Y` can be an:
  - *n*-by-*r* numeric matrix. *n* is the number of observations and *r* is the number of responses.
  - *h*-by-*w*-by-*c*-by-*n* numeric array. *h*-by-*w*-by-*c* is the size of a single response and *n* is the number of observations.

Responses must not contain `NaN`s.

Data Types: `categorical` | `double`

tbl — Input data

`table`

Input data, specified as a table. `tbl` must contain the predictors in the first column as either absolute or relative image paths or images. The type and location of the responses depend on the problem:

- For a classification problem, the response must be a categorical variable containing labels for the images. If the name of the response variable is not specified in the call to `augmentedImageSource`, the responses must be in the second column. If the responses are in a different column of `tbl`, then you must specify the response variable name using the `responseNames` argument.
- For a regression problem, the responses must be numerical values in the column or columns after the first one. The responses can be either in multiple columns as scalars or in a single column as numeric vectors or cell arrays containing numeric 3-D arrays. When you do not specify the name of the response variable or variables, `augmentedImageSource` accepts the remaining columns of `tbl` as the response variables. You can specify the response variable names using the `responseNames` argument.

Responses must not contain `NaN`s. If there are `NaN`s in the predictor data, they are propagated through the training, however, in most cases the training fails to converge.

Data Types: `table`

`responseNames` — Names of response variables in the input table
character vector | cell array of character vectors | string array

Names of the response variables in the input table, specified as one of the following:

- For classification or regression tasks with a single response, `responseNames` must be a character vector or string scalar containing the response variable in the input table.

  For regression tasks with multiple responses, `responseNames` must be string array or cell array of character vectors containing the response variables in the input table.

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `augmentedImageSource([28,28],myTable,'OutputSizeMode','centercrop')` creates an augmented image datastore that sets the `OutputSizeMode` property to crop images from the center.

`ColorPreprocessing` — Preprocessing color operations
`'none'` (default) | `'gray2rgb'` | `'rgb2gray'`

Preprocessing operations performed on color channels of input images, specified as the comma-separated pair consisting of `'ColorPreprocessing'` and `'none'`, `'gray2rgb'`, or `'rgb2gray'`. This argument sets the `ColorPreprocessing on page 1-0` property of the returned augmented image datastore, `auimds`. The `ColorPreprocessing` property ensures that all output images from the augmented image datastore have the number of color channels required by `inputImageLayer`.

`DataAugmentation` — Preprocessing applied to input images
`'none'` (default) | `imageDataAugmenter` object

Preprocessing applied to input images, specified as the comma-separated pair consisting of `'DataAugmentation'` and an `imageDataAugmenter` object or `'none'`. This argument sets the `DataAugmentation on page 1-0` property of the returned augmented image datastore, `auimds`. When `DataAugmentation` is `'none'`, no preprocessing is applied to input images.

`OutputSizeMode` — Method used to resize output images
`'resize'` (default) | `'centercrop'` | `'randcrop'`

Method used to resize output images, specified as the comma-separated pair consisting of `'OutputSizeMode'` and one of the following. This argument sets the `OutputSizeMode on page 1-0` property of the returned augmented image datastore, `auimds`.

- `'resize'` — Scale the image to fit the output size. For more information, see `imresize`.
- `'centercrop'` — Take a crop from the center of the training image. The crop has the same size as the output size.
- `'randcrop'` — Take a random crop from the training image. The random crop has the same size as the output size.

Data Types: `char` | `string`

`BackgroundExecution` — Perform augmentation in parallel
`false` (default) | `true`

Perform augmentation in parallel, specified as the comma-separated pair consisting of `'BackgroundExecution'` and `false` or `true`. This argument sets the `DispatchInBackground` on page 1-0 property of the returned augmented image datastore, `auimds`. If `'BackgroundExecution'` is `true`, and you have Parallel Computing Toolbox software installed, then the augmented image datastore `auimds` performs image augmentation in parallel.

Output Arguments

`auimds` — Augmented image datastore
`augmentedImageDatastore` object

Augmented image datastore, returned as an `augmentedImageDatastore` object.

Compatibility Considerations

`augmentedImageSource` object is removed

In R2017b, you could create an `augmentedImageSource` object to preprocess images for training deep learning networks. Starting in R2018a, the `augmentedImageSource` object has been removed. Use an `augmentedImageDatastore` object instead.

An `augmentedImageDatastore` has additional properties and methods to assist with data preprocessing. Unlike `augmentedImageSource`, which could be used for training only, you can use an `augmentedImageDatastore` for both training and prediction.

To create an `augmentedImageDatastore` object, you can use either the `augmentedImageDatastore` function (recommended) or the `augmentedImageSource` function.

`augmentedImageSource` function will be removed
Not recommended starting in R2018a

The `augmentedImageSource` function will be removed in a future release. Create an `augmentedImageDatastore` using the `augmentedImageDatastore` function instead.

To update your code, change instances of the function name `augmentedImageSource` to `augmentedImageDatastore`. You do not need to change the input arguments.

See Also
`augmentedImageDatastore`

Introduced in R2017b

averagePooling1dLayer

1-D average pooling layer

Description

A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Creation

Syntax

```
layer = averagePooling1dLayer(poolSize)
layer = averagePooling1dLayer(poolSize,Name=Value)
```

Description

`layer = averagePooling1dLayer(poolSize)` creates a 1-D average pooling layer and sets the `PoolSize` property.

`layer = averagePooling1dLayer(poolSize,Name=Value)` also specifies the padding or sets the `Stride` and Name properties using one or more optional name-value arguments. For example, `averagePooling1dLayer(3,Padding=1,Stride=2)` creates a 1-D average pooling layer with a pool size of `3`, a stride of `2`, and padding of size `1` on both the left and right of the input.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `averagePooling1dLayer(2,Padding=1)` creates a 1-D max pooling layer with a pool size of 3 and padding of size 1 on the left and right of the layer input.

`Padding` — Padding to apply to input
`[0 0]` (default) | `"same"` | nonnegative integer | vector of nonnegative integers

Padding to apply to the input, specified as one of the following:

- `"same"` — Apply padding such that the output size is `ceil(inputSize/stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same size as the input.
- Nonnegative integer `sz` — Add padding of size `sz` to both ends of the input.
- Vector `[l r]` of nonnegative integers — Add padding of size `l` to the left and `r` to the right of the input.

Example: `Padding=[2 1]` adds padding of size 2 to the left and size 1 to the right.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Properties

Average Pooling

`PoolSize` — Width of pooling regions

positive integer

Width of the pooling regions, specified as a positive integer.

The width of the pooling regions `PoolSize` must be greater than or equal to the padding dimensions `PaddingSize`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Stride` — Step size for traversing input

`1` (default) | positive integer

Step size for traversing the input, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PaddingSize` — Size of padding

`[0 0]` (default) | vector of two nonnegative integers

Size of padding to apply to each side of the input, specified as a vector `[l r]` of two nonnegative integers, where `l` is the padding applied to the left and `r` is the padding applied to the right.

When you create a layer, use the `Padding` name-value argument to specify the padding size.

Data Types: `double`

`PaddingMode` — Method to determine padding size

`'manual'` (default) | `'same'`

This property is read-only.

Method to determine padding size, specified as one of the following:

- `'manual'` – Pad using the integer or vector specified by `Padding`.
- `'same'` – Apply padding such that the output size is `ceil(inputSize/Stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same as the input.

To specify the layer padding, use the `Padding` name-value argument.

Data Types: `char`

`PaddingValue` — Value used to pad input
`0` (default) | `"mean"`

Value used to pad input, specified as `0` or `"mean"`.

When you use the `Padding` option to add padding to the input, the value of the padding applied can be one of the following:

- `0` — Input is padded with zeros at the positions specified by the `Padding` property. The padded areas are included in the calculation of the average value of the pooling regions along the edges.
- `"mean"` — Input is padded with the mean of the pooling region at the positions specified by the `Padding` option. The padded areas are effectively excluded from the calculation of the average value of each pooling region.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 1-D Average Pooling Layer

Create a 1-D average pooling layer with a pool size of 3.

```
layer = averagePooling1dLayer(3)
```

```
layer =
  AveragePooling1DLayer with properties:

           Name: ''

   Hyperparameters
        PoolSize: 3
          Stride: 1
     PaddingMode: 'manual'
     PaddingSize: [0 0]
    PaddingValue: 0
```

Include a 1-D average pooling layer in a layer array.

```
layers = [
    sequenceInputLayer(12)
    convolution1dLayer(11,96)
    reluLayer
    averagePooling1dLayer(3)
    convolution1dLayer(11,96)
    reluLayer
    globalMaxPooling1dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  10x1 Layer array with layers:

     1   ''   Sequence Input           Sequence input with 12 dimensions
     2   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
     3   ''   ReLU                     ReLU
     4   ''   1-D Average Pooling      average pooling with pool size 3, stride 1, and padding [0
     5   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
     6   ''   ReLU                     ReLU
     7   ''   1-D Global Max Pooling   1-D global max pooling
     8   ''   Fully Connected          10 fully connected layer
     9   ''   Softmax                  softmax
    10   ''   Classification Output    crossentropyex
```

Algorithms

1-D Average Pooling Layer

A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region. The layer pools the input by moving the pooling regions along the input horizontally.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Topics

“Sequence Classification Using 1-D Convolutions”
“Sequence-to-Sequence Classification Using 1-D Convolutions”
“Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2021b

averagePooling2dLayer

Average pooling layer

Description

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

Creation

Syntax

```
layer = averagePooling2dLayer(poolSize)
layer = averagePooling2dLayer(poolSize,Name,Value)
```

Description

`layer = averagePooling2dLayer(poolSize)` creates an average pooling layer and sets the `PoolSize` property.

`layer = averagePooling2dLayer(poolSize,Name,Value)` sets the optional `Stride` and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `averagePooling2dLayer(2,'Stride',2)` creates an average pooling layer with pool size `[2 2]` and stride `[2 2]`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the `Stride` and `Name` properties. Enclose names in single quotes.

Example: `averagePooling2dLayer(2,'Stride',2)` creates an average pooling layer with pool size `[2 2]` and stride `[2 2]`.

`Padding` — Input edge padding
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If

the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Vector `[a b]` of nonnegative integers — Add padding of size `a` to the top and bottom of the input and padding of size `b` to the left and right.
- Vector `[t b l r]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, and `r` to the right of the input.

Example: `'Padding',1` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Average Pooling

`PoolSize` — Dimensions of pooling regions

vector of two positive integers

Dimensions of the pooling regions, specified as a vector of two positive integers `[h w]`, where `h` is the height and `w` is the width. When creating the layer, you can specify `PoolSize` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 1]` specifies pooling regions of height 2 and width 1.

`Stride` — Step size for traversing input

`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector of two positive integers `[a b]`, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`PaddingSize` — Size of padding

`[0 0 0 0]` (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector `[t b l r]` of four nonnegative integers, where `t` is the padding applied to the top, `b` is the padding applied to the bottom, `l` is the padding applied to the left, and `r` is the padding applied to the right.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 1 2 2]` adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

PaddingMode — Method to determine padding size

`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the `'Padding'` value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

PaddingValue — Value used to pad input

`0` (default) | `"mean"`

Value used to pad input, specified as `0` or `"mean"`.

When you use the `Padding` option to add padding to the input, the value of the padding applied can be one of the following:

- `0` — Input is padded with zeros at the positions specified by the `Padding` property. The padded areas are included in the calculation of the average value of the pooling regions along the edges.
- `"mean"` — Input is padded with the mean of the pooling region at the positions specified by the `Padding` option. The padded areas are effectively excluded from the calculation of the average value of each pooling region.

Padding — Size of padding

`[0 0]` (default) | vector of two nonnegative integers

Note `Padding` property will be removed in a future release. Use `PaddingSize` instead. When creating a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector `[a b]` of two nonnegative integers, where `a` is the padding applied to the top and bottom of the input data and `b` is the padding applied to the left and right.

Example: `[1 1]` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Layer

Name — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Average Pooling Layer

Create an average pooling layer with the name `'avg1'`.

```
layer = averagePooling2dLayer(2,'Name','avg1')
```

```
layer =
  AveragePooling2DLayer with properties:

           Name: 'avg1'

   Hyperparameters
        PoolSize: [2 2]
```

```
          Stride: [1 1]
     PaddingMode: 'manual'
     PaddingSize: [0 0 0 0]
    PaddingValue: 0
```

Include an average pooling layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    averagePooling2dLayer(2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Average Pooling         2x2 average pooling with stride [1  1] and padding [0  0  0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Average Pooling Layer with Nonoverlapping Pooling Regions

Create an average pooling layer with nonoverlapping pooling regions.

```
layer = averagePooling2dLayer(2,'Stride',2)
```

```
layer =
  AveragePooling2DLayer with properties:

           Name: ''

   Hyperparameters
        PoolSize: [2 2]
          Stride: [2 2]
     PaddingMode: 'manual'
     PaddingSize: [0 0 0 0]
    PaddingValue: 0
```

The height and width of the rectangular regions (pool size) are both 2. The pooling regions do not overlap because the step size for traversing the images vertically and horizontally (stride) is also 2.

Include an average pooling layer with nonoverlapping regions in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
```

```
    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

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  0
     3   ''   ReLU                    ReLU
     4   ''   Average Pooling         2x2 average pooling with stride [2  2] and padding [0  0  0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Average Pooling Layer with Overlapping Pooling Regions

Create an average pooling layer with overlapping pooling regions.

```
layer = averagePooling2dLayer([3 2],'Stride',2)
```

```
layer =
  AveragePooling2DLayer with properties:

           Name: ''

   Hyperparameters
        PoolSize: [3 2]
          Stride: [2 2]
     PaddingMode: 'manual'
     PaddingSize: [0 0 0 0]
    PaddingValue: 0
```

This layer creates pooling regions of size [3 2] and takes the average of the six elements in each region. The pooling regions overlap because `Stride` includes dimensions that are less than the respective pooling dimensions `PoolSize`.

Include an average pooling layer with overlapping pooling regions in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    averagePooling2dLayer([3 2],'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

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  0
     3   ''   ReLU                    ReLU
```

```
4   ''   Average Pooling         3x2 average pooling with stride [2  2] and padding [0  0  0
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex
```

More About

Average Pooling Layer

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

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.

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* [1]. 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.

References

[1] 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`convolution2dLayer` | `globalAveragePooling2dLayer` | `maxPooling2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”

“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

averagePooling3dLayer

3-D average pooling layer

Description

A 3-D average pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the average values of each region.

Creation

Syntax

```
layer = averagePooling3dLayer(poolSize)
layer = averagePooling3dLayer(poolSize,Name,Value)
```

Description

`layer = averagePooling3dLayer(poolSize)` creates an average pooling layer and sets the `PoolSize` property.

`layer = averagePooling3dLayer(poolSize,Name,Value)` sets the optional `Stride` and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `averagePooling3dLayer(2,'Stride',2)` creates a 3-D average pooling layer with pool size `[2 2 2]` and stride `[2 2 2]`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the `Stride` and `Name` properties. Enclose names in single quotes.

Example: `averagePooling3dLayer(2,'Stride',2)` creates a 3-D average pooling layer with pool size `[2 2 2]` and stride `[2 2 2]`.

`Padding` — Input edge padding
`0` (default) | array of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra

padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Three-element vector `[a b c]` of nonnegative integers — Add padding of size `a` to the top and bottom, padding of size `b` to the left and right, and padding of size `c` to the front and back of the input.
- 2-by-3 matrix `[t l f;b r k]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, `r` to the right, `f` to the front, and `k` to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: `'Padding',1` adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Average Pooling

`PoolSize` — Dimensions of pooling regions

vector of three positive integers

Dimensions of the pooling regions, specified as a vector of three positive integers `[h w d]`, where `h` is the height, `w` is the width, and `d` is the depth. When creating the layer, you can specify `PoolSize` as a scalar to use the same value for all three dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 1 1]` specifies pooling regions of height 2, width 1, and depth 1.

`Stride` — Step size for traversing input

`[1 1 1]` (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector `[a b c]` of three positive integers, where `a` is the vertical step size, `b` is the horizontal step size, and `c` is the step size along the depth direction. When creating the layer, you can specify `Stride` as a scalar to use the same value for step sizes in all three directions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 3 1]` specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

`PaddingSize` — Size of padding

`[0 0 0;0 0 0]` (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix `[t l f;b r k]` of nonnegative integers, where `t` and `b` are the padding applied to the top and bottom in the vertical direction, `l` and `r` are the padding applied to the left and right in the horizontal direction, and `f` and `k` are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 2 4;1 2 4]` adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the 'Padding' value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

`PaddingValue` — Value used to pad input
`0` (default) | `"mean"`

Value used to pad input, specified as `0` or `"mean"`.

When you use the `Padding` option to add padding to the input, the value of the padding applied can be one of the following:

- `0` — Input is padded with zeros at the positions specified by the `Padding` property. The padded areas are included in the calculation of the average value of the pooling regions along the edges.
- `"mean"` — Input is padded with the mean of the pooling region at the positions specified by the `Padding` option. The padded areas are effectively excluded from the calculation of the average value of each pooling region.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 3-D Average Pooling Layer

Create a 3-D average pooling layer with nonoverlapping pooling regions that downsamples by a factor of 2.

```
layer = averagePooling3dLayer(2,'Stride',2)
```

```
layer =
  AveragePooling3DLayer with properties:

           Name: ''

   Hyperparameters
        PoolSize: [2 2 2]
          Stride: [2 2 2]
     PaddingMode: 'manual'
     PaddingSize: [2x3 double]
    PaddingValue: 0
```

Include a 3-D average pooling layer in a `Layer` array.

```
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    averagePooling3dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input         28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             20 5x5x5 convolutions with stride [1  1  1] and padding [0
     3   ''   ReLU                    ReLU
     4   ''   Average 3D Pooling      2x2x2 average pooling with stride [2  2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create 3-D Average Pooling Layer with Overlapping Pooling Regions

Create a 3-D average pooling layer with overlapping pooling regions and padding for the top and bottom of the input.

```
layer = averagePooling3dLayer([3 2 2],'Stride',2,'Padding',[1 0 0])
```

```
layer =
  AveragePooling3DLayer with properties:

           Name: ''

   Hyperparameters
        PoolSize: [3 2 2]
          Stride: [2 2 2]
     PaddingMode: 'manual'
     PaddingSize: [2x3 double]
    PaddingValue: 0
```

This layer creates pooling regions of size 3-by-2-by-2 and takes the average of the twelve elements in each region. The stride is 2 in all dimensions. The pooling regions overlap because there are stride dimensions `Stride` that are less than the respective pooling dimensions `PoolSize`.

More About

3-D Average Pooling Layer

A 3-D average pooling layer extends the functionality of an average pooling layer to a third dimension, depth. An average pooling layer performs down-sampling by dividing the input into rectangular or cuboidal pooling regions, and computing the average of each region. To learn more,

see the definition of average pooling layer on page 1-162 on the `averagePooling2dLayer` reference page.

See Also

`averagePooling2dLayer` | `convolution3dLayer` | `maxPooling3dLayer`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a

avgpool

Pool data to average values over spatial dimensions

Syntax

```
dlY = avgpool(dlX,poolsize)
dlY = avgpool(dlX,'global')
dlY = avgpool( ___ ,'DataFormat',FMT)
dlY = avgpool( ___ ,Name,Value)
```

Description

The average pooling operation performs downsampling by dividing the input into pooling regions and computing the average value of each region.

The `avgpool` function applies the average pooling operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply average pooling within a `layerGraph` object or `Layer` array, use one of the following layers:

- `averagePooling2dLayer`
- `averagePooling3dLayer`
- `globalAveragePooling2dLayer`
- `globalAveragePooling3dLayer`

`dlY = avgpool(dlX,poolsize)` applies the average pooling operation to the formatted `dlarray` object `dlX`. The function downsamples the input by dividing it into regions defined by `poolsize` and calculating the average value of the data in each region. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`.

The function, by default, pools over up to three dimensions of `dlX` labeled `'S'` (spatial). To pool over dimensions labeled `'T'` (time), specify a pooling region with a `'T'` dimension using the `'PoolFormat'` option.

`dlY = avgpool(dlX,'global')` computes the global average over the spatial dimensions of the input `dlX`. This syntax is equivalent to setting `poolsize` in the previous syntax to the size of the `'S'` dimensions of `dlX`.

`dlY = avgpool( ___ ,'DataFormat',FMT)` applies the average pooling operation to the unformatted `dlarray` object `dlX` with format specified by `FMT` using any of the previous syntaxes. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For

example, `'DataFormat','SSCB'` specifies data for 2-D average pooling with format `'SSCB'` (spatial, spatial, channel, batch).

`dlY = avgpool( ___ ,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'PoolFormat','T'` specifies a pooling region for 1-D pooling with format `'T'` (time).

Examples

Perform 2-D Average Pooling

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Apply 2-D average pooling with 2-by-2 pooling regions using the `avgpool` function.

```
poolSize = [2 2];
dlY = avgpool(dlX,poolSize);
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

    27    27     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Perform 2-D Global Average Pooling

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Apply 2-D global average pooling using the `avgpool` function by specifying the `'global'` option.

```
dlY = avgpool(dlX,'global');
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

     1     1     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Perform 1-D Average Pooling

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 12 channels. Specify the format `'CBT'` (channel, batch, time).

```
miniBatchSize = 128;
sequenceLength = 100;
numChannels = 12;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×3

    12   128   100
```

```
dims(dlX)

ans =
'CBT'
```

Apply 1-D average pooling with pooling regions of size 2 with a stride of 2 using the `avgpool` function by specifying the `'PoolFormat'` and `'Stride'` options.

```
poolSize = 2;
dlY = avgpool(dlX,poolSize,'PoolFormat','T','Stride',2);
```

View the size and format of the output.

```
size(dlY)

ans = 1×3

    12   128    50
```

```
dims(dlY)

ans =
'CBT'
```

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted or unformatted `dlarray` object.

If `dlX` is an unformatted `dlarray`, then you must specify the format using the `'DataFormat'` option.

The function, by default, pools over up to three dimensions of `dlX` labeled `'S'` (spatial). To pool over dimensions labeled `'T'` (time), specify a pooling region with a `'T'` dimension using the `'PoolFormat'` option.

`poolsize` — Size of pooling regions

positive integer | vector of positive integers

Size of the pooling regions, specified as a numeric scalar or numeric vector.

To pool using a pooling region with edges of the same size, specify `poolsize` as a scalar. The pooling regions have the same size along all dimensions specified by `'PoolFormat'`.

To pool using a pooling region with edges of different sizes, specify `poolsize` as a vector, where `poolsize(i)` is the size of corresponding dimension in `'PoolFormat'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Stride',2` specifies the stride of the pooling regions as 2.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`PoolFormat` — Dimension order of pooling region
character vector | string scalar

Dimension order of the pooling region, specified as the comma-separated pair consisting of `'PoolFormat'` and a character vector or string scalar that provides a label for each dimension of the pooling region.

The default value of `'PoolFormat'` depends on the task:

Task	Default
1-D pooling	`'S'` (spatial)
2-D pooling	`'SS'` (spatial, spatial)
3-D pooling	`'SSS'` (spatial, spatial, spatial)

The format must have either no `'S'` (spatial) dimensions, or as many `'S'` (spatial) dimensions as the input data.

The function, by default, pools over up to three dimensions of `dlX` labeled `'S'` (spatial). To pool over dimensions labeled `'T'` (time), specify a pooling region with a `'T'` dimension using the `'PoolFormat'` option.

Example: `'PoolFormat','T'`

`Stride` — Step size for traversing input data
`1` (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of `'Stride'` and a numeric scalar or numeric vector. If you specify `'Stride'` as a scalar, the same value is used for all spatial dimensions. If you specify `'Stride'` as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of `'Stride'` is `1`. If `'Stride'` is less than `poolsize` in any dimension, then the pooling regions overlap.

The `Stride` parameter is not supported for global pooling using the `'global'` option.

Example: `'Stride',3`

Data Types: `single` | `double`

`Padding` — Size of padding applied to edges of data
`0` (default) | `'same'` | numeric scalar | numeric vector | numeric matrix

Size of padding applied to edges of data, specified as the comma-separated pair consisting of `'Padding'` and one of the following:

- `'same'` — Padding size is set so that the output size is the same as the input size when the stride is `1`. More generally, the output size of each spatial dimension is `ceil(inputSize/stride)`, where `inputSize` is the size of the input along a spatial dimension.
- Numeric scalar — The same amount of padding is applied to both ends of all spatial dimensions.
- Numeric vector — A different amount of padding is applied along each spatial dimension. Use a vector of size `d`, where `d` is the number of spatial dimensions of the input data. The `i`th element of the vector specifies the size of padding applied to the start and the end along the `i`th spatial dimension.
- Numeric matrix — A different amount of padding is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-`d`, where `d` is the number of spatial dimensions of the input data. The element `(1,d)` specifies the size of padding applied to the start of spatial dimension `d`. The element `(2,d)` specifies the size of padding applied to the end of spatial dimension `d`. For example, in 2-D, the format is `[top, left; bottom, right]`.

The `'Padding'` parameter is not supported for global pooling using the `'global'` option.

Example: `'Padding','same'`

Data Types: `single` | `double`

`PaddingValue` — Value used to pad input
`0` (default) | `"mean"`

Value used to pad input, specified as `0` or `"mean"`.

When you use the `Padding` option to add padding to the input, the value of the padding applied can be one of the following:

- `0` — Input is padded with zeros at the positions specified by the `Padding` property. The padded areas are included in the calculation of the average value of the pooling regions along the edges.

- `"mean"` — Input is padded with the mean of the pooling region at the positions specified by the `Padding` option. The padded areas are effectively excluded from the calculation of the average value of each pooling region.

Output Arguments

`dlY` — Pooled data

`dlarray`

Pooled data, returned as a `dlarray` with the same underlying data type as `dlX`.

If the input data `dlX` is a formatted `dlarray`, then `dlY` has the same format as `dlX`. If the input data is not a formatted `dlarray`, then `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

More About

Average Pooling

The `avgpool` function pools the input data to average values. For more information, see the definition of "Average Pooling Layer" on page 1-162 on the `averagePooling2dLayer` reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`dlconv` | `maxpool` | `dlarray` | `dlgradient` | `dlfeval`

Topics

"Define Custom Training Loops, Loss Functions, and Networks"
"Train Network Using Model Function"
"List of Functions with dlarray Support"

Introduced in R2019b

batchnorm

Normalize data across all observations for each channel independently

Syntax

```
dlY = batchnorm(dlX,offset,scaleFactor)
[dlY,popMu,popSigmaSq] = batchnorm(dlX,offset,scaleFactor)
[dlY,updatedMu,updatedSigmaSq] = batchnorm(dlX,offset,scaleFactor,runningMu,
runningSigmaSq)
dlY = batchnorm(dlX,offset,scaleFactor,trainedMu,trainedSigmaSq)
[___] = batchnorm(___,'DataFormat',FMT)
[___] = batchnorm(___,Name,Value)
```

Description

The batch normalization operation normalizes the input data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization between convolution and nonlinear operations such as `relu`.

After normalization, the operation shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

The `batchnorm` function applies the batch normalization operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply batch normalization within a `layerGraph` object or `Layer` array, use `batchNormalizationLayer`.

`dlY = batchnorm(dlX,offset,scaleFactor)` applies the batch normalization operation to the input data `dlX` using the population mean and variance of the input data and the specified offset and scale factor.

The function normalizes over the `'S'` (spatial), `'T'` (time), `'B'` (batch), and `'U'` (unspecified) dimensions of `dlX` for each channel in the `'C'` (channel) dimension, independently.

For unformatted input data, use the `'DataFormat'` option.

`[dlY,popMu,popSigmaSq] = batchnorm(dlX,offset,scaleFactor)` applies the batch normalization operation and also returns the population mean and variance of the input data `dlX`.

`[dlY,updatedMu,updatedSigmaSq] = batchnorm(dlX,offset,scaleFactor,runningMu, runningSigmaSq)` applies the batch normalization operation and also returns the updated moving mean and variance statistics. `runningMu` and `runningSigmaSq` are the mean and variance values after the previous training iteration, respectively.

Use this syntax to maintain running values for the mean and variance statistics during training. When you have finished training, use the final updated values of the mean and variance for the batch normalization operation during prediction and classification.

`dlY = batchnorm(dlX,offset,scaleFactor,trainedMu,trainedSigmaSq)` applies the batch normalization operation using the mean `trainedMu` and variance `trainedSigmaSq`.

Use this syntax during classification and prediction, where `trainedMu` and `trainedSigmaSq` are the final values of the mean and variance after you have finished training, respectively.

`[___] = batchnorm(___,'DataFormat',FMT)` applies the batch normalization operation to unformatted input data with format specified by `FMT` using any of the input or output combinations in previous syntaxes. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D image input with the format `'SSCB'` (spatial, spatial, channel, batch).

`[___] = batchnorm(___,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'MeanDecay',0.3` sets the decay rate of the moving average computation.

Examples

Apply Batch Normalization

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Initialize the scale and offset for batch normalization. For the scale, specify a vector of ones. For the offset, specify a vector of zeros.

```
scaleFactor = ones(numChannels,1);
offset = zeros(numChannels,1);
```

Apply the batch normalization operation using the `batchnorm` function and return the mini-batch statistics.

```
[dlY,mu,sigmaSq] = batchnorm(dlX,offset,scaleFactor);
```

View the size and format of the output `dlY`.

```
size(dlY)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

View the mini-batch mean `mu`.

```
mu
```

```
mu = 3×1

    0.4998
    0.4993
    0.5011
```

View the mini-batch variance `sigmaSq`.

```
sigmaSq
```

```
sigmaSq = 3×1

    0.0831
    0.0832
    0.0835
```

Update Mean and Variance over Multiple Batches of Data

Use the `batchnorm` function to normalize several batches of data and update the statistics of the whole data set after each normalization.

Create three batches of data. The data consists of 10-by-10 random arrays with five channels. Each batch contains 20 observations. The second and third batches are scaled by a multiplicative factor of `1.5` and `2.5`, respectively, so the mean of the data set increases with each batch.

```
height = 10;
width = 10;
numChannels = 5;
observations = 20;

X1 = rand(height,width,numChannels,observations);
dlX1 = dlarray(X1,"SSCB");

X2 = 1.5*rand(height,width,numChannels,observations);
```

```
dlX2 = dlarray(X2,"SSCB");

X3 = 2.5*rand(height,width,numChannels,observations);
dlX3 = dlarray(X3,"SSCB");
```

Create the learnable parameters.

```
offset = zeros(numChannels,1);
scale = ones(numChannels,1);
```

Normalize the first batch of data `dlX1` using `batchnorm`. Obtain the values of the mean and variance of this batch as outputs.

```
[dlY1,mu,sigmaSq] = batchnorm(dlX1,offset,scale);
```

Normalize the second batch of data `dlX2`. Use `mu` and `sigmaSq` as inputs to obtain the values of the combined mean and variance of the data in batches `dlX1` and `dlX2`.

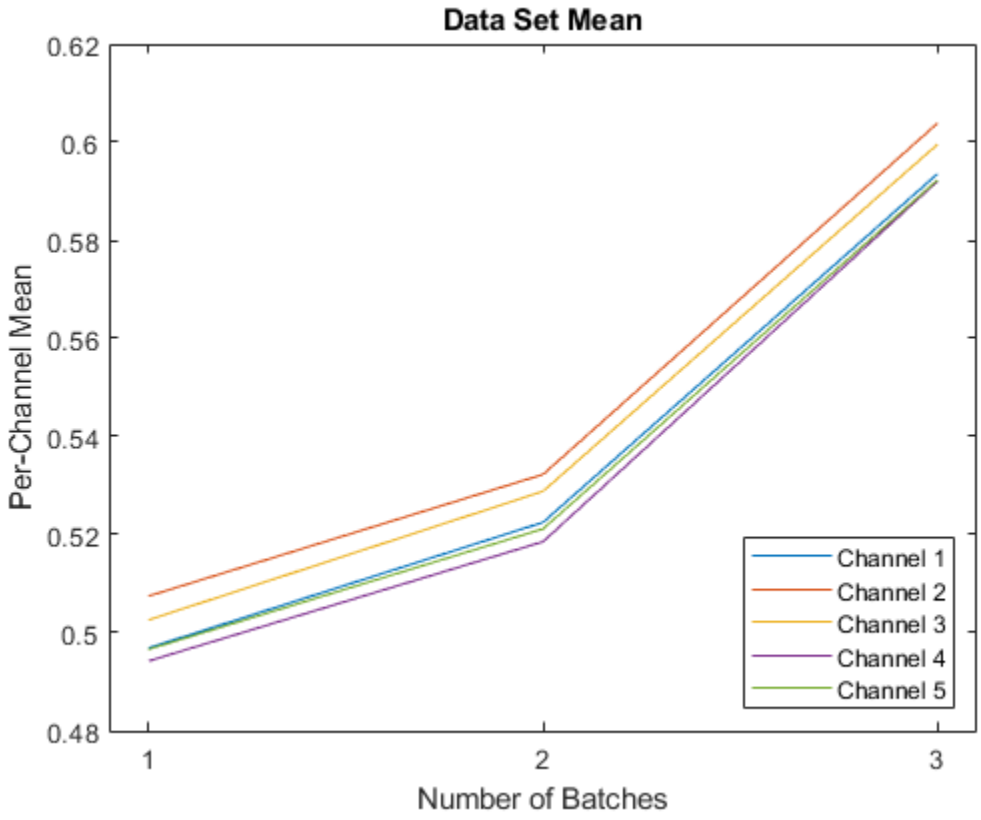
```
[dlY2,datasetMu,datasetSigmaSq] = batchnorm(dlX2,offset,scale,mu,sigmaSq);
```

Normalize the final batch of data `dlX3`. Update the data set statistics `datasetMu` and `datasetSigmaSq` to obtain the values of the combined mean and variance of all data in batches `dlX1`, `dlX2`, and `dlX3`.

```
[dlY3,datasetMuFull,datasetSigmaSqFull] = batchnorm(dlX3,offset,scale,datasetMu,datasetSigmaSq);
```

Observe the change in the mean of each channel as each batch is normalized.

```
plot([mu datasetMu datasetMuFull]')
legend("Channel " + string(1:5),"Location","southeast")
xticks([1 2 3])
xlabel("Number of Batches")
xlim([0.9 3.1])
ylabel("Per-Channel Mean")
title("Data Set Mean")
```



Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `dlX` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `scaleFactor` or `offset` must be a `dlarray` object.

`dlX` must have a `'C'` (channel) dimension.

`offset` — Offset

`dlarray` | numeric array

Offset β, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`.

If `offset` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).

`scaleFactor` — Scale factor

`dlarray` | numeric array

Scale factor γ, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`.

If `scaleFactor` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).

`runningMu` — Running value of mean statistic
numeric vector

Running value of mean statistic, specified as a numeric vector of the same length as the `'C'` dimension of the input data.

To maintain a running value for the mean during training, provide `runningMu` as the `updatedMu` output of the previous training iteration.

Data Types: `single` | `double`

`runningSigmaSq` — Running value of variance statistic
numeric vector

Running value of variance statistic, specified as a numeric vector of the same length as the `'C'` dimension of the input data.

To maintain a running value for the variance during training, provide `runningSigmaSq` as the `updatedSigmaSq` output of the previous training iteration.

Data Types: `single` | `double`

`trainedMu` — Final value of mean statistic after training
numeric vector

Final value of mean statistic after training, specified as a numeric vector of the same length as the `'C'` dimension of the input data.

During classification and prediction, provide `trainedMu` as the `updatedMu` output of the final training iteration.

Data Types: `single` | `double`

`trainedSigmaSq` — Final value of variance statistic after training
numeric vector

Final value of variance statistic after training, specified as a numeric vector of the same length as the `'C'` dimension of the input data.

During classification and prediction, provide `trainedSigmaSq` as the `updatedSigmaSq` output of the final training iteration.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MeanDecay',0.3,'VarianceDecay',0.5` sets the decay rate for the moving average computations of the mean and variance of several batches of data to `0.3` and `0.5`, respectively.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`Epsilon` — Variance offset
`1e-5` (default) | numeric scalar

Variance offset for preventing divide-by-zero errors, specified as the comma-separated pair consisting of `'Epsilon'` and a numeric scalar greater than or equal to `1e-5`.

Data Types: `single` | `double`

`MeanDecay` — Decay value for moving mean computation
`0.1` (default) | numeric scalar between `0` and `1`

Decay value for the moving mean computation, specified as a numeric scalar between `0` and `1`.

The function updates the moving mean value using

$$\mu* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu,$$

where $\mu*$ denotes the updated mean `updatedMu`, λ_μ denotes the mean decay value `'MeanDecay'`, $\widehat{\mu}$ denotes the mean of the input data, and μ denotes the current value of the mean `mu`.

Data Types: `single` | `double`

`VarianceDecay` — Decay value for moving variance computation
`0.1` (default) | numeric scalar between `0` and `1`

Decay value for the moving variance computation, specified as a numeric scalar between `0` and `1`.

The function updates the moving variance value using

$$\sigma^{2}* = \lambda_{\sigma^2}\widehat{\sigma^2} + (1 - \lambda_{\sigma^2})\sigma^2,$$

where σ^{2*} denotes the updated variance `updatedSigmaSq`, λ_{σ^2} denotes the variance decay value `'VarianceDecay'`, $\widehat{\sigma^2}$ denotes the variance of the input data, and σ^2 denotes the current value of the variance `sigmaSq`.

Data Types: `single` | `double`

Output Arguments

`dlY` — Normalized data

`dlarray`

Normalized data, returned as a `dlarray` with the same underlying data type as `dlX`.

If the input data `dlX` is a formatted `dlarray`, then `dlY` has the same format as `dlX`. If the input data is not a formatted `dlarray`, then `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

The size of the output `dlY` matches the size of the input `dlX`.

`popMu` — Per-channel mean

numeric column vector

Per-channel mean of the input data, returned as a numeric column vector with length equal to the size of the `'C'` dimension of the input data.

`popSigmaSq` — Per-channel variance

numeric column vector

Per-channel variance of the input data, returned as a numeric column vector with length equal to the size of the `'C'` dimension of the input data.

`updatedMu` — Updated mean statistic

numeric vector

Updated mean statistic, returned as a numeric vector with length equal to the size of the `'C'` dimension of the input data.

The function updates the moving mean value using

$$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu,$$

where μ^* denotes the updated mean `updatedMu`, λ_μ denotes the mean decay value `'MeanDecay'`, $\widehat{\mu}$ denotes the mean of the input data, and μ denotes the current value of the mean `mu`.

`updatedSigmaSq` — Updated variance statistic

numeric vector

Updated variance statistic, returned as a numeric vector with length equal to the size of the `'C'` dimension of the input data.

The function updates the moving variance value using

$$\sigma^{2*} = \lambda_{\sigma^2}\widehat{\sigma^2} + (1 - \lambda_{\sigma^2})\sigma^2,$$

where σ^{2*} denotes the updated variance `updatedSigmaSq`, $\lambda_\sigma 2$ denotes the variance decay value `'VarianceDecay'`, $\widehat{\sigma^2}$ denotes the variance of the input data, and σ^2 denotes the current value of the variance `sigmaSq`.

Algorithms

The batch normalization operation normalizes the elements x_i of the input by first calculating the mean μ_B and variance ${\sigma_B}^2$ over the spatial, time, and observation dimensions for each channel independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow batch normalization, the batch normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

To make predictions with the network after training, batch normalization requires a fixed mean and variance to normalize the data. This fixed mean and variance can be calculated from the training data after training, or approximated during training using running statistic computations.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `offset`
  - `scaleFactor`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`relu` | `fullyconnect` | `dlconv` | `dlarray` | `dlgradient` | `dlfeval` | `groupnorm` | `layernorm`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Update Batch Normalization Statistics Using Model Function”
“Train Network Using Model Function”

“Train Network with Multiple Outputs”
“List of Functions with dlarray Support”

Introduced in R2019b

batchNormalizationLayer

Batch normalization layer

Description

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

After normalization, the layer scales the input with a learnable scale factor γ and shifts it by a learnable offset β.

Creation

Syntax

```
layer = batchNormalizationLayer
layer = batchNormalizationLayer(Name,Value)
```

Description

`layer = batchNormalizationLayer` creates a batch normalization layer.

`layer = batchNormalizationLayer(Name,Value)` creates a batch normalization layer and sets the optional `TrainedMean`, `TrainedVariance`, `Epsilon`, "Parameters and Initialization" on page 1-188, "Learning Rate and Regularization" on page 1-190, and `Name` properties using one or more name-value pairs. For example, `batchNormalizationLayer('Name','batchnorm')` creates a batch normalization layer with the name `'batchnorm'`.

Properties

Batch Normalization

`TrainedMean` — Mean statistic used for prediction
numeric array

Mean statistic used for prediction, specified as one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

If the `'BatchNormalizationStatistics'` training option is `'moving'`, then the software approximates the batch normalization statistics during training using a running estimate and, after training, sets the `TrainedMean` and `TrainedVariance` properties to the latest values of the moving estimates of the mean and variance, respectively.

If the 'BatchNormalizationStatistics' training option is 'population', then after network training finishes, the software passes through the data once more and sets the TrainedMean and TrainedVariance properties to the mean and variance computed from the entire training data set, respectively.

The layer uses TrainedMean and TrainedVariance to normalize the input during prediction.

TrainedVariance — Variance statistic used for prediction
numeric array

Variance statistic used for prediction, specified as one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-NumChannels
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-NumChannels
- For feature or sequence input, a numeric array of size NumChannels-by-1

If the 'BatchNormalizationStatistics' training option is 'moving', then the software approximates the batch normalization statistics during training using a running estimate and, after training, sets the TrainedMean and TrainedVariance properties to the latest values of the moving estimates of the mean and variance, respectively.

If the 'BatchNormalizationStatistics' training option is 'population', then after network training finishes, the software passes through the data once more and sets the TrainedMean and TrainedVariance properties to the mean and variance computed from the entire training data set, respectively.

The layer uses TrainedMean and TrainedVariance to normalize the input during prediction.

Epsilon — Constant to add to mini-batch variances
1e-5 (default) | numeric scalar

Constant to add to the mini-batch variances, specified as a numeric scalar equal to or larger than 1e-5.

The layer adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

NumChannels — Number of input channels
'auto' (default) | positive integer

Number of input channels, specified as 'auto' or a positive integer.

This property is always equal to the number of channels of the input to the layer. If NumChannels is 'auto', then the software automatically determines the correct value for the number of channels at training time.

Parameters and Initialization

ScaleInitializer — Function to initialize channel scale factors
'ones' (default) | 'narrow-normal' | function handle

Function to initialize the channel scale factors, specified as one of the following:

- 'ones' – Initialize the channel scale factors with ones.

- `'zeros'` – Initialize the channel scale factors with zeros.
- `'narrow-normal'` – Initialize the channel scale factors by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel scale factors with a custom function. If you specify a function handle, then the function must be of the form `scale = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel scale factors when the `Scale` property is empty.

Data Types: `char` | `string` | `function_handle`

`OffsetInitializer` — Function to initialize channel offsets
`'zeros'` (default) | `'ones'` | `'narrow-normal'` | function handle

Function to initialize the channel offsets, specified as one of the following:

- `'zeros'` – Initialize the channel offsets with zeros.
- `'ones'` – Initialize the channel offsets with ones.
- `'narrow-normal'` – Initialize the channel offsets by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel offsets with a custom function. If you specify a function handle, then the function must be of the form `offset = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel offsets when the `Offset` property is empty.

Data Types: `char` | `string` | `function_handle`

`Scale` — Channel scale factors
`[]` (default) | numeric array

Channel scale factors γ, specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network, if `Scale` is nonempty, then `trainNetwork` uses the `Scale` property as the initial value. If `Scale` is empty, then `trainNetwork` uses the initializer specified by `ScaleInitializer`.

At training time, `Scale` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

`Offset` — Channel offsets
`[]` (default) | numeric array

Channel offsets β, specified as a numeric array.

The channel offsets are learnable parameters. When you train a network, if `Offset` is nonempty, then `trainNetwork` uses the `Offset` property as the initial value. If `Offset` is empty, then `trainNetwork` uses the initializer specified by `OffsetInitializer`.

At training time, `Offset` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

`MeanDecay` — Decay value for moving mean computation
0.1 (default) | numeric scalar between `0` and `1`

Decay value for the moving mean computation, specified as a numeric scalar between `0` and `1`.

When the `'BatchNormalizationStatistics'` training option is `'moving'`, at each iteration, the layer updates the moving mean value using

$$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu,$$

where μ^* denotes the updated mean, λ_μ denotes the mean decay value, $\widehat{\mu}$ denotes the mean of the layer input, and μ denotes the latest value of the moving mean value.

If the `'BatchNormalizationStatistics'` training option is `'population'`, then this option has no effect.

Data Types: `single` | `double`

`VarianceDecay` — Decay value for moving variance computation
0.1 (default) | numeric scalar between `0` and `1`

Decay value for the moving variance computation, specified as a numeric scalar between `0` and `1`.

When the `'BatchNormalizationStatistics'` training option is `'moving'`, at each iteration, the layer updates the moving variance value using

$$\sigma^{2*} = \lambda_{\sigma^2} \widehat{\sigma^2} + (1 - \lambda_{\sigma^2})\sigma^2,$$

where σ^{2*} denotes the updated variance, λ_{σ^2} denotes the variance decay value, $\widehat{\sigma^2}$ denotes the variance of the layer input, and σ^2 denotes the latest value of the moving variance value.

If the `'BatchNormalizationStatistics'` training option is `'population'`, then this option has no effect.

Data Types: `single` | `double`

Learning Rate and Regularization

`ScaleLearnRateFactor` — Learning rate factor for scale factors
1 (default) | nonnegative scalar

Learning rate factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the scale factors in a layer. For example, if `ScaleLearnRateFactor` is `2`, then the learning rate for the scale factors in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`OffsetLearnRateFactor` — Learning rate factor for offsets
1 (default) | nonnegative scalar

Learning rate factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the offsets in a layer. For example, if `OffsetLearnRateFactor` is `2`, then the learning rate for the offsets in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`ScaleL2Factor` — L_2 regularization factor for scale factors
`1` (default) | nonnegative scalar

L_2 regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

`OffsetL2Factor` — L_2 regularization factor for offsets
`1` (default) | nonnegative scalar

L_2 regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Batch Normalization Layer

Create a batch normalization layer with the name `'BN1'`.

```
layer = batchNormalizationLayer('Name','BN1')
```

```
layer =
  BatchNormalizationLayer with properties:

                 Name: 'BN1'
        NumChannels: 'auto'
        TrainedMean: []
    TrainedVariance: []

   Hyperparameters
          MeanDecay: 0.1000
      VarianceDecay: 0.1000
            Epsilon: 1.0000e-05

   Learnable Parameters
             Offset: []
              Scale: []

  Show all properties
```

Include batch normalization layers in a `Layer` array.

```
layers = [
    imageInputLayer([32 32 3])

    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)
```

```
    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
    ]
```

```
layers =
  11x1 Layer array with layers:

     1   ''   Image Input             32x32x3 images with 'zerocenter' normalization
     2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [1  1
     3   ''   Batch Normalization     Batch normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [1  1
     7   ''   Batch Normalization     Batch normalization
     8   ''   ReLU                    ReLU
     9   ''   Fully Connected         10 fully connected layer
    10   ''   Softmax                 softmax
    11   ''   Classification Output   crossentropyex
```

More About

Batch Normalization Layer

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

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

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

Algorithms

The batch normalization operation normalizes the elements x_i of the input by first calculating the mean μ_B and variance σ_B^2 over the spatial, time, and observation dimensions for each channel independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow batch normalization, the batch normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

To make predictions with the network after training, batch normalization requires a fixed mean and variance to normalize the data. This fixed mean and variance can be calculated from the training data after training, or approximated during training using running statistic computations.

If the `'BatchNormalizationStatistics'` training option is `'moving'`, then the software approximates the batch normalization statistics during training using a running estimate and, after training, sets the `TrainedMean` and `TrainedVariance` properties to the latest values of the moving estimates of the mean and variance, respectively.

If the `'BatchNormalizationStatistics'` training option is `'population'`, then after network training finishes, the software passes through the data once more and sets the `TrainedMean` and `TrainedVariance` properties to the mean and variance computed from the entire training data set, respectively.

The layer uses `TrainedMean` and `TrainedVariance` to normalize the input during prediction.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `trainingOptions` | `reluLayer` | `convolution2dLayer` | `fullyConnectedLayer` | `groupNormalizationLayer` | `layerNormalizationLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2017b

bilstmLayer

Bidirectional long short-term memory (BiLSTM) layer

Description

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.

Creation

Syntax

```
layer = bilstmLayer(numHiddenUnits)
layer = bilstmLayer(numHiddenUnits,Name,Value)
```

Description

`layer = bilstmLayer(numHiddenUnits)` creates a bidirectional LSTM layer and sets the `NumHiddenUnits` property.

`layer = bilstmLayer(numHiddenUnits,Name,Value)` sets additional `OutputMode`, “Activations” on page 1-197, “State” on page 1-198, “Parameters and Initialization” on page 1-198, “Learning Rate and Regularization” on page 1-201, and `Name` properties using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.

Properties

BiLSTM

`NumHiddenUnits` — Number of hidden units

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data. This value can vary from a few dozen to a few thousand.

The hidden state does not limit the number of time steps that are processed in an iteration. To split your sequences into smaller sequences for training, use the `'SequenceLength'` option in `trainingOptions`.

Example: 200

`OutputMode` — Output mode

`'sequence'` (default) | `'last'`

Output mode, specified as one of the following:

- `'sequence'` – Output the complete sequence.
- `'last'` – Output the last time step of the sequence.

`HasStateInputs` — Flag for state inputs to layer

`0` (false) (default) | `1` (true)

Flag for state inputs to the layer, specified as `0` (false) or `1` (true).

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

`HasStateOutputs` — Flag for state outputs from layer

`0` (false) (default) | `1` (true)

Flag for state outputs from the layer, specified as `0` (false) or `1` (true).

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

`InputSize` — Input size

`'auto'` (default) | positive integer

Input size, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically assigns the input size at training time.

Example: 100

Activations

`StateActivationFunction` — Activation function to update the cell and hidden state

`'tanh'` (default) | `'softsign'`

Activation function to update the cell and hidden state, specified as one of the following:

- `'tanh'` – Use the hyperbolic tangent function (tanh).
- `'softsign'` – Use the softsign function $\text{softsign}(x) = \frac{x}{1 + |x|}$.

The layer uses this option as the function σ_c in the calculations to update the cell and hidden state. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1000.

`GateActivationFunction` — Activation function to apply to the gates
`'sigmoid'` (default) | `'hard-sigmoid'`

Activation function to apply to the gates, specified as one of the following:

- `'sigmoid'` – Use the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$.
- `'hard-sigmoid'` – Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if} -2.5 \le x \le 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}.$$

The layer uses this option as the function σ_g in the calculations for the layer gates.

State

`CellState` — Cell state
numeric vector

Cell state to use in the layer operation, specified as a `2*NumHiddenUnits`-by-1 numeric vector. This value corresponds to the initial cell state when data is passed to the layer.

After setting this property manually, calls to the `resetState` function set the cell state to this value.

If `HasStateInputs` is `true`, then the `CellState` property must be empty.

`HiddenState` — Hidden state
numeric vector

Hidden state to use in the layer operation, specified as a `2*NumHiddenUnits`-by-1 numeric vector. This value corresponds to the initial hidden state when data is passed to the layer.

After setting this property manually, calls to the `resetState` function set the hidden state to this value.

If `HasStateInputs` is `true`, then the `HiddenState` property must be empty.

Parameters and Initialization

`InputWeightsInitializer` — Function to initialize input weights
`'glorot'` (default) | `'he'` | `'orthogonal'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the input weights, specified as one of the following:

- `'glorot'` – Initialize the input weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(InputSize + numOut)`, where `numOut = 8*NumHiddenUnits`.
- `'he'` – Initialize the input weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/InputSize`.

- `'orthogonal'` – Initialize the input weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [3]
- `'narrow-normal'` – Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the input weights with zeros.
- `'ones'` – Initialize the input weights with ones.
- Function handle – Initialize the input weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the input weights.

The layer only initializes the input weights when the `InputWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`RecurrentWeightsInitializer` — Function to initialize recurrent weights
`'orthogonal'` (default) | `'glorot'` | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the recurrent weights, specified as one of the following:

- `'orthogonal'` – Initialize the input weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [3]
- `'glorot'` – Initialize the recurrent weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = NumHiddenUnits` and `numOut = 8*NumHiddenUnits`.
- `'he'` – Initialize the recurrent weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/NumHiddenUnits`.
- `'narrow-normal'` – Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the recurrent weights with zeros.
- `'ones'` – Initialize the recurrent weights with ones.
- Function handle – Initialize the recurrent weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the recurrent weights.

The layer only initializes the recurrent weights when the `RecurrentWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'unit-forget-gate'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'unit-forget-gate'` – Initialize the forget gate bias with ones and the remaining biases with zeros.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'ones'` – Initialize the bias with ones.

- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`InputWeights` — Input weights
`[ ]` (default) | matrix

Input weights, specified as a matrix.

The input weight matrix is a concatenation of the eight input weight matrices for the components (gates) in the bidirectional LSTM layer. The eight matrices are concatenated vertically in the following order:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

The input weights are learnable parameters. When training a network, if `InputWeights` is nonempty, then `trainNetwork` uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then `trainNetwork` uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is an `8*NumHiddenUnits`-by-`InputSize` matrix.

`RecurrentWeights` — Recurrent weights
`[ ]` (default) | matrix

Recurrent weights, specified as a matrix.

The recurrent weight matrix is a concatenation of the eight recurrent weight matrices for the components (gates) in the bidirectional LSTM layer. The eight matrices are concatenated vertically in the following order:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

The recurrent weights are learnable parameters. When training a network, if `RecurrentWeights` is nonempty, then `trainNetwork` uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then `trainNetwork` uses the initializer specified by `RecurrentWeightsInitializer`.

At training time, `RecurrentWeights` is an `8*NumHiddenUnits`-by-`NumHiddenUnits` matrix.

`Bias` — Layer biases

`[ ]` (default) | numeric vector

Layer biases, specified as a numeric vector.

The bias vector is a concatenation of the eight bias vectors for the components (gates) in the bidirectional LSTM layer. The eight vectors are concatenated vertically in the following order:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is an `8*NumHiddenUnits`-by-1 numeric vector.

Learning Rate and Regularization

`InputWeightsLearnRateFactor` — Learning rate factor for input weights

1 (default) | numeric scalar | 1-by-8 numeric vector

Learning rate factor for the input weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input weights of the layer. For example, if `InputWeightsLearnRateFactor` is 2, then the learning rate factor for the input weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the four individual matrices in `InputWeights`, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)

5 Input gate (Backward)
6 Forget gate (Backward)
7 Cell candidate (Backward)
8 Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `0.1`

`RecurrentWeightsLearnRateFactor` — Learning rate factor for recurrent weights
1 (default) | numeric scalar | 1-by-8 numeric vector

Learning rate factor for the recurrent weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the recurrent weights of the layer. For example, if `RecurrentWeightsLearnRateFactor` is 2, then the learning rate for the recurrent weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learn rate for the four individual matrices in `RecurrentWeights`, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1 Input gate (Forward)
2 Forget gate (Forward)
3 Cell candidate (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Cell candidate (Backward)
8 Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `0.1`

Example: `[1 2 1 1 1 2 1 1]`

`BiasLearnRateFactor` — Learning rate factor for biases
1 (default) | nonnegative scalar | 1-by-8 numeric vector

Learning rate factor for the biases, specified as a nonnegative scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

To control the value of the learning rate factor for the four individual matrices in `Bias`, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1 1 2 1 1]`

`InputWeightsL2Factor` — L2 regularization factor for input weights

1 (default) | numeric scalar | 1-by-8 numeric vector

L2 regularization factor for the input weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the L2 regularization factor for the input weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `InputWeights`, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `0.1`

Example: `[1 2 1 1 1 2 1 1]`

`RecurrentWeightsL2Factor` — L2 regularization factor for recurrent weights

1 (default) | numeric scalar | 1-by-8 numeric vector

L2 regularization factor for the recurrent weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the L2 regularization factor for the recurrent weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `RecurrentWeights`, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `0.1`

Example: `[1 2 1 1 1 2 1 1]`

`BiasL2Factor` — L2 regularization factor for biases
0 (default) | nonnegative scalar | 1-by-8 numeric vector

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `Bias`, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `2`

Example: `[1 2 1 1 1 2 1 1]`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` | `3`

Number of inputs of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

Data Types: `double`

`InputNames` — Input names
`{'in'}` | `{'in','hidden','cell'}`

Input names of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

`NumOutputs` — Number of outputs
`1` | `3`

Number of outputs of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` | `{'out','hidden','cell'}`

Output names of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Examples

Create Bidirectional LSTM Layer

Create a bidirectional LSTM layer with the name `'bilstm1'` and 100 hidden units.

```
layer = bilstmLayer(100,'Name','bilstm1')
```

```
layer =
  BiLSTMLayer with properties:

                      Name: 'bilstm1'
                 InputNames: {'in'}
                OutputNames: {'out'}
                  NumInputs: 1
                 NumOutputs: 1
             HasStateInputs: 0
            HasStateOutputs: 0

   Hyperparameters
                  InputSize: 'auto'
             NumHiddenUnits: 100
                 OutputMode: 'sequence'
    StateActivationFunction: 'tanh'
     GateActivationFunction: 'sigmoid'

   Learnable Parameters
               InputWeights: []
           RecurrentWeights: []
                       Bias: []

   State Parameters
                HiddenState: []
                  CellState: []

  Show all properties
```

Include a bidirectional LSTM layer in a `Layer` array.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    bilstmLayer(numHiddenUnits)
```

```
    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
```

Algorithms

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `BiLSTMLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	OutputMode	Output Format
`"CBT"` (channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)

In `dlnetwork` objects, `BiLSTMLayer` objects also support the following input and output format combinations.

Input Format	OutputMode	Output Format
`"SCBT"` (spatial, channel, batch)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)
`"SSCBT"` (spatial, spatial, channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)
`"SSSCBT"` (spatial, spatial, spatial, channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)

To use these input formats in `trainNetwork` workflows, first convert the data to `"CBT"` (channel, batch, time) format using `flattenLayer`.

If the `HasStateInputs` property is `1` (true), then the layer has two additional inputs with names `'hidden'` and `'cell'`, which correspond to the hidden state and cell state, respectively. These additional inputs expect input format `"CB"` (channel, batch).

If the `HasStateOutputs` property is `1` (true), then the layer has two additional outputs with names `'hidden'` and `'cell'`, which correspond to the hidden state and cell state, respectively. These additional outputs have output format `"CB"` (channel, batch).

Compatibility Considerations

Default input weights initialization is Glorot
Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer input weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer input weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'InputWeightsInitializer'` option of the layer to `'narrow-normal'`.

Default recurrent weights initialization is orthogonal
Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer recurrent weights of this layer with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer recurrent weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'RecurrentWeightsInitializer'` option of the layer to `'narrow-normal'`.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

[3] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." *arXiv preprint arXiv:1312.6120* (2013).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When generating code with Intel MKL-DNN:

- The `StateActivationFunction` property must be set to `'tanh'`.
- The `GateActivationFunction` property must be set to `'sigmoid'`.
- The `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `StateActivationFunction` property must be set to `'tanh'`.
- For GPU code generation, the `GateActivationFunction` property must be set to `'sigmoid'`.
- The `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `gruLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Topics
“Sequence Classification Using Deep Learning”
“Sequence Classification Using 1-D Convolutions”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Classify Videos Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2018a

calibrate

Simulate and collect ranges of a deep neural network

Syntax

```
calibrationResults = calibrate(quantObj, calData)
calibrationResults = calibrate(quantObj, calData,Name,Value)
```

Description

`calibrationResults = calibrate(quantObj, calData)` exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network specified by `dlquantizer` object, `quantObj`, using the data specified by `calData`.

`calibrationResults = calibrate(quantObj, calData,Name,Value)` exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network specified by `dlquantizer` object, `quantObj`, using the data specified by `calData`, with additional arguments specified by one or more name-value pair arguments.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”

Examples

Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load net
net
```

```
net =
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

In this example, use the images in the `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

Create a `dlquantizer` object and specify the network to quantize.

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare with predicted label with actual ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=95×5 table
                  Optimized Layer Name                       Network Layer Name       Learnables
    ___________________________________________________    ____________________________    __________

    {'conv1_relu_conv1_Weights'                       }    {'relu_conv1'           }          "We
    {'conv1_relu_conv1_Bias'                          }    {'relu_conv1'           }          "Bi
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}          "We
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}          "Bi
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }          "We
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }          "Bi
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }          "We
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }          "Bi
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}          "We
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}          "Bi
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }          "We
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }          "Bi
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }          "We
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }          "Bi
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}          "We
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}          "Bi
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

Quantize a Neural Network for FPGA Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working directory called `getLogoNetwork.m`. Enter these lines into the file:

```
function net = getLogoNetwork()
    data = getLogoData();
    net  = data.convnet;
end

function data = getLogoData()
    if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
end
```

Load the pretrained network.

```
snet = getLogoNetwork();
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

This example uses the images in the `logos_dataset` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
```

Create a `dlquantizer` object and specify the network to quantize.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
 dlQuantObj.calibrate(calibrationData)
```

```
ans =
       Optimized Layer Name        Network Layer Name    Learnables / Activations    MinValue      MaxValue
    ___________________________    __________________    ________________________    __________    _________

    {'conv_1_Weights'          }      {'conv_1'    }          "Weights"                -0.048978      0.039352
    {'conv_1_Bias'             }      {'conv_1'    }          "Bias"                     0.99996        1.0028
    {'conv_2_Weights'          }      {'conv_2'    }          "Weights"                -0.055518      0.061901
    {'conv_2_Bias'             }      {'conv_2'    }          "Bias"                  -0.00061171       0.00227
    {'conv_3_Weights'          }      {'conv_3'    }          "Weights"                -0.045942      0.046927
    {'conv_3_Bias'             }      {'conv_3'    }          "Bias"                   -0.0013998     0.0015218
    {'conv_4_Weights'          }      {'conv_4'    }          "Weights"                -0.045967          0.051
    {'conv_4_Bias'             }      {'conv_4'    }          "Bias"                    -0.00164     0.0037892
    {'fc_1_Weights'            }      {'fc_1'      }          "Weights"                -0.051394      0.054344
    {'fc_1_Bias'               }      {'fc_1'      }          "Bias"                  -0.00052319    0.00084454
    {'fc_2_Weights'            }      {'fc_2'      }          "Weights"                 -0.05016      0.051557
    {'fc_2_Bias'               }      {'fc_2'      }          "Bias"                   -0.0017564     0.0018502
    {'fc_3_Weights'            }      {'fc_3'      }          "Weights"                -0.050706       0.04678
    {'fc_3_Bias'               }      {'fc_3'      }          "Bias"                    -0.02951      0.024855
    {'imageinput'              }      {'imageinput'}          "Activations"                    0           255
    {'imageinput_normalization'}      {'imageinput'}          "Activations"             -139.34        198.72
```

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To create the target object, enter:

```
hTarget = dlhdl.Target('Intel', 'Interface', 'JTAG');
```

Define a metric function to use to compare the behavior of the network before and after quantization. Save this function in a local file.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% hComputeModelAccuracy test helper function computes model level accuracy statistics

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx, :));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
```

```
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
options = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x, snet, validationData)},'Bitstream','arria10soc_int8',...
'Target',hTarget);
```

To compile and deploy the quantized network, run the `validate` function of the `dlquantizer` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function checks for the Intel Quartus tool and the supported tool version. It then starts programming the FPGA device by using the sof file, displays progress messages, and the time it takes to deploy the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
prediction = dlQuantObj.validate(validationData,options);
```

```
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"          "0x00000000"     "48.0 MB"
    "OutputResultOffset"       "0x03000000"     "4.0 MB"
    "SystemBufferOffset"       "0x03400000"     "60.0 MB"
    "InstructionDataOffset"    "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"     "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"       "0x08000000"     "12.0 MB"
    "EndOffset"                "0x08c00000"     "Total: 140.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Jul-2020 12:45:10
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Jul-2020 12:45:26
### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                  LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                        -------------              -------------              ---------        ---------       ---------
Network                   13570959                    0.09047                    30          380609145            11.8
    conv_module           12667786                    0.08445
        conv_1             3938907                    0.02626
        maxpool_1          1544560                    0.01030
        conv_2             2910954                    0.01941
        maxpool_2           577524                    0.00385
        conv_3             2552707                    0.01702
        maxpool_3           676542                    0.00451
        conv_4              455434                    0.00304
        maxpool_4            11251                    0.00008
    fc_module               903173                    0.00602
        fc_1                536164                    0.00357
        fc_2                342643                    0.00228
        fc_3                 24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results
```

```
                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                   -------------------------  -------------------------       ---------      -------------     ---------
Network                     13570364                    0.09047                   30          380612682            11.8
    conv_module             12667103                    0.08445
        conv_1               3939296                    0.02626
        maxpool_1            1544371                    0.01030
        conv_2               2910747                    0.01940
        maxpool_2             577654                    0.00385
        conv_3               2551829                    0.01701
        maxpool_3             676548                    0.00451
        conv_4                455396                    0.00304
        maxpool_4              11355                    0.00008
    fc_module                 903261                    0.00602
        fc_1                  536206                    0.00357
        fc_2                  342688                    0.00228
        fc_3                   24365                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                   -------------------------  -------------------------       ---------      -------------     ---------
Network                     13571561                    0.09048                   30          380608338            11.8
    conv_module             12668340                    0.08446
        conv_1               3939070                    0.02626
        maxpool_1            1545327                    0.01030
        conv_2               2911061                    0.01941
        maxpool_2             577557                    0.00385
        conv_3               2552082                    0.01701
        maxpool_3             676506                    0.00451
        conv_4                455582                    0.00304
        maxpool_4              11248                    0.00007
    fc_module                 903221                    0.00602
        fc_1                  536167                    0.00357
        fc_2                  342643                    0.00228
        fc_3                   24409                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                   -------------------------  -------------------------       ---------      -------------     ---------
Network                     13569862                    0.09047                   30          380613327            11.8
    conv_module             12666756                    0.08445
        conv_1               3939212                    0.02626
        maxpool_1            1543267                    0.01029
        conv_2               2911184                    0.01941
        maxpool_2             577275                    0.00385
        conv_3               2552868                    0.01702
        maxpool_3             676438                    0.00451
        conv_4                455353                    0.00304
        maxpool_4              11252                    0.00008
    fc_module                 903106                    0.00602
        fc_1                  536050                    0.00357
        fc_2                  342645                    0.00228
        fc_3                   24409                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.
```

```
              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13570823                    0.09047                      30          380619836            11.8
    conv_module            12667607                    0.08445
        conv_1              3939074                    0.02626
        maxpool_1           1544519                    0.01030
        conv_2              2910636                    0.01940
        maxpool_2            577769                    0.00385
        conv_3              2551800                    0.01701
        maxpool_3            676795                    0.00451
        conv_4               455859                    0.00304
        maxpool_4             11248                    0.00007
    fc_module                903216                    0.00602
        fc_1                 536165                    0.00357
        fc_2                 342643                    0.00228
        fc_3                  24406                    0.00016
 * The clock frequency of the DL processor is: 150MHz

          offset_name          offset_address     allocated_space
    ___________________________    _____________    _________________

    "InputDataOffset"             "0x00000000"     "48.0 MB"
    "OutputResultOffset"          "0x03000000"     "4.0 MB"
    "SystemBufferOffset"          "0x03400000"     "60.0 MB"
    "InstructionDataOffset"       "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"        "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"          "0x08000000"     "12.0 MB"
    "EndOffset"                   "0x08c00000"     "Total: 140.0 MB"

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target FPGA.
### Deep learning network programming has been skipped as the same network is already loaded on the target FPGA.
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13572329                    0.09048                      10          127265075            11.8
    conv_module            12669135                    0.08446
        conv_1              3939559                    0.02626
        maxpool_1           1545378                    0.01030
        conv_2              2911243                    0.01941
        maxpool_2            577422                    0.00385
        conv_3              2552064                    0.01701
        maxpool_3            676678                    0.00451
        conv_4               455657                    0.00304
        maxpool_4             11227                    0.00007
    fc_module                903194                    0.00602
        fc_1                 536140                    0.00357
        fc_2                 342688                    0.00228
        fc_3                  24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13572527                    0.09048                      10          127266427            11.8
    conv_module            12669266                    0.08446
        conv_1              3939776                    0.02627
```

```
       maxpool_1          1545632                 0.01030
       conv_2             2911169                 0.01941
       maxpool_2           577592                 0.00385
       conv_3             2551613                 0.01701
       maxpool_3           676811                 0.00451
       conv_4              455418                 0.00304
       maxpool_4            11348                 0.00008
   fc_module               903261                 0.00602
       fc_1                536205                 0.00357
       fc_2                342689                 0.00228
       fc_3                 24365                 0.00016
 * The clock frequency of the DL processor is: 150MHz
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
validateOut = prediction.MetricResults.Result
```

```
ans =
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}         0.9875
     {'Quantized'     }         0.9875
```

Examine the `QuantizedNetworkFPS` field of the validation output to see the frames per second performance of the quantized network.

```
prediction.QuantizedNetworkFPS
```

```
ans = 11.8126
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

Input Arguments

`quantObj` — Network to quantize

`dlquantizer` object

`dlquantizer` object containing the network to quantize.

`calData` — Data to use for calibration of quantized network

`imageDatastore` object | `augmentedImageDatastore` object | `pixelLabelImageDatastore` object

Data to use for calibration of quantized network, specified as an `imageDatastore` object, an `augmentedImageDatastore` object, or a `pixelLabelImageDatastore` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `calResults = calibrate(quantObj, calData, 'UseGPU', 'on')`

`UseGPU` — Logical flag to use GPU for calibration

'off' (default) | 'on'

Logical flag to use a GPU for calibration when the `dlquantizer` object `ExecutionEnvironment` is set to `'FPGA'` or `'CPU'`.

Example: `'UseGPU', 'on'`

Output Arguments

`calibrationResults` — Dynamic ranges of network

table

Dynamic ranges of layers of the network, returned as a table. Each row in the table displays the minimum and maximum values of a learnable parameter of a convolution layer of the optimized network. The software uses these minimum and maximum values to determine the scaling for the data type of the quantized parameter.

See Also

Apps
Deep Network Quantizer

Functions
`validate` | `dlquantizer` | `dlquantizationOptions`

Topics
“Quantization of Deep Neural Networks”

Introduced in R2020a

checkLayer

Check validity of custom or function layer

Syntax

```
checkLayer(layer,validInputSize)
checkLayer(layer,validInputSize,Name=Value)
```

Description

`checkLayer(layer,validInputSize)` checks the validity of a custom or function layer using generated data of the sizes in `validInputSize`. For layers with a single input, set `validInputSize` to a typical size of input data to the layer. For layers with multiple inputs, set `validInputSize` to a cell array of typical sizes, where each element corresponds to a layer input.

`checkLayer(layer,validInputSize,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Check Custom Layer Validity

Check the validity of the example custom layer `preluLayer`.

The custom layer `preluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check that it is valid using `checkLayer`. Set the valid input size to the typical size of a single observation input to the layer. For a single input, the layer expects observations of size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* are the height, width, and number of channels of the previous layer output, respectively.

Specify `validInputSize` as the typical size of an input array.

```
layer = preluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

```
Skipping multi-observation tests. To enable tests with multiple observations, specify the 'Observ
For 2-D image data, set 'ObservationDimension' to 4.
For 3-D image data, set 'ObservationDimension' to 5.
For sequence data, set 'ObservationDimension' to 2.

Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ..
Done nnet.checklayer.TestLayerWithoutBackward
```

```
__________

Test Summary:
     12 Passed, 0 Failed, 0 Incomplete, 16 Skipped.
     Time elapsed: 0.17942 seconds.
```

The results show the number of passed, failed, and skipped tests. If you do not specify the `ObservationsDimension` option, or do not have a GPU, then the function skips the corresponding tests.

Check Multiple Observations

For multi-observation input, the layer expects an array of observations of size *h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* are the height, width, and number of channels, respectively, and *N* is the number of observations.

To check the layer validity for multiple observations, specify the typical size of an observation and set the `ObservationDimension` option to 4.

```
layer = preluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,ObservationDimension=4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 0.077629 seconds.
```

In this case, the function does not detect any issues with the layer.

Check Function Layer Validity

Create a function layer object that applies the softsign operation to the input. The softsign operation is given by the function $f(x) = \frac{x}{1 + |x|}$.

```
layer = functionLayer(@(X) X./(1 + abs(X)))
```

```
layer =
  FunctionLayer with properties:

           Name: ''
    PredictFcn: @(X)X./(1+abs(X))
   Formattable: 0

   Learnable Parameters
    No properties.
```

```
   State Parameters
    No properties.

  Show all properties
```

Check that the layer it is valid using the `checkLayer` function. Set the valid input size to the typical size of a single observation input to the layer. For example, 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.

```
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

```
Skipping multi-observation tests. To enable tests with multiple observations, specify the 'Observ
For 2-D image data, set 'ObservationDimension' to 4.
For 3-D image data, set 'ObservationDimension' to 5.
For sequence data, set 'ObservationDimension' to 2.

Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ..
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     12 Passed, 0 Failed, 0 Incomplete, 16 Skipped.
     Time elapsed: 0.26747 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 image input, the layer expects an array of observations of size *h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* are the height, width, and number of channels, respectively, and *N* is the number of observations.

To check the layer validity for multiple observations, specify the typical size of an observation and set the `ObservationDimension` option to 4.

```
layer = functionLayer(@(X) X./(1 + abs(X)));
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,ObservationDimension=4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
```

```
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 0.22641 seconds.
```

In this case, the function does not detect any issues with the layer.

Check Custom Layer for Code Generation Compatibility

Check the code generation compatibility of the custom layer `codegenPreluLayer`.

The custom layer `codegenPreluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size as the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set the `'ObservationDimension'` option to 4. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to `true`. The `checkLayer` function does not check for functions that are not compatible with code generation. To check that the custom layer definition is supported for code generation, first use the **Code Generation Readiness** app. For more information, see “Check Code by Using the Code Generation Readiness Tool” (MATLAB Coder).

```
layer = codegenPreluLayer(20,"prelu");
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,ObservationDimension=4,CheckCodegenCompatibility=true)
```

```
Skipping GPU tests. No compatible GPU device found.

Running nnet.checklayer.TestLayerWithoutBackward
.......... .......... ...
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     23 Passed, 0 Failed, 0 Incomplete, 5 Skipped.
     Time elapsed: 1.0234 seconds.
```

The function does not detect any issues with the layer.

Input Arguments

`layer` — Layer to check

`nnet.layer.Layer` object | `nnet.layer.ClassificationLayer` object | `nnet.layer.RegressionLayer` object

Layer to check, specified as an `nnet.layer.Layer`, `nnet.layer.ClassificationLayer`, `nnet.layer.RegressionLayer`, or `FunctionLayer` object.

For an example showing how to define your own custom layer, see “Define Custom Deep Learning Layer with Learnable Parameters”. To create a layer that applies a specified function, use `functionLayer`.

The `checkLayer` function does not support layers that inherit from `nnet.layer.Formattable`.

`validInputSize` — Valid input sizes
vector of positive integers | cell array of vectors of positive integers

Valid input sizes of the layer, specified as a vector of positive integers or cell array of vectors of positive integers.

- For layers with a single input, specify `validInputSize` as a vector of integers corresponding to the dimensions of the input data. For example, `[5 5 10]` corresponds to valid input data of size 5-by-5-by-10.
- For layers with multiple inputs, specify `validInputSize` as a cell array of vectors, where each vector corresponds to a layer input and the elements of the vectors correspond to the dimensions of the corresponding input data. For example, `{[24 24 20],[24 24 10]}` corresponds to the valid input sizes of two inputs, where 24-by-24-by-20 is a valid input size for the first input and 24-by-24-by-10 is a valid input size for the second input.

For more information, see “Layer Input Sizes” on page 1-225.

For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Example: `[5 5 10]`

Example: `{[24 24 20],[24 24 10]}`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ObservationDimension=4` sets the observation dimension to 4

`ObservationDimension` — Observation dimension
positive integer

Observation dimension, specified as a positive integer.

The observation dimension specifies which dimension of the layer input data corresponds to observations. For example, if the layer expects input data is of size *h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the input data, respectively, and *N* corresponds to the number of observations, then the observation dimension is 4. For more information, see “Layer Input Sizes” on page 1-225.

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 the observation dimension, then the function skips the corresponding tests.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CheckCodegenCompatibility` — Flag to enable code generation tests
`0` (false) (default) | `1` (true)

Flag to enable code generation tests, specified as `0` (false) or `1` (true).

If `CheckCodegenCompatibility` is `1` (true), then you must specify the `ObservationDimension` option.

Code generation supports intermediate layers with 2-D image or feature input only. Code generation does not support layers with state properties (properties with attribute `State`).

The `checkLayer` function does not check that functions used by the layer are compatible with code generation. To check that functions used by the custom layer also support code generation, first use the **Code Generation Readiness** app. For more information, see "Check Code by Using the Code Generation Readiness Tool" (MATLAB Coder).

For an example showing how to define a custom layer that supports code generation, see "Define Custom Deep Learning Layer for Code Generation".

Data Types: `logical`

More About

Layer Input Sizes

For each layer, the valid input size and the observation dimension depend on the output of the previous layer.

Intermediate Layers

For intermediate layers (layers of type `nnet.layer.Layer`), the valid input size and the observation dimension depend on the type of data input to the layer.

- For layers with a single input, specify `validInputSize` as a vector of integers corresponding to the dimensions of the input data.
- For layers with multiple inputs, specify `validInputSize` as a cell array of vectors, where each vector corresponds to a layer input and the elements of the vectors correspond to the dimensions of the corresponding input data.

For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4

Layer Input	Input Size	Observation Dimension
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For example, for 2-D image classification problems, set `validInputSize` to `[h w c]`, where `h`, `w`, and `c` correspond to the height, width, and number of channels of the images, respectively, and `ObservationDimension` to `4`.

Code generation supports intermediate layers with 2-D image input only.

Output Layers

For output layers (layers of type `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`), set `validInputSize` to the typical size of a single input observation `Y` to the layer.

For classification problems, the valid input size and the observation dimension of `Y` 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

Classification Task	Input Size	Observation Dimension
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

For example, for 2-D image classification problems, set `validInputSize` to `[1 1 K]`, where `K` is the number of classes, and `ObservationDimension` to `4`.

For regression problems, the dimensions of `Y` also depend on the type of problem. The following table describes the dimensions of `Y`.

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, for 2-D image regression problems, set `validInputSize` to `[1 1 R]`, where `R` is the number of responses, and `ObservationDimension` to `4`.

Algorithms

List of Tests

The `checkLayer` function checks the validity of a custom layer by performing a series of tests, described in these tables. For more information on the tests used by `checkLayer`, see “Check Custom Layer Validity”.

Intermediate Layers

The `checkLayer` function uses these tests to check the validity of custom intermediate layers (layers of type `nnet.layer.Layer`).

Test	Description
`functionSyntaxesAreCorrect`	The syntaxes of the layer functions are correctly defined.
`predictDoesNotError`	`predict` function does not error.
`forwardDoesNotError`	When specified, the `forward` function does not error.
`forwardPredictAreConsistentInSize`	When `forward` is specified, `forward` and `predict` output values of the same size.
`backwardDoesNotError`	When specified, `backward` does not error.
`backwardIsConsistentInSize`	When `backward` is specified, the outputs of `backward` are consistent in size: • The derivatives with respect to each input are the same size as the corresponding input. • The derivatives with respect to each learnable parameter are the same size as the corresponding learnable parameter.
`predictIsConsistentInType`	The outputs of `predict` are consistent in type with the inputs.
`forwardIsConsistentInType`	When `forward` is specified, the outputs of `forward` are consistent in type with the inputs.
`backwardIsConsistentInType`	When `backward` is specified, the outputs of `backward` are consistent in type with the inputs.
`gradientsAreNumericallyCorrect`	When `backward` is specified, the gradients computed in `backward` are consistent with the numerical gradients.
`backwardPropagationDoesNotError`	When `backward` is not specified, the derivatives can be computed using automatic differentiation.
`predictReturnsValidStates`	For layers with state properties, the `predict` function returns valid states.
`forwardReturnsValidStates`	For layers with state properties, the `forward` function, if specified, returns valid states.

Test	Description
`resetStateDoesNotError`	For layers with state properties, the `resetState` function, if specified, does not error and resets the states to valid states.
`codegenPragmaDefinedInClassDef`	The pragma `"%#codegen"` for code generation is specified in class file.
`checkForSupportedLayerPropertiesForCodegen`	The layer properties support code generation.
`predictIsValidForCodeGeneration`	`predict` is valid for code generation.
`doesNotHaveStateProperties`	For code generation, the layer does not have state properties.
`supportedFunctionLayer`	For code generation, the layer is not a `FunctionLayer` object.

Some tests run multiple times. These tests also check different data types and for GPU compatibility:

- `predictIsConsistentInType`
- `forwardIsConsistentInType`
- `backwardIsConsistentInType`

To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

Output Layers

The `checkLayer` function uses these tests to check the validity of custom output layers (layers of type `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`).

Test	Description
`forwardLossDoesNotError`	`forwardLoss` does not error.
`backwardLossDoesNotError`	`backwardLoss` does not error.
`forwardLossIsScalar`	The output of `forwardLoss` is scalar.
`backwardLossIsConsistentInSize`	When `backwardLoss` is specified, the output of `backwardLoss` is consistent in size: `dLdY` is the same size as the predictions `Y`.
`forwardLossIsConsistentInType`	The output of `forwardLoss` is consistent in type: `loss` is the same type as the predictions `Y`.
`backwardLossIsConsistentInType`	When `backwardLoss` is specified, the output of `backwardLoss` is consistent in type: `dLdY` must be the same type as the predictions `Y`.
`gradientsAreNumericallyCorrect`	When `backwardLoss` is specified, the gradients computed in `backwardLoss` are numerically correct.
`backwardPropagationDoesNotError`	When `backwardLoss` is not specified, the derivatives can be computed using automatic differentiation.

The `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` tests also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

See Also

`trainNetwork` | `trainingOptions` | `analyzeNetwork`

Topics

“Check Custom Layer Validity”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Define Custom Deep Learning Layer with Multiple Inputs”
“Define Custom Classification Output Layer”
“Define Custom Regression Output Layer”
“Define Custom Deep Learning Layer for Code Generation”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2018a

classificationLayer

Classification output layer

Syntax

```
layer = classificationLayer
layer = classificationLayer(Name,Value)
```

Description

A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.

The layer infers the number of classes from the output size of the previous layer. For example, to specify the number of classes K of the network, you can include a fully connected layer with output size K and a softmax layer before the classification layer.

`layer = classificationLayer` creates a classification layer.

`layer = classificationLayer(Name,Value)` sets the optional `Name`, `ClassWeights`, and `Classes` properties using one or more name-value pairs. For example, `classificationLayer('Name','output')` creates a classification layer with the name `'output'`.

Examples

Create Classification Layer

Create a classification layer with the name `'output'`.

```
layer = classificationLayer('Name','output')
```

```
layer =
  ClassificationOutputLayer with properties:

            Name: 'output'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Include a classification output layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
```

```
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Weighted Classification Layer

Create a weighted classification layer for three classes with names "cat", "dog", and "fish", with weights 0.7, 0.2, and 0.1, respectively.

```
classes = ["cat" "dog" "fish"];
classWeights = [0.7 0.2 0.1];

layer = classificationLayer( ...
    'Classes',classes, ...
    'ClassWeights',classWeights)

layer =
  ClassificationOutputLayer with properties:

            Name: ''
         Classes: [cat    dog    fish]
    ClassWeights: [3x1 double]
      OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Include a weighted classification output layer in a Layer array.

```
numClasses = numel(classes);

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',classes,'ClassWeights',classWeights)]

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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         3 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   Class weighted crossentropyex with 'cat' and 2 other classe
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `classificationLayer('Name','output')` creates a classification layer with the name `'output'`

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`ClassWeights` — Class weights for weighted cross-entropy loss
`'none'` (default) | vector of positive numbers

Class weights for weighted cross-entropy loss, specified as a vector of positive numbers or `'none'`.

For vector class weights, each element represents the weight for the corresponding class in the `Classes` property. To specify a vector of class weights, you must also specify the classes using `'Classes'`.

If the `ClassWeights` property is `'none'`, then the layer applies unweighted cross-entropy loss.

`Classes` — Classes of the output layer
`'auto'` (default) | categorical vector | string array | cell array of character vectors

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)`.

Data Types: `char` | `categorical` | `string` | `cell`

Output Arguments

`layer` — Classification layer
`ClassificationOutputLayer` object

Classification layer, returned as a `ClassificationOutputLayer` object.

For information on concatenating layers to construct convolutional neural network architecture, see `Layer`.

More About

Classification Layer

A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.

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

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

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

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
`trainingOptions` | `trainNetwork` | `ClassificationOutputLayer` | `softmaxLayer` | `regressionLayer`

Topics
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2016a

ClassificationOutputLayer

Classification layer

Description

A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.

Creation

Create a classification layer using `classificationLayer`.

Properties

Classification Output

`ClassWeights` — Class weights for weighted cross-entropy loss
`'none'` (default) | vector of positive numbers

Class weights for weighted cross-entropy loss, specified as a vector of positive numbers or `'none'`.

For vector class weights, each element represents the weight for the corresponding class in the `Classes` property. To specify a vector of class weights, you must also specify the classes using `'Classes'`.

If the `ClassWeights` property is `'none'`, then the layer applies unweighted cross-entropy loss.

`Classes` — Classes of the output layer
`'auto'` (default) | categorical vector | string array | cell array of character vectors

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)`.

Data Types: `char` | `categorical` | `string` | `cell`

`OutputSize` — Size of the output
`'auto'` (default) | positive integer

This property is read-only.

Size of the output, specified as a positive integer. This value is the number of labels in the data. Before the training, the output size is set to `'auto'`.

`LossFunction` — Loss function for training
`'crossentropyex'`

This property is read-only.

Loss function for training, specified as `'crossentropyex'`, which stands for *Cross Entropy Function for k Mutually Exclusive Classes*.

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs

1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names

`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs

0 (default)

Number of outputs of the layer. The layer has no outputs.

Data Types: `double`

`OutputNames` — Output names

`{}` (default)

Output names of the layer. The layer has no outputs.

Data Types: `cell`

Examples

Create Classification Layer

Create a classification layer with the name `'output'`.

```
layer = classificationLayer('Name','output')
```

```
layer =
  ClassificationOutputLayer with properties:

            Name: 'output'
```

```
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Include a classification output layer in a `Layer` array.

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

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Weighted Classification Layer

Create a weighted classification layer for three classes with names "cat", "dog", and "fish", with weights 0.7, 0.2, and 0.1, respectively.

```
classes = ["cat" "dog" "fish"];
classWeights = [0.7 0.2 0.1];

layer = classificationLayer( ...
    'Classes',classes, ...
    'ClassWeights',classWeights)
```

```
layer =
  ClassificationOutputLayer with properties:

            Name: ''
         Classes: [cat    dog    fish]
    ClassWeights: [3x1 double]
      OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Include a weighted classification output layer in a Layer array.

```
numClasses = numel(classes);

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',classes,'ClassWeights',classWeights)]

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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         3 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   Class weighted crossentropyex with 'cat' and 2 other classe
```

More About

Classification Output Layer

A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.

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

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

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

Compatibility Considerations

`ClassNames` property will be removed

Not recommended starting in R2018b

`ClassNames` will be removed. Use `Classes` instead. To update your code, replace all instances of `ClassNames` with `Classes`. There are some differences between the properties that require additional updates to your code.

The `ClassNames` property of the output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

References

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

See Also

`regressionLayer` | `softmaxLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

classify

Classify data using a trained deep learning neural network

Syntax

```
YPred = classify(net,imds)
YPred = classify(net,ds)
YPred = classify(net,X)
YPred = classify(net,X1,...,XN)
YPred = classify(net,sequences)
YPred = classify(net,tbl)
YPred = classify( ___ ,Name,Value)
[YPred,scores] = classify( ___ )
```

Description

You can make predictions using a trained neural network for deep learning on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value pair argument.

For networks with multiple outputs, use the `predict` and set the `'ReturnCategorical'` option to `true`.

`YPred = classify(net,imds)` predicts class labels for the images in the image datastore `imds` using the trained network `net`.

`YPred = classify(net,ds)` predicts class labels for the data in the datastore `ds`.

`YPred = classify(net,X)` predicts class labels for the image or feature data specified by the numeric array `X`.

`YPred = classify(net,X1,...,XN)` predicts class labels for the data in the numeric arrays `X1`, ..., `XN` for the mutli-input network `net`. The input `Xi` corresponds to the network input `net.InputNames(i)`.

`YPred = classify(net,sequences)` predicts class labels for the time series or sequence data in `sequences` for the recurrent network (for example, an LSTM or GRU network) `net`.

`YPred = classify(net,tbl)` predicts class labels for the data in the table `tbl`.

`YPred = classify( ___ ,Name,Value)` predicts class labels with additional options specified by one or more name-value pair arguments using any of the previous syntaxes.

`[YPred,scores] = classify( ___ )` also returns the classification scores corresponding to the class labels using any of the previous syntaxes.

Tip 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, respectively.

Examples

Classify Images Using Trained ConvNet

Load the sample data.

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

`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.

Construct the convolutional neural network architecture.

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

Set the options to default settings for the stochastic gradient descent with momentum.

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

Train the network.

```
rng('default')
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 |       10.16% |       2.3195 |          0.0100 |
|       2 |          50 |       00:00:05 |       50.78% |       1.7102 |          0.0100 |
|       3 |         100 |       00:00:09 |       63.28% |       1.1632 |          0.0100 |
|       4 |         150 |       00:00:13 |       60.16% |       1.0859 |          0.0100 |
|       6 |         200 |       00:00:18 |       68.75% |       0.8996 |          0.0100 |
|       7 |         250 |       00:00:22 |       76.56% |       0.7919 |          0.0100 |
|       8 |         300 |       00:00:27 |       73.44% |       0.8411 |          0.0100 |
|       9 |         350 |       00:00:31 |       81.25% |       0.5514 |          0.0100 |
|      11 |         400 |       00:00:35 |       90.62% |       0.4744 |          0.0100 |
|      12 |         450 |       00:00:40 |       92.19% |       0.3614 |          0.0100 |
|      13 |         500 |       00:00:44 |       94.53% |       0.3159 |          0.0100 |
|      15 |         550 |       00:00:48 |       96.09% |       0.2543 |          0.0100 |
|      16 |         600 |       00:00:53 |       92.19% |       0.2765 |          0.0100 |
|      17 |         650 |       00:00:59 |       95.31% |       0.2461 |          0.0100 |
```

```
|      18 |         700 |       00:01:04 |       99.22% |       0.1418 |          0.0100 |
|      20 |         750 |       00:01:11 |       98.44% |       0.1000 |          0.0100 |
|      21 |         800 |       00:01:18 |       98.44% |       0.1448 |          0.0100 |
|      22 |         850 |       00:01:24 |       98.44% |       0.0989 |          0.0100 |
|      24 |         900 |       00:01:31 |       96.88% |       0.1316 |          0.0100 |
|      25 |         950 |       00:01:38 |      100.00% |       0.0859 |          0.0100 |
|      26 |        1000 |       00:01:45 |      100.00% |       0.0701 |          0.0100 |
|      27 |        1050 |       00:01:50 |      100.00% |       0.0759 |          0.0100 |
|      29 |        1100 |       00:01:53 |       99.22% |       0.0663 |          0.0100 |
|      30 |        1150 |       00:01:57 |       98.44% |       0.0775 |          0.0100 |
|      30 |        1170 |       00:01:59 |       99.22% |       0.0732 |          0.0100 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Run the trained network on a test set.

```
[XTest,YTest]= digitTest4DArrayData;
YPred = classify(net,XTest);
```

Display the first 10 images in the test data and compare to the classification from `classify`.

```
[YTest(1:10,:) YPred(1:10,:)]
```

```
ans = 10x2 categorical
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
```

The results from `classify` match the true digits for the first ten images.

Calculate the accuracy over all test data.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9820
```

Classify Sequences Using a Trained LSTM Network

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;
```

Classify the test data.

```
YPred = classify(net,XTest);
```

View the labels of the first 10 sequences with their predicted labels.

```
[YTest(1:10) YPred(1:10)]
```

```
ans = 10x2 categorical
     1      1
     1      1
     1      1
     1      1
     1      1
     1      1
     1      1
     1      1
     1      1
     1      1
```

Calculate the classification accuracy of the predictions.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.8595
```

Classify Feature Data Using Trained Network

Load the pretrained network `TransmissionCasingNet`. This network classifies the gear tooth condition of a transmission system given a mixture of numeric sensor readings, statistics, and categorical inputs.

```
load TransmissionCasingNet.mat
```

View the network architecture.

```
net.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'input'         Feature Input           22 features with 'zscore' normalization
```

```
     2   'fc_1'          Fully Connected         50 fully connected layer
     3   'batchnorm'     Batch Normalization     Batch normalization with 50 channels
     4   'relu'          ReLU                    ReLU
     5   'fc_2'          Fully Connected         2 fully connected layer
     6   'softmax'       Softmax                 softmax
     7   'classoutput'   Classification Output   crossentropyex with classes 'No Tooth Fault' and
```

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

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

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

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

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

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

Loop over the categorical input variables. For each variable:

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

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

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

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

View the first few rows of the table.

```
head(tbl)
```

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

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

```
    1.0459    1.0257    1.4402    0.98047    2.8157    3.6314    -0.035405    2.2
```

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

```
YPred = classify(net,tbl(:,1:end-1));
```

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

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

```
accuracy = 0.9952
```

Input Arguments

`net` — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

`imds` — Image datastore

`ImageDatastore` object

Image datastore, specified as an `ImageDatastore` object.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

Tip Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.

`ds` — Datastore

datastore

Datastore for out-of-memory data and preprocessing. The datastore must return data in a table or a cell array. The format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output

Single input	Table or cell array, where the first column specifies the predictors.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom datastores must output tables.
		`data = read(ds)`
		
		`data =`
		
		`  4×1 table`
		
		`  Predictors`
		`  ____________________`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

		`data = read(ds)`
		
		`data =`
		
		`  4×1 cell array`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

Multiple input	Cell array with at least `numInputs` columns, where `numInputs` is the number of network inputs.
	
	The first `numInputs` columns specify the predictors for each input.
	
	The order of inputs is given by the `InputNames` property of the network.
		`data = read(ds)`
		
		`data =`
		
		`  4×2 cell array`
		
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`

The format of the predictors depend on the type of data.

Data	Format of Predictors
2-D image	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the image, respectively.
3-D image	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the image, respectively.
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.

Data	Format of Predictors
1-D image sequence	*h*-by-*c*-by-*s* array, where *h* and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
Features	*c*-by-1 column vector, where *c* is the number of features.

For more information, see “Datastores for Deep Learning”.

X — Image or feature data

numeric array

Image or feature data, specified as a numeric array. The size of the array depends on the type of input:

Input	Description
2-D images	A *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
3-D images	A *h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.
Features	A *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

If the array contains `NaN`s, then they are propagated through the network.

For networks with multiple inputs, you can specify multiple arrays `X1`, ..., `XN`, where `N` is the number of network inputs and the input `Xi` corresponds to the network input `net.InputNames(i)`.

sequences — Sequence or time series data

cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

`tbl` — Table of image or feature data
`table`

Table of image or feature data. Each row in the table corresponds to an observation.

The arrangement of predictors in the table columns depend on the type of input data.

Input	Predictors
Image data	• Absolute or relative file path to an image, specified as a character vector in a single column • Image specified as a 3-D numeric array Specify predictors in a single column.
Feature data	Numeric scalar. Specify predictors in the first `numFeatures` columns of the table, where `numFeatures` is the number of features of the input data.

This argument supports networks with a single input only.

Data Types: `table`

Name-Value Pair Arguments

Specify optional comma-separated pair of `Name,Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`).

Example: `'MiniBatchSize','256'` specifies the mini-batch size as 256.

`MiniBatchSize` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

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, respectively.

Example: `'MiniBatchSize',256`

`Acceleration` — Performance optimization

`'auto'` (default) | `'mex'` | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'mex'` — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'none'` — Disable all acceleration.

The default option is `'auto'`. If `'auto'` is specified, MATLAB will apply a number of compatible optimizations. If you use the `'auto'` option, MATLAB does not ever generate a MEX function.

Using the `'Acceleration'` options `'auto'` and `'mex'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The `'mex'` option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The `'mex'` option is only available when you are using a GPU. MEX acceleration supports single GPU execution using the name-value option `'ExecutionEvironment','gpu'` only.

To use the `'mex'` option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The `'mex'` option does not support all layers. For a list of supported layers, see “Supported Layers” (GPU Coder). Only networks with an `imageInputLayer` are supported.

You cannot use MATLAB Compiler to deploy your network when using the `'mex'` option.

Example: `'Acceleration','mex'`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following:

- `'auto'` — Use a GPU if one is available; otherwise, use the CPU.
- `'gpu'` — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'cpu'` — Use the CPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform computation. If the pool does not have GPUs, then computation takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

The `'multi-gpu'` and `'parallel'` options do not support recurrent neural networks (RNNs) containing `lstmLayer`, `bilstmLayer`, or `gruLayer` objects.

Example: `'ExecutionEnvironment','cpu'`

`SequenceLength` — Option to pad, truncate, or split input sequences
`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: `'SequenceLength','shortest'`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

Output Arguments

`YPred` — Predicted class labels
categorical vector | cell array of categorical vectors

Predicted class labels, returned as a categorical vector, or a cell array of categorical vectors. The format of `YPred` depends on the type of task.

The following table describes the format for classification tasks.

Task	Format
Image or feature classification	*N*-by-1 categorical vector of labels, where *N* is the number of observations.
Sequence-to-label classification	

Task	Format
Sequence-to-sequence classification	*N*-by-1 cell array of categorical sequences of labels, where *N* is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently. For sequence-to-sequence classification tasks with one observation, `sequences` can be a matrix. In this case, `YPred` is a categorical sequence of labels.

`scores` — Predicted class scores
matrix | cell array of matrices

Predicted scores or responses, returned as a matrix or a cell array of matrices. The format of `scores` depends on the type of task.

The following table describes the format of `scores`.

Task	Format
Image classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes
Sequence-to-label classification	
Feature classification	
Sequence-to-sequence classification	*N*-by-1 cell array of matrices, where *N* is the number of observations. The sequences are matrices with *K* rows, where *K* is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

For sequence-to-sequence classification tasks with one observation, `sequences` can be a matrix. In this case, `scores` is a matrix of predicted class scores.

For an example exploring classification scores, see “Classify Webcam Images Using Deep Learning”.

Algorithms

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

Alternatives

For networks with multiple outputs, use the `predict` function and set the `'ReturnCategorical'` option to `true`.

You can compute the predicted scores from a trained network using `predict`.

You can also compute the activations from a network layer using `activations`.

For sequence-to-label and sequence-to-sequence classification networks, you can make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - `[YPred,scores] = classify(net,X)`
  - `[YPred,scores] = classify(net,sequences)`
  - `[YPred,scores] = classify(__,Name,Value)`
- C++ code generation for the `classify` function is not supported for regression networks and networks with multiple outputs.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- If you use a GCC C/C++ compiler version 8.2 or above, you might get a `-Wstringop-overflow` warning.
- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:
  - `[YPred,scores] = classify(net,X)`
  - `[YPred,scores] = classify(net,sequences)`
  - `[YPred,scores] = classify(__,Name,Value)`
- GPU code generation for the `classify` function is not supported for regression networks and networks with multiple outputs.
- GPU code generation does not support `gpuArray` inputs to the `classify` function.
- The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences. The ARM `Compute Library` for GPU does not support recurrent networks.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- GPU code generation for the `classify` function supports inputs that are defined as half-precision floating point data types. For more information, see `half`.
- If you use a GCC C/C++ compiler version 8.2 or above, you might get a `-Wstringop-overflow` warning.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run computations in parallel, set the `'ExecutionEnvironment'` option to `'multi-gpu'` or `'parallel'`.

For details, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- When input data is a `gpuArray`, a cell array or table containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`predict` | `activations` | `classifyAndUpdateState` | `predictAndUpdateState`

Topics
“Classify Image Using GoogLeNet”
“Classify Webcam Images Using Deep Learning”

Introduced in R2016a

classifyAndUpdateState

Classify data using a trained recurrent neural network and update the network state

Syntax

```
[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences)
[updatedNet,YPred] = classifyAndUpdateState( ___ ,Name,Value)
[updatedNet,YPred,scores] = classifyAndUpdateState( ___ )
```

Description

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Specify the hardware requirements using the `'ExecutionEnvironment'` name-value pair argument.

`[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences)` classifies the data in `sequences` using the trained recurrent neural network `recNet` and updates the network state.

This function supports recurrent neural networks only. The input `recNet` must have at least one recurrent layer.

`[updatedNet,YPred] = classifyAndUpdateState( ___ ,Name,Value)` uses any of the arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. For example, `'MiniBatchSize',27` classifies data using mini-batches of size 27

“Classify and Update Network State” on page 1-255

`[updatedNet,YPred,scores] = classifyAndUpdateState( ___ )` uses any of the arguments in the previous syntaxes, returns a matrix of classification scores, and updates the network state.

Tip 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, respectively.

Examples

Classify and Update Network State

Classify data using a recurrent neural network and update the network state.

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

```
ans =
  5x1 Layer array with layers:

     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load the test data.

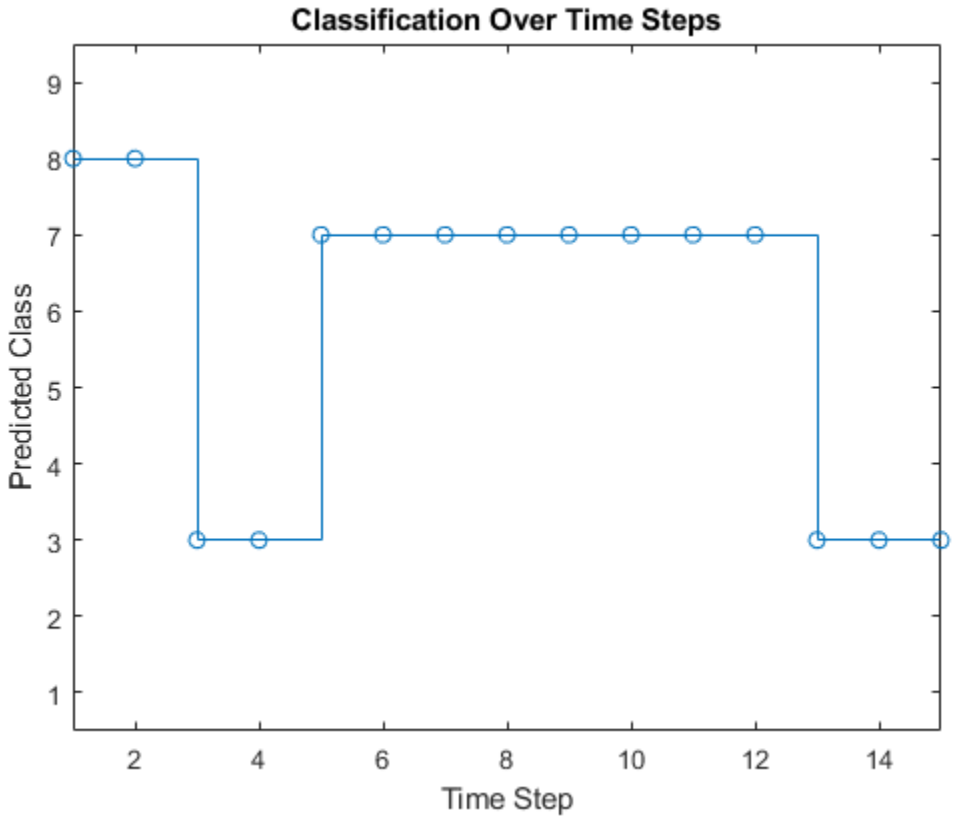
```
[XTest,YTest] = japaneseVowelsTestData;
```

Loop over the time steps in a sequence. Classify each time step and update the network state.

```
X = XTest{94};
numTimeSteps = size(X,2);
for i = 1:numTimeSteps
    v = X(:,i);
    [net,label,score] = classifyAndUpdateState(net,v);
    labels(i) = label;
end
```

Plot the predicted labels in a stair plot. The plot shows how the predictions change between time steps.

```
figure
stairs(labels, '-o')
xlim([1 numTimeSteps])
xlabel("Time Step")
ylabel("Predicted Class")
title("Classification Over Time Steps")
```

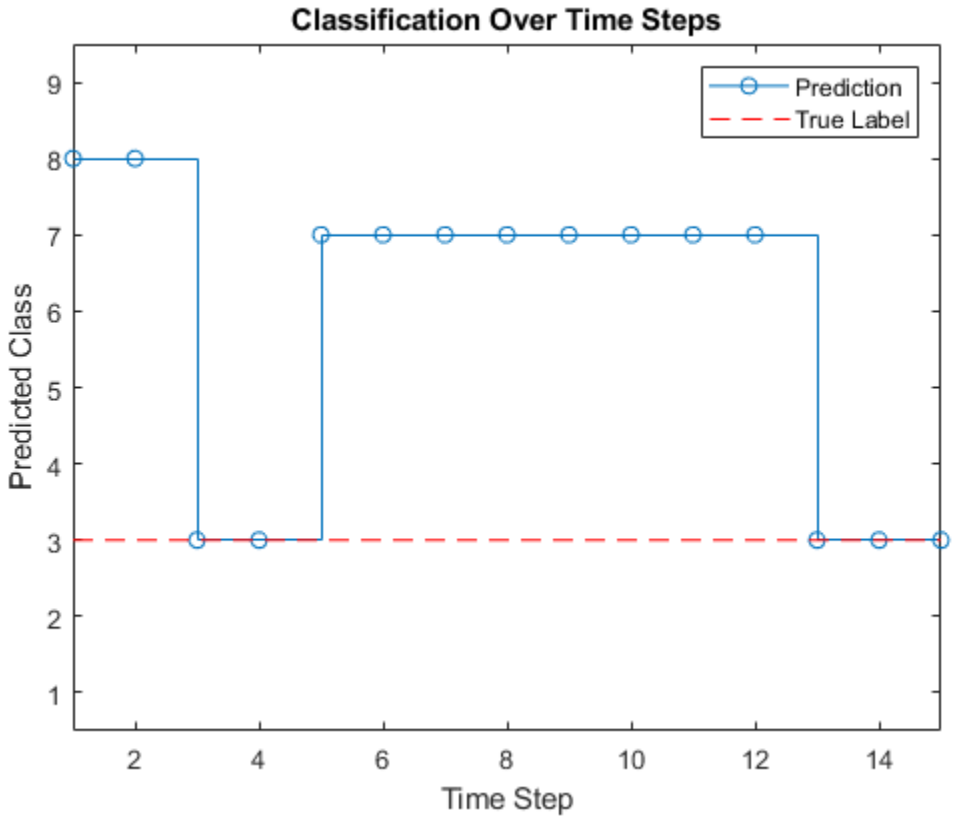


Compare the predictions with the true label. Plot a horizontal line showing the true label of the observation.

```
trueLabel = YTest(94)
```

```
trueLabel = categorical
     3
```

```
hold on
line([1 numTimeSteps],[trueLabel trueLabel], ...
    'Color','red', ...
    'LineStyle','--')
legend(["Prediction" "True Label"])
```



Input Arguments

`recNet` — Trained recurrent neural network

`SeriesNetwork` object | `DAGNetwork` object

Trained recurrent neural network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function.

`recNet` is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network).

`sequences` — Sequence or time series data

cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[updatedNet, YPred] = classifyAndUpdateState(recNet,C,'MiniBatchSize',27)` classifies data using mini-batches of size 27.

`MiniBatchSize` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

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, respectively.

Example: `'MiniBatchSize',256`

`Acceleration` — Performance optimization

`'auto'` (default) | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'none'` — Disable all acceleration.

The default option is `'auto'`.

Using the `'Acceleration'` option `'auto'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

Example: `'Acceleration','auto'`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following:

- `'auto'` — Use a GPU if one is available; otherwise, use the CPU.
- `'gpu'` — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'cpu'` — Use the CPU.

Example: `'ExecutionEnvironment','cpu'`

`SequenceLength` — Option to pad, truncate, or split input sequences
`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: `'SequenceLength','shortest'`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

Output Arguments

`updatedNet` — Updated network
`SeriesNetwork` object | `DAGNetwork` object

Updated network. `updatedNet` is the same type of network as the input network.

`YPred` — Predicted class labels
categorical vector | cell array of categorical vectors

Predicted class labels, returned as a categorical vector, or a cell array of categorical vectors. The format of `YPred` depends on the type of problem.

The following table describes the format of `YPred`.

Task	Format
Sequence-to-label classification	*N*-by-1 categorical vector of labels, where *N* is the number of observations.
Sequence-to-sequence classification	*N*-by-1 cell array of categorical sequences of labels, where *N* is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently. For sequence-to-sequence classification problems with one observation, `sequences` can be a matrix. In this case, `YPred` is a categorical sequence of labels.

`scores` — Predicted class scores
matrix | cell array of matrices

Predicted class scores, returned as a matrix or a cell array of matrices. The format of `scores` depends on the type of problem.

The following table describes the format of `scores`.

Task	Format
Sequence-to-label classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.
Sequence-to-sequence classification	*N*-by-1 cell array of matrices, where *N* is the number of observations. The sequences are matrices with *K* rows, where *K* is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

For sequence-to-sequence classification problems with one observation, `sequences` can be a matrix. In this case, `scores` is a matrix of predicted class scores.

Algorithms

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - `[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences)`
  - `[updatedNet,YPred] = classifyAndUpdateState(__,Name,Value)`
  - `[updatedNet,YPred,scores] = classifyAndUpdateState(__)`
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.

- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:
  - `[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences)`
  - `[updatedNet,YPred] = classifyAndUpdateState(__,Name,Value)`
  - `[updatedNet,YPred,scores] = classifyAndUpdateState(__)`
- GPU code generation for the `classifyAndUpdateState` function is only supported for recurrent neural networks and cuDNN target library.
- GPU code generation does not support `gpuArray` inputs to the `classifyAndUpdateState` function.
- The cuDNN library supports vector and 2-D image sequences.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- When input data is a `gpuArray`, a cell array containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `predictAndUpdateState` | `predict` | `classify` | `resetState`

Topics
“Sequence Classification Using Deep Learning”

“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”

Introduced in R2017b

clearCache

Clear accelerated deep learning function trace cache

Syntax

`clearCache(accfun)`

Description

`clearCache(accfun)` clears the trace cache of the `AcceleratedFunction` object `accfun`

Examples

Clear Cache of Accelerated Function

Load the `dlnetwork` object and class names from the MAT file `dlnetDigits.mat`.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Accelerate the model gradients function `modelGradients` listed at the end of the example.

```
fun = @modelGradients;
accfun = dlaccelerate(fun);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

View the properties of the accelerated function. Because the cache is empty, the `Occupancy` property is 0.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @modelGradients
           Enabled: 1
         CacheSize: 50
           HitRate: 0
         Occupancy: 0
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

The returned `AcceleratedFunction` object stores the traces of underlying function calls and reuses the cached result when the same input pattern reoccurs. To use the accelerated function in a custom training loop, replace calls to the model gradients function with calls to the accelerated function. You can invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Evaluate the accelerated model gradients function with random data using the `dlfeval` function.

```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');

T = categorical(classNames(randi(10,[128 1])));
T = onehotencode(T,2)';
dlT = dlarray(T,'CB');

[gradients,state,loss] = dlfeval(accfun,dlnet,dlX,dlT);
```

View the `Occupancy` property of the accelerated function. Because the function has been evaluated, the cache is nonempty.

```
accfun.Occupancy
```

```
ans = 2
```

Clear the cache using the `clearCache` function.

```
clearCache(accfun)
```

View the `Occupancy` property of the accelerated function. Because the cache has been cleared, the cache is empty.

```
accfun.Occupancy
```

```
ans = 0
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `dlT` 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, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,flT)

[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,flT);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Input Arguments

`accfun` — Accelerated function

`AcceleratedFunction` object

Accelerated function, specified as an `AcceleratedFunction` object.

See Also

`dlaccelerate` | `AcceleratedFunction` | `dlarray` | `dlgradient` | `dlfeval`

Topics

“Deep Learning Function Acceleration for Custom Training Loops”

“Accelerate Custom Training Loop Functions”
“Check Accelerated Deep Learning Function Outputs”
“Evaluate Performance of Accelerated Deep Learning Function”

Introduced in R2021a

clippedReluLayer

Clipped Rectified Linear Unit (ReLU) layer

Description

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.

This operation is equivalent to:

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

This clipping prevents the output from becoming too large.

Creation

Syntax

```
layer = clippedReluLayer(ceiling)
layer = clippedReluLayer(ceiling,'Name',Name)
```

Description

`layer = clippedReluLayer(ceiling)` returns a clipped ReLU layer with the clipping ceiling equal to `ceiling`.

`layer = clippedReluLayer(ceiling,'Name',Name)` sets the optional `Name` property.

Properties

Clipped ReLU

`Ceiling` — Ceiling for input clipping
positive scalar

Ceiling for input clipping, specified as a positive scalar.

Example: `10`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Clipped ReLU Layer

Create a clipped ReLU layer with the name `'clip1'` and the clipping ceiling equal to 10.

```
layer = clippedReluLayer(10,'Name','clip1')
```

```
layer =
  ClippedReLULayer with properties:

       Name: 'clip1'

   Hyperparameters
```

```
    Ceiling: 10
```

Include a clipped ReLU layer in a `Layer` array.

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

```
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  0
     3   ''   Clipped ReLU            Clipped ReLU with ceiling 10
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

References

[1] Hannun, Awni, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, et al. "Deep speech: Scaling up end-to-end speech recognition." Preprint, submitted 17 Dec 2014. http://arxiv.org/abs/1412.5567

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

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

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2017b

concatenationLayer

Concatenation layer

Description

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.

Specify the number of inputs to the layer when you create it. The inputs have the names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer by using `connectLayers` or `disconnectLayers`.

Creation

Syntax

```
layer = concatenationLayer(dim,numInputs)
layer = concatenationLayer(dim,numInputs,'Name',name)
```

Description

`layer = concatenationLayer(dim,numInputs)` creates a concatenation layer that concatenates `numInputs` inputs along the specified dimension, `dim`. This function also sets the `Dim` and `NumInputs` properties.

`layer = concatenationLayer(dim,numInputs,'Name',name)` also sets the `Name` property.

Properties

Concatenation

`Dim` — Concatenation dimension
positive integer

Concatenation dimension, specified as a positive integer.

Example: 4

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer greater than or equal to 2.

The inputs have the names `'in1','in2',...,'inN'`, where `N` is `NumInputs`. For example, if `NumInputs` is 3, then the inputs have the names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer using the `connectLayers` or `disconnectLayers` functions.

`InputNames` — Input Names
`{'in1','in2',…,'inN'}` (default)

Input names, specified as `{'in1','in2',...,'inN'}`, where `N` is the number of inputs of the layer.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create and Connect Concatenation Layer

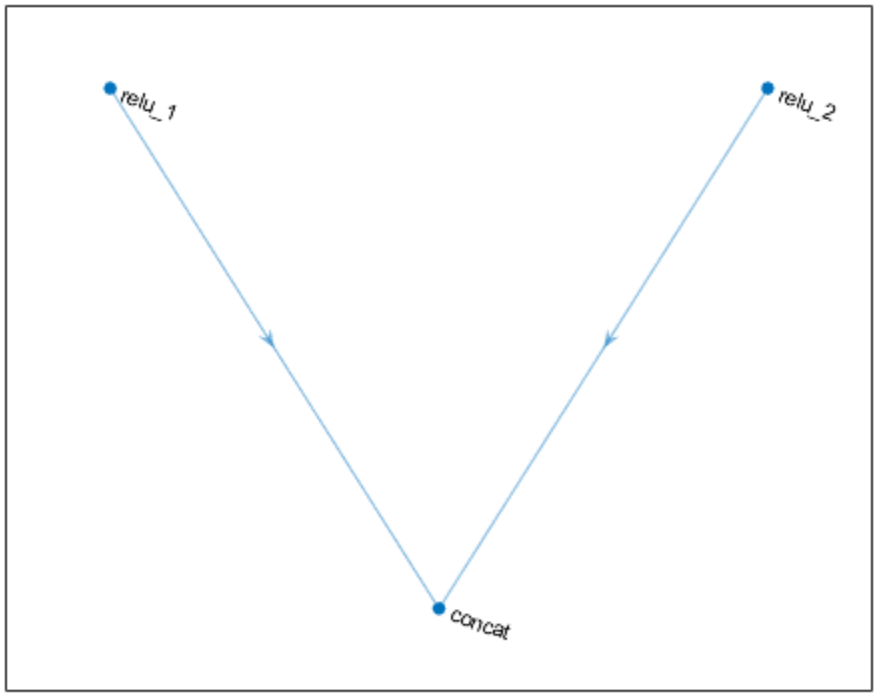
Create a concatenation layer that concatenates two inputs along the fourth dimension (channels). Name the concatenation layer `'concat'`.

```
concat = concatenationLayer(4,2,'Name','concat')
```

```
concat =
  ConcatenationLayer with properties:

         Name: 'concat'
          Dim: 4
    NumInputs: 2
   InputNames: {'in1'  'in2'}
```

Create two ReLU layers and connect them to the concatenation layer. The concatenation layer concatenates the outputs from the ReLU layers.

```
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph();
lgraph = addLayers(lgraph, relu_1);
lgraph = addLayers(lgraph, relu_2);
lgraph = addLayers(lgraph, concat);

lgraph = connectLayers(lgraph, 'relu_1', 'concat/in1');
lgraph = connectLayers(lgraph, 'relu_2', 'concat/in2');
plot(lgraph)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `layerGraph` | `additionLayer` | `connectLayers` | `disconnectLayers`

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”

“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”

Introduced in R2019a

confusionchart

Create confusion matrix chart for classification problem

Syntax

```
confusionchart(trueLabels,predictedLabels)
confusionchart(m)
confusionchart(m,classLabels)
confusionchart(parent, ___ )
confusionchart( ___ ,Name,Value)
cm = confusionchart( ___ )
```

Description

`confusionchart(trueLabels,predictedLabels)` creates a confusion matrix chart from true labels `trueLabels` and predicted labels `predictedLabels` and returns a `ConfusionMatrixChart` object. The rows of the confusion matrix correspond to the true class and the columns correspond to the predicted class. Diagonal and off-diagonal cells correspond to correctly and incorrectly classified observations, respectively. Use `cm` to modify the confusion matrix chart after it is created. For a list of properties, see ConfusionMatrixChart Properties.

`confusionchart(m)` creates a confusion matrix chart from the numeric confusion matrix `m`. Use this syntax if you already have a numeric confusion matrix in the workspace.

`confusionchart(m,classLabels)` specifies class labels that appear along the *x*-axis and *y*-axis. Use this syntax if you already have a numeric confusion matrix and class labels in the workspace.

`confusionchart(parent, ___ )` creates the confusion chart in the figure, panel, or tab specified by `parent`.

`confusionchart( ___ ,Name,Value)` specifies additional `ConfusionMatrixChart` properties using one or more name-value pair arguments. Specify the properties after all other input arguments. For a list of properties, see ConfusionMatrixChart Properties.

`cm = confusionchart( ___ )` returns the `ConfusionMatrixChart` object. Use `cm` to modify properties of the chart after creating it. For a list of properties, see ConfusionMatrixChart Properties.

Examples

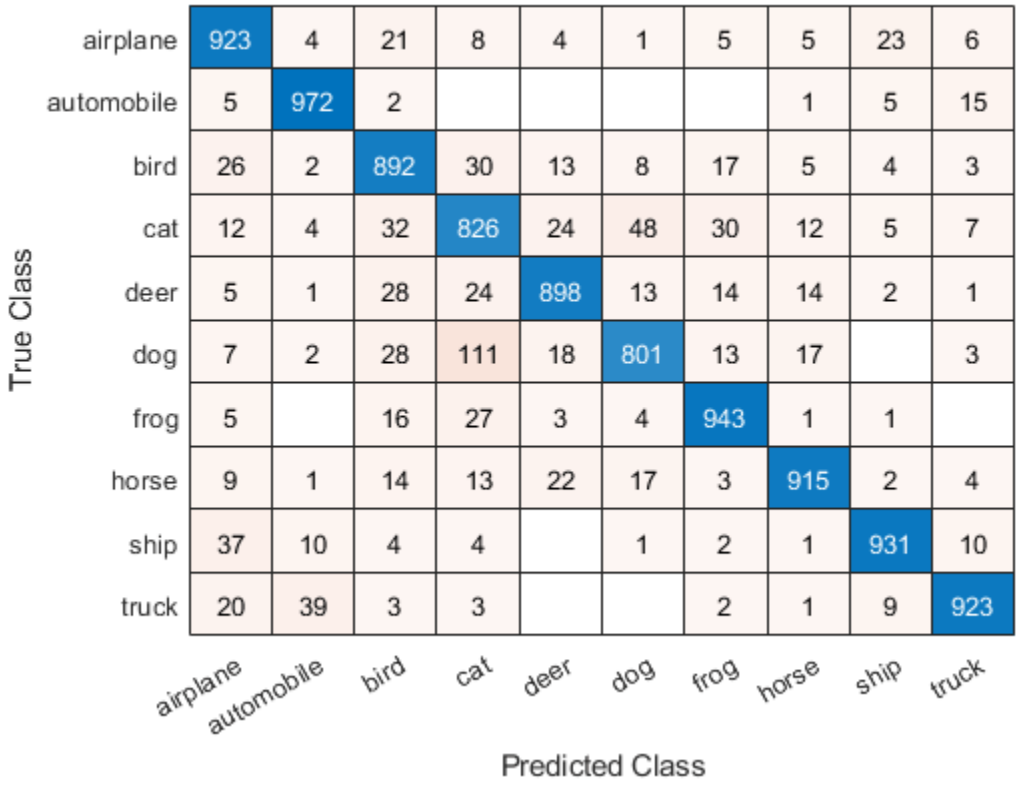
Create Confusion Matrix Chart

Load a sample of predicted and true labels for a classification problem. `trueLabels` is the true labels for an image classification problem and `predictedLabels` is the predictions of a convolutional neural network.

```
load('Cifar10Labels.mat','trueLabels','predictedLabels');
```

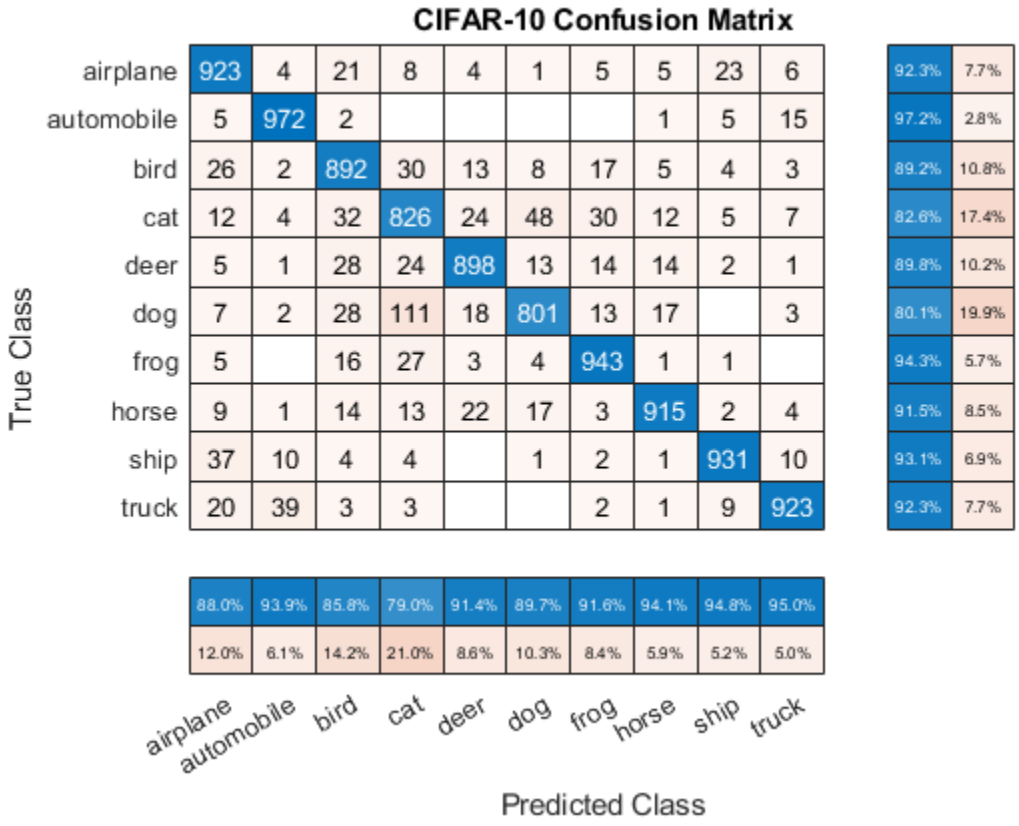
Create a confusion matrix chart.

```
figure
cm = confusionchart(trueLabels,predictedLabels);
```



Modify the appearance and behavior of the confusion matrix chart by changing property values. Add column and row summaries and a title. A column-normalized column summary displays the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. A row-normalized row summary displays the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class.

```
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
cm.Title = 'CIFAR-10 Confusion Matrix';
```



Create Confusion Matrix Chart from Numeric Confusion Matrix

You can use `confusionchart` to create a confusion matrix chart from a numeric confusion matrix.

Load a sample confusion matrix `m` and the associated class labels `classLabels`.

```
load('Cifar10ConfusionMat.mat','m','classLabels');
m
```

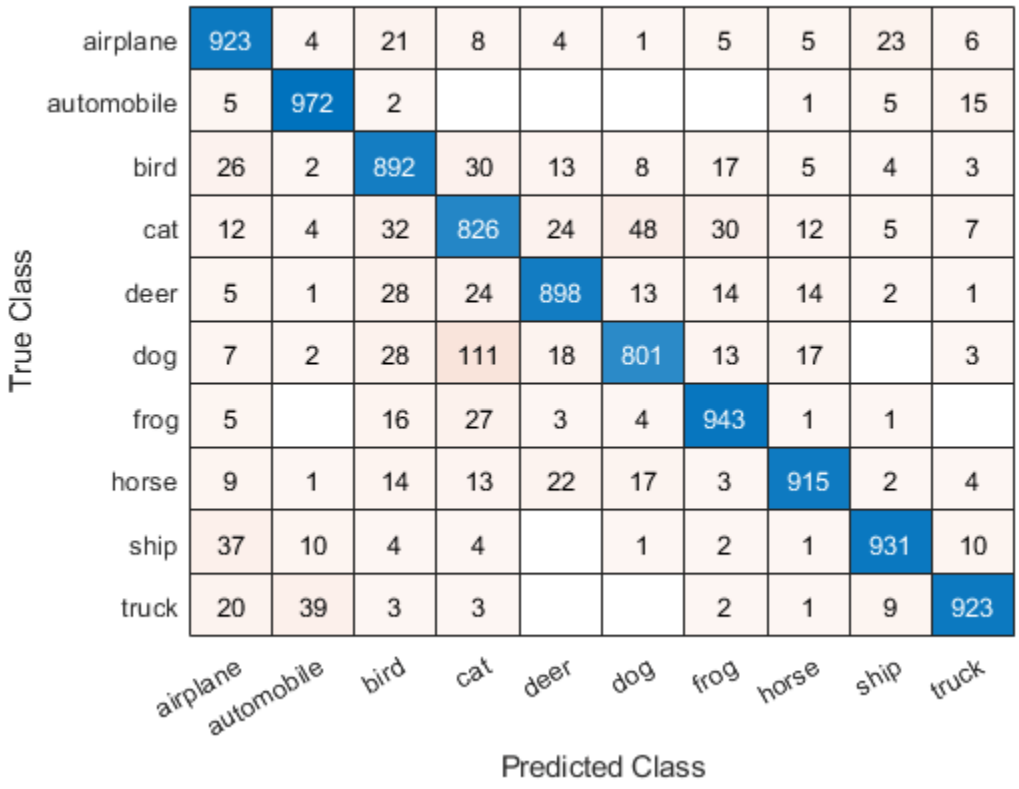
```
m = 10×10

   923     4    21     8     4     1     5     5    23     6
     5   972     2     0     0     0     0     1     5    15
    26     2   892    30    13     8    17     5     4     3
    12     4    32   826    24    48    30    12     5     7
     5     1    28    24   898    13    14    14     2     1
     7     2    28   111    18   801    13    17     0     3
     5     0    16    27     3     4   943     1     1     0
     9     1    14    13    22    17     3   915     2     4
    37    10     4     4     0     1     2     1   931    10
    20    39     3     3     0     0     2     1     9   923
```

```
classLabels
```

```
classLabels = 10x1 categorical
     airplane
     automobile
     bird
     cat
     deer
     dog
     frog
     horse
     ship
     truck
```

Create a confusion matrix chart from the numeric confusion matrix and the class labels.

```
cm = confusionchart(m,classLabels);
```

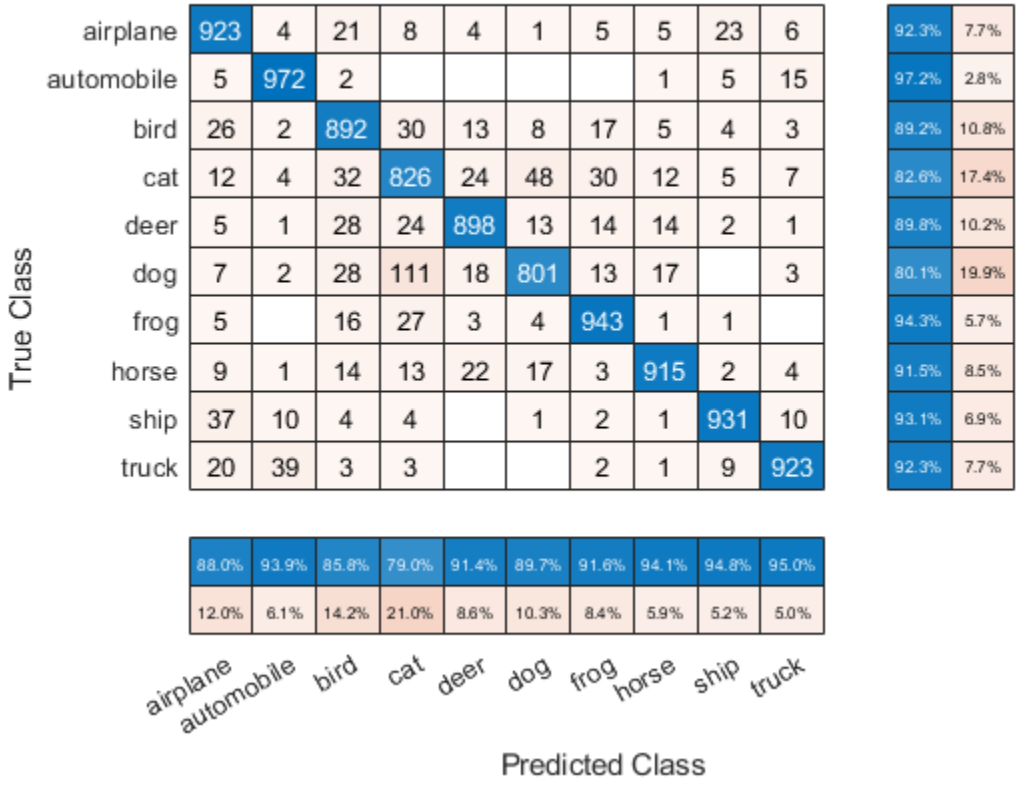


Sort Classes by Precision or Recall

Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network. Create a confusion matrix chart with column and row summaries

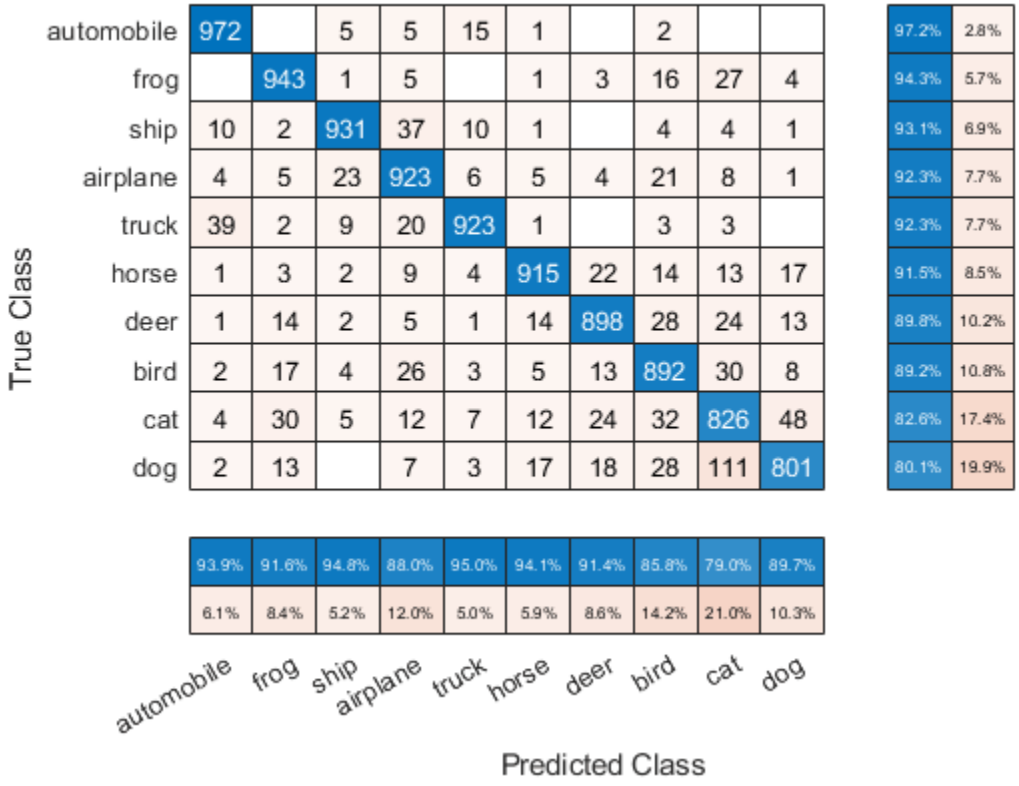
```
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
```

```
cm = confusionchart(trueLabels,predictedLabels, ...
    'ColumnSummary','column-normalized', ...
    'RowSummary','row-normalized');
```



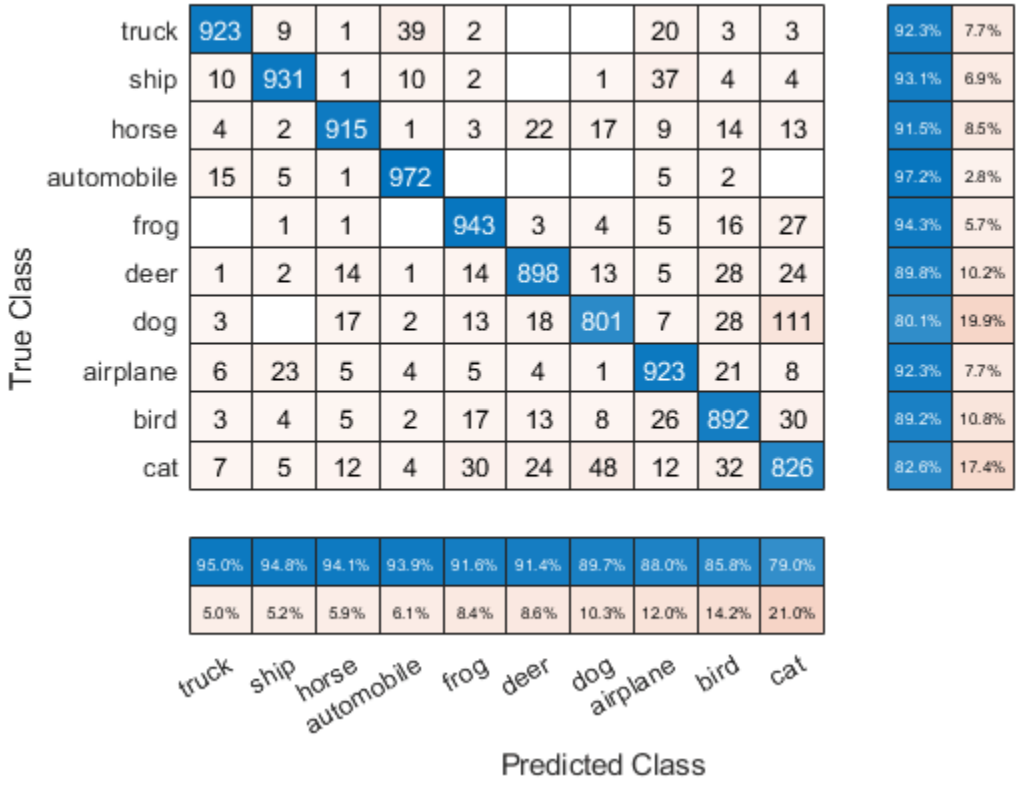
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the row summaries to the right are decreasing.

```
cm.Normalization = 'row-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```



To sort the classes by class-wise precision (positive predictive value), normalize the cell values across each column, that is, by the number of observations that have the same predicted class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the column summaries at the bottom are decreasing.

```
cm.Normalization = 'column-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```



Input Arguments

`trueLabels` — True labels of classification problem

categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

True labels of classification problem, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `trueLabels` is a vector, then each element corresponds to one observation. If `trueLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one observation.

`predictedLabels` — Predicted labels of classification problem

categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

Predicted labels of classification problem, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `predictedLabels` is a vector, then each element corresponds to one observation. If `predictedLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one observation.

`m` — Confusion matrix

matrix

Confusion matrix, specified as a matrix. `m` must be square and its elements must be positive integers. The element `m(i,j)` is the number of times an observation of the `i`th true class was predicted to be

of the `j`th class. Each colored cell of the confusion matrix chart corresponds to one element of the confusion matrix `m`.

`classLabels` — Class labels
categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

Class labels of the confusion matrix chart, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `classLabels` is a vector, then it must have the same number of elements as the confusion matrix has rows and columns. If `classLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one class.

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object | `GridLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, `TiledChartLayout`, or `GridLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `cm = confusionchart(trueLabels,predictedLabels,'Title','My Title Text','ColumnSummary','column-normalized')`

Note The properties listed here are only a subset. For a complete list, see ConfusionMatrixChart Properties.

`Title` — Title
`''` (default) | character vector | string scalar

Title of the confusion matrix chart, specified as a character vector or string scalar.

Example: `cm = confusionchart(__,'Title','My Title Text')`

Example: `cm.Title = 'My Title Text'`

`ColumnSummary` — Column summary
`'off'` (default) | `'absolute'` | `'column-normalized'` | `'total-normalized'`

Column summary of the confusion matrix chart, specified as one of the following:

Option	Description
`'off'`	Do not display a column summary.
`'absolute'`	Display the total number of correctly and incorrectly classified observations for each predicted class.

Option	Description
`'column-normalized'`	Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. The percentages of correctly classified observations can be thought of as class-wise precisions (or positive predictive values).
`'total-normalized'`	Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the total number of observations.

Example: `cm = confusionchart(__,'ColumnSummary','column-normalized')`

Example: `cm.ColumnSummary = 'column-normalized'`

`RowSummary` — Row summary

`'off'` (default) | `'absolute'` | `'row-normalized'` | `'total-normalized'`

Row summary of the confusion matrix chart, specified as one of the following:

Option	Description
`'off'`	Do not display a row summary.
`'absolute'`	Display the total number of correctly and incorrectly classified observations for each true class.
`'row-normalized'`	Display the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class. The percentages of correctly classified observations can be thought of as class-wise recalls (or true positive rates).
`'total-normalized'`	Display the number of correctly and incorrectly classified observations for each true class as percentages of the total number of observations.

Example: `cm = confusionchart(__,'RowSummary','row-normalized')`

Example: `cm.RowSummary = 'row-normalized'`

`Normalization` — Normalization of cell values

`'absolute'` (default) | `'column-normalized'` | `'row-normalized'` | `'total-normalized'`

Normalization of cell values, specified as one of the following:

Option	Description
`'absolute'`	Display the total number of observations in each cell.
`'column-normalized'`	Normalize each cell value by the number of observations that has the same predicted class.

Option	Description
`'row-normalized'`	Normalize each cell value by the number of observations that has the same true class.
`'total-normalized'`	Normalize each cell value by the total number of observations.

Modifying the normalization of cell values also affects the colors of the cells.

Example: `cm = confusionchart(__,'Normalization','total-normalized')`

Example: `cm.Normalization = 'total-normalized'`

Output Arguments

`cm` — Confusion matrix chart object
`ConfusionMatrixChart` object

`ConfusionMatrixChart` object, which is a standalone visualization on page 1-284. Use `cm` to set properties of the confusion matrix chart after creating it.

Limitations

- MATLAB code generation is not supported for `ConfusionMatrixChart` objects.

More About

Standalone Visualization

A standalone visualization is a chart designed for a special purpose that works independently from other charts. Unlike other charts such as `plot` and `surf`, a standalone visualization has a preconfigured axes object built into it, and some customizations are not available. A standalone visualization also has these characteristics:

- It cannot be combined with other graphics elements, such as lines, patches, or surfaces. Thus, the `hold` command is not supported.
- The `gca` function can return the chart object as the current axes.
- You can pass the chart object to many MATLAB functions that accept an axes object as an input argument. For example, you can pass the chart object to the `title` function.

Tips

- If you have one-hot (one-of-N) data, use `onehotdecode` to prepare your data for use with `confusionchart`. For example, suppose you have true labels `targets` and predicted labels `outputs`, with observations in columns. You can create a confusion matrix chart using

```
numClasses = size(targets,1);
trueLabels = onehotdecode(targets,1:numClasses,1);
predictedLabels = onehotdecode(outputs,1:numClasses,1);
confusionchart(trueLabels,predictedLabels)
```

- If you have Statistics and Machine Learning Toolbox, you can create a confusion matrix chart for tall arrays. For details, see `confusionchart` and “Confusion Matrix for Classification Using Tall Arrays” (Statistics and Machine Learning Toolbox).

See Also

Functions
`categorical` | `sortClasses` | `classify` | `confusionmat`

Properties
ConfusionMatrixChart Properties

Topics
“Deep Learning in MATLAB”

Introduced in R2018b

confusionmat

Compute confusion matrix for classification problem

Syntax

```
C = confusionmat(group,grouphat)
C = confusionmat(group,grouphat,'Order',grouporder)
[C,order] = confusionmat( ___ )
```

Description

`C = confusionmat(group,grouphat)` returns the confusion matrix `C` determined by the known and predicted groups in `group` and `grouphat`, respectively.

`C = confusionmat(group,grouphat,'Order',grouporder)` uses `grouporder` to order the rows and columns of `C`.

`[C,order] = confusionmat( ___ )` also returns the order of the rows and columns of `C` in the variable `order` using any of the input arguments in previous syntaxes.

Examples

Calculate Confusion Matrix

Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network.

```
load('Cifar10Labels.mat','trueLabels','predictedLabels');
```

Calculate the numeric confusion matrix. `order` is the order of the classes in the confusion matrix.

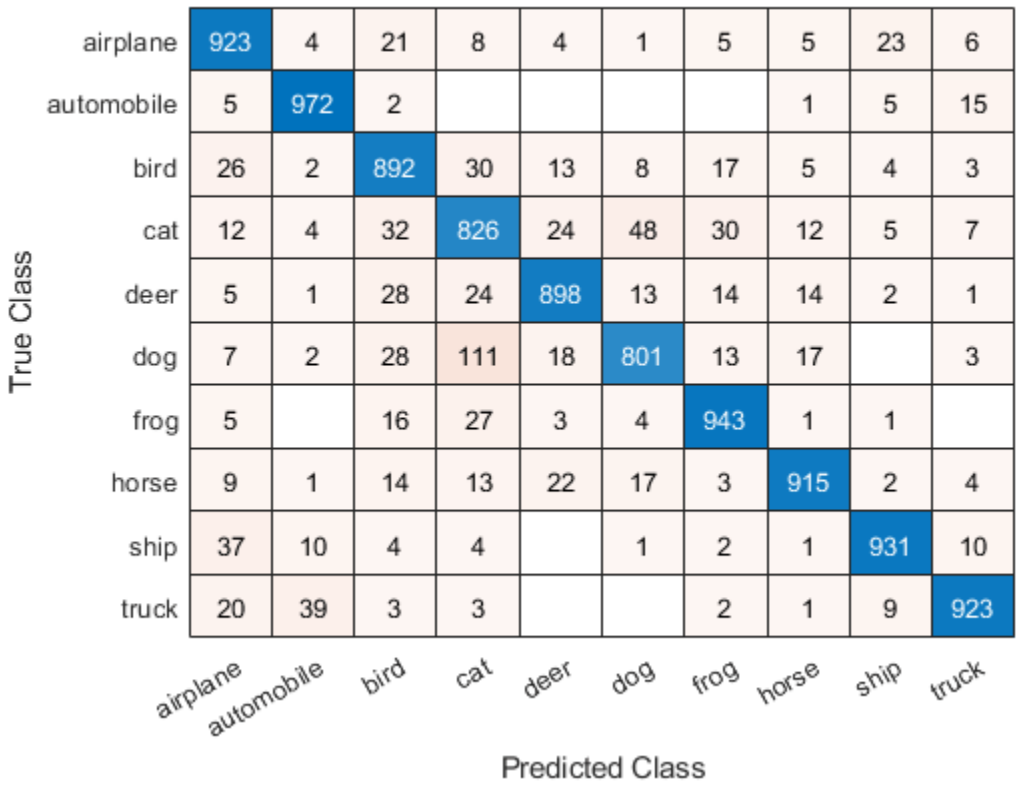
```
[m,order] = confusionmat(trueLabels,predictedLabels)
```

```
m = 10×10

   923     4    21     8     4     1     5     5    23     6
     5   972     2     0     0     0     0     1     5    15
    26     2   892    30    13     8    17     5     4     3
    12     4    32   826    24    48    30    12     5     7
     5     1    28    24   898    13    14    14     2     1
     7     2    28   111    18   801    13    17     0     3
     5     0    16    27     3     4   943     1     1     0
     9     1    14    13    22    17     3   915     2     4
    37    10     4     4     0     1     2     1   931    10
    20    39     3     3     0     0     2     1     9   923


order = 10x1 categorical
     airplane
     automobile
```

```
bird
cat
deer
dog
frog
horse
ship
truck
```

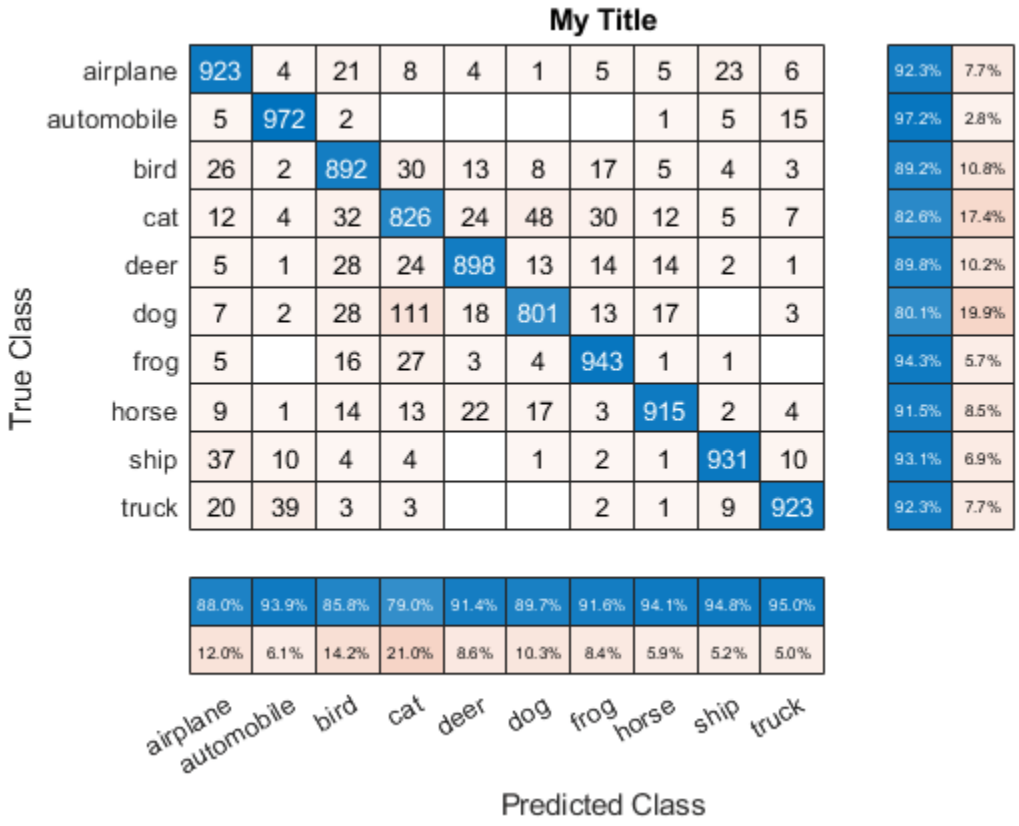
You can use `confusionchart` to plot a the confusion matrix as a confusion matrix chart.

```
figure
cm = confusionchart(m,order);
```



You do not need to calculate the confusion matrix first and then plot it. Instead, plot a confusion matrix chart directly from the true and predicted labels. You can also add column and row summaries and a title.

```
figure
cm = confusionchart(trueLabels,predictedLabels, ...
    'Title','My Title', ...
    'RowSummary','row-normalized', ...
    'ColumnSummary','column-normalized');
```



The `ConfusionMatrixChart` object stores the numeric confusion matrix in the `NormalizedValues` property and classes in the `ClassLabels` property.

```
cm.NormalizedValues
```

```
ans = 10×10

   923     4    21     8     4     1     5     5    23     6
     5   972     2     0     0     0     0     1     5    15
    26     2   892    30    13     8    17     5     4     3
    12     4    32   826    24    48    30    12     5     7
     5     1    28    24   898    13    14    14     2     1
     7     2    28   111    18   801    13    17     0     3
     5     0    16    27     3     4   943     1     1     0
     9     1    14    13    22    17     3   915     2     4
    37    10     4     4     0     1     2     1   931    10
    20    39     3     3     0     0     2     1     9   923
```

```
cm.ClassLabels
```

```
ans = 10x1 categorical
     airplane
     automobile
     bird
     cat
     deer
     dog
```

```
frog
horse
ship
truck
```

Input Arguments

`group` — Known groups

numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Known groups for categorizing observations, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

`group` is a grouping variable of the same type as `grouphat`. The `group` argument must have the same number of observations as `grouphat`, as described in “Grouping Variables” (Statistics and Machine Learning Toolbox). The `confusionmat` function treats character arrays and string arrays as cell arrays of character vectors. Additionally, `confusionmat` treats `NaN`, empty, and `'undefined'` values in `group` as missing values and does not count them as distinct groups or categories.

Example: `{'Male','Female','Female','Male','Female'}`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell` | `categorical`

`grouphat` — Predicted groups

numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Predicted groups for categorizing observations, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

`grouphat` is a grouping variable of the same type as `group`. The `grouphat` argument must have the same number of observations as `group`, as described in “Grouping Variables” (Statistics and Machine Learning Toolbox). The `confusionmat` function treats character arrays and string arrays as cell arrays of character vectors. Additionally, `confusionmat` treats `NaN`, empty, and `'undefined'` values in `grouphat` as missing values and does not count them as distinct groups or categories.

Example: `[1 0 0 1 0]`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell` | `categorical`

`grouporder` — Group order

numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Group order, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

`grouporder` is a grouping variable containing all the distinct elements in `group` and `grouphat`. Specify `grouporder` to define the order of the rows and columns of `C`. If `grouporder` contains elements that are not in `group` or `grouphat`, the corresponding entries in `C` are `0`.

By default, the group order depends on the data type of `s = [group;grouphat]`:

- For numeric and logical vectors, the order is the sorted order of `s`.
- For categorical vectors, the order is the order returned by `categories(s)`.
- For other data types, the order is the order of first appearance in `s`.

Example: `'order',{'setosa','versicolor','virginica'}`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell` | `categorical`

Output Arguments

`C` — Confusion matrix

matrix

Confusion matrix, returned as a square matrix with size equal to the total number of distinct elements in the `group` and `grouphat` arguments. `C(i,j)` is the count of observations known to be in group `i` but predicted to be in group `j`.

The rows and columns of `C` have identical ordering of the same group indices. By default, the group order depends on the data type of `s = [group;grouphat]`:

- For numeric and logical vectors, the order is the sorted order of `s`.
- For categorical vectors, the order is the order returned by `categories(s)`.
- For other data types, the order is the order of first appearance in `s`.

To change the order, specify `grouporder`,

The `confusionmat` function treats `NaN`, empty, and `'undefined'` values in the grouping variables as missing values and does not include them in the rows and columns of `C`.

`order` — Order of rows and columns

numeric vector | logical vector | categorical vector | cell array of character vectors

Order of rows and columns in `C`, returned as a numeric vector, logical vector, categorical vector, or cell array of character vectors. If `group` and `grouphat` are character arrays, string arrays, or cell arrays of character vectors, then the variable `order` is a cell array of character vectors. Otherwise, `order` is of the same type as `group` and `grouphat`.

Alternative Functionality

- Use `confusionchart` to calculate and plot a confusion matrix. Additionally, `confusionchart` displays summary statistics about your data and sorts the classes of the confusion matrix according to the class-wise precision (positive predictive value), class-wise recall (true positive rate), or total number of correctly classified observations.

See Also

`categories` | `classify` | `confusionchart`

Topics

“Deep Learning in MATLAB”

ConfusionMatrixChart Properties

Confusion matrix chart appearance and behavior

Description

`ConfusionMatrixChart` properties control the appearance and behavior of a `ConfusionMatrixChart` object. By changing property values, you can modify certain aspects of the confusion matrix chart. For example, you can add a title:

```
cm = confusionchart([1 3 5; 2 4 6; 11 7 3]);
cm.Title = 'My Confusion Matrix Title';
```

Properties

Labels

`Title` — Title

`''` (default) | character vector | string scalar

Title of the confusion matrix chart, specified as a character vector or string scalar.

Example: `cm = confusionchart(__,'Title','My Title Text')`

Example: `cm.Title = 'My Title Text'`

`XLabel` — Label for *x*-axis

`'Predicted class'` (default) | string scalar | character vector

Label for the *x*-axis, specified as a string scalar or character vector.

Example: `cm = confusionchart(__,'XLabel','My Label')`

Example: `cm.XLabel = 'My Label'`

`YLabel` — Label for *y*-axis

`'True class'` (default) | string scalar | character vector

Label for the *x*-axis, specified as a string scalar or character vector.

Example: `cm = confusionchart(__,'YLabel','My Label')`

Example: `cm.YLabel = 'My Label'`

`ClassLabels` — Class labels

categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

This property is read-only.

Class labels of the confusion matrix chart, stored as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector.

Row and Column Summaries

ColumnSummary — Column summary
`'off'` (default) | `'absolute'` | `'column-normalized'` | `'total-normalized'`

Column summary of the confusion matrix chart, specified as one of the following:

Option	Description
`'off'`	Do not display a column summary.
`'absolute'`	Display the total number of correctly and incorrectly classified observations for each predicted class.
`'column-normalized'`	Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. The percentages of correctly classified observations can be thought of as class-wise precisions (or positive predictive values).
`'total-normalized'`	Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the total number of observations.

Example: `cm = confusionchart(__,'ColumnSummary','column-normalized')`

Example: `cm.ColumnSummary = 'column-normalized'`

RowSummary — Row summary
`'off'` (default) | `'absolute'` | `'row-normalized'` | `'total-normalized'`

Row summary of the confusion matrix chart, specified as one of the following:

Option	Description
`'off'`	Do not display a row summary.
`'absolute'`	Display the total number of correctly and incorrectly classified observations for each true class.
`'row-normalized'`	Display the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class. The percentages of correctly classified observations can be thought of as class-wise recalls (or true positive rates).
`'total-normalized'`	Display the number of correctly and incorrectly classified observations for each true class as percentages of the total number of observations.

Example: `cm = confusionchart(__,'RowSummary','row-normalized')`

Example: `cm.RowSummary = 'row-normalized'`

Data

`NormalizedValues` — Values of the confusion matrix

numeric matrix

This property is read-only.

Values of the confusion matrix, stored as a numeric matrix. This property equals the values of the confusion matrix normalized using the method of the `Normalization` property. The software recalculates the normalized values of the confusion matrix each time you modify the `Normalization` property.

`Normalization` — Normalization of cell values

`'absolute'` (default) | `'column-normalized'` | `'row-normalized'` | `'total-normalized'`

Normalization of cell values, specified as one of the following:

Option	Description
`'absolute'`	Display the total number of observations in each cell.
`'column-normalized'`	Normalize each cell value by the number of observations that has the same predicted class.
`'row-normalized'`	Normalize each cell value by the number of observations that has the same true class.
`'total-normalized'`	Normalize each cell value by the total number of observations.

Modifying the normalization of cell values also affects the colors of the cells.

Example: `cm = confusionchart(__,'Normalization','total-normalized')`

Example: `cm.Normalization = 'total-normalized'`

Color and Styling

`GridVisible` — State of grid visibility

`'on'` (default) | on/off logical value

State of grid visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to true, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- `'on'` — Display grid lines between the chart cells.
- `'off'` — Do not display grid lines between the chart cells.

Example: `cm = confusionchart(__,'GridVisible','off')`

Example: `cm.GridVisible = 'off'`

`DiagonalColor` — Color for diagonal cells

`[0 0.4471 0.7412]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color for diagonal cells, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color of each diagonal cell is proportional to the cell value and the `DiagonalColor`

property, normalized to the largest cell value of the confusion matrix chart. Cells with positive values are colored with a minimum amount of color, proportional to the `DiagonalColor` property.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: `cm = confusionchart(__,'DiagonalColor','blue')`

Example: `cm.DiagonalColor = 'blue'`

`OffDiagonalColor` — Color for off-diagonal cells

`[0.8510 0.3255 0.0980]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color for off-diagonal cells, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color of each diagonal cell is proportional to the cell value and the `OffDiagonalColor` property, normalized to the largest cell value of the confusion matrix chart. Cells with positive values are colored with a minimum amount of color, proportional to the `OffDiagonalColor` property.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: `cm = confusionchart(__,'OffDiagonalColor','blue')`

Example: `cm.OffDiagonalColor = 'blue'`

`FontColor` — Text color for title, axis labels, and class labels

`[0.1500 0.1500 0.1500]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color for title, axis labels, and class labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: `cm = confusionchart(__,'FontColor','blue')`

Example: `cm.FontColor = 'blue'`

Font

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

Example: `cm = confusionchart(__,'FontName','Cambria')`

Example: `cm.FontName = 'Cambria'`

`FontSize` — Font size

positive scalar

Font size used for the title, axis labels, class labels, and cell labels, specified as a positive scalar. The default font depends on the specific operating system and locale.

The title and axis labels use a slightly larger font size (scaled up by 10%). If there is not enough room to display the cell labels within the cells, then the cell labels use a smaller font size. If the cell labels become too small, then they are hidden.

Example: `cm = confusionchart(__,'FontSize',12)`

Example: `cm.FontSize = 12`

Position

`PositionConstraint` — Position to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Outer size and position

`[0 0 1 1]` (default) | four-element vector

Outer size and position within the parent container (a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. The outer position includes the title, axis labels, and class labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the chart.
- The `width` and `height` elements are the chart dimensions, which include the chart cells, plus a margin for the surrounding text.

The default value of `[0 0 1 1]` is the whole interior of the container.

By default, the values are normalized to the container. To change the units, set the `Units` property.

Example: `cm = confusionchart(__,'OuterPosition',[0.1 0.1 0.8 0.8])`

Example: `cm.OuterPosition = [0.1 0.1 0.8 0.8]`

`InnerPosition` — Inner size and position
`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector

Inner size and position of the chart within the parent container (a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. The inner position does not include the title, axis labels, or class labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the chart.
- The `width` and `height` elements are the chart dimensions, which include only the chart cells.

Example: `cm = confusionchart(__,'InnerPosition',[0.1 0.1 0.8 0.8])`

Example: `cm.InnerPosition = [0.1 0.1 0.8 0.8]`

`Position` — Inner size and position
four-element vector

Inner size and position of the chart within the parent container (a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values:

`Units`	Description
`'normalized'`	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.

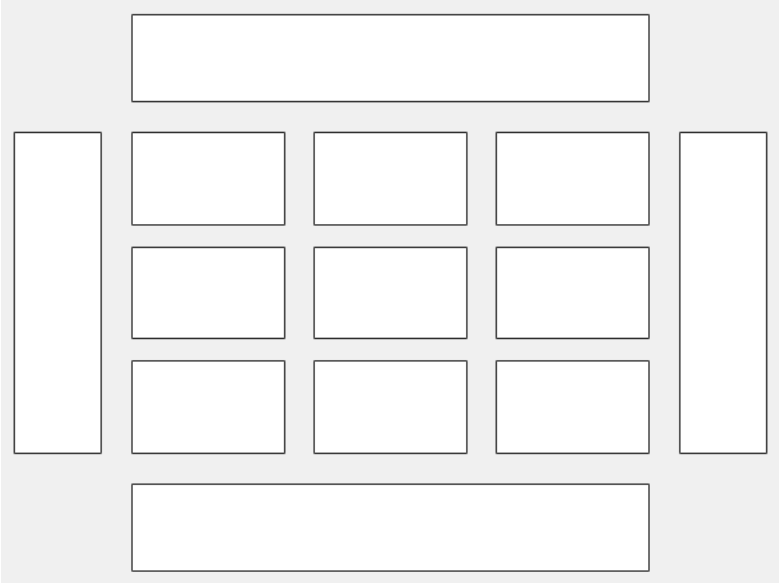
Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows® and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux® systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units for, such as `OuterPosition`.

Layout — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object | `GridLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` or `GridLayoutOptions` object. This property is useful when the chart is either in a tiled chart layout or a grid layout.

To position the chart within the grid of a tiled chart layout, set the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object. For example, consider a 3-by-3 tiled chart layout. The layout has a grid of tiles in the center, and four tiles along the outer edges. In practice, the grid is invisible and the outer tiles do not take up space until you populate them with axes or charts.



This code places the chart `c` in the third tile of the grid..

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans 2 rows and 3 columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

To place the chart in one of the surrounding tiles, specify the `Tile` property as `'north'`, `'south'`, `'east'`, or `'west'`. For example, setting the value to `'east'` places the chart in the tile to the right of the grid.

```
c.Layout.Tile = 'east';
```

To place the chart into a layout within an app, specify this property as a `GridLayoutOptions` object. For more information about working with grid layouts in apps, see `uigridlayout`.

If the chart is not a child of either a tiled chart layout or a grid layout (for example, if it is a child of a figure or panel) then this property is empty and has no effect.

`Visible` — State of visibility

`'on'` (default) | on/off logical value

State of visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to true, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- `'on'` — Display the chart.
- `'off'` — Hide the chart without deleting it. You still can access the properties of an invisible chart.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object | `GridLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, `TiledChartLayout`, or `GridLayout` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the chart object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the `HandleVisibility` to `'off'`.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but allows callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

See Also

Functions
`categorical` | `confusionchart` | `sortClasses`

Topics
“Deep Learning in MATLAB”

Introduced in R2018b

connectLayers

Connect layers in layer graph

Syntax

```
newlgraph = connectLayers(lgraph,s,d)
```

Description

`newlgraph = connectLayers(lgraph,s,d)` connects the source layer `s` to the destination layer `d` in the layer graph `lgraph`. The new layer graph, `newlgraph`, contains the same layers as `lgraph` and includes the new connection.

Examples

Create and Connect Addition Layer

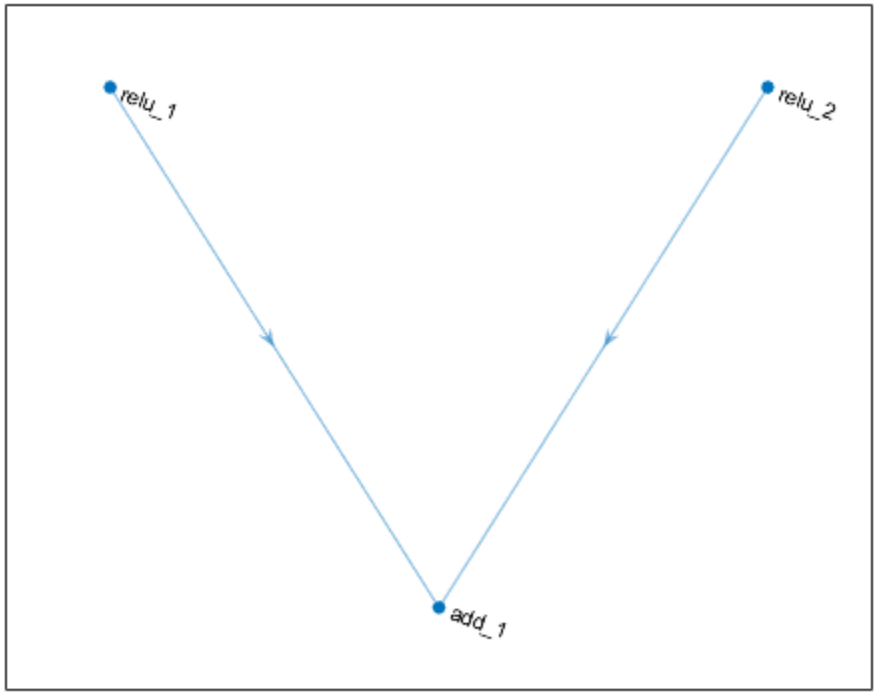
Create an addition layer with two inputs and the name `'add_1'`.

```
add = additionLayer(2,'Name','add_1')
```

```
add =
  AdditionLayer with properties:

          Name: 'add_1'
     NumInputs: 2
    InputNames: {'in1'  'in2'}
```

Create two ReLU layers and connect them to the addition layer. The addition layer sums the outputs from the ReLU layers.

```
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph;
lgraph = addLayers(lgraph,relu_1);
lgraph = addLayers(lgraph,relu_2);
lgraph = addLayers(lgraph,add);

lgraph = connectLayers(lgraph,'relu_1','add_1/in1');
lgraph = connectLayers(lgraph,'relu_2','add_1/in2');

plot(lgraph)
```



Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

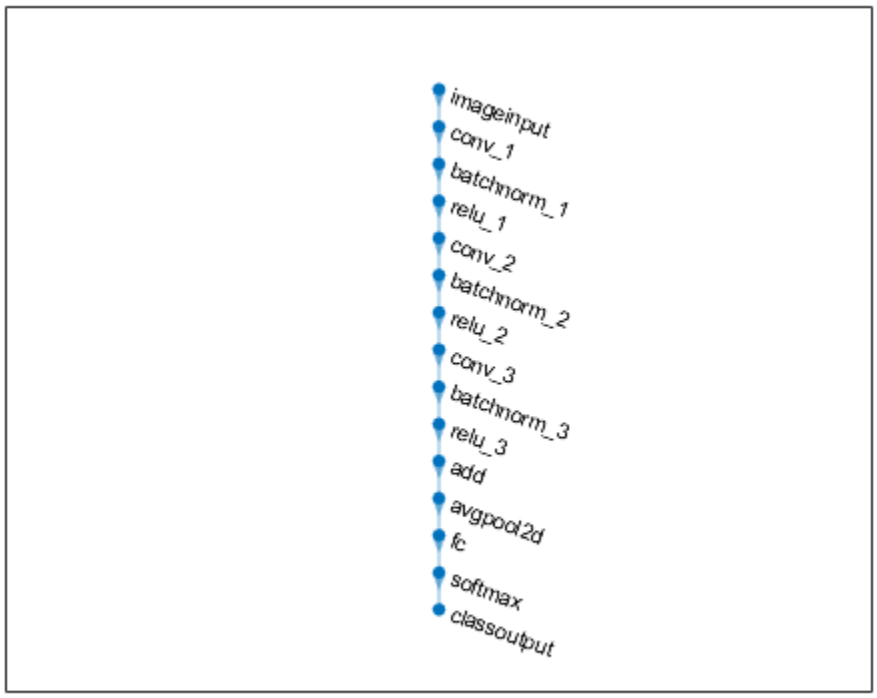
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(5,16,'Padding','same')
    batchNormalizationLayer
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
```

```
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

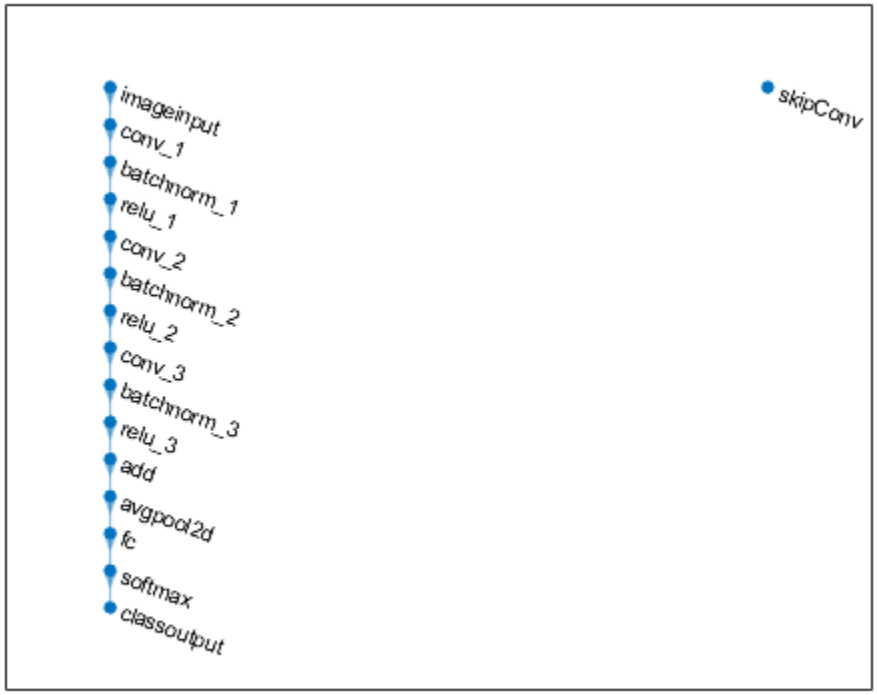
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



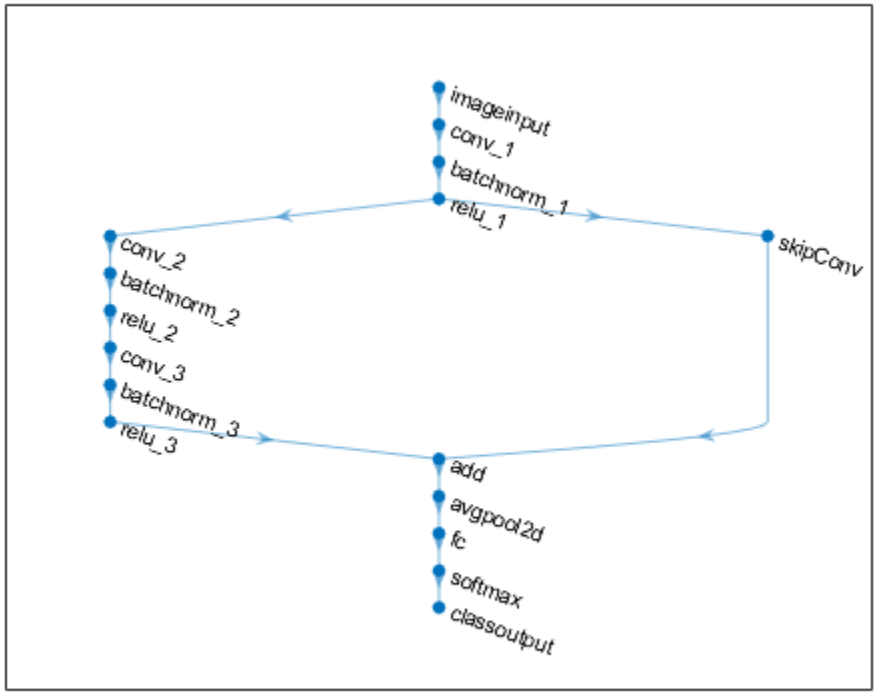
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 1-by-1 convolutional layer. To check that the layer is in the graph, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` layer to the `'add'` layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named `'in1'` and `'in2'`. The third ReLU layer is already connected to the `'in1'` input. Connect the `'relu_1'` layer to the `'skipConv'` layer and the `'skipConv'` layer to the `'in2'` input of the `'add'` layer. The addition layer now sums the outputs of the third ReLU layer and the `'skipConv'` layer. To check that the layers are connected correctly, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

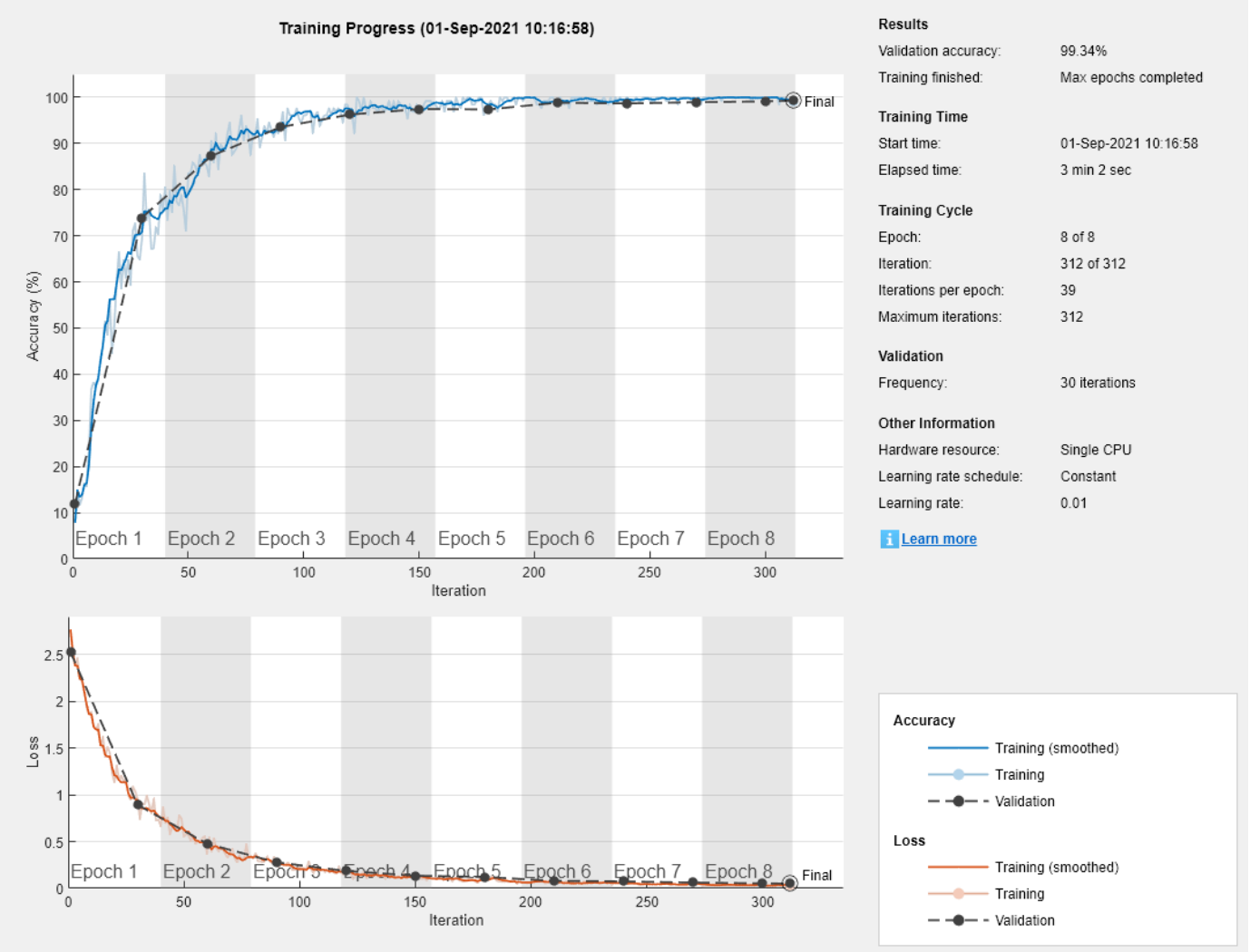


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



Display the properties of the trained network. The network is a `DAGNetwork` object.

```
net
```

```
net =
  DAGNetwork with properties:

         Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9934
```

Input Arguments

`lgraph` — Layer graph

`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`s` — Connection source

character vector | string scalar

Connection source, specified as a character vector or a string scalar.

- If the source layer has a single output, then `s` is the name of the layer.
- If the source layer has multiple outputs, then `s` is the layer name followed by the character `/` and the name of the layer output: `'layerName/outputName'`.

Example: `'conv1'`

Example: `'mpool/indices'`

`d` — Connection destination

character vector | string scalar

Connection destination, specified as a character vector or a string scalar.

- If the destination layer has a single input, then `d` is the name of the layer.
- If the destination layer has multiple inputs, then `d` is the layer name followed by the character `/` and the name of the layer input: `'layerName/inputName'`.

Example: `'fc'`

Example: `'addlayer1/in2'`

Output Arguments

`newlgraph` — Output layer graph

`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `addLayers` | `removeLayers` | `replaceLayer` | `disconnectLayers` | `plot` | `assembleNetwork`

Topics

“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b

convolution1dLayer

1-D convolutional layer

Description

A 1-D convolutional layer applies sliding convolutional filters to 1-D input. The layer convolves the input by moving the filters along the input and computing the dot product of the weights and the input, then adding a bias term.

The dimension that the layer convolves over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer convolves over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer convolves over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer convolves over the spatial dimension.

Creation

Syntax

```
layer = convolution1dLayer(filterSize,numFilters)
layer = convolution1dLayer(filterSize,numFilters,Name=Value)
```

Description

`layer = convolution1dLayer(filterSize,numFilters)` creates a 1-D convolutional layer and sets the `FilterSize` and `NumFilters` properties.

`layer = convolution1dLayer(filterSize,numFilters,Name=Value)` also sets the optional `Stride`, `DilationFactor`, `NumChannels`, “Parameters and Initialization” on page 1-312, “Learning Rate and Regularization” on page 1-313, and `Name` properties using one or more name-value arguments. To specify input padding, use the `Padding` name-value argument. For example, `convolution1dLayer(11,96,Padding=1)` creates a 1-D convolutional layer with 96 filters of size 11, and specifies padding of size 1 on the left and right of the layer input.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `convolution1dLayer(11,96,Padding=1)` creates a 1-D convolutional layer with 96 filters of size 11, and specifies padding of size 1 on the left and right of the layer input.

`Padding` — Padding to apply to input

`[0 0]` (default) | `"same"` | `"causal"` | nonnegative integer | vector of nonnegative integers

Padding to apply to the input, specified as one of the following:

- `"same"` — Apply padding such that the output size is `ceil(inputSize/stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same size as the input.
- `"causal"` — Apply left padding to the input, equal to `(FilterSize - 1) .* DilationFactor`. When `Stride` is `1`, the output is the same size as the input.
- Nonnegative integer `sz` — Add padding of size `sz` to both ends of the input.
- Vector `[l r]` of nonnegative integers — Add padding of size `l` to the left and `r` to the right of the input.

Example: `Padding=[2 1]` adds padding of size 2 to the left and size 1 to the right of the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Properties

Convolution

`FilterSize` — Width of filters

positive integer

This property is read-only.

Width of the filters, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumFilters` — Number of filters

positive integer

This property is read-only.

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Stride` — Step size for traversing input

`1` (default) | positive integer

Step size for traversing the input, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DilationFactor` — Factor for dilated convolution

`1` (default) | positive integer

Factor for dilated convolution (also known as atrous convolution), specified as a positive integer.

Use dilated convolutions to increase the receptive field (the area of the input that 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 `(FilterSize − 1) .* DilationFactor + 1`. For example, a 1-by-3 filter with a dilation factor of 2 is equivalent to a 1-by-5 filter with zeros between the elements.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PaddingSize` — Size of padding
`[0 0]` (default) | vector of two nonnegative integers

Size of padding to apply to each side of the input, specified as a vector `[l r]` of two nonnegative integers, where `l` is the padding applied to the left and `r` is the padding applied to the right.

When you create a layer, use the `Padding` name-value argument to specify the padding size.

Data Types: `double`

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'` | `'causal'`

This property is read-only.

Method to determine padding size, specified as one of the following:

- `'manual'` – Pad using the integer or vector specified by `Padding`.
- `'same'` – Apply padding such that the output size is `ceil(inputSize/Stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same as the input.
- `'causal'` – Apply causal padding. Pad the left of the input with padding size `(FilterSize - 1) .* DilationFactor`.

To specify the layer padding, use the `Padding` name-value argument.

Data Types: `char`

`PaddingValue` — Value to pad data
`0` (default) | scalar | `'symmetric-include-edge'` | `'symmetric-exclude-edge'` | `'replicate'`

This property is read-only.

Value to pad data, specified as one of the following:

`PaddingValue`	Description	Example
Scalar	Pad with the specified scalar value.	[3 1 4] → [0 0 3 1 4 0 0]
`'symmetric-include-edge'`	Pad using mirrored values of the input, including the edge values.	[3 1 3] → [1 3 3 1 4 4 1]
`'symmetric-exclude-edge'`	Pad using mirrored values of the input, excluding the edge values.	[3 1 4] → [4 1 3 1 4 1 3]
`'replicate'`	Pad using repeated border elements of the input.	[3 1 3] → [3 3 3 1 4 4 4]

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`NumChannels` — Number of channels for each filter
`'auto'` (default) | positive integer

This property is read-only.

Number of channels for each filter, specified as `'auto'` or a positive integer.

If `NumChannels` is `'auto'`, then the software automatically determines the number of channels at training time.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as the Xavier initializer). The Glorot initializer independently samples from a uniform distribution with a mean of zero and a variance of `2/(numIn + numOut)`, where `numIn = FilterSize*NumChannels` and `numOut = FilterSize*NumFilters`.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize*NumChannels`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights

`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize`-by-`NumChannels`-by-`NumFilters` array.

Data Types: `single` | `double`

`Bias` — Layer biases

`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-`NumFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 1-D Convolutional Layer

Create a 1-D convolutional layer with 96 filters of width of 11.

```
layer = convolution1dLayer(11,96)
```

```
layer =
  Convolution1DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: 11
       NumChannels: 'auto'
        NumFilters: 96
            Stride: 1
    DilationFactor: 1
       PaddingMode: 'manual'
       PaddingSize: [0 0]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Include a 1-D convolutional layer in a `Layer` array.

```
layers = [
    sequenceInputLayer(12)
    convolution1dLayer(11,96)
    reluLayer
    globalMaxPooling1dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:
```

```
1   ''   Sequence Input           Sequence input with 12 dimensions
2   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
3   ''   ReLU                     ReLU
4   ''   1-D Global Max Pooling   1-D global max pooling
5   ''   Fully Connected          10 fully connected layer
6   ''   Softmax                  softmax
7   ''   Classification Output    crossentropyex
```

Algorithms

1-D Convolutional Layer

A 1-D convolutional layer applies sliding convolutional filters to 1-D input. The layer convolves the input by moving the filters along the input and computing the dot product of the weights and the input, then adding a bias term.

The dimension that the layer convolves over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer convolves over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer convolves over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer convolves over the spatial dimension.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `Convolution1DLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	Output Format
`"SCB"` (spatial, channel, batch)	`"SCB"` (spatial, channel, batch)
`"CBT"` (channel, batch, time)	`"CBT"` (channel, batch, time)
`"SCBT"` (spatial, channel, batch, time)	`"SCBT"` (spatial, channel, batch, time)

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the*

2015 IEEE International Conference on Computer Vision, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Topics

“Sequence Classification Using 1-D Convolutions”
“Sequence-to-Sequence Classification Using 1-D Convolutions”
“Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2021b

convolution2dLayer

2-D convolutional layer

Description

A 2-D convolutional layer applies sliding convolutional filters to 2-D 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.

Creation

Syntax

```
layer = convolution2dLayer(filterSize,numFilters)
layer = convolution2dLayer(filterSize,numFilters,Name,Value)
```

Description

`layer = convolution2dLayer(filterSize,numFilters)` creates a 2-D convolutional layer and sets the `FilterSize` and `NumFilters` properties.

`layer = convolution2dLayer(filterSize,numFilters,Name,Value)` sets the optional `Stride`, `DilationFactor`, `NumChannels`, “Parameters and Initialization” on page 1-322, “Learning Rate and Regularization” on page 1-323, and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `convolution2dLayer(11,96,'Stride',4,'Padding',1)` creates a 2-D convolutional layer with 96 filters of size `[11 11]`, a stride of `[4 4]`, and padding of size 1 along all edges of the layer input. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input or to set the `Stride`, `DilationFactor`, `NumChannels`, “Parameters and Initialization” on page 1-322, “Learning Rate and Regularization” on page 1-323, and `Name` properties. Enclose names in single quotes.

Example: `convolution2dLayer(3,16,'Padding','same')` creates a 2-D convolutional layer with 16 filters of size `[3 3]` and `'same'` padding. At training time, the software calculates and sets the size of the padding so that the layer output has the same size as the input.

`Padding` — Input edge padding
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.
- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Vector `[a b]` of nonnegative integers — Add padding of size `a` to the top and bottom of the input and padding of size `b` to the left and right.
- Vector `[t b l r]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, and `r` to the right of the input.

Example: `'Padding',1` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Convolution

`FilterSize` — Height and width of filters

vector of two positive integers

Height and width of the filters, specified as a vector `[h w]` of two positive integers, where `h` is the height and `w` is the width. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify `FilterSize` as a scalar to use the same value for the height and width.

Example: `[5 5]` specifies filters with a height of 5 and a width of 5.

`NumFilters` — Number of filters

positive integer

This property is read-only.

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Stride` — Step size for traversing input

`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector `[a b]` of two positive integers, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both step sizes.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`DilationFactor` — Factor for dilated convolution
`[1 1]` (default) | vector of two positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector `[h w]` of two positive integers, where `h` is the vertical dilation and `w` is the horizontal dilation. When creating the layer, you can specify `DilationFactor` as a scalar to use the same value for both horizontal and vertical dilations.

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.

Example: `[2 3]`

`PaddingSize` — Size of padding
`[0 0 0 0]` (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector `[t b l r]` of four nonnegative integers, where `t` is the padding applied to the top, `b` is the padding applied to the bottom, `l` is the padding applied to the left, and `r` is the padding applied to the right.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 1 2 2]` adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the `'Padding'` value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

`Padding` — Size of padding
`[0 0]` (default) | vector of two nonnegative integers

Note `Padding` property will be removed in a future release. Use `PaddingSize` instead. When creating a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector `[a b]` of two nonnegative integers, where `a` is the padding applied to the top and bottom of the input data and `b` is the padding applied to the left and right.

Example: `[1 1]` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

`PaddingValue` — Value to pad data

0 (default) | scalar | `'symmetric-include-edge'` | `'symmetric-exclude-edge'` | `'replicate'`

Value to pad data, specified as one of the following:

`PaddingValue`	Description	Example
Scalar	Pad with the specified scalar value.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&3&1&4&0&0\\0&0&1&5&9&0&0\\0&0&2&6&5&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}$
`'symmetric-include-edge'`	Pad using mirrored values of the input, including the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&1&1&5&9&9&5\\1&3&3&1&4&4&1\\1&3&3&1&4&4&1\\5&1&1&5&9&9&5\\6&2&2&6&5&5&6\\6&2&2&6&5&5&6\\5&1&1&5&9&9&5\end{bmatrix}$
`'symmetric-exclude-edge'`	Pad using mirrored values of the input, excluding the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\\9&5&1&5&9&5&1\\5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\end{bmatrix}$
`'replicate'`	Pad using repeated border elements of the input	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\1&1&1&5&9&9&9\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\end{bmatrix}$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`NumChannels` — Number of channels for each filter
`'auto'` (default) | positive integer

Number of channels for each filter, specified as `'auto'` or a positive integer.

This parameter is always equal to the number of channels of the input to the convolutional layer. For example, if the input is a color image, then the number of channels for the input is 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for the current layer is 16.

If `NumChannels` is `'auto'`, then the software determines the number of channels at training time.

Example: 256

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannels` and `numOut = FilterSize(1)*FilterSize(2)*NumFilters`.
- `'he'` – Initialize the weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannels`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights

`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`NumChannels`-by-`NumFilters` array.

Data Types: `single` | `double`

`Bias` — Layer biases

`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-`NumFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Convolutional Layer

Create a convolutional layer with 96 filters, each with a height and width of 11. Use a stride (step size) of 4 in the horizontal and vertical directions.

```
layer = convolution2dLayer(11,96,'Stride',4)
```

```
layer =
  Convolution2DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: [11 11]
       NumChannels: 'auto'
        NumFilters: 96
            Stride: [4 4]
    DilationFactor: [1 1]
       PaddingMode: 'manual'
       PaddingSize: [0 0 0 0]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Include a convolutional layer in a `Layer` array.

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

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Specify Initial Weights and Biases in Convolutional Layer

To specify the weights and bias initializer functions, use the `WeightsInitializer` and `BiasInitializer` properties respectively. To specify the weights and biases directly, use the `Weights` and `Bias` properties respectively.

Specify Initialization Functions

Create a convolutional layer with 32 filters, each with a height and width of 5 and specify the weights initializer to be the He initializer.

```
filterSize = 5;
numFilters = 32;
layer = convolution2dLayer(filterSize,numFilters, ...
    'WeightsInitializer','he')
```

```
layer =
  Convolution2DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: [5 5]
       NumChannels: 'auto'
        NumFilters: 32
            Stride: [1 1]
    DilationFactor: [1 1]
       PaddingMode: 'manual'
       PaddingSize: [0 0 0 0]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Note that the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Functions

To specify your own initialization function for the weights and biases, set the `WeightsInitializer` and `BiasInitializer` properties to a function handle. For these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a convolutional layer with 32 filters, each with a height and width of 5 and specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```
filterSize = 5;
numFilters = 32;

layer = convolution2dLayer(filterSize,numFilters, ...
    'WeightsInitializer', @(sz) rand(sz) * 0.0001, ...
    'BiasInitializer', @(sz) rand(sz) * 0.0001)
```

```
layer =
  Convolution2DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: [5 5]
       NumChannels: 'auto'
        NumFilters: 32
            Stride: [1 1]
    DilationFactor: [1 1]
       PaddingMode: 'manual'
       PaddingSize: [0 0 0 0]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Again, the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Weights and Bias Directly

Create a fully connected layer with an output size of 10 and set the weights and bias to `W` and `b` in the MAT file `Conv2dWeights.mat` respectively.

```
filterSize = 5;
numFilters = 32;
load Conv2dWeights

layer = convolution2dLayer(filterSize,numFilters, ...
    'Weights',W, ...
    'Bias',b)
```

```
layer =
  Convolution2DLayer with properties:

             Name: ''
```

```
  Hyperparameters
       FilterSize: [5 5]
      NumChannels: 3
       NumFilters: 32
           Stride: [1 1]
   DilationFactor: [1 1]
      PaddingMode: 'manual'
      PaddingSize: [0 0 0 0]
     PaddingValue: 0

  Learnable Parameters
          Weights: [5x5x3x32 double]
             Bias: [1x1x32 double]

Show all properties
```

Here, the `Weights` and `Bias` properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

Create Convolutional Layer That Fully Covers Input

Suppose the size of the input is 28-by-28-by-1. Create a convolutional layer with 16 filters, each with a height of 6 and a width of 4. Set the horizontal and vertical stride to 4.

Make sure the convolution covers the input completely. For the convolution to fully cover the input, both the horizontal and vertical output dimensions must be integer numbers. For the horizontal output dimension to be an integer, one row of padding is required on the top and bottom of the image: (28 – 6+ 2 * 1)/4 + 1 = 7. For the vertical output dimension to be an integer, no zero padding is required: (28 – 4+ 2 * 0)/4 + 1 = 7.

Construct the convolutional layer.

```
layer = convolution2dLayer([6 4],16,'Stride',4,'Padding',[1 0])
```

```
layer =
  Convolution2DLayer with properties:

             Name: ''

  Hyperparameters
       FilterSize: [6 4]
      NumChannels: 'auto'
       NumFilters: 16
           Stride: [4 4]
   DilationFactor: [1 1]
      PaddingMode: 'manual'
      PaddingSize: [1 1 0 0]
     PaddingValue: 0

  Learnable Parameters
          Weights: []
             Bias: []
```

Show all properties

More About

Convolutional Layer

A 2-D convolutional layer applies sliding convolutional filters to 2-D input. The layer convolves the input by moving the filters along the input vertically and horizontally, computing the dot product of the weights and the input, and then adding a bias term.

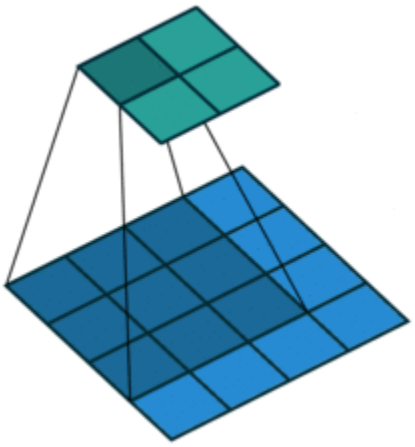
The convolutional layer consists of various components.[1]

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.

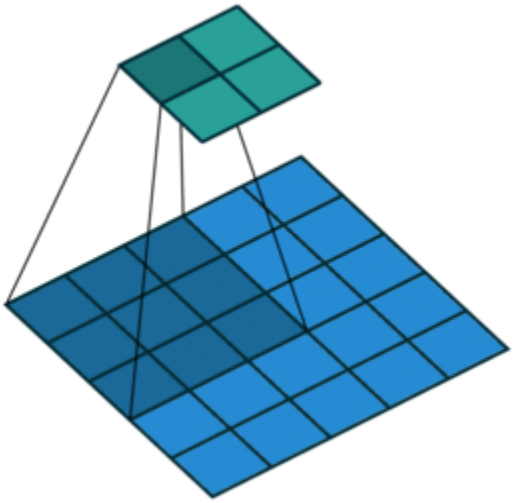
This image shows a 3-by-3 filter scanning through the input. The lower map represents the input and the upper map represents the output.



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.

1. Image credit: Convolution arithmetic (License)

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.

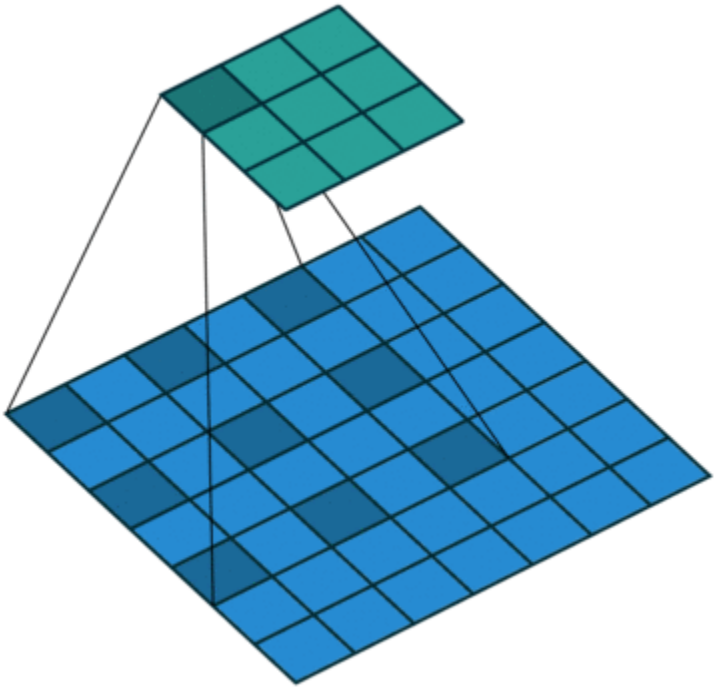
Dilated Convolutions

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

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

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

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



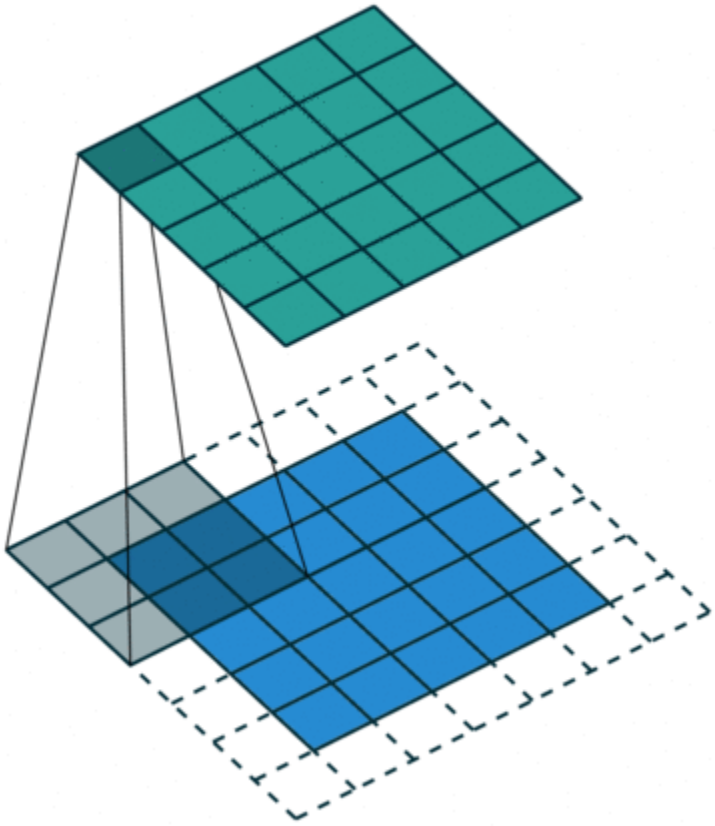
Feature Maps

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

Padding

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

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



Output Size

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

Number of Neurons

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

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

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

Learnable 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”.

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.

Compatibility Considerations

Default weights initialization is Glorot

Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'WeightsInitializer'` option of the layer to `'narrow-normal'`.

References

[1] LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Handwritten Digit Recognition with a Back-Propagation Network." In *Advances in Neural Information Processing Systems* 2 (D. Touretzky, ed.). San Francisco: Morgan Kaufmann, 1990.

[2] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. ''Gradient-Based Learning Applied to Document Recognition.'' *Proceedings of the IEEE*. Vol. 86, Number 11, 1998, pp. 2278–2324.

[3] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press, 2012.

[4] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[5] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.

See Also

`trainNetwork` | `reluLayer` | `batchNormalizationLayer` | `maxPooling2dLayer` | `fullyConnectedLayer` | `groupedConvolution2dLayer` | **Deep Network Designer**

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2016a

convolution3dLayer

3-D convolutional layer

Description

A 3-D convolutional layer applies sliding cuboidal convolution filters to 3-D input. The layer convolves the input by moving the filters along the input vertically, horizontally, and along the depth, computing the dot product of the weights and the input, and then adding a bias term.

Creation

Syntax

```
layer = convolution3dLayer(filterSize,numFilters)
layer = convolution3dLayer(filterSize,numFilters,Name,Value)
```

Description

`layer = convolution3dLayer(filterSize,numFilters)` creates a 3-D convolutional layer and sets the `FilterSize` and `NumFilters` properties.

`layer = convolution3dLayer(filterSize,numFilters,Name,Value)` sets the optional `Stride`, `DilationFactor`, `NumChannels`, “Parameters and Initialization” on page 1-339, “Learning Rate and Regularization” on page 1-340, and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `convolution3dLayer(11,96,'Stride',4,'Padding',1)` creates a 3-D convolutional layer with 96 filters of size `[11 11 11]`, a stride of `[4 4 4]`, and padding of size 1 along all edges of the layer input. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input or to set the `Stride`, `DilationFactor`, `NumChannels`, “Parameters and Initialization” on page 1-339, “Learning Rate and Regularization” on page 1-340, and `Name` properties. Enclose names in single quotes.

Example: `convolution3dLayer(3,16,'Padding','same')` creates a 3-D convolutional layer with 16 filters of size `[3 3 3]` and `'same'` padding. At training time, the software calculates and sets the size of the padding so that the layer output has the same size as the input.

`Padding` — Input edge padding
`0` (default) | array of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.
- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Three-element vector `[a b c]` of nonnegative integers — Add padding of size `a` to the top and bottom, padding of size `b` to the left and right, and padding of size `c` to the front and back of the input.
- 2-by-3 matrix `[t l f;b r k]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, `r` to the right, `f` to the front, and `k` to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: `'Padding',1` adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Convolution

`FilterSize` — Height, width, and depth of filters

vector of three positive integers

Height, width, and depth of the filters, specified as a vector `[h w d]` of three positive integers, where `h` is the height, `w` is the width, and `d` is the depth. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify `FilterSize` as a scalar to use the same value for the height, width, and depth.

Example: `[5 5 5]` specifies filters with a height, width, and depth of 5.

`NumFilters` — Number of filters

positive integer

This property is read-only.

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Stride` — Step size for traversing input

`[1 1 1]` (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector `[a b c]` of three positive integers, where `a` is the vertical step size, `b` is the horizontal step size, and `c` is the step size along the depth. When creating the layer, you can specify `Stride` as a scalar to use the same value for step sizes in all three directions.

Example: `[2 3 1]` specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

`DilationFactor` — Factor for dilated convolution
`[1 1 1]` (default) | vector of three positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector `[h w d]` of three positive integers, where `h` is the vertical dilation, `w` is the horizontal dilation, and `d` is the dilation along the depth. When creating the layer, you can specify `DilationFactor` as a scalar to use the same value for dilation in all three directions.

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-by-3 filter with the dilation factor `[2 2 2]` is equivalent to a 5-by-5-by-5 filter with zeros between the elements.

Example: `[2 3 1]` dilates the filter vertically by a factor of 2, horizontally by a factor of 3, and along the depth by a factor of 1.

`PaddingSize` — Size of padding
`[0 0 0;0 0 0]` (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix `[t l f;b r k]` of nonnegative integers, where `t` and `b` are the padding applied to the top and bottom in the vertical direction, `l` and `r` are the padding applied to the left and right in the horizontal direction, and `f` and `k` are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 2 4;1 2 4]` adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the 'Padding' value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is

`ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

`PaddingValue` — Value to pad data

0 (default) | scalar | `'symmetric-include-edge'` | `'symmetric-exclude-edge'` | `'replicate'`

Value to pad data, specified as one of the following:

`PaddingValue`	Description	Example
Scalar	Pad with the specified scalar value.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&3&1&4&0&0\\0&0&1&5&9&0&0\\0&0&2&6&5&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}$
`'symmetric-include-edge'`	Pad using mirrored values of the input, including the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&1&1&5&9&9&5\\1&3&3&1&4&4&1\\1&3&3&1&4&4&1\\5&1&1&5&9&9&5\\6&2&2&6&5&5&6\\6&2&2&6&5&5&6\\5&1&1&5&9&9&5\end{bmatrix}$
`'symmetric-exclude-edge'`	Pad using mirrored values of the input, excluding the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\\9&5&1&5&9&5&1\\5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\end{bmatrix}$
`'replicate'`	Pad using repeated border elements of the input	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\1&1&1&5&9&9&9\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\end{bmatrix}$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`NumChannels` — Number of channels for each filter

`'auto'` (default) | positive integer

Number of channels for each filter, specified as `'auto'` or a positive integer.

This parameter is always equal to the number of channels of the input to the convolutional layer. For example, if the input is a color image, then the number of channels for the input is 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for the current layer is 16.

If `NumChannels` is `'auto'`, then the software determines the number of channels at training time.

Example: 256

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights

`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumChannels` and `numOut = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumFilters`.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumChannels`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias

`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`NumChannels`-by-`NumFilters` array.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-1-by-`NumFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 3-D Convolution Layer

Create a 3-D convolution layer with 16 filters, each with a height, width, and depth of 5. Use a stride (step size) of 4 in all three directions.

```
layer = convolution3dLayer(5,16,'Stride',4)
```

```
layer =
  Convolution3DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: [5 5 5]
       NumChannels: 'auto'
        NumFilters: 16
            Stride: [4 4 4]
    DilationFactor: [1 1 1]
       PaddingMode: 'manual'
       PaddingSize: [2x3 double]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Include a 3-D convolution layer in a `Layer` array.

```
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,16,'Stride',4)
    reluLayer
    maxPooling3dLayer(2,'Stride',4)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input         28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             16 5x5x5 convolutions with stride [4  4  4] and padding [0
     3   ''   ReLU                    ReLU
     4   ''   3-D Max Pooling         2x2x2 max pooling with stride [4  4  4] and padding [0  0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Specify Initial Weights and Biases in 3-D Convolutional Layer

To specify the weights and bias initializer functions, use the `WeightsInitializer` and `BiasInitializer` properties respectively. To specify the weights and biases directly, use the `Weights` and `Bias` properties respectively.

Specify Initialization Functions

Create a 3-D convolutional layer with 32 filters, each with a height, width, and depth of 5. Specify the weights initializer to be the He initializer.

```
filterSize = 5;
numFilters = 32;
layer = convolution3dLayer(filterSize,numFilters, ...
    'WeightsInitializer','he')
```

```
layer =
  Convolution3DLayer with properties:

             Name: ''

   Hyperparameters
        FilterSize: [5 5 5]
       NumChannels: 'auto'
        NumFilters: 32
            Stride: [1 1 1]
    DilationFactor: [1 1 1]
       PaddingMode: 'manual'
       PaddingSize: [2x3 double]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

Note that the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Functions

To specify your own initialization function for the weights and biases, set the `WeightsInitializer` and `BiasInitializer` properties to a function handle. For these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a convolutional layer with 32 filters, each with a height, width, and depth of 5. Specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```
filterSize = 5;
numFilters = 32;

layer = convolution3dLayer(filterSize,numFilters, ...
    'WeightsInitializer', @(sz) rand(sz) * 0.0001, ...
    'BiasInitializer', @(sz) rand(sz) * 0.0001)
```

```
layer =
  Convolution3DLayer with properties:

             Name: ''

   Hyperparameters
       FilterSize: [5 5 5]
      NumChannels: 'auto'
       NumFilters: 32
           Stride: [1 1 1]
    DilationFactor: [1 1 1]
      PaddingMode: 'manual'
      PaddingSize: [2x3 double]
     PaddingValue: 0

   Learnable Parameters
          Weights: []
             Bias: []

  Show all properties
```

Again, the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Weights and Bias Directly

Create a 3-D convolutional layer compatible with color images. Set the weights and bias to `W` and `b` in the MAT file `Conv3dWeights.mat` respectively.

```
filterSize = 5;
numFilters = 32;
load Conv3dWeights

layer = convolution3dLayer(filterSize,numFilters, ...
    'Weights',W, ...
    'Bias',b)
```

```
layer =
  Convolution3DLayer with properties:
```

```
            Name: ''

   Hyperparameters
        FilterSize: [5 5 5]
       NumChannels: 3
        NumFilters: 32
            Stride: [1 1 1]
    DilationFactor: [1 1 1]
       PaddingMode: 'manual'
       PaddingSize: [2x3 double]
      PaddingValue: 0

   Learnable Parameters
           Weights: [5-D double]
              Bias: [1x1x1x32 double]

  Show all properties
```

Here, the `Weights` and `Bias` properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

Create Convolutional Layer That Fully Covers 3-D Input

Suppose the size of the input is 28-by-28-by-28-by-1. Create a 3-D convolutional layer with 16 filters, each with a height of 6, a width of 4, and a depth of 5. Set the stride in all dimensions to 4.

Make sure the convolution covers the input completely. For the convolution to fully cover the input, the output dimensions must be integer numbers. When there is no dilation, the *i*-th output dimension is calculated as (imageSize(*i*) - filterSize(*i*) + padding(*i*)) / stride(*i*) + 1.

- For the horizontal output dimension to be an integer, two rows of padding are required: (28 – 6 + 2)/4 + 1 = 7. Distribute the padding symmetrically by adding one row of padding at the top and bottom of the image.
- For the vertical output dimension to be an integer, no padding is required: (28 – 4+ 0)/4 + 1 = 7.
- For the depth output dimension to be an integer, one plane of padding is required: (28 – 5 + 1)/4 + 1 = 7. You must distribute the padding asymmetrically across the front and back of the image. This example adds one plane of padding to the back of the image.

Construct the convolutional layer. Specify `'Padding'` as a 2-by-3 matrix. The first row specifies prepadding and the second row specifies postpadding in the three dimensions.

```
layer = convolution3dLayer([6 4 5],16,'Stride',4,'Padding',[1 0 0;1 0 1])
```

```
layer =
  Convolution3DLayer with properties:

            Name: ''

   Hyperparameters
        FilterSize: [6 4 5]
       NumChannels: 'auto'
        NumFilters: 16
```

```
          Stride: [4 4 4]
    DilationFactor: [1 1 1]
       PaddingMode: 'manual'
       PaddingSize: [2x3 double]
      PaddingValue: 0

   Learnable Parameters
          Weights: []
             Bias: []

  Show all properties
```

More About

3-D Convolutional Layer

A convolutional layer applies sliding convolutional filters to the input. A 3-D convolutional layer extends the functionality of a 2-D convolutional layer to a third dimension, depth. The layer convolves the input by moving the filters along the input vertically, horizontally, and along the depth, computing the dot product of the weights and the input, and then adding a bias term. To learn more, see the definition of convolutional layer on page 1-329 on the `convolution2dLayer` reference page.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`convolution2dLayer` | `globalAveragePooling3dLayer` | `maxPooling3dLayer` | `image3dInputLayer`

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2019a

crop2dLayer

2-D crop layer

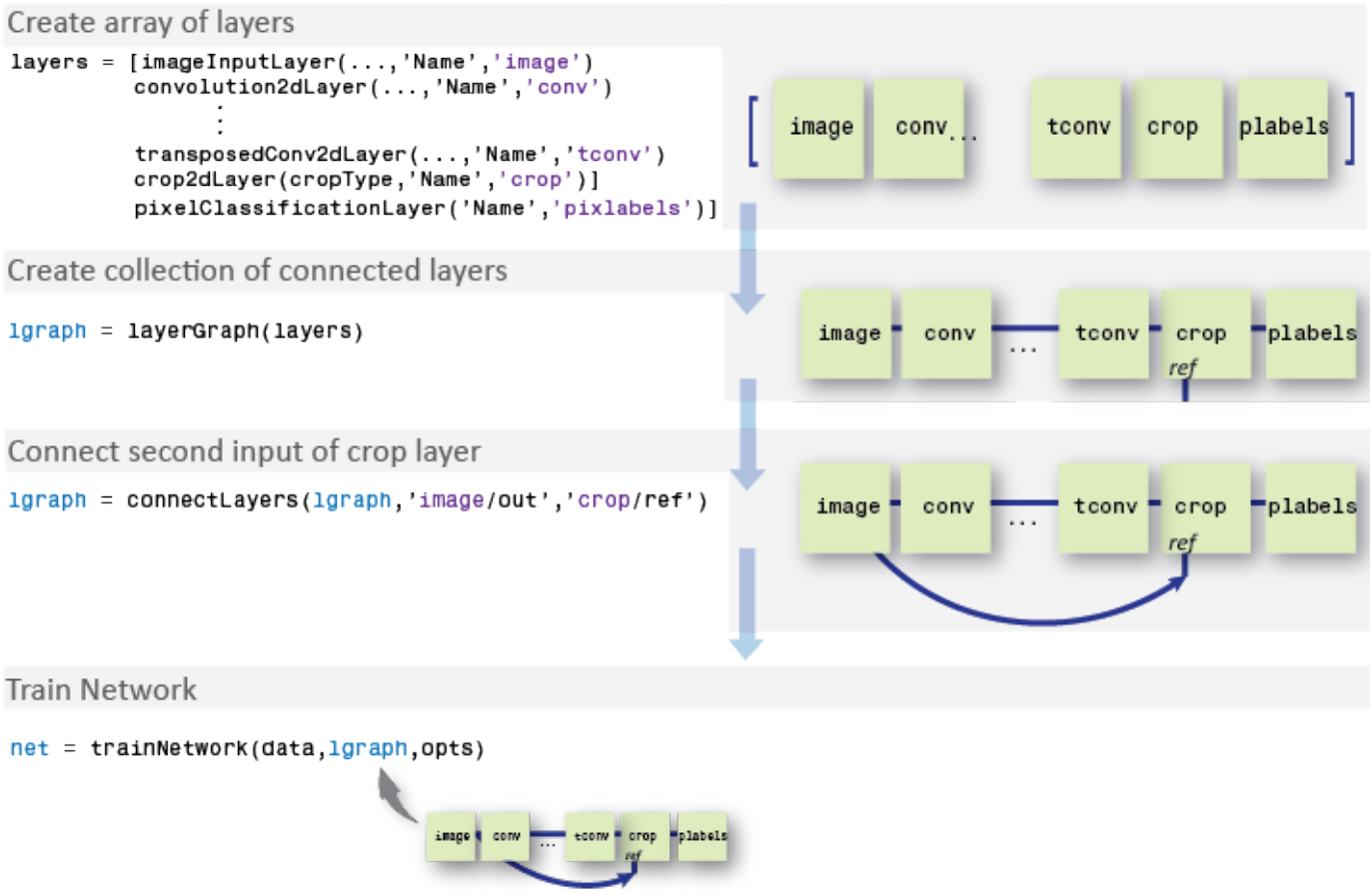
Description

A 2-D crop layer applies 2-D cropping to the input.

There are two inputs to this layer:

- `'in'` — The feature map that will be cropped
- `'ref'` — A reference layer used to determine the size, [*height width*], of the cropped output

Once you create this layer, you can add it to a `layerGraph` to make serial connections between layers. To connect the crop layer to other layers, call `connectLayers` and specify the input names. The `connectLayers` function returns a connected `LayerGraph` object ready to train a network.



Creation

Syntax

```
layer = crop2dLayer(Mode)
layer = crop2dLayer(Location)
layer = crop2dLayer( ___ ,'Name',Name)
```

Description

`layer = crop2dLayer(Mode)` returns a layer that crops an input feature map, and sets the `Mode` property.

`layer = crop2dLayer(Location)` returns a layer that crops an input feature map using a rectangular window, and sets the `Location` property that indicates the position of the window.

`layer = crop2dLayer( ___ ,'Name',Name)` creates a layer for cropping and sets the optional `Name` property.

Properties

`Mode` — Cropping mode
`'centercrop'` (default) | `'custom'`

Cropping mode, specified as `'centercrop'` or `'custom'`.

Mode	Description
`'centercrop'`	The location of the cropping window is the center of the input feature map.
`'custom'`	The location of the cropping window is based on the `Location` property. This value is automatically set when the `Location` property is specified as a 2-element row vector.

Data Types: `char`

`Location` — Cropping window location
`'auto'` (default) | 2-element row vector

Cropping window location, specified as `'auto'` or a 2-element row vector.

Location	Description
2-element row vector in the format [*x y*]	The upper-left corner of the cropping window is at the location [*x y*] of the input feature map. *x* indicates the location in the horizontal direction and *y* is the vertical direction.
`'auto'`	The cropping window is located at the center of the input feature map. This value is automatically set when the `Mode` property is specified as `'centercrop'`.

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
2 (default)

Number of inputs of the layer. This layer has two inputs.

Data Types: `double`

`InputNames` — Input names
`{'in' 'ref'}` (default)

Input names of the layer. This layer has two inputs, named `'in'` and `'ref'`.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 2-D Crop Layer

Create a 2-D crop layer and connect both of the inputs using a `layerGraph` object.

Create the layers.

```
layers = [
     imageInputLayer([32 32 3],'Name','image')
     crop2dLayer('centercrop','Name','crop')]
```

```
layers =
  2x1 Layer array with layers:

     1   'image'   Image Input   32x32x3 images with 'zerocenter' normalization
     2   'crop'    Crop 2D       center crop
```

Create a `layerGraph`. The first input of `crop2dLayer` is automatically connected to the first output of the image input layer.

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

```
lgraph =
  LayerGraph with properties:

         Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [1x2 table]
     InputNames: {'image'}
    OutputNames: {1x0 cell}
```

Connect the image input layer to the `"ref"` input of the 2-D crop layer.

```
lgraph = connectLayers(lgraph,'image','crop/ref')
```

```
lgraph =
  LayerGraph with properties:

         Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [2x2 table]
     InputNames: {'image'}
    OutputNames: {1x0 cell}
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`pixelClassificationLayer` | `layerGraph` | `fcnLayers` | `segnetLayers` | `unetLayers` | `trainNetwork` | `semanticseg` | `deeplabv3plusLayers`

Topics

“Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
“Deep Learning in MATLAB”

Introduced in R2017b

crop3dLayer

3-D crop layer

Description

A 3-D crop layer crops a 3-D volume to the size of the input feature map.

Specify the number of inputs to the layer when you create it. The inputs to the layer have the names `'in'` and `'ref'`. Use the input names when connecting or disconnecting the layer by using `connectLayers` or `disconnectLayers`. All inputs to a 3-D crop layer must have the same number of dimensions.

Creation

Syntax

```
layer = crop3dLayer
layer = crop3dLayer([X Y Z])
layer = crop3dLayer( ___ ,'Name',Name)
```

Description

`layer = crop3dLayer` creates a 3-D crop layer that crops an input feature map from the center of the feature map. The size of the cropped region is equal to the size of a second reference input feature map.

`layer = crop3dLayer([X Y Z])` also sets the cropLocation property with the (X,Y,Z) coordinate of the crop window. X is the coordinate in the horizontal direction, Y is the coordinate in the vertical direction, and Z is the coordinate in the depth direction.

`layer = crop3dLayer( ___ ,'Name',Name)` also sets the `Name` property. To create a network containing a 3-D crop layer, you must specify a layer name.

Properties

Crop

`cropLocation` — Crop location
`'centercrop'` (default) | three-element numeric vector

Crop location, specified as `'centercrop'` or a three-element numeric vector representing the (x,y,z) coordinate of the crop window.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
2 (default)

Number of inputs of the layer. This layer accepts two inputs.

Data Types: `double`

`InputNames` — Input names
`{'in','ref'}` (default)

Input names of the layer, specified as `{'in','ref'}`. This layer accepts two inputs.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create and Connect 3-D Crop Layer

Create a 3-D crop layer and connect both of its inputs using a layerGraph object.

```
layers = [
    image3dInputLayer([32 32 32 3],'Name','image')
    convolution3dLayer(3,16,'Padding','same','Name','conv')
    crop3dLayer('Name','crop')
    concatenationLayer(4,2,'Name','concat')
    ]
```

```
layers =
  4x1 Layer array with layers:

     1   'image'    3-D Image Input   32x32x32x3 images with 'zerocenter' normalization
     2   'conv'     Convolution       16 3x3x3 convolutions with stride [1  1  1] and padding 'sa
```

```
     3   'crop'     Crop 3D           center crop
     4   'concat'   Concatenation     Concatenation of 2 inputs along dimension 4
```

Create a layer graph. The first input of the 3-D crop layer is automatically connected to the output of the 3-D convolutional layer.

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

Add a max pooling layer to the layer graph.

```
maxPool = maxPooling3dLayer(2,'stride',2,'Name','pool');
lgraph = addLayers(lgraph,maxPool);
lgraph = connectLayers(lgraph,'image','pool');
```

Connect the second input of the crop layer to the output of the max pooling layer.

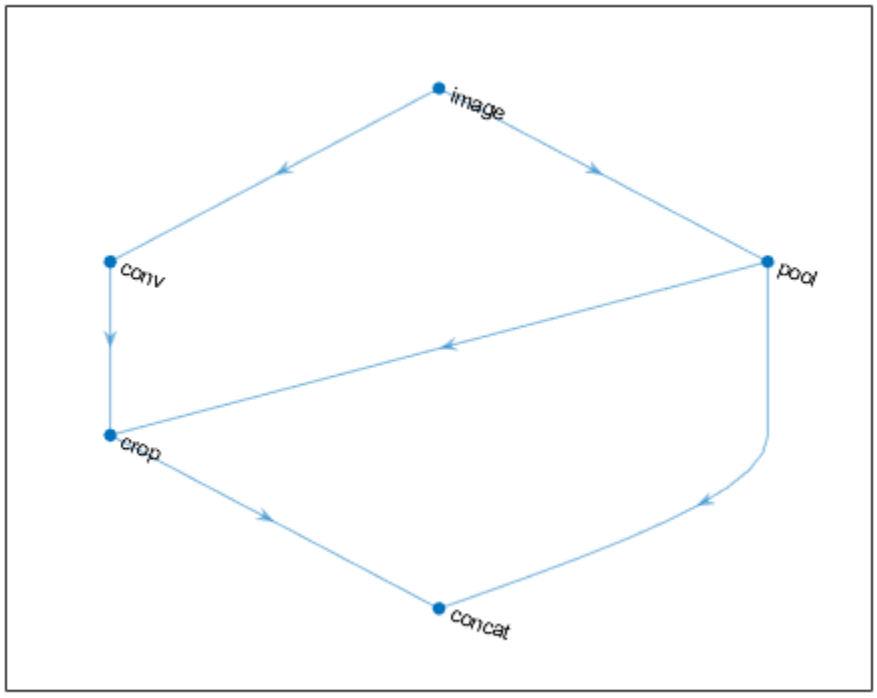
```
lgraph = connectLayers(lgraph,'pool','crop/ref');
```

Concatenate the crop layer output and the max pooling layer output.

```
lgraph = connectLayers(lgraph,'pool','concat/in2');
```

Display the layer graph.

```
plot(lgraph)
```



See Also

`trainNetwork` | `layerGraph` | `crop2dLayer`

Topics

“Deep Learning in MATLAB”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b

crossChannelNormalizationLayer

Channel-wise local response normalization layer

Description

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

Creation

Syntax

```
layer = crossChannelNormalizationLayer(windowChannelSize)
layer = crossChannelNormalizationLayer(windowChannelSize,Name,Value)
```

Description

`layer = crossChannelNormalizationLayer(windowChannelSize)` creates a channel-wise local response normalization layer and sets the `WindowChannelSize` property.

`layer = crossChannelNormalizationLayer(windowChannelSize,Name,Value)` sets the optional properties `WindowChannelSize`, `Alpha`, `Beta`, `K`, and `Name` using name-value pairs. For example, `crossChannelNormalizationLayer(5,'K',1)` creates a local response normalization layer for channel-wise normalization with a window size of 5 and *K* hyperparameter 1. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

Cross-Channel Normalization

`WindowChannelSize` — Size of the channel window
positive integer

Size of the channel window, which controls the number of channels that are used for the normalization of each element, specified as a positive integer.

If `WindowChannelSize` is even, then the window is asymmetric. The software looks at the previous `floor((w-1)/2)` channels and the following `floor(w/2)` channels. For example, if `WindowChannelSize` is 4, then the layer normalizes each element by its neighbor in the previous channel and by its neighbors in the next two channels.

Example: 5

`Alpha` — *α* hyperparameter in normalization
0.0001 (default) | numeric scalar

α hyperparameter in the normalization (the multiplier term), specified as a numeric scalar.

Example: `0.0002`

`Beta` — β hyperparameter in normalization
0.75 (default) | numeric scalar

β hyperparameter in the normalization, specified as a numeric scalar. The value of `Beta` must be greater than or equal to 0.01.

Example: `0.8`

`K` — K hyperparameter in the normalization
2 (default) | numeric scalar

K hyperparameter in the normalization, specified as a numeric scalar. The value of `K` must be greater than or equal to 10^{-5}.

Example: `2.5`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Local Response Normalization Layer

Create a local response normalization layer for channel-wise normalization, where a window of five channels normalizes each element, and the additive constant for the normalizer K is 1.

```
layer = crossChannelNormalizationLayer(5,'K',1)
```

```
layer =
  CrossChannelNormalizationLayer with properties:

                Name: ''

   Hyperparameters
    WindowChannelSize: 5
                Alpha: 1.0000e-04
                 Beta: 0.7500
                    K: 1
```

Include a local response normalization layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    crossChannelNormalizationLayer(3)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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
     3   ''   ReLU                          ReLU
     4   ''   Cross Channel Normalization   cross channel normalization with 3 channels per eleme
     5   ''   Fully Connected               10 fully connected layer
     6   ''   Softmax                       softmax
     7   ''   Classification Output         crossentropyex
```

Limitations

- This layer does not support 3-D image inputs or vector sequence inputs.

More About

Local Response Normalization

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 [1]. 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 [1]. You can obtain the equivalent formula by multiplying the `alpha` value by the `windowChannelSize`.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`averagePooling2dLayer` | `convolution2dLayer` | `maxPooling2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

crosschannelnorm

Cross channel square-normalize using local responses

Syntax

```
dlY = crosschannelnorm(dlX,windowSize)
dlY = crosschannelnorm(dlX,windowSize,'DataFormat',FMT)
dlY = crosschannelnorm( ___ ,Name,Value)
```

Description

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.

Note This function applies the cross-channel normalization operation to `dlarray` data. If you want to apply cross-channel normalization within a `layerGraph` object or `Layer` array, use the following layer:

- `crossChannelNormalizationLayer`

`dlY = crosschannelnorm(dlX,windowSize)` normalizes each element of `dlX` with respect to local values in the same position in nearby channels. The normalized elements in `dlY` are calculated from the elements in `dlX` using the following formula.

$$y = \frac{x}{\left(K + \frac{\alpha * ss}{windowSize}\right)^{\beta}}$$

where y is an element of `dlY`, x is the corresponding element of `dlX`, ss is the sum of the squares of the elements in the channel region defined by `windowSize`, and α, β, and K are hyperparameters in the normalization.

`dlY = crosschannelnorm(dlX,windowSize,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is an unformatted `dlarray`, in addition to the input arguments the previous syntax. The output `dlY` is an unformatted dlarray with the same dimension order as `dlX`.

`dlY = crosschannelnorm( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'Beta',0.8` sets the value of the β contrast constant to `0.8`.

Examples

Normalize Data Using Values of Adjacent Channels

Use `crosschannelnorm` to normalize each observation of a mini-batch using values from adjacent channels.

Create the input data as ten observations of random values with a height and width of eight and six channels.

```
height = 8;
width = 8;
channels = 6;
observations = 10;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the cross-channel normalization using a channel window size of three.

```
dlY = crosschannelnorm(dlX,3);
```

Each value in each observation of `dlX` is normalized using the element in the previous channel and the element in the next channel.

Compare Normalized and Original Data

Values at the edges of an array are normalized using contributions from fewer channels, depending on the size of the channel window.

Create the input data as an array of ones with a height and width of two and three channels.

```
height = 2;
width = 2;
channels = 3;

X = ones(height,width,channels);
dlX = dlarray(X);
```

Normalize the data using a channel-window size of `3`, an α of `1`, a β of `1`, and a K of `1e-5`. Specify a data format of `'SSC'`.

```
dlY = crosschannelnorm(dlX,3,'Alpha',1,'Beta',1,'K',1e-5,'DataFormat','SSC');
```

Compare the values in the original and the normalized data by reshaping the three-channel arrays into 2-D matrices.

```
dlX = reshape(dlX,2,6)
```

```
dlX =
  2x6 dlarray

     1     1     1     1     1     1
     1     1     1     1     1     1
```

```
dlY = reshape(dlY,2,6)
```

```
dlY =
  2x6 dlarray

    1.5000    1.5000    1.0000    1.0000    1.5000    1.5000
    1.5000    1.5000    1.0000    1.0000    1.5000    1.5000
```

For the first and last channels, the sum of squares is calculated using only two values. For the middle channel, the sum of squares contains the values of all three channels.

Use Cross-Channel Normalization in a Model Function

Typically, the cross-channel normalization operation follows a ReLU operation. For example, the GoogLeNet architecture contains convolutional operations followed by ReLU and cross-channel normalization operations.

The function `modelFunction` defined at the end of this example shows how you can use cross-channel normalization in a model. Use `modelFunction` to find the grouped convolution and ReLU activation of some input data and then normalize the result using cross-channel normalization with a window size of `5`.

Create the input data as a single observation of random values with a height and width of ten and four channels.

```
height = 10;
width = 10;
channels = 4;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Create the parameters for the grouped convolution operation. For the weights, use a filter height and width of three, two channels per group, three filters per group, and two groups. Use a value of zero for the bias.

```
filterSize = [3 3];
numChannelsPerGroup = 2;
numFiltersPerGroup = 3 ;
numGroups = 2;

params = struct;
params.conv.weights = rand(filterSize(1),filterSize(2),numChannelsPerGroup,numFiltersPerGroup,nu
params.conv.bias = 0;
```

Apply the `modelFunction` to the data `dlX`.

```
dlY = modelFunction(dlX,params);

function dlY = modelFunction(dlX,params)

dlY = dlconv(dlX,params.conv.weights,params.conv.bias);
dlY = relu(dlY);
dlY = crosschannelnorm(dlY,5);

end
```

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a `dlarray` with or without data format. When `dlX` is an unformatted `dlarray`, you must specify the data format using the `'DataFormat',FMT` name-value pair.

You can specify up to two dimensions in `dlX` as `'S'` dimensions.

Data Types: `single` | `double`

`windowSize` — Size of channel window

scalar integer

Size of the channel window, which controls the number of channels that are used for the normalization of each element, specified as a positive integer.

If `windowSize` is even, then the window is asymmetric. The software looks at the previous `floor((windowSize-1)/2)` channels and the following `floor((windowSize)/2)` channels. For example, if `windowSize` is 4, then the function normalizes each element by its neighbor in the previous channel and by its neighbors in the next two channels.

Example: 3

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Alpha',2e-4,'Beta',0.8` sets the multiplicative normalization constant to 0.0002 and the contrast constant exponent to 0.8.

`DataFormat` — Dimension order of unformatted data

character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`Alpha` — Normalization constant (α)
`1e-4` (default) | numeric scalar

Normalization constant (α) that multiplies the sum of the squared values, specified as the comma-separated pair consisting of `'Alpha'` and a numeric scalar. The default value is `1e-4`.

Example: `'Alpha',2e-4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

`Beta` — Contrast constant (β)
`0.75` (default) | numeric scalar greater than or equal to `0.01`

Contrast constant (β), specified as the comma-separated pair consisting of `'Beta'` and a numeric scalar greater than or equal to `0.01`. The default value is `0.75`.

Example: `'Beta',0.8`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

`K` — Normalization hyperparameter (K)
`2` (default) | numeric scalar greater than or equal to `1e-5`

Normalization hyperparameter (K) used to avoid singularities in the normalization, specified as the comma-separated pair consisting of `'K'` and a numeric scalar greater than or equal to `1e-5`. The default value is `2`.

Example: `'K',2.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

Output Arguments

`dlY` — Normalized data
`dlarray`

Normalized data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension labels as `dlX`. If the input data is an unformatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

More About

Cross-Channel Normalization

The `crosschannelnorm` function normalizes each activation response based on the local responses in a specified channel window. For more information, see the definition of “Local Response Normalization” on page 1-358 on the `crossChannelNormalizationLayer` reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `avgpool` | `dlconv` | `maxpool`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2020a

crossentropy

Cross-entropy loss for classification tasks

Syntax

```
loss = crossentropy(dlY,targets)
loss = crossentropy(dlY,targets,weights)
loss = crossentropy( ___ ,'DataFormat',FMT)
loss = crossentropy( ___ ,Name,Value)
```

Description

The cross-entropy operation computes the cross-entropy loss between network predictions and target values for single-label and multi-label classification tasks.

The `crossentropy` function computes the cross-entropy loss between predictions and targets represented as `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To calculate the cross-entropy loss within a `layerGraph` object or `Layer` array for use with the `trainNetwork` function, use `classificationLayer`.

`loss = crossentropy(dlY,targets)` returns the categorical cross-entropy loss between the formatted `dlarray` object `dlY` containing the predictions and the target values `targets` for single-label classification tasks. The output `loss` is an unformatted scalar `dlarray` scalar.

For unformatted input data, use the `'DataFormat'` option.

`loss = crossentropy(dlY,targets,weights)` applies weights to the calculated loss values. Use this syntax to weight the contributions of classes, observations, regions, or individual elements of the input to the calculated loss values.

`loss = crossentropy( ___ ,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlY` is not a formatted `dlarray`.

`loss = crossentropy( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'TargetCategories','independent'` computes the cross-entropy loss for a multi-label classification task.

Examples

Cross-Entropy Loss for Single-Label Classification

Create an array of prediction scores for 12 observations over 10 classes.

```
numClasses = 10;
numObservations = 12;

Y = rand(numClasses,numObservations);
dlY = dlarray(Y,'CB');
dlY = softmax(dlY);
```

View the size and format of the prediction scores.

```
size(dlY)
```

```
ans = 1×2

    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create an array of targets encoded as one-hot vectors.

```
labels = randi(numClasses,[1 numObservations]);
targets = onehotencode(labels,1,'ClassNames',1:numClasses);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the cross-entropy loss between the predictions and the targets.

```
loss = crossentropy(dlY,targets)
```

```
loss =
  1x1 dlarray

    2.3343
```

Cross-Entropy Loss for Multi-Label Classification

Create an array of prediction scores for 12 observations over 10 classes.

```
numClasses = 10;
numObservations = 12;
Y = rand(numClasses,numObservations);
dlY = dlarray(Y,'CB');
```

View the size and format of the prediction scores.

```
size(dlY)
```

```
ans = 1×2

    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create a random array of targets encoded as a numeric array of zeros and ones. Each observation can have multiple classes.

```
targets = rand(numClasses,numObservations) > 0.75;
targets = single(targets);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the cross-entropy loss between the predictions and the targets. To specify cross-entropy loss for multi-label classification, set the `'TargetCategories'` option to `'independent'`.

```
loss = crossentropy(dlY,targets,'TargetCategories','independent')
```

```
loss =
  1x1 single dlarray

    9.8853
```

Weighted Cross-Entropy Loss

Create an array of prediction scores for 12 observations over 10 classes.

```
numClasses = 10;
numObservations = 12;

Y = rand(numClasses,numObservations);
dlY = dlarray(Y,'CB');
dlY = softmax(dlY);
```

View the size and format of the prediction scores.

```
size(dlY)
```

```
ans = 1×2
```

```
    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create an array of targets encoded as one-hot vectors.

```
labels = randi(numClasses,[1 numObservations]);
targets = onehotencode(labels,1,'ClassNames',1:numClasses);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the weighted cross-entropy loss between the predictions and the targets using a vector class weights. Specify a weights format of `'UC'` (unspecified, channel) using the `'WeightsFormat'` option.

```
weights = rand(1,numClasses);
loss = crossentropy(dlY,targets,weights,'WeightsFormat','UC')
```

```
loss =
  1x1 dlarray

    1.1261
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `dlY` is a numeric array, `targets` must be a `dlarray`.

`targets` — Target classification labels

`dlarray` | numeric array

Target classification labels, specified as a formatted or unformatted `dlarray` or a numeric array.

Specify the targets as an array containing one-hot encoded labels with the same size and format as `dlY`. For example, if `dlY` is a `numObservations`-by-`numClasses` array, then `targets(n,i)` = 1 if observation `n` belongs to class `i` `targets(n,i)` = 0 otherwise.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`weights` — Weights

`dlarray` | numeric array

Weights, specified as a `dlarray` or a numeric array.

To specify class weights, specify a vector with a `'C'` (channel) dimension with size matching the `'C'` (channel) dimension of the `dlX`. Specify the `'C'` (channel) dimension of the class weights by using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

To specify observation weights, specify a vector with a `'B'` (batch) dimension with size matching the `'B'` (batch) dimension of the `dlY`. Specify the `'B'` (batch) dimension of the class weights by using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

To specify weights for each element of the input independently, specify the weights as an array of the same size as `dlY`. In this case, if `weights` is not a formatted `dlarray` object, then the function uses the same format as `dlY`. Alternatively, specify the weights format using the `'WeightsFormat'` option.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'TargetCategories','independent','DataFormat','CB'` evaluates the cross-entropy loss for multi-label classification tasks and specifies the dimension order of the input data as `'CB'`

`TargetCategories` — Type of classification task

`'exclusive'` (default) | `'independent'`

Type of classification task, specified as the comma-separated pair consisting of `'TargetCategories'` and one of the following:

- `'exclusive'` — Single-label classification. Each observation in the predictions `dlY` is exclusively assigned to one category. The function computes the loss between the target value for the single category specified by `targets` and the corresponding prediction in `dlY`, averaged over the number of observations.
- `'independent'`— Multi-label classification. Each observation in the predictions `dlY` can be assigned to one or more independent categories. The function computes the sum of the loss between each category specified by `targets` and the predictions in `dlY` for those categories, averaged over the number of observations. Cross-entropy loss for this type of classification task is also known as binary cross-entropy loss.

`Mask` — Mask indicating which elements to include for loss computation
`dlarray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the input data for loss computation when the corresponding value in the mask is 1 and 0, respectively.

The default value is a logical array of ones with the same size as `dlY`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

`Reduction` — Mode for reducing array of loss values
`"sum"` (default) | `"none"`

Mode for reducing the array of loss values, specified as one of the following:

- `"sum"` — Sum all of the elements in the array of loss values. In this case, the output `loss` is scalar.
- `"none"` — Do not reduce the array of loss values. In this case, the output `loss` is an unformatted `dlarray` object with the same size as `dlY`.

`NormalizationFactor` — Divisor for normalizing reduced loss
`"batch-size"` (default) | `"all-elements"` | `"mask-included"` | `"none"`

Divisor for normalizing the reduced loss when `Reduction` is `"sum"`, specified as one of the following:

- `"batch-size"` — Normalize the loss by dividing it by the number of observations in `dlX`.
- `"all-elements"` — Normalize the loss by dividing it by the number of elements of `dlX`.
- `"mask-included"` — Normalize the loss by dividing the loss values by the number of included elements specified by the mask for each observation independently. To use this option, you must specify a mask using the `Mask` option.
- `"none"` — Do not normalize the loss.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial

- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`WeightsFormat` — Dimension order of weights
character vector | string scalar

Dimension order of the weights, specified as a character vector or string scalar that provides a label for each dimension of the weights.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `WeightsFormat` when `weights` is a numeric vector and `dlY` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `dlY` are vectors, then default value of `WeightsFormat` is the same as the format of `dlY`.

Data Types: `char` | `string`

Output Arguments

`loss` — Cross-entropy loss
`dlarray`

Cross-entropy loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `dlY`.

The size of `loss` depends on the `'Reduction'` option.

Algorithms

Cross-Entropy Loss

For each element Y_j of the input, the `crossentropy` function computes the corresponding cross-entropy element-wise loss values using the formula

$$\text{loss}_j = T_j \ln Y_j + (1 - T_j)\ln(1 - Y_j),$$

where T_j is the corresponding target value to Y_j.

To reduce the loss values to a scalar, the function then reduces the element-wise loss using the formula

$$\text{loss} = -\frac{1}{N}\sum_j m_j w_j \text{loss}_j,$$

where N is the normalization factor, m_j is the mask value for element j, and w_j is the weight value for element j.

If you do not opt to reduce the loss, then the function applies the mask and the weights to the loss values directly:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

This table shows the loss formulations for different tasks.

Task	Description	Loss
Single-label classification	Cross-entropy loss for mutually exclusive classes. This is useful when observations must have a single label only.	$$\text{loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} T_{ni}\ln Y_{ni},$$ where N and K are the numbers of observations, and classes, respectively.
Multi-label classification	Cross-entropy loss for independent classes. This is useful when observations can have multiple labels.	$$\text{loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} (T_{ni}\log(Y_{ni}) + (1 - T_{ni})\log(1 - Y_{ni})),$$ where N and K are the numbers of observations and classes, respectively.

Task	Description	Loss
Single-label classification with weighted classes	Cross-entropy loss with class weights. This is useful for datasets with imbalanced classes.	$$\text{loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} w_i T_{ni} \ln Y_{ni},$$ where N and K are the numbers of observations and classes, respectively, and w_i denotes the weight for class i.
Sequence-to-sequence classification	Cross-entropy loss with masked time-steps. This is useful for ignoring loss values that correspond to padded data.	$$\text{loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{S} m_{nt}\sum_{i=1}^{K} T_{nti} \ln Y_{nti},$$ where N, S, and K are the numbers of observations, time steps, and classes, m_{nt} denotes the mask value for time step t of observation n.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlY`
  - `targets`
  - `weights`
  - `'Mask'`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `huber` | `mse` | `l1loss` | `l2loss`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”

“Train Network Using Model Function”
“Train Network with Multiple Outputs”
“List of Functions with dlarray Support”

Introduced in R2019b

ctc

Connectionist temporal classification (CTC) loss for unaligned sequence classification

Syntax

```
loss = ctc(dlY,targets,YMask,targetsMask)
loss = ctc(dlY,targets,YMask,targetsMask,'DataFormat',FMT)
loss = ctc( ___ ,Name,Value)
```

Description

The CTC operation computes the connectionist temporal classification (CTC) loss between unaligned sequences.

The `ctc` function computes the CTC loss between predictions and targets represented as `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

`loss = ctc(dlY,targets,YMask,targetsMask)` returns the CTC loss between the formatted `dlarray` object `dlY` containing the predictions and the target values `targets` using the prediction and target masks `YMask` and `targetsMask`, respectively.

For unformatted input data, use the `'DataFormat'` option.

`loss = ctc(dlY,targets,YMask,targetsMask,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlY` is not a formatted `dlarray`.

`loss = ctc( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'BlankIndex','last'` specifies a blank index corresponding to the last element of the vocabulary.

Examples

CTC Loss for Unaligned Sequences

Create an array of 2 target sequences of different lengths over 10 classes. The target sequences must not contain the blank index which is 1 by default.

```
numObservations = 2;
numClasses = 10;

targets = cell(numObservations,1);
targets{1} = [2 3 5 7 9 2 3 5 3 2 3];
targets{2} = [2 3 3 3 4 4 4 6 8 8 8 10 3];
```

Create random arrays of prediction sequences. The length of the prediction sequences must be greater than or equal to the length plus the number of repeated indices of the corresponding target sequence. In this case, the first sequence has length 11 with no repeated indices, the second sequence has length 13 with 6 repeated indices.

```
Y = cell(numObservations,1);

Y{1} = rand(numClasses,11);
Y{2} = rand(numClasses,13 + 6);
```

View the cell arrays of predictions and targets

```
Y
```

```
Y=2×1 cell array
    {10×11 double}
    {10×19 double}
```

```
targets
```

```
targets=2×1 cell array
    {[     2 3 5 7 9 2 3 5 3 2 3]}
    {[2 3 3 3 4 4 4 6 8 8 8 10 3]}
```

Pad the prediction and target sequences in the second dimension using the `padsequences` function and also return the corresponding mask.

```
[Y,YMask] = padsequences(Y,2);
```

Pad the targets using the `padsequences` function. The targets must be positive integers between 1 and the number of classes, and must not contain the blank index, so specify a padding value of 2.

```
[targets,targetsMask] = padsequences(targets,2,'PaddingValue',2);
```

The `ctc` function requires the targets and target mask specified as 2-D arrays, remove the singleton channel dimension using the `squeeze` function.

```
targets = squeeze(targets);
targetsMask = squeeze(targetsMask);
```

Convert the padded prediction sequences and mask to `dlarray` with format `'CTB'` (channel, time, batch). Because formatted `dlarray` objects automatically sort the dimensions, keep the dimensions of the targets and mask consistent by also converting them to a formatted `dlarray` objects with the same formats.

```
dlY = dlarray(Y,'CTB');
YMask = dlarray(YMask,'CTB');
```

Similarly, convert the padded target sequences and mask to `dlarray` with format `'TB'` (time, batch).

```
targets = dlarray(targets,'TB');
targetsMask = dlarray(targetsMask,'TB');
```

Compute the CTC loss between the predictions and the targets using the `ctc` function.

```
loss = ctc(dlY,targets,YMask,targetsMask)
```

```
loss =
  1×1 dlarray

   12.1568
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `'DataFormat'` option.

The predictions `dlY` must have a `'B'` (batch), `'C'` (channel), and `'T'` (time) dimension and can have different sequence lengths to the corresponding targets in `targets`.

If `dlY` is a numeric array, then `targets`, `YMask`, or `targetsMask` must be a `dlarray`.

`targets` — Target sequences

`dlarray` | numeric array

Target sequences, specified as a formatted or unformatted `dlarray` or a numeric array.

Specify the targets as an array with dimensions corresponding to the observations and the time steps of the target sequences. For example, specify the targets as a formatted `dlarray` object with format `'BT'` (batch, time).

The targets must have the same number of observations as the predictions. The target values corresponding to mask values equal to 1 must be positive integers between 1 and the number of channels of `dlY` and must not include the blank index.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`YMask` — Mask indicating which prediction elements to include for loss computation

`dlarray` | logical array | numeric array

Mask indicating which prediction elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the predictions for loss computation when the corresponding value in the mask is 1 and 0, respectively.

For each time-step and observation in the mask, the corresponding elements in channel dimension must be all ones or all zeros.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

`targetsMask` — Mask indicating which target elements to include for loss computation

`dlarray` | logical array | numeric array

Mask indicating which target elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the targets for loss computation when the corresponding value in the mask is 1 and 0, respectively.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'BlankIndex','last'` specifies a blank index corresponding to the last element of the vocabulary

`BlankIndex` — Index of blank character

1 (default) | positive integer | `'last'`

Index of blank character, specified as the comma-separated pair consisting of `'BlankIndex'` and one of the following:

- Positive integer – Use the element in the vocabulary with the specified index as the blank character. If `'BlankIndex'` is an integer, then it must between 1 and the number of channels of `dlY` inclusive.
- `'last'` – Use the last element of the vocabulary as the blank character.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`DataFormat` — Dimension order of unformatted data

character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

Output Arguments

`loss` — CTC loss

`dlarray`

CTC loss, returned as an unformatted `dlarray` scalar with the same underlying data type as the input `dlY`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlY`
  - `targets`
  - `YMask`
  - `targetsMask`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `crossentropy` | `mse` | `l2loss` | `l1loss`

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2021a

DAGNetwork

Directed acyclic graph (DAG) network for deep learning

Description

A DAG network is a neural network for deep learning with layers arranged as a directed acyclic graph. A DAG network can have a more complex architecture in which layers have inputs from multiple layers and outputs to multiple layers.

Creation

There are several ways to create a `DAGNetwork` object:

- Load a pretrained network such as `squeezenet`, `googlenet`, `resnet50`, `resnet101`, or `inceptionv3`. For an example, see "Load SqueezeNet Network" on page 1-1318. For more information about pretrained networks, see "Pretrained Deep Neural Networks".
- Train or fine-tune a network using `trainNetwork`. For an example, see "Train Deep Learning Network to Classify New Images".
- Import a pretrained network from TensorFlow™-Keras, TensorFlow 2, Caffe, or the ONNX (Open Neural Network Exchange) model format.
  - For a Keras model, use `importKerasNetwork`. For an example, see "Import and Plot Keras Network" on page 1-794.
  - For a TensorFlow model in the saved model format, use `importTensorFlowNetwork`. For an example, see "Import TensorFlow Network as DAGNetwork to Classify Image" on page 1-881.
  - For a Caffe model, use `importCaffeNetwork`. For an example, see "Import Caffe Network" on page 1-772.
  - For an ONNX model, use `importONNXNetwork`. For an example, see "Import ONNX Network as DAGNetwork" on page 1-848.
- Assemble a deep learning network from pretrained layers using the `assembleNetwork` function.

Note To learn about other pretrained networks, see "Pretrained Deep Neural Networks".

Properties

`Layers` — Network layers
`Layer` array

This property is read-only.

Network layers, specified as a `Layer` array.

`Connections` — Layer connections
table

This property is read-only.

Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, `Source`, specifies the source of each connection. The second column, `Destination`, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form `'layerName/IOName'`, where `'IOName'` is the name of the layer input or output.

Data Types: `table`

`InputNames` — Network input layer names
cell array of character vectors

This property is read-only.

Network input layer names, specified as a cell array of character vectors.

Data Types: `cell`

`OutputNames` — Network output layer names
cell array

Network output layer names, specified as a cell array of character vectors.

Data Types: `cell`

Object Functions

activations	Compute deep learning network layer activations
classify	Classify data using a trained deep learning neural network
predict	Predict responses using a trained deep learning neural network
plot	Plot neural network layer graph

Examples

Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

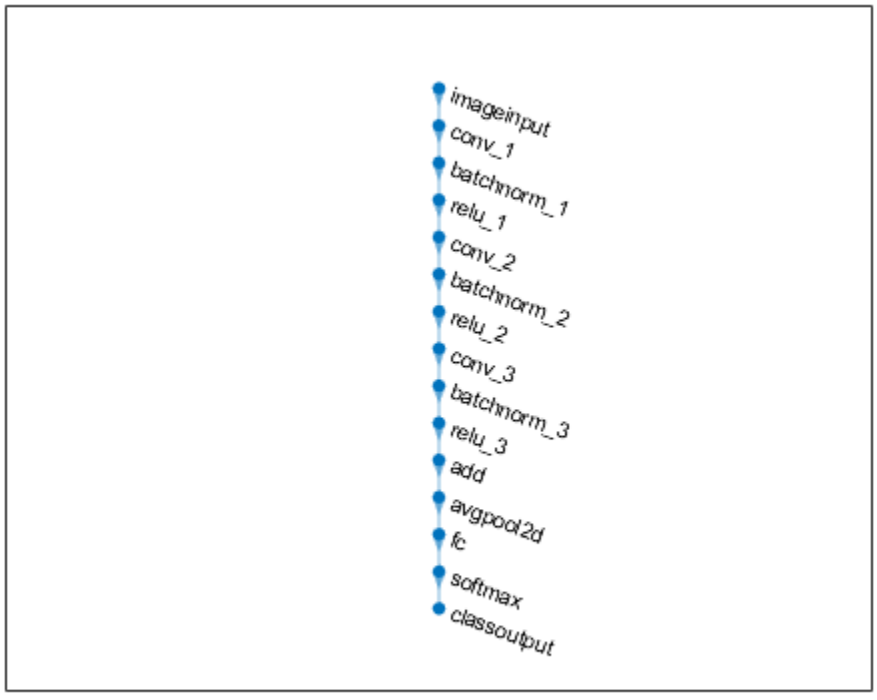
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(5,16,'Padding','same')
    batchNormalizationLayer
```

```
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

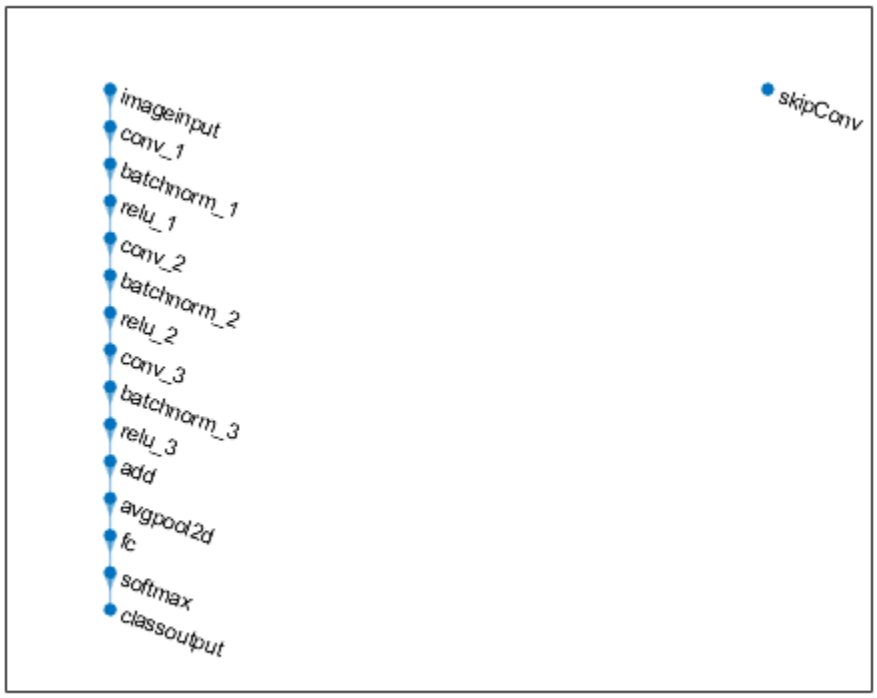
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



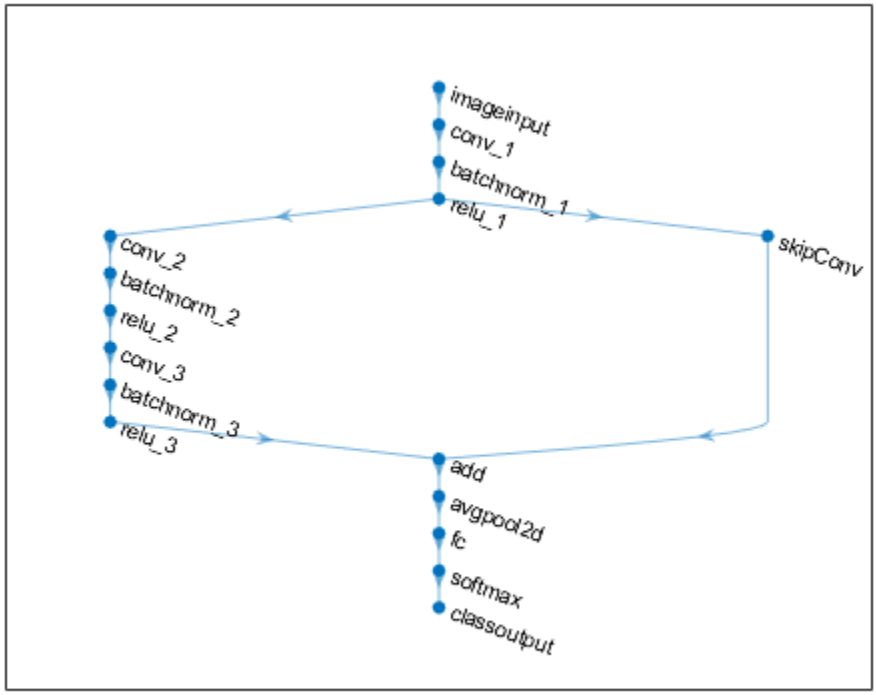
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 1-by-1 convolutional layer. To check that the layer is in the graph, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` layer to the `'add'` layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named `'in1'` and `'in2'`. The third ReLU layer is already connected to the `'in1'` input. Connect the `'relu_1'` layer to the `'skipConv'` layer and the `'skipConv'` layer to the `'in2'` input of the `'add'` layer. The addition layer now sums the outputs of the third ReLU layer and the `'skipConv'` layer. To check that the layers are connected correctly, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

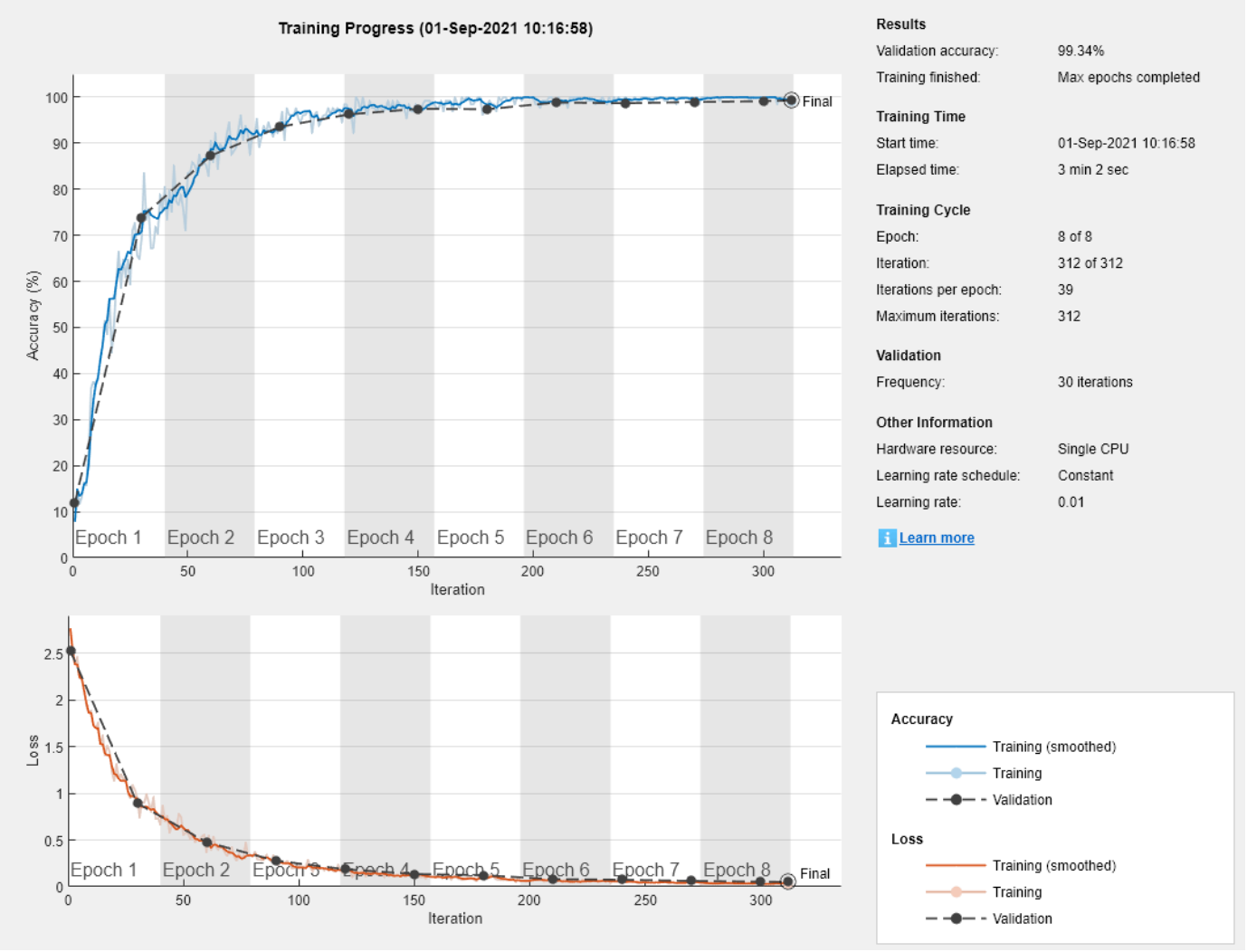


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



Display the properties of the trained network. The network is a `DAGNetwork` object.

```
net
```

```
net =
  DAGNetwork with properties:

         Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9934
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `activations`, `predict`, and `classify` object functions are supported.
- To create a `DAGNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `predict`, and `classify` methods are supported.
- To create a `DAGNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (GPU Coder).

See Also

`trainNetwork` | `trainingOptions` | `importKerasNetwork` | `layerGraph` | `classify` | `predict` | `plot` | `googlenet` | `resnet18` | `resnet50` | `resnet101` | `inceptionv3` | `inceptionresnetv2` | `squeezenet` | `SeriesNetwork` | `analyzeNetwork` | `assembleNetwork`

Topics

"Deep Learning in MATLAB"
"Classify Image Using GoogLeNet"
"Train Residual Network for Image Classification"
"Train Deep Learning Network to Classify New Images"
"Pretrained Deep Neural Networks"

Introduced in R2017b

darknet19

DarkNet-19 convolutional neural network

Syntax

```
net = darknet19
net = darknet19('Weights','imagenet')

layers = darknet19('Weights','none')
```

Description

DarkNet-19 is a convolutional neural network that is 19 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 256-by-256. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the DarkNet-19 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with DarkNet-19.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load DarkNet-19 instead of GoogLeNet.

DarkNet-19 is often used as the foundation for object detection problems and YOLO workflows [2]. For an example of how to train a you only look once (YOLO) v2 object detector, see “Object Detection Using YOLO v2 Deep Learning”. This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as DarkNet-19, DarkNet-53, MobileNet-v2, or ResNet-18 depending on application requirements.

`net = darknet19` returns a DarkNet-19 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for DarkNet-19 Network* support package. If this support package is not installed, then the function provides a download link.

`net = darknet19('Weights','imagenet')` returns a DarkNet-19 network trained on the ImageNet data set. This syntax is equivalent to `net = darknet19`.

`layers = darknet19('Weights','none')` returns the untrained DarkNet-19 network architecture. The untrained model does not require the support package.

Examples

Download DarkNet-19 Support Package

Download and install the Deep Learning Toolbox Model *for DarkNet-19 Network* support package.

Type `darknet19` at the command line.

```
darknet19
```

If the Deep Learning Toolbox Model *for DarkNet-19 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `darknet19` at the command line. If the required support package is installed, then the function returns a `SeriesNetwork` object.

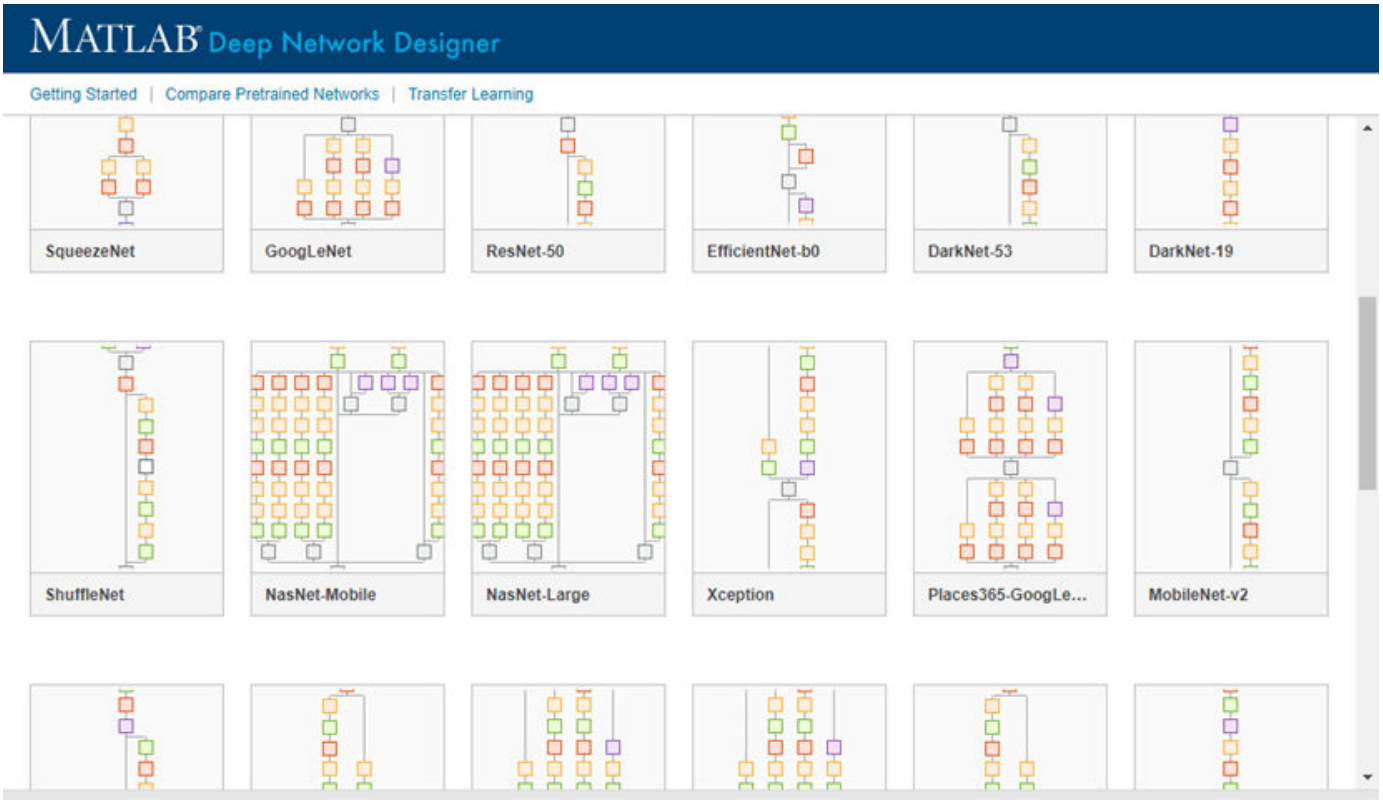
```
darknet19
```

```
ans =

  SeriesNetwork with properties:

         Layers: [64×1 nnet.cnn.layer.Layer]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(darknet19)
```

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



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

Transfer Learning with DarkNet-19

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the DarkNet-19 network instead of GoogLeNet.

```
net = darknet19
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

Output Arguments

`net` — Pretrained DarkNet-19 convolutional network

`SeriesNetwork`

Pretrained DarkNet-19 convolutional neural network, returned as a `SeriesNetwork` object.

`layers` — Untrained DarkNet-19 convolutional neural network architecture

`Layer` array

Untrained DarkNet-19 convolutional neural network architecture, returned as a `Layer` array.

References

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

[2] Redmon, Joseph. “Darknet: Open Source Neural Networks in C.” https://pjreddie.com/darknet.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = darknet19` or by passing the `darknet19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet19')`.

The syntax `darknet19('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = darknet19` or by passing the `darknet19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet19')`.

For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).

- The syntax `darknet19('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `darknet53` | `googlenet` | `trainNetwork` | `SeriesNetwork` | `layerGraph` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201` | `nasnetmobile` | `nasnetlarge`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2020a

darknet53

DarkNet-53 convolutional neural network

Syntax

```
net = darknet53
net = darknet53('Weights','imagenet')

lgraph = darknet53('Weights','none')
```

Description

DarkNet-53 is a convolutional neural network that is 53 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 256-by-256. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the DarkNet-53 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with DarkNet-53.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load DarkNet-53 instead of GoogLeNet.

DarkNet-53 is often used as the foundation for object detection problems and YOLO workflows [2]. For an example of how to train a you only look once (YOLO) v2 object detector, see “Object Detection Using YOLO v2 Deep Learning”. This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as DarkNet-19, DarkNet-53, MobileNet-v2, or ResNet-18 depending on application requirements.

`net = darknet53` returns a DarkNet-53 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for DarkNet-53 Network* support package. If this support package is not installed, then the function provides a download link.

`net = darknet53('Weights','imagenet')` returns a DarkNet-53 network trained on the ImageNet data set. This syntax is equivalent to `net = darknet53`.

`lgraph = darknet53('Weights','none')` returns the untrained DarkNet-53 network architecture. The untrained model does not require the support package.

Examples

Download DarkNet-53 Support Package

Download and install the Deep Learning Toolbox Model *for DarkNet-53 Network* support package.

Type `darknet53` at the command line.

```
darknet53
```

If the Deep Learning Toolbox Model *for DarkNet-53 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `darknet53` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

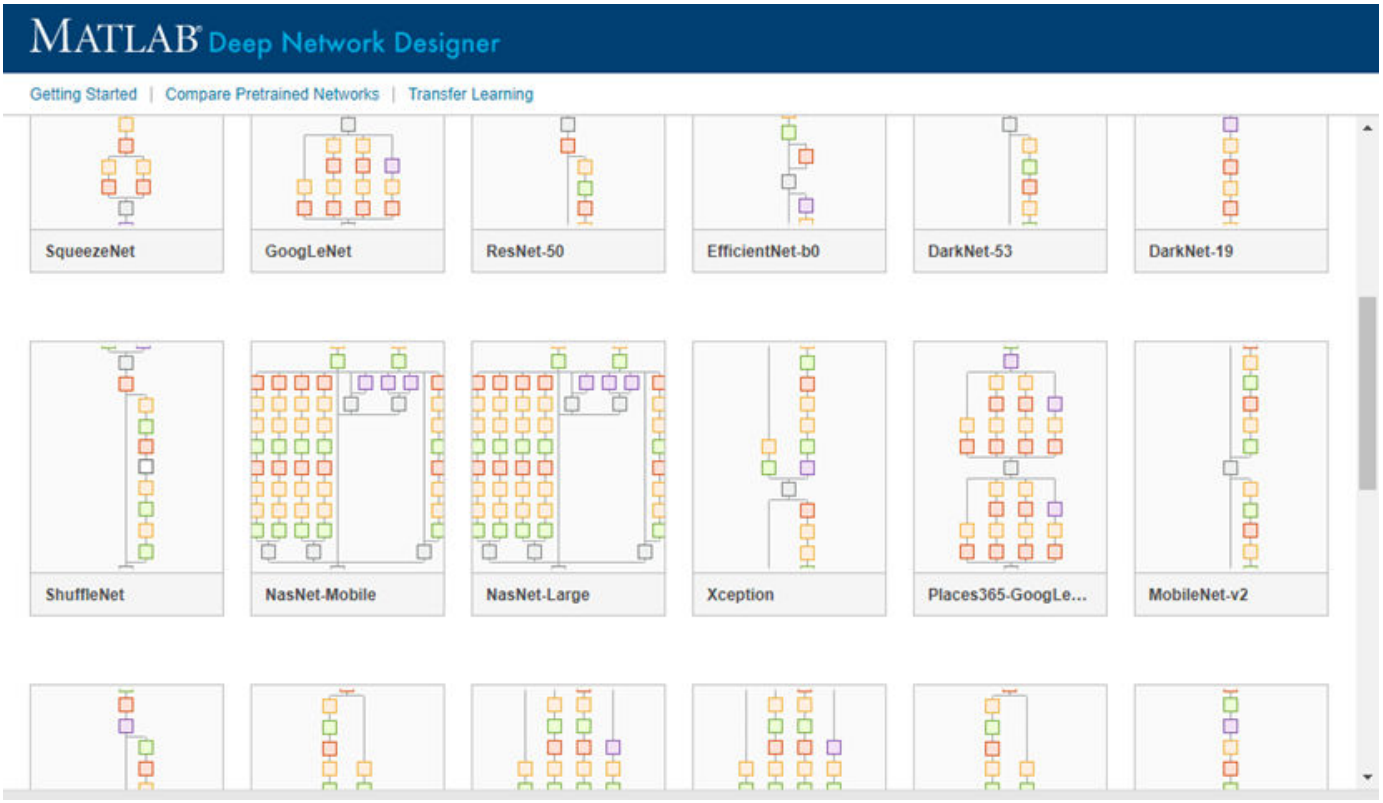
```
darknet53
```

```
ans =

  DAGNetwork with properties:

         Layers: [184×1 nnet.cnn.layer.Layer]
    Connections: [206×2 table]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(darknet53)
```

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



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

Transfer Learning with DarkNet-53

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the DarkNet-53 network instead of GoogLeNet.

```
net = darknet53
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

Output Arguments

`net` — Pretrained DarkNet-53 convolutional network
`DAGNetwork`

Pretrained DarkNet-53 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained DarkNet-53 convolutional neural network architecture
`LayerGraph` object

Untrained DarkNet-53 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Redmon, Joseph. “Darknet: Open Source Neural Networks in C.” https://pjreddie.com/darknet.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = darknet53` or by passing the `darknet53` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet53')`

The syntax `darknet53('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = darknet53` or by passing the `darknet53` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet53')`.

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `darknet53('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `darknet19` | `trainNetwork` | `DAGNetwork` | `layerGraph` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201` | `nasnetmobile` | `nasnetlarge`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2020a

deepDreamImage

Visualize network features using deep dream

Syntax

```
I = deepDreamImage(net,layer,channels)
I = deepDreamImage(net,layer,channels,Name,Value)
```

Description

`I = deepDreamImage(net,layer,channels)` returns an array of images that strongly activate the channels `channels` within the network `net` of the layer with numeric index or name given by `layer`. These images highlight the features learned by a network.

`I = deepDreamImage(net,layer,channels,Name,Value)` returns an image with additional options specified by one or more `Name,Value` pair arguments.

Examples

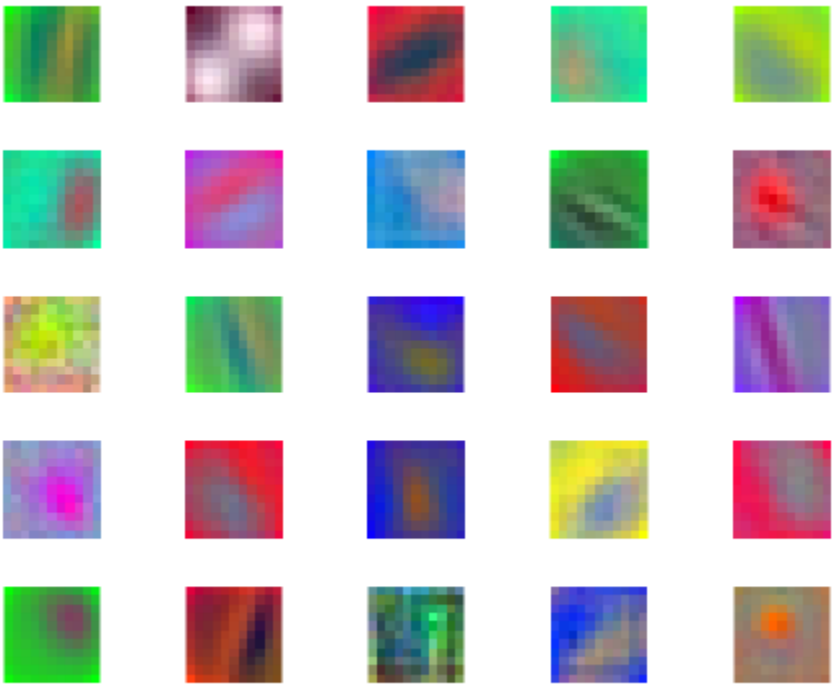
Visualize Convolutional Neural Network Features

Load a pretrained AlexNet network.

```
net = alexnet;
```

Visualize the first 25 features learned by the first convolutional layer (`'conv1'`) using `deepDreamImage`. Set `'PyramidLevels'` to 1 so that the images are not scaled.

```
layer = 'conv1';
channels = 1:25;

I = deepDreamImage(net,layer,channels, ...
    'PyramidLevels',1, ...
    'Verbose',0);

figure
for i = 1:25
    subplot(5,5,i)
    imshow(I(:,:,:,i))
end
```



Input Arguments

`net` — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` object or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function. For more information about pretrained networks, see “Pretrained Deep Neural Networks”.

`deepDreamImage` only supports networks with an image input layer.

`layer` — Layer index or name

positive integer | character vector | string scalar

Layer to visualize, specified as a positive integer, a character vector, or a string scalar. If `net` is a `DAGNetwork` object, specify `layer` as a character vector or string scalar only. Specify `layer` as the index or the name of the layer you want to visualize the activations of. To visualize classification layer features, select the last fully connected layer before the classification layer.

Tip Selecting ReLU or dropout layers for visualization may not produce useful images because of the effect that these layers have on the network gradients.

`channels` — Channel index

numeric index | vector of numeric indices

Queried channels, specified as scalar or vector of channel indices. If `channels` is a vector, the layer activations for each channel are optimized independently. The possible choices for `channels` depend on the selected layer. For convolutional layers, the `NumFilters` property specifies the number of output channels. For fully connected layers, the `OutputSize` property specifies the number of output channels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `deepDreamImage(net,layer,channels,'NumItetations',100,'ExecutionEnvironment','gpu')` generates images using 100 iterations per pyramid level and uses the GPU.

`InitialImage` — Image to initialize Deep Dream

array

Image to initialize Deep Dream. Use this syntax to see how an image is modified to maximize network layer activations. The minimum height and width of the initial image depend on all the layers up to and including the selected layer:

- For layers towards the end of the network, the initial image must be at least the same height and width as the image input layer.
- For layers towards the beginning of the network, the height and width of the initial image can be smaller than the image input layer. However, it must be large enough to produce a scalar output at the selected layer.
- The number of channels of the initial image must match the number of channels in the image input layer of the network.

If you do not specify an initial image, the software uses a random image with pixels drawn from a standard normal distribution. See also `'PyramidLevels'` on page 1-0 .

`PyramidLevels` — Number of pyramid levels

3 (default) | positive integer

Number of multi-resolution image pyramid levels to use to generate the output image, specified as a positive integer. Increase the number of pyramid levels to produce larger output images at the expense of additional computation. To produce an image of the same size as the initial image, set the number of levels to 1.

Example: `'PyramidLevels',3`

`PyramidScale` — Scale between pyramid levels

1.4 (default) | scalar with value > 1

Scale between each pyramid level, specified as a scalar with value > 1. Reduce the pyramid scale to incorporate fine grain details into the output image. Adjusting the pyramid scale can help generate more informative images for layers at the beginning of the network.

Example: `'PyramidScale',1.4`

`NumIterations` — Number of iterations per pyramid level
10 (default) | positive integer

Number of iterations per pyramid level, specified as a positive integer. Increase the number of iterations to produce more detailed images at the expense of additional computation.

Example: `'NumIterations',10`

`OutputScaling` — Type of scaling to apply to output
`'linear'` (default) | `'none'`

Type of scaling to apply to output image, specified as the comma-separated pair consisting of `'OutputScaling'` and one of the following:

Value	Description
`'linear'`	Scale output pixel values in the interval [0,1]. The output image corresponding to each layer channel, `I(:,:,:,channel)`, is scaled independently.
`'none'`	Disable output scaling.

Scaling the pixel values can cause the network to misclassify the output image. If you want to classify the output image, set the `'OutputScaling'` value to `'none'`.

Example: `'OutputScaling','linear'`

`Verbose` — Indicator to display progress information
1 (default) | 0

Indicator to display progress information in the command window, specified as the comma-separated pair consisting of `'Verbose'` and either 1 (`true`) or 0 (`false`). The displayed information includes the pyramid level, iteration, and the activation strength.

Example: `'Verbose',0`

Data Types: `logical`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following:

- `'auto'` — Use a GPU if one is available; otherwise, use the CPU.
- `'gpu'` — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'cpu'` — Use the CPU.

Example: `'ExecutionEnvironment','cpu'`

Output Arguments

`I` — Output image

array

Output image, specified by a sequence of grayscale or truecolor (RGB) images stored in a 4–D array. Images are concatenated along the fourth dimension of `I` such that the image that maximizes the output of `channels(k)` is `I(:,:,:,k)`. You can display the output image using `imshow`.

Algorithms

This function implements a version of deep dream that uses a multi-resolution image pyramid and Laplacian Pyramid Gradient Normalization to generate high-resolution images. For more information on Laplacian Pyramid Gradient Normalization, see this blog post: DeepDreaming with TensorFlow.

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

References

[1] *DeepDreaming with TensorFlow*. https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/deepdream.ipynb

See Also

`activations` | `alexnet` | `vgg16` | `vgg19` | `googlenet` | `squeezenet`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Deep Dream Images Using GoogLeNet”
“Visualize Features of a Convolutional Neural Network”
“Visualize Activations of a Convolutional Neural Network”
“Visualize Activations of LSTM Network”

Introduced in R2017a

densenet201

DenseNet-201 convolutional neural network

Syntax

```
net = densenet201
net = densenet201('Weights','imagenet')

lgraph = densenet201('Weights','none')
```

Description

DenseNet-201 is a convolutional neural network that is 201 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the DenseNet-201 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with DenseNet-201.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load DenseNet-201 instead of GoogLeNet.

`net = densenet201` returns a DenseNet-201 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model for DenseNet-201 Network support package. If this support package is not installed, then the function provides a download link.

`net = densenet201('Weights','imagenet')` returns a DenseNet-201 network trained on the ImageNet data set. This syntax is equivalent to `net = densenet201`.

`lgraph = densenet201('Weights','none')` returns the untrained DenseNet-201 network architecture. The untrained model does not require the support package.

Examples

Download DenseNet-201 Support Package

Download and install the Deep Learning Toolbox Model *for DenseNet-201 Network* support package.

Type `densenet201` at the command line.

```
densenet201
```

If the Deep Learning Toolbox Model *for DenseNet-201 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `densenet201` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

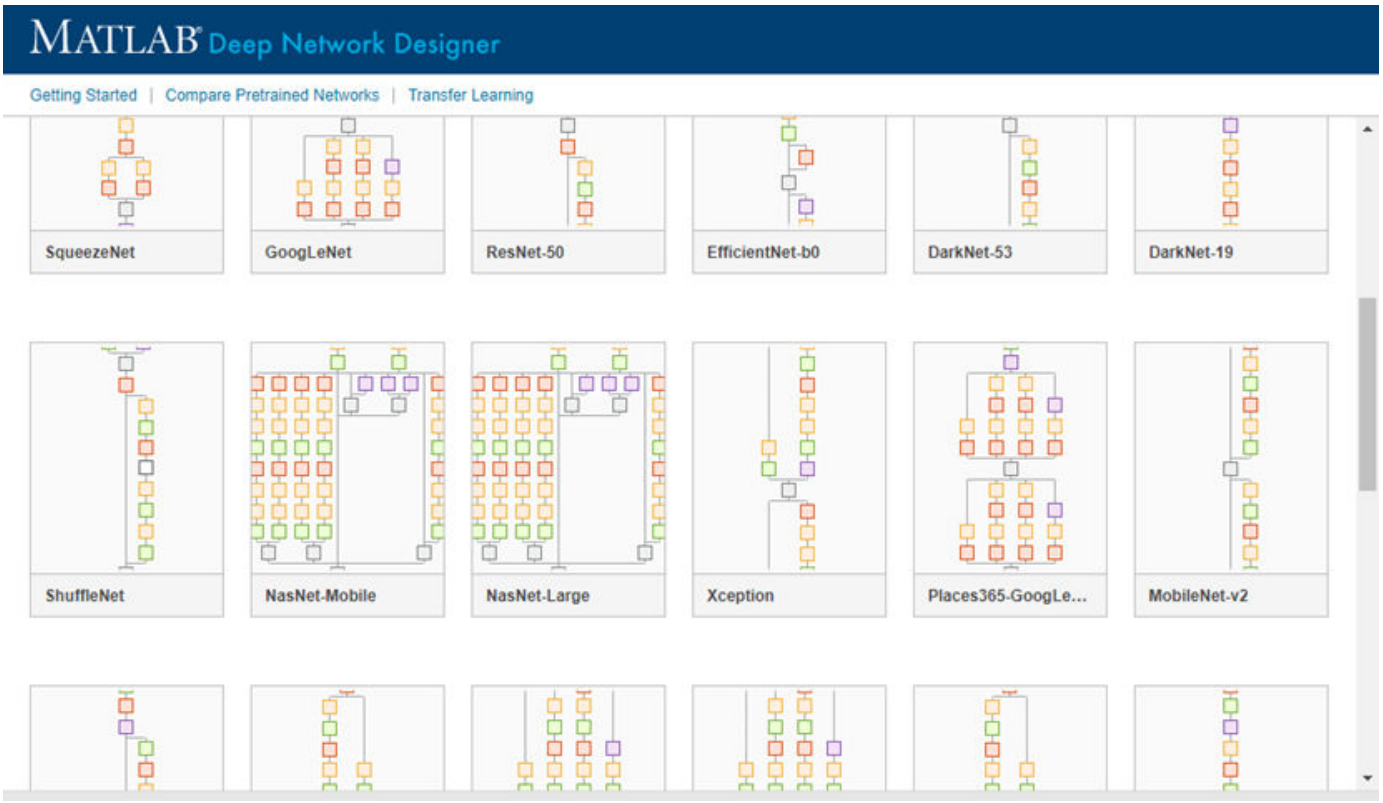
```
densenet201
```

```
ans =

  DAGNetwork with properties:

         Layers: [709×1 nnet.cnn.layer.Layer]
    Connections: [806×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(densenet201)
```

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



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

Output Arguments

net — Pretrained DenseNet-201 convolutional neural network

`DAGNetwork` object

Pretrained DenseNet-201 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained DenseNet-201 convolutional neural network architecture
`LayerGraph` object

Untrained DenseNet-201 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks." In *CVPR*, vol. 1, no. 2, p. 3. 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = densenet201` or by passing the `densenet201` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('densenet201')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `densenet201('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = densenet201` or by passing the `densenet201` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('densenet201')`.

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `densenet201('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `resnet18` | `resnet50` | `resnet101` | `googlenet` | `inceptionv3` | `inceptionresnetv2` | `squeezenet` | `trainNetwork` | `layerGraph` | `DAGNetwork`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a

depthConcatenationLayer

Depth concatenation layer

Description

A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).

Specify the number of inputs to the layer when you create it. The inputs have the names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer by using `connectLayers` or `disconnectLayers`.

Creation

Syntax

```
layer = depthConcatenationLayer(numInputs)
layer = depthConcatenationLayer(numInputs,'Name',name)
```

Description

`layer = depthConcatenationLayer(numInputs)` creates a depth concatenation layer that concatenates `numInputs` inputs along the third (channel) dimension. This function also sets the `NumInputs` property.

`layer = depthConcatenationLayer(numInputs,'Name',name)` also sets the `Name` property.

Properties

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer greater than or equal to 2.

The inputs have the names `'in1','in2',...,'inN'`, where `N` is `NumInputs`. For example, if `NumInputs` is 3, then the inputs have the names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer using the `connectLayers` or `disconnectLayers` functions.

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`InputNames` — Input Names
`{'in1','in2',…,'inN'}` (default)

Input names, specified as `{'in1','in2',...,'inN'}`, where `N` is the number of inputs of the layer.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create and Connect Depth Concatenation Layer

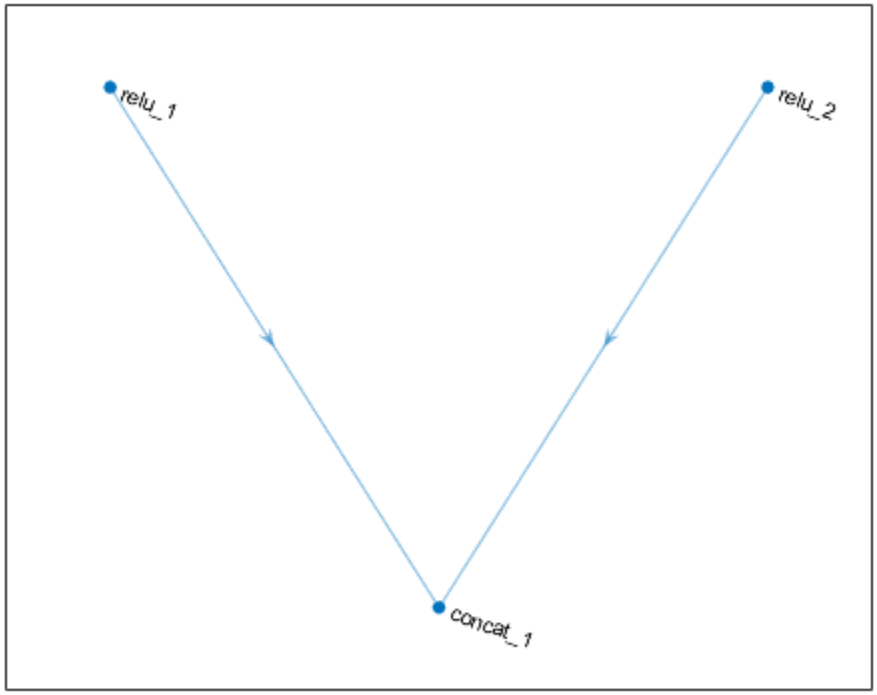
Create a depth concatenation layer with two inputs and the name `'concat_1'`.

```
concat = depthConcatenationLayer(2,'Name','concat_1')
```

```
concat =
  DepthConcatenationLayer with properties:

         Name: 'concat_1'
    NumInputs: 2
   InputNames: {'in1'  'in2'}
```

Create two ReLU layers and connect them to the depth concatenation layer. The depth concatenation layer concatenates the outputs from the ReLU layers.

```
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph;
lgraph = addLayers(lgraph,relu_1);
lgraph = addLayers(lgraph,relu_2);
lgraph = addLayers(lgraph,concat);

lgraph = connectLayers(lgraph,'relu_1','concat_1/in1');
lgraph = connectLayers(lgraph,'relu_2','concat_1/in2');

plot(lgraph)
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
`trainNetwork` | `layerGraph` | `additionLayer` | `connectLayers` | `disconnectLayers`

Topics
“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”

Introduced in R2017b

dims

Dimension labels of `dlarray`

Syntax

```
d = dims(dlX)
```

Description

`d = dims(dlX)` returns the data format of `dlX` as a character array. The data format provides the dimension labels for each dimension in `dlX`.

Examples

Obtain Dimension Labels

Obtain the dimension labels of a formatted `dlarray`.

```
dlX = dlarray(randn(3,4),'TS');
d = dims(dlX)
```

```
d =
'ST'
```

Obtain the labels of an unformatted `dlarray`.

```
y = stripdims(dlX);
d = dims(y)
```

```
d =

  0x0 empty char array
```

Input Arguments

`dlX` — Input `dlarray`

`dlarray` object

Input `dlarray`, specified as a `dlarray` object.

Example: `dlX = dlarray(randn(3,4),'ST')`

Output Arguments

`d` — Dimension labels

character vector

Dimension labels, returned as a character vector. If the input `dlX` is unformatted, `d` is empty.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`finddim` | `stripdims` | `dlarray`

Introduced in R2019b

dlaccelerate

Accelerate deep learning function for custom training loops

Syntax

```
accfun = dlaccelerate(fun)
```

Description

Use `dlaccelerate` to speed up deep learning function evaluation for custom training loops.

The returned `AcceleratedFunction` object caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` objects, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

Invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Note When using the `dlfeval` function, the software automatically accelerates the `forward` and `predict` functions for `dlnetwork` input. If you accelerate a deep learning function where the majority of the computation takes place in calls to the `forward` or `predict` functions for `dlnetwork` input, then you might not see an improvement in training time.

For more information, see “Deep Learning Function Acceleration for Custom Training Loops”.

`accfun = dlaccelerate(fun)` creates an `AcceleratedFunction` object that retains the underlying traces of the specified function handle `fun`.

Caution An `AcceleratedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the accelerated function, then clear the cache using the `clearCache` object function or alternatively use the command `clear functions`.

Examples

Accelerate Model Gradients Function

Load the `dlnetwork` object and class names from the MAT file `dlnetDigits.mat`.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Accelerate the model gradients function `modelGradients` listed at the end of the example.

```
fun = @modelGradients;
accfun = dlaccelerate(fun);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

View the properties of the accelerated function. Because the cache is empty, the `Occupancy` property is 0.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

         Function: @modelGradients
          Enabled: 1
        CacheSize: 50
          HitRate: 0
        Occupancy: 0
        CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

The returned `AcceleratedFunction` object stores the traces of underlying function calls and reuses the cached result when the same input pattern reoccurs. To use the accelerated function in a custom training loop, replace calls to the model gradients function with calls to the accelerated function. You can invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Evaluate the accelerated model gradients function with random data using the `dlfeval` function.

```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');

T = categorical(classNames(randi(10,[128 1])));
T = onehotencode(T,2)';
dlT = dlarray(T,'CB');

[gradients,state,loss] = dlfeval(accfun,dlnet,dlX,dlT);
```

View the `Occupancy` property of the accelerated function. Because the function has been evaluated, the cache is nonempty.

```
accfun.Occupancy
```

```
ans = 2
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `dlT` 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, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,dlT)
```

```
[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,dlT);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Clear Cache of Accelerated Function

Load the `dlnetwork` object and class names from the MAT file `dlnetDigits.mat`.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Accelerate the model gradients function `modelGradients` listed at the end of the example.

```
fun = @modelGradients;
accfun = dlaccelerate(fun);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

View the properties of the accelerated function. Because the cache is empty, the `Occupancy` property is 0.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @modelGradients
           Enabled: 1
         CacheSize: 50
           HitRate: 0
         Occupancy: 0
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

The returned `AcceleratedFunction` object stores the traces of underlying function calls and reuses the cached result when the same input pattern reoccurs. To use the accelerated function in a custom training loop, replace calls to the model gradients function with calls to the accelerated function. You can invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Evaluate the accelerated model gradients function with random data using the `dlfeval` function.

```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');

T = categorical(classNames(randi(10,[128 1])));
T = onehotencode(T,2)';
dlT = dlarray(T,'CB');

[gradients,state,loss] = dlfeval(accfun,dlnet,dlX,dlT);
```

View the `Occupancy` property of the accelerated function. Because the function has been evaluated, the cache is nonempty.

```
accfun.Occupancy
```

```
ans = 2
```

Clear the cache using the `clearCache` function.

```
clearCache(accfun)
```

View the `Occupancy` property of the accelerated function. Because the cache has been cleared, the cache is empty.

```
accfun.Occupancy
```

```
ans = 0
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `dlT` 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, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,flT)

[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,flT);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Check Accelerated Deep Learning Function Outputs

This example shows how to check that the outputs of accelerated functions match the outputs of the underlying function.

In some cases, the outputs of accelerated functions differ to the outputs of the underlying function. For example, you must take care when accelerating functions that use random number generation, such as a function that generates random noise to add to the network input. When caching the trace of a function that generates random numbers that are not `dlarray` objects, the accelerated function caches resulting random numbers in the trace. When reusing the trace, the accelerated function uses the cached random values. The accelerated function does not generate new random values.

To check that the outputs of the accelerated function match the outputs of the underlying function, use the `CheckMode` property of the accelerated function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ by more than a specified tolerance, the accelerated function throws a warning.

Accelerate the function `myUnsupportedFun`, listed at the end of the example using the `dlaccelerate` function. The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
accfun = dlaccelerate(@myUnsupportedFun)
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @myUnsupportedFun
           Enabled: 1
         CacheSize: 50
           HitRate: 0
         Occupancy: 0
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun.CheckMode = 'tolerance'
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @myUnsupportedFun
           Enabled: 1
         CacheSize: 50
           HitRate: 0
         Occupancy: 0
         CheckMode: 'tolerance'
    CheckTolerance: 1.0000e-04
```

Evaluate the accelerated function with an array of ones as input, specified as a `dlarray` input.

```
dlX = dlarray(ones(3,3));
dlY = accfun(dlX)
```

```
dlY =
  3x3 dlarray

    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Evaluate the accelerated function again with the same input. Because the accelerated function reuses the cached random noise values instead of generating new random values, the outputs of the reused trace differs from the outputs of the underlying function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ, the accelerated function throws a warning.

```
dlY = accfun(dlX)
```

```
Warning: Accelerated outputs differ from underlying function outputs.
```

```
dlY =
  3x3 dlarray
```

```
    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Random number generation using the `'like'` option of the `rand` function with a `dlarray` object supports acceleration. To use random number generation in an accelerated function, ensure that the function uses the `rand` function with the `'like'` option set to a traced `dlarray` object (a `dlarray` object that depends on an input `dlarray` object).

Accelerate the function `mySupportedFun`, listed at the end of the example. The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
accfun2 = dlaccelerate(@mySupportedFun);
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun2)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun2.CheckMode = 'tolerance';
```

Evaluate the accelerated function twice with the same input as before. Because the outputs of the reused cache match the outputs of the underlying function, the accelerated function does not throw a warning.

```
dlY = accfun2(dlX)
```

```
dlY =
  3x3 dlarray

    1.7922    1.0357    1.6787
    1.9595    1.8491    1.7577
    1.6557    1.9340    1.7431
```

```
dlY = accfun2(dlX)
```

```
dlY =
  3x3 dlarray

    1.3922    1.7060    1.0462
    1.6555    1.0318    1.0971
    1.1712    1.2769    1.8235
```

Checking the outputs match requires extra processing and increases the time required for function evaluation. After checking the outputs, set the `CheckMode` property to `'none'`.

```
accfun1.CheckMode = 'none';
accfun2.CheckMode = 'none';
```

Example Functions

The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
function out = myUnsupportedFun(dlX)

sz = size(dlX);
noise = rand(sz);
out = dlX + noise;

end
```

The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
function out = mySupportedFun(dlX)

sz = size(dlX);
noise = rand(sz,'like',dlX);
out = dlX + noise;

end
```

Input Arguments

`fun` — Deep learning function

function handle

Deep learning function to accelerate, specified as a function handle.

To learn more about developing deep learning functions for acceleration, see “Deep Learning Function Acceleration for Custom Training Loops”.

Example: `@modelGradients`

Data Types: `function_handle`

Output Arguments

`accfun` — Accelerated deep learning function

`AcceleratedFunction` object

Accelerated deep learning function, returned as an `AcceleratedFunction` object.

More About

Acceleration Considerations

Because of the nature of caching traces, not all functions support acceleration.

The caching process can cache values that you might expect to change or that depend on external factors. You must take care when accelerating functions that:

- have inputs with random or frequently changing values
- have outputs with frequently changing values
- generate random numbers
- use `if` statements and `while` loops with conditions that depend on the values of `dlarray` objects
- have inputs that are handles or that depend on handles
- Read data from external sources (for example, by using a datastore or a `minibatchqueue` object)

Accelerated functions can do the following when calculating a new trace only.

- modify the global state such as, the random number stream or global variables
- use file input or output
- display data using graphics or the command line display

When using accelerated functions in parallel, such as when using a `parfor` loop, then each worker maintains its own cache. The cache is not transferred to the host.

Functions and custom layers used in accelerated functions must also support acceleration.

For more information, see “Deep Learning Function Acceleration for Custom Training Loops”.

`dlode45` Does Not Support Acceleration When `GradientMode` is `"direct"`

The `dlaccelerate` function does not support accelerating the `dlode45` function when the `GradientMode` option is `"direct"`. The resulting accelerated function might return unexpected results. To accelerate the code that calls the `dlode45` function, set the `GradientMode` option to `"adjoint"` or accelerate parts of your code that do not call the `dlode45` with the `GradientMode` option set to `"direct"`.

See Also

`AcceleratedFunction` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Topics

“Deep Learning Function Acceleration for Custom Training Loops”
“Accelerate Custom Training Loop Functions”
“Check Accelerated Deep Learning Function Outputs”
“Evaluate Performance of Accelerated Deep Learning Function”

Introduced in R2021a

dlarray

Deep learning array for custom training loops

Description

A deep learning array stores data with optional data format labels for custom training loops, and enables functions to compute and use derivatives through automatic differentiation.

Tip 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". Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see "Define Deep Learning Network for Custom Training Loops".

Creation

Syntax

```
dlX = dlarray(X)
dlX = dlarray(X,fmt)
dlX = dlarray(v,dim)
```

Description

`dlX = dlarray(X)` returns a `dlarray` object representing `X`. If `X` is a `dlarray`, `dlX` is a copy of `X`.

`dlX = dlarray(X,fmt)` formats the data in `dlX` with dimension labels according to the data format in `fmt`. Dimension labels help in passing deep learning data between functions. For more information on dimension labels, see "Usage" on page 1-420. If `X` is a formatted `dlarray`, then `fmt` replaces the existing format.

`dlX = dlarray(v,dim)` accepts a vector `v` and a single character format `dim`, and returns a column vector `dlarray`. The first dimension of `dlX` has the dimension label `dim`, and the second (singleton) dimension has the dimension label `'U'`.

Input Arguments

X — Data array
numeric array of data type `double` or `single` | `logical` array | `gpuArray` object | `dlarray` object

Data array, specified as a numeric array of data type `double` or `single`, a `logical` array, a `gpuArray` object, or a `dlarray` object. `X` must be full, not sparse.

Example: `rand(31*23,23)`

Data Types: `single` | `double` | `logical`
Complex Number Support: Yes

`fmt` — Data format
character vector | string scalar

Data format, specified as a character vector or string scalar. Each character in `fmt` must be one of the following dimension labels:

- `S` — Spatial
- `C` — Channel
- `B` — Batch observations
- `T` — Time or sequence
- `U` — Unspecified

You can specify any number of `S` and `U` labels. You can specify at most one of each of the `C`, `B`, and `T` labels.

Each element of `fmt` labels the matching dimension of `dlX`. If `fmt` is not in the listed order (`'S'` followed by `'C'` and so on), then `dlarray` implicitly permutes both `fmt` and the data to match the order without changing the storage of the data.

`fmt` must contain at least the same number of dimension labels as the number of dimensions of `dlX`. If you specify more than that number of dimension labels, `dlarray` creates empty (singleton) dimensions for the additional labels.

The following table indicates recommended data formats for common types of data.

Data	**Example**	
	Shape	**Data Format**
2-D images	*h*-by-*w*-by-*c*-by-*n* numeric array, where *h*, *w*, *c* and *n* are the height, width, number of channels of the images, and number of observations, respectively.	`"SSCB"`
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*n* numeric array, where *h*, *w*, *d*, *c* and *n* are the height, width, depth, number of channels of the images, and number of image observations, respectively.	`"SSSCB"`
Vector sequences	*c*-by-*s*-by-*n* matrix, where *c* is the number of features of the sequence, *s* is the sequence length, and *n* is the number of sequence observations.	`"CTB"`

Data	Example	
	Shape	Data Format
2-D image sequences	*h*-by-*w*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations.	`"SSCTB"`
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations.	`"SSSCTB"`
Features	*c*-by-*n* array, where *c* is the number of features, and *n* is the number of observations.	`"CB"`

For information on how data formats are used in deep learning functions, see “Usage” on page 1-420.

Example: `"SSB"`

Example: `'CBUSS'`, which `dlarray` reorders to `'SSCBU'`

v — Data vector

numeric vector of data type double or single | logical vector | `dlarray` vector object

Data vector, specified as a numeric vector of data type double or single, logical vector, or `dlarray` vector object. Here, "vector" means any array with exactly one nonsingleton dimension.

Example: `rand(100,1)`

dim — Dimension label

single character

Dimension label, specified as a single character of the type allowed for `fmt`.

Example: `"S"`

Example: `'S'`

Output Arguments

dlX — Deep learning array

`dlarray` object

Deep learning array, returned as a `dlarray` object. `dlX` enables automatic differentiation using `dlgradient` and `dlfeval`. If you supply the `fmt` argument, `dlX` has labels.

- If `X` is a numeric or logical array, `dlX` contains its data, possibly reordered because of the data format in `fmt`.

- If `X` is a `gpuArray`, the data in `dlX` is also on the GPU. Subsequent calculations using `dlX` are performed on the GPU.

Usage

`dlarray` data formats enable you to execute the functions in the following table with assurance that the data has the appropriate shape.

Function	Operation	Validates Input Dimension	Affects Size of Input Dimension
`avgpool`	Compute the average of the input data over moving rectangular (or cuboidal) spatial (`'S'`) regions defined by a pool size parameter.	`'S'`	`'S'`
`batchnorm`	Normalize the values contained in each channel (`'C'`) of the input data.	`'C'`	
`crossentropy`	Compute the cross-entropy between estimates and target values, averaged by the size of the batch (`'B'`) dimension.	`'S'`, `'C'`, `'B'`, `'T'`, `'U'` (Estimates and target arrays must have the same sizes.)	`'S'`, `'C'`, `'B'`, `'T'`, `'U'` (Output is an unformatted scalar.)
`dlconv`	Compute the deep learning convolution of the input data using an array of filters, matching the number of spatial (`'S'`) and (a function of the) channel (`'C'`) dimensions of the input, and adding a constant bias.	`'S'`, `'C'`	`'S'`, `'C'`
`dltranspconv`	Compute the deep learning transposed convolution of the input data using an array of filters, matching the number of spatial (`'S'`) and (a function of the) channel (`'C'`) dimensions of the input, and adding a constant bias.	`'S'`, `'C'`	`'S'`, `'C'`
`fullyconnect`	Compute a weighted sum of the input data and apply a bias for each batch (`'B'`) and time (`'T'`) dimension.	`'S'`, `'C'`, `'U'`	`'S'`, `'C'`, `'B'`, `'T'`, `'U'` (Output always has data format `'CB'`, `'CT'`, or `'CTB'`.)
`gru`	Apply a gated recurrent unit calculation to the input data.	`'S'`, `'C'`, `'T'`	`'C'`
`lstm`	Apply a long short-term memory calculation to the input data.	`'S'`, `'C'`, `'T'`	`'C'`

Function	Operation	Validates Input Dimension	Affects Size of Input Dimension
`maxpool`	Compute the maximum of the input data over moving rectangular spatial (`'S'`) regions defined by a pool size parameter.	`'S'`	`'S'`
`maxunpool`	Compute the unpooling operation over the spatial (`'S'`) dimensions.	`'S'`	`'S'`
`mse`	Compute the half mean squared error between estimates and target values, averaged by the size of the batch (`'B'`) dimension.	`'S'`, `'C'`, `'B'`, `'T'`, `'U'` (Estimates and target arrays must have the same sizes.)	`'S'`, `'C'`, `'B'`, `'T'`, `'U'` (Output is an unformatted scalar.)
`softmax`	Apply the softmax activation to each channel (`'C'`) of the input data.	`'C'`	

These functions require each dimension to have a label. You can specify the dimension label format by providing the first input as a formatted `dlarray`, or by using the `'DataFormat'` name-value argument of the function.

`dlarray` enforces the dimension label ordering of `'SCBTU'`. This enforcement eliminates ambiguous semantics in operations which implicitly match labels between inputs. `dlarray` also enforces that the dimension labels `'C'`, `'B'`, and `'T'` can each appear at most once. The functions that use these dimension labels accept at most one dimension for each label.

`dlarray` provides functions for obtaining the data format associated with a `dlarray` (`dims`), removing the data format (`stripdims`), and obtaining the dimensions associated with specific dimension labels (`finddim`).

For more information on how a `dlarray` behaves with formats, see “Notable dlarray Behaviors”.

Object Functions

avgpool	Pool data to average values over spatial dimensions
batchnorm	Normalize data across all observations for each channel independently
crossentropy	Cross-entropy loss for classification tasks
dims	Dimension labels of dlarray
dlconv	Deep learning convolution
dlgradient	Compute gradients for custom training loops using automatic differentiation
dltranspconv	Deep learning transposed convolution
extractdata	Extract data from dlarray
finddim	Find dimensions with specified label
fullyconnect	Sum all weighted input data and apply a bias
gru	Gated recurrent unit
leakyrelu	Apply leaky rectified linear unit activation
lstm	Long short-term memory
maxpool	Pool data to maximum value
maxunpool	Unpool the output of a maximum pooling operation

mse	Half mean squared error
relu	Apply rectified linear unit activation
sigmoid	Apply sigmoid activation
softmax	Apply softmax activation to channel dimension
stripdims	Remove dlarray data format

A `dlarray` also allows functions for numeric, matrix, and other operations. See the full list in “List of Functions with dlarray Support”.

Examples

Create Unformatted `dlarray`

Create an unformatted `dlarray` from a matrix.

```
X = randn(3,5);
dlX = dlarray(X)
```

```
dlX =
  3x5 dlarray

    0.5377    0.8622   -0.4336    2.7694    0.7254
    1.8339    0.3188    0.3426   -1.3499   -0.0631
   -2.2588   -1.3077    3.5784    3.0349    0.7147
```

Create Formatted `dlarray`

Create a `dlarray` that has a data format containing the dimension labels `'S'` and `'C'`.

```
X = randn(3,5);
dlX = dlarray(X,'SC')
```

```
dlX =
  3(S) x 5(C) dlarray

    0.5377    0.8622   -0.4336    2.7694    0.7254
    1.8339    0.3188    0.3426   -1.3499   -0.0631
   -2.2588   -1.3077    3.5784    3.0349    0.7147
```

If you specify the dimension labels in the opposite order, `dlarray` implicitly reorders the underlying data.

```
dlX = dlarray(X,'CS')
```

```
dlX =
  5(S) x 3(C) dlarray

    0.5377    1.8339   -2.2588
    0.8622    0.3188   -1.3077
   -0.4336    0.3426    3.5784
    2.7694   -1.3499    3.0349
```

```
    0.7254   -0.0631    0.7147
```

Create Formatted `dlarray` Vector

Create a `dlarray` vector with the first dimension label `'T'`. The second dimension label, which `dlarray` creates automatically, is `'U'`.

```
X = randn(6,1);
dlX = dlarray(X,'T')

dlX =
  6(T) x 1(U) dlarray

    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
```

If you specify a row vector for `X`, `dlarray` implicitly reorders the result to be a column vector.

```
X = X';
dlX = dlarray(X,'T')

dlX =
  6(T) x 1(U) dlarray

    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
```

Tips

- A `dlgradient` call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using `dlfeval`, and the argument to the function must be a `dlarray`. See “Use Automatic Differentiation In Deep Learning Toolbox”.
- To enable the correct evaluation of gradients, `dlfeval` must call functions that use only supported functions for `dlarray`. See “List of Functions with dlarray Support”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For usage recommendations and list of functions with `dlarray` code generation support, see “Code Generation for dlarray” (MATLAB Coder). For an example showing how to use `dlnetwork` and `dlarray` objects to generate MEX for a trained variational autoencoder (VAE) network, see “Generate Digit Images Using Variational Autoencoder on Intel CPUs” (MATLAB Coder).

For `dlarray` code generation limitations, see “dlarray Limitations for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For usage recommendations and list of functions with `dlarray` code generation support, see “Code Generation for dlarray” (GPU Coder). For an example showing how to use `dlnetwork` and `dlarray` objects to generate CUDA® MEX for a trained variational autoencoder (VAE) network, see “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder” (GPU Coder).

For `dlarray` code generation limitations, see “dlarray Limitations for Code Generation” (GPU Coder).

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`dlgradient` | `dlfeval` | `dlnetwork` | `finddim` | `dims` | `stripdims`

Topics
“Train Generative Adversarial Network (GAN)”
“Define Custom Training Loops, Loss Functions, and Networks”
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”

Introduced in R2019b

dlconv

Deep learning convolution

Syntax

```
dlY = dlconv(dlX,weights,bias)
dlY = dlconv(dlX,weights,bias,'DataFormat',FMT)
dlY = dlconv( ___ ,Name,Value)
```

Description

The convolution operation applies sliding filters to the input data. Use the `dlconv` function for deep learning convolution, grouped convolution, and channel-wise separable convolution.

The `dlconv` function applies the deep learning convolution operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply convolution within a `layerGraph` object or `Layer` array, use one of the following layers:

- `convolution2dLayer`
- `groupedConvolution2dLayer`
- `convolution3dLayer`

`dlY = dlconv(dlX,weights,bias)` applies the deep learning convolution operation to the formatted `dlarray` object `dlX`. The function uses sliding convolutional filters defined by `weights` and adds the constant `bias`. The output `dlY` is a formatted `dlarray` object with the same format as `dlX`.

The function, by default, convolves over up to three dimensions of `dlX` labeled `'S'` (spatial). To convolve over dimensions labeled `'T'` (time), specify `weights` with a `'T'` dimension using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

For unformatted input data, use the `'DataFormat'` option.

`dlY = dlconv(dlX,weights,bias,'DataFormat',FMT)` applies the deep learning convolution operation to the unformatted `dlarray` object `dlX` with format specified by `FMT` using any of the previous syntaxes. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D convolution with format `'SSCB'` (spatial, spatial, channel, batch).

`dlY = dlconv( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'WeightsFormat','TCU'` specifies weights for 1-D convolution with format `'TCU'` (time, channel, unspecified).

Examples

Perform 2-D Convolution

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Initialize the weights and bias for 2-D convolution. For the weights, specify 64 3-by-3 filters. For the bias, specify a vector of zeros.

```
filterSize = [3 3];
numFilters = 64;
weights = rand(filterSize(1),filterSize(2),numChannels,numFilters);
bias = zeros(1,numFilters);
```

Apply 2-D convolution using the `dlconv` function.

```
dlY = dlconv(dlX,weights,bias);
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

    26    26    64   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Perform Grouped Convolution

Convolve the input data in three groups of two channels each. Apply four filters per group.

Create the input data as 10 observations of size 100-by-100 with six channels.

```
height = 100;
width = 100;
channels = 6;
numObservations = 10;

X = rand(height,width,channels,numObservations);
dlX = dlarray(X,'SSCB');
```

Initialize the convolutional filters. Specify three groups of convolutions that each apply four convolution filters to two channels of the input data.

```
filterHeight = 8;
filterWidth = 8;
numChannelsPerGroup = 2;
numFiltersPerGroup = 4;
numGroups = 3;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);
```

Initialize the bias term.

```
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the convolution.

```
dlY = dlconv(dlX,weights,bias);
size(dlY)
```

```
ans = 1×4

    93    93    12    10
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

The 12 channels of the convolution output represent the three groups of convolutions with four filters per group.

Perform Channel-Wise Separable Convolution

Separate the input data into channels and perform convolution on each channel separately.

Create the input data as a single observation with a size of 64-by-64 and 10 channels. Create the data as an unformatted `dlarray`.

```
height = 64;
width = 64;
channels = 10;

X = rand(height,width,channels);
dlX = dlarray(X);
```

Initialize the convolutional filters. Specify an ungrouped convolution that applies a single convolution to all three channels of the input data.

```
filterHeight = 8;
filterWidth = 8;
numChannelsPerGroup = 1;
numFiltersPerGroup = 1;
numGroups = channels;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);
```

Initialize the bias term.

```
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the convolution. Specify the dimension labels of the input data using the `'DataFormat'` option.

```
dlY = dlconv(dlX,weights,bias,'DataFormat','SSC');
size(dlY)
```

```
ans = 1×3

    57    57    10
```

Each channel is convolved separately, so there are 10 channels in the output.

Perform 1-D Convolution

Create a formatted `dlarray` object containing 128 sequences of length 512 containing 5 features. Specify the format `'CBT'` (channel, batch, time).

```
numChannels = 5;
miniBatchSize = 128;
sequenceLength = 512;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

Initialize the weights and bias for 1-D convolution. For the weights, specify 64 filters with a filter size of 3. For the bias, specify a vector of zeros.

```
filterSize = 3;
numFilters = 64;
weights = rand(filterSize,numChannels,numFilters);
bias = zeros(1,numFilters);
```

Apply 1-D convolution using the `dlconv` function. To convolve over the `'T'` (time) dimension of the input data, specify the weights format `'TCU'` (time, channel, unspecified) using the `'WeightsFormat'` option.

```
dlY = dlconv(dlX,weights,bias,'WeightsFormat','TCU');
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×3

    64   128   510
```

```
dims(dlY)
```

```
ans =
'CBT'
```

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `dlX` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `weights` or `bias` must be a `dlarray` object.

The function, by default, convolves over up to three dimensions of `dlX` labeled `'S'` (spatial). To convolve over dimensions labeled `'T'` (time), specify `weights` with a `'T'` dimension using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

`weights` — Convolutional filters

`dlarray` | numeric array

Convolutional filters, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

The size and format of the weights depends on the type of task. If `weights` is an unformatted `dlarray` or a numeric array, then the size and shape of `weights` depends on the `'WeightsFormat'` option.

The following table describes the size and format of the weights for various tasks. You can specify an array with the dimensions in any order using formatted `dlarray` objects or by using the `'WeightsFormat'` option. When the weights has multiple dimensions with the same label (for example, multiple dimensions labeled `'S'`), then those dimensions must be in ordered as described in this table.

Task	Required Dimensions	Size	Example	
			Weights	Format
1-D convolution	`'S'` (spatial) or `'T'` (time)	Filter size	`filterSize`-by-`numChannels`-by-`numFilters` array, where `filterSize` is the size of the 1-D filters, `numChannels` is the number of channels of the input data, and `numFilters` is the number of filters.	`'SCU'` (spatial, channel, unspecified)
	`'C'` (channel)	Number of channels		
	`'U'` (unspecified)	Number of filters		
1-D grouped convolution	`'S'` (spatial) or `'T'` (time)	Filter size	`filterSize`-by-`numChannelsPerGroup`-by-`numFiltersPerGroup`-by-`numGroups` array, where `filterSize` is the size of the 1-D filters, `numChannelsPerGroup` is the number of channels per group of the input data, and `numFiltersPerGroup` is the number of filters per group. `numChannelsPerGroup` must equal the number of the channels of the input data divided by `numGroups`.	`'SCUU'` (spatial, channel, unspecified, unspecified)
	`'C'` (channel)	Number of channels per group		
	First `'U'` (unspecified)	Number of filters per group		
	Second `'U'` (unspecified)	Number of groups		
2-D convolution	First `'S'` (spatial)	Filter height	`filterSize(1)`-by-`filterSize(2)`-by-`numChannels`-by-`numFilters` array, where `filterSize(1)` and	`'SSCU'` (spatial, spatial, channel, unspecified)
	Second `'S'` (spatial) or `'T'` (time)	Filter width		
	`'C'` (channel)	Number of channels		

Task	Required Dimensions	Size	Example	
			Weights	Format
	`'U'` (unspecified)	Number of filters	`filterSize(2)` are the height and width of the 2-D filters, respectively, `numChannels` is the number of channels of the input data, and `numFilters` is the number of filters.	
2-D grouped convolution	First `'S'` (spatial)	Filter height	`filterSize(1)`-by-`filterSize(2)`-by-`numChannelsPerGroup`-by-`numFiltersPerGroup`-by-`numGroups` array, where `filterSize(1)` and `filterSize(2)` are the height and width of the 2-D filters, respectively, `numChannelsPerGroup` is the number of channels per group of the input data, and `numFiltersPerGroup` is the number of filters per group. `numChannelsPerGroup` must equal the number of the channels of the input data divided by `numGroups`.	`'SSCUU'` (spatial, spatial, channel, unspecified, unspecified)
	Second `'S'` (spatial) or `'T'` (time)	Filter width		
	`'C'` (channel)	Number of channels per group		
	First `'U'` (unspecified)	Number of filters per group		
	Second `'U'` (unspecified)	Number of groups		
3-D convolution	First `'S'` (spatial)	Filter height	`filterSize(1)`-by-`filterSize(2)`-by-	`'SSSCU'` (spatial, spatial, spatial, channel, unspecified)
	Second `'S'` (spatial)	Filter width		

Task	Required Dimensions	Size	Example	
			Weights	Format
	Third `'S'` (spatial) or `'T'` (time)	Filter depth	`filterSize(3)`-by-`numChannels`-by-`numFilters` array, where `filterSize(1)`, `filterSize(2)`, and `filterSize(3)` are the height, width, and depth of the 3-D filters, respectively, `numChannels` is the number of channels of the input data, and `numFilters` is the number of filters.	
	`'C'` (channel)	Number of channels		
	`'U'` (unspecified)	Number of filters		

For channel-wise separable (also known as depth-wise separable) convolution, use grouped convolution with number of groups equal to the number of channels.

Tip The function, by default, convolves over up to three dimensions of `dlX` labeled `'S'` (spatial). To convolve over dimensions labeled `'T'` (time), specify `weights` with a `'T'` dimension using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

`bias` — Bias constant

`dlarray` | numeric vector | numeric scalar

Bias constant, specified as a formatted `dlarray`, an unformatted `dlarray`, a numeric vector, or a numeric scalar.

- If `bias` is a scalar, then the same bias is applied to each output.
- If `bias` has a nonsingleton dimension, then each element of `bias` is the bias applied to the corresponding convolutional filter specified by `weights`. The number of elements of `bias` must match the number of filters specified by `weights`.
- If `bias` is `0`, then the bias term is disabled and no bias is added during the convolution operation.

If `bias` is a formatted `dlarray`, then the nonsingleton dimension must be a channel dimension with label `'C'` (channel).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DilationFactor',2` sets the dilation factor for each convolutional filter to 2.

`DataFormat` — Dimension order of unformatted data

character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`WeightsFormat` — Dimension order of weights

character vector | string scalar

Dimension order of the weights, specified as the comma-separated pair consisting of `'WeightsFormat'` and a character vector or string scalar that provides a label for each dimension of the weights.

The default value of `'WeightsFormat'` depends on the task:

Task	Default
1-D convolution	`'SCU'` (spatial, channel, unspecified)
1-D grouped convolution	`'SCUU'` (spatial, channel, unspecified, unspecified)
2-D convolution	`'SSCU'` (spatial, spatial, channel, unspecified)
2-D grouped convolution	`'SSCUU'` (spatial, spatial, channel, unspecified, unspecified)
3-D convolution	`'SSSCU'` (spatial, spatial, spatial, channel, unspecified)

The supported combinations of dimension labels depends on the type of convolution, for more information, see the `weights` argument.

Tip The function, by default, convolves over up to three dimensions of `dlX` labeled `'S'` (spatial). To convolve over dimensions labeled `'T'` (time), specify `weights` with a `'T'` dimension using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

Example: `'WeightsFormat','TCU'`

Data Types: char | string

Stride — Step size for traversing input data
1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1.

Example: 'Stride',3

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

DilationFactor — Filter dilation factor
1 (default) | numeric scalar | numeric vector

Filter dilation factor, specified as the comma-separated pair consisting of 'DilationFactor' and one of the following.

- Numeric scalar — The same dilation factor value is applied for all spatial dimensions.
- Numeric vector — A different dilation factor value is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the dilation factor applied to the ith spatial dimension.

Use the dilation factor to increase the receptive field of the filter (the area of the input that the filter can see) on the input data. Using a dilation factor corresponds to an effective filter size of filterSize + (filterSize-1)*(dilationFactor-1).

Example: 'DilationFactor',2

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Padding — Size of padding
0 (default) | 'same' | 'causal' | numeric scalar | numeric vector | numeric matrix

Size of padding applied to the 'S' and 'T' dimensions given by the format of the weights, specified as the comma-separated pair consisting of 'Padding' and one of the following:

- 'same' — Apply padding such that the output dimension sizes are ceil(inputSize/stride), where inputSize is the size of the corresponding input dimension. When Stride is 1, the output is the same size as the input.
- 'causal' – Apply left padding with size (FilterSize - 1) .* DilationFactor. This option supports convolving over a single time or spatial dimension only. When Stride is 1, the output is the same size as the input.
- Nonnegative integer sz — Add padding of size sz to both ends of the 'S' or 'T' dimensions given by the format of the weights.
- Vector of integers sz — Add padding of size sz(i) to both ends of the ith 'S' or 'T' dimensions given by the format of the weights. The number of elements of sz must match the number of 'S' or 'T' dimensions of the weights.
- Matrix of integers sz — Add padding of size sz(1,i) and sz(2,i) to the start and end of the ith 'S' or 'T' dimensions given by the format of the weights. For example, for 2-D input, [t l; b

`r]` applies padding of size `t`, `b`, `l`, and `r` to the top, bottom, left, and right of the input, respectively.

Example: `'Padding','same'`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`PaddingValue` — Value to pad data
0 (default) | scalar | `'symmetric-include-edge'` | `'symmetric-exclude-edge'` | `'replicate'`

Value to pad data, specified as one of the following:

`PaddingValue`	Description	Example
Scalar	Pad with the specified scalar value.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&3&1&4&0&0\\0&0&1&5&9&0&0\\0&0&2&6&5&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}$
`'symmetric-include-edge'`	Pad using mirrored values of the input, including the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&1&1&5&9&9&5\\1&3&3&1&4&4&1\\1&3&3&1&4&4&1\\5&1&1&5&9&9&5\\6&2&2&6&5&5&6\\6&2&2&6&5&5&6\\5&1&1&5&9&9&5\end{bmatrix}$
`'symmetric-exclude-edge'`	Pad using mirrored values of the input, excluding the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\\9&5&1&5&9&5&1\\5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\end{bmatrix}$
`'replicate'`	Pad using repeated border elements of the input	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\3&3&3&1&4&4&4\\1&1&1&5&9&9&9\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\\2&2&2&6&5&5&5\end{bmatrix}$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Output Arguments

dlY — Convolved feature map

dlarray

Convolved feature map, returned as a dlarray with the same underlying data type as dlX.

If the input data dlX is a formatted dlarray, then dlY has the same format as dlX. If the input data is not a formatted dlarray, then dlY is an unformatted dlarray with the same dimension order as the input data.

The size of the 'C' (channel) dimension of dlY depends on the task.

Task	Size of 'C' Dimension
Convolution	Number of filters
Grouped convolution	Number of filters per group multiplied by the number of groups

More About

Deep Learning Convolution

The dlconv function applies sliding convolution filters to the input data. The dlconv function supports convolution in one, two, or three spatial dimensions or one time dimension. To learn more about deep learning convolution, see the definition of convolutional layer on page 1-329 on the convolution2dLayer reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.
  - dlX
  - weights
  - bias

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

dlarray | batchnorm | fullyconnect | maxpool | relu | dlgradient | dlfeval

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Sequence-to-Sequence Classification Using 1-D Convolutions”
“List of Functions with dlarray Support”

Introduced in R2019b

dlfeval

Evaluate deep learning model for custom training loops

Syntax

```
[y1,...,yk] = dlfeval(fun,x1,...,xn)
```

Description

Use `dlfeval` to evaluate custom deep learning models for custom training loops.

Tip 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". Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see "Define Deep Learning Network for Custom Training Loops".

`[y1,...,yk] = dlfeval(fun,x1,...,xn)` evaluates the deep learning array function `fun` at the input arguments `x1,...,xn`. Functions passed to `dlfeval` can contain calls to `dlgradient`, which compute gradients from the inputs `x1,...,xn` by using automatic differentiation.

Examples

Compute Gradient Using Automatic Differentiation

Rosenbrock's function is a standard test function for optimization. The `rosenbrock.m` helper function computes the function value and uses automatic differentiation to compute its gradient.

```
type rosenbrock.m
```

```
function [y,dydx] = rosenbrock(x)

y = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
dydx = dlgradient(y,x);

end
```

To evaluate Rosenbrock's function and its gradient at the point `[–1,2]`, create a `dlarray` of the point and then call `dlfeval` on the function handle `@rosenbrock`.

```
x0 = dlarray([-1,2]);
[fval,gradval] = dlfeval(@rosenbrock,x0)
```

```
fval =
  1x1 dlarray
```

```
   104


gradval =
  1x2 dlarray

   396   200
```

Alternatively, define Rosenbrock's function as a function of two inputs, `x1` and `x2`.

```
type rosenbrock2.m
```

```
function [y,dydx1,dydx2] = rosenbrock2(x1,x2)

y = 100*(x2 - x1.^2).^2 + (1 - x1).^2;
[dydx1,dydx2] = dlgradient(y,x1,x2);

end
```

Call `dlfeval` to evaluate `rosenbrock2` on two `dlarray` arguments representing the inputs `-1` and `2`.

```
x1 = dlarray(-1);
x2 = dlarray(2);
[fval,dydx1,dydx2] = dlfeval(@rosenbrock2,x1,x2)
```

```
fval =
  1x1 dlarray

   104


dydx1 =
  1x1 dlarray

   396


dydx2 =
  1x1 dlarray

   200
```

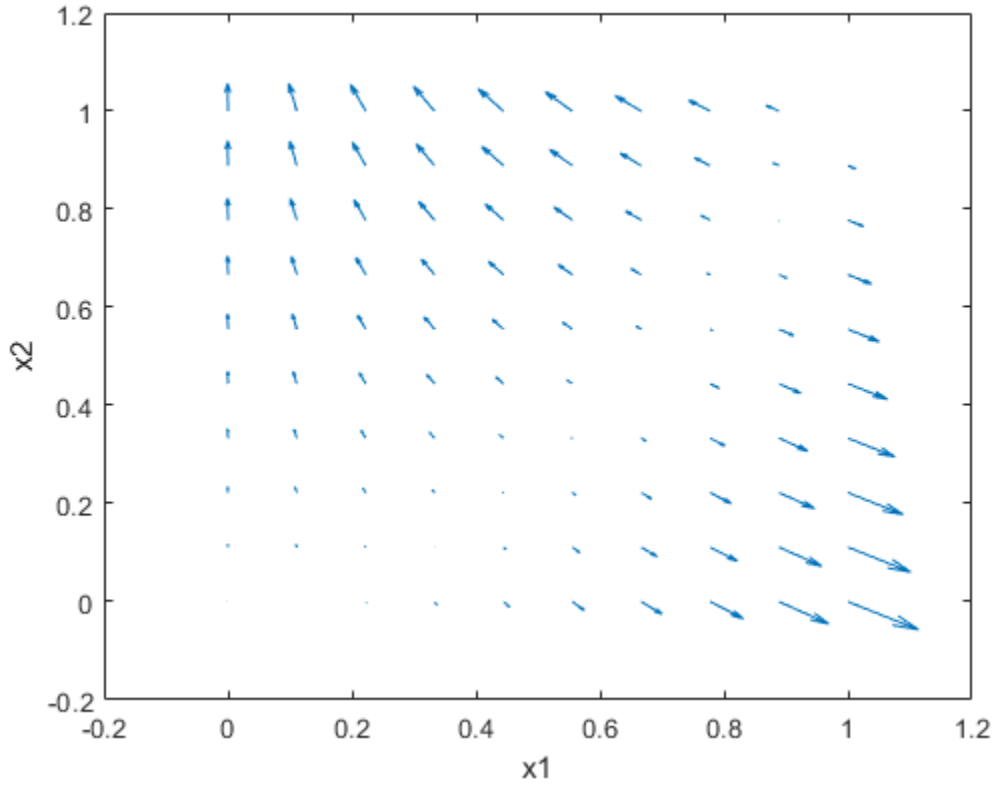
Plot the gradient of Rosenbrock's function for several points in the unit square. First, initialize the arrays representing the evaluation points and the output of the function.

```
[X1 X2] = meshgrid(linspace(0,1,10));
X1 = dlarray(X1(:));
X2 = dlarray(X2(:));
Y = dlarray(zeros(size(X1)));
DYDX1 = Y;
DYDX2 = Y;
```

Evaluate the function in a loop. Plot the result using `quiver`.

```
for i = 1:length(X1)
    [Y(i),DYDX1(i),DYDX2(i)] = dlfeval(@rosenbrock2,X1(i),X2(i));
```

```
end
quiver(extractdata(X1),extractdata(X2),extractdata(DYDX1),extractdata(DYDX2))
xlabel('x1')
ylabel('x2')
```



Compute Gradients Involving Complex Numbers

Use `dlgradient` and `dlfeval` to compute the value and gradient of a function that involves complex numbers. You can compute complex gradients, or restrict the gradients to real numbers only.

Define the function `complexFun`, listed at the end of this example. This function implements the following complex formula:

$$f(x) = (2 + 3i)x$$

Define the function `gradFun`, listed at the end of this example. This function calls `complexFun` and uses `dlgradient` to calculate the gradient of the result with respect to the input. For automatic differentiation, the value to differentiate — i.e., the value of the function calculated from the input — must be a real scalar, so the function takes the sum of the real part of the result before calculating the gradient. The function returns the real part of the function value and the gradient, which can be complex.

Define the sample points over the complex plane between -2 and 2 and -2i and 2i and convert to `dlarray`.

```
functionRes = linspace(-2,2,100);
x = functionRes + 1i*functionRes.';
x = dlarray(x);
```

Calculate the function value and gradient at each sample point.

```
[y, grad] = dlfeval(@gradFun,x);
y = extractdata(y);
```

Define the sample points at which to display the gradient.

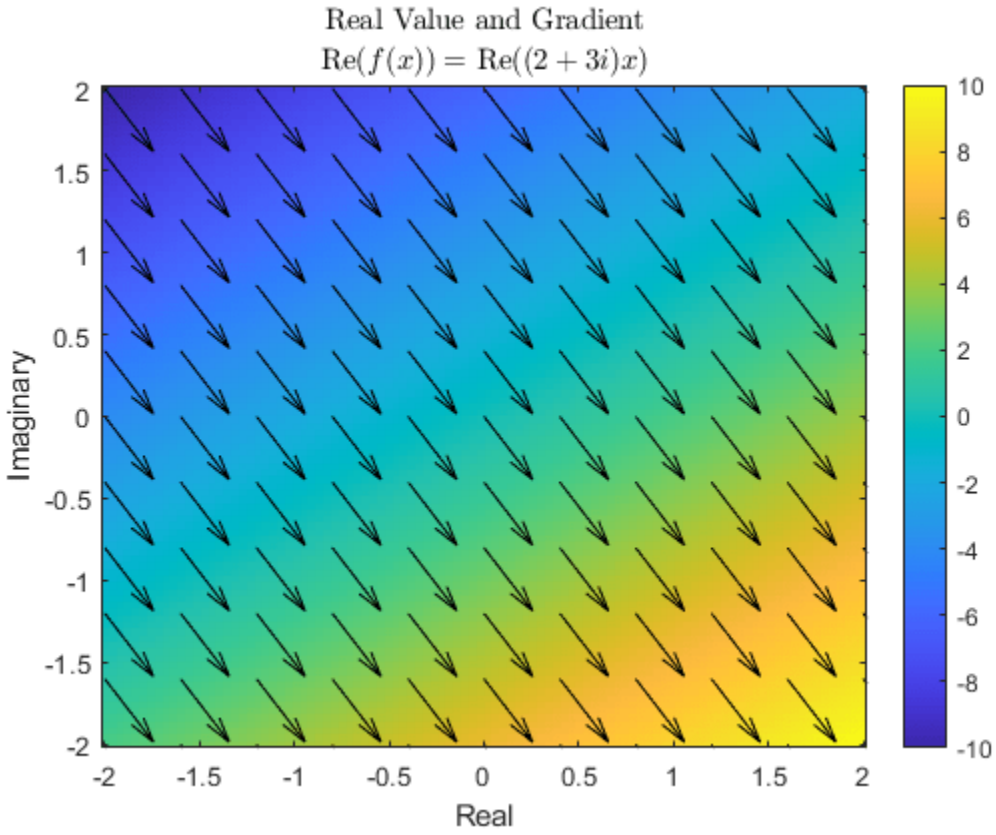
```
gradientRes = linspace(-2,2,11);
xGrad = gradientRes + 1i*gradientRes.';
```

Extract the gradient values at these sample points.

```
[~,gradPlot] = dlfeval(@gradFun,dlarray(xGrad));
gradPlot = extractdata(gradPlot);
```

Plot the results. Use `imagesc` to show the value of the function over the complex plane. Use `quiver` to show the direction and magnitude of the gradient.

```
imagesc([-2,2],[-2,2],y);
axis xy
colorbar
hold on
quiver(real(xGrad),imag(xGrad),real(gradPlot),imag(gradPlot),"k");
xlabel("Real")
ylabel("Imaginary")
title("Real Value and Gradient","Re$(f(x)) = $ Re$((2+3i)x)$","interpreter","latex")
```



The gradient of the function is the same across the entire complex plane. Extract the value of the gradient calculated by automatic differentation.

```
grad(1,1)
```

```
ans =
  1x1 dlarray

   2.0000 - 3.0000i
```

By inspection, the complex derivative of the function has the value

$$\frac{\mathrm{df(x)}}{\mathrm{dx}} = 2 + 3i$$

However, the function $\mathrm{Re}(f(x))$ is not analytic, and therefore no complex derivative is defined. For automatic differentiation in MATLAB, the value to differentiate must always be real, and therefore the function can never be complex analytic. Instead, the derivative is computed such that the returned gradient points in the direction of steepest ascent, as seen in the plot. This is done by interpreting the function $\mathrm{Re}(f(x))$: $\mathbf{C} \to \mathbf{R}$ as a function $\mathrm{Re}(f(x_R + ix_I))$: $\mathbf{R} \times \mathbf{R} \to \mathbf{R}$.

```
function y = complexFun(x)
    y = (2+3i)*x;
end

function [y,grad] = gradFun(x)
```

```
    y = complexFun(x);
    y = real(y);

    grad = dlgradient(sum(y,"all"),x);
end
```

Input Arguments

`fun` — Function to evaluate

function handle

Function to evaluate, specified as a function handle. If `fun` includes a `dlgradient` call, then `dlfeval` evaluates the gradient by using automatic differentiation. In this gradient evaluation, each argument of the `dlgradient` call must be a `dlarray` or a cell array, structure, or table containing a `dlarray`. The number of input arguments to `dlfeval` must be the same as the number of input arguments to `fun`.

Example: `@rosenbrock`

Data Types: `function_handle`

`x1,...,xn` — Function arguments

any MATLAB data type | `dlnetwork`

Function arguments, specified as any MATLAB data type or a `dlnetwork` object.

An input argument `xj` that is a variable of differentiation in a `dlgradient` call must be a traced `dlarray` or a cell array, structure, or table containing a traced `dlarray`. An extra variable such as a hyperparameter or constant data array does not have to be a `dlarray`.

To evaluate gradients for deep learning, you can provide a `dlnetwork` object as a function argument and evaluate the forward pass of the network inside `fun`.

Example: `dlarray([1 2;3 4])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

Output Arguments

`y1,...,yk` — Function outputs

any data type | `dlarray`

Function outputs, returned as any data type. If the output results from a `dlgradient` call, the output is a `dlarray`.

Tips

- A `dlgradient` call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using `dlfeval`, and the argument to the function must be a `dlarray`. See “Use Automatic Differentiation In Deep Learning Toolbox”.

- To enable the correct evaluation of gradients, the function `fun` must use only supported functions for `dlarray`. See “List of Functions with dlarray Support”.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `dlfeval` supports providing `x1,...,xn` as a `gpuArray` or as a `dlarray` that contains a `gpuArray`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlnetwork`

Topics

“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Generative Adversarial Network (GAN)”

Introduced in R2019b

dlgradient

Compute gradients for custom training loops using automatic differentiation

Syntax

```
[dydx1,...,dydxk] = dlgradient(y,x1,...,xk)
[dydx1,...,dydxk] = dlgradient(y,x1,...,xk,Name,Value)
```

Description

Use `dlgradient` to compute derivatives using automatic differentiation for custom training loops.

Tip 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". Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see "Define Deep Learning Network for Custom Training Loops".

`[dydx1,...,dydxk] = dlgradient(y,x1,...,xk)` returns the gradients of `y` with respect to the variables `x1` through `xk`.

Call `dlgradient` from inside a function passed to `dlfeval`. See "Compute Gradient Using Automatic Differentiation" on page 1-445 and "Use Automatic Differentiation In Deep Learning Toolbox".

`[dydx1,...,dydxk] = dlgradient(y,x1,...,xk,Name,Value)` returns the gradients and specifies additional options using one or more name-value pairs. For example, `dydx = dlgradient(y,x,'RetainData',true)` causes the gradient to retain intermediate values for reuse in subsequent `dlgradient` calls. This syntax can save time, but uses more memory. For more information, see "Tips" on page 1-452.

Examples

Compute Gradient Using Automatic Differentiation

Rosenbrock's function is a standard test function for optimization. The `rosenbrock.m` helper function computes the function value and uses automatic differentiation to compute its gradient.

```
type rosenbrock.m
```

```
function [y,dydx] = rosenbrock(x)

y = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
```

```
dydx = dlgradient(y,x);

end
```

To evaluate Rosenbrock's function and its gradient at the point `[-1,2]`, create a `dlarray` of the point and then call `dlfeval` on the function handle `@rosenbrock`.

```
x0 = dlarray([-1,2]);
[fval,gradval] = dlfeval(@rosenbrock,x0)
```

```
fval =
  1x1 dlarray

   104


gradval =
  1x2 dlarray

   396   200
```

Alternatively, define Rosenbrock's function as a function of two inputs, `x1` and `x2`.

```
type rosenbrock2.m
```

```
function [y,dydx1,dydx2] = rosenbrock2(x1,x2)

y = 100*(x2 - x1.^2).^2 + (1 - x1).^2;
[dydx1,dydx2] = dlgradient(y,x1,x2);

end
```

Call `dlfeval` to evaluate `rosenbrock2` on two `dlarray` arguments representing the inputs `-1` and `2`.

```
x1 = dlarray(-1);
x2 = dlarray(2);
[fval,dydx1,dydx2] = dlfeval(@rosenbrock2,x1,x2)
```

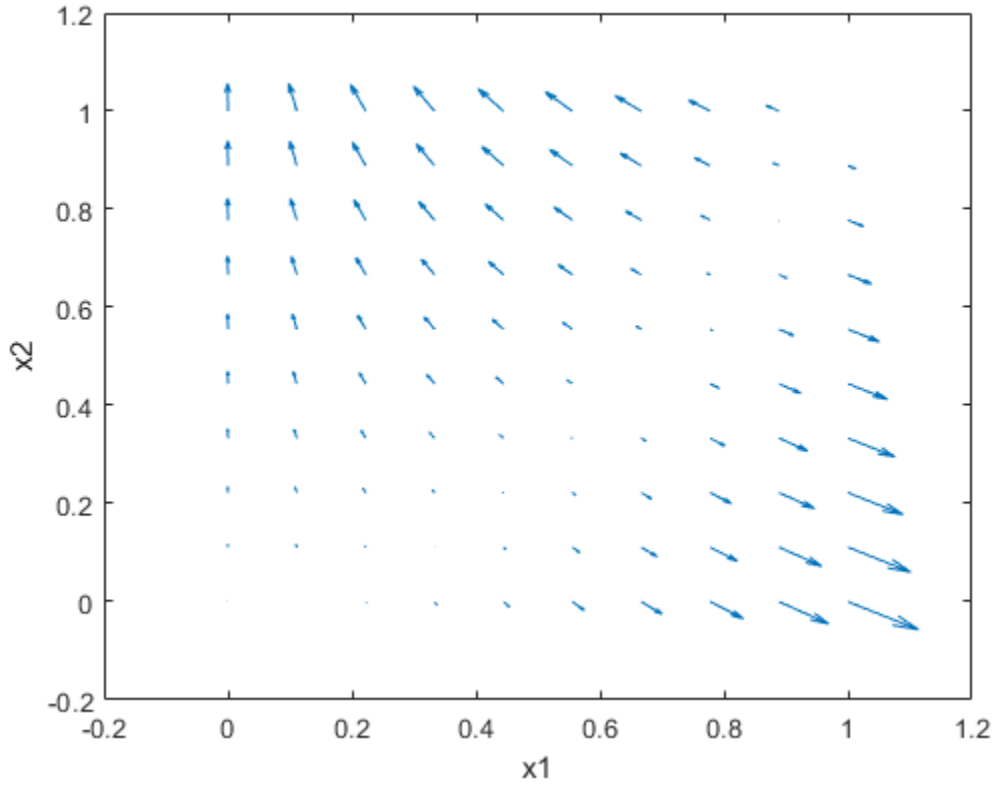
```
fval =
  1x1 dlarray

   104


dydx1 =
  1x1 dlarray

   396


dydx2 =
  1x1 dlarray

   200
```

Plot the gradient of Rosenbrock's function for several points in the unit square. First, initialize the arrays representing the evaluation points and the output of the function.

```
[X1 X2] = meshgrid(linspace(0,1,10));
X1 = dlarray(X1(:));
X2 = dlarray(X2(:));
Y = dlarray(zeros(size(X1)));
DYDX1 = Y;
DYDX2 = Y;
```

Evaluate the function in a loop. Plot the result using `quiver`.

```
for i = 1:length(X1)
    [Y(i),DYDX1(i),DYDX2(i)] = dlfeval(@rosenbrock2,X1(i),X2(i));
end
quiver(extractdata(X1),extractdata(X2),extractdata(DYDX1),extractdata(DYDX2))
xlabel('x1')
ylabel('x2')
```



Compute Gradients Involving Complex Numbers

Use `dlgradient` and `dlfeval` to compute the value and gradient of a function that involves complex numbers. You can compute complex gradients, or restrict the gradients to real numbers only.

Define the function `complexFun`, listed at the end of this example. This function implements the following complex formula:

$$f(x) = (2 + 3i)x$$

Define the function `gradFun`, listed at the end of this example. This function calls `complexFun` and uses `dlgradient` to calculate the gradient of the result with respect to the input. For automatic differentiation, the value to differentiate — i.e., the value of the function calculated from the input — must be a real scalar, so the function takes the sum of the real part of the result before calculating the gradient. The function returns the real part of the function value and the gradient, which can be complex.

Define the sample points over the complex plane between -2 and 2 and -2*i* and 2*i* and convert to `dlarray`.

```
functionRes = linspace(-2,2,100);
x = functionRes + 1i*functionRes.';
x = dlarray(x);
```

Calculate the function value and gradient at each sample point.

```
[y, grad] = dlfeval(@gradFun,x);
y = extractdata(y);
```

Define the sample points at which to display the gradient.

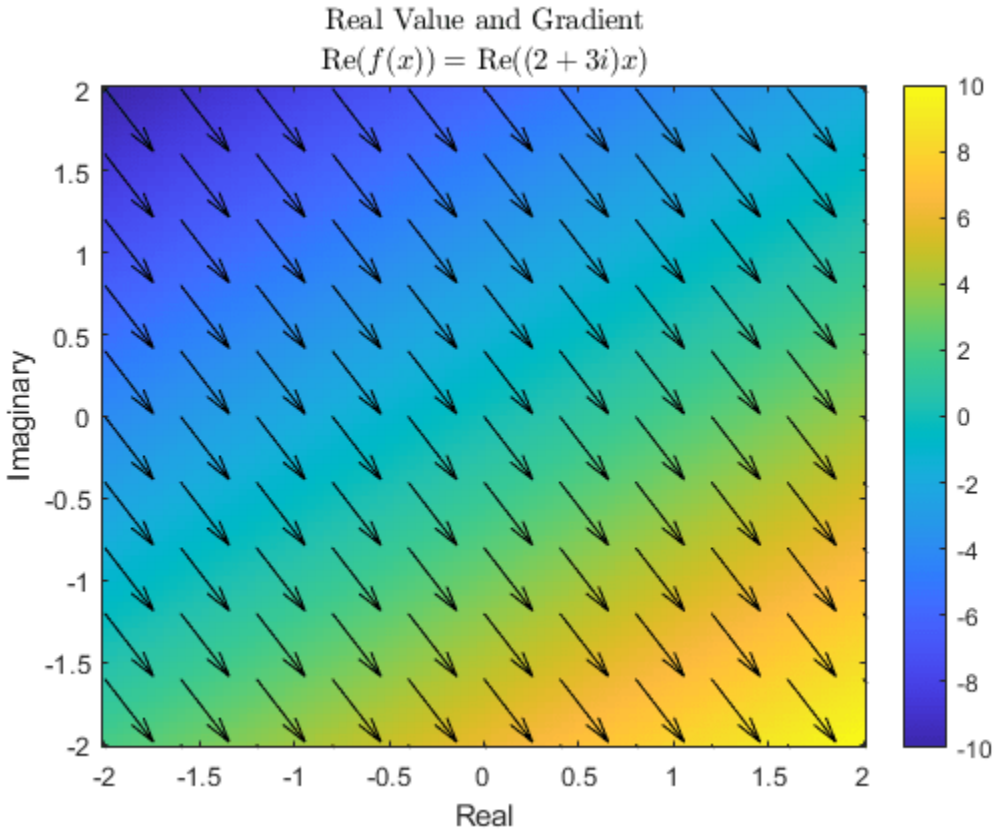
```
gradientRes = linspace(-2,2,11);
xGrad = gradientRes + 1i*gradientRes.';
```

Extract the gradient values at these sample points.

```
[~,gradPlot] = dlfeval(@gradFun,dlarray(xGrad));
gradPlot = extractdata(gradPlot);
```

Plot the results. Use `imagesc` to show the value of the function over the complex plane. Use `quiver` to show the direction and magnitude of the gradient.

```
imagesc([-2,2],[-2,2],y);
axis xy
colorbar
hold on
quiver(real(xGrad),imag(xGrad),real(gradPlot),imag(gradPlot),"k");
xlabel("Real")
ylabel("Imaginary")
title("Real Value and Gradient","Re$(f(x)) = $ Re$((2+3i)x)$","interpreter","latex")
```



The gradient of the function is the same across the entire complex plane. Extract the value of the gradient calculated by automatic differentation.

```
grad(1,1)
```

```
ans =
  1x1 dlarray

   2.0000 - 3.0000i
```

By inspection, the complex derivative of the function has the value

$$\frac{\mathrm{df(x)}}{\mathrm{dx}} = 2 + 3i$$

However, the function $\mathrm{Re}(f(x))$ is not analytic, and therefore no complex derivative is defined. For automatic differentiation in MATLAB, the value to differentiate must always be real, and therefore the function can never be complex analytic. Instead, the derivative is computed such that the returned gradient points in the direction of steepest ascent, as seen in the plot. This is done by interpreting the function $\mathrm{Re}(f(x))$: $\mathbf{C} \to \mathbf{R}$ as a function $\mathrm{Re}(f(x_R + ix_I))$: $\mathbf{R} \times \mathbf{R} \to \mathbf{R}$.

```
function y = complexFun(x)
    y = (2+3i)*x;
end

function [y,grad] = gradFun(x)
```

```
    y = complexFun(x);
    y = real(y);

    grad = dlgradient(sum(y,"all"),x);
end
```

Input Arguments

y — Variable to differentiate

scalar `dlarray` object

Variable to differentiate, specified as a scalar `dlarray` object. For differentiation, `y` must be a traced function of `dlarray` inputs (see "Traced dlarray" on page 1-452) and must consist of supported functions for `dlarray` (see "List of Functions with dlarray Support").

Variable to differentiate must be real even when the name-value option `'AllowComplex'` is set to `true`.

Example: `100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2`

Example: `relu(X)`

Data Types: `single` | `double` | `logical`

x1,...,xk — Variable in function

`dlarray` object | cell array containing `dlarray` objects | structure containing `dlarray` objects | table containing `dlarray` objects

Variable in the function, specified as a `dlarray` object, a cell array, structure, or table containing `dlarray` objects, or any combination of such arguments recursively. For example, an argument can be a cell array containing a cell array that contains a structure containing `dlarray` objects.

If you specify `x1,...,xk` as a table, the table must contain the following variables:

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

Example: `dlarray([1 2;3 4])`

Data Types: `single` | `double` | `logical` | `struct` | `cell`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `dydx = dlgradient(y,x,'RetainData',true)` causes the gradient to retain intermediate values for reuse in subsequent `dlgradient` calls

RetainData — Flag to retain trace data during function call

`false` or `0` (default) | `true` or `1`

Flag to retain trace data during the function call, specified as `false` or `true`. When this argument is `false`, a `dlarray` discards the derivative trace immediately after computing a derivative. When this argument is `true`, a `dlarray` retains the derivative trace until the end of the `dlfeval` function call that evaluates the `dlgradient`. The `true` setting is useful only when the `dlfeval` call contains more than one `dlgradient` call. The `true` setting causes the software to use more memory, but can save time when multiple `dlgradient` calls use at least part of the same trace.

When `'EnableHigherDerivatives'` is `true`, then intermediate values are retained and the `'RetainData'` option has no effect.

Example: `dydx = dlgradient(y,x,'RetainData',true)`

Data Types: `logical`

`EnableHigherDerivatives` — Flag to enable higher-order derivatives
`true` or `1` | `false` or `0`

Flag to enable higher-order derivatives, specified as one of the following:

- `true` — Enable higher-order derivatives. Trace the backward pass so that the returned gradients can be used in further computations for subsequent calls to the `dlgradient` function. If `'EnableHigherDerivatives'` is `true`, then intermediate values are retained and the `'RetainData'` option has no effect.
- `false` — Disable higher-order derivatives. Do not trace the backward pass. Use this option when you need to compute first-order derivatives only as this is usually quicker and requires less memory.

When using the `dlgradient` function inside an `AcceleratedFunction` object, the default value is `true`. Otherwise, the default value is `false`.

For examples showing how to train models that require calculating higher-order derivatives, see:

- “Train Wasserstein GAN with Gradient Penalty (WGAN-GP)”
- “Solve Partial Differential Equations Using Deep Learning”

Data Types: `logical`

`AllowComplex` — Flag to allow complex variables and gradients
`true` or `1` (default) | `false` or `0`

Flag to allow complex variables in function and complex gradients, specified as one of the following:

- `true` — Allow complex variables in function and complex gradients. Variables in the function can be specified as complex numbers. Gradients can be complex even if all variables are real. Variable to differentiate must be real.
- `false` — Do not allow complex variables and gradients. Variable to differentiate and any variables in the function must be real numbers. Gradients are always real. Intermediate values can still be complex.

Variable to differentiate must be real even when the name-value option `'AllowComplex'` is set to `true`.

Data Types: `logical`

Output Arguments

dydx1,...,dydxk — Gradient
dlarray object | cell array containing dlarray objects | structure containing dlarray objects | table containing dlarray objects

Gradient, returned as a dlarray object, or a cell array, structure, or table containing dlarray objects, or any combination of such arguments recursively. The size and data type of dydx1,...,dydxk are the same as those of the associated input variable x1,…,xk.

Limitations

- The dlgradient function does not support calculating higher-order derivatives when using dlnetwork objects containing custom layers with a custom backward function.
- The dlgradient function does not support calculating higher-order derivatives when using dlnetwork objects containing the following layers:
  - gruLayer
  - lstmLayer
  - bilstmLayer
- The dlgradient function does not support calculating higher-order derivatives that depend on the following functions:
  - gru
  - lstm
  - embed
  - prod
  - interp1

More About

Traced dlarray

During the computation of a function, a dlarray internally records the steps taken in a trace, enabling reverse mode automatic differentiation. The trace occurs within a dlfeval call. See “Automatic Differentiation Background”.

Tips

- A dlgradient call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using dlfeval, and the argument to the function must be a dlarray. See “Use Automatic Differentiation In Deep Learning Toolbox”.
- To enable the correct evaluation of gradients, the y argument must use only supported functions for dlarray. See “List of Functions with dlarray Support”.
- If you set the 'RetainData' name-value pair argument to true, the software preserves tracing for the duration of the dlfeval function call instead of erasing the trace immediately after the derivative computation. This preservation can cause a subsequent dlgradient call within the same dlfeval call to be executed faster, but uses more memory. For example, in training an

adversarial network, the `'RetainData'` setting is useful because the two networks share data and functions during training. See “Train Generative Adversarial Network (GAN)”.

- When you need to calculate first-order derivatives only, ensure that the `'EnableHigherDerivatives'` option is `false` as this is usually quicker and requires less memory.
- Complex gradients are calculated using the Wirtinger derivative. The gradient is defined in the direction of increase of the real part of the function to differentiate. This is because the variable to differentiate — for example, the loss — must be real, even if the function is complex.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the variable to differentiate input argument `y` is a `dlarray` object that contains a `gpuArray`, then this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlfeval` | `dlnetwork`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”
“Train Generative Adversarial Network (GAN)”

Introduced in R2019b

dlmtimes

(Not recommended) Batch matrix multiplication for deep learning

Note `dlmtimes` is not recommended. Use `pagemtimes` instead. For more information, see "Compatibility Considerations"

Syntax

```
dlC = dlmtimes(dlA,dlB)
```

Description

`dlC = dlmtimes(dlA,dlB)` computes matrix multiplication for each page of `dlA` and `dlB`. For 3-D inputs `dlA` and `dlB`, `dlC` is calculated as

```
dlC(:,:,i) = dlA(:,:,i) * dlB(:,:,i)
```

Similarly, for n-dimensional inputs `dlA` and `dlB`, `dlC` is calculated as

```
dlC(:,:,i1,...,in) = dlA(:,:,i1,...,in) * dlB(:,:,i1,...,in)
```

If one of `dlA` or `dlB` is a two-dimensional matrix, this matrix multiplies each page of the other input.

Examples

Multiply Two 4-D Arrays

Create two 4-D arrays.

```
A = rand(3,4,8,2);
B = rand(4,5,8,2);

dlA = dlarray(A);
dlB = dlarray(B);
```

Calculate the batch matrix multiplication of `dlA` and `dlB`.

```
dlC = dlmtimes(dlA,dlB);
size(dlC)
```

```
ans = 1×4
     3     5     8     2
```

Multiply Two Inputs Using Scalar Expansion

If one of the inputs is a 2-D matrix, the function uses scalar expansion to expand this matrix to the same size as the other input in the third and higher dimensions. The function then performs batch matrix multiplication to the expanded matrix and the input array.

Create a random array of size 15-by-20-by-3-by-128. Convert to `dlarray`.

```
A = rand(15,20,3,128);
dlA = dlarray(A);
```

Create a random matrix of size 20-by-15.

```
B = rand(20,15);
```

Multiply `dlA` and `B` using `dlmtimes`.

```
dlC = dlmtimes(dlA,B);
size(dlC)
```

```
ans = 1×4
    15    15     3   128
```

Input Arguments

`dlA,dlB` — Operands

scalars | vectors | matrices | arrays

Operands, specified as scalars, vectors, matrices, or N-D arrays. At least one of `dlA` or `dlB` must be a `dlarray`. The inputs `dlA` or `dlB` must not be formatted unless one of `dlA` or `dlB` is an unformatted scalar.

The number of columns of `dlA` must match the number of rows of `dlB`. If one of `dlA` or `dlB` is a two-dimensional matrix, this matrix multiplies each page of the other input. Otherwise, the size of `dlA` and `dlB` for each dimension greater than two must match.

Output Arguments

`dlC` — Product

scalar | vector | matrix | array

Product, returned as a scalar, vector, matrix, or an N-D array.

Array `dlC` has the same number of rows as input `dlA` and the same number of columns as input `dlB`, unless one of `dlA` or `dlB` is a scalar. The size of the other dimensions of `dlC` match the size of the dimensions greater than two of both `dlA` and `dlB`. If `dlA` or `dlB` is a matrix, the size of the other dimensions matches the size of the other (non-matrix) input. If one of `dlA` or `dlB` is a scalar, `dlC` has the same size as the non-scalar input.

Compatibility Considerations

`dlmtimes` is not recommended

Not recommended starting in R2020b

`dlmtimes` is not recommended. Use `pagemtimes` instead. The two-input syntax of `pagemtimes` performs the same functionality as `dlmtimes`. For information on how to use `pagemtimes` with `dlarray` inputs, see the `pagemtimes` entry in “List of Functions with dlarray Support”

See Also

`dlarray` | `mtimes` | `pagefun` | `pagemtimes`

Topics
“Sequence-to-Sequence Translation Using Attention”
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”

Introduced in R2020a

dlnetwork

Deep learning network for custom training loops

Description

A `dlnetwork` object enables support for custom training loops using automatic differentiation.

Tip 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”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Deep Learning Network for Custom Training Loops”.

Creation

Syntax

```
dlnet = dlnetwork(layers)
dlnet = dlnetwork(layers,dlX1,...,dlXn)
dlnet = dlnetwork(layers,'Initialize',tf)
dlnet = dlnetwork( ___ ,'OutputNames',names)
```

Description

`dlnet = dlnetwork(layers)` converts the network layers specified in `layers` to an initialized `dlnetwork` object representing a deep neural network for use with custom training loops. `layers` can be a `LayerGraph` object or a `Layer` array. `layers` must contain an input layer.

An initialized `dlnetwork` object is ready for training. The learnable parameters and state values of `dlnet` are initialized for training with initial values based on the input size defined by the network input layer.

`dlnet = dlnetwork(layers,dlX1,...,dlXn)` creates an initialized `dlnetwork` object using example inputs `dlX1,...,dlXn`. The learnable parameters and state values of `dlnet` are initialized with initial values based on the input size and format defined by the example inputs. Use this syntax to create an initialized `dlnetwork` with inputs that are not connected to an input layer.

`dlnet = dlnetwork(layers,'Initialize',tf)` specifies whether to return an initialized or uninitialized `dlnetwork`. Use this syntax to create an uninitialized network.

An uninitialized network has unset, empty values for learnable and state parameters and is not ready for training. You must initialize an uninitialized `dlnetwork` before you can use it. Create an uninitialized network when you want to defer initialization to a later point. You can use uninitialized

`dlnetwork` objects to create complex networks using intermediate building blocks that you then connect together, for example, using "Deep Learning Network Composition" workflows. You can initialize an uninitialized `dlnetwork` using the `initialize` function.

`dlnet = dlnetwork( ___ ,'OutputNames',names)` also sets the `OutputNames` property using any of the previous syntaxes. The `OutputNames` property specifies the layers that return the network outputs. To set the output names, the network must be initialized.

Input Arguments

`layers` — Network layers
`LayerGraph` object | `Layer` array

Network layers, specified as a `LayerGraph` object or as a `Layer` array.

If `layers` is a `Layer` array, then the `dlnetwork` function connects the layers in series.

The network layers must not contain output layers. When training the network, calculate the loss separately.

For a list of layers supported by `dlnetwork`, see "Supported Layers" on page 1-470.

`dlX1,...,dlXn` — Example network inputs
`dlarray`

Example network inputs, specified as formatted `dlarray` objects. The software propagates the example inputs through the network to determine the appropriate sizes and formats of the learnable and state parameters of the `dlnetwork`.

Example inputs must be formatted `dlarray` objects. When `layers` is a `Layer` array, provide example inputs in the same order that the layers that require inputs appear in the `Layer` array. When `layers` is a `LayerGraph` object, provide example inputs in the same order as the layers that require inputs appear in the `Layers` property of the `LayerGraph`.

Example inputs are not supported when `tf` is false.

`tf` — Flag to return initialized `dlnetwork`
`true` or `1` (default) | `false` or `0`

Flag to return initialized `dlnetwork`, specified as a numeric or logical `1` (`true`) or `0` (`false`).

If `tf` is `true` or `1`, learnable and state parameters of `dlnet` are initialized with initial values for training, according to the network input layer or the example inputs provided.

If `tf` is false, learnable and state parameters are not initialized. Before you use an uninitialized network, you must first initialize it using the `initialize` function. Example inputs are not supported when `tf` is false.

Properties

`Layers` — Network layers
`Layer` array

This property is read-only.

Network layers, specified as a `Layer` array.

`Connections` — Layer connections

table

This property is read-only.

Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, `Source`, specifies the source of each connection. The second column, `Destination`, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form `'layerName/IOName'`, where `'IOName'` is the name of the layer input or output.

Data Types: `table`

`Learnables` — Network learnable parameters

table

Network learnable parameters, specified as a table with three columns:

- `Layer` – Layer name, specified as a string scalar.
- `Parameter` – Parameter name, specified as a string scalar.
- `Value` – Value of parameter, specified as a `dlarray` object.

The network learnable parameters contain the features learned by the network. For example, the weights of convolution and fully connected layers.

Data Types: `table`

`State` — Network state

table

Network state, specified as a table.

The network state is a table with three columns:

- `Layer` – Layer name, specified as a string scalar.
- `Parameter` – State parameter name, specified as a string scalar.
- `Value` – Value of state parameter, specified as a `dlarray` object.

Layer states contain information calculated during the layer operation to be retained for use in subsequent forward passes of the layer. For example, the cell state and hidden state of LSTM layers, or running statistics in batch normalization layers.

For recurrent layers, such as LSTM layers, with the `HasStateInputs` property set to `1` (true), the state table does not contain entries for the states of that layer.

During training or inference, you can update the network state using the output of the `forward` and `predict` functions.

Data Types: `table`

`InputNames` — Network input layer names

cell array of character vectors

This property is read-only.

Network input layer names, specified as a cell array of character vectors.

Data Types: `cell`

`OutputNames` — Names of layers that return network outputs
cell array of character vectors | string array

Names of layers that return network outputs, specified as a cell array of character vectors or a string array.

To set the output names, the network must be initialized.

If you do not specify the output names, then the software sets the `OutputNames` property to the layers with disconnected outputs. If a layer has multiple outputs, then the disconnected outputs are specified as `'layerName/outputName'`.

The `predict` and `forward` functions, by default, return the data output by the layers given by the `OutputNames` property.

Data Types: `cell` | `string`

`Initialized` — Flag for initialized network
`0` (false) | `1` (true)

This property is read-only.

Flag for initialized network, specified as `0` (false) or `1` (true).

If `Initialized` is `0` (false), the network is not initialized. You must initialize the network before you can use it. Initialize the network using the `initialize` function.

If `Initialized` is `1` (true), the network is initialized and can be used for training and inference. If you change the values of learnable parameters — for example, during training — the value of `Initialized` remains `1` (true).

Data Types: `logical`

Object Functions

predict	Compute deep learning network output for inference
forward	Compute deep learning network output for training
initialize	Initialize learnable and state parameters of a dlnetwork
layerGraph	Graph of network layers for deep learning
setL2Factor	Set L2 regularization factor of layer learnable parameter
setLearnRateFactor	Set learn rate factor of layer learnable parameter
getLearnRateFactor	Get learn rate factor of layer learnable parameter
getL2Factor	Get L2 regularization factor of layer learnable parameter

Examples

Convert Pretrained Network to `dlnetwork` Object

To implement a custom training loop for your network, first convert it to a `dlnetwork` object. Do not include output layers in a `dlnetwork` object. Instead, you must specify the loss function in the custom training loop.

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.

```
net = googlenet;
```

Convert the network to a layer graph and remove the layers used for classification using `removeLayers`.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,["prob" "output"]);
```

Convert the network to a `dlnetwork` object.

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

```
dlnet =
  dlnetwork with properties:

         Layers: [142x1 nnet.cnn.layer.Layer]
    Connections: [168x2 table]
     Learnables: [116x3 table]
          State: [0x3 table]
     InputNames: {'data'}
    OutputNames: {'loss3-classifier'}
    Initialized: 1
```

Create Initialized `dlnetwork` with Unconnected Inputs

Use example inputs to create a multi-input `dlnetwork` that is ready for training. The software propagates the example inputs through the network to determine the appropriate sizes and formats of the learnable and state parameters of the `dlnetwork`.

Define the network architecture. Construct a network with two branches. The network takes two inputs, with one input per branch. Connect the branches using an addition layer.

```
numFilters = 24;

layersBranch1 = [
    convolution2dLayer(3,6*numFilters,'Padding','same','Stride',2,'Name','conv1Branch1')
    groupNormalizationLayer('all-channels')
    reluLayer
    convolution2dLayer(3,numFilters,'Padding','same')
    groupNormalizationLayer('channel-wise')
    additionLayer(2,'Name','add')
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer];
```

```
layersBranch2 = [
    convolution2dLayer(1,numFilters,'Name','convBranch2')
    groupNormalizationLayer('all-channels','Name','gnBranch2')];

lgraph = layerGraph(layersBranch1);
lgraph = addLayers(lgraph,layersBranch2);
lgraph = connectLayers(lgraph,'gnBranch2','add/in2');
```

Create example network inputs of the same size format as typical network inputs. For both inputs, use a batch size of 32. Use an input of size 64-by-64 with three channels for the input to the layer `convBranch1`. Use an input of size 64-by-64 with 18 channels for the input for the input to the layer `convBranch2`.

```
dlX1 = dlarray(rand([64 64 3 32]),"SSCB");
dlX2 = dlarray(rand([32 32 18 32]),"SSCB");
```

Create the dlnetwork. Provide the inputs in the same order that the unconnected layers appear in the `Layers` property of lgraph.

```
dlnet = dlnetwork(lgraph,dlX1,dlX2);
```

Check that the network is initialized and ready for training.

```
dlnet.Initialized
```

```
ans = logical
   1
```

Train Network Using Custom Training Loop

This example shows how to train a network that classifies handwritten digits with a custom learning rate schedule.

If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation.

This example trains a network to classify handwritten digits with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + k\,t}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.

Load Training Data

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

```
dataFolder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ....
    'LabelSource','foldernames');
```

Partition the data into training and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9,'randomize');
```

The network used in this example requires input images of size 28-by-28-by-1. To automatically resize the training images, use an augmented image datastore. Specify additional augmentation operations to perform on the training images: randomly translate the images up to 5 pixels in the horizontal and vertical axes. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
inputSize = [28 28 1];
pixelRange = [-5 5];
imageAugmenter = imageDataAugmenter( ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain,'DataAugmentation',imageAugmente
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

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

Determine the number of classes in the training data.

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

Define Network

Define the network for image classification.

```
layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','input')
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding','same','Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding','same','Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

```
dlnet =
  dlnetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [11×2 table]
     Learnables: [14×3 table]
          State: [6×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object, a mini-batch of input data with corresponding labels and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Specify Training Options

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

```
numEpochs = 10;
miniBatchSize = 128;
```

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

```
initialLearnRate = 0.01;
decay = 0.01;
momentum = 0.9;
```

Train Model

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

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

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

Initialize the training progress plot.

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

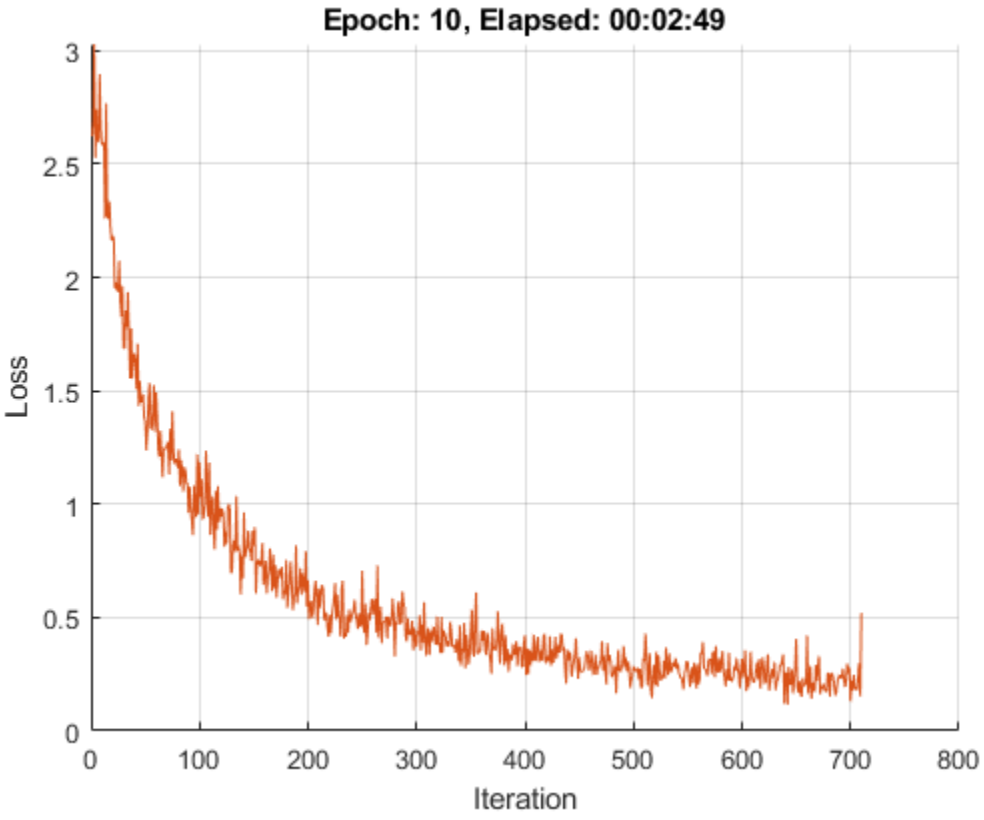
Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Train the network using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using the `dlfeval` and `modelGradients` functions and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq);

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

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

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

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

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

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

After training, making predictions on new data does not require the labels. Create `minibatchqueue` object containing only the predictors of the test data:

- To ignore the labels for testing, set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessMiniBatchPredictors` function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCB'` (spatial, spatial, channel, batch).

```
numOutputs = 1;
mbqTest = minibatchqueue(augimdsValidation,numOutputs, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatchPredictors, ...
    'MiniBatchFormat','SSCB');
```

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

```
predictions = modelPredictions(dlnet,mbqTest,classes);
```

Evaluate the classification accuracy.

```
YTest = imdsValidation.Labels;
accuracy = mean(predictions == YTest)
```

```
accuracy = 0.9530
```

Model Gradients Function

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

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

[dlYPred,state] = forward(dlnet,dlX);

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

loss = double(gather(extractdata(loss)));

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)

predictions = [];

while hasdata(mbq)

    dlXTest = next(mbq);
    dlYPred = predict(dlnet,dlXTest);

    YPred = onehotdecode(dlYPred,classes,1)';

    predictions = [predictions; YPred];
end

end
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Preprocess the images using the `preprocessMiniBatchPredictors` function.
2. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.

3 One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

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

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate.
Y = cat(2,YCell{1:end});

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

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

end
```

Freeze Learnable Parameters of `dlnetwork` Object

Load a pretrained network.

```
net = squeezenet;
```

Convert the network to a layer graph, remove the output layer, and convert it to a `dlnetwork` object.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,'ClassificationLayer_predictions');
dlnet = dlnetwork(lgraph);
```

The `Learnables` property of the `dlnetwork` object is a table that contains the learnable parameters of the network. The table includes parameters of nested layers in separate rows. View the first few rows of the learnables table.

```
learnables = dlnet.Learnables;
head(learnables)
```

```
ans=8×3 table
          Layer            Parameter           Value
    ___________________    _________    ____________________

    "conv1"                "Weights"    {3x3x3x64  dlarray}
    "conv1"                "Bias"       {1x1x64    dlarray}
    "fire2-squeeze1x1"     "Weights"    {1x1x64x16 dlarray}
```

```
    "fire2-squeeze1x1"    "Bias"       {1x1x16    dlarray}
    "fire2-expand1x1"     "Weights"    {1x1x16x64 dlarray}
    "fire2-expand1x1"     "Bias"       {1x1x64    dlarray}
    "fire2-expand3x3"     "Weights"    {3x3x16x64 dlarray}
    "fire2-expand3x3"     "Bias"       {1x1x64    dlarray}
```

To freeze the learnable parameters of the network, loop over the learnable parameters and set the learn rate to 0 using the `setLearnRateFactor` function.

```
factor = 0;

numLearnables = size(learnables,1);
for i = 1:numLearnables
    layerName = learnables.Layer(i);
    parameterName = learnables.Parameter(i);

    dlnet = setLearnRateFactor(dlnet,layerName,parameterName,factor);
end
```

To use the updated learn rate factors when training, you must pass the dlnetwork object to the update function in the custom training loop. For example, use the command

```
[dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity);
```

Create Uninitialized `dlnetwork`

Create an uninitialized `dlnetwork` object without an input layer. Creating an uninitialized `dlnetwork` is useful when you do not yet know the size and format of the network inputs, for example, when the `dlnetwork` is nested inside a custom layer.

Define the network layers. This network has a single input, which is not connected to an input layer.

```
layers = [
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer];
```

Create an uninitialized `dlnetwork`. Set the `Initialize` option to `false`.

```
dlnet = dlnetwork(layers,'Initialize',false);
```

Check that the network is not initialized.

```
dlnet.Initialized
```

```
ans = logical
   0
```

The learnable and state parameters of this network are not initialized for training. To initialize the network, use the `initialize` function.

If you want to use `dlnet` directly in a custom training loop, then you can initialize it by using the `initialize` function and providing an example input.

If you want to use `dlnet` inside a custom layer, then you can take advantage of automatic initialization. If you use the custom layer inside a `dlnetwork`, then `dlnet` is initialized when the parent `dlnetwork` is constructed (or when the parent network is initialized if it is constructed as an uninitialized `dlnetwork`). If you use the custom layer inside a network that is trained using the `trainNetwork` function, then `dlnet` is automatically initialized at training time. For more information, see “Deep Learning Network Composition”.

More About

Supported Layers

The `dlnetwork` function supports the layers listed below and custom layers without forward functions returning a nonempty memory value.

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.
 `featureInputLayer`	A feature input layer inputs feature data to a network and applies data normalization. Use this layer when you have a data set of numeric scalars representing features (data without spatial or time dimensions).

Convolution and Fully Connected Layers

Layer	Description
 `convolution1dLayer`	A 1-D convolutional layer applies sliding convolutional filters to 1-D input.
 `convolution2dLayer`	A 2-D convolutional layer applies sliding convolutional filters to 2-D input.
 `convolution3dLayer`	A 3-D convolutional layer applies sliding cuboidal convolution filters to 3-D input.
 `groupedConvolution2dLayer`	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.

Layer	Description
 `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.
 `flattenLayer`	A flatten layer collapses the spatial dimensions of the input into the channel dimension.

For `lstmLayer`, `bilstmLayer`, and `gruLayer` objects, `dlnetwork` objects support layers with the default values for the `StateActivationFunction` and `GateActivationFunction` properties.

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.

Layer	Description
 `swishLayer`	A swish activation layer applies the swish function on the layer inputs.
 `tanhLayer`	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.
 `softmaxLayer`	A softmax layer applies a softmax function to the input.
 `sigmoidLayer`	A sigmoid layer applies a sigmoid function to the input such that the output is bounded in the interval (0,1).
 `functionLayer`	A function layer applies a specified function to the layer input.

Normalization, Dropout, and Cropping Layers

Layer	Description
 `batchNormalizationLayer`	A batch normalization layer normalizes a mini-batch of data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.
 `groupNormalizationLayer`	A group normalization layer normalizes a mini-batch of data across grouped subsets of channels for each observation independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use group normalization layers between convolutional layers and nonlinearities, such as ReLU layers.
 `layerNormalizationLayer`	A layer normalization layer normalizes a mini-batch of data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization layers after the learnable layers, such as LSTM and fully connected layers.
 `crossChannelNormalizationLayer`	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.
 `dropoutLayer`	A dropout layer randomly sets input elements to zero with a given probability.
 `crop2dLayer`	A 2-D crop layer applies 2-D cropping to the input.

Layer	Description
 `stftLayer`	An STFT layer computes the short-time Fourier transform of the input.

Pooling and Unpooling Layers

Layer	Description
 `averagePooling1dLayer`	A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region.
 `averagePooling2dLayer`	A 2-D average pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the average values of each region.
 `averagePooling3dLayer`	A 3-D average pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the average values of each region.
 `globalAveragePooling1dLayer`	A 1-D global average pooling layer performs downsampling by outputting the average of the time or spatial dimensions of the input.
 `globalAveragePooling2dLayer`	A 2-D global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input.
 `globalAveragePooling3dLayer`	A 3-D global average pooling layer performs downsampling by computing the mean of the height, width, and depth dimensions of the input.
 `maxPooling1dLayer`	A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.
 `maxPooling2dLayer`	A 2-D max pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the maximum of each region.
 `maxPooling3dLayer`	A 3-D max pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the maximum of each region.
 `globalMaxPooling1dLayer`	A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.
 `globalMaxPooling2dLayer`	A 2-D global max pooling layer performs downsampling by computing the maximum of the height and width dimensions of the input.

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

Combination Layers

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

Compatibility Considerations

`dlnetwork` state values are `dlarray` objects

The `State` of a `dlnetwork` object is a table containing the state parameter names and values for each layer in the network.

Starting in R2021a, the state values are `dlarray` objects. This change enables better support when using `AcceleratedFunction` objects. To accelerate deep learning functions that have frequently changing input values, for example, an input containing the network state, the frequently changing values must be specified as `dlarray` objects.

In previous versions, the state values are numeric arrays.

In most cases, you will not need to update your code. If you have code that requires the state values to be numeric arrays, then to reproduce the previous behavior, extract the data from the state values manually using the `extractdata` function with the `dlupdate` function.

```
state = dlupdate(@extractdata,dlnet.State);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports only the `InputNames` and `OutputNames` properties.
- Code generation does not support `dlnetwork` objects without input layers. The `Initialized` property of the `dlnetwork` object must be set to true.
- You can generate code for `dlnetwork` that have vector sequence inputs. For ARM Compute, the `dlnetwork` can have sequence and non-sequence input layers. For Intel MKL-DNN, input layers must be all sequence input layers. Code generation support includes:
  - `dlarray` containing vector sequences that have `'CT'` or `'CBT'` data formats.
  - A `dlnetwork` object that has multiple inputs. For RNN networks, multiple input is not supported.
- Code generation supports only the `predict` object function. The `dlarray` input to the `predict` method must be a `single` datatype.
- Code generation does not support `dlnetwork` for plain C/C++ target.
- Code generation supports MIMO `dlnetworks`.
- To create a `dlnetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation supports only the `InputNames` and `OutputNames` properties.
- Code generation does not support `dlnetwork` objects without input layers. The `Initialized` property of the `dlnetwork` object must be set to true.
- You can generate code for `dlnetwork` that have vector sequence inputs. Code generation support includes:
  - `dlarray` containing vector sequences that have `'CT'` or `'CBT'` data formats.
  - A `dlnetwork` object that has multiple inputs. For RNN networks, multiple input is not supported.
- Code generation supports only the `predict` object function. The `dlarray` input to the `predict` method must be a `single` datatype.
- Code generation supports `dlnetwork` for cuDNN and TensorRT targets. Code generation does not support `dlnetwork` for ARM Mali and plain C/C++ targets.
- When targeting TensorRT with `INT8` precision, the last layer(s) of the network must be a `softmaxLayer` layer.
- Code generation supports MIMO `dlnetworks`.
- To create a `dlnetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also
`dlarray` | `dlgradient` | `dlfeval` | `forward` | `predict` | `layerGraph` | `initialize`

Topics
“Train Generative Adversarial Network (GAN)”
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2019b

predict

Compute deep learning network output for inference

Syntax

```
dlY = predict(dlnet,dlX)
dlY = predict(dlnet,dlX1,...,dlXM)
[dlY1,...,dlYN] = predict( ___ )
[dlY1,...,dlYK] = predict( ___ ,'Outputs',layerNames)
[ ___ ] = predict( ___ ,'Acceleration',acceleration)
[ ___ ,state] = predict( ___ )
```

Description

Some deep learning layers behave differently during training and inference (prediction). For example, during training, dropout layers randomly set input elements to zero to help prevent overfitting, but during inference, dropout layers do not change the input.

To compute network outputs for inference, use the `predict` function. To compute network outputs for training, use the `forward` function. For prediction with `SeriesNetwork` and `DAGNetwork` objects, see `predict`.

Tip For prediction with `SeriesNetwork` and `DAGNetwork` objects, see `predict`.

`dlY = predict(dlnet,dlX)` returns the network output `dlY` during inference given the input data `dlX` and the network `dlnet` with a single input and a single output.

`dlY = predict(dlnet,dlX1,...,dlXM)` returns the network output `dlY` during inference given the `M` inputs `dlX1, ...,dlXM` and the network `dlnet` that has `M` inputs and a single output.

`[dlY1,...,dlYN] = predict( ___ )` returns the `N` outputs `dlY1`, ..., `dlYN` during inference for networks that have `N` outputs using any of the previous syntaxes.

`[dlY1,...,dlYK] = predict( ___ ,'Outputs',layerNames)` returns the outputs `dlY1`, ..., `dlYK` during inference for the specified layers using any of the previous syntaxes.

`[ ___ ] = predict( ___ ,'Acceleration',acceleration)` also specifies performance optimization to use during inference, in addition to the input arguments in previous syntaxes.

`[ ___ ,state] = predict( ___ )` also returns the updated network state.

Examples

Make Predictions Using `dlnetwork` Object

This example shows how to make predictions using a `dlnetwork` object by splitting data into mini-batches.

For large data sets, or when predicting on hardware with limited memory, make predictions by splitting the data into mini-batches. When making predictions with `SeriesNetwork` or `DAGNetwork` objects, the `predict` function automatically splits the input data into mini-batches. For `dlnetwork` objects, you must split the data into mini-batches manually.

Load `dlnetwork` Object

Load a trained `dlnetwork` object and the corresponding classes.

```
s = load("digitsCustom.mat");
dlnet = s.dlnet;
classes = s.classes;
```

Load Data for Prediction

Load the digits data for prediction.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true);
```

Make Predictions

Loop over the mini-batches of the test data and make predictions using a custom prediction loop.

Use `minibatchqueue` to process and manage the mini-batches of images. Specify a mini-batch size of 128. Set the read size property of the image datastore to the mini-batch size.

For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to concatenate the data into a batch and normalize the images.
- Format the images with the dimensions `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Make predictions on a GPU if one is available. By default, the `minibatchqueue` object converts the output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(imds,...
    "MiniBatchSize",miniBatchSize,...
    "MiniBatchFcn", @preprocessMiniBatch,...
    "MiniBatchFormat","SSCB");
```

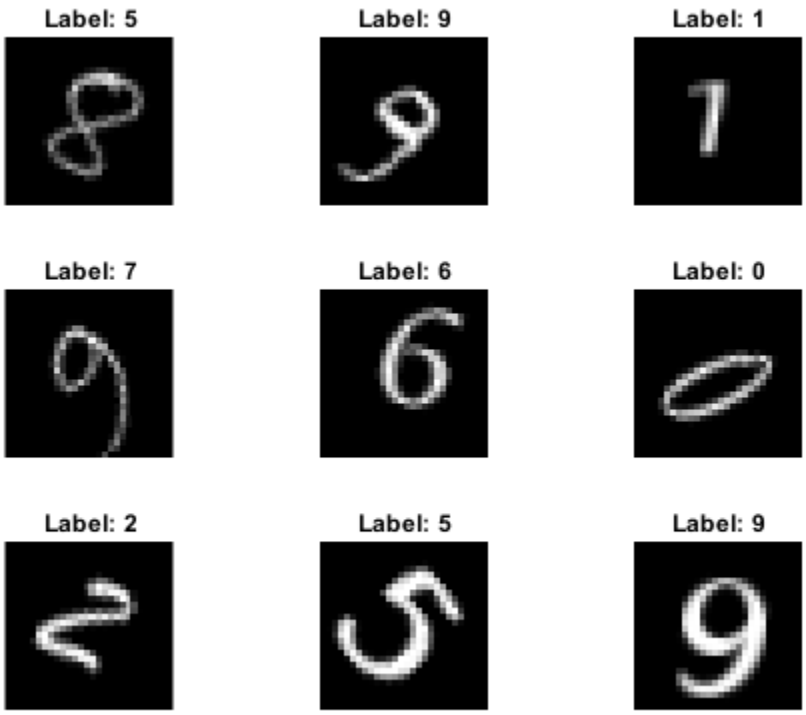
Loop over the minibatches of data and make predictions using the `predict` function. Use the `onehotdecode` function to determing the class labels. Store the predicted class labels.

```
numObservations = numel(imds.Files);
YPred = strings(1,numObservations);

predictions = [];
```

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

    % Read mini-batch of data.
    dlX = next(mbq);

    % Make predictions using the predict function.
    dlYPred = predict(dlnet,dlX);

    % Determine corresponding classes.
    predBatch = onehotdecode(dlYPred,classes,1);
    predictions = [predictions predBatch];

end
```

Visualize some of the predictions.

```
idx = randperm(numObservations,9);

figure
for i = 1:9
    subplot(3,3,i)
    I = imread(imds.Files{idx(i)});
    label = predictions(idx(i));
    imshow(I)
    title("Label: " + string(label))

end
```



Mini-Batch Preprocessing Function

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

1. Extract the data from the incoming cell array and concatenate into a numeric array. Concatenating over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Normalize the pixel values between `0` and `1`.

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

    % Normalize the images.
    X = X/255;
end
```

Input Arguments

`dlnet` — Network for custom training loops

`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`dlX` — Input data

formatted `dlarray`

Input data, specified as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

`layerNames` — Layers to extract outputs from

string array | cell array of character vectors

Layers to extract outputs from, specified as a string array or a cell array of character vectors containing the layer names.

- If `layerNames(i)` corresponds to a layer with a single output, then `layerNames(i)` is the name of the layer.
- If `layerNames(i)` corresponds to a layer with multiple outputs, then `layerNames(i)` is the layer name followed by the character `"/"` and the name of the layer output: `'layerName/outputName'`.

`acceleration` — Performance optimization

`'auto'` (default) | `'mex'` | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'mex'` — Compile and execute a MEX function. This option is available when using a GPU only. The input data or the network learnable parameters must be stored as `gpuArray` objects. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on

supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

- `'none'` — Disable all acceleration.

The default option is `'auto'`. If `'auto'` is specified, MATLAB will apply a number of compatible optimizations. If you use the `'auto'` option, MATLAB does not ever generate a MEX function.

Using the `'Acceleration'` options `'auto'` and `'mex'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The `'mex'` option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The `'mex'` option is only available when you are using a GPU. You must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The `'mex'` option has the following limitations:

- The `state` output argument is not supported.
- Only `single` precision is supported. The input data or the network learnable parameters must have underlying type `single`.
- Networks with inputs that are not connected to an input layer are not supported.
- Traced `dlarray` objects are not supported. This means that the `'mex'` option is not supported inside a call to `dlfeval`.
- Not all layers are supported. For a list of supported layers, see “Supported Layers” (GPU Coder).
- You cannot use MATLAB Compiler to deploy your network when using the `'mex'` option.

Example: `'Acceleration','mex'`

Output Arguments

`dlY` — Output data

formatted `dlarray`

Output data, returned as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

`state` — Updated network state

table

Updated network state, returned as a table.

The network state is a table with three columns:

- `Layer` – Layer name, specified as a string scalar.
- `Parameter` – State parameter name, specified as a string scalar.
- `Value` – Value of state parameter, specified as a `dlarray` object.

Layer states contain information calculated during the layer operation to be retained for use in subsequent forward passes of the layer. For example, the cell state and hidden state of LSTM layers, or running statistics in batch normalization layers.

For recurrent layers, such as LSTM layers, with the `HasStateInputs` property set to `1` (true), the state table does not contain entries for the states of that layer.

Update the state of a `dlnetwork` using the `State` property.

Compatibility Considerations

`predict` returns state values as `dlarray` objects

Behavior changed in R2021a

For `dlnetwork` objects, the `state` output argument returned by the `predict` function is a table containing the state parameter names and values for each layer in the network.

Starting in R2021a, the state values are `dlarray` objects. This change enables better support when using `AcceleratedFunction` objects. To accelerate deep learning functions that have frequently changing input values, for example, an input containing the network state, the frequently changing values must be specified as `dlarray` objects.

In previous versions, the state values are numeric arrays.

In most cases, you will not need to update your code. If you have code that requires the state values to be numeric arrays, then to reproduce the previous behavior, extract the data from the state values manually using the `extractdata` function with the `dlupdate` function.

```
state = dlupdate(@extractdata,dlnet.State);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - `dlY = predict(dlnet,dlX)`
  - `dlY = predict(dlnet,dlX1,...,dlXM)`
  - `[dlY1,...,dlYN] = predict(__)`
  - `[dlY1,...,dlYK] = predict(__,'Outputs',layerNames)`
- The input data `dlX` must not have variable size. The size must be fixed at code generation time.
- The `dlarray` input to the `predict` method must be a `single` datatype.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:

  - `dlY = predict(dlnet,dlX)`
  - `dlY = predict(dlnet,dlX1,...,dlXM)`
  - `[dlY1,...,dlYN] = predict(__)`
  - `[dlY1,...,dlYK] = predict(__,'Outputs',layerNames)`
- The input data `dlX` must not have variable size. The size must be fixed at code generation time.
- Code generation for TensorRT library does not support marking an input layer as an output by using the `[dlY1,...,dlYK] = predict(__,'Outputs',layerNames)` syntax.
- The `dlarray` input to the `predict` method must be a `single` datatype.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function runs on the GPU if either or both of the following conditions are met:
  - Any of the values of the network learnable parameters inside `dlnet.Learnables.Value` are `dlarray` objects with underlying data of type `gpuArray`
  - The input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`dlarray` | `dlgradient` | `dlfeval` | `forward` | `dlnetwork`

Topics
“Train Generative Adversarial Network (GAN)”
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2019b

dlode45

Deep learning solution of nonstiff ordinary differential equation (ODE)

Syntax

```
dlY = dlode45(odefun,tspan,dlY0,theta)
dlY = dlode45(odefun,tspan,dlY0,theta,DataFormat=FMT)
dlY = dlode45(odefun,tspan,dlY0,theta,Name=Value)
```

Description

The neural ordinary differential equation (ODE) operation returns the solution of a specified ODE.

The `dlode45` function applies the neural ODE operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note The `dlode45` function best suits neural ODE and custom training loop workflows. To solve ODEs for other workflows, use `ode45`.

`dlY = dlode45(odefun,tspan,dlY0,theta)` integrates the system of ODEs given by `odefun` on the time interval defined by the first and last elements of `tspan`, with the initial conditions `dlY0` and parameters `theta`.

`dlY = dlode45(odefun,tspan,dlY0,theta,DataFormat=FMT)` specifies the data format for the unformatted initial conditions `dlY0`. The format must contain `"S"` (spatial), `"C"` (channel), and `"B"` (batch) dimension labels only.

`dlY = dlode45(odefun,tspan,dlY0,theta,Name=Value)` specifies additional options using one or more name-value arguments. For example, `dlY = dlode45(odefun,tspan,dlY0,theta,GradientsMode="adjoint")` integrates the system of ODEs given by `odefun` and computes gradients by solving the associated adjoint ODE system.

Examples

Apply Neural ODE Operation

For the initial conditions, create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 64 channels. Specify the format `"SSCB"` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 64;
Y0 = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlY0 = dlarray(Y0,"SSCB");
```

View the size and format of the initial conditions.

```
size(dlY0)
```

```
ans = 1×4

    28    28    64   128
```

```
dims(dlY0)
```

```
ans =
'SSCB'
```

Specify the ODE function. Define the function `odeModel`, listed in the ODE Function on page 1-0 section of the example, which applies a convolution operation followed by a hyperbolic tangent operation to the input data.

```
odefun = @odeModel;
```

Initialize the parameters for the convolution operation in the ODE function. The output size of the ODE function must match the size of the initial conditions, so specify the same number of filters as the number of input channels.

```
filterSize = [3 3];
numFilters = numChannels;

parameters = struct;
parameters.Weights = dlarray(rand(filterSize(1),filterSize(2),numChannels,numFilters));
parameters.Bias = dlarray(zeros(1,numFilters));
```

Specify an interval of integration of [0 0.1].

```
tspan = [0 0.1];
```

Apply the neural ODE operation.

```
dlY = dlode45(odefun,tspan,dlY0,parameters);
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

    28    28    64   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

ODE Function

The ODE function `odeModel` takes as input the function inputs `t` (unused) and `y`, and the ODE function parameters `p` containing the convolution weights and biases, and returns the output of the convolution-tanh block operation. The convolution operation applies padding such that the output size matches the input size.

```
function z = odeModel(t,y,p)

weights = p.Weights;
bias = p.Bias;
z = dlconv(y,weights,bias,Padding="same");
z = tanh(z);

end
```

Input Arguments

odefun — Function to solve

function handle

Function to solve, specified as a function handle that defines the function to integrate.

Specify `odefun` as a function handle with syntax `z = fcn(t,y,p)`, where `t` is a scalar, `y` is a `dlarray`, and `p` is a set of parameters. The function returns a `dlarray` with the same size and format as `y`. The function must accept all three input arguments `t`, `y`, and `p`, even if not all the arguments are used in the function. The size of the ODE function output `z` must match the size of the initial conditions.

For example, specify the ODE function that applies a convolution operation followed by a tanh operation.

```
function z = dlconvtanh(t,y,p)

weights = p.Weights;
bias = p.Bias;
z = dlconv(y,weights,bias,Padding="same");
z = tanh(z);

end
```

Note here that the `t` argument is unused.

Data Types: `function_handle`

tspan — Interval of integration

numeric vector | unformatted `dlarray` vector

Interval of integration, specified as a numeric vector or an unformatted `dlarray` vector with two or more elements. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `dlY0` at the initial time `tspan(1)`, then integrates the ODE function from `tspan(1)` to `tspan(end)`.

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at point `tf`.
- If `tspan` has more than two elements, `[t0 t1 t2 ... tf]`, then the solver returns the solution evaluated at the given points `[t1 t2 ... tf]`. The solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the points specified in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

  Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

Note The behavior of the `dlode45` function differs from the `ode45` function.

If `InitialStep` or `MaxStep` is `[]`, then the software uses the values of `tspan` to initialize the values.

- If `InitialStep` is `[]`, then the software uses the elements of `tspan` as an indication of the scale of the task. When you specify `tspan` with different numbers of elements, the solution of the solver can change.
- If `MaxStep` is `[]`, then the software calculates the maximum step size using the first and last elements of `tspan`. When you change the initial or final values of `tspan`, the solution of the solver can change because the solver uses a different step sequence.

`dlY0` — Initial conditions

`dlarray` object

Initial conditions, specified as a formatted or unformatted `dlarray` object.

If `dlY0` is an unformatted `dlarray`, then you must specify the format using the `DataFormat` option.

For neural ODE operations, the data format must contain `"S"`, `"C"`, and `"B"` dimension labels only. The initial conditions must not have a `"T"` or `"U"` dimension.

`theta` — Parameters of ODE function

`dlarray` object | cell array of `dlarray` objects | structure of `dlarray` objects | table

Parameters of ODE function, specified as one of the following:

- `dlarray` object
- Cell array of `dlarray` objects
- Structure of `dlarray` objects or nested structures of `dlarray` objects
- Table with the variables `Layer`, `Parameter`, and `Value`, where `Layer` and `Parameter` contain the layer and parameter names, and `Value` contains the parameter value. Specify the variables as `dlarray` objects.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `dlY = dlode45(odefun,tspan,dlY0,theta,GradientsMode="adjoint")` integrates the system of ODEs given by `odefun` and computes gradients by solving the associated adjoint ODE system.

`DataFormat` — Dimension order of unformatted data

character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

For neural ODE operations, the data format must contain `"S"`, `"C"`, and `"B"` dimension labels only. The initial conditions must not have a `"T"` or `"U"` dimension.

You must specify `DataFormat` when the `dlY0` is not a formatted `dlarray`.

Example: `DataFormat="SSCB"`

Data Types: `char` | `string`

`GradientMode` — Method to compute gradients
`"direct"` (default) | `"adjoint"`

Method to compute gradients with respect to the initial conditions and parameters when using the `dlgradient` function, specified as one of the following:

- `"direct"` – Compute gradients by backpropagating through the operations undertaken by the numerical solver. This option best suits large mini-batch sizes or when `tspan` contains many values.
- `"adjoint"` – Compute gradients by solving the associated adjoint ODE system. This option best suits small mini-batch sizes or when `tspan` contains a small number of values.

Warning The `dlaccelerate` function does not support accelerating the `dlode45` function when the `GradientMode` option is `"direct"`. The resulting accelerated function might return unexpected results. To accelerate the code that calls the `dlode45` function, set the `GradientMode` option to `"adjoint"` or accelerate parts of your code that do not call the `dlode45` with the `GradientMode` option set to `"direct"`.

`InitialStepSize` — Initial step size
`[]` (default) | positive scalar

Initial step size, specified as a positive scalar or `[]`.

If `InitialStepSize` is `[]`, then the function automatically determines the initial step size based on the interval of integration and the output of the ODE function corresponding to the initial conditions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MaxStepSize` — Maximum step size
`[]` (default) | positive scalar

Maximum step size, specified as a positive scalar or `[]`.

If `MaxStepSize` is `[]`, then the function uses a tenth of the interval of integration size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RelativeTolerance` — Relative error tolerance
`1e-3` (default) | positive scalar

Relative error tolerance, specified as a positive scalar. The relative tolerance applies to all components of the solution.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`AbsoluteTolerance` — Absolute error tolerance
`1e-6` (default) | positive scalar

Absolute error tolerance, specified as a positive scalar. The relative tolerance applies to all components of the solution.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`dlY` — Solution of neural ODE
`dlarray` object

Solution of the neural ODE at the times given by `tspan(2:end)`, returned as a `dlarray` object with the same underlying data type as `dlY0`.

If `dlY0` is a formatted `dlarray` and `tspan` contains exactly two elements, then `dlY` has the same format as `dlY0`. If `dlY0` is not a formatted `dlarray` and `tspan` contains exactly two elements, then `dlY` is an unformatted `dlarray` with the same dimension order as `dlY0`.

If `dlY0` is a formatted `dlarray` and `tspan` contains more than two elements, then `dlY` has the same format as `dlY0` with an additional appended `"T"` (time) dimension. If `dlY0` is not a formatted `dlarray` and `tspan` contains more than two elements, then `dlY` is an unformatted `dlarray` with the same dimension order as `dlY0` with an additional appended dimension corresponding to time.

Algorithms

The neural ordinary differential equation (ODE) operation returns the solution of a specified ODE. In particular, given an input, a neural ODE operation outputs the numerical solution of the ODE $y' = f(t, y, \theta)$ for the time horizon (t_0, t_1) and with the initial condition $y(t_0) = y_0$, where t and y denote the ODE function inputs and θ is a set of learnable parameters. Typically, the initial condition y_0 is either the network input or the output of another deep learning operation.

The `dlode45` function uses the `ode45` function, which is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a single-step solver–in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$ [2] [3].

References

[1] Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. “Neural Ordinary Differential Equations.” Preprint, submitted June 19, 2018. https://arxiv.org/abs/1806.07366.

[2] Dormand, J. R., and P. J. Prince. “A Family of Embedded Runge-Kutta Formulae.” *Journal of Computational and Applied Mathematics* 6, no. 1 (March 1980): 19–26. https://doi.org/10.1016/0771-050X(80)90013-3.

[3] Shampine, Lawrence F., and Mark W. Reichelt. “The MATLAB ODE Suite.” SIAM Journal on Scientific Computing 18, no. 1 (January 1997): 1–22. https://doi.org/10.1137/S1064827594276424.

See Also

`dlarray` | `dlgradient` | `dlfeval`

Topics

“Train Neural ODE Network”
“Dynamical System Modeling Using Neural ODE”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2021b

dlquantizationOptions

Options for quantizing a trained deep neural network

Description

The `dlquantizationOptions` object provides options for quantizing a trained deep neural network to scaled 8-bit integer data types. Use the `dlquantizationOptions` object to define the metric function to use that compares the accuracy of the network before and after quantization.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

Creation

Syntax

```
quantOpts = dlquantizationOptions
quantOpts = dlquantizationOptions(Name,Value)
```

Description

`quantOpts = dlquantizationOptions` creates a `dlquantizationOptions` object with default property values.

`quantOpts = dlquantizationOptions(Name,Value)` creates a `dlquantizationOptions` object with additional properties specified by one or more name-value pair arguments.

Properties

`MetricFcn` — Function to use for calculating validation metrics

cell array of function handles

Cell array of function handles specifying the functions for calculating validation metrics of quantized network.

Example: `options = dlquantizationOptions('MetricFcn', {@(x)hComputeModelAccuracy(x, net, groundTruth)});`

Data Types: `cell`

FPGA Execution Environment Options

`Bitstream` — Bitstream name

'zcu102_int8' | 'zc706_int8' | 'arria10soc_int8'

This property affects FPGA targeting only.

Name of the FPGA bitstream specified as a character vector.

Example: `'Bitstream', 'zcu102_int8'`

`Target` — Name of the `dlhdl.Target` object
hT

This property affects FPGA targeting only.

Name of the `dlhdl.Target` object that has the board name and board interface information.

Example: `'Target', hT`

Examples

Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load net
net
```

```
net =
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

In this example, use the images in the `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
```

```
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

Create a `dlquantizer` object and specify the network to quantize.

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare with predicted label with actual ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=95×5 table
                  Optimized Layer Name                      Network Layer Name       Learnables
    ____________________________________________________    ____________________    ____________

    {'conv1_relu_conv1_Weights'                       }    {'relu_conv1'          }          "We
    {'conv1_relu_conv1_Bias'                          }    {'relu_conv1'          }          "Bi
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}          "We
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}          "Bi
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }          "We
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }          "Bi
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }          "We
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }          "Bi
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}          "We
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}          "Bi
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }          "We
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }          "Bi
```

```
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }          "We
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }          "Bi
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}          "We
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}          "Bi
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}               2.9003e+06
     {'Quantized'     }               7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

Quantize a Neural Network for FPGA Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

To run this example, you need the products listed under FPGA in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working directory called `getLogoNetwork.m`. Enter these lines into the file:

```
function net = getLogoNetwork()
    data = getLogoData();
    net  = data.convnet;
end

function data = getLogoData()
    if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
end
```

Load the pretrained network.

```
snet = getLogoNetwork();
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

This example uses the images in the `logos_dataset` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
```

Create a `dlquantizer` object and specify the network to quantize.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
dlQuantObj.calibrate(calibrationData)
```

```
ans =
      Optimized Layer Name         Network Layer Name    Learnables / Activations     MinValue       MaxValue
    ___________________________    __________________    ________________________    ___________    __________

    {'conv_1_Weights'          }    {'conv_1'    }           "Weights"                -0.048978      0.039352
    {'conv_1_Bias'             }    {'conv_1'    }           "Bias"                     0.99996        1.0028
    {'conv_2_Weights'          }    {'conv_2'    }           "Weights"                -0.055518      0.061901
    {'conv_2_Bias'             }    {'conv_2'    }           "Bias"                  -0.00061171       0.00227
    {'conv_3_Weights'          }    {'conv_3'    }           "Weights"                -0.045942      0.046927
    {'conv_3_Bias'             }    {'conv_3'    }           "Bias"                   -0.0013998     0.0015218
    {'conv_4_Weights'          }    {'conv_4'    }           "Weights"                -0.045967          0.051
    {'conv_4_Bias'             }    {'conv_4'    }           "Bias"                    -0.00164     0.0037892
    {'fc_1_Weights'            }    {'fc_1'      }           "Weights"                -0.051394      0.054344
    {'fc_1_Bias'               }    {'fc_1'      }           "Bias"                  -0.00052319    0.00084454
    {'fc_2_Weights'            }    {'fc_2'      }           "Weights"                  -0.05016      0.051557
    {'fc_2_Bias'               }    {'fc_2'      }           "Bias"                   -0.0017564     0.0018502
    {'fc_3_Weights'            }    {'fc_3'      }           "Weights"                -0.050706       0.04678
    {'fc_3_Bias'               }    {'fc_3'      }           "Bias"                    -0.02951      0.024855
    {'imageinput'              }    {'imageinput'}           "Activations"                    0           255
    {'imageinput_normalization'}    {'imageinput'}           "Activations"             -139.34        198.72
```

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To create the target object, enter:

```
hTarget = dlhdl.Target('Intel', 'Interface', 'JTAG');
```

Define a metric function to use to compare the behavior of the network before and after quantization. Save this function in a local file.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% hComputeModelAccuracy test helper function computes model level accuracy statistics

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx, :));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
options = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x, snet, validationData)},'Bitstream','arria10soc_int8',...
'Target',hTarget);
```

To compile and deploy the quantized network, run the `validate` function of the `dlquantizer` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function checks for the Intel Quartus tool and the supported tool version. It then starts programming the FPGA device by using the sof file, displays progress messages, and the time it takes to deploy the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
prediction = dlQuantObj.validate(validationData,options);
```

```
          offset_name          offset_address     allocated_space
    ___________________________    _____________    _________________

    "InputDataOffset"            "0x00000000"     "48.0 MB"
    "OutputResultOffset"         "0x03000000"     "4.0 MB"
    "SystemBufferOffset"         "0x03400000"     "60.0 MB"
    "InstructionDataOffset"      "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"       "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"         "0x08000000"     "12.0 MB"
    "EndOffset"                  "0x08c00000"     "Total: 140.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Jul-2020 12:45:10
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Jul-2020 12:45:26
### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------              -------------              ---------        ---------       ---------
Network                   13570959                    0.09047                    30          380609145            11.8
    conv_module           12667786                    0.08445
        conv_1             3938907                    0.02626
        maxpool_1          1544560                    0.01030
        conv_2             2910954                    0.01941
        maxpool_2           577524                    0.00385
        conv_3             2552707                    0.01702
        maxpool_3           676542                    0.00451
        conv_4              455434                    0.00304
        maxpool_4            11251                    0.00008
    fc_module               903173                    0.00602
        fc_1                536164                    0.00357
        fc_2                342643                    0.00228
        fc_3                 24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------              -------------              ---------        ---------       ---------
Network                   13570364                    0.09047                    30          380612682            11.8
    conv_module           12667103                    0.08445
        conv_1             3939296                    0.02626
        maxpool_1          1544371                    0.01030
        conv_2             2910747                    0.01940
        maxpool_2           577654                    0.00385
        conv_3             2551829                    0.01701
        maxpool_3           676548                    0.00451
        conv_4              455396                    0.00304
        maxpool_4            11355                    0.00008
    fc_module               903261                    0.00602
        fc_1                536206                    0.00357
        fc_2                342688                    0.00228
        fc_3                 24365                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results
```

```
                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                ------------               -------------       ---------        ---------     ---------
Network                          13571561                  0.09048             30          380608338            11.8
    conv_module                  12668340                  0.08446
        conv_1                    3939070                  0.02626
        maxpool_1                 1545327                  0.01030
        conv_2                    2911061                  0.01941
        maxpool_2                  577557                  0.00385
        conv_3                    2552082                  0.01701
        maxpool_3                  676506                  0.00451
        conv_4                     455582                  0.00304
        maxpool_4                   11248                  0.00007
    fc_module                      903221                  0.00602
        fc_1                       536167                  0.00357
        fc_2                       342643                  0.00228
        fc_3                        24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                ------------               -------------       ---------        ---------     ---------
Network                          13569862                  0.09047             30          380613327            11.8
    conv_module                  12666756                  0.08445
        conv_1                    3939212                  0.02626
        maxpool_1                 1543267                  0.01029
        conv_2                    2911184                  0.01941
        maxpool_2                  577275                  0.00385
        conv_3                    2552868                  0.01702
        maxpool_3                  676438                  0.00451
        conv_4                     455353                  0.00304
        maxpool_4                   11252                  0.00008
    fc_module                      903106                  0.00602
        fc_1                       536050                  0.00357
        fc_2                       342645                  0.00228
        fc_3                        24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                ------------               -------------       ---------        ---------     ---------
Network                          13570823                  0.09047             30          380619836            11.8
    conv_module                  12667607                  0.08445
        conv_1                    3939074                  0.02626
        maxpool_1                 1544519                  0.01030
        conv_2                    2910636                  0.01940
        maxpool_2                  577769                  0.00385
        conv_3                    2551800                  0.01701
        maxpool_3                  676795                  0.00451
        conv_4                     455859                  0.00304
        maxpool_4                   11248                  0.00007
    fc_module                      903216                  0.00602
        fc_1                       536165                  0.00357
        fc_2                       342643                  0.00228
        fc_3                        24406                  0.00016
 * The clock frequency of the DL processor is: 150MHz


          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________
```

```
    "InputDataOffset"           "0x00000000"     "48.0 MB"
    "OutputResultOffset"        "0x03000000"     "4.0 MB"
    "SystemBufferOffset"        "0x03400000"     "60.0 MB"
    "InstructionDataOffset"     "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"        "0x08000000"     "12.0 MB"
    "EndOffset"                 "0x08c00000"     "Total: 140.0 MB"

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target FPGA.
### Deep learning network programming has been skipped as the same network is already loaded on the target FPGA.
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------              -------------              ---------        ---------       ---------
Network                   13572329                    0.09048                    10          127265075            11.8
    conv_module           12669135                    0.08446
        conv_1             3939559                    0.02626
        maxpool_1          1545378                    0.01030
        conv_2             2911243                    0.01941
        maxpool_2           577422                    0.00385
        conv_3             2552064                    0.01701
        maxpool_3           676678                    0.00451
        conv_4              455657                    0.00304
        maxpool_4            11227                    0.00007
    fc_module               903194                    0.00602
        fc_1                536140                    0.00357
        fc_2                342688                    0.00228
        fc_3                 24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------              -------------              ---------        ---------       ---------
Network                   13572527                    0.09048                    10          127266427            11.8
    conv_module           12669266                    0.08446
        conv_1             3939776                    0.02627
        maxpool_1          1545632                    0.01030
        conv_2             2911169                    0.01941
        maxpool_2           577592                    0.00385
        conv_3             2551613                    0.01701
        maxpool_3           676811                    0.00451
        conv_4              455418                    0.00304
        maxpool_4            11348                    0.00008
    fc_module               903261                    0.00602
        fc_1                536205                    0.00357
        fc_2                342689                    0.00228
        fc_3                 24365                    0.00016
 * The clock frequency of the DL processor is: 150MHz
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
validateOut = prediction.MetricResults.Result
```

```
ans =
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}         0.9875
```

```
    {'Quantized'    }         0.9875
```

Examine the `QuantizedNetworkFPS` field of the validation output to see the frames per second performance of the quantized network.

```
prediction.QuantizedNetworkFPS
```

```
ans = 11.8126
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `validate` | `dlquantizer`

Topics
“Quantization of Deep Neural Networks”

Introduced in R2020a

dlquantizer

Quantize a deep neural network to 8-bit scaled integer data types

Description

Use the `dlquantizer` object to reduce the memory requirement of a deep neural network by quantizing weights, biases, and activations to 8-bit scaled integer data types.

Creation

Syntax

```
quantObj = dlquantizer(net)
quantObj = dlquantizer(net,Name,Value)
```

Description

`quantObj = dlquantizer(net)` creates a `dlquantizer` object for the specified network.

`quantObj = dlquantizer(net,Name,Value)` creates a `dlquantizer` object for the specified network, with additional options specified by one or more name-value pair arguments.

Use `dlquantizer` to create an quantized network for GPU, FPGA, or CPU deployment. To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.

Input Arguments

`net` — Pretrained neural network
`DAGNetwork` object | `SeriesNetwork` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object

Pretrained neural network, specified as a `DAGNetwork`, `SeriesNetwork`, `yolov2ObjectDetector`, or a `ssdObjectDetector` object.

Quantization of `yolov2ObjectDetector` and `ssdObjectDetector` networks requires a GPU Coder license.

Properties

`NetworkObject` — Pretrained neural network
`DAGNetwork` object | `SeriesNetwork` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object

Pre-trained neural network, specified as a `DAGNetwork`, `SeriesNetwork`, `yolov2ObjectDetector`, or a `ssdObjectDetector` object.

`ExecutionEnvironment` — Execution environment
'GPU' (default) | 'FPGA' | 'CPU'

Specify the execution environment for the quantized network. When this parameter is not specified the default execution environment is GPU. To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.

Example: `'ExecutionEnvironment','FPGA'`

`Simulation` — Enable or disable MATLAB simulation workflow
'off (default) | 'on'

Enable or disable the MATLAB simulation workflow. When this parameter is set to on, the quantized network is validated by simulating the quantized network in MATLAB and comparing the single data type network prediction results to the simulated network prediction results.

Example: `'Simulation', 'on'`

Object Functions

calibrate	Simulate and collect ranges of a deep neural network
validate	Quantize and validate a deep neural network

Examples

Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load net
net
```

```
net =
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all

layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

In this example, use the images in the `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

Create a `dlquantizer` object and specify the network to quantize.

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare with predicted label with actual ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=95×5 table
                  Optimized Layer Name                    Network Layer Name        Learnables
```

```
    _____________________________________________________    ____________________________    _________

    {'conv1_relu_conv1_Weights'                         }    {'relu_conv1'           }          "We
    {'conv1_relu_conv1_Bias'                            }    {'relu_conv1'           }          "Bi
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}          "We
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}          "Bi
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }          "We
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }          "Bi
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }          "We
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }          "Bi
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}          "We
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}          "Bi
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }          "We
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }          "Bi
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }          "We
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }          "Bi
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}          "We
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}          "Bi
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

Quantize a Neural Network for FPGA Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working directory called `getLogoNetwork.m`. Enter these lines into the file:

```
function net = getLogoNetwork()
    data = getLogoData();
    net  = data.convnet;
end

function data = getLogoData()
    if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
end
```

Load the pretrained network.

```
snet = getLogoNetwork();
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

This example uses the images in the `logos_dataset` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
```

Create a `dlquantizer` object and specify the network to quantize.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
 dlQuantObj.calibrate(calibrationData)
```

```
ans =
        Optimized Layer Name         Network Layer Name    Learnables / Activations    MinValue      MaxValue
    _____________________________    __________________    ________________________    __________    _________

    {'conv_1_Weights'          }      {'conv_1'    }           "Weights"                 -0.048978      0.039352
    {'conv_1_Bias'             }      {'conv_1'    }           "Bias"                      0.99996        1.0028
    {'conv_2_Weights'          }      {'conv_2'    }           "Weights"                 -0.055518      0.061901
    {'conv_2_Bias'             }      {'conv_2'    }           "Bias"                   -0.00061171       0.00227
    {'conv_3_Weights'          }      {'conv_3'    }           "Weights"                 -0.045942      0.046927
    {'conv_3_Bias'             }      {'conv_3'    }           "Bias"                    -0.0013998     0.0015218
    {'conv_4_Weights'          }      {'conv_4'    }           "Weights"                 -0.045967          0.051
    {'conv_4_Bias'             }      {'conv_4'    }           "Bias"                     -0.00164     0.0037892
    {'fc_1_Weights'            }      {'fc_1'      }           "Weights"                 -0.051394      0.054344
    {'fc_1_Bias'               }      {'fc_1'      }           "Bias"                   -0.00052319    0.00084454
    {'fc_2_Weights'            }      {'fc_2'      }           "Weights"                  -0.05016      0.051557
    {'fc_2_Bias'               }      {'fc_2'      }           "Bias"                    -0.0017564     0.0018502
    {'fc_3_Weights'            }      {'fc_3'      }           "Weights"                 -0.050706       0.04678
    {'fc_3_Bias'               }      {'fc_3'      }           "Bias"                     -0.02951      0.024855
    {'imageinput'              }      {'imageinput'}           "Activations"                     0           255
    {'imageinput_normalization'}      {'imageinput'}           "Activations"              -139.34        198.72
```

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To create the target object, enter:

```
hTarget = dlhdl.Target('Intel', 'Interface', 'JTAG');
```

Define a metric function to use to compare the behavior of the network before and after quantization. Save this function in a local file.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% hComputeModelAccuracy test helper function computes model level accuracy statistics

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx, :));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
options = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x, snet, validationData)},'Bitstream','arria10soc_int8',...
'Target',hTarget);
```

To compile and deploy the quantized network, run the `validate` function of the `dlquantizer` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function checks for the Intel Quartus tool and the supported tool version. It then starts programming the FPGA device by using the sof file, displays progress messages, and the time it takes to deploy the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
prediction = dlQuantObj.validate(validationData,options);
```

```
          offset_name          offset_address     allocated_space
    ___________________________    ______________    _________________

    "InputDataOffset"            "0x00000000"     "48.0 MB"
    "OutputResultOffset"         "0x03000000"     "4.0 MB"
    "SystemBufferOffset"         "0x03400000"     "60.0 MB"
    "InstructionDataOffset"      "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"       "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"         "0x08000000"     "12.0 MB"
    "EndOffset"                  "0x08c00000"     "Total: 140.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Jul-2020 12:45:10
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Jul-2020 12:45:26
### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         _____________              _____________              _________        _________       _________
Network                   13570959                    0.09047                     30          380609145            11.8
    conv_module           12667786                    0.08445
        conv_1             3938907                    0.02626
        maxpool_1          1544560                    0.01030
        conv_2             2910954                    0.01941
        maxpool_2           577524                    0.00385
        conv_3             2552707                    0.01702
        maxpool_3           676542                    0.00451
        conv_4              455434                    0.00304
        maxpool_4            11251                    0.00008
    fc_module               903173                    0.00602
        fc_1                536164                    0.00357
        fc_2                342643                    0.00228
        fc_3                 24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         _____________              _____________              _________        _________       _________
Network                   13570364                    0.09047                     30          380612682            11.8
    conv_module           12667103                    0.08445
        conv_1             3939296                    0.02626
```

```
        maxpool_1          1544371                  0.01030
        conv_2             2910747                  0.01940
        maxpool_2           577654                  0.00385
        conv_3             2551829                  0.01701
        maxpool_3           676548                  0.00451
        conv_4              455396                  0.00304
        maxpool_4            11355                  0.00008
    fc_module               903261                  0.00602
        fc_1                536206                  0.00357
        fc_2                342688                  0.00228
        fc_3                 24365                  0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                          -------------              -------------               ---------        ---------       ---------
Network                    13571561                  0.09048                     30          380608338            11.8
    conv_module            12668340                  0.08446
        conv_1              3939070                  0.02626
        maxpool_1           1545327                  0.01030
        conv_2              2911061                  0.01941
        maxpool_2            577557                  0.00385
        conv_3              2552082                  0.01701
        maxpool_3            676506                  0.00451
        conv_4               455582                  0.00304
        maxpool_4             11248                  0.00007
    fc_module                903221                  0.00602
        fc_1                 536167                  0.00357
        fc_2                 342643                  0.00228
        fc_3                  24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                          -------------              -------------               ---------        ---------       ---------
Network                    13569862                  0.09047                     30          380613327            11.8
    conv_module            12666756                  0.08445
        conv_1              3939212                  0.02626
        maxpool_1           1543267                  0.01029
        conv_2              2911184                  0.01941
        maxpool_2            577275                  0.00385
        conv_3              2552868                  0.01702
        maxpool_3            676438                  0.00451
        conv_4               455353                  0.00304
        maxpool_4             11252                  0.00008
    fc_module                903106                  0.00602
        fc_1                 536050                  0.00357
        fc_2                 342645                  0.00228
        fc_3                  24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                          -------------              -------------               ---------        ---------       ---------
```

```
Network                   13570823                 0.09047                      30          380619836            11.8
    conv_module           12667607                 0.08445
        conv_1             3939074                 0.02626
        maxpool_1          1544519                 0.01030
        conv_2             2910636                 0.01940
        maxpool_2           577769                 0.00385
        conv_3             2551800                 0.01701
        maxpool_3           676795                 0.00451
        conv_4              455859                 0.00304
        maxpool_4            11248                 0.00007
    fc_module               903216                 0.00602
        fc_1                536165                 0.00357
        fc_2                342643                 0.00228
        fc_3                 24406                 0.00016
 * The clock frequency of the DL processor is: 150MHz


          offset_name          offset_address     allocated_space
    ___________________________    _____________    _________________

    "InputDataOffset"             "0x00000000"     "48.0 MB"
    "OutputResultOffset"          "0x03000000"     "4.0 MB"
    "SystemBufferOffset"          "0x03400000"     "60.0 MB"
    "InstructionDataOffset"       "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"        "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"          "0x08000000"     "12.0 MB"
    "EndOffset"                   "0x08c00000"     "Total: 140.0 MB"

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target FPGA.
### Deep learning network programming has been skipped as the same network is already loaded on the target FPGA.
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                  -------------              -------------                   ---------        ---------       ---------
Network                    13572329                 0.09048                      10          127265075            11.8
    conv_module            12669135                 0.08446
        conv_1              3939559                 0.02626
        maxpool_1           1545378                 0.01030
        conv_2              2911243                 0.01941
        maxpool_2            577422                 0.00385
        conv_3              2552064                 0.01701
        maxpool_3            676678                 0.00451
        conv_4               455657                 0.00304
        maxpool_4             11227                 0.00007
    fc_module                903194                 0.00602
        fc_1                 536140                 0.00357
        fc_2                 342688                 0.00228
        fc_3                  24364                 0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                  -------------              -------------                   ---------        ---------       ---------
Network                    13572527                 0.09048                      10          127266427            11.8
    conv_module            12669266                 0.08446
        conv_1              3939776                 0.02627
        maxpool_1           1545632                 0.01030
        conv_2              2911169                 0.01941
        maxpool_2            577592                 0.00385
        conv_3              2551613                 0.01701
        maxpool_3            676811                 0.00451
        conv_4               455418                 0.00304
```

```
       maxpool_4            11348                0.00008
   fc_module               903261                0.00602
       fc_1                536205                0.00357
       fc_2                342689                0.00228
       fc_3                 24365                0.00016
 * The clock frequency of the DL processor is: 150MHz
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
validateOut = prediction.MetricResults.Result
```

```
ans =
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}          0.9875
     {'Quantized'     }          0.9875
```

Examine the `QuantizedNetworkFPS` field of the validation output to see the frames per second performance of the quantized network.

```
prediction.QuantizedNetworkFPS
```

```
ans = 11.8126
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

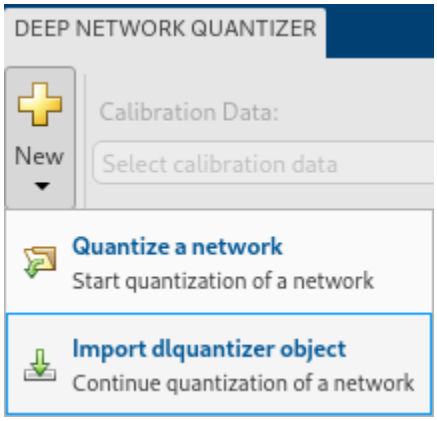
Import a `dlquantizer` Object into the Deep Network Quantizer App

This example shows you how to import a `dlquantizer` object from the base workspace into the **Deep Network Quantizer** app. This allows you to begin quantization of a deep neural network using the command line or the app, and resume your work later in the app.

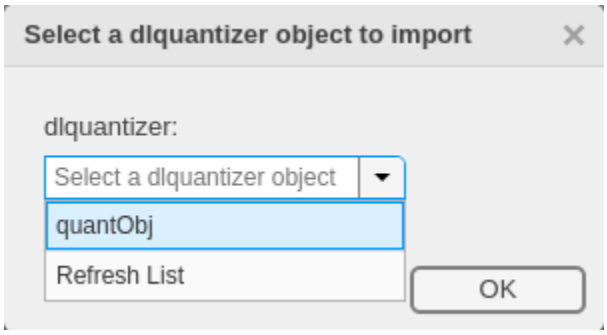
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

In the app, click **New** and select `Import dlquantizer object`.

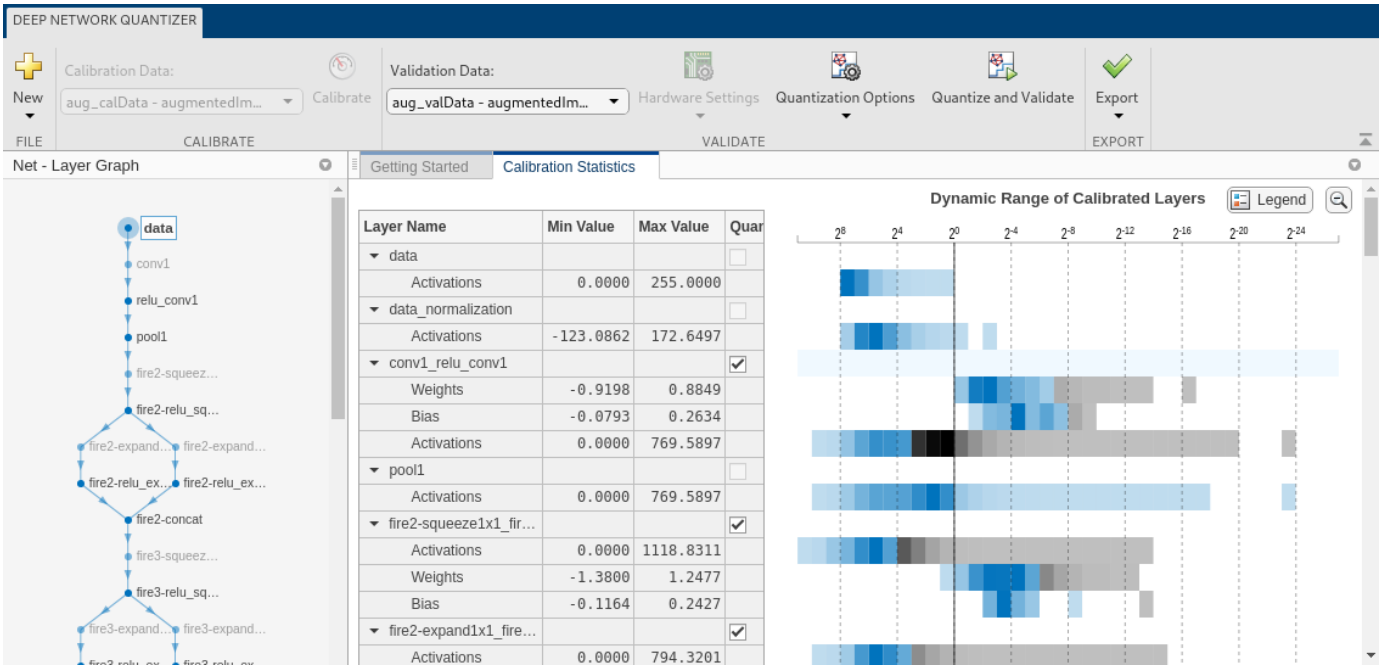


In the dialog, select the `dlquantizer` object to import from the base workspace. For this example, use `quantObj` that you create in the above example Quantize a Neural Network for GPU Target.



The app imports any data contained in the `dlquantizer` object that was collected at the command line. This data can include the network to quantize, calibration data, validation data, and calibration statistics.

The app displays a table containing the calibration data contained in the imported `dlquantizer` object, `quantObj`. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks”.



Quantize a Network for FPGA Deployment

To explore the behavior of a neural network that has quantized convolution layers, use the **Deep Network Quantizer** app. This example quantizes the learnable parameters of the convolution layers of the `LogoNet` neural network.

For this example, you need the products listed under FPGA in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working folder called `getLogoNetwork.m`. In the file, enter:

```
function net = getLogoNetwork
 if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
    net  = data.convnet;
end
```

Load the pretrained network.

```
snet = getLogoNetwork;
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The app uses calibration data to exercise the network and collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network. The app also exercises the dynamic ranges of the activations in all layers of the LogoNet network. For the best quantization results, the calibration data must be representative of inputs to the LogoNet network.

After quantization, the app uses the validation data set to test the network to understand the effects of the limited range and precision of the quantized learnable parameters of the convolution layers in the network.

In this example, use the images in the `logos_dataset` data set to calibrate and validate the LogoNet network. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

Expedite the calibration and validation process by using a subset of the `calibrationData` and `validationData`. Store the new reduced calibration data set in `calibrationData_concise` and the new reduced validation data set in `validationData_concise`.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_concise = calibrationData.subset(1:20);
validationData_concise = vaidationData.subset(1:1);
```

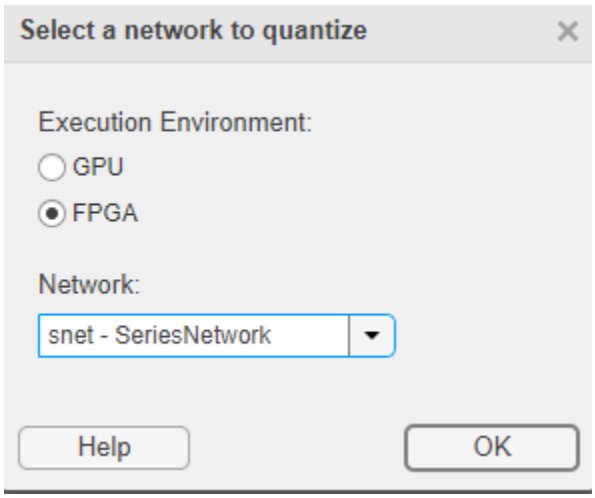
At the MATLAB command prompt, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Click **New** and select `Quantize a network`.

The app verifies your execution environment.

Select the execution environment and the network to quantize from the base workspace. For this example, select a FPGA execution environment and the series network `snet`.



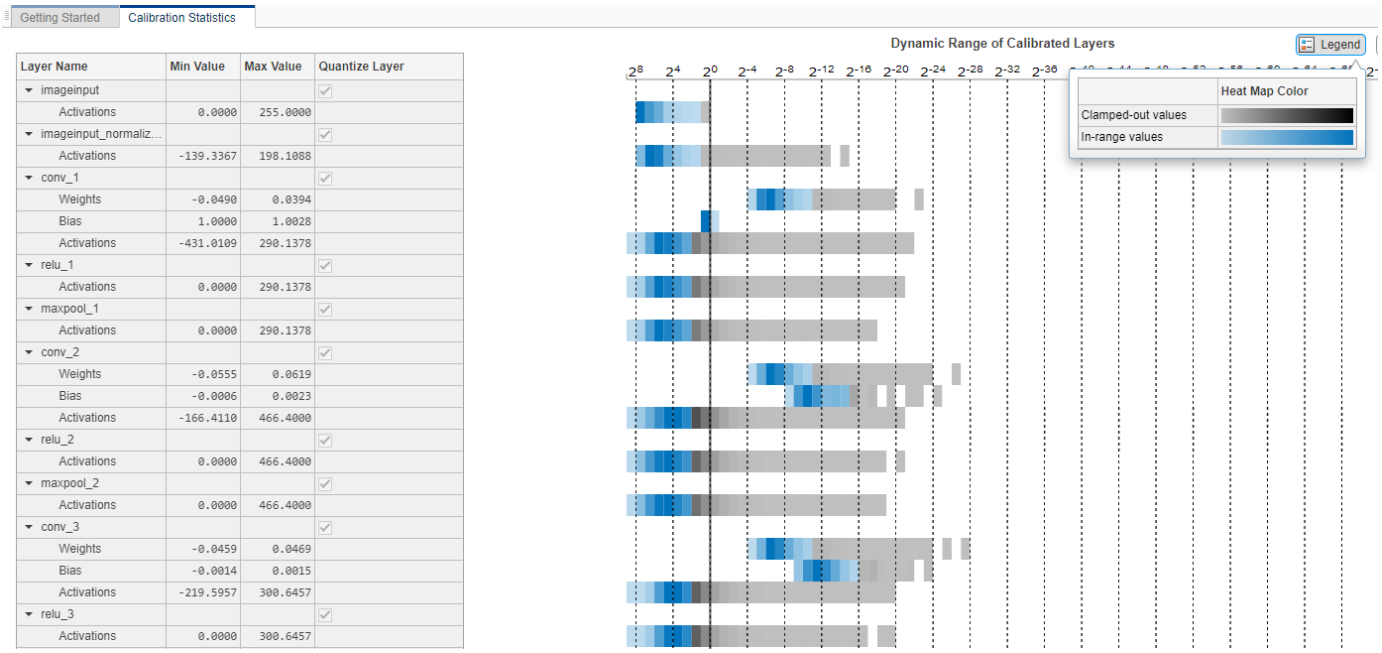
The app displays the layer graph of the selected network.

In the **Calibrate** section of the app toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data `calibrationData_concise`.

Click **Calibrate**.

The **Deep Network Quantizer** app uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

When the calibration is complete, the app displays a table containing the weights and biases in the convolution and fully connected layers of the network. Also displayed are the dynamic ranges of the activations in all layers of the network and their minimum and maximum values during the calibration. The app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks”.



In the **Quantize** column of the table, indicate whether to quantize the learnable parameters in the layer. You cannot quantize layers that are not convolution layers. Layers that are not quantized remain in single-precision.

In the **Validate** section of the app toolstrip, under **Validation Data**, select the `augmentedImageDatastore` object from the base workspace containing the validation data `validationData_concise`.

In the **Hardware Settings** section of the toolstrip, select from the options listed in the table:

Simulation Environment	**Action**
MATLAB (Simulate in MATLAB)	Simulates the quantized network in MATLAB. Validates the quantized network by comparing performance to single-precision version of the network.
Intel Arria 10 SoC (arria10soc_int8)	Deploys the quantized network to an Intel Arria 10 SoC board by using the `arria10soc_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.
Xilinx ZCU102 (zcu102_int8)	Deploys the quantized network to a Xilinx Zynq UltraScale+ MPSoC ZCU102 10 SoC board by using the `zcu102_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.

Xilinx ZC706 (zc706_int8)	Deploys the quantized network to a Xilinx Zynq-7000 ZC706 board by using the `zc706_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network.

When you select the `Intel Arria 10 SoC (arria10soc_int8)`, `Xilinx ZCU102 (zcu102_int8)`, or `Xilinx ZC706 (zc706_int8)` options, select the interface to use to deploy and validate the quantized network. The **Target** interface options are listed in this table.

Target Option	**Action**
JTAG	Programs the target FPGA board selected in **Simulation Environment** by using a JTAG cable. For more information, see "JTAG Connection" (Deep Learning HDL Toolbox)
Ethernet	Programs the target FPGA board selected in **Simulation Environment** through the Ethernet interface. Specify the IP address for your target board in **IP Address**.

For this example, select `Xilinx ZCU102 (zcu102_int8)`, select **Ethernet**, and enter the board IP address.

In the **Validate** section of the app toolstrip, under **Quantization Options**, select the **Default** metric function.

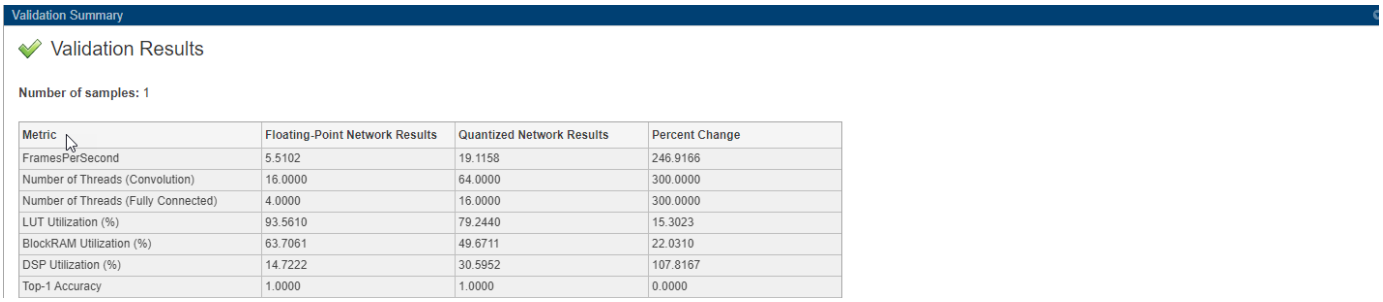
Click **Quantize and Validate**.

The **Deep Network Quantizer** app quantizes the weights, activations, and biases of convolution layers in the network to scaled 8-bit integer data types and uses the validation data to exercise the network. The app determines a metric function to use for the validation based on the type of network that is being quantized.

Type of Network	Metric Function
Classification	**Top-1 Accuracy** – Accuracy of the network
Object Detection	**Average Precision** – Average precision over all detection results. See `evaluateDetectionPrecision`.
Regression	**MSE** – Mean squared error of the network
Semantic Segmentation	`evaluateSemanticSegmentation` – Evaluate semantic segmentation data set against ground truth
Single Shot Detector (SSD)	**WeightedIOU** – Average IoU of each class, weighted by the number of pixels in that class

When the validation is complete, the app displays the results of the validation, including:

- Metric function used for validation
- Result of the metric function before and after quantization



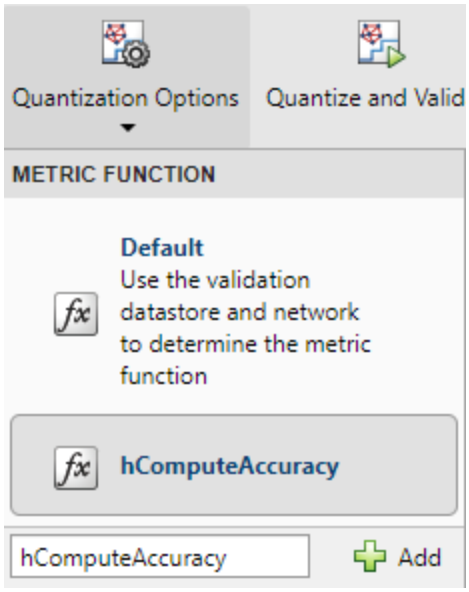
If you want to use a different metric function for validation, for example to use the Top-5 accuracy metric function instead of the default Top-1 accuracy metric function, you can define a custom metric function. Save this function in a local file.

```
function accuracy = hComputeAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
```

```
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

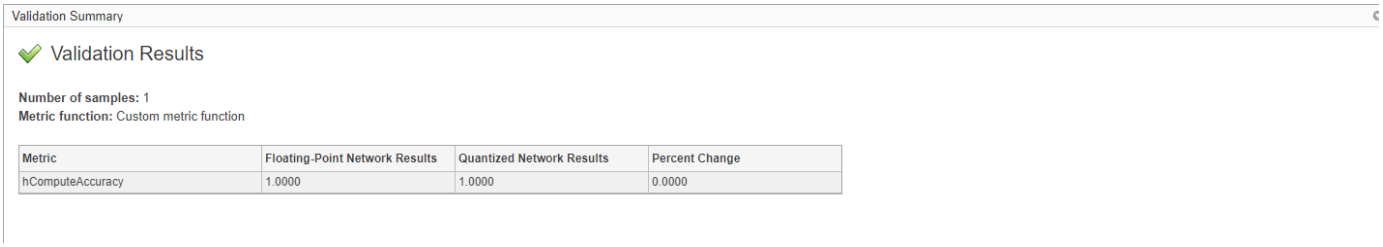
To revalidate the network by using this custom metric function, under **Quantization Options**, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function to use.

The custom metric function must be on the path. If the metric function is not on the path, this step produces an error.



Click **Quantize and Validate**.

The app quantizes the network and displays the validation results for the custom metric function.



The app displays only scalar values in the validation results table. To view the validation results for a custom metric function with nonscalar output, export the `dlquantizer` object, then validate the quantized network by using the `validate` function in the MATLAB command window.

After quantizing and validating the network, you can choose to export the quantized network.

Click the **Export** button. In the drop-down list, select `Export Quantizer` to create a `dlquantizer` object in the base workspace. You can deploy the quantized network to your target FPGA board and retrieve the prediction results by using MATLAB. See, “Deploy Quantized Network Example” (Deep Learning HDL Toolbox).

See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `validate` | `dlquantizationOptions`

Topics
“Quantization of Deep Neural Networks”
“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”
“Quantize Object Detectors and Generate CUDA® Code”
“Quantize Network for FPGA Deployment” (Deep Learning HDL Toolbox)
“Deploy Quantized Network Example” (Deep Learning HDL Toolbox)
“Classify Images on an FPGA Using a Quantized DAG Network” (Deep Learning HDL Toolbox)
“Code Generation for Quantized Deep Learning Network on Raspberry Pi” (MATLAB Coder)

Introduced in R2020a

dlupdate

Update parameters using custom function

Syntax

```
dlnet = dlupdate(fun,dlnet)
params = dlupdate(fun,params)
[___] = dlupdate(fun,___A1,...,An)
[___,X1,...,Xm] = dlupdate(fun,___)
```

Description

`dlnet = dlupdate(fun,dlnet)` updates the learnable parameters of the `dlnetwork` object `dlnet` by evaluating the function `fun` with each learnable parameter as an input. `fun` is a function handle to a function that takes one parameter array as an input argument and returns an updated parameter array.

`params = dlupdate(fun,params)` updates the learnable parameters in `params` by evaluating the function `fun` with each learnable parameter as an input.

`[___] = dlupdate(fun,___A1,...,An)` also specifies additional input arguments, in addition to the input arguments in previous syntaxes, when `fun` is a function handle to a function that requires `n+1` input values.

`[___,X1,...,Xm] = dlupdate(fun,___)` returns multiple outputs `X1,...,Xm` when `fun` is a function handle to a function that returns `m+1` output values.

Examples

L1 Regularization with dlupdate

Perform L1 regularization on a structure of parameter gradients.

Create the sample input data.

```
dlX = dlarray(rand(100,100,3),'SSC');
```

Initialize the learnable parameters for the convolution operation.

```
params.Weights = dlarray(rand(10,10,3,50));
params.Bias = dlarray(rand(50,1));
```

Calculate the gradients for the convolution operation using the helper function `convGradients`, defined at the end of this example.

```
gradients = dlfeval(@convGradients,dlX,params);
```

Define the regularization factor.

```
L1Factor = 0.001;
```

Create an anonymous function that regularizes the gradients. By using an anonymous function to pass a scalar constant to the function, you can avoid having to expand the constant value to the same size and structure as the parameter variable.

```
L1Regularizer = @(grad,param) grad + L1Factor.*sign(param);
```

Use `dlupdate` to apply the regularization function to each of the gradients.

```
gradients = dlupdate(L1Regularizer,gradients,params);
```

The gradients in `grads` are now regularized according to the function `L1Regularizer`.

`convGradients` Function

The `convGradients` helper function takes the learnable parameters of the convolution operation and a mini-batch of input data `dlX`, and returns the gradients with respect to the learnable parameters.

```
function gradients = convGradients(dlX,params)
dlY = dlconv(dlX,params.Weights,params.Bias);
dlY = sum(dlY,'all');
gradients = dlgradient(dlY,params);
end
```

Use `dlupdate` to Train Network Using Custom Update Function

Use `dlupdate` to train a network using a custom update function that implements the stochastic gradient descent algorithm (without momentum).

Load Training Data

Load the digits training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

Define the Network

Define the network architecture and specify the average image value 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')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of this example. The function takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Define Stochastic Gradient Descent Function

Create the helper function `sgdFunction`, listed at the end of this example. The function takes `param` and `paramGradient`, a learnable parameter and the gradient of the loss with respect to that parameter, respectively, and returns the updated parameter using the stochastic gradient descent algorithm, expressed as

$$\theta_{l+1} = \theta - \alpha \nabla E(\theta_l)$$

where l is the iteration number, $\alpha > 0$ is the learning rate, θ is the parameter vector, and $E(\theta)$ is the loss function.

Specify Training Options

Specify the options to use during training.

```
miniBatchSize = 128;
numEpochs = 30;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Specify the learning rate.

```
learnRate = 0.01;
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

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

Visualize the training progress in a plot.

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

Train Network

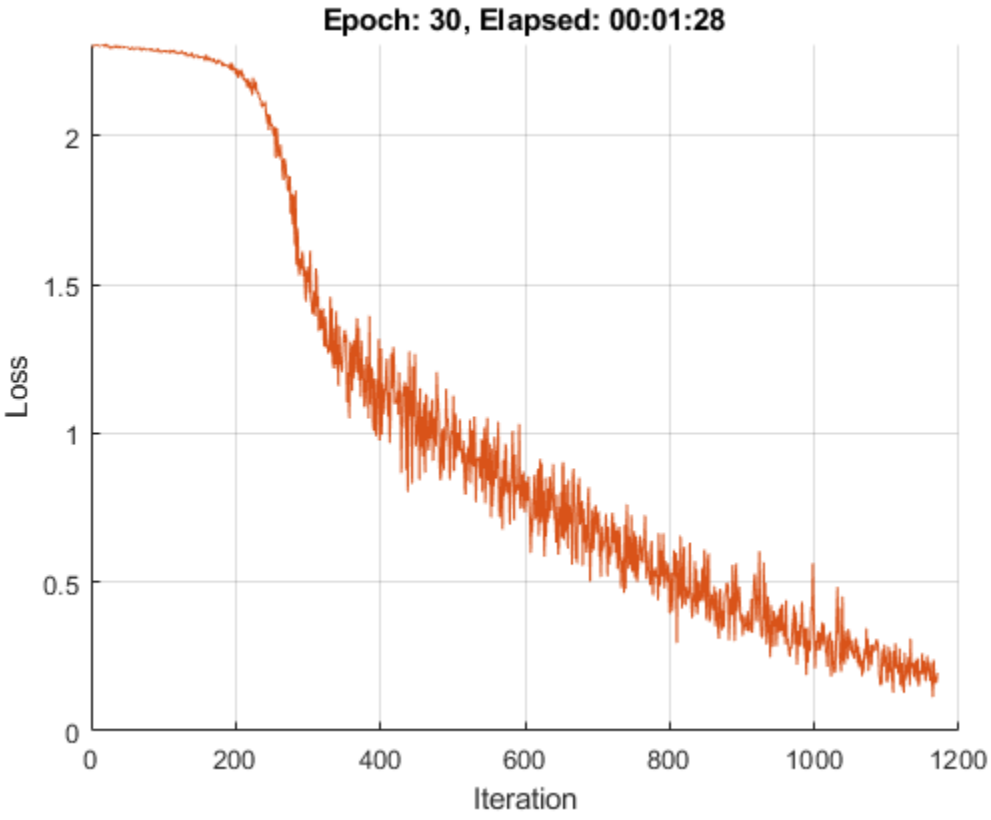
Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters by calling `dlupdate` with the function `sgdFunction` defined at the end of this example. At the end of each epoch, display the training progress.

Initialize the training progress plot.

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

Train the network.

```
iteration = 0;
start = tic;

for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);

    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 a gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients helper function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

        % Update the network parameters using the SGD algorithm defined in
        % the sgdFunction helper function.
        updateFcn = @(dlnet,gradients) sgdFunction(dlnet,gradients,learnRate);
        dlnet = dlupdate(updateFcn,dlnet,gradients);

        % 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 Network

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

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

To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.9386
```

Model Gradients Function

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

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

dlYPred = forward(dlnet,dlX);

loss = crossentropy(dlYPred,Y);

gradients = dlgradient(loss,dlnet.Learnables);

end
```

Stochastic Gradient Descent Function

The helper function `sgdFunction` takes the learnable parameter `parameter`, the gradients of that parameter with respect to the loss `gradient`, and the learning rate `learnRate`, and returns the updated parameter using the stochastic gradient descent algorithm, expressed as

$$\theta_{l+1} = \theta - \alpha \nabla E(\theta_l)$$

where l is the iteration number, $\alpha > 0$ is the learning rate, θ is the parameter vector, and $E(\theta)$ is the loss function.

```
function parameter = sgdFunction(parameter,gradient,learnRate)

parameter = parameter - learnRate .* gradient;

end
```

Input Arguments

`fun` — Function to apply

function handle

Function to apply to the learnable parameters, specified as a function handle.

`dlupdate` evaluates `fun` with each network learnable parameter as an input. `fun` is evaluated as many times as there are arrays of learnable parameters in `dlnet` or `params`.

`dlnet` — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object. `dlnet.Learnables` is a table with three variables:

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.

- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

`params` — Network learnable parameters
`dlarray` | numeric array | cell array | structure | table

Network learnable parameters, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

If you specify `params` as a table, it must contain the following three variables.

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

You can specify `params` as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `A1,...,An` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

Data Types: `single` | `double` | `struct` | `table` | `cell`

`A1,...,An` — Additional input arguments
`dlarray` | numeric array | cell array | structure | table

Additional input arguments to `fun`, specified as `dlarray` objects, numeric arrays, cell arrays, structures, or tables with a `Value` variable.

The exact form of `A1,...,An` depends on the input network or learnable parameters. The following table shows the required format for `A1,...,An` for possible inputs to `dlupdate`.

Input	Learnable Parameters	`A1,...,An`
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `A1,...,An` must have a `Value` variable consisting of cell arrays that contain the additional input arguments for the function `fun` to apply to each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`.
	Numeric array	Numeric array with the same data type and ordering as `params`.
	Cell array	Cell array with the same data types, structure, and ordering as `params`.

Input	Learnable Parameters	A1,...,An
	Structure	Structure with the same data types, fields, and ordering as `params`.
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables and ordering as `params`. `A1,...,An` must have a `Value` variable consisting of cell arrays that contain the additional input argument for the function `fun` to apply to each learnable parameter.

Output Arguments

`dlnet` — Updated network

`dlnetwork` object

Network, returned as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object.

`params` — Updated network learnable parameters

`dlarray` | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table with a `Value` variable containing the updated learnable parameters of the network.

`X1,...,Xm` — Additional output arguments

`dlarray` | numeric array | cell array | structure | table

Additional output arguments from the function `fun`, where `fun` is a function handle to a function that returns multiple outputs, returned as `dlarray` objects, numeric arrays, cell arrays, structures, or tables with a `Value` variable.

The exact form of `X1,...,Xm` depends on the input network or learnable parameters. The following table shows the returned format of `X1,...,Xm` for possible inputs to `dlupdate`.

Input	Learnable parameters	X1,...,Xm
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `X1,...,Xm` has a `Value` variable consisting of cell arrays that contain the additional output arguments of the function `fun` applied to each learnable parameter.

Input	Learnable parameters	X1,...,Xm
params	dlarray	dlarray with the same data type and ordering as params.
	Numeric array	Numeric array with the same data type and ordering as params.
	Cell array	Cell array with the same data types, structure, and ordering as params.
	Structure	Structure with the same data types, fields, and ordering as params.
	Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.	Table with the same data types, variables. and ordering as params. X1,...,Xm has a Value variable consisting of cell arrays that contain the additional output argument of the function fun applied to each learnable parameter.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.
  - params
  - A1,...,An

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

dlnetwork | dlarray | adamupdate | rmspropupdate | sgdmupdate | dlgradient | dlfeval

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”
“Sequence-to-Sequence Translation Using Attention”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2019b

dltranspconv

Deep learning transposed convolution

Syntax

```
dlY = dltranspconv(dlX,weights,bias)
dlY = dltranspconv(dlX,weights,bias,'DataFormat',FMT)
dlY = dltranspconv( ___ Name,Value)
```

Description

The transposed convolution operation upsamples feature maps.

Note This function applies the deep learning transposed convolution operation to `dlarray` data. If you want to apply transposed convolution within a `layerGraph` object or `Layer` array, use one of the following layers:

- `transposedConv2dLayer`
- `transposedConv3dLayer`

`dlY = dltranspconv(dlX,weights,bias)` computes the deep learning transposed convolution of the input `dlX` using the filters defined by `weights`, and adds a constant `bias`. The input `dlX` must be a formatted `dlarray`. Transposed convolution acts on dimensions that you specify as `'S'` and `'C'` dimensions. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`.

`dlY = dltranspconv(dlX,weights,bias,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`.

`dlY = dltranspconv( ___ Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'Stride',3` sets the stride of the convolution operation.

Examples

Upsample Image Using Transposed Convolution

Convolve an image and then use transposed convolution to resize the convolved image to the same size as the original image.

Import the image data and convert it to a `dlarray`.

```
X = imread('sherlock.jpg');
dlX = dlarray(single(X),'SSC');
```

Display the image.

```
imshow(X)
```

Initialize the convolutional filters and bias term. Specify an ungrouped convolution that applies a single filter to all three channels of the input data.

```
filterHeight = 10;
filterWidth = 10;
numChannelsPerGroup = 3;
numFiltersPerGroup = 1;
numGroups = 1;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the convolution. Use a `'Stride'` value of 2 and a `'DilationFactor'` value of 2.

```
dlY = dlconv(dlX,weights,bias,'Stride',2,'DilationFactor',3);
```

Display the convolved image.

```
Y = extractdata(dlY);
imshow(rescale(Y))
```

Initialize the transposed convolutional filters and bias. Specify an ungrouped transposed convolution that applies three filters to the input. Use the same filter height and filter width as for the convolution operation.

```
numChannelsPerGroupTC = 1;
numFiltersPerGroupTC = 3;

weightsTC = rand(filterHeight,filterWidth,numFiltersPerGroupTC,numChannelsPerGroupTC,numGroups);
biasTC = rand(numFiltersPerGroupTC*numGroups,1);
```

Perform the transposed convolution. Use the same stride and dilation factor as for the convolution operation.

```
dlZ = dltranspconv(dlY,weightsTC,biasTC,'Stride',2,'DilationFactor',3);
```

Display the image after the transposed convolution.

```
Z = extractdata(dlZ);
imshow(rescale(Z))
```

Compare the size of the original image, the convolved image, and the image after the transposed convolution.

```
sizeX = size(X)
```

```
sizeX = 1×3

   640   960     3
```

```
sizeY = size(Y)
```

```
sizeY = 1×2

   307   467
```

```
sizeZ = size(Z)
```

```
sizeZ = 1×3

   640   960     3
```

The transposed convolution upsamples the convolved data to the size of the original input data.

Perform Grouped Transposed Convolution

Apply transposed convolution to the input data in three groups of two channels each. Apply four filters per group.

Create the input data as ten observations of size 100-by-100 with six channels.

```
height = 100;
width = 100;
channels = 6;
numObservations = 10;

X = rand(height,width,channels,numObservations);
dlX = dlarray(X,'SSCB');
```

Initialize the filters for the transposed convolution operation. Specify three groups of transposed convolutions that each apply four filters to two channels of the input data.

```
filterHeight = 8;
filterWidth = 8;
numChannelsPerGroup = 2;
numFiltersPerGroup = 4;
numGroups = 3;

weights = rand(filterHeight,filterWidth,numFiltersPerGroup,numChannelsPerGroup,numGroups);
```

Initialize the bias term.

```
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the transposed convolution.

```
dlY = dltranspconv(dlX,weights,bias);
size(dlY)
```

```
ans = 1×4

   107   107    12    10
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

The 12 channels of the convolution output represent the three groups of transposed convolutions with four filters per group.

Input Arguments

dlX — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `weights` or `bias` must be a `dlarray`.

Convolution acts on dimensions that you specify as spatial dimensions using the `'S'` dimension label. You can specify up to three dimensions in `dlX` as `'S'` dimensions.

Data Types: `single` | `double`

`weights` — Filters
`dlarray` | numeric array

Filters, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. The `weights` argument specifies the size and values of the filters, as well as the number of filters and the number of groups for grouped transposed convolutions.

Specify weights as a `filterSize`-by-`numFiltersPerGroup`-by-`numChannelsPerGroup`-by-`numGroups` array.

- `filterSize` — Size of the convolutional filters. `filterSize` can have up to three dimensions, depending on the number of spatial dimensions in the input data.

Input Data `'S'` Dimensions	`filterSize`
1-D	*h*, where *h* corresponds to the height of the filter
2-D	*h*-by-*w*, where *h* and *w* correspond to the height and width of the filter, respectively
3-D	*h*-by-*w*-by-*d*, where *h*, *w*, and *d* correspond to the height, width, and depth of the filter, respectively

- `numFiltersPerGroup` — Number of filters to apply within each group.
- `numChannelsPerGroup` — Number of channels within each group for grouped transposed convolutions. `numChannelsPerGroup` must equal the number of channels in the input data divided by `numGroups`, the number of groups. For ungrouped convolutions, where `numGroups = 1`, `numChannelsPerGroup` must equal the number of channels in the input data.
- `numGroups` — Number of groups (optional). When `numGroups > 1`, the function performs grouped transposed convolutions. When `numGroups = 1`, the function performs ungrouped transposed convolutions; in this case, this dimension is singleton and can be omitted.

If `weights` is a formatted `dlarray`, it can have multiple spatial dimensions labeled `'S'`, one channel dimension labeled `'C'`, and up to two other dimensions labeled `'U'`. The number of `'S'` dimensions must match the number of `'S'` dimensions of the input data. The labeled dimensions correspond to the filter specifications as follows.

Filter Specification	Dimension Labels
`filterSize`	Up to three `'S'` dimensions
`numFiltersPerGroup`	`'C'` dimension
`numChannelsPerGroup`	First `'U'` dimension
`numGroups` (optional)	Second `'U'` dimension

Data Types: `single` | `double`

bias — Bias constant
dlarray vector | dlarray scalar | numeric vector | numeric scalar

Bias constant, specified as a formatted or unformatted dlarray vector or dlarray scalar, a numeric vector, or a numeric scalar.

- If bias is a scalar or has only singleton dimensions, the same bias is applied to each entry of the output.
- If bias has a nonsingleton dimension, each element of bias is the bias applied to the corresponding convolutional filter specified by weights. The number of elements of bias must match the number of filters specified by .

If bias is a formatted dlarray, the nonsingleton dimension must be a channel dimension labeled 'C'.

Data Types: single | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Stride',2 sets the stride of each filter to 2.

DataFormat — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar FMT that provides a label for each dimension of the data.

When you specify the format of a dlarray object, each character provides a label for each dimension of the data and must be one of the following:

- "S" — Spatial
- "C" — Channel
- "B" — Batch (for example, samples and observations)
- "T" — Time (for example, time steps of sequences)
- "U" — Unspecified

You can specify multiple dimensions labeled "S" or "U". You can use the labels "C", "B", and "T" at most once.

You must specify DataFormat when the input data is not a formatted dlarray.

Data Types: char | string

Stride — Step size for traversing input data
1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of `'Stride'` is `1`.

Example: `'Stride',3`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DilationFactor` — Filter dilation factor
1 (default) | numeric scalar | numeric vector

Filter dilation factor, specified as the comma-separated pair consisting of `'DilationFactor'` and one of the following.

- Numeric scalar — The same dilation factor value is applied for all spatial dimensions.
- Numeric vector — A different dilation factor value is applied along each spatial dimension. Use a vector of size `d`, where `d` is the number of spatial dimensions of the input data. The `i`th element of the vector specifies the dilation factor applied to the `i`th spatial dimension.

Use the dilation factor to increase the receptive field of the filter (the area of the input that the filter can see) on the input data. Using a dilation factor corresponds to an effective filter size of `filterSize + (filterSize-1)*(dilationFactor-1)`.

Example: `'DilationFactor',2`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Cropping` — Cropping applied to edges of data
0 (default) | `'same'` | numeric scalar | numeric vector | numeric matrix

Cropping applied to edges of data, specified as the comma-separated pair consisting of `'Cropping'` and one of the following.

- `'same'` — Cropping is set so that the output size is the same as the input size when the stride is `1`. More generally, the output size of each spatial dimension is `inputSize*stride`, where `inputSize` is the size of the input along a spatial dimension.
- Numeric scalar — The same cropping value is applied to both ends of all spatial dimensions.
- Numeric vector — A different cropping value is applied along each spatial dimension. Use a vector of size `d`, where `d` is the number of spatial dimensions of the input data. The `i`th element of the vector specifies the cropping applied to the start and the end along the `i`th spatial dimension.
- Numeric matrix — A different cropping value is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-`d`, where `d` is the number of spatial dimensions of the input data. The element `(1,d)` specifies the cropping applied to the start of spatial dimension `d`. The element `(2,d)` specifies the cropping applied to the end of spatial dimension `d`. For example, in 2-D the format is `[top, left; bottom, right]`.

Example: `'Cropping','same'`

Data Types: `single` | `double`

Output Arguments

`dlY` — Feature map
`dlarray`

Feature map, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, then `dlY` has the same format as `dlX`. If the input data is not a formatted `dlarray`, then `dlY` is an unformatted `dlarray` or numeric array with the same dimension order as the input data.

The size of the `'C'` channel dimension of `dlY` depends on the size of the `weights` input. The size of the `'C'` dimension of output `Y` is the product of the size of the dimensions `numFiltersPerGroup` and `numGroups` in the `weights` argument. If `weights` is a formatted `dlarray`, this product is the same as the product of the size of the `'C'` dimension and the second `'U'` dimension.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.
  - `dlX`
  - `weights`
  - `bias`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `avgpool` | `dlconv` | `maxunpool` | `maxpool` | `dlgradient` | `dlfeval`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

disconnectLayers

Disconnect layers in layer graph

Syntax

```
newlgraph = disconnectLayers(lgraph,s,d)
```

Description

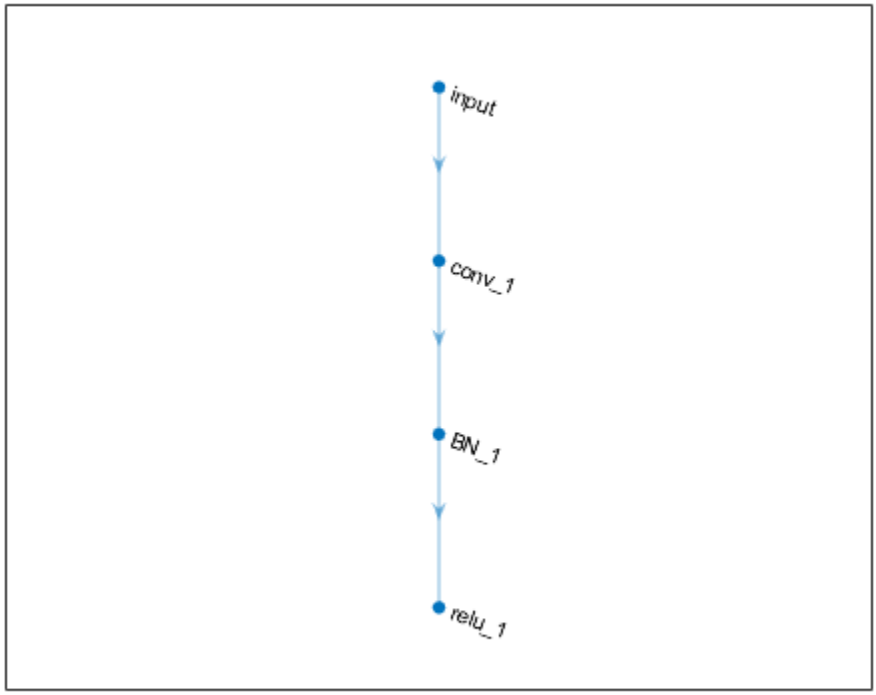
`newlgraph = disconnectLayers(lgraph,s,d)` disconnects the source layer `s` from the destination layer `d` in the layer graph `lgraph`. The new layer graph, `newlgraph`, contains the same layers as `lgraph`, but excludes the connection between `s` and `d`.

Examples

Disconnect Layers in Layer Graph

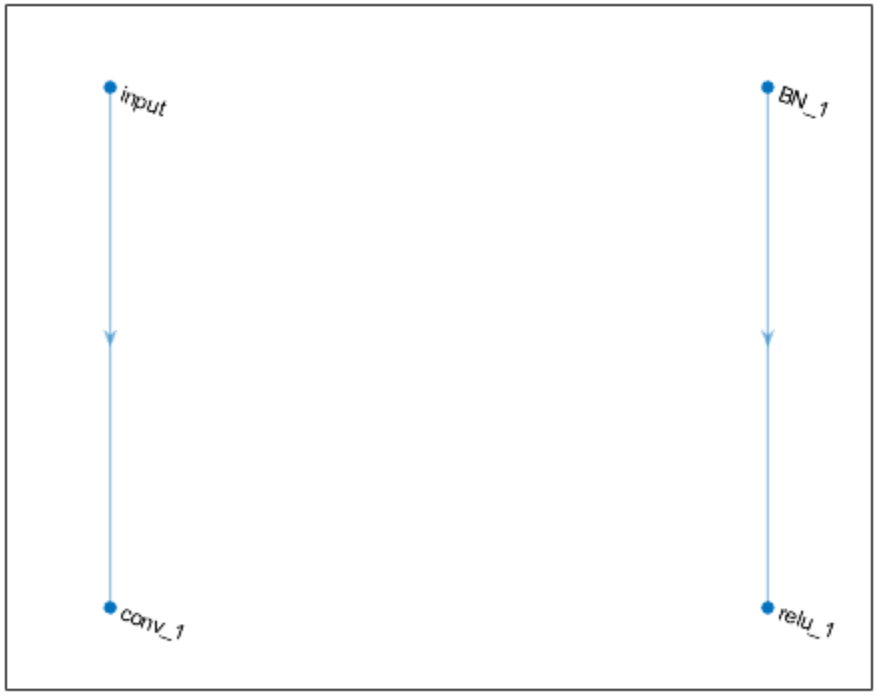
Create a layer graph from an array of layers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];

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



Disconnect the `'conv_1'` layer from the `'BN_1'` layer.

```
lgraph = disconnectLayers(lgraph,'conv_1','BN_1');
figure
plot(lgraph)
```



Input Arguments

`lgraph` — Layer graph

`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`s` — Connection source

character vector | string scalar

Connection source, specified as a character vector or a string scalar.

- If the source layer has a single output, then `s` is the name of the layer.
- If the source layer has multiple outputs, then `s` is the layer name followed by the character / and the name of the layer output: `'layerName/outputName'`.

Example: `'conv1'`

Example: `'mpool/indices'`

`d` — Connection destination

character vector | string scalar

Connection destination, specified as a character vector or a string scalar.

- If the destination layer has a single input, then `d` is the name of the layer.
- If the destination layer has multiple inputs, then `d` is the layer name followed by the character `/` and the name of the layer input: `'layerName/inputName'`.

Example: `'fc'`

Example: `'addlayer1/in2'`

Output Arguments

`newlgraph` — Output layer graph
`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `addLayers` | `removeLayers` | `replaceLayer` | `connectLayers` | `plot` | `assembleNetwork`

Topics
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b

dropoutLayer

Dropout layer

Description

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

Creation

Syntax

```
layer = dropoutLayer
layer = dropoutLayer(probability)
layer = dropoutLayer( ___ ,'Name',Name)
```

Description

`layer = dropoutLayer` creates a dropout layer.

`layer = dropoutLayer(probability)` creates a dropout layer and sets the `Probability` property.

`layer = dropoutLayer( ___ ,'Name',Name)` sets the optional `Name` property using a name-value pair and any of the arguments in the previous syntaxes. For example, `dropoutLayer(0.4,'Name','drop1')` creates a dropout layer with dropout probability 0.4 and name `'drop1'`. Enclose the property name in single quotes.

Properties

Dropout

`Probability` — Probability to drop out input elements
`0.5` (default) | nonnegative number less than 1

Probability for dropping out input elements, specified as a nonnegative number less than 1.

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 [1], [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.

For image input, the layer applies a different mask for each channel of each image. For sequence input, the layer applies a different dropout mask for each time step of each sequence.

Example: `0.4`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Dropout Layer

Create a dropout layer with name `'drop1'`.

```
layer = dropoutLayer('Name','drop1')
```

```
layer =
  DropoutLayer with properties:
```

```
           Name: 'drop1'

   Hyperparameters
    Probability: 0.5000
```

Include a dropout layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    dropoutLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Dropout                 50% dropout
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

More About

Dropout Layer

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 [1], [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.

For image input, the layer applies a different mask for each channel of each image. For sequence input, the layer applies a different dropout mask for each time step of each sequence.

References

[1] 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.

[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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`imageInputLayer` | `reluLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

efficientnetb0

EfficientNet-b0 convolutional neural network

Syntax

```
net = efficientnetb0
net = efficientnetb0('Weights','imagenet')

lgraph = efficientnetb0('Weights','none')
```

Description

EfficientNet-b0 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the EfficientNet-b0 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with EfficientNet-b0.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load EfficientNet-b0 instead of GoogLeNet.

`net = efficientnetb0` returns an EfficientNet-b0 model network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for EfficientNet-b0 Network* support package. If this support package is not installed, then the function provides a download link.

`net = efficientnetb0('Weights','imagenet')` returns a EfficientNet-b0 model network trained on the ImageNet data set. This syntax is equivalent to `net = efficientnetb0`.

`lgraph = efficientnetb0('Weights','none')` returns the untrained EfficientNet-b0 model network architecture. The untrained model does not require the support package.

Examples

Download EfficientNet-b0 Support Package

Download and install the Deep Learning Toolbox Model *for EfficientNet-b0 Network* support package.

Type `efficientnetb0` at the command line.

```
efficientnetb0
```

If the Deep Learning Toolbox Model *for EfficientNet-b0 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `efficientnetb0` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

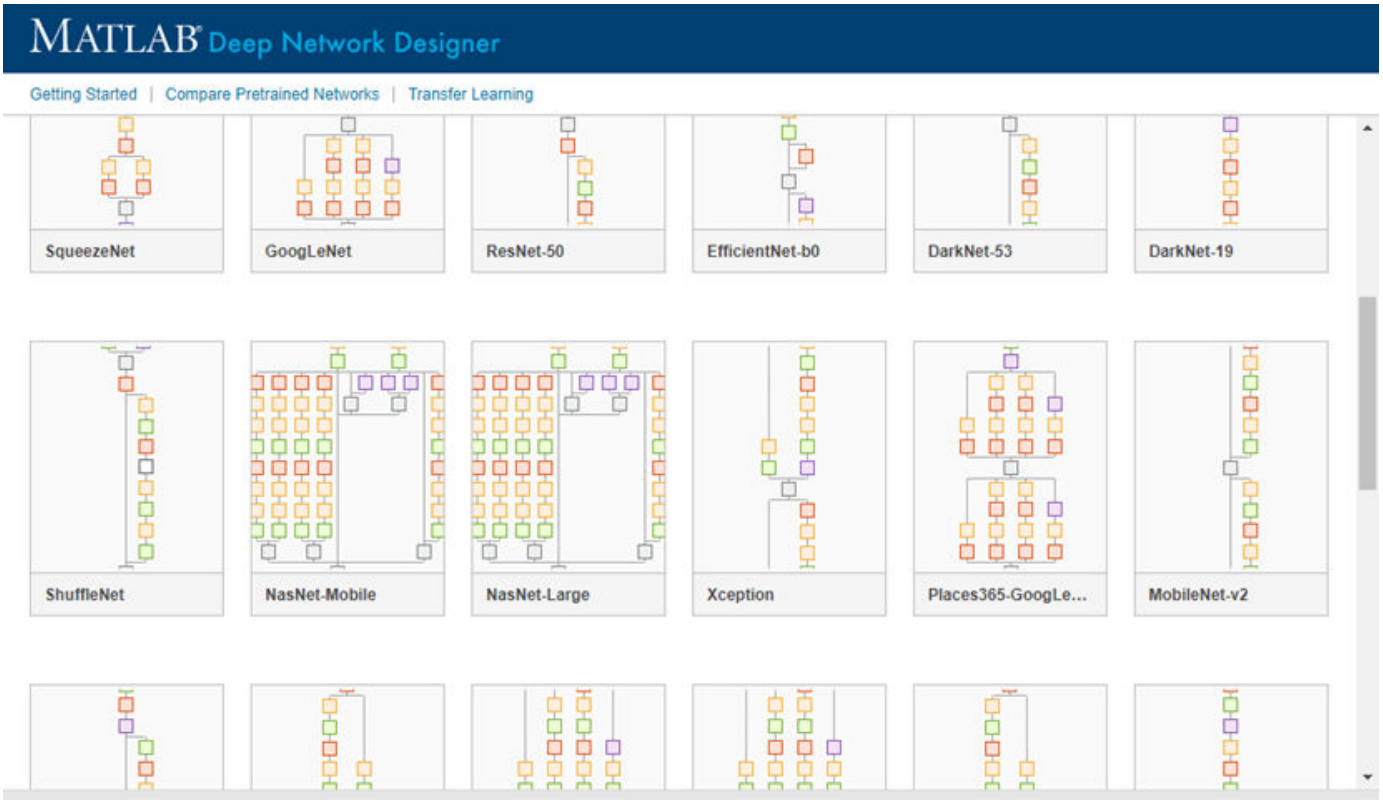
```
efficientnetb0
```

```
ans =

  DAGNetwork with properties:

         Layers: [290×1 nnet.cnn.layer.Layer]
    Connections: [363×2 table]
     InputNames: {'ImageInput'}
    OutputNames: {'classification'}
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(efficientnetb0)
```

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



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

Output Arguments

net — Pretrained EfficientNet-b0 convolutional neural network

`DAGNetwork` object

Pretrained EfficientNet-b0 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained EfficientNet-b0 convolutional neural network architecture
`LayerGraph` object

Untrained EfficientNet-b0 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Mingxing Tan and Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *ArXiv Preprint ArXiv:1905.1194*, 2019.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = efficientnetb0` or by passing the `efficientnetb0` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('efficientnetb0')`

For more information, see "Load Pretrained Networks for Code Generation" (MATLAB Coder).

The syntax `efficientnetb0('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = efficientnetb0` or by passing the `efficientnetb0` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('efficientnetb0')`

  For more information, see "Load Pretrained Networks for Code Generation" (GPU Coder).
- The syntax `efficientnetb0('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `squeezenet` | `vgg16` | `vgg19` | `resnet18` | `resnet50` | `googlenet` | `inceptionv3` | `inceptionresnetv2` | `densenet201` | `trainNetwork` | `layerGraph` | `DAGNetwork`

Topics

"Transfer Learning with Deep Network Designer"
"Deep Learning in MATLAB"
"Pretrained Deep Neural Networks"
"Classify Image Using GoogLeNet"
"Train Deep Learning Network to Classify New Images"
"Train Residual Network for Image Classification"

Introduced in R2020b

eluLayer

Exponential linear unit (ELU) layer

Description

An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.

The layer performs the following operation:

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(\exp(x) - 1), & x < 0 \end{cases}$$

The default value of α is 1. Specify a value of α for the layer by setting the `Alpha` property.

Creation

Syntax

```
layer = eluLayer
layer = eluLayer(alpha)
layer = eluLayer( ___ ,'Name',Name)
```

Description

`layer = eluLayer` creates an ELU layer.

`layer = eluLayer(alpha)` creates an ELU layer and specifies the `Alpha` property.

`layer = eluLayer( ___ ,'Name',Name)` additionally sets the optional `Name` property using any of the previous syntaxes. For example, `eluLayer('Name','elu1')` creates an ELU layer with the name `'elu1'`.

Properties

ELU

`Alpha` — Nonlinearity parameter

1 (default) | numeric scalar

Nonlinearity parameter α, specified as a numeric scalar. The minimum value of the output of the ELU layer equals $-\alpha$ and the slope at negative inputs approaching 0 is α.

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create ELU Layer

Create an exponential linear unit (ELU) layer with the name `'elu1'` and a default value of 1 for the nonlinearity parameter `Alpha`.

```
layer = eluLayer('Name','elu1')
```

```
layer =
  ELULayer with properties:

     Name: 'elu1'
    Alpha: 1
```

```
   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Include an ELU layer in a `Layer` array.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    eluLayer

    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    eluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  11x1 Layer array with layers:

     1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [0  0  0
     3   ''   Batch Normalization     Batch normalization
     4   ''   ELU                     ELU with Alpha 1
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [0  0  0
     7   ''   Batch Normalization     Batch normalization
     8   ''   ELU                     ELU with Alpha 1
     9   ''   Fully Connected         10 fully connected layer
    10   ''   Softmax                 softmax
    11   ''   Classification Output   crossentropyex
```

References

[1] Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (ELUs)." *arXiv preprint arXiv:1511.07289* (2015).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `batchNormalizationLayer` | `leakyReluLayer` | `clippedReluLayer` | `reluLayer` | `swishLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2019a

embed

Embed discrete data

Syntax

```
dlY = embed(dlX,weights)
dlY = embed(dlX,weights,'DataFormat',FMT)
```

Description

The embed operation converts numeric indices to numeric vectors, where the indices correspond to discrete data. Use embeddings to map discrete data such as categorical values or words to numeric vectors.

Note This function applies the embed operation to `dlarray` data. If you want to apply the embed operation within a `layerGraph` object or `Layer` array, use a `wordEmbeddingLayer` object.

`dlY = embed(dlX,weights)` returns the embedding vectors in `weights` corresponding to the numeric indices in the formatted `dlarray` object `dlX`.

`dlY = embed(dlX,weights,'DataFormat',FMT)` also specifies dimension format `FMT` when `dlX` is not a formatted `dlarray` object. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`.

Examples

Embed Categorical Data

Embed a mini-batch of categorical features.

Create an array of categorical features containing 5 observations with values `"Male"` or `"Female"`.

```
X = categorical(["Male" "Female" "Male" "Female" "Female"])';
```

Initialize the embedding weights. Specify an embedding dimension of 10, and a vocabulary corresponding to the number of categories of the input data plus one.

```
embeddingDimension = 10;
vocabularySize = numel(categories(X));
weights = rand(embeddingDimension,vocabularySize+1);
```

To embed the categorical data, first convert it to mini-batch of numeric indices.

```
X = double(X)
```

```
X = 5×1

     2
     1
```

```
     2
     1
     1
```

For formatted `dlarray` input, the embed function expands into a singleton `'C'` (channel) dimension with size 1. Create a formatted `dlarray` object containing the data. To specify that the rows correspond to observations, specify the format `'BC'` (batch, channel).

```
dlX = dlarray(X,'BC')
```

```
dlX =
  1(C) x 5(B) dlarray

     2     1     2     1     1
```

Embed the numeric indices using the `embed` function. The embed function expands into the `'C'` dimension.

```
dlY = embed(dlX,weights)
```

```
dlY =
  10(C) x 5(B) dlarray

    0.1576    0.8147    0.1576    0.8147    0.8147
    0.9706    0.9058    0.9706    0.9058    0.9058
    0.9572    0.1270    0.9572    0.1270    0.1270
    0.4854    0.9134    0.4854    0.9134    0.9134
    0.8003    0.6324    0.8003    0.6324    0.6324
    0.1419    0.0975    0.1419    0.0975    0.0975
    0.4218    0.2785    0.4218    0.2785    0.2785
    0.9157    0.5469    0.9157    0.5469    0.5469
    0.7922    0.9575    0.7922    0.9575    0.9575
    0.9595    0.9649    0.9595    0.9649    0.9649
```

In this case, the output is an `embeddingDimension`-by-`N` matrix with format `'CB'` (channel, batch), where `N` is the number of observations. Each column contains the embedding vectors.

Embed Text Data

Embed a mini-batch of text data.

```
textData = [
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."];
```

Create an array of tokenized documents.

```
documents = tokenizedDocument(textData);
```

To encode text data as sequences of numeric indices, create a `wordEncoding` object.

```
enc = wordEncoding(documents);
```

Initialize the embedding weights. Specify an embedding dimension of 100, and a vocabulary size to be consistent with the vocabulary size corresponding to the number of words in the word encoding plus one.

```
embeddingDimension = 100;
vocabularySize = enc.NumWords;
weights = rand(embeddingDimension,vocabularySize+1);
```

Convert the tokenized documents to sequences of word vectors using the `doc2sequence` function. The `doc2sequence` function, by default, discards out-of-vocabulary tokens in the input data. To map out-of-vocabulary tokens to the last vector of embedding weights, set the `'UnknownWord'` option to `'nan'`. The `doc2sequence` function, by default, left-pads the input sequences with zeros to have the same length

```
sequences = doc2sequence(enc,documents,'UnknownWord','nan')
```

```
sequences=2×1 cell array
    {[         0 1 2 3 4 5 6 7 8 9 10]}
    {[11 12 13 14 15 2 16 17 18 19 10]}
```

The output is a cell array, where each element corresponds to an observation. Each element is a row vector with elements representing the individual tokens in the corresponding observation including the padding values.

Convert the cell array to a numeric array by vertically concatenating the rows.

```
X = cat(1,sequences{:})
```

```
X = 2×11

     0     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15     2    16    17    18    19    10
```

Convert the numeric indices to `dlarray`. Because the rows and columns of `X` correspond to observations and time steps, respectively, specify the format `'BT'`.

```
dlX = dlarray(X,'BT')
```

```
dlX =
  2(B) x 11(T) dlarray

     0     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15     2    16    17    18    19    10
```

Embed the numeric indices using the `embed` function. The `embed` function maps the padding tokens (tokens with index 0) and any other out-of-vocabulary tokens to the same out-of-vocabulary embedding vector.

```
dlY = embed(dlX,weights);
```

In this case, the output is an `embeddingDimension`-by-`N`-by-`S` matrix with format `'CBT'`, where `N` and `S` are the number of observations and the number of time steps, respectively. The vector `dlY(:,n,t)` corresponds to the embedding vector of time-step `t` of observation `n`.

Input Arguments

`dlX` — Input data

`dlarray` object | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. The elements of `dlX` must be nonnegative integers or `NaN`.

The function returns the embedding vectors in `weights` corresponding to the numeric indices in `dlX`. If any values in `dlX` are zero, `NaN`, or greater than the vocabulary size, then the function returns the out-of-vocabulary vector for that element.

When `dlX` is not a formatted `dlarray` object, you must specify the dimension label format using the `'DataFormat'` option. Also, if `dlX` is a numeric array, then `weights` must be a `dlarray` object.

The embed operation expands into a singleton channel dimension of the input data specified by the `'C'` dimension label. If the data has no specified channel dimension, then the function assumes an unspecified singleton channel dimension.

`weights` — Embedding weights

`dlarray` object | numeric array

Embedding weights, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

The matrix `weights` specifies the dimension of the embedding, the vocabulary size, and the embedding vectors.

The embedding dimension is the number of components `K` of the embedding. That is, the embedding maps numeric indices to vectors of length `K`. The vocabulary size is the number of discrete elements `V` in the embedding. That is, the number of discrete elements of the underlying data that the embedding supports. The embedding maps out-of-vocabulary indices to the same out-of-vocabulary embedding vector.

If `weights` is a formatted `dlarray` object, then it must have format `'CU'` or `'UC'`. The dimensions corresponding to the labels `'C'` and `'U'` must have size `K` and `V+1`, respectively, where `K` and `V` represent the embedding dimension and the vocabulary size, respectively. The extra vector corresponds to the out-of-vocabulary embedding vector.

If `weights` is not a formatted `dlarray` object, then `weights` must be a `K`-by-`(V+1)` matrix, where `K` and `V` represent the embedding dimension and vocabulary size, respectively.

The function returns the embedding vectors in `weights` corresponding to the numeric indices in `dlX`. If any values in `dlX` are zero, `NaN`, or greater than the vocabulary size, then the function returns the out-of-vocabulary vector for that element.

`FMT` — Dimension order of unformatted data

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel

- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`dlY` — Embedding vectors

`dlarray`

Embedding vectors, returned as a `dlarray` object. The output `dlY` has the same underlying data type as the input `dlX`.

The function returns the embedding vectors in `weights` corresponding to the numeric indices in `dlX`. If any values in `dlX` are zero, `NaN`, or greater than the vocabulary size, then the function returns the out-of-vocabulary vector for that element.

The embedding vectors have `K` elements, where `K` is the embedding dimension. The size of dimensions `dlY` depend on the input data:

- If `dlX` is a formatted `dlarray` with a `'C'` dimension label, then the embed operation expands into that dimension. That is, the output has the same dimension format as the input, the `'C'` dimension has size `K`, the other dimensions have the same size as the corresponding dimensions of the input.
- If `dlX` is a formatted `dlarray` without a `'C'` dimension. Then the operation assumes a singleton channel dimension. The output has a `'C'` dimension and all other dimensions have the same size and dimension labels. That is, the output has the same format as the input and also a `'C'` dimension, the `'C'` dimension has size `K`, the other dimensions have the same size as the corresponding dimensions of the input.
- If `dlX` is not a formatted `dlarray` object and `'DataFormat'` contains a `'C'` dimension, then the embed operation expands into that dimension. That is, the output has the number of dimensions as the input, the dimension corresponding to the `'C'` dimension has size `K`, the other dimensions have the same size as the corresponding dimensions of the input.
- If `dlX` is not a formatted `dlarray` object and `'DataFormat'` does not contain a 'C' dimension, then the embed operation inserts a new dimension at the beginning. That is, the output has one more dimension as the input, the first dimension corresponding to the `'C'` dimension has size `K`, the other dimensions have the same size as the corresponding dimensions of the input.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.
  - `dlX`
  - `weights`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlfeval` | `dlgradient` | `lstm`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Sequence-to-Sequence Translation Using Attention”
“List of Functions with dlarray Support”

Introduced in R2020b

experiments.Monitor

Update results table and training plots for custom training experiments

Description

When running a custom training experiment in **Experiment Manager**, use an `experiments.Monitor` object to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots. For more information on custom training experiments, see “Configure Custom Training Experiment” on page 1-45.

Creation

When you run a custom training experiment, Experiment Manager creates an `experiments.Monitor` object for each trial of your experiment. Access the object as the second input argument of the training function.

Properties

`Status` — Training status

`""` (default) | string | character vector

Training status for a trial, specified as a string or character vector.

Example: `monitor.Status = "Loading Data";`

Data Types: `char` | `string`

`Progress` — Training progress

`0` (default) | numeric scalar

Training progress for a trial, specified as a numeric scalar between 0 and 100.

Example: `monitor.Progress = 17;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`Info` — Information column names

`""` (default) | string | character vector | string array | cell array of character vectors

Information column names, specified as a string, character vector, string array, or cell array of character vectors. Valid names begin with a letter, and can contain letters, digits, and underscores. These names appear as column headers in the experiment results table. The values in the information columns do not appear in the training plot.

You can set this value only once in your training function.

Example: `monitor.Info = ["GradientDecayFactor","SquaredGradientDecayFactor"];`

Data Types: `char` | `string`

`Metrics` — Metric column names

`""` (default) | string | character vector | string array | cell array of character vectors

Metric column names, specified as a string, character vector, string array, or cell array of character vectors. Valid names begin with a letter, and can contain letters, digits, and underscores. These names appear as column headers in the experiment results table. Additionally, each metric appears in its own training subplot. To plot more than one metric in a single subplot, use the function `groupSubPlot`.

You can set this value only once in your training function.

Example: `monitor.Metrics = ["TrainingLoss","ValidationLoss"];`

Data Types: `char` | `string`

`XLabel` — Horizontal axis label

`""` (default) | string | character vector

Horizontal axis label in the training plot, specified as a string or character vector.

Set this value before calling the function `recordMetrics`.

Example: `monitor.XLabel = "Iteration";`

Data Types: `char` | `string`

`Stop` — Flag to stop trial

`false` or 0 (default) | `true` or 1

This property is read-only.

Flag to stop trial, specified as a numeric or logical 1 (`true`) or 0 (`false`). The value of this property changes to `true` when you click **Stop** in the Experiment Manager toolstrip or the results table.

Data Types: `logical`

Object Functions

groupSubPlot	Group metrics in experiment training plot
recordMetrics	Record metric values in experiment results table and training plot
updateInfo	Update information columns in experiment results table

Examples

Track Progress, Display Information and Record Metric Values, and Produce Training Plots

Use an `experiments.Monitor` object to track the progress of the training, display information and metric values in the experiment results table, and produce training plots for custom training experiments.

Before starting the training, specify the names of the information and metric columns of the Experiment Manager results table.

```
monitor.Info = ["GradientDecayFactor","SquaredGradientDecayFactor"];
monitor.Metrics = ["TrainingLoss","ValidationLoss"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot.

```
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
```

Update the values of the gradient decay factor and the squared gradient decay factor for the trial in the results table.

```
updateInfo(monitor, ...
    GradientDecayFactor=gradientDecayFactor, ...
    SquaredGradientDecayFactor=squaredGradientDecayFactor);
```

After each iteration of the custom training loop, record the value of training and validation loss for the trial in the results table and the training plot.

```
recordMetrics(monitor,iteration, ...
    TrainingLoss=trainingLoss, ...
    ValidationLoss=validationLoss);
```

Update the training progress for the trial based on the fraction of iterations completed.

```
monitor.Progress = (iteration/numIterations) * 100;
```

Tips

- Both information and metric columns display values in the results table for your experiment. Additionally, the training plot shows a record of the metric values. Use information columns for text and for numerical values that you want to display in the results table but not in the training plot.

See Also

Experiment Manager

Introduced in R2021a

exportONNXNetwork

Export network to ONNX model format

Syntax

```
exportONNXNetwork(net,filename)
exportONNXNetwork(net,filename,Name,Value)
```

Description

`exportONNXNetwork(net,filename)` exports the deep learning network `net` with weights to the ONNX format file `filename`. If `filename` exists, then `exportONNXNetwork` overwrites the file.

This function requires 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.

`exportONNXNetwork(net,filename,Name,Value)` exports a network using additional options specified by one or more name-value pair arguments.

Examples

Export Network in ONNX Format

Load a pretrained SqueezeNet convolutional neural network.

```
net = squeezenet
```

```
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

Export the network as an ONNX format file in the current folder called `squeezenet.onnx`. If the Deep Learning Toolbox Converter for ONNX Model Format support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

```
filename = 'squeezenet.onnx';
exportONNXNetwork(net,filename)
```

Now, you can import the `squeezenet.onnx` file into any deep learning framework that supports ONNX import.

Export Layer Graph to ONNX Format

Export a layer graph with or without an output layer to the ONNX format by using `exportONNXNetwork`.

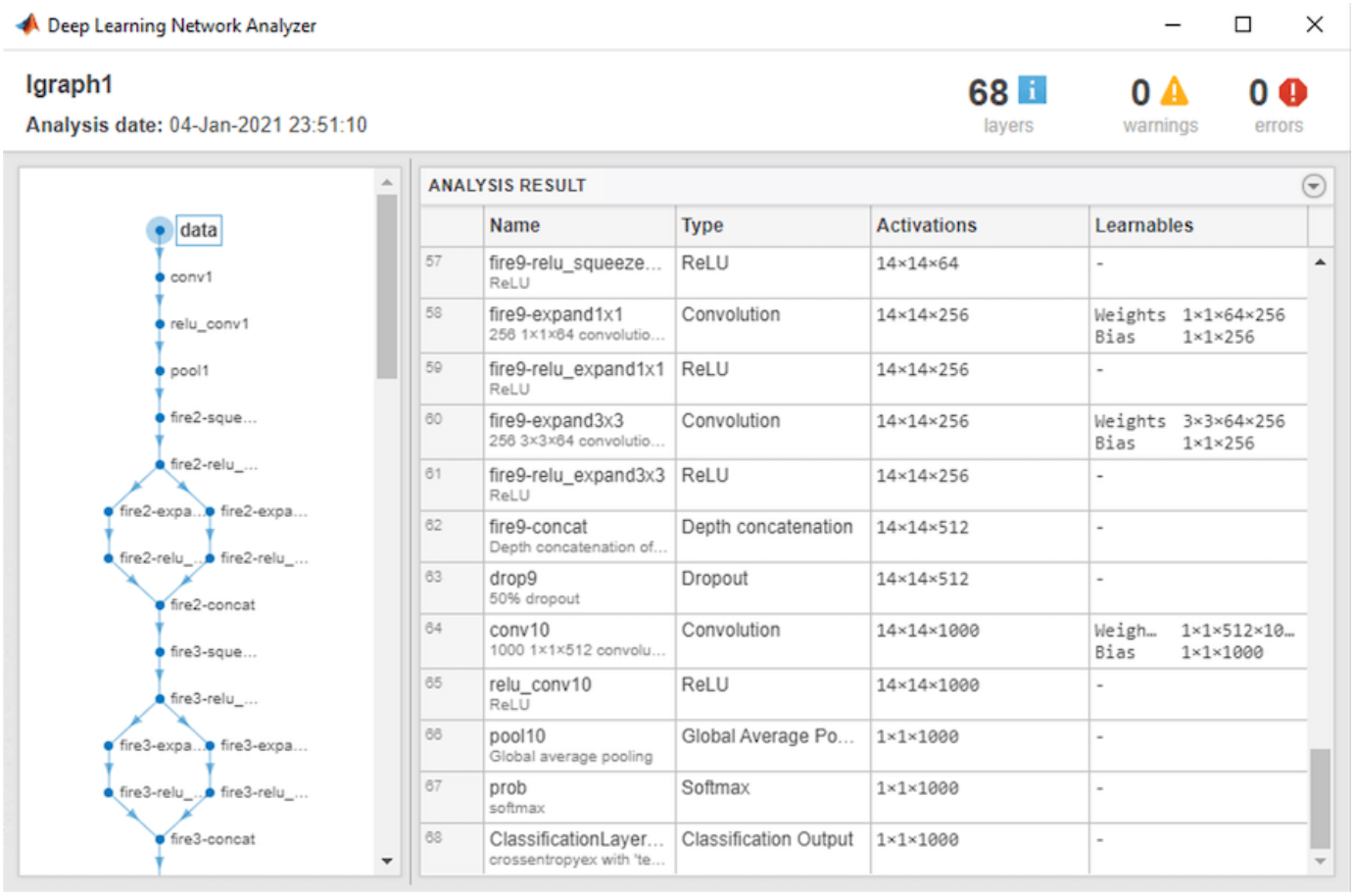
Load a pretrained SqueezeNet convolutional neural network, and convert the pretrained network to a layer graph.

```
net = squeezenet;
lgraph1 = layerGraph(net)
```

```
lgraph1 =
  LayerGraph with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

Analyze the layer graph. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers. You can also detect errors and issues in the layer graph `lgraph1` before exporting to the ONNX format. `lgraph1` is error free.

```
analyzeNetwork(lgraph1)
```



Export the layer graph `lgraph1` as an ONNX format file in the current folder called `squeezeLayers1.onnx`.

```
exportONNXNetwork(lgraph1,'squeezeLayers1.onnx')
```

Now, you can import the `squeezeLayers1.onnx` file into any deep learning framework that supports ONNX import.

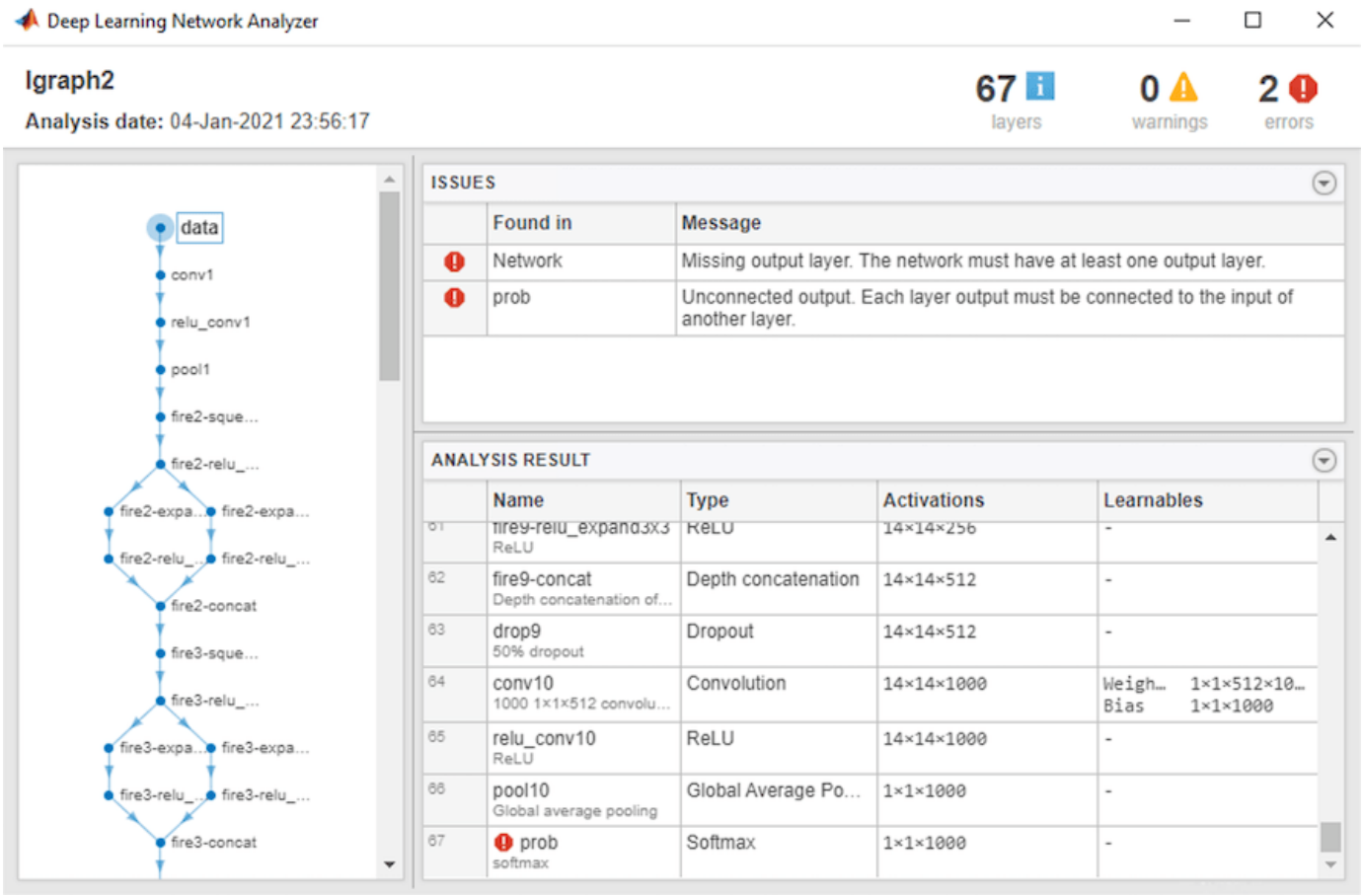
Remove the output layer of `lgraph1`.

```
lgraph2 = removeLayers(lgraph1,lgraph1.Layers(end).Name)
```

```
lgraph2 =
  LayerGraph with properties:

         Layers: [67×1 nnet.cnn.layer.Layer]
    Connections: [74×2 table]
     InputNames: {'data'}
    OutputNames: {1×0 cell}
```

Analyze the layer graph `lgraph2` by using `analyzeNetwork`. The layer graph analysis detects a missing output layer and an unconnected output. You can still export `lgraph2` to the ONNX format.

```
analyzeNetwork(lgraph2)
```



Export the layer graph `lgraph2` as an ONNX format file in the current folder called `squeezeLayers2.onnx`.

```
exportONNXNetwork(lgraph2,'squeezeLayers2.onnx')
```

Now, you can import the `squeezeLayers2.onnx` file into any deep learning framework that supports ONNX import.

Input Arguments

`net` — Trained network or graph of network layers

`SeriesNetwork` object | `DAGNetwork` object | `dlnetwork` object | `LayerGraph` object

Trained network or graph of network layers, specified as a `SeriesNetwork`, `DAGNetwork`, `dlnetwork`, or `LayerGraph` object.

You can get a trained network (`SeriesNetwork`, `DAGNetwork`, or `dlnetwork`) in these ways:

- Import a pretrained network. For example, use the `googlenet` function.
- Train your own network. Use `trainNetwork` to train a `SeriesNetwork` or `DAGNetwork`. Use a custom training loop to train a `dlnetwork`.

A `LayerGraph` object is a graph of network layers. Some of the layer parameters of this graph might be empty (for example, the weights and bias of convolution layers, and the mean and variance of batch normalization layers). Before using the layer graph as an input argument to `exportONNXNetwork`, initialize the empty parameters by assigning random values. Alternatively, you can do one of the following before exporting:

- Convert a `LayerGraph` object to a `dlnetwork` object by using the layer graph as an input argument to `dlnetwork`. The empty parameters are automatically initialized.
- Convert a `LayerGraph` object to a trained `DAGNetwork` object by using `trainNetwork`. Use the layer graph as the `layers` input argument to `trainNetwork`.

You can detect errors and issues in a trained network or graph of network layers before exporting to an ONNX network by using `analyzeNetwork`. `exportONNXNetwork` requires `SeriesNetwork`, `DAGNetwork`, and `dlnetwork` objects to be error free. `exportONNXNetwork` permits exporting a `LayerGraph` object with a missing or unconnected output layer.

`filename` — Name of file

character vector | string scalar

Name of file, specified as a character vector or string scalar.

Example: `'network.onnx'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `exportONNXNetwork(net,filename,'NetworkName','my_net')` exports a network and specifies `'my_net'` as the network name in the saved ONNX network.

`NetworkName` — Name of ONNX network
`'Network'` (default) | character vector | string scalar

Name of ONNX network to store in the saved file, specified as a character vector or a string scalar.

Example: `'my_squeezenet'`

`OpsetVersion` — Version of ONNX operator set
8 (default) | 6 | 7 | 9

Version of ONNX operator set to use in the exported model. If the default operator set does not support the network you are trying to export, then try using a later version. If you import the exported network to another framework and you used an operator set during export that the importer does not support, then the import can fail.

To ensure that you use the appropriate operator set version, consult the ONNX operator documentation [3]. For example, `'OpsetVersion',9` exports the `maxUnpooling2dLayer` to the `MaxUnpool-9` ONNX operator.

Example: 6

Limitations

- `exportONNXNetwork` supports ONNX versions as follows:
  - The function supports ONNX intermediate representation version 6.
  - The function supports ONNX operator sets 6, 7, 8, and 9.
- `exportONNXNetwork` does not export settings or properties related to network training such as training options, learning rate factors, or regularization factors.
- If you export a network containing a layer that the ONNX format does not support (see “Layers Supported for ONNX Export” on page 1-566), then `exportONNXNetwork` saves a placeholder ONNX operator in place of the unsupported layer and returns a warning. You cannot import an ONNX network with a placeholder operator into other deep learning frameworks.
- Because of architectural differences between MATLAB and ONNX, an exported network can have a different structure compared to the original network.

Note If you import an exported network, layers of the reimported network might differ from the original network and might not be supported.

More About

Layers Supported for ONNX Export

`exportONNXNetwork` can export the following:

- Networks that have both convolutional and LSTM layers, such as those for video classification applications.
- All custom layers (except `nnet.onnx.layer.Flatten3dLayer`) that are created when you import networks from ONNX or TensorFlow-Keras using either Deep Learning Toolbox Converter for ONNX Model Format or Deep Learning Toolbox Converter for TensorFlow Models.

- The layers listed in the following table:

ONNX Exporter Supported Layers
Deep Learning Toolbox Layers
additionLayer
averagePooling1dLayer
averagePooling2dLayer
averagePooling3dLayer
batchNormalizationLayer
bilstmLayer
ClassificationOutputLayer
clippedReluLayer
concatenationLayer
convolution1dLayer
convolution2dLayer
convolution3dLayer
crop2dLayer
CrossChannelNormalizationLayer
depthConcatenationLayer
dropoutLayer
eluLayer
featureInputLayer
flattenLayer
fullyConnectedLayer
globalAveragePooling1dLayer
globalAveragePooling2dLayer
globalMaxPooling1dLayer
globalMaxPooling2dLayer
groupedConvolution2dLayer
groupNormalizationLayer
gruLayer
imageInputLayer
image3dInputLayer
leakyReluLayer
lstmLayer
maxPooling1dLayer
maxPooling2dLayer
maxPooling3dLayer

ONNX Exporter Supported Layers
maxUnpooling2dLayer
multiplicationLayer
RegressionOutputLayer
reluLayer
sequenceInputLayer
sigmoidLayer
softmaxLayer
swishLayer
tanhLayer
transposedConv2dLayer
transposedConv3dLayer
ONNX Importer Custom Layers
nnet.onnx.layer.ClipLayer
nnet.onnx.layer.ElementwiseAffineLayer
nnet.onnx.layer.FlattenLayer
nnet.onnx.layer.GlobalAveragePooling2dLayer
nnet.onnx.layer.IdentityLayer
nnet.onnx.layer.PReluLayer
nnet.onnx.layer.TanhLayer
Keras Importer Custom Layers
nnet.keras.layer.FlattenCStyleLayer
nnet.keras.layer.GlobalAveragePooling2dLayer
nnet.keras.layer.TanhLayer
nnet.keras.layer.ZeroPadding2dLayer
Caffe Importer Custom Layers
nnet.caffe.layer.TanhLayer
Computer Vision Toolbox™ Layers
pixelClassificationLayer
rcnnBoxRegressionLayer
roiInputLayer
roiMaxPooling2dLayer

ONNX Exporter Supported Layers
Image Processing Toolbox™ Layers
`depthToSpace2dLayer`
`resize2dLayer`
`resize3dLayer`
`spaceToDepthLayer`
Text Analytics Toolbox™ Layers
`wordEmbeddingLayer`

For the `groupNormalizationLayer`, specify `numGroups` as `"channel-wise"` to map the exported layer to the ONNX `InstanceNormalization` operator. `GroupNormalization` is not a standard ONNX operator [3].

Tips

- You can export a trained MATLAB deep learning network that includes multiple inputs and multiple outputs to the ONNX model format. To learn about a multiple-input and multiple-output deep learning network, see “Multiple-Input and Multiple-Output Networks”.

References

[1] *Open Neural Network Exchange*. https://github.com/onnx/.

[2] *ONNX*. https://onnx.ai/.

[3] *ONNX Operators*. https://github.com/onnx/onnx/blob/master/docs/Operators.md.

See Also

`importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `importONNXLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importTensorFlowLayers` | `layerGraph`

Topics
“Pretrained Deep Neural Networks”
“Deep Learning in MATLAB”

Introduced in R2018a

extractdata

Extract data from `dlarray`

Syntax

```
y = extractdata(dlX)
```

Description

`y = extractdata(dlX)` returns the data in the `dlarray` `dlX`. The output `y` has the same data type as the underlying data in `dlX` and is unformatted.

Examples

Extract Data from `dlarray`

Create a logical `dlarray` with data format `'SS'`.

```
rng default % For reproducibility
dlX = dlarray(rand(4,3) > 0.5,'SS')
```

```
dlX =
  4(S) x 3(S) logical dlarray

   1   1   1
   1   0   1
   0   0   0
   1   1   1
```

Extract the data from `dlX`.

```
y = extractdata(dlX)
```

```
y = 4x3 logical array

   1   1   1
   1   0   1
   0   0   0
   1   1   1
```

Input Arguments

`dlX` — Input `dlarray`

`dlarray` object

Input `dlarray`, specified as a `dlarray` object.

Example: `dlX = dlarray(randn(50,3),'SC')`

Output Arguments

y — Data array

single array | double array | logical array | `gpuArray`

Data array, returned as a single, double, or logical array, or as a `gpuArray` of one of these array types. The output `y` has the same data type as the underlying data type in `dlX`. The output `y` is unformatted.

Tips

- If `dlX` contains an implicit permutation because of formatting, `y` has that permutation explicitly.
- The output `y` has no tracing for the computation of derivatives. See “Derivative Trace”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation does not support `gpuArray` data type.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For recommendations and limitations on `gpuArray`, see “Support for GPU Arrays” (GPU Coder).

See Also

`dlarray` | `gather`

Introduced in R2019b

featureInputLayer

Feature input layer

Description

A feature input layer inputs feature data to a network and applies data normalization. Use this layer when you have a data set of numeric scalars representing features (data without spatial or time dimensions).

For image input, use `imageInputLayer`.

Creation

Syntax

```
layer = featureInputLayer(numFeatures)
layer = featureInputLayer(numFeatures,Name,Value)
```

Description

`layer = featureInputLayer(numFeatures)` returns a feature input layer and sets the `InputSize` property to the specified number of features.

`layer = featureInputLayer(numFeatures,Name,Value)` sets the optional properties using name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in single quotes.

Properties

Feature Input

`InputSize` — Number of features
positive integer

Number of features for each observation in the data, specified as a positive integer.

For image input, use `imageInputLayer`.

Example: `10`

`Normalization` — Data normalization
`'none'` (default) | `'zerocenter'` | `'zscore'` | `'rescale-symmetric'` | `'rescale-zero-one'` | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by `Mean`.

- `'zscore'` — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form `Y = func(X)`, where `X` is the input data and the output `Y` is the normalized data.

Tip The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

`NormalizationDimension` — Normalization dimension

`'auto'` (default) | `'channel'` | `'all'`

Normalization dimension, specified as one of the following:

- `'auto'` – If the training option is `false` and you specify any of the normalization statistics (`Mean`, `StandardDeviation`, `Min`, or `Max`), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- `'channel'` – Channel-wise normalization.
- `'all'` – Normalize all values using scalar statistics.

`Mean` — Mean for zero-center and z-score normalization

`[]` (default) | column vector | numeric scalar

Mean for zero-center and z-score normalization, specified as a `numFeatures`-by-1 vector of means per feature, a numeric scalar, or `[]`.

If you specify the `Mean` property, then `Normalization` must be `'zerocenter'` or `'zscore'`. If `Mean` is `[]`, then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StandardDeviation` — Standard deviation for z-score normalization

`[]` (default) | column vector | numeric scalar

Standard deviation for z-score normalization, specified as a `numFeatures`-by-1 vector of means per feature, a numeric scalar, or `[]`.

If you specify the `StandardDeviation` property, then `Normalization` must be `'zscore'`. If `StandardDeviation` is `[]`, then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Min` — Minimum value for rescaling
`[]` (default) | column vector | numeric scalar

Minimum value for rescaling, specified as a `numFeatures`-by-1 vector of minima per feature, a numeric scalar, or `[]`.

If you specify the `Min` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Min` is `[]`, then the software calculates the minimum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Max` — Maximum value for rescaling
`[]` (default) | column vector | numeric scalar

Maximum value for rescaling, specified as a `numFeatures`-by-1 vector of maxima per feature, a numeric scalar, or `[]`.

If you specify the `Max` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Max` is `[]`, then the software calculates the maximum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

`InputNames` — Input names
`{}` (default)

Input names of the layer. The layer has no inputs.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Feature Input Layer

Create a feature input layer with the name `'input'` for observations consisting of 21 features.

```
layer = featureInputLayer(21,'Name','input')
```

```
layer =
  FeatureInputLayer with properties:

                     Name: 'input'
                InputSize: 21

   Hyperparameters
             Normalization: 'none'
    NormalizationDimension: 'auto'
```

Include a feature input layer in a `Layer` array.

```
numFeatures = 21;
numClasses = 3;

layers = [
    featureInputLayer(numFeatures,'Name','input')
    fullyConnectedLayer(numClasses, 'Name','fc')
    softmaxLayer('Name','sm')
    classificationLayer('Name','classification')]
```

```
layers =
  4x1 Layer array with layers:

     1   'input'            Feature Input           21 features
     2   'fc'               Fully Connected         3 fully connected layer
     3   'sm'               Softmax                 softmax
     4   'classification'   Classification Output   crossentropyex
```

Combine Image and Feature Input Layers

To train a network containing both an image input layer and a feature input layer, you must use a `dlnetwork` object in a custom training loop.

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

```
imageInputSize = [28 28 1];
numFeatures = 1;
numClasses = 10;
filterSize = 5;
numFilters = 16;
```

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

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

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

Convert the layers to a layer graph.

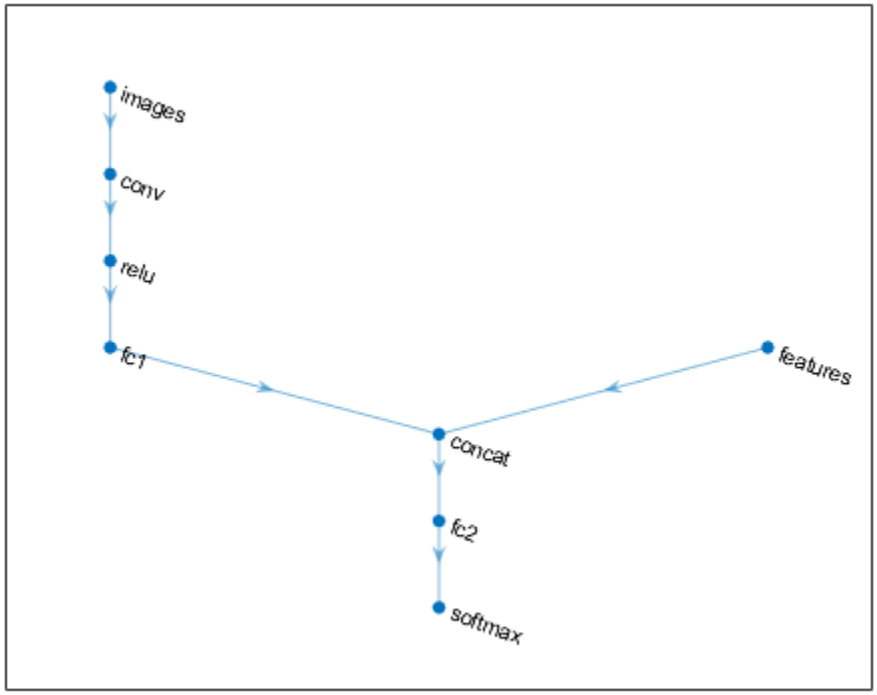
```
lgraph = layerGraph(layers);
```

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

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

Visualize the network.

```
plot(lgraph)
```



Create a `dlnetwork` object.

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

```
dlnet =
  dlnetwork with properties:

         Layers: [8x1 nnet.cnn.layer.Layer]
    Connections: [7x2 table]
     Learnables: [6x3 table]
          State: [0x3 table]
     InputNames: {'images'  'features'}
    OutputNames: {'softmax'}
    Initialized: 1
```

Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

To generate CUDA or C++ code by using GPU Coder, you must first construct and train a deep neural network. Once the network is trained and evaluated, you can configure the code generator to generate code and deploy the convolutional neural network on platforms that use NVIDIA® or ARM GPU processors. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

See Also

`trainNetwork` | `fullyConnectedLayer` | `image3dInputLayer` | **Deep Network Designer** | `imageInputLayer` | `sequenceInputLayer` | `dlnetwork`

Topics

“Train Network with Numeric Features”
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2020b

finddim

Find dimensions with specified label

Syntax

```
dim = finddim(dlX,label)
```

Description

`dim = finddim(dlX,label)` returns the dimensions in `dlX` that have the dimension label `label`. If the data format of `dlX` does not contain the dimension label `label`, `dim` is empty.

Examples

Obtain Dimension with Specified Labels

Create a formatted `dlarray` with some repeated dimension labels. Specify the dimension labels as `'TSSU'`. The `dlarray` call reorders the labels, because it enforces the order `'SCBTU'`. For more information about dimension labels, see “Usage” on page 1-420.

```
dlX = dlarray(randn(5,4,3,2),'TSSU');
```

Obtain the dimensions with the label `'T'`.

```
dimU = finddim(dlX,'T')
```

```
dimU = 3
```

Obtain the dimensions with the label `'S'`.

```
dimS = finddim(dlX,'S')
```

```
dimS = 1×2

     1     2
```

Obtain the dimensions with the label `'B'`.

```
dimB = finddim(dlX,'B')
```

```
dimB =

  1x0 empty double row vector
```

Obtain the size of the `dlX` dimensions labeled `'S'`.

```
SSize = size(dlX,finddim(dlX,'S'))
```

```
SSize = 1×2
```

```
     4     3
```

Input Arguments

`dlX` — Input `dlarray`

`dlarray` object

Input `dlarray`, specified as a `dlarray` object.

Example: `dlX = dlarray(randn(3,4),'ST')`

`label` — Single dimension label

`'S'` | `'C'` | `'B'` | `'T'` | `'U'`

Single dimension label, specified as one of the following `dlarray` dimension labels:

- `S` — Spatial
- `C` — Channel
- `B` — Batch observations
- `T` — Time or sequence
- `U` — Unspecified

Example: `"C"`

Data Types: `char` | `string`

Output Arguments

`dim` — Dimension

real vector

Dimension, returned as a real vector. If no label in the input array `dlX` matches `label`, `dim` is empty. So if `dlX` is unformatted, `dim` is empty.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The label argument must be a compile-time constant.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

The label argument must be a compile-time constant.

See Also

`dims` | `stripdims` | `dlarray`

Introduced in R2019b

findPlaceholderLayers

Find placeholder layers in network architecture imported from Keras or ONNX

Syntax

```
placeholderLayers = findPlaceholderLayers(importedLayers)
[placeholderLayers,indices] = findPlaceholderLayers(importedLayers)
```

Description

`placeholderLayers = findPlaceholderLayers(importedLayers)` returns all placeholder layers that exist in the network architecture `importedLayers` imported by the `importKerasLayers` or `importONNXLayers` functions, or created by the `functionToLayerGraph` function. Placeholder layers are the layers that these functions insert in place of layers that are not supported by Deep Learning Toolbox.

To use with an imported network, this function requires either the Deep Learning Toolbox Converter for TensorFlow Models support package or the Deep Learning Toolbox Converter for ONNX Model Format support package.

`[placeholderLayers,indices] = findPlaceholderLayers(importedLayers)` also returns the indices of the placeholder layers.

Examples

Find and Explore Placeholder Layers

Specify the Keras network file to import layers from.

```
modelfile = 'digitsDAGnetwithnoise.h5';
```

Import the network architecture. The network includes some layer types that are not supported by Deep Learning Toolbox. The `importKerasLayers` function replaces each unsupported layer with a placeholder layer and returns a warning message.

```
lgraph = importKerasLayers(modelfile)
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

```
lgraph =
  LayerGraph with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Display the imported layers of the network. Two placeholder layers replace the Gaussian noise layers in the Keras network.

```
lgraph.Layers

ans =
  15x1 Layer array with layers:

     1   'input_1'                             Image Input             28x28x1 images
     2   'conv2d_1'                            Convolution             20 7x7 convolutions with st
     3   'conv2d_1_relu'                       ReLU                    ReLU
     4   'conv2d_2'                            Convolution             20 3x3 convolutions with st
     5   'conv2d_2_relu'                       ReLU                    ReLU
     6   'gaussian_noise_1'                    PLACEHOLDER LAYER       Placeholder for 'GaussianNo
     7   'gaussian_noise_2'                    PLACEHOLDER LAYER       Placeholder for 'GaussianNo
     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
```

Find the placeholder layers using `findPlaceholderLayers`. The output argument contains the two placeholder layers that `importKerasLayers` inserted in place of the Gaussian noise layers of the Keras network.

```
placeholders = findPlaceholderLayers(lgraph)

placeholders =
  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
```

Specify a name for each placeholder layer.

```
gaussian1 = placeholders(1);
gaussian2 = placeholders(2);
```

Display the configuration of each placeholder layer.

```
gaussian1.KerasConfiguration

ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


gaussian2.KerasConfiguration

ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

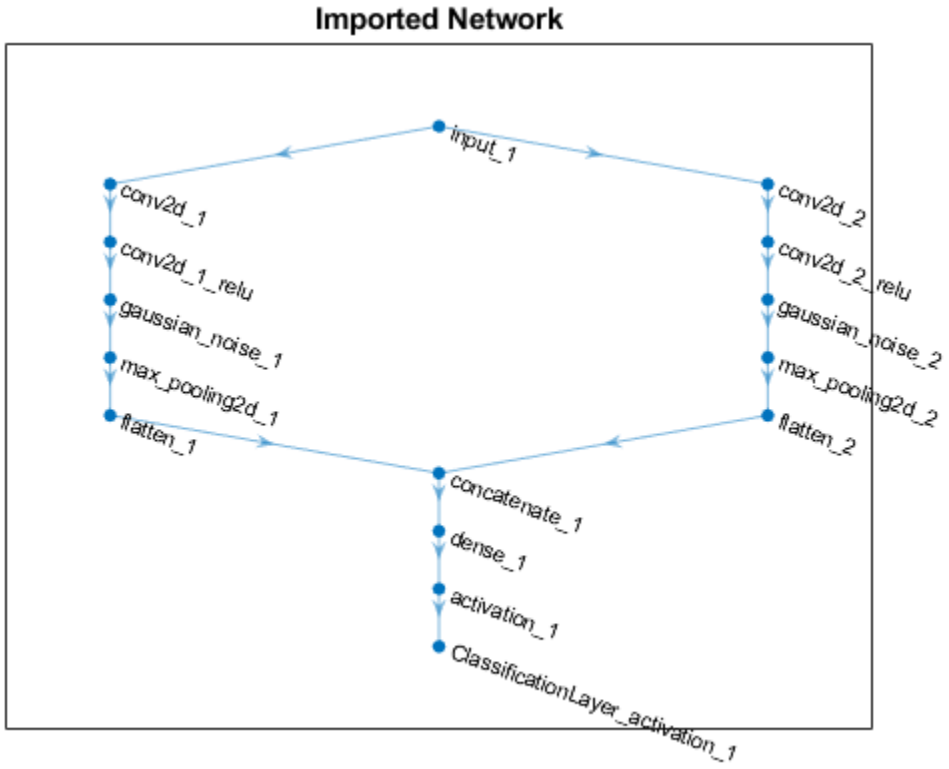
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

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



Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```
placeholderLayers.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

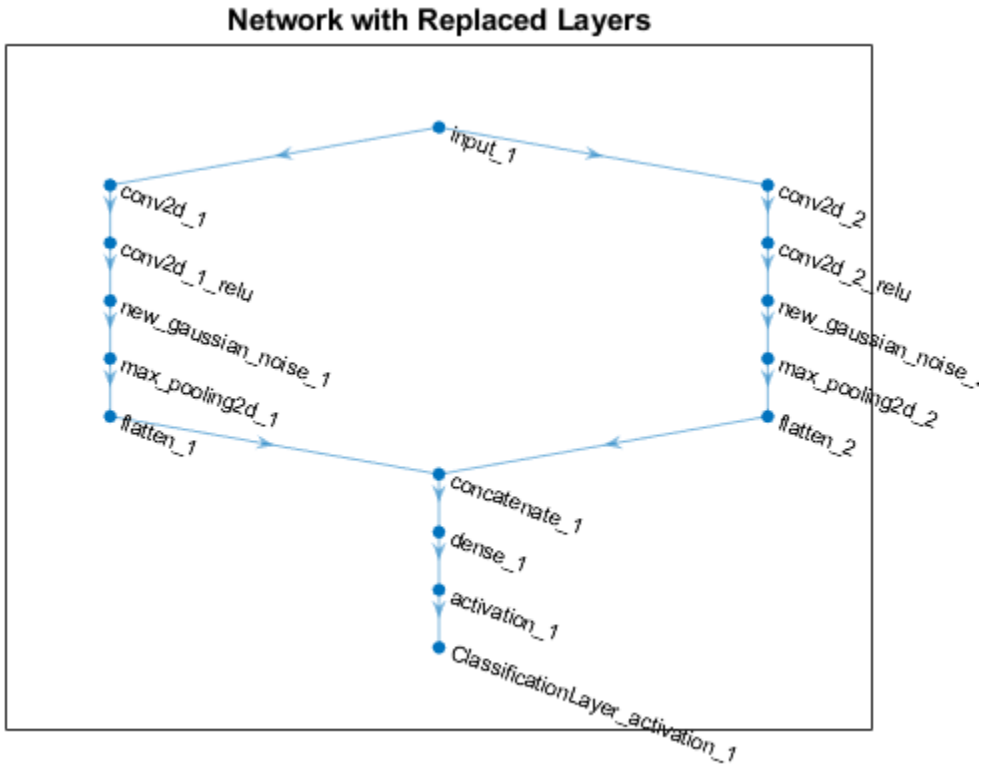
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```
figure
plot(lgraph)
title("Network with Replaced Layers")
```



Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                          Image Input             28x28x1 images
     2   'conv2d_1'                         Convolution             20 7x7x1 convolutions with
     3   'conv2d_1_relu'                    ReLU                    ReLU
     4   'conv2d_2'                         Convolution             20 3x3x1 convolutions with
     5   'conv2d_2_relu'                    ReLU                    ReLU
     6   'new_gaussian_noise_1'             Gaussian Noise          Gaussian noise with standar
     7   'new_gaussian_noise_2'             Gaussian Noise          Gaussian noise with standar
     8   'max_pooling2d_1'                  Max Pooling             2x2 max pooling with stride
     9   'max_pooling2d_2'                  Max Pooling             2x2 max pooling with stride
    10   'flatten_1'                        Keras Flatten           Flatten activations into 1
    11   'flatten_2'                        Keras Flatten           Flatten activations into 1
    12   'concatenate_1'                    Depth concatenation     Depth concatenation of 2 in
    13   'dense_1'                          Fully Connected         10 fully connected layer
    14   'activation_1'                     Softmax                 softmax
    15   'ClassificationLayer_activation_1' Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_activation_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `'auto'`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: [0    1    2    3    4    5    6    7    8    9]
    ClassWeights: 'none'
      OutputSize: 10

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Input Arguments

`importedLayers` — Network architecture imported from Keras or ONNX or created by `functionToLayerGraph`

`Layer` array | `LayerGraph` object

Network architecture imported from Keras or ONNX or created by `functionToLayerGraph`, specified as a `Layer` array or `LayerGraph` object.

Output Arguments

`placeholderLayers` — All placeholder layers in network architecture

array of `PlaceholderLayer` objects

All placeholder layers in the network architecture, returned as an array of `PlaceholderLayer` objects.

`indices` — Indices of placeholder layers

vector

Indices of placeholder layers, returned as a vector.

- If `importedLayers` is a layer array, then `indices` are the indices of the placeholder layers in `importedLayers`.
- If `importedLayers` is a `LayerGraph` object, then `indices` are the indices of the placeholder layers in `importedLayers.Layers`.

If you remove a layer from or add a layer to a `Layer` array or `LayerGraph` object, then the indices of the other layers in the object can change. You must use `findPlaceholderLayers` again to find the updated indices of the rest of the placeholder layers.

Tips

- If you have installed Deep Learning Toolbox Converter for TensorFlow Models and `findPlaceholderLayers` is unable to find placeholder layers created when the ONNX network is imported, then try updating the Deep Learning Toolbox Converter for TensorFlow Models support package in the Add-On Explorer.

See Also

`importKerasLayers` | `PlaceholderLayer` | `replaceLayer` | `assembleNetwork` | `importONNXLayers` | `functionToLayerGraph` | `functionLayer`

Topics

“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

Introduced in R2017b

flattenLayer

Flatten layer

Description

A flatten layer collapses the spatial dimensions of the input into the channel dimension.

For example, if the input to the layer is an *H*-by-*W*-by-*C*-by-*N*-by-*S* array (sequences of images), then the flattened output is an (*H**W**C*)-by-*N*-by-*S* array.

This layer supports sequence input only.

Creation

Syntax

```
layer = flattenLayer
layer = flattenLayer('Name',Name)
```

Description

`layer = flattenLayer` creates a flatten layer.

`layer = flattenLayer('Name',Name)` sets the optional `Name` property using a name-value pair. For example, `flattenLayer('Name','flatten1')` creates a flatten layer with name `'flatten1'`.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Object Functions

Examples

Create Flatten Layer

Create a flatten layer with the name `'flatten1'`.

```
layer = flattenLayer('Name','flatten1')
```

```
layer =
  FlattenLayer with properties:

    Name: 'flatten1'
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of `[28 28 1]`.
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

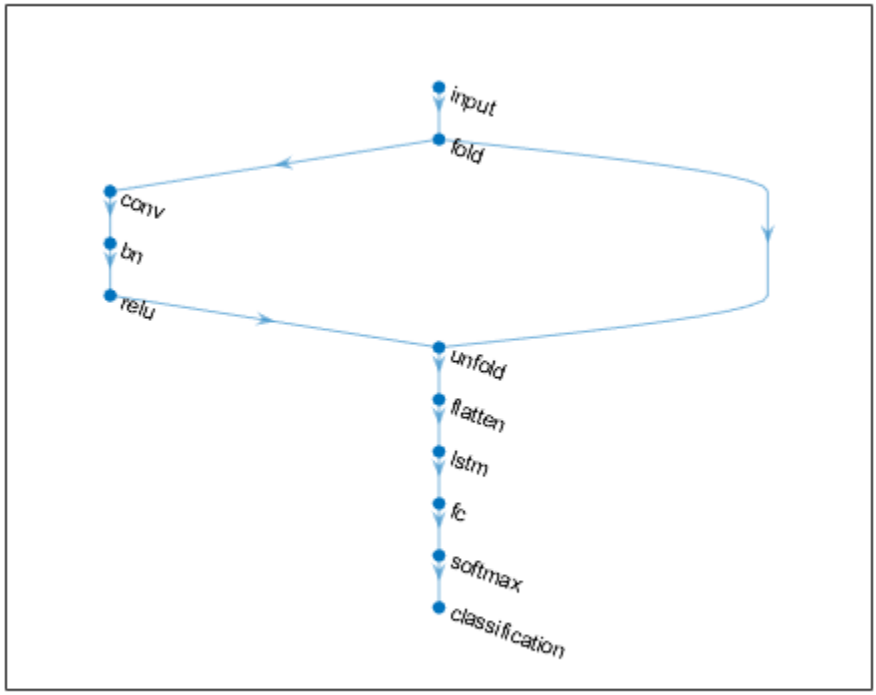
```
inputSize = [28 28 1];
filterSize = 5;
numFilters = 20;
numHiddenUnits = 200;
numClasses = 10;

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');
```

View the final network architecture using the `plot` function.

```
figure
plot(lgraph)
```



Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `resetState` | `sequenceFoldingLayer` | `sequenceUnfoldingLayer` | `sequenceInputLayer`

Topics

“Classify Videos Using Deep Learning”
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2019a

forward

Compute deep learning network output for training

Syntax

```
dlY = forward(dlnet,dlX)
dlY = forward(dlnet,dlX1,...,dlXM)
[dlY1,...,dlYN] = forward( ___ )
[dlY1,...,dlYK] = forward( ___ ,'Outputs',layerNames)
[ ___ ] = forward( ___ ,'Acceleration',acceleration)
[ ___ ,state] = forward( ___ )
```

Description

Some deep learning layers behave differently during training and inference (prediction). For example, during training, dropout layers randomly set input elements to zero to help prevent overfitting, but during inference, dropout layers do not change the input.

To compute network outputs for training, use the `forward` function. To compute network outputs for inference, use the `predict` function.

`dlY = forward(dlnet,dlX)` returns the network output `dlY` during training given the input data `dlX`.

`dlY = forward(dlnet,dlX1,...,dlXM)` returns the network output `dlY` during training given the `M` inputs `dlX1, ...,dlXM` and the network `dlnet` that has `M` inputs and a single output.

`[dlY1,...,dlYN] = forward( ___ )` returns the `N` outputs `dlY1`, ..., `dlYN` during training for networks that have `N` outputs using any of the previous syntaxes.

`[dlY1,...,dlYK] = forward( ___ ,'Outputs',layerNames)` returns the outputs `dlY1`, ..., `dlYK` during training for the specified layers using any of the previous syntaxes.

`[ ___ ] = forward( ___ ,'Acceleration',acceleration)` also specifies performance optimization to use during training, in addition to the input arguments in previous syntaxes.

`[ ___ ,state] = forward( ___ )` also returns the updated network state.

Examples

Train Network Using Custom Training Loop

This example shows how to train a network that classifies handwritten digits with a custom learning rate schedule.

If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation.

This example trains a network to classify handwritten digits with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + k\,t}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.

Load Training Data

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

```
dataFolder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ....
    'LabelSource','foldernames');
```

Partition the data into training and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9,'randomize');
```

The network used in this example requires input images of size 28-by-28-by-1. To automatically resize the training images, use an augmented image datastore. Specify additional augmentation operations to perform on the training images: randomly translate the images up to 5 pixels in the horizontal and vertical axes. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
inputSize = [28 28 1];
pixelRange = [-5 5];
imageAugmenter = imageDataAugmenter( ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain,'DataAugmentation',imageAugmente
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

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

Determine the number of classes in the training data.

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

Define Network

Define the network for image classification.

```
layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','input')
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding','same','Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding','same','Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
```

```
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

```
dlnet =
  dlnetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [11×2 table]
     Learnables: [14×3 table]
          State: [6×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object, a mini-batch of input data with corresponding labels and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Specify Training Options

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

```
numEpochs = 10;
miniBatchSize = 128;
```

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

```
initialLearnRate = 0.01;
decay = 0.01;
momentum = 0.9;
```

Train Model

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

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

```
mbq = minibatchqueue(augimdsTrain,...
    'MiniBatchSize',miniBatchSize,...
```

```
    'MiniBatchFcn',@preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Initialize the training progress plot.

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

Initialize the velocity parameter for the SGDM solver.

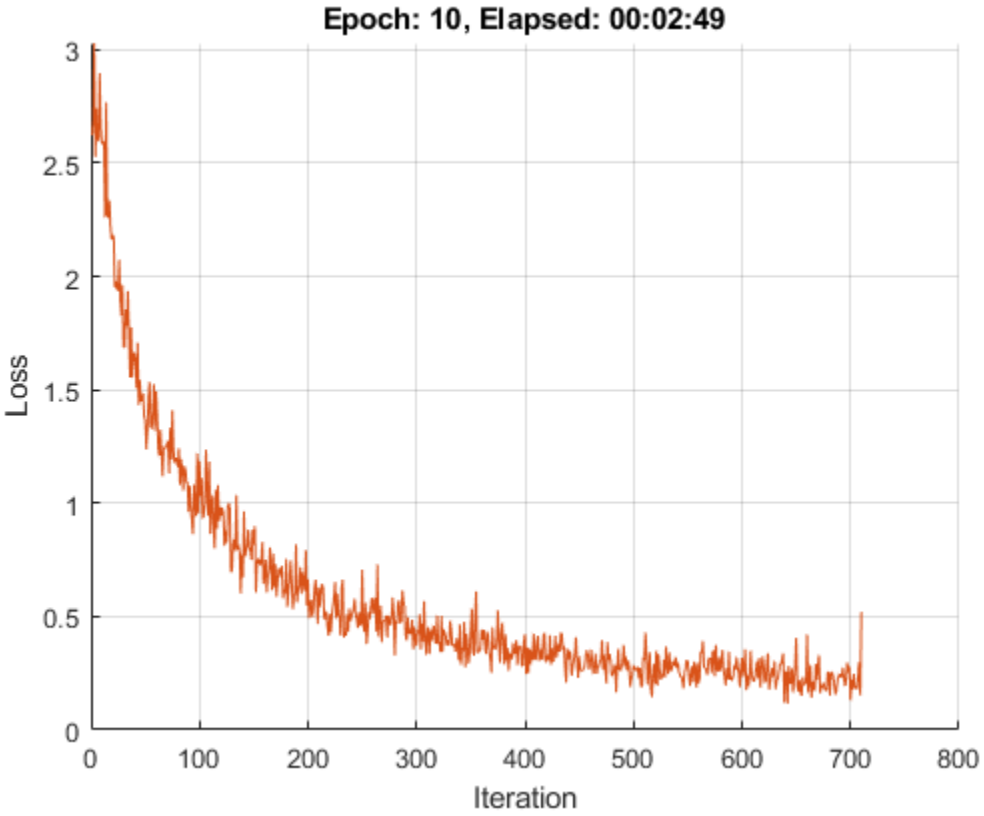
```
velocity = [];
```

Train the network using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using the `dlfeval` and `modelGradients` functions and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq);

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

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

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

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

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

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
```

```
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

After training, making predictions on new data does not require the labels. Create `minibatchqueue` object containing only the predictors of the test data:

- To ignore the labels for testing, set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessMiniBatchPredictors` function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCB'` (spatial, spatial, channel, batch).

```
numOutputs = 1;
mbqTest = minibatchqueue(augimdsValidation,numOutputs, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatchPredictors, ...
    'MiniBatchFormat','SSCB');
```

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

```
predictions = modelPredictions(dlnet,mbqTest,classes);
```

Evaluate the classification accuracy.

```
YTest = imdsValidation.Labels;
accuracy = mean(predictions == YTest)
```

```
accuracy = 0.9530
```

Model Gradients Function

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

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

[dlYPred,state] = forward(dlnet,dlX);

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

loss = double(gather(extractdata(loss)));

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)

predictions = [];

while hasdata(mbq)

    dlXTest = next(mbq);
    dlYPred = predict(dlnet,dlXTest);

    YPred = onehotdecode(dlYPred,classes,1)';

    predictions = [predictions; YPred];
end

end
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Preprocess the images using the `preprocessMiniBatchPredictors` function.
2. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.

3 One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

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

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate.
Y = cat(2,YCell{1:end});

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

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

end
```

Input Arguments

`dlnet` — Network for custom training loops

`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`dlX` — Input data

formatted `dlarray`

Input data, specified as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

`layerNames` — Layers to extract outputs from

string array | cell array of character vectors

Layers to extract outputs from, specified as a string array or a cell array of character vectors containing the layer names.

- If `layerNames(i)` corresponds to a layer with a single output, then `layerNames(i)` is the name of the layer.
- If `layerNames(i)` corresponds to a layer with multiple outputs, then `layerNames(i)` is the layer name followed by the character `"/"` and the name of the layer output: `'layerName/outputName'`.

`acceleration` — Performance optimization
`'auto'` (default) | `'none'`

Performance optimization, specified as one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'none'` — Disable all acceleration.

The default option is `'auto'`.

Using the `'auto'` acceleration option can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using different input data with the same size and shape.

Output Arguments

`dlY` — Output data
formatted `dlarray`

Output data, returned as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

`state` — Updated network state
table

Updated network state, returned as a table.

The network state is a table with three columns:

- `Layer` – Layer name, specified as a string scalar.
- `Parameter` – State parameter name, specified as a string scalar.
- `Value` – Value of state parameter, specified as a `dlarray` object.

Layer states contain information calculated during the layer operation to be retained for use in subsequent forward passes of the layer. For example, the cell state and hidden state of LSTM layers, or running statistics in batch normalization layers.

For recurrent layers, such as LSTM layers, with the `HasStateInputs` property set to `1` (true), the state table does not contain entries for the states of that layer.

Update the state of a `dlnetwork` using the `State` property.

Compatibility Considerations

`forward` returns state values as `dlarray` objects
Behavior changed in R2021a

For `dlnetwork` objects, the `state` output argument returned by the `forward` function is a table containing the state parameter names and values for each layer in the network.

Starting in R2021a, the state values are `dlarray` objects. This change enables better support when using `AcceleratedFunction` objects. To accelerate deep learning functions that have frequently

changing input values, for example, an input containing the network state, the frequently changing values must be specified as `dlarray` objects.

In previous versions, the state values are numeric arrays.

In most cases, you will not need to update your code. If you have code that requires the state values to be numeric arrays, then to reproduce the previous behavior, extract the data from the state values manually using the `extractdata` function with the `dlupdate` function.

```
state = dlupdate(@extractdata,dlnet.State);
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function runs on the GPU if either or both of the following conditions are met:
  - Any of the values of the network learnable parameters inside `dlnet.Learnables.Value` are `dlarray` objects with underlying data of type `gpuArray`
  - The input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

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

Topics

“Train Generative Adversarial Network (GAN)”
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2019b

freezeParameters

Convert learnable network parameters in `ONNXParameters` to nonlearnable

Syntax

```
params = freezeParameters(params,names)
```

Description

`params = freezeParameters(params,names)` freezes the network parameters specified by `names` in the `ONNXParameters` object `params`. The function moves the specified parameters from `params.Learnables` in the input argument `params` to `params.Nonlearnables` in the output argument `params`.

Examples

Train Imported ONNX Function Using Custom Training Loop

Import the `squeezenet` convolution neural network as a function and fine-tune the pretrained network with transfer learning to perform classification on a new collection of images.

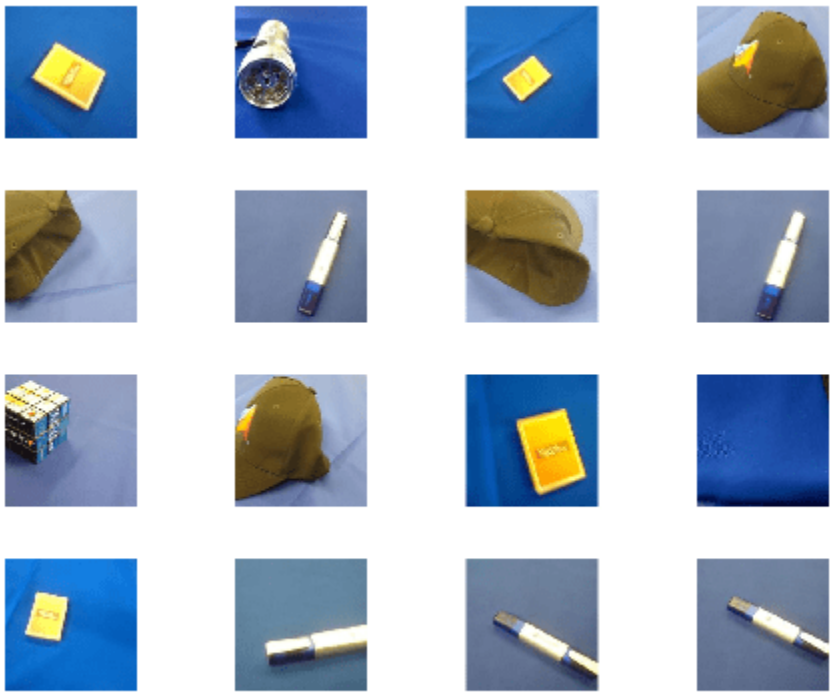
This example uses several helper functions. To view the code for these functions, see Helper Functions on page 1-0 .

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. Specify the mini-batch size.

```
unzip('MerchData.zip');
miniBatchSize = 8;
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames',...
    'ReadSize', miniBatchSize);
```

This data set is small, containing 75 training images. Display some sample images.

```
numImages = numel(imds.Labels);
idx = randperm(numImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imds,idx(i));
    imshow(I)
end
```



Extract the training set and one-hot encode the categorical classification labels.

```
XTrain = readall(imds);
XTrain = single(cat(4,XTrain{:}));
YTrain_categ = categorical(imds.Labels);
YTrain = onehotencode(YTrain_categ,2)';
```

Determine the number of classes in the data.

```
classes = categories(YTrain_categ);
numClasses = numel(classes)
```

```
numClasses = 5
```

`squeezenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

Import the pretrained `squeezenet` network as a function.

```
squeezenetONNX()
params = importONNXFunction('squeezenet.onnx','squeezenetFcn')
```

```
A function containing the imported ONNX network has been saved to the file squeezenetFcn.m.
To learn how to use this function, type: help squeezenetFcn.

params =
  ONNXParameters with properties:
```

```
            Learnables: [1×1 struct]
       Nonlearnables: [1×1 struct]
               State: [1×1 struct]
       NumDimensions: [1×1 struct]
NetworkFunctionName: 'squeezenetFcn'
```

`params` is an `ONNXParameters` object that contains the network parameters. `squeezenetFcn` is a model function that contains the network architecture. `importONNXFunction` saves `squeezenetFcn` in the current folder.

Calculate the classification accuracy of the pretrained network on the new training set.

```
accuracyBeforeTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy before transfer learning\n',accuracyBeforeTraining);
```

```
0.01 accuracy before transfer learning
```

The accuracy is very low.

Display the learnable parameters of the network by typing `params.Learnables`. These parameters, such as the weights (`W`) and bias (`B`) of convolution and fully connected layers, are updated by the network during training. Nonlearnable parameters remain constant during training.

The last two learnable parameters of the pretrained network are configured for 1000 classes.

```
conv10_W: [1×1×512×1000 dlarray]
```

```
conv10_B: [1000×1 dlarray]
```

The parameters `conv10_W` and `conv10_B` must be fine-tuned for the new classification problem. Transfer the parameters to classify five classes by initializing the parameters.

```
params.Learnables.conv10_W = rand(1,1,512,5);
params.Learnables.conv10_B = rand(5,1);
```

Freeze all the parameters of the network to convert them to nonlearnable parameters. Because you do not need to compute the gradients of the frozen layers, freezing the weights of many initial layers can significantly speed up network training.

```
params = freezeParameters(params,'all');
```

Unfreeze the last two parameters of the network to convert them to learnable parameters.

```
params = unfreezeParameters(params,'conv10_W');
params = unfreezeParameters(params,'conv10_B');
```

Now the network is ready for training. Initialize the training progress plot.

```
plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end
```

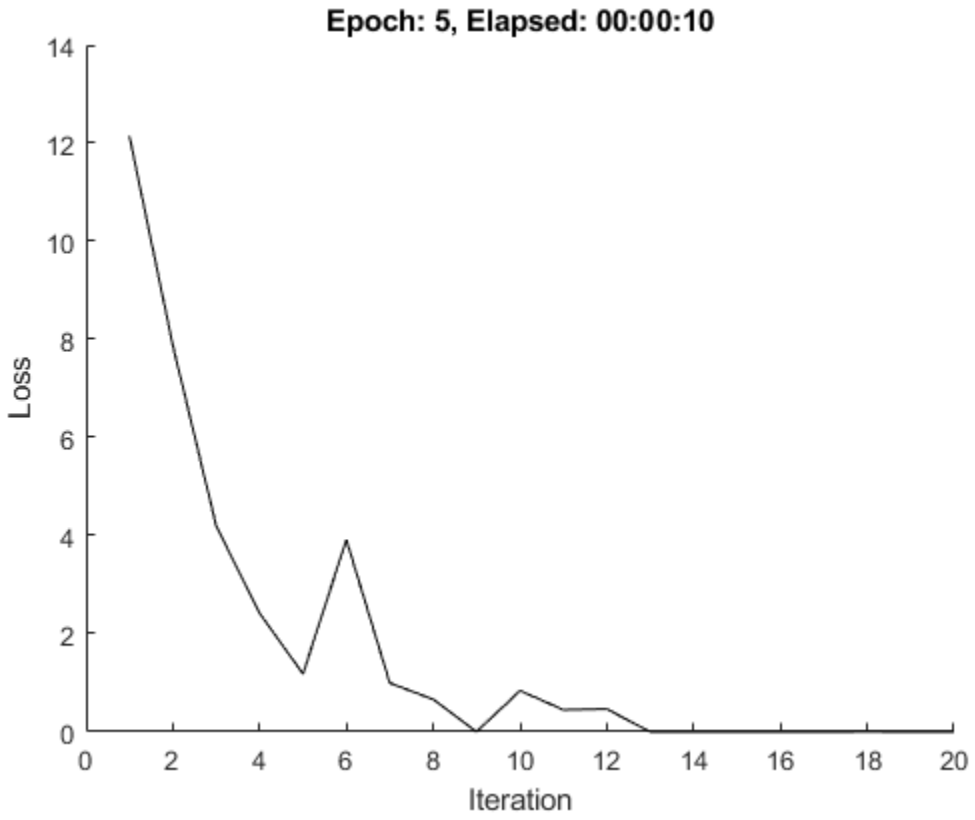
Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Train the network.

```
iteration = 0;
start = tic;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle 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.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);
        Y = YTrain(:,idx);

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

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
        params.State = state;

        % Determine the learning rate for the time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);

        % 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
```



Calculate the classification accuracy of the network after fine-tuning.

```
accuracyAfterTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy after transfer learning\n',accuracyAfterTraining);
```

```
1.00 accuracy after transfer learning
```

Helper Functions

This section provides the code of the helper functions used in this example.

The `getNetworkAccuracy` function evaluates the network performance by calculating the classification accuracy.

```
function accuracy = getNetworkAccuracy(X,Y,onnxParams)

N = size(X,4);
Ypred = squeezenetFcn(X,onnxParams,'Training',false);

[~,YIdx] = max(Y,[],1);
[~,YpredIdx] = max(Ypred,[],1);
numIncorrect = sum(abs(YIdx-YpredIdx) > 0);
accuracy = 1 - numIncorrect/N;

end
```

The `modelGradients` function calculates the loss and gradients.

```
function [grad, loss, state] = modelGradients(X,Y,onnxParams)

[y,state] = squeezenetFcn(X,onnxParams,'Training',true);
loss = crossentropy(y,Y,'DataFormat','CB');
grad = dlgradient(loss,onnxParams.Learnables);

end
```

The `squeezenetONNX` function generates an ONNX model of the `squeezenet` network.

```
function squeezenetONNX()

exportONNXNetwork(squeezenet,'squeezenet.onnx');

end
```

Input Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, specified as an `ONNXParameters` object. `params` contains the network parameters of the imported ONNX model.

`names` — Names of parameters to freeze

`'all'` | string array

Names of the parameters to freeze, specified as `'all'` or a string array. Freeze all learnable parameters by setting `names` to `'all'`. Freeze `k` learnable parameters by defining the parameter names in the 1-by-`k` string array `names`.

Example: `'all'`

Example: `["gpu_0_sl_pred_b_0", "gpu_0_sl_pred_w_0"]`

Data Types: `char` | `string`

Output Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, returned as an `ONNXParameters` object. `params` contains the network parameters updated by `freezeParameters`.

See Also

`importONNXFunction` | `ONNXParameters` | `unfreezeParameters`

Introduced in R2020b

fullyconnect

Sum all weighted input data and apply a bias

Syntax

```
dlY = fullyconnect(dlX,weights,bias)
dlY = fullyconnect(dlX,weights,bias,'DataFormat',FMT)
```

Description

The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.

Note This function applies the fully connect operation to `dlarray` data. If you want to apply the fully connect operation within a `layerGraph` object or `Layer` array, use the following layer:

- `fullyConnectedLayer`

`dlY = fullyconnect(dlX,weights,bias)` computes the weighted sum of the spatial, channel, and unspecified data in `dlX` using the weights specified by `weights`, and adds a bias. The input `dlX` must be a formatted `dlarray`. The output `dlY` is a formatted `dlarray`.

`dlY = fullyconnect(dlX,weights,bias,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray`.

Examples

Fully Connect All Input Data to Output Features

The `fullyconnect` function uses the weighted sum to connect all inputs of an observation to each output feature.

Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```
height = 12;
width = 12;
channels = 32;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Create the learnable parameters. For this operation there are ten output features.

```
outputFeatures = 10;

weights = ones(outputFeatures,height,width,channels);
bias = ones(outputFeatures,1);
```

Apply the `fullyconnect` operation.

```
dlY = fullyconnect(dlX,weights,bias);
```

```
dlY =
  10(C) × 1(B) dlarray

   1.0e+03 *

    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
    2.3266
```

The output `dlY` is a 2-D `dlarray` with one channel dimension of size ten and one singleton batch dimension.

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using `'DataFormat',FMT`. If `dlX` is a numeric array, at least one of `weights` or `bias` must be a `dlarray`.

The `fullyconnect` operation sums over the `'S'`, `'C'`, and `'U'` dimensions of `dlX` for each output feature specified by `weights`. The size of each `'B'` or `'T'` dimension of `dlX` is preserved.

Data Types: `single` | `double`

`weights` — Weights

`dlarray` | numeric array

Weights, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `weights` is an unformatted `dlarray` or a numeric array, the first dimension of `weights` must match the number of output features. If `weights` is a formatted `dlarray`, the size of the `'C'` dimension must match the number of output features. `weights` must contain the same number of elements as the combined size of the `'S'`, `'C'`, and `'U'` dimensions of input `dlX` multiplied by the number of output features.

Data Types: `single` | `double`

`bias` — Bias constant

`dlarray` vector | numeric vector

Bias constant, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Each element of `bias` is the bias applied to the corresponding feature output. The number of elements of `bias` must match the number of output features specified by the first dimension of `weights`.

If `bias` is a formatted `dlarray`, the nonsingleton dimension must be a channel dimension labeled `'C'`.

Data Types: `single` | `double`

`FMT` — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`dlY` — Weighted output features
`dlarray`

Weighted output features, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input `dlX` is a formatted `dlarray`, the output `dlY` has one dimension labeled `'C'` representing the output features, and the same number of `'B'` or `'T'` dimensions as the input `dlX`, if either or both are present. If `dlX` has no `'B'` or `'T'` dimensions, `dlY` has the format `'CB'`, where the `'B'` dimension is singleton.

If the input `dlX` is not a formatted `dlarray`, output `dlY` is unformatted. The first dimension of `dlY` contains the output features. Other dimensions of `dlY` correspond to the `'B'` and `'T'` dimensions of `dlX`, if either or both are present, and are provided in the same order as in `FMT`. If `dlX` has no `'B'` or `'T'` dimensions, the first dimension of `dlY` contains the output features and the second dimension is singleton.

More About

Fully Connect Operation

The `fullyconnect` function connects all outputs of the previous operation to the outputs of the `fullyconnect` function. For more information, see the definition of "Fully Connected Layer" on page 1-618 on the `fullyConnectedLayer` reference page.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `weights`
  - `bias`

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also
`dlarray` | `batchnorm` | `relu` | `dlconv` | `sigmoid` | `softmax` | `dlgradient` | `dlfeval`

Topics
"Define Custom Training Loops, Loss Functions, and Networks"
"Train Network Using Model Function"
"Make Predictions Using Model Function"
"Train a Siamese Network to Compare Images"
"Train Network with Multiple Outputs"
"List of Functions with dlarray Support"

Introduced in R2019b

fullyConnectedLayer

Fully connected layer

Description

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

Creation

Syntax

```
layer = fullyConnectedLayer(outputSize)
layer = fullyConnectedLayer(outputSize,Name,Value)
```

Description

`layer = fullyConnectedLayer(outputSize)` returns a fully connected layer and specifies the `OutputSize` property.

`layer = fullyConnectedLayer(outputSize,Name,Value)` sets the optional "Parameters and Initialization" on page 1-612, "Learning Rate and Regularization" on page 1-614, and `Name` properties using name-value pairs. For example, `fullyConnectedLayer(10,'Name','fc1')` creates a fully connected layer with an output size of 10 and the name `'fc1'`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

Fully Connected

`OutputSize` — Output size
positive integer

Output size for the fully connected layer, specified as a positive integer.

Example: `10`

`InputSize` — Input size
`'auto'` (default) | positive integer

Input size for the fully connected layer, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically determines the input size during training.

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'orthogonal'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(InputSize + OutputSize)`.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/InputSize`.
- `'orthogonal'` – Initialize the input weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [3]
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | matrix

Layer weights, specified as a matrix.

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

At training time, `Weights` is an `OutputSize`-by-`InputSize` matrix.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | matrix

Layer biases, specified as a matrix.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is an `OutputSize`-by-`1` matrix.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Fully Connected Layer

Create a fully connected layer with an output size of 10 and the name `'fc1'`.

```
layer = fullyConnectedLayer(10,'Name','fc1')
```

```
layer =
  FullyConnectedLayer with properties:

         Name: 'fc1'

   Hyperparameters
     InputSize: 'auto'
    OutputSize: 10

   Learnable Parameters
       Weights: []
          Bias: []

  Show all properties
```

Include a fully connected layer in a `Layer` array.

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

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Specify Initial Weights and Biases in Fully Connected Layer

To specify the weights and bias initializer functions, use the `WeightsInitializer` and `BiasInitializer` properties respectively. To specify the weights and biases directly, use the `Weights` and `Bias` properties respectively.

Specify Initialization Function

Create a fully connected layer with an output size of 10 and specify the weights initializer to be the He initializer.

```
outputSize = 10;
layer = fullyConnectedLayer(outputSize,'WeightsInitializer','he')
```

```
layer =
  FullyConnectedLayer with properties:
```

```
          Name: ''

   Hyperparameters
     InputSize: 'auto'
    OutputSize: 10

   Learnable Parameters
       Weights: []
          Bias: []

  Show all properties
```

Note that the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Function

To specify your own initialization function for the weights and biases, set the `WeightsInitializer` and `BiasInitializer` properties to a function handle. For these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a fully connected layer with output size 10 and specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```
outputSize = 10;
weightsInitializationFcn = @(sz) rand(sz) * 0.0001;
biasInitializationFcn = @(sz) rand(sz) * 0.0001;

layer = fullyConnectedLayer(outputSize, ...
    'WeightsInitializer',@(sz) rand(sz) * 0.0001, ...
    'BiasInitializer',@(sz) rand(sz) * 0.0001)
```

```
layer =
  FullyConnectedLayer with properties:

          Name: ''

   Hyperparameters
     InputSize: 'auto'
    OutputSize: 10

   Learnable Parameters
       Weights: []
          Bias: []

  Show all properties
```

Again, the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Weights and Bias Directly

Create a fully connected layer with an output size of 10 and set the weights and bias to `W` and `b` in the MAT file `FCWeights.mat` respectively.

```
outputSize = 10;
load FCWeights
```

```
layer = fullyConnectedLayer(outputSize, ...
    'Weights',W, ...
    'Bias',b)

layer =
  FullyConnectedLayer with properties:

          Name: ''

   Hyperparameters
     InputSize: 720
    OutputSize: 10

   Learnable Parameters
       Weights: [10x720 double]
          Bias: [10x1 double]

  Show all properties
```

Here, the `Weights` and `Bias` properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

Algorithms

Fully Connected Layer

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”.

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.

The fully connected layer flattens the output. It reshapes the array such that the spatial data is encoded in the channel dimension.

For sequence input, the layer applies the fully connect operation independently to each time step of the input.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `FullyConnectedLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	Output Format
`"CB"` (channel, batch)	`"CB"` (channel, batch)
`"SCB"` (spatial, channel, batch)	`"CB"` (channel, batch)
`"SSCB"` (spatial, spatial, channel, batch)	`"CB"` (channel, batch)
`"SSSCB"` (spatial, spatial, spatial, channel, batch)	`"CB"` (channel, batch)
`"CBT"` (channel, batch, time)	`"CBT"` (channel, batch, time)

In `dlnetwork` objects, `FullyConnectedLayer` objects also support the following input and output format combinations.

Input Format	Output Format
`"SCBT"` (spatial, channel, batch)	`"CBT"` (channel, batch, time)
`"SSCBT"` (spatial, spatial, channel, batch, time)	`"CBT"` (channel, batch, time)
`"SSSCBT"` (spatial, spatial, spatial, channel, batch, time)	`"CBT"` (channel, batch, time)

To use these input formats in `trainNetwork` workflows, first convert the data to `"CBT"` (channel, batch, time) format using `flattenLayer`.

Compatibility Considerations

Default weights initialization is Glorot

Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'WeightsInitializer'` option of the layer to `'narrow-normal'`.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

[3] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." *arXiv preprint arXiv:1312.6120* (2013).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `convolution2dLayer` | `reluLayer` | `batchNormalizationLayer` | **Deep Network Designer**

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2016a

functionLayer

Function layer

Description

A function layer applies a specified function to the layer input.

If Deep Learning Toolbox does not provide the layer that you need for your task, then you can define new layers by creating function layers using `functionLayer`. Function layers only support operations that do not require additional properties, learnable parameters, or states. For layers that require this functionality, define the layer as a custom layer. For more information, see “Define Custom Deep Learning Layers”.

Creation

Syntax

```
layer = functionLayer(fun)
layer = functionLayer(fun,Name=Value)
```

Description

`layer = functionLayer(fun)` creates a function layer and sets the `PredictFcn` property.

`layer = functionLayer(fun,Name=Value)` sets optional properties on page 1-621 using one or more name-value arguments. For example, `functionLayer(fun,NumInputs=2,NumOutputs=3)` specifies that the layer has two inputs and three outputs. You can specify multiple name-value arguments.

Properties

Function

`PredictFcn` — Function to apply to layer input
function handle

This property is read-only.

Function to apply to layer input, specified as a function handle.

The specified function must have syntax `[Y1,...,YM] = fun(X1,...,XN)`, where the inputs and outputs are `dlarray` objects, and `M` and `N` correspond to the `NumOutputs` and `NumInputs` properties, respectively.

The inputs `X1`, …, `XN` correspond to the layer inputs with names given by `InputNames`. The outputs `Y1`, …, `YM` correspond to the layer outputs with names given by `OutputNames`.

For a list of functions that support `dlarray` input, see “List of Functions with dlarray Support”.

Tip When using the layer, you must ensure that the specified function is accessible. For example, to ensure that the layer can be reused in multiple live scripts, save the function in its own separate file.

Data Types: `function_handle`

`Formattable` — Flag indicating that function operates on formatted `dlarray` objects
`0` (false) (default) | `1` (true)

This property is read-only.

Flag indicating that layer function operates on formatted `dlarray` objects, specified as `0` (false) or `1` (true).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`Description` — One-line description of the layer
string scalar | character vector

This property is read-only.

One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array.

If you do not specify a layer description, then the software displays the layer operation.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
positive integer

This property is read-only.

Number of inputs, specified as a positive integer.

The layer must have a fixed number of inputs. If `PredictFcn` supports a variable number of input arguments using `varargin`, then you must specify the number of layer inputs using `NumInputs`.

If you do not specify `NumInputs`, then the software sets `NumInputs` to `nargin(PredictFcn)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InputNames` — Input names
string array | cell array of character vectors

This property is read-only.

Input names of the layer, specified as a positive integer.

If you do not specify `InputNames` and `NumInputs` is `1`, then the software sets `InputNames` to `{'in'}`. If you do not specify `InputNames` and `NumInputs` is greater than `1`, then the software sets `InputNames` to `{'in1',...,'inN'}`, where `N` is the number of inputs.

Data Types: `string` | `cell`

`NumOutputs` — Number of outputs
1 (default) | positive integer

This property is read-only.

Number of outputs of the layer, specified as a positive integer.

The layer must have a fixed number of outputs. If `PredictFcn` supports a variable number of output arguments, then you must specify the number of layer outputs using `NumOutputs`.

If you do not specify `NumOutputs`, then the software sets `NumOutputs` to `nargout(PrecitcFcn)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`OutputNames` — Output names
string array | cell array of character vectors

This property is read-only.

Output names of the layer, specified as a string array or a cell array of character vectors.

If you do not specify `OutputNames` and `NumOutputs` is `1`, then the software sets `OutputNames` to `{'out'}`. If you do not specify `OutputNames` and `NumOutputs` is greater than `1`, then the software sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is the number of outputs.

Data Types: `string` | `cell`

Examples

Define Softsign Layer as Function Layer

Create a function layer object that applies the softsign operation to the input. The softsign operation is given by the function $f(x) = \frac{x}{1+|x|}$.

```
layer = functionLayer(@(X) X./(1 + abs(X)))
```

```
layer =
  FunctionLayer with properties:

           Name: ''
    PredictFcn: @(X)X./(1+abs(X))
   Formattable: 0

  Learnable Parameters
   No properties.

  State Parameters
```

```
    No properties.

  Show all properties
```

Include a softsign layer, specified as a function layer, in a layer array. Specify that the layer has description `"softsign"`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    functionLayer(@(X) X./(1 + abs(X)),Description="softsign")
    maxPooling2dLayer(2,Stride=2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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  0
     3   ''   Function                softsign
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Reformat Data Using Function Layer

Create a function layer that reformats input data with format `"CB"`(channel, batch) to have format `"SBC"` (spatial, batch, channel). To specify that the layer operates on formatted data, set the `Formattable` option to `true`.

```
layer = functionLayer(@(X) dlarray(X,"SBC"),Formattable=true)
```

```
layer =
  FunctionLayer with properties:

           Name: ''
     PredictFcn: @(X)dlarray(X,"SBC")
    Formattable: 1

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Include a function layer that reformats the input to have format `"SB"` in a layer array. Specify that the layer has description `"channel to spatial"`.

```
layers = [
    featureInputLayer(10)
    functionLayer(@(X) dlarray(X,"SBC"),Formattable=true,Description="channel to spatial")
    convolution1dLayer(3,16)]
```

```
layers =
  3x1 Layer array with layers:

     1   ''   Feature Input   10 features
     2   ''   Function        channel to spatial
     3   ''   Convolution     16 3 convolutions with stride 1 and padding [0  0]
```

In this network, the 1-D convolution layer convolves over the `"S"` (spatial) dimension of its input data. This is equivalent to convolving over the `"C"` (channel) dimension of the network input data.

Convert the layer array to a `dlnetwork` object and pass a random array of data with format `"CB"`.

```
dlnet = dlnetwork(layers);

X = rand(10,64);
dlX = dlarray(X,"CB");

dlY = forward(dlnet,dlX);
```

View the size and format of the output data.

```
size(dlY)
```

```
ans = 1×3

     8    16    64
```

```
dims(dlY)
```

```
ans =
'SCB'
```

Replace Unsupported Keras Layer with Function Layer

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with function layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `"digitsNet.h5"` classifies images of digits.

```
filename = "digitsNet.h5";
layers = importKerasLayers(filename,ImportWeights=true)
```

```
Warning: Unable to import layer. Keras layer 'Activation' with the specified settings is not supp

Warning: Unable to import layer. Keras layer 'Activation' with the specified settings is not supp

Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

```
layers =
  13x1 Layer array with layers:

     1   'ImageInputLayer'               Image Input             28x28x1 images
     2   'conv2d'                        Convolution             8 3x3x1 convolutions with stride
     3   'conv2d_softsign'               PLACEHOLDER LAYER       Placeholder for 'Activation' Ker
     4   'max_pooling2d'                 Max Pooling             2x2 max pooling with stride [2
     5   'conv2d_1'                      Convolution             16 3x3x8 convolutions with strid
     6   'conv2d_1_softsign'             PLACEHOLDER LAYER       Placeholder for 'Activation' Ker
     7   'max_pooling2d_1'               Max Pooling             2x2 max pooling with stride [2
     8   'flatten'                       Keras Flatten           Flatten activations into 1-D ass
     9   'dense'                         Fully Connected         100 fully connected layer
    10   'dense_relu'                    ReLU                    ReLU
    11   'dense_1'                       Fully Connected         10 fully connected layer
    12   'dense_1_softmax'               Softmax                 softmax
    13   'ClassificationLayer_dense_1'   Classification Output   crossentropyex
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using the `findPlaceholderLayers` function.

```
placeholderLayers = findPlaceholderLayers(layers)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'conv2d_softsign'     PLACEHOLDER LAYER   Placeholder for 'Activation' Keras layer
     2   'conv2d_1_softsign'   PLACEHOLDER LAYER   Placeholder for 'Activation' Keras layer
```

Replace the placeholder layers with function layers with function specified by the `softsign` function, listed at the end of the example.

Create a function layer with function specified by the `softsign` function, attached to this example as a supporting file. To access this function, open this example as a live script. Set the layer description to `"softsign"`.

```
layer = functionLayer(@softsign,Description="softsign");
```

Replace the layers using the `replaceLayer` function. To use the `replaceLayer` function, first convert the layer array to a layer graph.

```
lgraph = layerGraph(layers);
lgraph = replaceLayer(lgraph,"conv2d_softsign",layer);
lgraph = replaceLayer(lgraph,"conv2d_1_softsign",layer);
```

Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers

ans =
  13x1 Layer array with layers:

     1   'ImageInputLayer'               Image Input             28x28x1 images
     2   'conv2d'                        Convolution             8 3x3x1 convolutions with stride
     3   'layer'                         Function                softsign
     4   'max_pooling2d'                 Max Pooling             2x2 max pooling with stride [2
     5   'conv2d_1'                      Convolution             16 3x3x8 convolutions with stri
     6   'layer_1'                       Function                softsign
     7   'max_pooling2d_1'               Max Pooling             2x2 max pooling with stride [2
     8   'flatten'                       Keras Flatten           Flatten activations into 1-D as
     9   'dense'                         Fully Connected         100 fully connected layer
    10   'dense_relu'                    ReLU                    ReLU
    11   'dense_1'                       Fully Connected         10 fully connected layer
    12   'dense_1_softmax'               Softmax                 softmax
    13   'ClassificationLayer_dense_1'   Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_dense_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)

cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_dense_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `"auto"`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9);
lgraph = replaceLayer(lgraph,"ClassificationLayer_dense_1",cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)

net =
  DAGNetwork with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [12x2 table]
     InputNames: {'ImageInputLayer'}
    OutputNames: {'ClassificationLayer_dense_1'}
```

Test Network

Make predictions with the network using a test data set.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
```

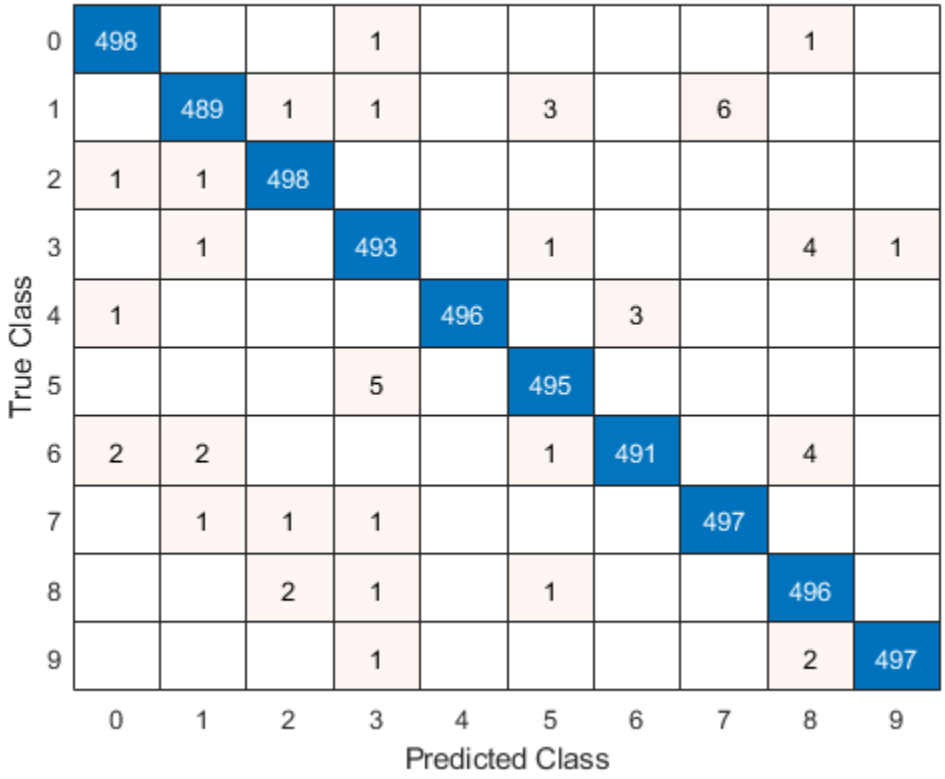
View the accuracy.

```
mean(YPred == YTest)
```

```
ans = 0.9900
```

Visualize the predictions in a confusion matrix.

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



See Also

`layerGraph` | `findPlaceholderLayers` | `PlaceholderLayer` | `connectLayers` | `disconnectLayers` | `addLayers` | `removeLayers` | `assembleNetwork` | `replaceLayer`

Topics

“Deep Learning in MATLAB”
“Replace Unsupported Keras Layer with Function Layer”
“Assemble Network from Pretrained Keras Layers”
“Define Custom Deep Learning Layers”

Introduced in R2021b

functionToLayerGraph

Convert deep learning model function to a layer graph

Syntax

```
lgraph = functionToLayerGraph(fun,x)
lgraph = functionToLayerGraph(fun,x,Name,Value)
```

Description

`lgraph = functionToLayerGraph(fun,x)` returns a layer graph based on the deep learning array function `fun`. `functionToLayerGraph` converts only those operations in `fun` that operate on `dlarray` objects among the inputs in `x`. To include extra parameters or data in `fun`, see the topic "Parameterizing Functions" or the example "Create Layer Graph from Function" on page 1-630.

`functionToLayerGraph` evaluates `fun(x)` and traces the execution to derive an equivalent layer graph, to the extent possible. The steps in `fun(x)` that `functionToLayerGraph` can trace are both based on `dlarray` arguments and are supported calls for `dlarray`. See "List of Functions with dlarray Support". For unsupported functions, `functionToLayerGraph` creates a `PlaceholderLayer`.

`lgraph = functionToLayerGraph(fun,x,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Create Layer Graph from Function

The `simplemodel` function at the end of this example creates fully connected outputs followed by a `softmax` operation. To create a layer graph from this function based on `dlarray` data, create input arrays as `dlarray` objects, and create a function handle to the `simplemodel` function including the data.

```
rng default % For reproducibility
dlX1 = dlarray(rand(10),'CB');
dlX2 = dlarray(zeros(10,1),'CB');
fun = @(x)simplemodel(x,dlX1,dlX2);
```

Call `functionToLayerGraph` using a `dlarray` for the input data `dlX`.

```
dlX = dlarray(ones(10,1),'CB');
lgraph = functionToLayerGraph(fun,dlX)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [1x2 table]
     InputNames: {1x0 cell}
```

```
    OutputNames: {1x0 cell}
```

Examine the resulting layers in `lgraph`.

```
disp(lgraph.Layers)
```

```
  2x1 Layer array with layers:

     1   'fc_1'   Fully Connected   10 fully connected layer
     2   'sm_1'   Softmax           softmax
```

```
function y = simplemodel(x,w,b)
y = fullyconnect(x,w,b);
y = softmax(y);
end
```

Input Arguments

`fun` — Function to convert

function handle

Function to convert, specified as a function handle.

Example: `@relu`

Data Types: `function_handle`

`x` — Data for function

any data type

Data for the function, specified as any data type. Only `dlarray` data is traced and converted to a layer graph.

Example: `dlarray(zeros(12*50,23))`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'GenerateLayer','placeholder-layer'`

`GenerateLayer` — Type of layer to generate for unsupported operations

`'custom-layer'` (default) | `'placeholder-layer'`

Type of layer to generate for unsupported operations in `fun`, specified as `'custom-layer'` or `'placeholder-layer'`.

When an operation in `fun` does not correspond to a layer in Deep Learning Toolbox, the software generates a layer to represent that functionality. The `'GenerateLayer'` option specifies the type of layer as follows.

- `'custom-layer'` — The software generates a custom layer that performs the operation.
- `'placeholder-layer'` — The software generates a `PlaceholderLayer` object. To create a working network in this case, see “Define Custom Deep Learning Layers” or “Define Network as Model Function”.

Example: `'GenerateLayer','placeholder-layer'`

`CustomLayerPrefix` — Prefix for generated custom layers
`'customLayer'` (default) | char vector

Prefix for generate custom layers, specified as a char vector.

This option applies only when the `'GenerateLayer'` option is `'custom-layer'`. The name of each generated custom layer starts with the specified prefix.

Example: `'CustomLayerPrefix','myGeneratedLayer'`

Output Arguments

`lgraph` — Layer graph
`LayerGraph` object

Layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `findPlaceholderLayers` | `PlaceholderLayer` | `dlarray`

Topics
“List of Functions with dlarray Support”

Introduced in R2019b

getL2Factor

Package: `nnet.cnn.layer`

Get L2 regularization factor of layer learnable parameter

Syntax

```
factor = getL2Factor(layer,parameterName)
factor = getL2Factor(layer,parameterPath)

factor = getL2Factor(dlnet,layerName,parameterName)
factor = getL2Factor(dlnet,parameterPath)
```

Description

`factor = getL2Factor(layer,parameterName)` returns the L2 regularization factor of the parameter with the name `parameterName` in `layer`.

For built-in layers, you can get the L2 regularization factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `factor = getL2Factor(layer,'Weights')` is equivalent to `factor = layer.WeightL2Factor`.

`factor = getL2Factor(layer,parameterPath)` returns the L2 regularization factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a `dlnetwork` object in a custom layer.

`factor = getL2Factor(dlnet,layerName,parameterName)` returns the L2 regularization factor of the parameter with the name `parameterName` in the layer with name `layerName` for the specified `dlnetwork` object.

`factor = getL2Factor(dlnet,parameterPath)` returns the L2 regularization factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a nested layer.

Examples

Set and Get L2 Regularization Factor of Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a layer.

Create a layer array containing the custom layer `preluLayer`, attached to this is example as a supporting file. To access this layer, open this example as a live script.

Create a layer array including a custom layer `preluLayer`.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
```

```
    preluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the L2 regularization factor of the `Alpha` learnable parameter of the `preluLayer` to 2.

```
layers(4) = setL2Factor(layers(4),"Alpha",2);
```

View the updated L2 regularization factor.

```
factor = getL2Factor(layers(4),"Alpha")
```

```
factor = 2
```

Set and Get L2 Regularization Factor of Nested Layer Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a nested layer.

Create a residual block layer using the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
numFilters = 64;
layer = residualBlockLayer(numFilters)
```

```
layer =
  residualBlockLayer with properties:

       Name: ''

   Learnable Parameters
    Network: [1x1 dlnetwork]

   State Parameters
    No properties.

  Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     2   'gn1'     Group Normalization   Group normalization
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     5   'gn2'     Group Normalization   Group normalization
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the L2 regularization factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setL2Factor` function.

```
factor = 2;
layer = setL2Factor(layer,'Network/conv1/Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(layer,'Network/conv1/Weights')
```

```
factor = 2
```

Set and Get L2 Regularization Factor of `dlnetwork` Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a `dlnetwork` object.

Create a `dlnetwork` object.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')
    convolution2dLayer(5,20,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

lgraph = layerGraph(layers);

dlnet = dlnetwork(lgraph);
```

Set the L2 regularization factor of the `'Weights'` learnable parameter of the convolution layer to 2 using the `setL2Factor` function.

```
factor = 2;
dlnet = setL2Factor(dlnet,'conv','Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(dlnet,'conv','Weights')
```

```
factor = 2
```

Set and Get L2 Regularization Factor of Nested `dlnetwork` Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a nested layer in a `dlnetwork` object.

Create a `dlnetwork` object containing the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
inputSize = [224 224 3];
numFilters = 32;
numClasses = 5;

layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    convolution2dLayer(7,numFilters,'Stride',2,'Padding','same','Name','conv')
    groupNormalizationLayer('all-channels','Name','gn')
    reluLayer('Name','relu')
    maxPooling2dLayer(3,'Stride',2,'Name','max')
    residualBlockLayer(numFilters,'Name','res1')
    residualBlockLayer(numFilters,'Name','res2')
    residualBlockLayer(2*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res3')
    residualBlockLayer(2*numFilters,'Name','res4')
    residualBlockLayer(4*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res5')
    residualBlockLayer(4*numFilters,'Name','res6')
    globalAveragePooling2dLayer('Name','gap')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')];

dlnet = dlnetwork(layers);
```

The `Learnables` property of the `dlnetwork` object is a table that contains the learnable parameters of the network. The table includes parameters of nested layers in separate rows. View the learnable parameters of the layer `"res1"`.

```
learnables = dlnet.Learnables;
idx = learnables.Layer == "res1";
learnables(idx,:)
```

```
ans=8×3 table
    Layer            Parameter                 Value
    _____    ________________________    ___________________

    "res1"    "Network/conv1/Weights"    {3x3x32x32 dlarray}
    "res1"    "Network/conv1/Bias"       {1x1x32    dlarray}
    "res1"    "Network/gn1/Offset"       {1x1x32    dlarray}
    "res1"    "Network/gn1/Scale"        {1x1x32    dlarray}
    "res1"    "Network/conv2/Weights"    {3x3x32x32 dlarray}
    "res1"    "Network/conv2/Bias"       {1x1x32    dlarray}
    "res1"    "Network/gn2/Offset"       {1x1x32    dlarray}
    "res1"    "Network/gn2/Scale"        {1x1x32    dlarray}
```

For the layer `"res1"`, set the L2 regularization factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setL2Factor` function.

```
factor = 2;
dlnet = setL2Factor(dlnet,'res1/Network/conv1/Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(dlnet,'res1/Network/conv1/Weights')
```

```
factor = 2
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name
character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

`parameterPath` — Path to parameter in nested layer
string scalar | character vector

Path to parameter in nested layer, specified as a string scalar or a character vector. A nested layer is a custom layer that itself defines a layer graph as a learnable parameter.

If the input to `getL2Factor` is a nested layer, then the parameter path has the form `"propertyName/layerName/parameterName"`, where:

- `propertyName` is the name of the property containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"propertyName1/layerName1/.../propertyNameN/layerNameN/parameterName"`, where `propertyName1` and `layerName1` correspond to the layer in the input to the `getL2Factor` function, and the subsequent parts correspond to the deeper levels.

Example: For layer input to `getL2Factor`, the path `"Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`.

If the input to `getL2Factor` is a `dlnetwork` object and the desired parameter is in a nested layer, then the parameter path has the form `"layerName1/propertyName/layerName/parameterName"`, where:

- `layerName1` is the name of the layer in the input `dlnetwork` object
- `propertyName` is the property of the layer containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"layerName1/propertyName1/.../layerNameN/propertyNameN/layerName/parameterName"`, where `layerName1` and `propertyName1` correspond to the layer in the input to the `getL2Factor` function, and the subsequent parts correspond to the deeper levels.

Example: For `dlnetwork` input to `getL2Factor`, the path `"res1/Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`, where `layer` is the layer with name `"res1"` in the input network `dlnet`.

Data Types: `char` | `string`

`dlnet` — Network for custom training loops
`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`layerName` — Layer name
string scalar | character vector

Layer name, specified as a string scalar or a character vector.

Data Types: `char` | `string`

Output Arguments

`factor` — L2 regularization factor

nonnegative scalar

L2 regularization factor for the parameter, returned as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the specified parameter. For example, if `factor` is 2, then the L2 regularization for the specified parameter is twice the current global L2 regularization. The software determines the global L2 regularization based on the settings specified with the `trainingOptions` function.

See Also

`setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `trainNetwork` | `trainingOptions`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b

getLearnRateFactor

Package: `nnet.cnn.layer`

Get learn rate factor of layer learnable parameter

Syntax

```
factor = getLearnRateFactor(layer,parameterName)
factor = getLearnRateFactor(layer,parameterPath)

factor = getLearnRateFactor(dlnet,layerName,parameterName)
factor = getLearnRateFactor(dlnet,parameterPath)
```

Description

`factor = getLearnRateFactor(layer,parameterName)` returns the learn rate factor of the learnable parameter with the name `parameterName` in `layer`.

For built-in layers, you can get the learn rate factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `factor = getLearnRateFactor(layer,'Weights')` is equivalent to `factor = layer.WeightLearnRateFactor`.

`factor = getLearnRateFactor(layer,parameterPath)` returns the learn rate factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a `dlnetwork` object in a custom layer.

`factor = getLearnRateFactor(dlnet,layerName,parameterName)` returns the learn rate factor of the parameter with the name `parameterName` in the layer with name `layerName` for the specified `dlnetwork` object.

`factor = getLearnRateFactor(dlnet,parameterPath)` returns the learn rate factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a nested layer.

Examples

Set and Get Learning Rate Factor of Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a custom PReLU layer.

Create a layer array containing the custom layer `preluLayer`, attached to this is example as a supporting file. To access this layer, open this example as a live script.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    preluLayer(20)
```

```
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the learn rate factor of the `Alpha` learnable parameter of the `preluLayer` to 2.

```
layers(4) = setLearnRateFactor(layers(4),"Alpha",2);
```

View the updated learn rate factor.

```
factor = getLearnRateFactor(layers(4),"Alpha")
```

```
factor = 2
```

Set and Get Learning Rate Factor of Nested Layer Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a nested layer.

Create a residual block layer using the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
numFilters = 64;
layer = residualBlockLayer(numFilters)
```

```
layer =
  residualBlockLayer with properties:

       Name: ''

   Learnable Parameters
    Network: [1x1 dlnetwork]

   State Parameters
    No properties.

  Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     2   'gn1'     Group Normalization   Group normalization
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     5   'gn2'     Group Normalization   Group normalization
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the learning rate factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setLearnRateFactor` function.

```
factor = 2;
layer = setLearnRateFactor(layer,'Network/conv1/Weights',factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(layer,'Network/conv1/Weights')
```

```
factor = 2
```

Set and Get Learn Rate Factor of `dlnetwork` Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a `dlnetwork` object.

Create a `dlnetwork` object.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')
    convolution2dLayer(5,20,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

lgraph = layerGraph(layers);

dlnet = dlnetwork(lgraph);
```

Set the learn rate factor of the `'Weights'` learnable parameter of the convolution layer to 2 using the `setLearnRateFactor` function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'conv','Weights',factor);
```

Get the updated learn rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(dlnet,'conv','Weights')
```

```
factor = 2
```

Set and Get Learning Rate Factor of Nested `dlnetwork` Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a nested layer in a `dlnetwork` object.

Create a `dlnetwork` object containing the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
inputSize = [224 224 3];
numFilters = 32;
numClasses = 5;

layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    convolution2dLayer(7,numFilters,'Stride',2,'Padding','same','Name','conv')
    groupNormalizationLayer('all-channels','Name','gn')
    reluLayer('Name','relu')
    maxPooling2dLayer(3,'Stride',2,'Name','max')
    residualBlockLayer(numFilters,'Name','res1')
    residualBlockLayer(numFilters,'Name','res2')
    residualBlockLayer(2*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res3')
    residualBlockLayer(2*numFilters,'Name','res4')
    residualBlockLayer(4*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res5')
    residualBlockLayer(4*numFilters,'Name','res6')
    globalAveragePooling2dLayer('Name','gap')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')];

dlnet = dlnetwork(layers);
```

View the layers of the nested network in the layer `'res1'`.

```
dlnet.Layers(6).Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           32 3x3x32 convolutions with stride [1  1] and padding 's
     2   'gn1'     Group Normalization   Group normalization with 32 channels split into 1 groups
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           32 3x3x32 convolutions with stride [1  1] and padding 's
     5   'gn2'     Group Normalization   Group normalization with 32 channels split into 32 group
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the learning rate factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setLearnRateFactor` function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'res1/Network/conv1/Weights',factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(dlnet,'res1/Network/conv1/Weights')
```

```
factor = 2
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name

character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

`parameterPath` — Path to parameter in nested layer

string scalar | character vector

Path to parameter in nested layer, specified as a string scalar or a character vector. A nested layer is a custom layer that itself defines a layer graph as a learnable parameter.

If the input to `getLearnRateFactor` is a nested layer, then the parameter path has the form `"propertyName/layerName/parameterName"`, where:

- `propertyName` is the name of the property containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"propertyName1/layerName1/.../propertyNameN/layerNameN/parameterName"`, where `propertyName1` and `layerName1` correspond to the layer in the input to the `getLearnRateFactor` function, and the subsequent parts correspond to the deeper levels.

Example: For layer input to `getLearnRateFactor`, the path `"Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`.

If the input to `getLearnRateFactor` is a `dlnetwork` object and the desired parameter is in a nested layer, then the parameter path has the form `"layerName1/propertyName/layerName/parameterName"`, where:

- `layerName1` is the name of the layer in the input `dlnetwork` object
- `propertyName` is the property of the layer containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"layerName1/propertyName1/.../layerNameN/propertyNameN/layerName/parameterName"`, where `layerName1` and `propertyName1` correspond to the layer in the input to the `getLearnRateFactor` function, and the subsequent parts correspond to the deeper levels.

Example: For `dlnetwork` input to `getLearnRateFactor`, the path `"res1/Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`, where `layer` is the layer with name `"res1"` in the input network `dlnet`.

Data Types: `char` | `string`

`dlnet` — Network for custom training loops
`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`layerName` — Layer name
string scalar | character vector

Layer name, specified as a string scalar or a character vector.

Data Types: `char` | `string`

Output Arguments

`factor` — Learning rate factor

nonnegative scalar

Learning rate factor for the parameter, returned as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the specified parameter. For example, if `factor` is 2, then the learning rate for the specified parameter is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

See Also

`setLearnRateFactor` | `setL2Factor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b

globalAveragePooling1dLayer

1-D global average pooling layer

Description

A 1-D global average pooling layer performs downsampling by outputting the average of the time or spatial dimensions of the input.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Creation

Syntax

```
layer = globalAveragePooling1dLayer
layer = globalAveragePooling1dLayer(Name=name)
```

Description

`layer = globalAveragePooling1dLayer` creates a 1-D global average pooling layer.

`layer = globalAveragePooling1dLayer(Name=name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 1-D Global Average Pooling Layer

Create a 1-D global average pooling layer.

```
layer = globalMaxPooling1dLayer
```

```
layer =
  GlobalMaxPooling1DLayer with properties:

    Name: ''
```

Include a 1-D global average pooling layer in a layer array.

```
layers = [
    sequenceInputLayer(12)
    convolution1dLayer(11,96)
    reluLayer
    globalAveragePooling1dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:
```

```
    1   ''   Sequence Input               Sequence input with 12 dimensions
    2   ''   Convolution                  96 11 convolutions with stride 1 and padding [0  0]
    3   ''   ReLU                         ReLU
    4   ''   1-D Global Average Pooling   1-D global average pooling
    5   ''   Fully Connected              10 fully connected layer
    6   ''   Softmax                      softmax
    7   ''   Classification Output        crossentropyex
```

Algorithms

1-D Global Average Pooling Layer

A 1-D global average pooling layer performs downsampling by outputting the average of the time or spatial dimensions of the input.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `GlobalAveragePooling1DLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	Output Format
`"SCB"` (spatial, channel, batch)	`"SCB"` (spatial, channel, batch)
`"CBT"` (channel, batch, time)	`"CB"` (channel, batch)
`"SCBT"` (spatial, channel, batch, time)	`"SCBT"` (spatial, channel, batch, time)

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer`

Topics

“Sequence Classification Using 1-D Convolutions”

“Sequence-to-Sequence Classification Using 1-D Convolutions”

“Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2021b

globalAveragePooling2dLayer

Global average pooling layer

Description

A 2-D global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input.

Creation

Syntax

```
layer = globalAveragePooling2dLayer
layer = globalAveragePooling2dLayer('Name',name)
```

Description

`layer = globalAveragePooling2dLayer` creates a global average pooling layer.

`layer = globalAveragePooling2dLayer('Name',name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Global Average Pooling Layer

Create a global average pooling layer with the name `'gap1'`.

```
layer = globalAveragePooling2dLayer('Name','gap1')
```

```
layer =
  GlobalAveragePooling2DLayer with properties:

    Name: 'gap1'
```

Include a global average pooling layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    globalAveragePooling2dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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
     3   ''   ReLU                        ReLU
     4   ''   2-D Global Average Pooling   2-D global average pooling
     5   ''   Fully Connected             10 fully connected layer
     6   ''   Softmax                     softmax
     7   ''   Classification Output       crossentropyex
```

Tips

- In an image classification network, you can use a `globalAveragePooling2dLayer` before the final fully connected layer to reduce the size of the activations without sacrificing performance. The reduced size of the activations means that the downstream fully connected layers will have fewer weights, reducing the size of your network.
- You can use a `globalAveragePooling2dLayer` towards the end of a classification network instead of a `fullyConnectedLayer`. Since global pooling layers have no learnable parameters, they can be less prone to overfitting and can reduce the size of the network. These networks can also be more robust to spatial translations of input data. You can also replace a fully connected layer with a `globalMaxPooling2dLayer` instead. Whether a `globalMaxPooling2dLayer` or a `globalAveragePooling2dLayer` is more appropriate depends on your data set.

  To use a global average pooling layer instead of a fully connected layer, the size of the input to `globalAveragePooling2dLayer` must match the number of classes in the classification problem

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`convolution2dLayer` | `averagePooling2dLayer` | `maxPooling2dLayer` | `globalAveragePooling3dLayer` | `globalMaxPooling2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b

globalAveragePooling3dLayer

3-D global average pooling layer

Description

A 3-D global average pooling layer performs downsampling by computing the mean of the height, width, and depth dimensions of the input.

Creation

Syntax

```
layer = globalAveragePooling3dLayer
layer = globalAveragePooling3dLayer('Name',name)
```

Description

`layer = globalAveragePooling3dLayer` creates a 3-D global average pooling layer.

`layer = globalAveragePooling3dLayer('Name',name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 3-D Global Average Pooling Layer

Create a 3-D global average pooling layer with the name `'gap1'`.

```
layer = globalAveragePooling3dLayer('Name','gap1')
```

```
layer =
  GlobalAveragePooling3DLayer with properties:

    Name: 'gap1'
```

Include a 3-D global average pooling layer in a `Layer` array.

```
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    globalAveragePooling3dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input             28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution                 20 5x5x5 convolutions with stride [1  1  1] and paddi
     3   ''   ReLU                        ReLU
     4   ''   3-D Global Average Pooling   3-D global average pooling
     5   ''   Fully Connected             10 fully connected layer
     6   ''   Softmax                     softmax
     7   ''   Classification Output       crossentropyex
```

Tips

- In an image classification network, you can use a `globalAveragePooling3dLayer` before the final fully connected layer to reduce the size of the activations without sacrificing performance. The reduced size of the activations means that the downstream fully connected layers will have fewer weights, reducing the size of your network.
- You can use a `globalAveragePooling3dLayer` towards the end of a classification network instead of a `fullyConnectedLayer`. Since global pooling layers have no learnable parameters, they can be less prone to overfitting and can reduce the size of the network. These networks can also be more robust to spatial translations of input data. You can also replace a fully connected layer with a `globalMaxPooling3dLayer` instead. Whether a `globalMaxPooling3dLayer` or a `globalAveragePooling3dLayer` is more appropriate depends on your data set.

  To use a global average pooling layer instead of a fully connected layer, the size of the input to `globalAveragePooling3dLayer` must match the number of classes in the classification problem

See Also

`averagePooling3dLayer` | `globalAveragePooling2dLayer` | `convolution3dLayer` | `maxPooling3dLayer` | `globalMaxPooling3dLayer`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b

globalMaxPooling1dLayer

1-D global max pooling layer

Description

A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Creation

Syntax

```
layer = globalMaxPooling1dLayer
layer = globalMaxPooling1dLayer(Name=name)
```

Description

`layer = globalMaxPooling1dLayer` creates a 1-D global max pooling layer.

`layer = globalMaxPooling1dLayer(Name=name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 1-D Global Max Pooling Layer

Create a 1-D global max pooling layer.

```
layer = globalMaxPooling1dLayer
```

```
layer =
  GlobalMaxPooling1DLayer with properties:

    Name: ''
```

Include a 1-D global max pooling layer in a layer array.

```
layers = [
    sequenceInputLayer(12)
    convolution1dLayer(11,96)
    reluLayer
    globalMaxPooling1dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:
```

```
    1   ''   Sequence Input           Sequence input with 12 dimensions
    2   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
    3   ''   ReLU                     ReLU
    4   ''   1-D Global Max Pooling   1-D global max pooling
    5   ''   Fully Connected          10 fully connected layer
    6   ''   Softmax                  softmax
    7   ''   Classification Output    crossentropyex
```

Algorithms

1-D Global Max Pooling Layer

A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `GlobalMaxPooling1DLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	Output Format
`"SCB"` (spatial, channel, batch)	`"SCB"` (spatial, channel, batch)
`"CBT"` (channel, batch, time)	`"CB"` (channel, batch)
`"SCBT"` (spatial, channel, batch, time)	`"SCBT"` (spatial, channel, batch, time)

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalAveragePooling1dLayer`

Topics

“Sequence Classification Using 1-D Convolutions”

“Sequence-to-Sequence Classification Using 1-D Convolutions”

“Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2021b

globalMaxPooling2dLayer

Global max pooling layer

Description

A 2-D global max pooling layer performs downsampling by computing the maximum of the height and width dimensions of the input.

Creation

Syntax

```
layer = globalMaxPooling2dLayer
layer = globalMaxPooling2dLayer('Name',name)
```

Description

`layer = globalMaxPooling2dLayer` creates a global max pooling layer.

`layer = globalMaxPooling2dLayer('Name',name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

NumOutputs — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

OutputNames — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Object Functions

Examples

Create Global Max Pooling Layer

Create a global max pooling layer with the name `'gmp1'`.

```
layer = globalMaxPooling2dLayer('Name','gmp1')
```

```
layer =
  GlobalMaxPooling2DLayer with properties:

    Name: 'gmp1'
```

Include a global max pooling layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    globalMaxPooling2dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
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   ''   ReLU                    ReLU
     4   ''   2-D Global Max Pooling   2-D global max pooling
     5   ''   Fully Connected          10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output    crossentropyex
```

Tips

- In an image classification network, you can use a `globalMaxPooling2dLayer` before the final fully connected layer to reduce the size of the activations without sacrificing performance. The reduced size of the activations means that the downstream fully connected layers will have fewer weights, reducing the size of your network.
- You can use a `globalMaxPooling2dLayer` towards the end of a classification network instead of a `fullyConnectedLayer`. Since global pooling layers have no learnable parameters, they can be less prone to overfitting and can reduce the size of the network. These networks can also be more robust to spatial translations of input data. You can also replace a fully connected layer with a `globalAveragePooling2dLayer` instead. Whether a `globalAveragePooling2dLayer` or a `globalMaxPooling2dLayer` is more appropriate depends on your data set.

  To use a global average pooling layer instead of a fully connected layer, the size of the input to `globalMaxPooling2dLayer` must match the number of classes in the classification problem

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`convolution2dLayer` | `averagePooling2dLayer` | `maxPooling2dLayer` | `globalMaxPooling3dLayer` | `globalAveragePooling2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2020a

globalMaxPooling3dLayer

3-D global max pooling layer

Description

A 3-D global max pooling layer performs downsampling by computing the maximum of the height, width, and depth dimensions of the input.

Creation

Syntax

```
layer = globalMaxPooling3dLayer
layer = globalMaxPooling3dLayer('Name',name)
```

Description

`layer = globalMaxPooling3dLayer` creates a 3-D global max pooling layer.

`layer = globalMaxPooling3dLayer('Name',name)` sets the optional `Name` property.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Object Functions

Examples

Create 3-D Global Max Pooling Layer

Create a 3-D global max pooling layer with name `'gmp1'`.

```
layer = globalMaxPooling3dLayer('Name','gmp1')
```

```
layer =
  GlobalMaxPooling3DLayer with properties:

    Name: 'gmp1'
```

Include a 3-D max pooling layer in a `Layer` array.

```
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    globalMaxPooling3dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input          28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution              20 5x5x5 convolutions with stride [1  1  1] and padding [0
     3   ''   ReLU                     ReLU
     4   ''   3-D Global Max Pooling   3-D global max pooling
     5   ''   Fully Connected          10 fully connected layer
     6   ''   Softmax                  softmax
     7   ''   Classification Output    crossentropyex
```

Tips

- In an image classification network, you can use a `globalMaxPooling3dLayer` before the final fully connected layer to reduce the size of the activations without sacrificing performance. The reduced size of the activations means that the downstream fully connected layers will have fewer weights, reducing the size of your network.
- You can use a `globalMaxPooling3dLayer` towards the end of a classification network instead of a `fullyConnectedLayer`. Since global pooling layers have no learnable parameters, they can be less prone to overfitting and can reduce the size of the network. These networks can also be more robust to spatial translations of input data. You can also replace a fully connected layer with a `globalAveragePooling3dLayer` instead. Whether a `globalAveragPooling3dLayer` or a `globalMaxPooling3dLayer` is more appropriate depends on your data set.

  To use a global average pooling layer instead of a fully connected layer, the size of the input to `globalMaxPooling3dLayer` must match the number of classes in the classification problem

See Also

`averagePooling3dLayer` | `globalMaxPooling2dLayer` | `convolution3dLayer` | `maxPooling3dLayer` | `globalAveragePooling3dLayer`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2020a

googlenet

GoogLeNet convolutional neural network

Syntax

```
net = googlenet
net = googlenet('Weights',weights)

lgraph = googlenet('Weights','none')
```

Description

GoogLeNet is a convolutional neural network that is 22 layers deep. You can load a pretrained version of the network trained on either the ImageNet [1] or Places365 [2] [3] data sets. The network trained on ImageNet classifies images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The network trained on Places365 is similar to the network trained on ImageNet, but classifies images into 365 different place categories, such as field, park, runway, and lobby. These networks have learned different feature representations for a wide range of images. The pretrained networks both have an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

To classify new images using GoogLeNet, use `classify`. For an example, see “Classify Image Using GoogLeNet”.

You can retrain a GoogLeNet network to perform a new task using transfer learning. When performing transfer learning, the most common approach is to use networks pretrained on the ImageNet data set. If the new task is similar to classifying scenes, then using the network trained on Places-365 can give higher accuracies. For an example showing how to retrain GoogLeNet on a new classification task, see “Train Deep Learning Network to Classify New Images”

`net = googlenet` returns a GoogLeNet network trained on the ImageNet data set.

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.

`net = googlenet('Weights',weights)` returns a GoogLeNet network trained on either the ImageNet or Places365 data set. The syntax `googlenet('Weights','imagenet')` (default) is equivalent to `googlenet`.

The network trained on ImageNet requires the Deep Learning Toolbox Model *for GoogLeNet Network* support package. The network trained on Places365 requires the Deep Learning Toolbox Model *for Places365-GoogLeNet Network* support package. If the required support package is not installed, then the function provides a download link.

`lgraph = googlenet('Weights','none')` returns the untrained GoogLeNet network architecture. The untrained model does not require the support package.

Examples

Download GoogLeNet Support Package

Download and install the Deep Learning Toolbox Model *for GoogLeNet Network* support package.

Type `googlenet` at the command line.

```
googlenet
```

If the Deep Learning Toolbox Model *for GoogLeNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `googlenet` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

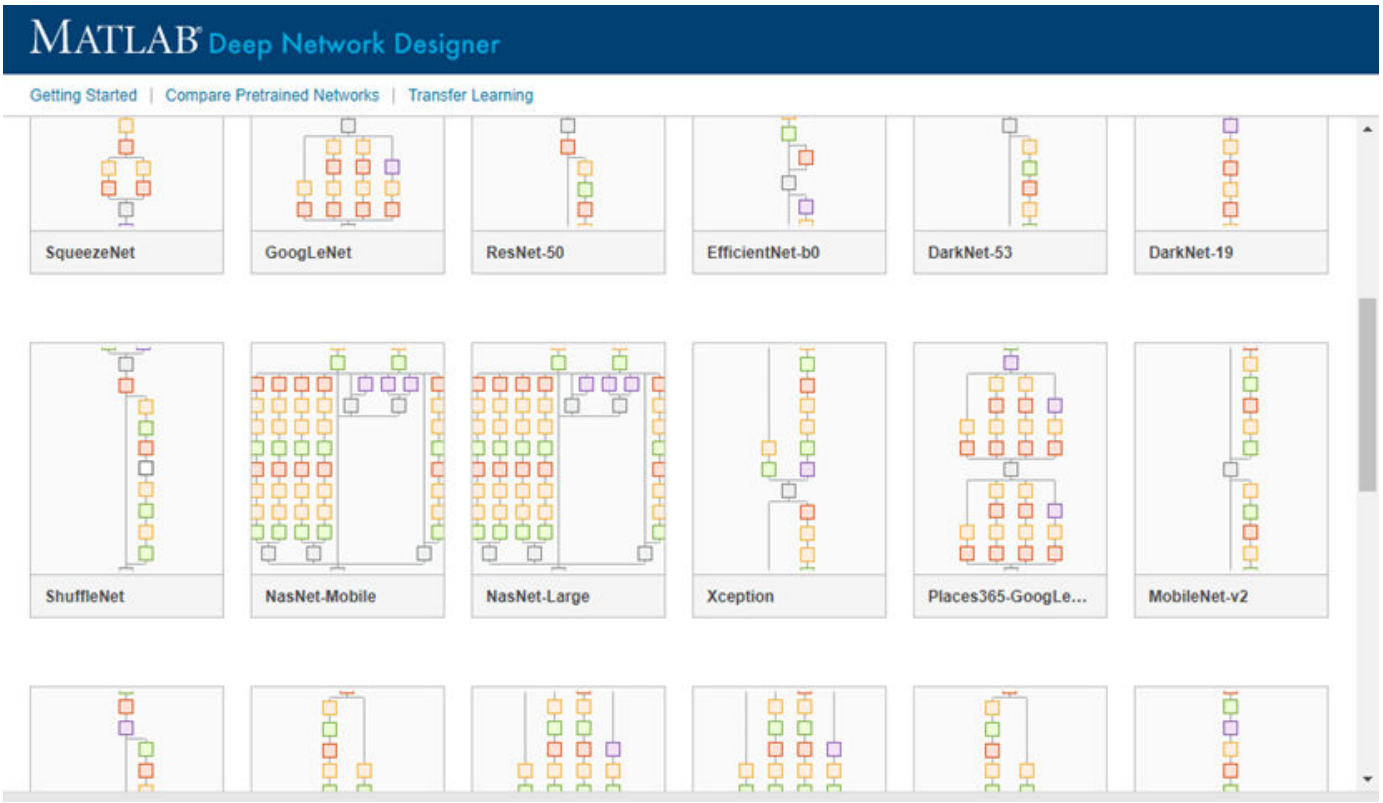
```
googlenet
```

```
ans =

  DAGNetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(googlenet)
```

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



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

Input Arguments

`weights` — Source of network parameters

`'imagenet'` (default) | `'places365'` | `'none'`

Source of network parameters, specified as `'imagenet'` ,`'places365'`, or `'none'`.

- If `weights` equals `'imagenet'`, then the network has weights trained on the ImageNet data set.
- If `weights` equals `'places365'`, then the network has weights trained on the Places365 data set.
- If `weights` equals `'none'`, then the untrained network architecture is returned.

Example: `'places365'`

Output Arguments

`net` — Pretrained GoogLeNet convolutional neural network

`DAGNetwork` object

Pretrained GoogLeNet convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained GoogLeNet convolutional neural network architecture

`LayerGraph` object

Untrained GoogLeNet convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] 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).

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

[4] 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.

[5] *BVLC GoogLeNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = googlenet` or by passing the `googlenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('googlenet')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `googlenet('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = googlenet` or by passing the `googlenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('googlenet')`.

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `googlenet('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `resnet18` | `resnet50` | `resnet101` | `densenet201` | `inceptionresnetv2` | `squeezenet` | `trainNetwork` | `layerGraph` | `inceptionv3` | `DAGNetwork`

Topics
“Transfer Learning with Deep Network Designer”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Train Residual Network for Image Classification”

Introduced in R2017b

gradCAM

Explain network predictions using Grad-CAM

Syntax

```
scoreMap = gradCAM(net,X,label)
scoreMap = gradCAM(net,X,reductionFcn)
[scoreMap,featureLayer,reductionLayer] = gradCAM( ___ )
 ___  = gradCAM( ___ ,Name,Value)
```

Description

`scoreMap = gradCAM(net,X,label)` returns the gradient-weighted class activation mapping (Grad-CAM) map of the change in the classification score of image `X`, when the network `net` evaluates the class score for the class given by `label`. Use this function to explain network predictions and check that your network is focusing on the right parts of an image.

The Grad-CAM interpretability technique uses the gradients of the classification score with respect to the final convolutional feature map. The parts of an image with a large value for the Grad-CAM map are those that most impact the network score for that class.

Use this syntax to compute the Grad-CAM map for image or pixel classification tasks.

`scoreMap = gradCAM(net,X,reductionFcn)` returns the Grad-CAM importance map using a reduction function. `reductionFcn` is a function handle that reduces the output activations of the reduction layer to a scalar value. This scalar fulfills the role of the class score for classification tasks, and generalizes the Grad-CAM technique to nonclassification tasks, such as regression.

The `gradCAM` function computes the Grad-CAM map by differentiating the reduced output of the reduction layer with respect to the features in the feature layer. `gradCAM` automatically selects reduction and feature layers to use when computing the map. To specify these layers, use the `'ReductionLayer'` and `'FeatureLayer'` name-value arguments.

Use this syntax to compute the Grad-CAM map for nonclassification tasks.

`[scoreMap,featureLayer,reductionLayer] = gradCAM( ___ )` also returns the names of the feature layer and reduction layer used to compute the Grad-CAM map. Use this syntax with any of the input-argument combinations in previous syntaxes.

`___  = gradCAM( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, `'ReductionLayer','prob'` sets the reduction layer to the `net` layer named `'prob'`.

Examples

Explore Network Classifications Using Grad-CAM

Use `gradCAM` to visualize which parts of an image are important to the classification decision of a network.

Import the pretrained network SqueezeNet.

```
net = squeezenet;
```

Import the image and resize it to match the input size for the network.

```
X = imread("laika_grass.jpg");
inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Display the image.

```
imshow(X)
```

Classify the image to get the class label.

```
label = classify(net,X)
```

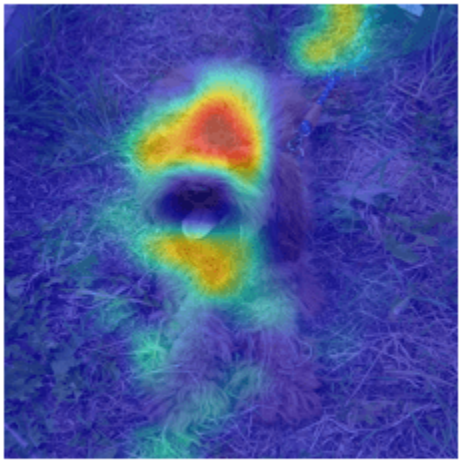
```
label = categorical
     toy poodle
```

Use `gradCAM` to determine which parts of the image are important to the classification result.

```
scoreMap = gradCAM(net,X,label);
```

Plot the result over the original image with transparency to see which areas of the image contribute most to the classification score.

```
figure
imshow(X)
hold on
imagesc(scoreMap,'AlphaData',0.5)
colormap jet
```



The network focuses predominantly on the back of the dog to make the classification decision.

Compute Grad-CAM Map for Image Regression Network

Use Grad-CAM to visualize which parts of an image are most important to the predictions of an image regression network.

Load the sample data, which consists of synthetic images of handwritten digits. The third output contains the corresponding angles of rotation of the digits, in degrees.

```
rng default
[XTrain,~,YTrain] = digitTrain4DArrayData;
[XTest,~,YTest] = digitTest4DArrayData;

numTrainImages = numel(YTrain);
idx = randperm(numTrainImages,20);
```

Construct an image regression network that can predict the rotation of an image.

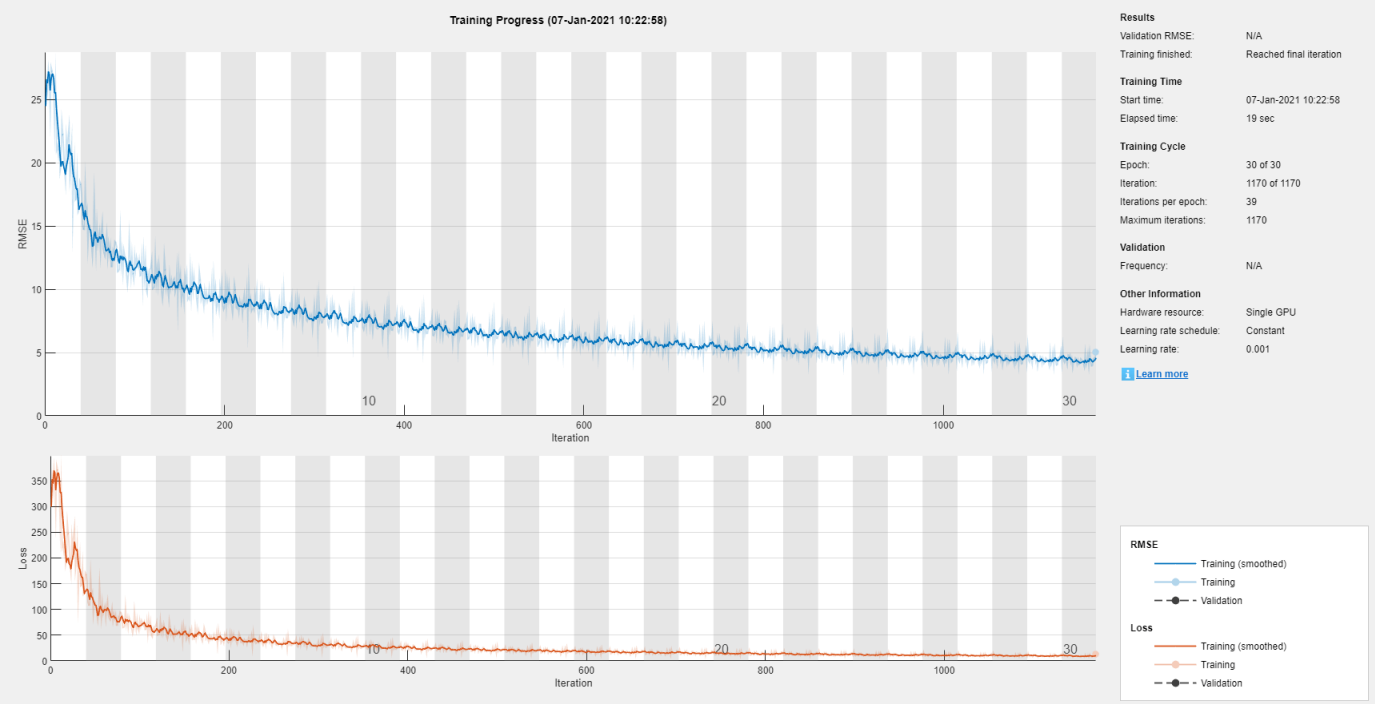
```
layers = [ ...
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(12,25,'Name','conv')
    reluLayer('Name','relu')
    fullyConnectedLayer(1,'Name','fc')
    regressionLayer('Name','output')];
```

Specify the training options.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

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



Evaluate the performance of the network on a test image.

```
testDigit = XTest(:,:,:,idx(4));
```

Use `predict` to predict the angle of rotation and compare the predicted rotation to the true rotation.

```
predRotation = predict(net,testDigit)
```

```
predRotation = single
    -47.5497
```

```
trueRotation = YTest(idx(4))
```

```
trueRotation = -40
```

Visualize the regions of the image most important to the network prediction using `gradCAM`. Select the ReLU layer as the feature layer and the fully connected layer as the reduction layer.

```
featureLayer = 'relu';
reductionLayer = 'fc';
```

Define the reduction function. The reduction function must reduce the output of the reduction layer to a scalar value. The Grad-CAM map displays the importance of different parts of the image to that scalar. In this regression problem, the network predicts the angle of rotation of the image. Therefore, the output of the fully connected layer is already a scalar value and so the reduction function is just the identity function.

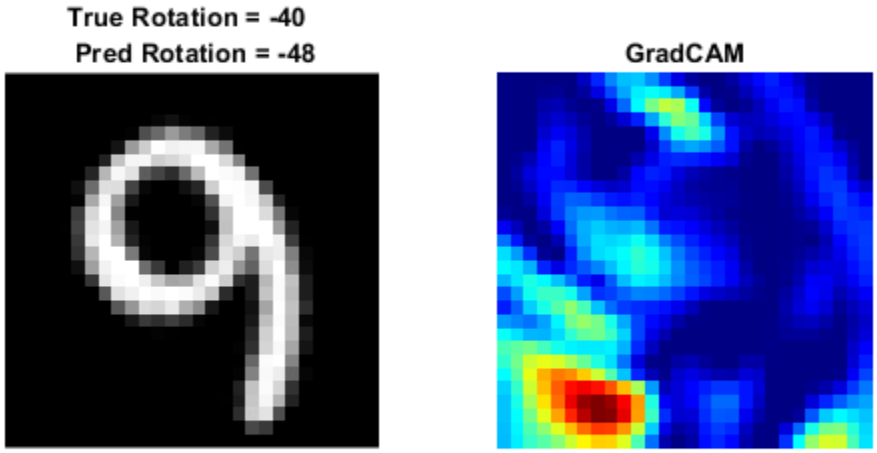
```
reductionFcn = @(x)x;
```

Compute the Grad-CAM map.

```
scoreMap = gradCAM(net,testDigit,reductionFcn, ...
    'ReductionLayer',reductionLayer, ...
    'FeatureLayer',featureLayer);
```

Display the Grad-CAM map over the test image.

```
ax(1) = subplot(1,2,1);
imshow(testDigit)
title("True Rotation = " + trueRotation + '\newline Pred Rotation = ' + round(predRotation,0))
colormap(ax(1),'gray')

ax(2) = subplot(1,2,2);
imshow(testDigit)
hold on
imagesc(scoreMap)
colormap(ax(2),'jet')
title("GradCAM")
hold off
```



The Grad-CAM map shows that the network is focusing on the area in the bottom left, which is where the tail of the digit would be if the image had zero rotation. The map suggests that to predict the negative rotation, the network is using the empty space.

Input Arguments

net — Trained network

SeriesNetwork | DAGNetwork | dlnetwork

Trained network, specified as a SeriesNetwork, DAGNetwork, or dlnetwork object. You can get a trained network by importing a pretrained network or by training your own network using the trainNetwork function or custom training. For more information about pretrained networks, see "Pretrained Deep Neural Networks".

net must contain a single input layer and a single output layer. The input layer of net must be an imageInputLayer or an image3dInputLayer.

X — Input data

numeric array | dlarray

Input data, specified as a numeric array or formatted dlarray object.

X must have size equal to the input size of the network.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

label — Class label

categorical | character vector | string scalar | numeric index

Class label to use for calculating the Grad-CAM map for image classification and semantic segmentation tasks, specified as a categorical, a character vector, a string scalar, a numeric index, or a vector of these values.

For dlnetwork objects, you must specify label as a categorical or a numeric index.

If you specify label as a vector, the software calculates the feature importance for each class label independently. In that case, scoreMap(:,:,k) corresponds to the map for label(k).

The gradCAM function sums the spatial dimensions of the reduction layer for class label. Therefore, you can specify label as the classes of interest for semantic segmentation tasks and gradCAM returns the Grad-CAM importance for each pixel.

Example: ["cat" "dog"]

Example: [1 5]

Data Types: char | string | categorical

reductionFcn — Reduction function

function handle

Reduction function, specified as a function handle. The reduction function reduces the output activations of the reduction layer to a single value and must reduce a dlarray object to a dlarray scalar. This scalar fulfills the role of label in classification tasks, and generalizes the Grad-CAM technique to nonclassification tasks, such as regression.

Grad-CAM uses the reduced output activations of the reduction layer to compute the gradients for the importance map.

Example: @x(x)

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FeatureLayer','conv10','ReductionLayer','prob','OutputUpsampling','bicubic','ExecutionEnvironment','gpu'` computes the Grad-CAM map with respect to layers `'conv10'` and `'prob'`, executes the calculation on the GPU, and upsamples the resulting map to the same size as the input image using bicubic interpolation.

`FeatureLayer` — Name of feature layer

string | character vector

Name of the feature layer to extract the feature map from when computing the Grad-CAM map, specified as a string or character vector. For most tasks, use the last ReLU layer with nonsingleton spatial dimensions or the last layer that gathers the outputs of ReLU layers (such as depth concatenation or addition layers). If your network does not contain any ReLU layers, specify the name of the final convolutional layer that has nonsingleton spatial dimensions in the output.

The default value is the final layer with nonsingleton spatial dimensions. Use the `analyzeNetwork` function to examine your network and select the correct layer.

Example: `'FeatureLayer','conv10'`

Data Types: `char` | `string`

`ReductionLayer` — Name of reduction layer

string | character vector

Name of the reduction layer to extract output activations from when computing the Grad-CAM map, specified as a string or character vector. For classification tasks, this layer is usually the final softmax layer. For other tasks, this layer is usually the penultimate layer for DAG and series networks and the final layer for `dlnetwork` objects.

The default value is the penultimate layer in DAG and series networks, and the final layer in `dlnetwork` objects. Use the `analyzeNetwork` function to examine your network and select the correct layer.

Example: `'ReductionLayer','prob'`

Data Types: `char` | `string`

`Format` — Data format

character vector | string

Data format assigning a label to each dimension of the input data, specified as a character vector or a string. Each character in the format must be one of the following dimension labels:

- `S` — Spatial
- `C` — Channel
- `B` — Batch

For more information, see `dlarray`.

Example: `'Format','SSC'`

Data Types: `char` | `string`

`OutputUpsampling` — Output upsampling method
`'bicubic'` (default) | `'nearest'` | `'none'`

Output upsampling method, specified as the comma-separated pair consisting of `'OutputUpsampling'` and one of the following values:

- `'bicubic'` — Use bicubic interpolation to produce a smooth map the same size as the input data.
- `'nearest'` — Use nearest-neighbor interpolation to expand the map to the same size as the input data.
- `'none'` — Use no upsampling. The map can be smaller than the input data.

If `'OutputUpsampling'` is `'nearest'` or `'bicubic'`, the computed map is upsampled to the size of the input data using the `imresize` function for 2-D data and the `imresize3` function for 3-D data. For 3-D data, the option `'bicubic'` uses `imresize3` with the `'cubic'` method.

Example: `'OutputUpsampling','bicubic'`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'cpu'` | `'gpu'`

Hardware resource for computing the map, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following.

- `'auto'` — Use the GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.

The GPU option requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose the `'gpu'` option and Parallel Computing Toolbox and a suitable GPU are not available, then the software returns an error.

Example: `'ExecutionEnvironment','gpu'`

Output Arguments

`scoreMap` — Grad-CAM importance map
numeric matrix | numeric array

Grad-CAM importance map, returned as a numeric matrix or a numeric array. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the prediction.

- For classification tasks, `scoreMap` is the gradient of the final classification score for the specified class, with respect to each feature in the feature layer.
- For other types of tasks, `scoreMap` is the gradient of the reduced output of the reduction layer, with respect to each feature in the feature layer.

`scoreMap(i,j)` corresponds to the Grad-CAM importance at the spatial location `(i,j)`. If you provide `label` as a vector of categoricals, character vectors, or strings, then `scoreMap(:,:,k)` corresponds to the map for `label(k)`.

`featureLayer` — Name of feature layer

string

Name of the feature layer to extract the feature map from when computing the Grad-CAM map, returned as a string.

By default, `gradCAM` chooses a feature layer to use to compute the Grad-CAM map. This layer is the final layer with nonsingleton spatial dimensions. You can specify which feature layer to use using the `'FeatureLayer'` name-value argument. When you specify the `'FeatureLayer'` name-value argument, `featureLayer` returns the same value.

`reductionLayer` — Name of reduction layer

string

Name of the reduction layer to extract output activations from when computing the Grad-CAM map, returned as a string.

By default, `gradCAM` chooses a reduction layer to use to compute the Grad-CAM map. This layer is the penultimate layer in DAG and series networks, and the final layer in `dlnetwork` objects. You can also specify which reduction layer to use using the `'ReductionLayer'` name-value argument. When you specify the `'ReductionLayer'` name-value argument, `reductionLayer` returns the same value.

More About

Grad-CAM

Gradient-weighted class activation mapping (Grad-CAM) is an explainability technique that can be used to help understand the predictions made by a deep neural network [1]. Grad-CAM, a generalization of the CAM technique, determines the importance of each neuron in a network prediction by considering the gradients of the target flowing through the deep network.

Grad-CAM computes the gradient of a differentiable output, for example class score, with respect to the convolutional features in the chosen layer. The gradients are spatially pooled to find the neuron importance weights. These weights are then used to linearly combine the activation maps and determine which features are most important to the prediction.

Suppose you have an image classification network with output y^c, representing the score for class c, and want to compute the Grad-CAM map for a convolutional layer with k feature maps (channels), $A^k_{i,j}$, where i,j indexes the pixels. The neuron importance weight is

$$\alpha_k^c = \underbrace{\frac{1}{N}\sum_i\sum_j}_{\text{Global average pooling}} \underbrace{\frac{\partial y^c}{\partial A^k_{i,j}}}_{\substack{\text{Gradients}\\ \text{via}\\ \text{backprop}}},$$

where N is the total number of pixels in the feature map. The Grad-CAM map is then a weighted combination of the feature maps with an applied ReLU:

$$M = \mathrm{ReLU}\left(\sum_k \alpha_k^c A^k\right).$$

The ReLU activation ensures you get only the features that have a positive contribution to the class of interest. The output is therefore a heatmap for the specified class, which is the same size as the feature map. The Grad-CAM map is then upsampled to the size of the input data.

Although Grad-CAM is commonly used for image classification tasks, you can compute a Grad-CAM map for any differentiable activation. For example, for semantic segmentation tasks, you can calculate the Grad-CAM map by replacing y^c with $\sum_{(i,j) \in S} y_{ij}^c$, where S is the set of pixels of interest and $y_{i,j}{}^c$ is 1 if pixel *(i,j)* is predicted to be class c, and 0 otherwise [2]. You can use the `gradCAM` function for nonclassification tasks by specifying a suitable reduction function that reduces the output activations of the reduction layer to a single value and takes the place of y^c in the neuron importance weight equation.

Tips

- The `reductionFcn` function receives the output from the reduction layer as a traced `dlarray` object. The function must reduce this output to a scalar `dlarray`, which `gradCAM` then differentiates with respect to the activations of the feature layer. For example, to compute the Grad-CAM map for channel 208 of the softmax activations of a network, the reduction function is `@(x)(x(208))`. This function receives the activations and extracts the 208th channel.
- The `gradCAM` function automatically chooses reduction and feature layers to use when computing the Grad-CAM map. For some networks, the chosen layers might not be suitable. For example, if your network has multiple layers that can be used as the feature layer, then the function chooses one of those layers, but its choice might not be the most suitable. For such networks, specify which feature layer to use using the `'FeatureLayer'` name-value argument.

References

[1] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization.” 2017 (October 2017): 618–626, https://doi.org/10.1109/ICCV.2017.74.

[2] Vinogradova, Kira, Alexandr Dibrov, and Gene Myers. “Towards Interpretable Semantic Segmentation via Gradient-Weighted Class Activation Mapping.” *Proceedings of the AAAI Conference on Artificial Intelligence* 34, no. 10 (April 2020): 13943–13944, https://doi.org/10.1609/aaai.v34i10.7244.

See Also

`occlusionSensitivity` | `imageLIME` | `activations`

Topics

“Grad-CAM Reveals the Why Behind Deep Learning Decisions”
“Explore Semantic Segmentation Network Using Grad-CAM”
“Understand Network Predictions Using LIME”
“Understand Network Predictions Using Occlusion”
“Investigate Network Predictions Using Class Activation Mapping”

Introduced in R2021a

groupedConvolution2dLayer

2-D grouped convolutional layer

Description

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.

For each group, 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. The layer combines the convolutions for each group independently. If the number of groups is equal to the number of channels, then this layer performs channel-wise convolution.

Creation

Syntax

```
layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup,numGroups)
layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup,'channel-
wise')
layer = groupedConvolution2dLayer( ___ ,Name,Value)
```

Description

`layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup,numGroups)` creates a 2-D grouped convolutional layer and sets the `FilterSize`, `NumFiltersPerGroup`, and `NumGroups` properties.

`layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup,'channel-wise')` creates a layer for channel-wise convolution (also known as depth-wise convolution). In this case, the software determines the `NumGroups` property at training time. This syntax is equivalent to setting `NumGroups` to the number of input channels.

`layer = groupedConvolution2dLayer( ___ ,Name,Value)` sets the optional `Stride`, `DilationFactor`, “Parameters and Initialization” on page 1-683, “Learning Rate and Regularization” on page 1-684, and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `groupedConvolution2dLayer(5,128,2,'Padding','same')` creates a 2-D grouped convolutional layer with 2 groups of 128 filters of size `[5 5]` and pads the input to so that the output has the same size. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input or to set the `Stride`, `DilationFactor`, “Parameters and Initialization” on

page 1-683, “Learning Rate and Regularization” on page 1-684, and `Name` properties. Enclose names in single quotes.

Example: `groupedConvolution2dLayer(5,128,2,'Padding','same')` creates a 2-D grouped convolutional layer with 2 groups of 128 filters of size `[5 5]` and pads the input to so that the output has the same size.

`Padding` — Input edge padding
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.
- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Vector `[a b]` of nonnegative integers — Add padding of size `a` to the top and bottom of the input and padding of size `b` to the left and right.
- Vector `[t b l r]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, and `r` to the right of the input.

Example: `'Padding',1` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Grouped Convolution

`FilterSize` — Height and width of filters
vector of two positive integers

Height and width of the filters, specified as a vector `[h w]` of two positive integers, where `h` is the height and `w` is the width. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify `FilterSize` as a scalar to use the same value for the height and width.

Example: `[5 5]` specifies filters with a height of 5 and a width of 5.

`NumFiltersPerGroup` — Number of filters per group
positive integer

Number of filters per group, specified as a positive integer. This property determines the number of channels in the output of the layer. The number of output channels is `FiltersPerGroup * NumGroups`.

Example: `10`

`NumGroups` — Number of groups
positive integer | `'channel-wise'`

Number of groups, specified as a positive integer or `'channel-wise'`.

If `NumGroups` is `'channel-wise'`, then the software creates a layer for channel-wise convolution (also known as depth-wise convolution). In this case, the layer determines the `NumGroups` property at training time. This value is equivalent to setting `NumGroups` to the number of input channels.

The number of groups must evenly divide the number of channels of the layer input.

Example: 2

`Stride` — Step size for traversing input
`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector `[a b]` of two positive integers, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both step sizes.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`DilationFactor` — Factor for dilated convolution
`[1 1]` (default) | vector of two positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector `[h w]` of two positive integers, where `h` is the vertical dilation and `w` is the horizontal dilation. When creating the layer, you can specify `DilationFactor` as a scalar to use the same value for both horizontal and vertical dilations.

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.

Example: `[2 3]`

`PaddingSize` — Size of padding
`[0 0 0 0]` (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector `[t b l r]` of four nonnegative integers, where `t` is the padding applied to the top, `b` is the padding applied to the bottom, `l` is the padding applied to the left, and `r` is the padding applied to the right.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 1 2 2]` adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

PaddingMode — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the `'Padding'` value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

PaddingValue — Value to pad data
0 (default) | scalar | `'symmetric-include-edge'` | `'symmetric-exclude-edge'` | `'replicate'`

Value to pad data, specified as one of the following:

PaddingValue	Description	Example
Scalar	Pad with the specified scalar value.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&3&1&4&0&0\\0&0&1&5&9&0&0\\0&0&2&6&5&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}$
`'symmetric-include-edge'`	Pad using mirrored values of the input, including the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&1&1&5&9&9&5\\1&3&3&1&4&4&1\\1&3&3&1&4&4&1\\5&1&1&5&9&9&5\\6&2&2&6&5&5&6\\6&2&2&6&5&5&6\\5&1&1&5&9&9&5\end{bmatrix}$
`'symmetric-exclude-edge'`	Pad using mirrored values of the input, excluding the edge values.	$\begin{bmatrix}3&1&4\\1&5&9\\2&6&5\end{bmatrix} \rightarrow \begin{bmatrix}5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\\9&5&1&5&9&5&1\\5&6&2&6&5&6&2\\9&5&1&5&9&5&1\\4&1&3&1&4&1&3\end{bmatrix}$

PaddingValue	Description	Example
`'replicate'`	Pad using repeated border elements of the input	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 3 & 3 & 1 & 4 & 4 & 4 \\ 3 & 3 & 3 & 1 & 4 & 4 & 4 \\ 3 & 3 & 3 & 1 & 4 & 4 & 4 \\ 1 & 1 & 1 & 5 & 9 & 9 & 9 \\ 2 & 2 & 2 & 6 & 5 & 5 & 5 \\ 2 & 2 & 2 & 6 & 5 & 5 & 5 \\ 2 & 2 & 2 & 6 & 5 & 5 & 5 \end{bmatrix}$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`NumChannelsPerGroup` — Number of channels per group
`'auto'` (default) | positive integer

Number of channels per group, specified as `'auto'` or a positive integer. The number of channels per group is equal to the number of input channels divided by the number of groups.

The software automatically sets this property at training time.

Example: 256

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannelsPerGroup` and `numOut = FilterSize(1)*FilterSize(2)*NumFiltersPerGroup`.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannelsPerGroup`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | numeric array

Layer weights for the layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`NumChannelsPerGroup`-by-`NumFiltersPerGroup`-by-`NumGroups` array, where `NumInputChannels` is the number of channels of the layer input.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-`NumFiltersPerGroup`-by-`NumGroups` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Grouped Convolution Layer

Create a grouped convolutional layer with 3 groups of 10 filters, each with a height and width of 11, and the name `'gconv1'`.

```
layer = groupedConvolution2dLayer(11,10,3,'Name','gconv1')
```

```
layer =
  GroupedConvolution2DLayer with properties:

                  Name: 'gconv1'

   Hyperparameters
             FilterSize: [11 11]
              NumGroups: 3
    NumChannelsPerGroup: 'auto'
     NumFiltersPerGroup: 10
                 Stride: [1 1]
         DilationFactor: [1 1]
            PaddingMode: 'manual'
            PaddingSize: [0 0 0 0]
           PaddingValue: 0

   Learnable Parameters
                Weights: []
                   Bias: []

  Show all properties
```

Create Channel-Wise Convolution Layer

Create a channel-wise convolutional (also known as depth-wise convolutional) layer with groups of 10 filters, each with a height and width of 11, and the name `'cwconv1'`.

```
layer = groupedConvolution2dLayer(11,10,'channel-wise','Name','cwconv1')
```

```
layer =
  GroupedConvolution2DLayer with properties:

                  Name: 'cwconv1'

   Hyperparameters
             FilterSize: [11 11]
              NumGroups: 'channel-wise'
    NumChannelsPerGroup: 'auto'
     NumFiltersPerGroup: 10
                 Stride: [1 1]
         DilationFactor: [1 1]
            PaddingMode: 'manual'
            PaddingSize: [0 0 0 0]
           PaddingValue: 0

   Learnable Parameters
                Weights: []
                   Bias: []

  Show all properties
```

Create Layers for Channel-Wise Separable Convolution

A typical convolutional neural network contains blocks of convolution, batch normalization, and ReLU layers. For example,

```
filterSize = 3;
numFilters = 16;

convLayers = [
    convolution2dLayer(filterSize,numFilters,'Stride',2,'Padding','same')
    batchNormalizationLayer
    reluLayer];
```

For channel-wise separable convolution (also known as depth-wise separable convolution), replace the convolution block with channel-wise convolution and point-wise convolution blocks.

Specify the filter size and the stride in the channel-wise convolution and the number of filters in the point-wise convolution. For the channel-wise convolution, specify one filter per group. For point-wise convolution, specify filters of size 1 in `convolution2dLayer`.

```
cwsConvLayers = [
    groupedConvolution2dLayer(filterSize,1,'channel-wise','Stride',2,'Padding','same')
    batchNormalizationLayer
```

```
    reluLayer

    convolution2dLayer(1,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer];
```

Create a network containing layers for channel-wise separable convolution.

```
layers = [
    imageInputLayer([227 227 3])

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    groupedConvolution2dLayer(3,1,'channel-wise','Stride',2,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(1,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    fullyConnectedLayer(5)
    softmaxLayer
    classificationLayer];
```

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation for the ARM Compute Library is not supported for a 2-D grouped convolution layer that has the `NumGroups` property set to an integer value greater than two.
- For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation for the ARM Mali GPU is not supported for a 2-D grouped convolution layer that has the `NumGroups` property set as `'channel-wise'` or a value greater than two.
- For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.

See Also

`trainNetwork` | `reluLayer` | `batchNormalizationLayer` | `maxPooling2dLayer` | `fullyConnectedLayer` | `convolution2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2019a

groupnorm

Normalize data across grouped subsets of channels for each observation independently

Syntax

```
dlY = groupnorm(dlX,numGroups,offset,scaleFactor)
dlY = groupnorm(dlX,numGroups,offset,scaleFactor,'DataFormat',FMT)
dlY = groupnorm(___,Name,Value)
```

Description

The group normalization operation normalizes the input data across grouped subsets of channels for each observation independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use group normalization between convolution and nonlinear operations such as `relu`.

After normalization, the operation shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

The `groupnorm` function applies the group normalization operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply group normalization within a `layerGraph` object or `Layer` array, use `groupNormalizationLayer`.

`dlY = groupnorm(dlX,numGroups,offset,scaleFactor)` applies the group normalization operation to the input data `dlX` using the specified number of groups and transforms it using the specified offset and scale factor.

The function normalizes over grouped subsets of the `'C'` (channel) dimension and the `'S'` (spatial), `'T'` (time), and `'U'` (unspecified) dimensions of `dlX` for each observation in the `'B'` (batch) dimension, independently.

For unformatted input data, use the `'DataFormat'` option.

`dlY = groupnorm(dlX,numGroups,offset,scaleFactor,'DataFormat',FMT)` applies the group normalization operation to the unformatted `dlarray` object `dlX` with format specified by `FMT`. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D image input with format `'SSCB'` (spatial, spatial, channel, batch).

`dlY = groupnorm(___,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, `'Epsilon',3e-5` sets the variance offset to `3e-5`.

Examples

Normalize Data

Use `groupnorm` to normalize input data across channel groups.

Create the input data as a single observation of random values with a height and width of four and six channels.

```
height = 4;
width = 4;
channels = 6;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Create the learnable parameters.

```
offset = zeros(channels,1);
scaleFactor = ones(channels,1);
```

Compute the group normalization. Divide the input into three groups of two channels each.

```
numGroups = 3;
dlY = groupnorm(dlX,numGroups,offset,scaleFactor);
```

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `dlX` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `scaleFactor` or `offset` must be a `dlarray` object.

`dlX` must have a `'C'` (channel) dimension.

`numGroups` — Number of channel groups

positive integer | `'all-channels'` | `'channel-wise'`

Number of channel groups to normalize across, specified as a positive integer, `'all-channels'`, or `'channel-wise'`.

`numGroups`	Description
positive integer	Divide the incoming channels into the specified number of groups. The specified number of groups must divide the number of channels of the input data exactly.

numGroups	Description
'all-channels'	Group all incoming channels into a single group. The input data is normalized across all channels. This operation is also known as layer normalization. Alternatively, use layernorm.
'channel-wise'	Treat all incoming channels as separate groups. This operation is also known as instance normalization. Alternatively, use instancenorm.

Data Types: single | double | char | string

offset — Offset
dlarray | numeric array

Offset β, specified as a formatted dlarray, an unformatted dlarray, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input dlX.

If offset is a formatted dlarray object, then the nonsingleton dimension must have label 'C' (channel).

scaleFactor — Scale factor
dlarray | numeric array

Scale factor γ, specified as a formatted dlarray, an unformatted dlarray, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input dlX.

If scaleFactor is a formatted dlarray object, then the nonsingleton dimension must have label 'C' (channel).

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Epsilon',3e-5 sets the variance offset to 3e-5.

DataFormat — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar FMT that provides a label for each dimension of the data.

When you specify the format of a dlarray object, each character provides a label for each dimension of the data and must be one of the following:

- "S" — Spatial
- "C" — Channel
- "B" — Batch (for example, samples and observations)
- "T" — Time (for example, time steps of sequences)
- "U" — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`Epsilon` — Variance offset
`1e-5` (default) | numeric scalar

Variance offset for preventing divide-by-zero errors, specified as the comma-separated pair consisting of `'Epsilon'` and a numeric scalar greater than or equal to `1e-5`.

Data Types: `single` | `double`

Output Arguments

`dlY` — Normalized data
`dlarray`

Normalized data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension labels as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Algorithms

The group normalization operation normalizes the elements x_i of the input by first calculating the mean μ_G and variance σ_G^2 over spatial, time, and grouped subsets of the channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\widehat{x}_i = \frac{x_i - \mu_G}{\sqrt{\sigma_G^2 + \varepsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small. To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow group normalization, the group normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

References

[1] Wu, Yuxin, and Kaiming He. “Group Normalization.” Preprint submitted June 11, 2018. https://arxiv.org/abs/1803.08494.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `offset`
  - `scaleFactor`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`relu` | `fullyconnect` | `dlconv` | `dlarray` | `dlgradient` | `dlfeval` | `batchnorm` | `layernorm` | `instancenorm`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2020b

groupNormalizationLayer

Group normalization layer

Description

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

After normalization, the layer scales the input with a learnable scale factor γ and shifts it by a learnable offset β.

Creation

Syntax

```
layer = groupNormalizationLayer(numGroups)
layer = groupNormalizationLayer(numGroups,Name,Value)
```

Description

`layer = groupNormalizationLayer(numGroups)` creates a group normalization layer.

`layer = groupNormalizationLayer(numGroups,Name,Value)` creates a group normalization layer and sets the optional `'Epsilon'`, “Parameters and Initialization” on page 1-696, “Learning Rate and Regularization” on page 1-697, and `Name` properties using one or more name-value arguments. You can specify multiple name-value arguments. Enclose each property name in quotes.

Input Arguments

`numGroups` — Number of groups
positive integer | `'all-channels'` | `'channel-wise'`

Number of groups into which to divide the channels of the input data, specified as one of the following:

- Positive integer – Divide the incoming channels into the specified number of groups. The specified number of groups must divide the number of channels of the input data exactly.
- `'all-channels'` – Group all incoming channels into a single group. This operation is also known as layer normalization. Alternatively, use `layerNormalizationLayer`.
- `'channel-wise'` – Treat all incoming channels as separate groups. This operation is also known as instance normalization. Alternatively, use `instanceNormalizationLayer`.

Properties

Group Normalization

`Epsilon` — Constant to add to mini-batch variances

`1e-5` (default) | numeric scalar

Constant to add to the mini-batch variances, specified as a numeric scalar equal to or larger than `1e-5`.

The layer adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`NumChannels` — Number of input channels

`'auto'` (default) | positive integer

Number of input channels, specified as `'auto'` or a positive integer.

This property is always equal to the number of channels of the input to the layer. If `NumChannels` is `'auto'`, then the software automatically determines the correct value for the number of channels at training time.

Parameters and Initialization

`ScaleInitializer` — Function to initialize channel scale factors

`'ones'` (default) | `'narrow-normal'` | function handle

Function to initialize the channel scale factors, specified as one of the following:

- `'ones'` – Initialize the channel scale factors with ones.
- `'zeros'` – Initialize the channel scale factors with zeros.
- `'narrow-normal'` – Initialize the channel scale factors by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel scale factors with a custom function. If you specify a function handle, then the function must be of the form `scale = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel scale factors when the `Scale` property is empty.

Data Types: `char` | `string` | `function_handle`

`OffsetInitializer` — Function to initialize channel offsets

`'zeros'` (default) | `'ones'` | `'narrow-normal'` | function handle

Function to initialize the channel offsets, specified as one of the following:

- `'zeros'` – Initialize the channel offsets with zeros.
- `'ones'` – Initialize the channel offsets with ones.
- `'narrow-normal'` – Initialize the channel offsets by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel offsets with a custom function. If you specify a function handle, then the function must be of the form `offset = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel offsets when the `Offset` property is empty.

Data Types: `char` | `string` | `function_handle`

`Scale` — Channel scale factors

`[ ]` (default) | numeric array

Channel scale factors γ, specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network, if `Scale` is nonempty, then `trainNetwork` uses the `Scale` property as the initial value. If `Scale` is empty, then `trainNetwork` uses the initializer specified by `ScaleInitializer`.

At training time, `Scale` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

`Offset` — Channel offsets

`[ ]` (default) | numeric array

Channel offsets β, specified as a numeric array.

The channel offsets are learnable parameters. When you train a network, if `Offset` is nonempty, then `trainNetwork` uses the `Offset` property as the initial value. If `Offset` is empty, then `trainNetwork` uses the initializer specified by `OffsetInitializer`.

At training time, `Offset` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

Learning Rate and Regularization

`ScaleLearnRateFactor` — Learning rate factor for scale factors

1 (default) | nonnegative scalar

Learning rate factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the scale factors in a layer. For example, if `ScaleLearnRateFactor` is `2`, then the learning rate for the scale factors in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`OffsetLearnRateFactor` — Learning rate factor for offsets

1 (default) | nonnegative scalar

Learning rate factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the offsets in a layer. For example, if `OffsetLearnRateFactor` is `2`, then the learning rate for the offsets in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`ScaleL2Factor` — L_2 regularization factor for scale factors
1 (default) | nonnegative scalar

L_2 regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

`OffsetL2Factor` — L_2 regularization factor for offsets
1 (default) | nonnegative scalar

L_2 regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Group Normalization Layer

Create a group normalization layer that normalizes incoming data across three groups of channels. Name the layer `'groupnorm'`.

```
layer = groupNormalizationLayer(3,'Name','groupnorm')
```

```
layer =
  GroupNormalizationLayer with properties:

           Name: 'groupnorm'
    NumChannels: 'auto'

   Hyperparameters
      NumGroups: 3
        Epsilon: 1.0000e-05

   Learnable Parameters
         Offset: []
          Scale: []

  Show all properties
```

Include a group normalization layer in a `Layer` array. Normalize the incoming 20 channels in four groups.

```
layers = [
    imageInputLayer([28 28 3])
    convolution2dLayer(5,20)
    groupNormalizationLayer(4)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  8x1 Layer array with layers:

     1   ''   Image Input             28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0  0
     3   ''   Group Normalization     Group normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
```

```
  6   ''   Fully Connected         10 fully connected layer
  7   ''   Softmax                 softmax
  8   ''   Classification Output   crossentropyex
```

More About

Group Normalization Layer

A group normalization layer divides the channels of the input data into groups and normalizes the activations across each group. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use group normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

You can also use a group normalization layer to perform layer normalization or instance normalization. Layer normalization combines and normalizes activations across all channels in a single observation. Instance normalization normalizes the activations of each channel of the observation separately.

The layer first normalizes the activations of each group by subtracting the group mean and dividing by the group standard deviation. Then, the layer shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

Group 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.

You can use a group normalization layer in place of a batch normalization layer. Doing so is particularly useful when training with small batch sizes, as it can increase the stability of training.

Algorithms

The group normalization operation normalizes the elements x_i of the input by first calculating the mean μ_G and variance σ_G^2 over spatial, time, and grouped subsets of the channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\widehat{x}_i = \frac{x_i - \mu_G}{\sqrt{\sigma_G^2 + \varepsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small. To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow group normalization, the group normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

References

[1] Wu, Yuxin, and Kaiming He. “Group Normalization.” Preprint submitted June 11, 2018. https://arxiv.org/abs/1803.08494.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `trainingOptions` | `reluLayer` | `convolution2dLayer` | `fullyConnectedLayer` | `batchNormalizationLayer` | `layerNormalizationLayer` | `instanceNormalizationLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2020b

groupSubPlot

Package: `experiments`

Group metrics in experiment training plot

Syntax

```
groupSubPlot(monitor,title,metricNames)
```

Description

`groupSubPlot(monitor,title,metricNames)` groups the specified metrics in a single training subplot with the title `title`. By default, **Experiment Manager** plots each ungrouped metric in its own training subplot.

Examples

Track Progress, Display Information and Record Metric Values, and Produce Training Plots

Use an `experiments.Monitor` object to track the progress of the training, display information and metric values in the experiment results table, and produce training plots for custom training experiments.

Before starting the training, specify the names of the information and metric columns of the Experiment Manager results table.

```
monitor.Info = ["GradientDecayFactor","SquaredGradientDecayFactor"];
monitor.Metrics = ["TrainingLoss","ValidationLoss"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot.

```
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
```

Update the values of the gradient decay factor and the squared gradient decay factor for the trial in the results table.

```
updateInfo(monitor, ...
    GradientDecayFactor=gradientDecayFactor, ...
    SquaredGradientDecayFactor=squaredGradientDecayFactor);
```

After each iteration of the custom training loop, record the value of training and validation loss for the trial in the results table and the training plot.

```
recordMetrics(monitor,iteration, ...
    TrainingLoss=trainingLoss, ...
    ValidationLoss=validationLoss);
```

Update the training progress for the trial based on the fraction of iterations completed.

```
monitor.Progress = (iteration/numIterations) * 100;
```

Input Arguments

`monitor` — Experiment monitor

`experiments.Monitor` object

Experiment monitor for the trial, specified as an `experiments.Monitor` object. When you run a custom training experiment, Experiment Manager passes this object as the second input argument of the training function.

`title` — Title of training subplot

string | character vector

Title of the training subplot, specified as a string or character vector.

Data Types: `char` | `string`

`metricNames` — Metric names

string | character vector | string array | cell array of character vectors

Metric names, specified as a string, character vector, string array, or cell array of character vectors. Each metric name must be an element of the `Metrics` property of the `experiments.Monitor` object `monitor`.

Data Types: `char` | `string`

Tips

- Use the `groupSubplot` function to define your training subplots before calling the function `recordMetrics`.

See Also

Apps
Experiment Manager

Objects
`experiments.Monitor`

Functions
`recordMetrics` | `updateInfo`

Introduced in R2021a

gru

Gated recurrent unit

Syntax

```
dlY = gru(dlX,H0,weights,recurrentWeights,bias)
[dlY,hiddenState] = gru(dlX,H0,weights,recurrentWeights,bias)
[___] = gru(___,'DataFormat',FMT)
```

Description

The gated recurrent unit (GRU) operation allows a network to learn dependencies between time steps in time series and sequence data.

Note This function applies the deep learning GRU operation to `dlarray` data. If you want to apply an GRU operation within a `layerGraph` object or `Layer` array, use the following layer:

- `gruLayer`

`dlY = gru(dlX,H0,weights,recurrentWeights,bias)` applies a gated recurrent unit (GRU) calculation to input `dlX` using the initial hidden state `H0`, and parameters `weights`, `recurrentWeights`, and `bias`. The input `dlX` must be a formatted `dlarray`. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`, except for any `'S'` dimensions.

The `gru` function updates the hidden state using the hyperbolic tangent function (tanh) as the state activation function. The `gru` function uses the sigmoid function given by $\sigma(x) = (1 + e^{-x})^{-1}$ as the gate activation function.

`[dlY,hiddenState] = gru(dlX,H0,weights,recurrentWeights,bias)` also returns the hidden state after the GRU operation.

`[___] = gru(___,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`, except for any `'S'` dimensions.

Examples

Apply GRU Operation to Sequence Data

Perform a GRU operation using 100 hidden units.

Create the input sequence data as 32 observations with ten channels and a sequence length of 64.

```
numFeatures = 10;
numObservations = 32;
sequenceLength = 64;
```

```
X = randn(numFeatures,numObservations,sequenceLength);
dlX = dlarray(X,'CBT');
```

Create the initial hidden state with 100 hidden units. Use the same initial hidden state for all observations.

```
numHiddenUnits = 100;
H0 = zeros(numHiddenUnits,1);
```

Create the learnable parameters for the GRU operation.

```
weights = dlarray(randn(3*numHiddenUnits,numFeatures));
recurrentWeights = dlarray(randn(3*numHiddenUnits,numHiddenUnits));
bias = dlarray(randn(3*numHiddenUnits,1));
```

Perform the GRU calculation.

```
[dlY,hiddenState] = gru(dlX,H0,weights,recurrentWeights,bias);
```

View the size and dimension format of `dlY`.

```
size(dlY)
```

```
ans = 1×3

   100    32    64
```

```
dlY.dims
```

```
ans =
'CBT'
```

View the size of `hiddenState`.

```
size(hiddenState)
```

```
ans = 1×2

   100    32
```

You can use the hidden state to keep track of the state of the GRU operation and input further sequential data.

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using `'DataFormat',FMT`. If `dlX` is a numeric array, at least one of `H0`, `weights`, `recurrentWeights`, or `bias` must be a `dlarray`.

`dlX` must contain a sequence dimension labeled `'T'`. If `dlX` has any spatial dimensions labeled `'S'`, they are flattened into the `'C'` channel dimension. If `dlX` does not have a channel dimension, then one is added. If `dlX` has any unspecified dimensions labeled `'U'`, they must be singleton.

Data Types: `single` | `double`

`H0` — Initial hidden state vector
`dlarray` | numeric array

Initial hidden state vector, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `H0` is a formatted `dlarray`, it must contain a channel dimension labeled `'C'` and optionally a batch dimension labeled `'B'` with the same size as the `'B'` dimension of `dlX`. If `H0` does not have a `'B'` dimension, the function uses the same hidden state vector for each observation in `dlX`.

If `H0` is a formatted `dlarray`, then the size of the `'C'` dimension determines the number of hidden units. Otherwise, the size of the first dimension determines the number of hidden units.

Data Types: `single` | `double`

`weights` — Weights
`dlarray` | numeric array

Weights, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Specify `weights` as a matrix of size `3*NumHiddenUnits`-by-`InputSize`, where `NumHiddenUnits` is the size of the `'C'` dimension of `H0`, and `InputSize` is the size of the `'C'` dimension of `dlX` multiplied by the size of each `'S'` dimension of `dlX`, where present.

If `weights` is a formatted `dlarray`, it must contain a `'C'` dimension of size `3*NumHiddenUnits` and a `'U'` dimension of size `InputSize`.

Data Types: `single` | `double`

`recurrentWeights` — Recurrent weights
`dlarray` | numeric array

Recurrent weights, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Specify `recurrentWeights` as a matrix of size `3*NumHiddenUnits`-by-`NumHiddenUnits`, where `NumHiddenUnits` is the size of the `'C'` dimension of `H0`.

If `recurrentWeights` is a formatted `dlarray`, it must contain a `'C'` dimension of size `3*NumHiddenUnits` and a `'U'` dimension of size `NumHiddenUnits`.

Data Types: `single` | `double`

`bias` — Bias
`dlarray` vector | numeric vector

Bias, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Specify `bias` as a vector of length `3*NumHiddenUnits`, where `NumHiddenUnits` is the size of the `'C'` dimension of `H0`.

If `bias` is a formatted `dlarray`, the nonsingleton dimension must be labeled with `'C'`.

Data Types: `single` | `double`

`FMT` — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`dlY` — GRU output
`dlarray`

GRU output, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`, except for any `'S'` dimensions. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

The size of the `'C'` dimension of `dlY` is the same as the number of hidden units, specified by the size of the `'C'` dimension of `H0`.

`hiddenState` — Hidden state vector
`dlarray` | numeric array

Hidden state vector for each observation, returned as a `dlarray` or a numeric array with the same data type as `H0`.

If the input `H0` is a formatted `dlarray`, then the output `hiddenState` is a formatted `dlarray` with the format `'CB'`.

Limitations

- `functionToLayerGraph` does not support the `gru` function. If you use `functionToLayerGraph` with a function that contains the `gru` operation, the resulting `LayerGraph` contains placeholder layers.

More About

Gated Recurrent Unit

The GRU operation allows a network to learn dependencies between time steps in time series and sequence data. For more information, see the "Gated Recurrent Unit Layer" on page 1-719 definition on the `gruLayer` reference page.

References

[1] Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `H0`
  - `weights`
  - `recurrentWeights`
  - `bias`

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`dlarray` | `fullyconnect` | `softmax` | `dlgradient` | `dlfeval` | `lstm`

Topics

"Define Custom Training Loops, Loss Functions, and Networks"
"Train Network Using Model Function"
"Sequence-to-Sequence Translation Using Attention"
"Multilabel Text Classification Using Deep Learning"
"List of Functions with dlarray Support"

Introduced in R2020a

gruLayer

Gated recurrent unit (GRU) layer

Description

A GRU layer learns dependencies between time steps in time series and sequence data.

Creation

Syntax

```
layer = gruLayer(numHiddenUnits)
layer = gruLayer(numHiddenUnits,Name,Value)
```

Description

`layer = gruLayer(numHiddenUnits)` creates a GRU layer and sets the `NumHiddenUnits` property.

`layer = gruLayer(numHiddenUnits,Name,Value)` sets additional `OutputMode`, "Activations" on page 1-711, "State" on page 1-711, "Parameters and Initialization" on page 1-711, "Learning Rate and Regularization" on page 1-714, and `Name` properties using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.

Properties

GRU

`NumHiddenUnits` — Number of hidden units
positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data. This value can vary from a few dozen to a few thousand.

The hidden state does not limit the number of time steps that are processed in an iteration. To split your sequences into smaller sequences for training, use the `'SequenceLength'` option in `trainingOptions`.

Example: 200

`OutputMode` — Output mode
`'sequence'` (default) | `'last'`

Output mode, specified as one of the following:

- `'sequence'` – Output the complete sequence.
- `'last'` – Output the last time step of the sequence.

`HasStateInputs` — Flag for state inputs to layer
`0` (false) (default) | `1` (true)

Flag for state inputs to the layer, specified as `1` (true) or `0` (false).

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` property for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has two inputs with names `'in'` and `'hidden'`, which correspond to the input data and hidden state, respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` property must be empty.

`HasStateOutputs` — Flag for state outputs from layer
`0` (false) (default) | `1` (true)

Flag for state outputs from the layer, specified as `true` or `false`.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has two outputs with names `'out'` and `'hidden'`, which correspond to the output data and hidden state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

`ResetGateMode` — Reset gate mode
`'after-multiplication'` (default) | `'before-multiplication'` | `'recurrent-bias-after-multiplication'`

Reset gate mode, specified as one of the following:

- `'after-multiplication'` – Apply reset gate after matrix multiplication. This option is cuDNN compatible.
- `'before-multiplication'` – Apply reset gate before matrix multiplication.
- `'recurrent-bias-after-multiplication'` – Apply reset gate after matrix multiplication and use an additional set of bias terms for the recurrent weights.

For more information about the reset gate calculations, see “Gated Recurrent Unit Layer” on page 1-719.

`InputSize` — Input size
`'auto'` (default) | positive integer

Input size, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically assigns the input size at training time.

Example: 100

Activations

`StateActivationFunction` — Activation function to update the hidden state

`'tanh'` (default) | `'softsign'`

Activation function to update the hidden state, specified as one of the following:

- `'tanh'` – Use the hyperbolic tangent function (tanh).
- `'softsign'` – Use the softsign function $\text{softsign}(x) = \frac{x}{1 + |x|}$.

The layer uses this option as the function σ_s in the calculations to update the hidden state.

`GateActivationFunction` — Activation function to apply to the gates

`'sigmoid'` (default) | `'hard-sigmoid'`

Activation function to apply to the gates, specified as one of the following:

- `'sigmoid'` – Use the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$.
- `'hard-sigmoid'` – Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if} -2.5 \le x \le 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}.$$

The layer uses this option as the function σ_g in the calculations for the layer gates.

State

`HiddenState` — Hidden state

numeric vector

Hidden state to use in the layer operation, specified as a `NumHiddenUnits`-by-1 numeric vector. This value corresponds to the initial hidden state when data is passed to the layer.

After setting this property manually, calls to the `resetState` function set the hidden state to this value.

If `HasStateInputs` is `true`, then the `HiddenState` property must be empty.

Parameters and Initialization

`InputWeightsInitializer` — Function to initialize input weights

`'glorot'` (default) | `'he'` | `'orthogonal'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the input weights, specified as one of the following:

- `'glorot'` – Initialize the input weights with the Glorot initializer [2] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(InputSize + numOut)`, where `numOut = 3*NumHiddenUnits`.
- `'he'` – Initialize the input weights with the He initializer [3]. The He initializer samples from a normal distribution with zero mean and variance `2/InputSize`.

- `'orthogonal'` – Initialize the input weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [4]
- `'narrow-normal'` – Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the input weights with zeros.
- `'ones'` – Initialize the input weights with ones.
- Function handle – Initialize the input weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the input weights.

The layer only initializes the input weights when the `InputWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`RecurrentWeightsInitializer` — Function to initialize recurrent weights
`'orthogonal'` (default) | `'glorot'` | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the recurrent weights, specified as one of the following:

- `'orthogonal'` – Initialize the recurrent weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [4]
- `'glorot'` – Initialize the recurrent weights with the Glorot initializer [2] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = NumHiddenUnits` and `numOut = 3*NumHiddenUnits`.
- `'he'` – Initialize the recurrent weights with the He initializer [3]. The He initializer samples from a normal distribution with zero mean and variance `2/NumHiddenUnits`.
- `'narrow-normal'` – Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the recurrent weights with zeros.
- `'ones'` – Initialize the recurrent weights with ones.
- Function handle – Initialize the recurrent weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the recurrent weights.

The layer only initializes the recurrent weights when the `RecurrentWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `zeros'` – Initialize the bias with zeros.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'ones'` – Initialize the bias with ones.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`InputWeights` — Input weights
`[]` (default) | matrix

Input weights, specified as a matrix.

The input weight matrix is a concatenation of the three input weight matrices for the components in the GRU layer. The three matrices are concatenated vertically in the following order:

1. Reset gate
2. Update gate
3. Candidate state

The input weights are learnable parameters. When training a network, if `InputWeights` is nonempty, then `trainNetwork` uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then `trainNetwork` uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is a `3*NumHiddenUnits`-by-`InputSize` matrix.

`RecurrentWeights` — Recurrent weights
`[]` (default) | matrix

Recurrent weights, specified as a matrix.

The recurrent weight matrix is a concatenation of the three recurrent weight matrices for the components in the GRU layer. The three matrices are vertically concatenated in the following order:

1. Reset gate
2. Update gate
3. Candidate state

The recurrent weights are learnable parameters. When training a network, if `RecurrentWeights` is nonempty, then `trainNetwork` uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then `trainNetwork` uses the initializer specified by `RecurrentWeightsInitializer`.

At training time `RecurrentWeights` is a `3*NumHiddenUnits`-by-`NumHiddenUnits` matrix.

`Bias` — Layer biases
`[]` (default) | numeric vector

Layer biases for the GRU layer, specified as a numeric vector.

If `ResetGateMode` is `'after-multiplication'` or `'before-multiplication'`, then the bias vector is a concatenation of three bias vectors for the components in the GRU layer. The three vectors are concatenated vertically in the following order:

1. Reset gate
2. Update gate

3 Candidate state

In this case, at training time, `Bias` is a `3*NumHiddenUnits`-by-1 numeric vector.

If `ResetGateMode` is `recurrent-bias-after-multiplication'`, then the bias vector is a concatenation of six bias vectors for the components in the GRU layer. The six vectors are concatenated vertically in the following order:

1 Reset gate
2 Update gate
3 Candidate state
4 Reset gate (recurrent bias)
5 Update gate (recurrent bias)
6 Candidate state (recurrent bias)

In this case, at training time, `Bias` is a `6*NumHiddenUnits`-by-1 numeric vector.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

For more information about the reset gate calculations, see “Gated Recurrent Unit Layer” on page 1-719.

Learning Rate and Regularization

`InputWeightsLearnRateFactor` — Learning rate factor for input weights
1 (default) | numeric scalar | 1-by-3 numeric vector

Learning rate factor for the input weights, specified as a numeric scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input weights of the layer. For example, if `InputWeightsLearnRateFactor` is 2, then the learning rate factor for the input weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the three individual matrices in `InputWeights`, specify a 1-by-3 vector. The entries of `InputWeightsLearnRateFactor` correspond to the learning rate factor of the following:

1 Reset gate
2 Update gate
3 Candidate state

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1]`

`RecurrentWeightsLearnRateFactor` — Learning rate factor for recurrent weights
1 (default) | numeric scalar | 1-by-3 numeric vector

Learning rate factor for the recurrent weights, specified as a numeric scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the recurrent weights of the layer. For example, if `RecurrentWeightsLearnRateFactor` is 2, then the learning rate for the recurrent weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the three individual matrices in `RecurrentWeights`, specify a 1-by-3 vector. The entries of `RecurrentWeightsLearnRateFactor` correspond to the learning rate factor of the following:

1. Reset gate
2. Update gate
3. Candidate state

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1]`

`BiasLearnRateFactor` — Learning rate factor for biases

1 (default) | nonnegative scalar | 1-by-3 numeric vector

Learning rate factor for the biases, specified as a nonnegative scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

To control the value of the learning rate factor for the three individual vectors in `Bias`, specify a 1-by-3 vector. The entries of `BiasLearnRateFactor` correspond to the learning rate factor of the following:

1. Reset gate
2. Update gate
3. Candidate state

If `ResetGateMode` is `'recurrent-bias-after-multiplication'`, then the software uses the same vector for the recurrent bias vectors.

To specify the same value for all the vectors, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1]`

`InputWeightsL2Factor` — L2 regularization factor for input weights

1 (default) | numeric scalar | 1-by-3 numeric vector

L2 regularization factor for the input weights, specified as a numeric scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the L2 regularization factor for the input weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the three individual matrices in `InputWeights`, specify a 1-by-3 vector. The entries of `InputWeightsL2Factor` correspond to the L2 regularization factor of the following:

1. Reset gate
2. Update gate
3. Candidate state

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1]`

`RecurrentWeightsL2Factor` — L2 regularization factor for recurrent weights
1 (default) | numeric scalar | 1-by-3 numeric vector

L2 regularization factor for the recurrent weights, specified as a numeric scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the L2 regularization factor for the recurrent weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the three individual matrices in `RecurrentWeights`, specify a 1-by-3 vector. The entries of `RecurrentWeightsL2Factor` correspond to the L2 regularization factor of the following:

1. Reset gate
2. Update gate
3. Candidate state

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1]`

`BiasL2Factor` — L2 regularization factor for biases
0 (default) | nonnegative scalar | 1-by-3 numeric vector

L2 regularization factor for the biases, specified as a nonnegative scalar or a 1-by-3 numeric vector.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

To control the value of the L2 regularization factor for the individual vectors in `Bias`, specify a 1-by-3 vector. The entries of `BiasL2Factor` correspond to the L2 regularization factor of the following:

1. Reset gate
2. Update gate
3. Candidate state

If `ResetGateMode` is `'recurrent-bias-after-multiplication'`, then the software uses the same vector for the recurrent bias vectors.

To specify the same value for all the vectors, specify a nonnegative scalar.

Example: `2`

Example: `[1 2 1]`

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs

`1` | `2`

Number of inputs of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` property for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has two inputs with names `'in'` and `'hidden'`, which correspond to the input data and hidden state, respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` property must be empty.

Data Types: `double`

`InputNames` — Input names

`{'in'}` | `{'in','hidden'}`

Input names of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` property for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has two inputs with names `'in'` and `'hidden'`, which correspond to the input data and hidden state, respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` property must be empty.

`NumOutputs` — Number of outputs
`1` | `2`

Number of outputs of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has two outputs with names `'out'` and `'hidden'`, which correspond to the output data and hidden state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` | `{'out','hidden'}`

Output names of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has two outputs with names `'out'` and `'hidden'`, which correspond to the output data and hidden state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Examples

Create GRU Layer

Create a GRU layer with the name `'gru1'` and 100 hidden units.

```
layer = gruLayer(100,'Name','gru1')
```

```
layer =
  GRULayer with properties:

                      Name: 'gru1'
                 InputNames: {'in'}
                OutputNames: {'out'}
                  NumInputs: 1
                 NumOutputs: 1
             HasStateInputs: 0
            HasStateOutputs: 0

   Hyperparameters
                  InputSize: 'auto'
             NumHiddenUnits: 100
                 OutputMode: 'sequence'
    StateActivationFunction: 'tanh'
     GateActivationFunction: 'sigmoid'
              ResetGateMode: 'after-multiplication'

   Learnable Parameters
               InputWeights: []
```

```
           RecurrentWeights: []
                       Bias: []

   State Parameters
                HiddenState: []

  Show all properties
```

Include a GRU layer in a `Layer` array.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    gruLayer(numHiddenUnits)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 12 dimensions
     2   ''   GRU                     GRU with 100 hidden units
     3   ''   Fully Connected         9 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

Algorithms

Gated Recurrent Unit Layer

A GRU layer learns dependencies between time steps in time series and sequence data.

The *hidden state* of the layer at time step t contains the output of the GRU layer for this time step. At each time step, the layer adds information to or removes information from the state. The layer controls these updates using *gates*.

The following components control the hidden state of the layer.

Component	Purpose
Reset gate (r)	Control level of state reset
Update gate (z)	Control level of state update
Candidate state ($\tilde{h}$)	Control level of update added to hidden state

The learnable weights of a GRU layer are the input weights W (`InputWeights`), the recurrent weights R (`RecurrentWeights`), and the bias b (`Bias`). If the `ResetGateMode` property is `'recurrent-bias-after-multiplication'`, then the gate and state calculations require two sets of bias values. The matrices W and R are concatenations of the input weights and the recurrent weights of each component, respectively. These matrices are concatenated as follows:

$$W = \begin{bmatrix} W_r \\ W_z \\ W_{\tilde{h}} \end{bmatrix}, R = \begin{bmatrix} R_r \\ R_z \\ R_{\tilde{h}} \end{bmatrix},$$

where r, z, and $\tilde{h}$ denote the reset gate, update gate, and candidate state, respectively.

The bias vector depends on the `ResetGateMode` property. If `ResetGateMode` is `'after-mutliplication'` or `'before-multiplication'`, then the bias vector is a concatenation of three vectors:

$$b = \begin{bmatrix} b_{W_r} \\ b_{W_z} \\ b_{W_{\tilde{h}}} \end{bmatrix},$$

where the subscript W indicates that this is the bias corresponding to the input weights multiplication.

If `ResetGateMode` is `'recurrent-bias-after-multiplication'`, then the bias vector is a concatenation of six vectors:

$$b = \begin{bmatrix} b_{W_r} \\ b_{W_z} \\ b_{W_{\tilde{h}}} \\ b_{R_r} \\ b_{R_z} \\ b_{R_{\tilde{h}}} \end{bmatrix},$$

where the subscript R indicates that this is the bias corresponding to the recurrent weights multiplication.

The hidden state at time step t is given by

$$\mathbf{h}_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot \mathbf{h}_{t-1}.$$

The following formulas describe the components at time step t.

Component	ResetGateMode	Formula
Reset gate	`'after-multiplication'`	$r_t = \sigma_g\left(W_r\mathbf{x}_t + b_{W_r} + R_r\mathbf{h}_{t-1}\right)$
	`'before-multiplication'`	
	`'recurrent-bias-after-multiplication'`	$r_t = \sigma_g\left(W_r\mathbf{x}_t + b_{W_r} + R_r\mathbf{h}_{t-1} + b_{R_r}\right)$
Update gate	`'after-multiplication'`	$z_t = \sigma_g\left(W_z\mathbf{x}_t + b_{W_z} + R_z\mathbf{h}_{t-1}\right)$

Component	ResetGateMode	Formula
	'before-multiplication'	
	'recurrent-bias-after-multiplication'	$z_t = \sigma_g\left(W_z\mathbf{x}_t + b_{W_z} + R_z\mathbf{h}_{t-1} + b_{R_z}\right)$
Candidate state	'after-multiplication'	$\tilde{h}_t = \sigma_s\left(W_{\tilde{h}}\mathbf{x}_t + b_{W_{\tilde{h}}} + r_t \odot \left(\mathrm{R}_{\tilde{h}}\mathbf{h}_{t-1}\right)\right)$
	'before-multiplication'	$\tilde{h}_t = \sigma_s\left(W_{\tilde{h}}\mathbf{x}_t + b_{W_{\tilde{h}}} + R_{\tilde{h}}\left(r_t \odot \mathbf{h}_{t-1}\right)\right)$
	'recurrent-bias-after-multiplication'	$\tilde{h}_t = \sigma_s\left(W_{\tilde{h}}\mathbf{x}_t + b_{W_{\tilde{h}}} + r_t \odot \left(\mathrm{R}_{\tilde{h}}\mathbf{h}_{t-1} + b_{R_{\tilde{h}}}\right)\right)$

In these calculations, σ_g and σ_s denotes the gate and state activation functions, respectively. The `gruLayer` function, by default, uses the sigmoid function given by $\sigma(x) = (1 + e^{-x})^{-1}$ to compute the gate activation function and the hyperbolic tangent function (tanh) to compute the state activation function. To specify the state and gate activation functions, use the `StateActivationFunction` and `GateActivationFunction` properties, respectively.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `GRULayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	OutputMode	Output Format
"CBT" (channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

In `dlnetwork` objects, `GRULayer` objects also support the following input and output format combinations.

Input Format	OutputMode	Output Format
"SCBT" (spatial, channel, batch)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

Input Format	OutputMode	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

To use these input formats in trainNetwork workflows, first convert the data to "CBT" (channel, batch, time) format using flattenLayer.

If the HasStateInputs property is 1 (true), then the layer has an additional input with name 'hidden', which corresponds to the hidden state. This additional input expects input format "CB" (channel, batch).

If the HasStateOutputs property is 1 (true), then the layer has one additional output with name 'hidden', which corresponds to the hidden state. This additional output has output format "CB" (channel, batch).

References

[1] Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).

[2] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

[4] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." *arXiv preprint arXiv:1312.6120* (2013).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The StateActivationFunction property must be set to 'tanh'.
- The GateActivationFunction property must be set to 'sigmoid'.
- The ResetGateMode property must be set to 'after-multiplication' or 'recurrent-bias-after-multiplication'.
- The HasStateInputs and HasStateOutputs properties must be set to 0 (false).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The `StateActivationFunction` property must be set to `'tanh'`.
- The `GateActivationFunction` property must be set to `'sigmoid'`.
- The `ResetGateMode` property must be set to `'after-multiplication'` or `'recurrent-bias-after-multiplication'`.
- The `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Topics

“Sequence Classification Using Deep Learning”
“Sequence Classification Using 1-D Convolutions”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Classify Videos Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2020a

hasdata

Determine if minibatchqueue can return mini-batch

Syntax

```
tf = hasdata(mbq)
```

Description

`tf = hasdata(mbq)` returns `1` (`true`) if `mbq` can return a mini-batch using the `next` function, and `0` (`false`) otherwise.

Use `hasdata` in combination with `next` to iterate over all data in the `minibatchqueue` object. You can call `next` on a `minibatchqueue` object until all data is returned. If mini-batches of data are still available in the `minibatchqueue` object, `hasdata` returns `1`. When you reach the end of the data, `hasdata` returns `0`. Then, use `reset` or `shuffle` to reset the `minibatchqueue` object and continue obtaining mini-batches with `next`.

Examples

Iterate Over All Mini-Batches

Use `hasdata` with a `while` loop to iterate over all data in the `minibatchqueue` object.

Create a `minibatchqueue` object from a datastore.

```
ds = digitDatastore;
mbq = minibatchqueue(ds,'MinibatchSize',256)
```

```
mbq =
minibatchqueue with 1 output and properties:

   Mini-batch creation:
           MiniBatchSize: 256
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
     DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

While data is still available in the `minibatchqueue` object, obtain the next mini-batch.

```
while hasdata(mbq)
    X = next(mbq)
end
```

The loop ends when `hasdata` returns false, and all mini-batches are returned.

Input Arguments

`mbq` — Queue of mini-batches

`minibatchqueue`

Queue of mini-batches, specified as a `minibatchqueue` object.

Output Arguments

`tf` — True or false result

`1` | `0`

True or false result, returned as a `1` or `0` of data type logical.

See Also

`shuffle` | `reset` | `minibatchqueue` | `next`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Generative Adversarial Network (GAN)”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

huber

Huber loss for regression tasks

Syntax

```
loss = huber(dlY,targets)
loss = huber(dlY,targets,weights)
loss = huber( ___ ,'DataFormat',FMT)
loss = huber( ___ ,Name,Value)
```

Description

The Huber operation computes the Huber loss between network predictions and target values for regression tasks. When the `'TransitionPoint'` option is 1, this is also known as *smooth L_1 loss*.

The `huber` function calculates the Huber loss using `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

`loss = huber(dlY,targets)` returns the Huber loss between the formatted `dlarray` object `dlY` containing the predictions and the target values `targets` for regression tasks. The input `dlY` is a formatted `dlarray`. The output `loss` is an unformatted `dlarray` scalar.

For unformatted input data, use the `'DataFormat'` option.

`loss = huber(dlY,targets,weights)` applies weights to the calculated loss vales. Use this syntax to weight the contributions of classes, observations, or regions of the input to the calculated loss values.

`loss = huber( ___ ,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlY` is not a formatted `dlarray`.

`loss = huber( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'NormalizationFactor','all-elements'` specifies to normalize the loss by dividing the reduced loss by the number of input elements.

Examples

Huber Loss

Create an array of predictions for 12 observations over 10 responses.

```
numResponses = 10;
numObservations = 12;
```

```
Y = rand(numResponses,numObservations);
dlY = dlarray(Y,'CB');
```

View the size and format of the predictions.

```
size(dlY)
```

```
ans = 1×2

    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create an array of random targets.

```
targets = rand(numResponses,numObservations);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the Huber loss between the predictions and the targets.

```
loss = huber(dlY,targets)
```

```
loss =
  1x1 dlarray

    0.7374
```

Masked Huber Loss for Padded Sequences

Create arrays of predictions and targets for 12 sequences of varying lengths over 10 responses.

```
numResponses = 10;
numObservations = 12;
maxSequenceLength = 15;

sequenceLengths = randi(maxSequenceLength,[1 numObservations]);

Y = cell(numObservations,1);
targets = cell(numObservations,1);

for i = 1:numObservations
    Y{i} = rand(numResponses,sequenceLengths(i));
    targets{i} = rand(numResponses,sequenceLengths(i));
end
```

View the cell arrays of predictions and targets.

```
Y
```

```
Y=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

```
targets
```

```
targets=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

Pad the prediction and target sequences in the second dimension using the `padsequences` function and also return the corresponding mask.

```
[Y,mask] = padsequences(Y,2);
targets = padsequences(targets,2);
```

Convert the padded sequences to `dlarray` with format `'CTB'` (channel, time, batch). Because formatted `dlarray` objects automatically sort the dimensions, keep the dimensions of the targets and mask consistent by also converting them to a formatted `dlarray` objects with the same formats.

```
dlY = dlarray(Y,'CTB');
targets = dlarray(targets,'CTB');
mask = dlarray(mask,'CTB');
```

View the sizes of the prediction scores, targets, and the mask.

```
size(dlY)
```

```
ans = 1×3

    10    12    15
```

```
size(targets)
```

```
ans = 1×3

    10    12    15
```

```
size(mask)
```

```
ans = 1×3

    10    12    15
```

Compute the Huber loss between the predictions and the targets. To prevent the loss values calculated from padding from contributing to the loss, set the `'Mask'` option to the mask returned by the `padsequences` function.

```
loss = huber(dlY,targets,'Mask',mask)
```

```
loss =
  1x1 dlarray

    8.1834
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `dlY` is a numeric array, `targets` must be a `dlarray`.

`targets` — Target responses

`dlarray` | numeric array

Target responses, specified as a formatted or unformatted `dlarray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `dlY`.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`weights` — Weights
`dlarray` | numeric array

Weights, specified as a `dlarray` or a numeric array.

To specify response weights, specify a vector with a `'C'` (channel) dimension with size matching the `'C'` (channel) dimension of the `dlX`. Specify the `'C'` (channel) dimension of the response weights by using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

To specify observation weights, specify a vector with a `'B'` (batch) dimension with size matching the `'B'` (batch) dimension of the `dlY`. Specify the `'B'` (batch) dimension of the class weights by using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

To specify weights for each element of the input independently, specify the weights as an array of the same size as `dlY`. In this case, if `weights` is not a formatted `dlarray` object, then the function uses the same format as `dlY`. Alternatively, specify the weights format using the `'WeightsFormat'` option.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NormalizationFactor','all-elements'` specifies to normalize the loss by dividing the reduced loss by the number of input elements

`TransitionPoint` — Point where Huber loss transitions to a linear function
1 (default) | positive scalar

Point where Huber loss transitions from a quadratic function to a linear function, specified as the comma-separated pair consisting of `'TransitionPoint'` and a positive scalar.

When `'TransitionPoint'` is 1, this is also known as *smooth L_1 loss*.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Mask` — Mask indicating which elements to include for loss computation
`dlarray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the input data for loss computation when the corresponding value in the mask is 1 and 0, respectively.

The default value is a logical array of ones with the same size as `dlY`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

`Reduction` — Mode for reducing array of loss values
`"sum"` (default) | `"none"`

Mode for reducing the array of loss values, specified as one of the following:

- `"sum"` — Sum all of the elements in the array of loss values. In this case, the output `loss` is scalar.
- `"none"` — Do not reduce the array of loss values. In this case, the output `loss` is an unformatted `dlarray` object with the same size as `dlY`.

`NormalizationFactor` — Divisor for normalizing reduced loss
`"batch-size"` (default) | `"all-elements"` | `"mask-included"` | `"none"`

Divisor for normalizing the reduced loss when `Reduction` is `"sum"`, specified as one of the following:

- `"batch-size"` — Normalize the loss by dividing it by the number of observations in `dlX`.
- `"all-elements"` — Normalize the loss by dividing it by the number of elements of `dlX`.
- `"mask-included"` — Normalize the loss by dividing the loss values by the number of included elements specified by the mask for each observation independently. To use this option, you must specify a mask using the `Mask` option.
- `"none"` — Do not normalize the loss.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`WeightsFormat` — Dimension order of weights
character vector | string scalar

Dimension order of the weights, specified as a character vector or string scalar that provides a label for each dimension of the weights.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `WeightsFormat` when `weights` is a numeric vector and `dlY` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `dlY` are vectors, then default value of `WeightsFormat` is the same as the format of `dlY`.

Data Types: `char` | `string`

Output Arguments

`loss` — Huber loss

`dlarray`

Huber loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `dlY`.

The size of `loss` depends on the `Reduction` option.

Algorithms

Huber Loss

For each element Y_j of the input, the `huber` function computes the corresponding element-wise loss values using the formula

$$\text{loss}_j = \begin{cases} \frac{1}{2}(Y_j - T_j)^2 & \text{if } |Y_j - T_j| \le \delta \\ \delta|Y_j - T_j| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases},$$

where T_j is the corresponding target value to the prediction Y_j and δ is the transition point where the loss transitions from a quadratic function to a linear function.

When the transition point is 1, this is also known as *smooth L_1 loss*.

To reduce the loss values to a scalar, the function then reduces the element-wise loss using the formula

$$\text{loss} = -\frac{1}{N}\sum_j m_j w_j \text{loss}_j,$$

where N is the normalization factor, m_j is the mask value for element j, and w_j is the weight value for element j.

If you do not opt to reduce the loss, then the function applies the mask and the weights to the loss values directly:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlY`
  - `targets`
  - `weights`
  - `'Mask'`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `crossentropy` | `mse` | `l1loss` | `l2loss`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2021a

imageDataAugmenter

Configure image data augmentation

Description

An image data augmenter configures a set of preprocessing options for image augmentation, such as resizing, rotation, and reflection.

The `imageDataAugmenter` is used by an `augmentedImageDatastore` to generate batches of augmented images. For more information, see “Augment Images for Training with Random Geometric Transformations”.

Creation

Syntax

```
aug = imageDataAugmenter
aug = imageDataAugmenter(Name,Value)
```

Description

`aug = imageDataAugmenter` creates an `imageDataAugmenter` object with default property values consistent with the identity transformation.

`aug = imageDataAugmenter(Name,Value)` configures a set of image augmentation options using name-value pairs to set properties on page 1-734. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`FillValue` — Fill value
numeric scalar | numeric vector

Fill value used to define out-of-bounds points when resampling, specified as a numeric scalar or numeric vector.

- If the augmented images are single channel, then `FillValue` must be a scalar.
- If the augmented images are multichannel, then `FillValue` can be a scalar or a vector with length equal to the number of channels of the input image. For example, if the input image is an RGB image, `FillValue` can be a vector of length 3.

For grayscale and color images, the default fill value is `0`. For categorical images, the default fill value is an `'<undefined>'` label and `trainNetwork` ignores filled pixels when training.

Example: 128

`RandXReflection` — Random reflection
`false` (default) | `true`

Random reflection in the left-right direction, specified as a logical scalar. When `RandXReflection` is `true` (`1`), each image is reflected horizontally with 50% probability. When `RandXReflection` is `false` (`0`), no images are reflected.

`RandYReflection` — Random reflection
`false` (default) | `true`

Random reflection in the top-bottom direction, specified as a logical scalar. When `RandYReflection` is `true` (`1`), each image is reflected vertically with 50% probability. When `RandYReflection` is `false` (`0`), no images are reflected.

`RandRotation` — Range of rotation
`[0 0]` (default) | 2-element numeric vector | function handle

Range of rotation, in degrees, applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The rotation angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the rotation angle as a numeric scalar. Use a function handle to pick rotation angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not rotated.

Example: `[-45 45]`

`RandScale` — Range of uniform scaling
`[1 1]` (default) | 2-element numeric vector | function handle

Range of uniform (isotropic) scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the scale factor as a numeric scalar. Use a function handle to pick scale factors from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not scaled.

Example: `[0.5 4]`

`RandXScale` — Range of horizontal scaling
`[1 1]` (default) | 2-element vector of positive numbers | function handle

Range of horizontal scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal scale factor as a numeric scalar. Use a function handle to pick horizontal scale factors from a disjoint

interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not scaled in the horizontal direction.

Note If you specify `RandScale`, then `imageDataAugmenter` ignores the value of `RandXScale` when scaling images.

Example: `[0.5 4]`

`RandYScale` — Range of vertical scaling
`[1 1]` (default) | 2-element vector of positive numbers | function handle

Range of vertical scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the vertical scale factor as a numeric scalar. Use a function handle to pick vertical scale factors from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not scaled in the vertical direction.

Note If you specify `RandScale`, then `imageDataAugmenter` ignores the value of `RandYScale` when scaling images.

Example: `[0.5 4]`

`RandXShear` — Range of horizontal shear
`[0 0]` (default) | 2-element numeric vector | function handle

Range of horizontal shear applied to the input image, specified as one of the following. Shear is measured as an angle in degrees, and is in the range (–90, 90).

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal shear angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal shear angle as a numeric scalar. Use a function handle to pick horizontal shear angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not sheared in the horizontal direction.

Example: `[0 45]`

`RandYShear` — Range of vertical shear
`[0 0]` (default) | 2-element numeric vector | function handle

Range of vertical shear applied to the input image, specified as one of the following. Shear is measured as an angle in degrees, and is in the range (–90, 90).

- 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical shear angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the vertical shear angle as a numeric scalar. Use a function handle to pick vertical shear angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not sheared in the vertical direction.

Example: `[0 45]`

`RandXTranslation` — Range of horizontal translation
`[0 0]` (default) | 2-element numeric vector | function handle

Range of horizontal translation applied to the input image, specified as one of the following. Translation distance is measured in pixels.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal translation distance is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal translation distance as a numeric scalar. Use a function handle to pick horizontal translation distances from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not translated in the horizontal direction.

Example: `[-5 5]`

`RandYTranslation` — Range of vertical translation
`[0 0]` (default) | 2-element numeric vector | function handle

Range of vertical translation applied to the input image, specified as one of the following. Translation distance is measured in pixels.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical translation distance is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the vertical translation distance as a numeric scalar. Use a function handle to pick vertical translation distances from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle”.

By default, augmented images are not translated in the vertical direction.

Example: `[-5 5]`

Object Functions

augment Apply identical random transformations to multiple images

Examples

Create Image Data Augmenter to Resize and Rotate Images

Create an image data augmenter that preprocesses images before training. This augmenter rotates images by random angles in the range [0, 360] degrees and resizes images by random scale factors in the range [0.5, 1].

```
augmenter = imageDataAugmenter( ...
    'RandRotation',[0 360], ...
    'RandScale',[0.5 1])
```

```
augmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [0 360]
           RandScale: [0.5000 1]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [0 0]
    RandYTranslation: [0 0]
```

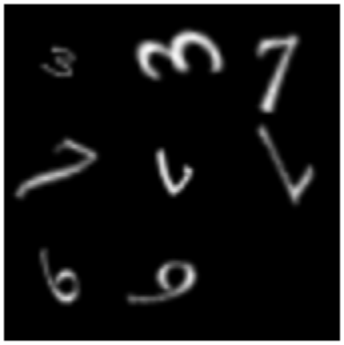
Create an augmented image datastore using the image data augmenter. The augmented image datastore also requires sample data, labels, and an output image size.

```
[XTrain,YTrain] = digitTrain4DArrayData;
imageSize = [56 56 1];
auimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',augmenter)
```

```
auimds =
  augmentedImageDatastore with properties:

         NumObservations: 5000
           MiniBatchSize: 128
        DataAugmentation: [1x1 imageDataAugmenter]
      ColorPreprocessing: 'none'
              OutputSize: [56 56]
          OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

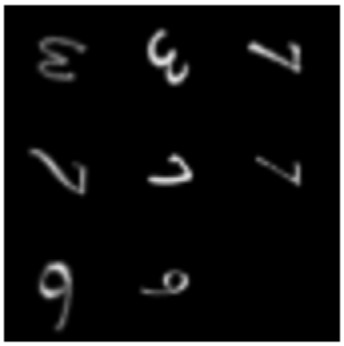
Preview the random transformations applied to the first eight images in the image datastore.

```
minibatch = preview(auimds);
imshow(imtile(minibatch.input));
```



Preview different random transformations applied to the same set of images.

```
minibatch = preview(auimds);
imshow(imtile(minibatch.input));
```



Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

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

`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 width of the images.

- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

```
idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,:,idx);
XTrain(:,:,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Create an `imageDataAugmenter` object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

```
imageAugmenter = imageDataAugmenter( ...
    'RandRotation',[-20,20], ...
    'RandXTranslation',[-3 3], ...
    'RandYTranslation',[-3 3])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-20 20]
           RandScale: [1 1]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [-3 3]
    RandYTranslation: [-3 3]
```

Create an `augmentedImageDatastore` object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

Specify the convolutional neural network architecture.

```
layers = [
    imageInputLayer(imageSize)

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
```

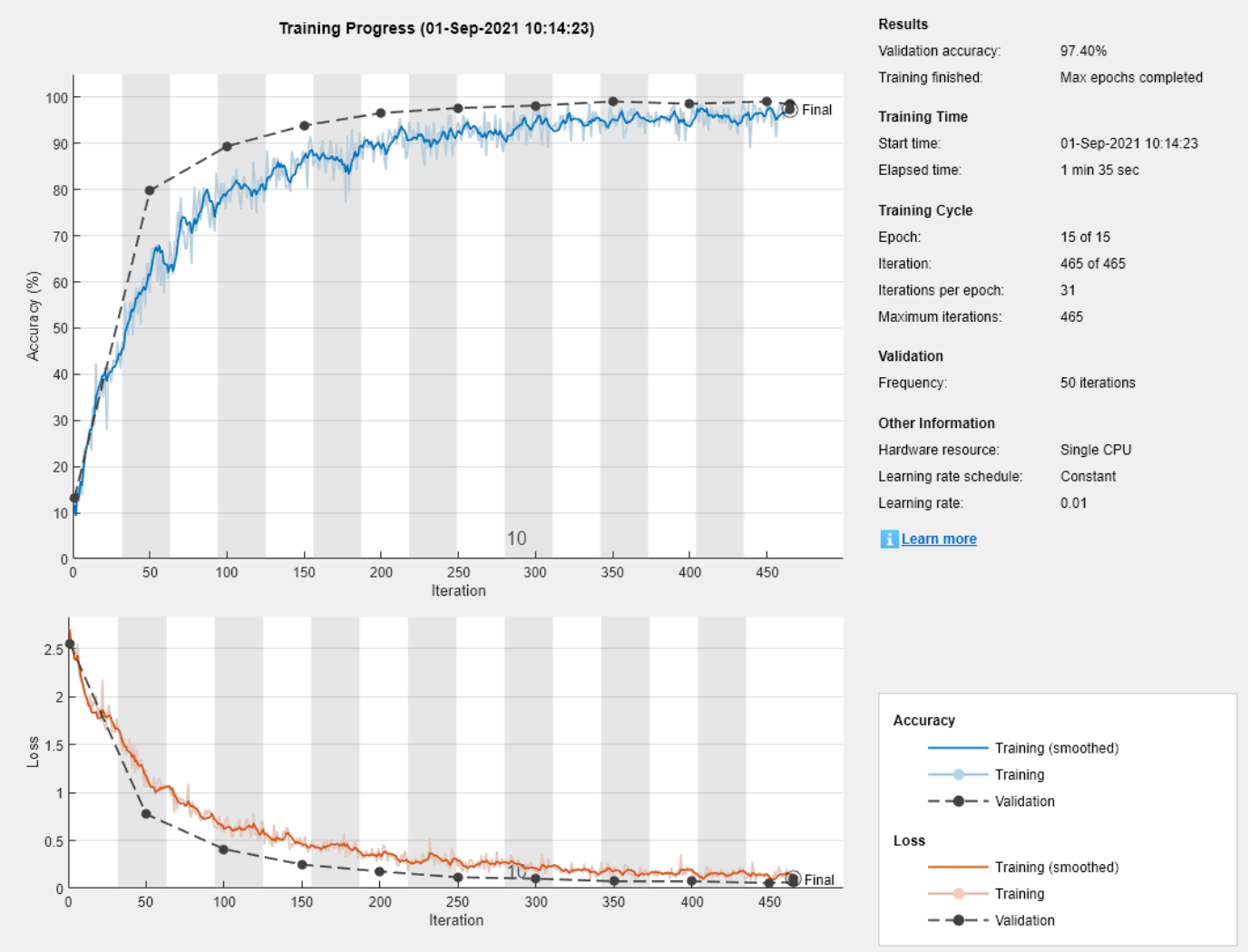
```
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify training options for stochastic gradient descent with momentum.

```
opts = trainingOptions('sgdm', ...
    'MaxEpochs',15, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation});
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```
net = trainNetwork(augimds,layers,opts);
```



Tips

- To preview the transformations applied to sample images, use the `augment` function.
- To perform image augmentation during training, create an `augmentedImageDatastore` and specify preprocessing options by using the `'DataAugmentation'` name-value pair with an `imageDataAugmenter`. The augmented image datastore automatically applies random transformations to the training data.

See Also

`augmentedImageDatastore` | `imageInputLayer` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”
“Create Function Handle”

Introduced in R2017b

image3dInputLayer

3-D image input layer

Description

A 3-D image input layer inputs 3-D images or volumes to a network and applies data normalization.

For 2-D image input, use `imageInputLayer`.

Creation

Syntax

```
layer = image3dInputLayer(inputSize)
layer = image3dInputLayer(inputSize,Name,Value)
```

Description

`layer = image3dInputLayer(inputSize)` returns a 3-D image input layer and specifies the `InputSize` property.

`layer = image3dInputLayer(inputSize,Name,Value)` sets the optional properties using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

3-D Image Input

`InputSize` — Size of the input
row vector of integers

Size of the input data, specified as a row vector of integers `[h w d c]`, where `h`, `w`, `d`, and `c` correspond to the height, width, depth, and number of channels respectively.

- For grayscale input, specify a vector with `c` equal to `1`.
- For RGB input, specify a vector with `c` equal to `3`.
- For multispectral or hyperspectral input, specify a vector with `c` equal to the number of channels.

For 2-D image input, use `imageInputLayer`.

Example: `[132 132 116 3]`

`Normalization` — Data normalization
`'zerocenter'` (default) | `'zscore'` | `'rescale-symmetric'` | `'rescale-zero-one'` | `'none'` | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by `Mean`.
- `'zscore'` — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form `Y = func(X)`, where `X` is the input data and the output `Y` is the normalized data.

Tip The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

`NormalizationDimension` — Normalization dimension

`'auto'` (default) | `'channel'` | `'element'` | `'all'`

Normalization dimension, specified as one of the following:

- `'auto'` – If the training option is `false` and you specify any of the normalization statistics (`Mean`, `StandardDeviation`, `Min`, or `Max`), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- `'channel'` – Channel-wise normalization.
- `'element'` – Element-wise normalization.
- `'all'` – Normalize all values using scalar statistics.

Data Types: `char` | `string`

`Mean` — Mean for zero-center and z-score normalization

`[]` (default) | 4-D array | numeric scalar

Mean for zero-center and z-score normalization, specified as a *h*-by-*w*-by-*d*-by-*c* array, a 1-by-1-by-1-by-*c* array of means per channel, a numeric scalar, or `[]`, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and the number of channels of the mean, respectively.

If you specify the `Mean` property, then `Normalization` must be `'zerocenter'` or `'zscore'`. If `Mean` is `[]`, then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StandardDeviation` — Standard deviation for z-score normalization

`[]` (default) | 4-D array | numeric scalar

Standard deviation for z-score normalization, specified as a *h*-by-*w*-by-*d*-by-*c* array, a 1-by-1-by-1-by-*c* array of means per channel, a numeric scalar, or `[]`, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and the number of channels of the standard deviation, respectively.

If you specify the `StandardDeviation` property, then `Normalization` must be `'zscore'`. If `StandardDeviation` is `[]`, then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Min` — Minimum value for rescaling
`[]` (default) | 4-D array | numeric scalar

Minimum value for rescaling, specified as a *h*-by-*w*-by-*d*-by-*c* array, a 1-by-1-by-1-by-*c* array of minima per channel, a numeric scalar, or `[]`, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and the number of channels of the minima, respectively.

If you specify the `Min` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Min` is `[]`, then the software calculates the minimum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Max` — Maximum value for rescaling
`[]` (default) | 4-D array | numeric scalar

Maximum value for rescaling, specified as a *h*-by-*w*-by-*d*-by-*c* array, a 1-by-1-by-1-by-*c* array of maxima per channel, a numeric scalar, or `[]`, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and the number of channels of the maxima, respectively.

If you specify the `Min` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Max` is `[]`, then the software calculates the maximum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

`InputNames` — Input names
`{}` (default)

Input names of the layer. The layer has no inputs.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 3-D Image Input Layer

Create a 3-D image input layer for 132-by-132-by-116 color 3-D images with name `'input'`. By default, the layer performs data normalization by subtracting the mean image of the training set from every input image.

```
layer = image3dInputLayer([132 132 116],'Name','input')
```

```
layer =
  Image3DInputLayer with properties:

                     Name: 'input'
                InputSize: [132 132 116 1]

   Hyperparameters
              Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                     Mean: []
```

Include a 3-D image input layer in a `Layer` array.

```
layers = [
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,16,'Stride',4)
    reluLayer
    maxPooling3dLayer(2,'Stride',4)
```

```
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input         28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             16 5x5x5 convolutions with stride [4  4  4] and padding [0
     3   ''   ReLU                    ReLU
     4   ''   3-D Max Pooling         2x2x2 max pooling with stride [4  4  4] and padding [0  0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Compatibility Considerations

`AverageImage` property will be removed
Not recommended starting in R2019b

`AverageImage` will be removed. Use `Mean` instead. To update your code, replace all instances of `AverageImage` with `Mean`. There are no differences between the properties that require additional updates to your code.

`imageInputLayer` and `image3dInputLayer`, by default, use channel-wise normalization
Behavior change in future release

Starting in R2019b, `imageInputLayer` and `image3dInputLayer`, by default, use channel-wise normalization. In previous versions, these layers use element-wise normalization. To reproduce this behavior, set the `NormalizationDimension` option of these layers to `'element'`.

See Also

`trainNetwork` | `convolution3dLayer` | `transposedConv3dLayer` | `averagePooling3dLayer` | `maxPooling3dLayer` | `fullyConnectedLayer` | `imageInputLayer`

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a

imageInputLayer

Image input layer

Description

An image input layer inputs 2-D images to a network and applies data normalization.

For 3-D image input, use `image3dInputLayer`.

Creation

Syntax

```
layer = imageInputLayer(inputSize)
layer = imageInputLayer(inputSize,Name,Value)
```

Description

`layer = imageInputLayer(inputSize)` returns an image input layer and specifies the `InputSize` property.

`layer = imageInputLayer(inputSize,Name,Value)` sets the optional properties on page 1-749 using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

Image Input

`InputSize` — Size of the input
row vector of integers

Size of the input data, specified as a row vector of integers `[h w c]`, where `h`, `w`, and `c` correspond to the height, width, and number of channels respectively.

- For grayscale images, specify a vector with `c` equal to `1`.
- For RGB images, specify a vector with `c` equal to `3`.
- For multispectral or hyperspectral images, specify a vector with `c` equal to the number of channels.

For 3-D image or volume input, use `image3dInputLayer`.

Example: `[224 224 3]`

`Normalization` — Data normalization
`'zerocenter'` (default) | `'zscore'` | `'rescale-symmetric'` | `'rescale-zero-one'` | `'none'` | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by `Mean`.
- `'zscore'` — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form `Y = func(X)`, where `X` is the input data and the output `Y` is the normalized data.

Tip The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

`NormalizationDimension` — Normalization dimension
`'auto'` (default) | `'channel'` | `'element'` | `'all'`

Normalization dimension, specified as one of the following:

- `'auto'` – If the training option is `false` and you specify any of the normalization statistics (`Mean`, `StandardDeviation`, `Min`, or `Max`), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- `'channel'` – Channel-wise normalization.
- `'element'` – Element-wise normalization.
- `'all'` – Normalize all values using scalar statistics.

Data Types: `char` | `string`

`Mean` — Mean for zero-center and z-score normalization
`[]` (default) | 3-D array | numeric scalar

Mean for zero-center and z-score normalization, specified as a *h*-by-*w*-by-*c* array, a 1-by-1-by-*c* array of means per channel, a numeric scalar, or `[]`, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the mean, respectively.

If you specify the `Mean` property, then `Normalization` must be `'zerocenter'` or `'zscore'`. If `Mean` is `[]`, then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StandardDeviation` — Standard deviation for z-score normalization
`[]` (default) | 3-D array | numeric scalar

Standard deviation for z-score normalization, specified as a *h*-by-*w*-by-*c* array, a 1-by-1-by-*c* array of means per channel, a numeric scalar, or `[]`, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the standard deviation, respectively.

If you specify the `StandardDeviation` property, then `Normalization` must be `'zscore'`. If `StandardDeviation` is `[]`, then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Min` — Minimum value for rescaling
`[]` (default) | 3-D array | numeric scalar

Minimum value for rescaling, specified as a *h*-by-*w*-by-*c* array, a 1-by-1-by-*c* array of minima per channel, a numeric scalar, or `[]`, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the minima, respectively.

If you specify the `Min` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Min` is `[]`, then the software calculates the minimum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Max` — Maximum value for rescaling
`[]` (default) | 3-D array | numeric scalar

Maximum value for rescaling, specified as a *h*-by-*w*-by-*c* array, a 1-by-1-by-*c* array of maxima per channel, a numeric scalar, or `[]`, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the maxima, respectively.

If you specify the `Max` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Max` is `[]`, then the software calculates the maximum at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataAugmentation` — Data augmentation transforms
`'none'` (default) | `'randcrop'` | `'randfliplr'` | cell array of `'randcrop'` and `'randfliplr'`

Note The `DataAugmentation` property is not recommended. To preprocess images with cropping, reflection, and other geometric transformations, use `augmentedImageDatastore` instead.

Data augmentation transforms to use during training, specified as one of the following.

- `'none'` — No data augmentation
- `'randcrop'` — Take a random crop from the training image. The random crop has the same size as the input size.

- `'randfliplr'` — Randomly flip the input images horizontally with a 50% chance.
- Cell array of `'randcrop'` and `'randfliplr'`. The software applies the augmentation in the order specified in the cell array.

Augmentation of image data is another way of reducing overfitting [1], [2].

Data Types: `string` | `char` | `cell`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

`InputNames` — Input names
`{}` (default)

Input names of the layer. The layer has no inputs.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Image Input Layer

Create an image input layer for 28-by-28 color images with name `'input'`. By default, the layer performs data normalization by subtracting the mean image of the training set from every input image.

```
inputlayer = imageInputLayer([28 28 3],'Name','input')
```

```
inputlayer =
  ImageInputLayer with properties:

                     Name: 'input'
                InputSize: [28 28 3]

   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                     Mean: []
```

Include an image input layer in a `Layer` array.

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

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Compatibility Considerations

`AverageImage` property will be removed
Not recommended starting in R2019b

`AverageImage` will be removed. Use `Mean` instead. To update your code, replace all instances of `AverageImage` with `Mean`. There are no differences between the properties that require additional updates to your code.

`imageInputLayer` and `image3dInputLayer`, by default, use channel-wise normalization
Behavior change in future release

Starting in R2019b, `imageInputLayer` and `image3dInputLayer`, by default, use channel-wise normalization. In previous versions, these layers use element-wise normalization. To reproduce this behavior, set the `NormalizationDimension` option of these layers to `'element'`.

References

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

[2] Cireşan, D., U. Meier, J. Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support `'Normalization'` specified using a function handle.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support `'Normalization'` specified using a function handle.

See Also

`trainNetwork` | `convolution2dLayer` | `fullyConnectedLayer` | `maxPooling2dLayer` | `augmentedImageDatastore` | `image3dInputLayer` | **Deep Network Designer** | `featureInputLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

imageLIME

Explain network predictions using LIME

Syntax

```
scoreMap = imageLIME(net,X,label)
[scoreMap,featureMap,featureImportance] = imageLIME(net,X,label)
___ = imageLIME( ___ ,Name,Value)
```

Description

`scoreMap = imageLIME(net,X,label)` uses the locally-interpretable model-agnostic explanation (LIME) technique to compute a map of the importance of the features in the input image `X` when the network `net` evaluates the class score for the class given by `label`. Use this function to explain classification decisions and check that your network is focusing on the appropriate features of the image.

The LIME technique approximates the classification behavior of the `net` using a simpler, more interpretable model. By generating synthetic data from input `X`, classifying the synthetic data using `net`, and then using the results to fit a simple regression model, the `imageLIME` function determines the importance of each feature of `X` to the network's classification score for class given by `label`.

This function requires Statistics and Machine Learning Toolbox.

`[scoreMap,featureMap,featureImportance] = imageLIME(net,X,label)` also returns a map of the features used to compute the LIME results and the calculated importance of each feature.

`___ = imageLIME( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'NumFeatures',100` sets the target number of features to 100.

Examples

Visualize Which Parts of an Image are Important for Classification

Use `imageLIME` to visualize the parts of an image are important to a network for a classification decision.

Import the pretrained network SqueezeNet.

```
net = squeezenet;
```

Import the image and resize to match the input size for the network.

```
X = imread("laika_grass.jpg");
inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Display the image. The image is of a dog named Laika.

```
imshow(X)
```

Classify the image to get the class label.

```
label = classify(net,X)
```

```
label = categorical
     toy poodle
```

Use `imageLIME` to determine which parts of the image are important to the classification result.

```
scoreMap = imageLIME(net,X,label);
```

Plot the result over the original image with transparency to see which areas of the image affect the classification score.

```
figure
imshow(X)
hold on
imagesc(scoreMap,'AlphaData',0.5)
colormap jet
```

The network focuses predominantly on Laika's head and back to make the classification decision. Laika's eye and ear are also important to the classification result.

Visualize Only the Most Important Features

Use `imageLIME` to determine the most important features in an image and isolate them from the unimportant features.

Import the pretrained network SqueezeNet.

```
net = squeezenet;
```

Import the image and resize to match the input size for the network.

```
X = imread("sherlock.jpg");
inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Classify the image to get the class label.

```
label = classify(net,X)
```

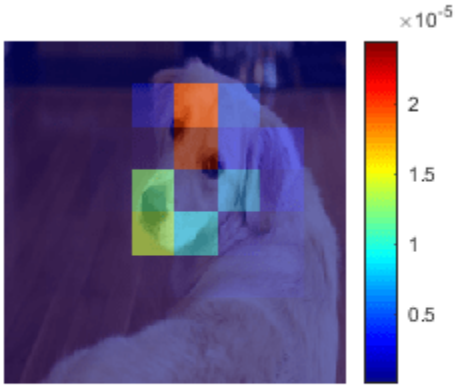
```
label = categorical
     golden retriever
```

Compute the map of the feature importance and also obtain the map of the features and the feature importance. Set the image segmentation method to `'grid'`, the number of features to `64`, and the number of synthetic images to `3072`.

```
[scoreMap,featureMap,featureImportance]  = imageLIME(net,X,label,'Segmentation','grid','NumFeatu
```

Plot the result over the original image with transparency to see which areas of the image affect the classification score.

```
figure
imshow(X)
hold on
imagesc(scoreMap,'AlphaData',0.5)
colormap jet
colorbar
```

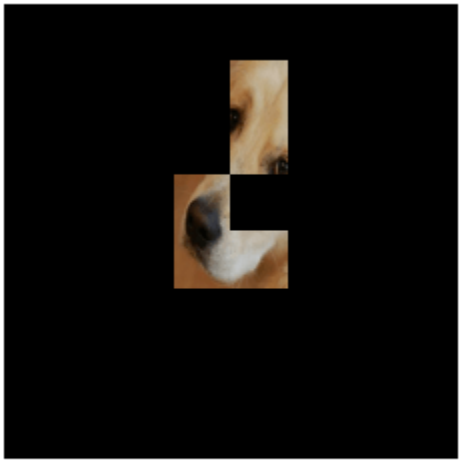


Use the feature importance to find the indices of the most important five features.

```
numTopFeatures = 5;
[~,idx] = maxk(featureImportance,numTopFeatures);
```

Use the map of the features to mask out the image so only the most important five features are visible. Display the masked image.

```
mask = ismember(featureMap,idx);
maskedImg = uint8(mask).*X;
figure
imshow(maskedImg);
```



View Important Features Using Custom Segmentation Map

Use `imageLIME` with a custom segmentation map to view the most important features for a classification decision.

Import the pretrained network GoogLeNet.

```
net = googlenet;
```

Import the image and resize to match the input size for the network.

```
X = imread("sherlock.jpg");
inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Classify the image to get the class label.

```
label = classify(net,X)
```

```
label = categorical
     golden retriever
```

Create a matrix defining a custom segmentation map which divides the image into triangular segments. Each triangular segment represents a feature.

Start by defining a matrix with size equal to the input size of the image.

```
segmentationMap = zeros(inputSize(1));
```

Next, create a smaller segmentation map which divides a 56-by-56 pixel region into two triangular features. Assign values 1 and 2 to the upper and lower segments, representing the first and second features, respectively.

```
blockSize = 56;

segmentationSubset = ones(blockSize);
segmentationSubset = tril(segmentationSubset) + segmentationSubset;

% Set the diagonal elements to alternate values 1 and 2.
segmentationSubset(1:(blockSize+1):end) = repmat([1 2],1,blockSize/2)';
```

To create a custom segmentation map for the whole image, repeat the small segmentation map. Each time you repeat the smaller map, increase the feature index values so that the pixels in each triangular segment correspond to a unique feature. In the final matrix, value 1 indicates the first feature, value 2 the second feature, and so on for each segment in the image.

```
blocksPerSide = inputSize(1)/blockSize;
subset = 0;
for i=1:blocksPerSide
    for j=1:blocksPerSide
        xidx = (blockSize*(i-1))+1:(blockSize*i);
        yidx = (blockSize*(j-1))+1:(blockSize*j);
        segmentationMap(xidx,yidx) = segmentationSubset + 2*subset;
        subset = subset + 1;
    end
end
```

View the segmentation map. This map divides the image into 32 triangular regions.

```
figure
imshow(X)
hold on
imagesc(segmentationMap,'AlphaData',0.8);
title('Custom Segmentation Map')
colormap gray
```

Use `imageLIME` with the custom segmentation map to determine which parts of the image are most important to the classification result.

```
scoreMap = imageLIME(net,X,label, ...
    'Segmentation',segmentationMap);
```

Plot the result of `imageLIME` over the original image to see which areas of the image affect the classification score.

```
figure;
imshow(X)
hold on
title('Image LIME (Golden Retriever)')
colormap jet;
imagesc(scoreMap, "AlphaData", 0.5);
```

Red areas of the map have a higher importance — when these areas are removed, the score for the golden retriever class goes down. The most important feature for this classification is the ear.

Input Arguments

`net` — Image classification network

`SeriesNetwork` object | `DAGNetwork` object

Image classification network, specified as a `SeriesNetwork` object or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function. For more information about pretrained networks, see “Pretrained Deep Neural Networks”.

`net` must contain a single input layer and a single output layer. The input layer must be an `imageInputLayer`. The output layer must be a `classificationLayer`.

`X` — Input image

numeric array

Input image, specified as a numeric array.

The image must be the same size as the image input size of the network `net`. The input size is specified by the `InputSize` property of the network's `imageInputLayer`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`label` — Class label

categorical | char vector | string scalar | vector

Class label used to calculate the feature importance map, specified as a categorical, a char vector, a string scalar or a vector of these values.

If you specify `label` as a vector, the software calculates the feature importance for each class label independently. In that case, `scoreMap(:,:,k)` and `featureImportance(idx,k)` correspond to the map of feature importance and the importance of feature `idx` for the `k`th element in `label`, respectively.

Example: `["cat" "dog"]`

Data Types: `char` | `string` | `categorical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumFeatures',100,'Segmentation','grid', 'OutputUpsampling','bicubic','ExecutionEnvironment','gpu'` segments the input image into a grid of approximately 100 features, executes the calculation on the GPU, and upsamples the resulting map to the same size as the input image using bicubic interpolation.

`NumFeatures` — Target number of features

`49` (default) | positive integer

Target number of features to divide the input image into, specified as the comma-separated pair consisting of `'NumFeatures'` and a positive integer.

A larger value of `'NumFeatures'` divides the input image into more, smaller features. To get the best results when using a larger number of features, also increase the number of synthetic images using the `'NumSamples'` name-value pair.

The exact number of features depends on the input image and segmentation method specified using the `'Segmentation'` name-value pair and can be less than the target number of features.

- When you specify `'Segmentation','superpixels'`, the actual number of features can be greater or less than the number specified using `'NumFeatures'`.
- When you specify `'Segmentation','grid'`, the actual number of features can be less than the number specified using `'NumFeatures'`. If your input image is square, specify `'NumFeatures'` as a square number.
- When you specify `'Segmentation',segmentation`, where `segmentation` is a two-dimensional matrix, `'NumFeatures'` is the same as the number of unique elements in the matrix.

Example: `'NumFeatures',100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumSamples` — Number of synthetic images

`2048` (default) | positive integer

Number of synthetic images to generate, specified as the comma-separated pair consisting of `'NumSamples'` and a positive integer.

A larger number of synthetic images gives better results but takes more time to compute.

Example: `'NumSamples',1024`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Segmentation — Segmentation method
'superpixels' (default) | 'grid' | numeric matrix

Segmentation method to use to divide the input image into features, specified as the comma-separated pair consisting of 'Segmentation' and 'superpixels', 'grid', or a two-dimensional segmentation matrix.

The imageLIME function segments the input image into features in the following ways depending on the segmentation method.

- 'superpixels' — Input image is divided into superpixel features, using the superpixels function. Features are irregularly shaped, based on the value of the pixels. This option requires Image Processing Toolbox.
- 'grid' — Input image is divided into a regular grid of features. Features are approximately square, based on the aspect ratio of the input image and the specified value of 'NumFeatures'. The number of grid cells can be smaller than the specified value of 'NumFeatures'. If the input image is square, specify 'NumFeatures' as a square number.
- numeric matrix — Input image is divided into custom features, using the numeric matrix as a map, where the integer value of each pixel specifies the feature of the corresponding pixel. 'NumFeatures' is the same as the number of unique elements in the matrix. The size of the matrix must match the size of the input image.

For photographic image data, the 'superpixels' option usually gives better results. In this case, features are based on the contents of the image, by segmenting the image into regions of similar pixel value. For other types of images, such as spectrograms, the more regular 'grid' option or a custom segmentation map can provide more useful results.

Example: 'Segmentation','grid'

Model — Type of simple model
'tree' (default) | 'linear'

Type of simple model to fit, specified as the specified as the comma-separated pair consisting of 'Model' and 'tree' or 'linear'.

The imageLIME function classifies the synthetic images using the network net and then uses the results to fit a simple, interpretable model. The methods used to fit the results and determine the importance of each feature depend on the type of simple model used.

- 'tree' — Fit a regression tree using fitrtree then compute the importance of each feature using predictorImportance
- 'linear' — Fit a linear model with lasso regression using fitrlinear then compute the importance of each feature using the weights of the linear model.

Example: 'Model','linear'

Data Types: char | string

OutputUpsampling — Output upsampling method
'nearest' (default) | 'bicubic' | 'none'

Output upsampling method to use when segmentation method is 'grid', specified as the comma-separated pair consisting of 'OutputUpsampling' and one of the following.

- `'nearest'` — Use nearest-neighbor interpolation expand the map to the same size as the input data. The map indicates the size of the each feature with respect to the size of the input data.
- `'bicubic'` — Use bicubic interpolation to produce a smooth map the same size as the input data.
- `'none'` — Use no upsampling. The map can be smaller than the input data.

If `'OutputUpsampling'` is `'nearest'` or `'bicubic'`, the computed map is upsampled to the size of the input data using the `imresize` function.

Example: `'OutputUpsampling','bicubic'`

`MiniBatchSize` — Size of mini-batch
128 (default) | positive integer

Size of the mini-batch to use to compute the map feature importance, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a positive integer.

A mini-batch is a subset of the set of synthetic images. The mini-batch size specifies the number of synthetic images that are passed to the network at once. Larger mini-batch sizes lead to faster computation, at the cost of more memory.

Example: `'MiniBatchSize',256`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'cpu'` | `'gpu'`

Hardware resource for computing map, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following.

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.

The GPU option requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose the `'ExecutionEnvironment','gpu'` option and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

Example: `'ExecutionEnvironment','gpu'`

Output Arguments

`scoreMap` — Map of feature importance
numeric matrix | numeric array

Map of feature importance, returned as a numeric matrix or a numeric array. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified classification label.

The value of `scoreMap(i,j)` denotes the importance of the image pixel `(i,j)` to the simple model, except when you use the options `'Segmentation','grid'`, and `'OutputUpsampling','none'`. In that case, the `scoreMap` is smaller than the input image, and the value of `scoreMap(i,j)` denotes the importance of the feature at position `(i,j)` in the grid of features.

If `label` is specified as a vector, the change in classification score for each class label is calculated independently. In that case, `scoreMap(:,:,k)` corresponds to the occlusion map for the `k`th element in `label`.

`featureMap` — Map of features
numeric matrix

Map of features, returned as a numeric matrix.

For each pixel `(i,j)` in the input image, `idx = featureMap(i,j)` is an integer corresponding to the index of the feature containing that pixel.

`featureImportance` — Feature importance
numeric vector | numeric matrix

Feature importance, returned as a numeric vector or a numeric matrix.

The value of `featureImportance(idx)` is the calculated importance of the feature specified by `idx`. If you provide labels as a vector of categorical values, char vectors, or string scalars, then `featureImportance(idx,k)` corresponds to the importance of feature `idx` for `label(k)`.

More About

LIME

The locally interpretable model-agnostic explanations (LIME) technique is an explainability technique used to explain the classification decisions made by a deep neural network.

Given the classification decision of deep network for a piece of input data, the LIME technique calculates the importance of each feature of the input data to the classification result.

The LIME technique approximates the behavior of a deep neural network using a simpler, more interpretable model, such as a regression tree. To map the importance of different parts of the input image, the `imageLIME` function of performs the following steps.

- Segment the image into features.
- Generate synthetic image data by randomly including or excluding features. Each pixel in an excluded feature is replaced with the value of the average image pixel.
- Classify the synthetic images using the deep network.
- Fit a regression model using the presence or absence of image features for each synthetic image as binary regression predictors for the scores of the target class.
- Compute the importance of each feature using the regression model.

The resulting map can be used to determine which features were most important to a particular classification decision. This can be especially useful for making sure your network is focusing on the appropriate features when classifying.

See Also

`activations` | `classify` | `occlusionSensitivity` | `gradCAM`

Topics
“Understand Network Predictions Using LIME”

“Investigate Spectrogram Classifications Using LIME”
“Interpret Deep Network Predictions on Tabular Data Using LIME”
“Understand Network Predictions Using Occlusion”
“Grad-CAM Reveals the Why Behind Deep Learning Decisions”
“Investigate Network Predictions Using Class Activation Mapping”

Introduced in R2020b

importCaffeLayers

Import convolutional neural network layers from Caffe

Syntax

```
layers = importCaffeLayers(protofile)
layers = importCaffeLayers(protofile,'InputSize',sz)
```

Description

`layers = importCaffeLayers(protofile)` imports the layers of a Caffe [1] network. The function returns the layers defined in the `.prototxt` file `protofile`.

This function requires Deep Learning Toolbox Importer for Caffe Models support package. If this support package is not installed, then the function provides a download link.

You can download pretrained networks from Caffe Model Zoo [2].

`layers = importCaffeLayers(protofile,'InputSize',sz)` specifies the size of the input data. If the `.prototxt` file does not specify the size of the input data, then you must specify the input size.

Examples

Download Deep Learning Toolbox Importer for Caffe Models Support Package

Download and install Deep Learning Toolbox Importer for Caffe Models support package.

Download the required support package by typing `importCaffeLayers` at the command line.

```
importCaffeLayers
```

If Deep Learning Toolbox Importer for Caffe Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

Import Layers from Caffe Network

Specify the example file `'digitsnet.prototxt'` to import.

```
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```
layers = importCaffeLayers(protofile)
```

```
layers =

  1x7 Layer array with layers:
```

```
  1   'testdata'   Image Input             28x28x1 images
  2   'conv1'      Convolution             20 5x5x1 convolutions with stride [1  1] and padding [0  0]
  3   'relu1'      ReLU                    ReLU
  4   'pool1'      Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0]
  5   'ip1'        Fully Connected         10 fully connected layer
  6   'loss'       Softmax                 softmax
  7   'output'     Classification Output   crossentropyex with 'class1', 'class2', and 8 other classes
```

Input Arguments

`protofile` — File name

character vector | string scalar

File name of the `.prototxt` file containing the network architecture, specified as a character vector or a string scalar. `protofile` must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file. If the `.prototxt` file does not specify the size of the input data, you must specify the size using the `sz` input argument.

Example: `'digitsnet.prototxt'`

`sz` — Size of input data

row vector

Size of input data, specified as a row vector. Specify a vector of two or three integer values `[h,w]`, or `[h,w,c]` corresponding to the height, width, and the number of channels of the input data.

Example: `[28 28 1]`

Output Arguments

`layers` — Network architecture

`Layer` array | `LayerGraph` object

Network architecture, returned as a `Layer` array or a `LayerGraph` object. Caffe networks that take color images as input expect the images to be in BGR format. During import, `importCaffeLayers` modifies the network so that the imported MATLAB network takes RGB images as input.

More About

Use Imported Network Layers on GPU

`importCaffeLayers` does not execute on a GPU. However, `importCaffeLayers` imports the layers of a pretrained neural network for deep learning as a `Layer` array or `LayerGraph` object, which you can use on a GPU.

- Convert the imported layers to a `DAGNetwork` object by using `assembleNetwork`. On the `DAGNetwork` object, you can then predict class labels on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function and specify the name-value argument `ReturnCategorical` as `true`.
- Convert the imported `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. On the `dlnetwork` object, you can then predict class labels on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.

- If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
- Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

  ```
  dlnet = dlupdate(@gpuarray,dlnet)
  ```

- You can train the imported layers on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- `importCaffeLayers` can import networks with the following Caffe layer types, with some limitations:

Caffe Layer	Deep Learning Toolbox Layer
`BatchNormLayer`	`batchNormalizationLayer`
`ConcatLayer`	`depthConcatenationLayer`
`ConvolutionLayer`	`convolution2dLayer`
`DeconvolutionLayer`	`transposedConv2dLayer`
`DropoutLayer`	`dropoutLayer`
`EltwiseLayer` (only sum)	`additionLayer`
`EuclideanLossLayer`	`RegressionOutputLayer`
`InnerProductLayer`	`fullyConnectedLayer`
`InputLayer`	`imageInputLayer`
`LRNLayer` (Local Response Normalization)	`crossChannelNormalizationLayer`
`PoolingLayer`	`maxPooling2dLayer` or `averagePooling2dLayer`
`ReLULayer`	`reluLayer` or `leakyReluLayer`
`ScaleLayer`	`batchNormalizationLayer`
`SigmoidLayer`	`nnet.caffe.layer.SigmoidLayer`
`SoftmaxLayer`	`softmaxLayer`
`TanHLayer`	`tanhLayer`

  If the network contains any other type of layer, then the software returns an error.

  The function imports only the layers that `protofile` specifies with the include-phase TEST. The function ignores any layers that `protofile` specifies with the include-phase TRAIN.

References

[1] *Caffe*. https://caffe.berkeleyvision.org/.

[2] *Caffe Model Zoo*. https://caffe.berkeleyvision.org/model_zoo.html.

See Also

`importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `assembleNetwork` | `exportONNXNetwork` | `importONNXLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”

Introduced in R2017a

importCaffeNetwork

Import pretrained convolutional neural network models from Caffe

Syntax

```
net = importCaffeNetwork(protofile,datafile)
net = importCaffeNetwork( ___ ,Name,Value)
```

Description

`net = importCaffeNetwork(protofile,datafile)` imports a pretrained network from Caffe [1]. The function returns the pretrained network with the architecture specified by the `.prototxt` file `protofile` and with network weights specified by the `.caffemodel` file `datafile`.

This function requires Deep Learning Toolbox Importer for Caffe Models support package. If this support package is not installed, the function provides a download link.

You can download pretrained networks from Caffe Model Zoo [2].

`net = importCaffeNetwork( ___ ,Name,Value)` returns a network with additional options specified by one or more `Name,Value` pair arguments using any of the previous syntaxes.

Examples

Download Deep Learning Toolbox Importer for Caffe Models Support Package

Download and install Deep Learning Toolbox Importer for Caffe Models support package.

To download the required support package, type `importCaffeNetwork` at the command line.

```
importCaffeNetwork
```

If Deep Learning Toolbox Importer for Caffe Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

Import Caffe Network

Specify files to import.

```
protofile = 'digitsnet.prototxt';
datafile = 'digits_iter_10000.caffemodel';
```

Import network.

```
net = importCaffeNetwork(protofile,datafile)
```

```
net =
  SeriesNetwork with properties:
```

```
    Layers: [7×1 nnet.cnn.layer.Layer]
 InputNames: {'testdata'}
OutputNames: {'ClassificationOutput'}
```

Input Arguments

`protofile` — File name

character vector | string scalar

File name of the `.prototxt` file containing the network architecture, specified as a character vector or a string scalar. `protofile` must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file. If the `.prototxt` file does not specify the size of the input data, you must specify the size using the `'InputSize'` name-value pair argument.

Example: `'digitsnet.prototxt'`

`datafile` — File name

character vector | string scalar

File name of the `.caffemodel` file containing the network weights, specified as a character vector or a string scalar. `datafile` must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file. To import network layers without weights, use `importCaffeLayers`.

Example: `'digits_iter_10000.caffemodel'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importCaffeNetwork(protofile,datafile,'AverageImage',I)` imports a pretrained network using the average image `I` for zero-center normalization.

`InputSize` — Size of input data

row vector

Size of input data, specified as a row vector. Specify a vector of two or three integer values `[h,w]`, or `[h,w,c]` corresponding to the height, width, and the number of channels of the input data. If the `.prototxt` file does not specify the size of the input data, then you must specify the input size.

Example: `[28 28 1]`

`AverageImage` — Average image

matrix

Average image for zero-center normalization, specified as a matrix. If you specify an image, then you must specify an image of the same size as the input data. If you do not specify an image, the software uses the data specified in the `.prototxt` file, if present. Otherwise, the function sets the `Normalization` property of the image input layer of the network to `'none'`.

`Classes` — Classes of the output layer

`'auto'` (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. 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)`. If `Classes` is `'auto'`, then the function sets the classes to `categorical(1:N)`, where `N` is the number of classes.

Data Types: `char` | `categorical` | `string` | `cell`

Output Arguments

net — Imported pretrained Caffe network

`SeriesNetwork` object | `DAGNetwork` object

Imported pretrained Caffe network, returned as a `SeriesNetwork` object or `DAGNetwork` object. Caffe networks that take color images as input expect the images to be in BGR format. During import, `importCaffeNetwork` modifies the network so that the imported MATLAB network takes RGB images as input.

More About

Use Imported Network on GPU

`importCaffeNetwork` does not execute on a GPU. However, `importCaffeNetwork` imports a pretrained neural network for deep learning as a `DAGNetwork` or `SeriesNetwork` object, which you can use on a GPU.

- You can make predictions with the imported network on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function.
- You can make predictions with the imported network on either a CPU or GPU by using `predict`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. If the network has multiple outputs, specify the name-value argument `ReturnCategorical` as `true`.
- You can train the imported network on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- `importCaffeNetwork` can import networks with the following Caffe layer types, with some limitations:

Caffe Layer	Deep Learning Toolbox Layer
`BatchNormLayer`	`batchNormalizationLayer`
`ConcatLayer`	`depthConcatenationLayer`
`ConvolutionLayer`	`convolution2dLayer`

Caffe Layer	Deep Learning Toolbox Layer
`DeconvolutionLayer`	`transposedConv2dLayer`
`DropoutLayer`	`dropoutLayer`
`EltwiseLayer` (only sum)	`additionLayer`
`EuclideanLossLayer`	`RegressionOutputLayer`
`InnerProductLayer`	`fullyConnectedLayer`
`InputLayer`	`imageInputLayer`
`LRNLayer` (Local Response Normalization)	`crossChannelNormalizationLayer`
`PoolingLayer`	`maxPooling2dLayer` or `averagePooling2dLayer`
`ReLULayer`	`reluLayer` or `leakyReluLayer`
`ScaleLayer`	`batchNormalizationLayer`
`SigmoidLayer`	`nnet.caffe.layer.SigmoidLayer`
`SoftmaxLayer`	`softmaxLayer`
`TanHLayer`	`tanhLayer`

If the network contains any other type of layer, then the software returns an error.

The function imports only the layers that `protofile` specifies with the include-phase TEST. The function ignores any layers that `protofile` specifies with the include-phase TRAIN.

Compatibility Considerations

`'ClassNames'` option will be removed
Not recommended starting in R2018b

`'ClassNames'` will be removed. Use `'Classes'` instead. To update your code, replace all instances of `'ClassNames'` with `'Classes'`. There are some differences between the corresponding properties in classification output layers that require additional updates to your code.

The `ClassNames` property of a classification output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

References

[1] *Caffe*. https://caffe.berkeleyvision.org/.

[2] *Caffe Model Zoo*. https://caffe.berkeleyvision.org/model_zoo.html.

Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = importCaffeNetwork`.

See Also

`importCaffeLayers` | `importKerasLayers` | `importKerasNetwork` | `assembleNetwork` | `exportONNXNetwork` | `importONNXLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”

Introduced in R2017a

importKerasLayers

Import layers from Keras network

Syntax

```
layers = importKerasLayers(modelfile)
layers = importKerasLayers(modelfile,Name,Value)
```

Description

`layers = importKerasLayers(modelfile)` imports the layers of a TensorFlow-Keras network from a model file. The function returns the layers defined in the HDF5 (`.h5`) or JSON (`.json`) file given by the file name `modelfile`.

This function requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, then the function provides a download link.

`layers = importKerasLayers(modelfile,Name,Value)` imports the layers from a TensorFlow-Keras network with additional options specified by one or more name-value pair arguments.

For example, `importKerasLayers(modelfile,'ImportWeights',true)` imports the network layers and the weights from the model file `modelfile`.

Examples

Download and Install Deep Learning Toolbox Converter for TensorFlow Models

Download and install the Deep Learning Toolbox Converter for TensorFlow Models support package.

Type `importKerasLayers` at the command line.

```
importKerasLayers
```

If the Deep Learning Toolbox Converter for TensorFlow Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by importing the layers from the model file `'digitsDAGnet.h5'` at the command line. If the required support package is installed, then the function returns a `LayerGraph` object.

```
modelfile = 'digitsDAGnet.h5';
net = importKerasLayers(modelfile)
```

```
net =
  LayerGraph with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Import Layers from Keras Network and Plot Architecture

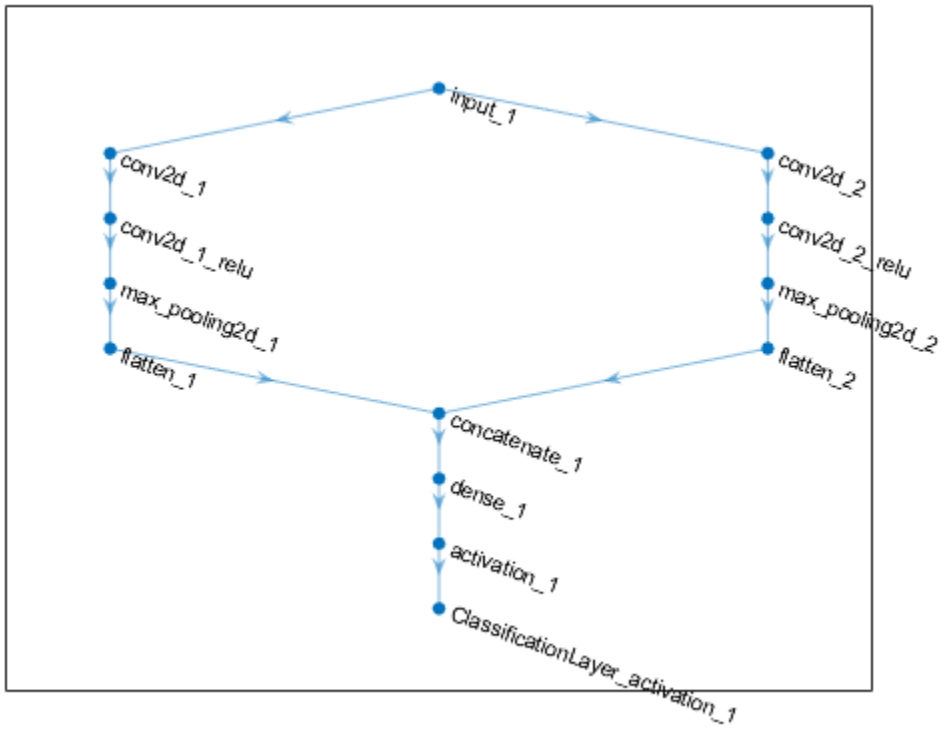
Import the network layers from the model file `digitsDAGnet.h5`.

```
modelfile = 'digitsDAGnet.h5';
layers = importKerasLayers(modelfile)
```

```
layers =
  LayerGraph with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Plot the network architecture.

```
plot(layers)
```



Import Keras Network Layers and Train Network

Specify the network file to import.

```
modelfile = 'digitsDAGnet.h5';
```

Import network layers.

```
layers = importKerasLayers(modelfile)
```

```
layers =
  LayerGraph with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Load a data set for training a classifier to recognize new digits.

```
folder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(folder, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Partition the dataset into training and test sets.

```
numTrainFiles = 750;
[imdsTrain,imdsTest] = splitEachLabel(imds,numTrainFiles,'randomize');
```

Set the training options.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',10, ...
    'InitialLearnRate',0.001);
```

Train network using training data.

```
net = trainNetwork(imdsTrain,layers,options);
```

```
Training on single CPU.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:02 |       15.62% |      12.6982 |          0.0010 |
|       1 |          50 |       00:00:14 |       63.28% |       1.2108 |          0.0010 |
|       2 |         100 |       00:00:25 |       85.16% |       0.4183 |          0.0010 |
|       3 |         150 |       00:00:37 |       96.09% |       0.1757 |          0.0010 |
|       4 |         200 |       00:00:50 |       99.22% |       0.0451 |          0.0010 |
|       5 |         250 |       00:01:05 |      100.00% |       0.0370 |          0.0010 |
|       6 |         300 |       00:01:19 |       96.88% |       0.1223 |          0.0010 |
|       7 |         350 |       00:01:31 |      100.00% |       0.0086 |          0.0010 |
|       7 |         400 |       00:01:43 |      100.00% |       0.0166 |          0.0010 |
|       8 |         450 |       00:01:56 |      100.00% |       0.0097 |          0.0010 |
|       9 |         500 |       00:02:11 |      100.00% |       0.0047 |          0.0010 |
|      10 |         550 |       00:02:24 |      100.00% |       0.0031 |          0.0010 |
|      10 |         580 |       00:02:32 |      100.00% |       0.0060 |          0.0010 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Run the trained network on the test set that was not used to train the network and predict the image labels (digits).

```
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9852
```

Import Keras Network Architecture and Weights from Same File

Specify the network file to import layers and weights from.

```
modelfile = 'digitsDAGnet.h5';
```

Import the network architecture and weights from the files you specified. To import the layer weights, specify `'ImportWeights'` to be `true`. The function also imports the layers with their weights from the same HDF5 file.

```
layers = importKerasLayers(modelfile,'ImportWeights',true)
```

```
layers =
  LayerGraph with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

View the size of the weights in the second layer.

```
weights = layers.Layers(2).Weights;
size(weights)
```

```
ans = 1×4

     7     7     1    20
```

The function has imported the weights so the layer weights are non-empty.

Import Keras Network Architecture and Weights from Separate Files

Specify the network file to import layers from and the file containing weights.

```
modelfile = 'digitsDAGnet.json';
weights = 'digitsDAGnet.weights.h5';
```

Import the network architecture and weights from the files you specified. The .json file does not include an output layer. Specify the output layer, so that importKerasLayers adds an output layer at the end of the networks architecture.

```
layers = importKerasLayers(modelfile, ...
    'ImportWeights',true, ...
    'WeightFile',weights, ...
    'OutputLayerType','classification')
```

```
layers =
  LayerGraph with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

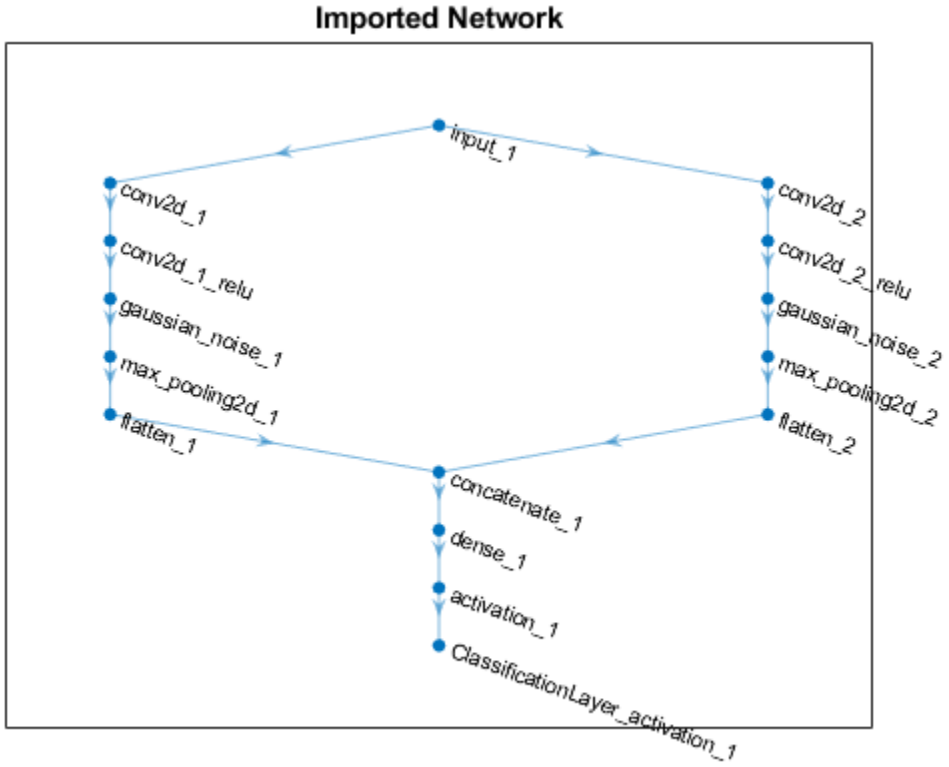
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

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



Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```
placeholderLayers.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

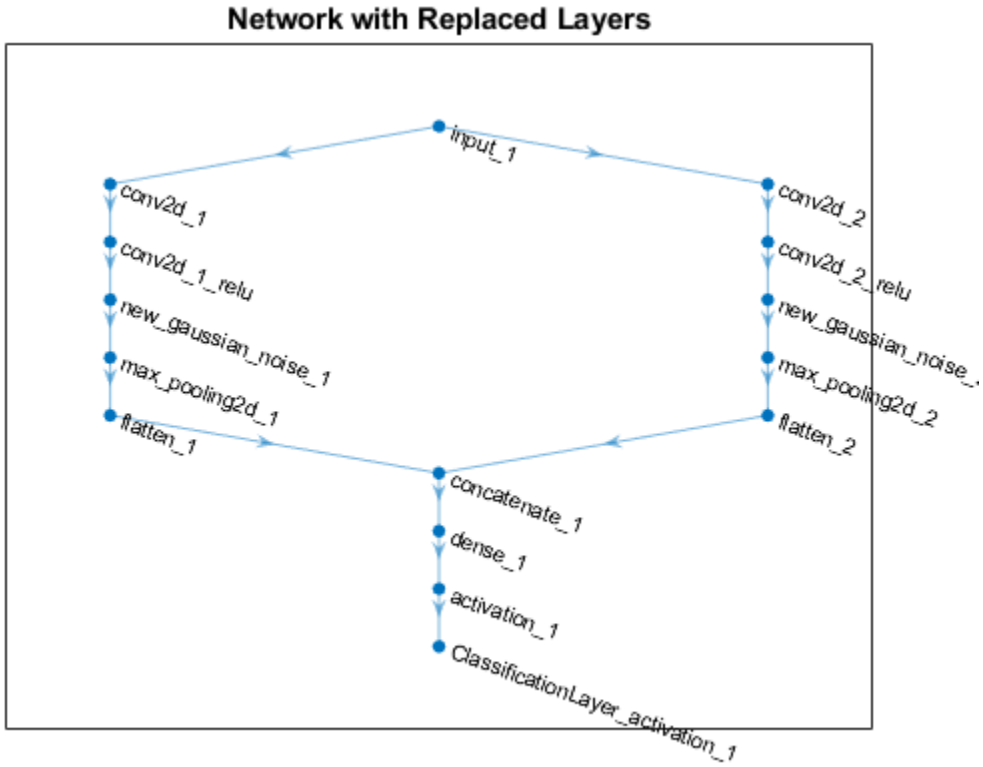
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```
figure
plot(lgraph)
title("Network with Replaced Layers")
```



Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                           Image Input             28x28x1 images
     2   'conv2d_1'                          Convolution             20 7x7x1 convolutions with
     3   'conv2d_1_relu'                     ReLU                    ReLU
     4   'conv2d_2'                          Convolution             20 3x3x1 convolutions with
     5   'conv2d_2_relu'                     ReLU                    ReLU
     6   'new_gaussian_noise_1'              Gaussian Noise          Gaussian noise with standar
     7   'new_gaussian_noise_2'              Gaussian Noise          Gaussian noise with standar
     8   'max_pooling2d_1'                   Max Pooling             2x2 max pooling with stride
     9   'max_pooling2d_2'                   Max Pooling             2x2 max pooling with stride
    10   'flatten_1'                         Keras Flatten           Flatten activations into 1-
    11   'flatten_2'                         Keras Flatten           Flatten activations into 1-
    12   'concatenate_1'                     Depth concatenation     Depth concatenation of 2 in
    13   'dense_1'                           Fully Connected         10 fully connected layer
    14   'activation_1'                      Softmax                 softmax
    15   'ClassificationLayer_activation_1'  Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_activation_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `'auto'`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: [0    1    2    3    4    5    6    7    8    9]
    ClassWeights: 'none'
      OutputSize: 10

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)

net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Import Keras PReLU Layer

Import layers from a Keras network that has parametric rectified linear unit (PReLU) layers.

A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar. The PReLU operation is given by

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ a_i x_i & \text{if } x_i \le 0 \end{cases}$$

where x_i is the input of the nonlinear activation f on channel i, and a_i is the scaling parameter controlling the slope of the negative part. The subscript i in a_i indicates that the parameter can be a vector and the nonlinear activation can vary on different channels.

`importKerasNetwork` and `importKerasLayers` can import a network that includes PReLU layers. These functions support both scalar-valued and vector-valued scaling parameters. If a scaling parameter is a vector, then the functions replace the vector with the average of the vector elements. You can modify a PReLU layer to have a vector-valued scaling parameter after import.

Specify the network file to import.

```
modelfile = 'digitsDAGnetwithPReLU.h5';
```

`digitsDAGnetwithPReLU` includes two PReLU layers. One has a scalar-valued scaling parameter, and the other has a vector-valued scaling parameter.

Import the network architecture and weights from `modelfile`.

```
layers = importKerasLayers(modelfile,'ImportWeights',true);

Warning: Layer 'p_re_lu_1' is a PReLU layer with a vector-valued parameter. The function replaces
```

The `importKerasLayers` function displays a warning for the PReLu layer `p_re_lu_1`. The function replaces the vector-valued scaling parameter of `p_re_lu_1` with the average of the vector elements. You can change the parameter back to a vector. First, find the index of the PReLU layer by viewing the `Layers` property.

```
layers.Layers

ans =
  13x1 Layer array with layers:

     1   'input_1'                     Image Input             28x28x1 images
     2   'conv2d_1'                    Convolution             20 7x7x1 convolutions with strid
```

```
     3   'conv2d_2'                      Convolution             20 3x3x1 convolutions with stric
     4   'p_re_lu_1'                     PReLU                   PReLU layer
     5   'p_re_lu_2'                     PReLU                   PReLU layer
     6   'max_pooling2d_1'               Max Pooling             2x2 max pooling with stride [2
     7   'max_pooling2d_2'               Max Pooling             2x2 max pooling with stride [2
     8   'flatten_1'                     Keras Flatten           Flatten activations into 1-D ass
     9   'flatten_2'                     Keras Flatten           Flatten activations into 1-D ass
    10   'concatenate_1'                 Depth concatenation     Depth concatenation of 2 inputs
    11   'dense_1'                       Fully Connected         10 fully connected layer
    12   'dense_1_softmax'               Softmax                 softmax
    13   'ClassificationLayer_dense_1'   Classification Output   crossentropyex
```

`layers` has two PReLU layers. Extract the fourth layer `p_re_lu_1`, which originally had a vector-valued scaling parameter for a channel dimension.

```
tempLayer = layers.Layers(4)
```

```
tempLayer =
  PreluLayer with properties:

        Name: 'p_re_lu_1'
    RawAlpha: [20x1 single]

   Learnable Parameters
       Alpha: 0.0044

   State Parameters
    No properties.

  Show all properties
```

The `RawAlpha` property contains the vector-valued scaling parameter, and the `Alpha` property contains a scalar that is an element average of the vector values. Reshape `RawAlpha` to place the vector values in the third dimension, which corresponds to the channel dimension. Then, replace `Alpha` with the reshaped `RawAlpha` values.

```
tempLayer.Alpha = reshape(tempLayer.RawAlpha,[1,1,numel(tempLayer.RawAlpha)])
```

```
tempLayer =
  PreluLayer with properties:

        Name: 'p_re_lu_1'
    RawAlpha: [20x1 single]

   Learnable Parameters
       Alpha: [1x1x20 single]

   State Parameters
    No properties.

  Show all properties
```

Replace the `p_re_lu_1` layer in `layers` with `tempLayer`.

```
layers = replaceLayer(layers,'p_re_lu_1', tempLayer);
layers.Layers(4)
```

```
ans =
  PreluLayer with properties:

       Name: 'p_re_lu_1'
    RawAlpha: [20x1 single]

   Learnable Parameters
       Alpha: [1x1x20 single]

   State Parameters
    No properties.

  Show all properties
```

Now the `p_re_lu_1` layer has a vector-valued scaling parameter.

Input Arguments

`modelfile` — Name of Keras model file

character vector | string scalar

Name of the model file containing the network architecture, and possibly the weights, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

If `modelfile` includes

- The network architecture and weights, then it must be in HDF5 (`.h5`) format.
- Only the network architecture, then it can be in HDF5 or JSON (`.json`) format.

If `modelfile` includes only the network architecture, then you can optionally supply the weights using the `'ImportWeights'` and `'WeightFile'` name-value pair arguments. If you supply the weights, then the weights file must be in HDF5 format.

Example: `'digitsnet.h5'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importKerasLayers(modelfile,'OutputLayerType','classification')` imports the network layers from the model file `modelfile` and adds an output layer for a classification problem at the end of the Keras layers.

`OutputLayerType` — Type of output layer

`'classification'` | `'regression'` | `'pixelclassification'`

Type of output layer that the function appends to the end of the imported network architecture when `modelfile` does not specify a loss function, specified as `'classification'`, `'regression'`, or `'pixelclassification'`. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox.

If a network in `modelfile` has multiple outputs, then you cannot specify the output layer types using this argument. `importKerasLayers` inserts placeholder layers for the outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively.

Example: `'OutputLayerType','regression'`

`ImageInputSize` — Size of input images
vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to `[height,width]` for grayscale images and `[height,width,channels]` for color images, respectively. The network uses this information when the `modelfile` does not specify the input size.

If a network in `modelfile` has multiple inputs, then you cannot specify the input sizes using this argument. `importKerasLayers` inserts placeholder layers for the inputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively.

Example: `'ImageInputSize',[28 28]`

`ImportWeights` — Indicator to import weights
`false` (default) | true

Indicator to import weights as well as the network architecture, specified as either `false` or `true`.

- If `'ImportWeights'` is `true` and `modelfile` includes the weights, then `importKerasLayers` imports the weights from `modelfile`, which must have HDF5 (`.h5`) format.
- If `'ImportWeights'` is `true` and `modelfile` does not include the weights, then you must specify a separate file that includes weights, using the `'WeightFile'` name-value pair argument.

Example: `'ImportWeights',true`

Data Types: `logical`

`WeightFile` — Weight file name
character vector | string scalar

Weight file name, from which to import weights when `modelfile` does not include weights, specified as a character vector or a string scalar. To use this name-value pair argument, you also must set `'ImportWeights'` to `true`.

Weight file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: `'WeightFile','weights.h5'`

Data Types: `char` | `string`

Output Arguments

`layers` — Network architecture
`Layer` array object | `LayerGraph` object

Network architecture, returned as a `Layer` array object when the Keras network is of type `Sequential`, or returned as a `LayerGraph` object when the Keras network is of type `Model`.

Limitations

- `importKerasLayers` supports TensorFlow-Keras versions as follows:
  - The function fully supports TensorFlow-Keras versions up to 2.2.4.
  - The function offers limited support for TensorFlow-Keras versions 2.2.5 to 2.4.0.

More About

Supported Keras Layers

`importKerasLayers` supports the following TensorFlow-Keras layer types for conversion into built-in MATLAB layers, with some limitations.

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`Add`	`additionLayer`
`Activation`, with activation names: • `'elu'` • `'relu'` • `'linear'` • `'softmax'` • `'sigmoid'` • `'swish'` • `'tanh'`	Layers: • `eluLayer` • `reluLayer` or `clippedReluLayer` • None • `softmaxLayer` • `sigmoidLayer` • `swishLayer` • `tanhLayer`
Advanced activations: • `ELU` • `Softmax` • `ReLU` • `LeakyReLU` • `PReLu*`	Layers: • `eluLayer` • `softmaxLayer` • `reluLayer`, `clippedReluLayer`, or `leakyReluLayer` • `leakyReluLayer` • `nnet.keras.layer.PreluLayer`
`AveragePooling1D`	`averagePooling1dLayer` with `PaddingValue` specified as `'mean'`
`AveragePooling2D`	`averagePooling2dLayer` with `PaddingValue` specified as `'mean'`
`BatchNormalization`	`batchNormalizationLayer`
`Bidirectional(LSTM(__))`	`bilstmLayer`
`Concatenate`	`depthConcatenationLayer`
`Conv1D`	`convolution1dLayer`

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`Conv2D`	`convolution2dLayer`
`Conv2DTranspose`	`transposedConv2dLayer`
`CuDNNGRU`	`gruLayer`
`CuDNNLSTM`	`lstmLayer`
`Dense`	`fullyConnectedLayer`
`DepthwiseConv2D`	`groupedConvolution2dLayer`
`Dropout`	`dropoutLayer`
`Embedding`	`wordEmbeddingLayer`
`Flatten`	`nnet.keras.layer.FlattenCStyleLayer`
`GlobalAveragePooling1D`	`globalAveragePooling1dLayer`
`GlobalAveragePooling2D`	`globalAveragePooling2dLayer`
`GlobalMaxPool1D`	`globalMaxPooling1dLayer`
`GlobalMaxPool2D`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`Input`	`imageInputLayer`, `sequenceInputLayer`, or `featureInputLayer`
`LSTM`	`lstmLayer`
`MaxPool1D`	`maxPooling1dLayer`
`MaxPool2D`	`maxPooling2dLayer`
`Multiply`	`multiplicationLayer`
`SeparableConv2D`	`groupedConvolution2dLayer` or `convolution2dLayer`
`TimeDistributed`	`sequenceFoldingLayer` before the wrapped layer, and `sequenceUnfoldingLayer` after the wrapped layer
`UpSampling2D`	`resize2dLayer`
`UpSampling3D`	`resize3dLayer`
`ZeroPadding1D`	`nnet.keras.layer.ZeroPadding1DLayer`
`ZeroPadding2D`	`nnet.keras.layer.ZeroPadding2DLayer`

* For a PReLU layer, `importKerasLayers` replaces a vector-valued scaling parameter with the average of the vector elements. You can change the parameter back to a vector after import. For an example, see “Import Keras PReLU Layer” on page 1-785.

Supported Keras Loss Functions

`importKerasLayers` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`

- `binary_crossentropy`

Use Imported Network Layers on GPU

`importKerasLayers` does not execute on a GPU. However, `importKerasLayers` imports the layers of a pretrained neural network for deep learning as a `Layer` array or `LayerGraph` object, which you can use on a GPU.

- Convert the imported layers to a `DAGNetwork` object by using `assembleNetwork`. On the `DAGNetwork` object, you can then predict class labels on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function and specify the name-value argument `ReturnCategorical` as `true`.
- Convert the imported `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. On the `dlnetwork` object, you can then predict class labels on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.
  - If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
  - Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

    ```
    dlnet = dlupdate(@gpuarray,dlnet)
    ```
- You can train the imported layers on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- If the network contains a layer that Deep Learning Toolbox Converter for TensorFlow Models does not support (see “Supported Keras Layers” on page 1-789), then `importKerasLayers` inserts a placeholder layer in place of the unsupported layer. To find the names and indices of the unsupported layers in the network, use the `findPlaceholderLayers` function. You then can replace a placeholder layer with a new layer that you define. To replace a layer, use `replaceLayer`.
- You can replace a placeholder layer with a new layer that you define.
  - If the network is a series network, then replace the layer in the array directly. For example, `layer(2) = newlayer;`.
  - If the network is a DAG network, then replace the layer using `replaceLayer`. For an example, see “Assemble Network from Pretrained Keras Layers” on page 1-781.
- You can import a Keras network with multiple inputs and multiple outputs (MIMO). Use `importKerasNetwork` if the network includes input size information for the inputs and loss information for the outputs. Otherwise, use `importKerasLayers`. The `importKerasLayers` function inserts placeholder layers for the inputs and outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`,

respectively. The workflow for importing MIMO Keras networks is the same as the workflow for importing MIMO ONNX networks. For an example, see "Import and Assemble ONNX Network with Multiple Outputs" on page 1-834. To learn about a deep learning network with multiple inputs and multiple outputs, see "Multiple-Input and Multiple-Output Networks".

- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see "Preprocess Images for Deep Learning".

Alternative Functionality

- Use `importKerasNetwork` or `importKerasLayers` to import a TensorFlow-Keras network in HDF5 or JSON format. If the TensorFlow network is in the saved model format, use `importTensorFlowNetwork` or `importTensorFlowLayers`.
- If you import a custom TensorFlow-Keras layer or if the software cannot convert a TensorFlow-Keras layer into an equivalent built-in MATLAB layer, you can use `importTensorFlowNetwork` or `importTensorFlowLayers`, which try to generate a custom layer. For example, `importTensorFlowNetwork` and `importTensorFlowLayers` generate a custom layer when you import a TensorFlow-Keras `Lambda` layer.

References

[1] *Keras: The Python Deep Learning library*. https://keras.io.

See Also

`importCaffeNetwork` | `findPlaceholderLayers` | `importKerasNetwork` | `replaceLayer` | `importCaffeLayers` | `assembleNetwork` | `exportONNXNetwork` | `importONNXLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics
"Deep Learning in MATLAB"
"Pretrained Deep Neural Networks"
"List of Deep Learning Layers"
"Define Custom Deep Learning Layers"
"Define Custom Deep Learning Layer with Learnable Parameters"
"Check Custom Layer Validity"

Introduced in R2017b

importKerasNetwork

Import pretrained Keras network and weights

Syntax

```
net = importKerasNetwork(modelfile)
net = importKerasNetwork(modelfile,Name,Value)
```

Description

`net = importKerasNetwork(modelfile)` imports a pretrained TensorFlow-Keras network and its weights from `modelfile`.

This function requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, the function provides a download link.

`net = importKerasNetwork(modelfile,Name,Value)` imports a pretrained TensorFlow-Keras network and its weights with additional options specified by one or more name-value pair arguments.

For example, `importKerasNetwork(modelfile,'WeightFile',weights)` imports the network from the model file `modelfile` and weights from the weight file `weights`. In this case, `modelfile` can be in HDF5 or JSON format, and the weight file must be in HDF5 format.

Examples

Download and Install Deep Learning Toolbox Converter for TensorFlow Models

Download and install the Deep Learning Toolbox Converter for TensorFlow Models support package.

Type `importKerasNetwork` at the command line.

```
importKerasNetwork
```

If the Deep Learning Toolbox Converter for TensorFlow Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file `'digitsDAGnet.h5'` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
modelfile = 'digitsDAGnet.h5';
net = importKerasNetwork(modelfile)
```

```
Warning: Saved Keras networks do not include classes. Classes will be set to categorical(1:N), wh

net =
  DAGNetwork with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
```

```
    OutputNames: {'ClassificationLayer_activation_1'}
```

Import and Plot Keras Network

Specify the file to import. The file `digitsDAGnet.h5` contains a directed acyclic graph convolutional neural network that classifies images of digits.

```
modelfile = 'digitsDAGnet.h5';
```

Import the network.

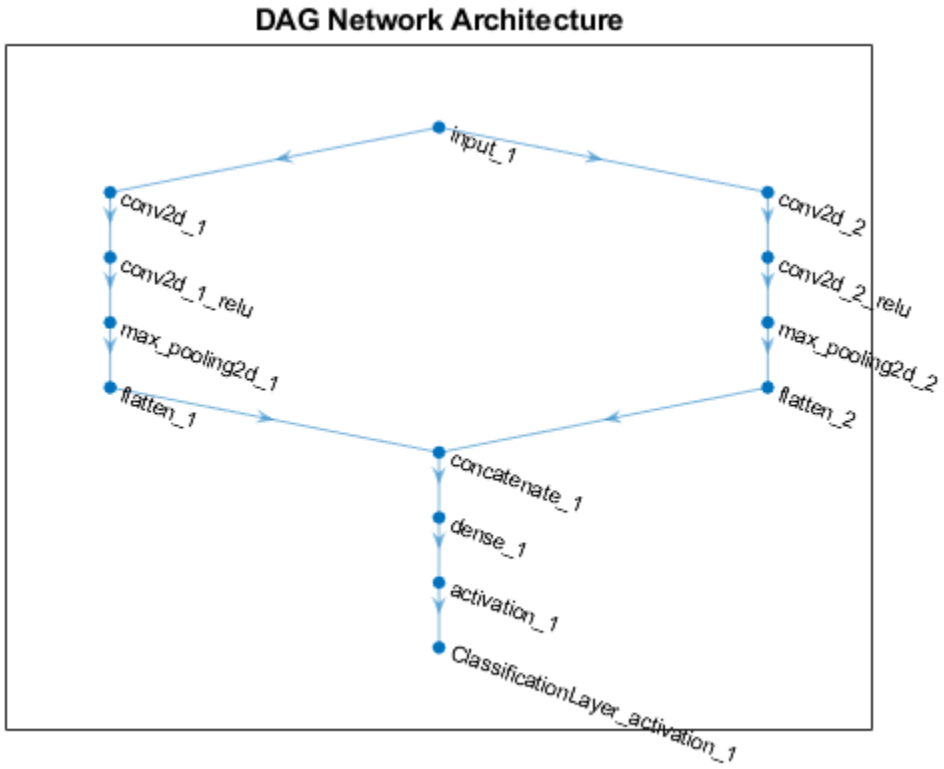
```
net = importKerasNetwork(modelfile)
```

```
Warning: Saved Keras networks do not include classes. Classes will be set to categorical(1:N), wh
```

```
net =
  DAGNetwork with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Plot the network architecture.

```
plot(net)
title('DAG Network Architecture')
```



Import Keras Network and Weights

Specify the network and the weight files to import.

```
modelfile = 'digitsDAGnet.json';
weights = 'digitsDAGnet.weights.h5';
```

This is a directed acyclic graph convolutional neural network trained on the digits data.

Import network architecture and import the weights from separate files. The .json file does not have an output layer or information on the cost function. Specify the output layer type when you import the files.

```
net = importKerasNetwork(modelfile,'WeightFile',weights, ...
      'OutputLayerType','classification')
```

```
Warning: Saved Keras networks do not include classes. Classes will be set to categorical(1:N), wh
```

```
net =
  DAGNetwork with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
     InputNames: {'input_1'}
```

```
    OutputNames: {'ClassificationLayer_activation_1'}
```

Import Pretrained Keras Network to Classify Image

Specify the model file.

```
modelfile = 'digitsDAGnet.h5';
```

Specify class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import the Keras network with the class names.

```
net = importKerasNetwork(modelfile,'Classes',classNames);
```

Read the image to classify.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
```

Classify the image using the pretrained network.

```
label = classify(net,I);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result: ' char(label)])
```

Input Arguments

`modelfile` — Name of Keras model file

character vector | string scalar

Name of the model file containing the network architecture, and possibly the weights, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

If `modelfile` includes

- The network architecture and weights, then it must be in HDF5 (`.h5`) format.

- Only the network architecture, then it can be in HDF5 or JSON (`.json`) format.

If `modelfile` includes only the network architecture, then you must supply the weights in an HDF5 file, using the `'WeightFile'` name-value pair argument.

Example: `'digitsnet.h5'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importKerasNetwork(modelfile,'OutputLayerType','classification','Classes',classes)` imports a network from the model file `modelfile`, adds an output layer for a classification problem at the end of the Keras layers, and specifies `classes` as the classes of the output layer.

`WeightFile` — Name of file containing weights

character vector | string scalar

Name of file containing weights, specified as a character vector or a string scalar. `WeightFile` must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: `'WeightFile','weights.h5'`

`OutputLayerType` — Type of output layer

`'classification'` | `'regression'` | `'pixelclassification'`

Type of output layer that the function appends to the end of the imported network architecture when `modelfile` does not specify a loss function, specified as `'classification'`, `'regression'`, or `'pixelclassification'`. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox.

If a network in `modelfile` has multiple outputs, then you cannot specify the output layer types using this argument. Use `importKerasLayers` instead. `importKerasLayers` inserts placeholder layers for the outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively.

Example: `'OutputLayerType','regression'`

`ImageInputSize` — Size of input images

vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to `[height,width]` for grayscale images and `[height,width,channels]` for color images, respectively. The network uses this information when the `modelfile` does not specify the input size.

If a network in `modelfile` has multiple inputs, then you cannot specify the input sizes using this argument. Use `importKerasLayers` instead. `importKerasLayers` inserts placeholder layers for the inputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively.

Example: `'ImageInputSize',[28 28]`

`Classes` — Classes of the output layer
`'auto'` (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. 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)`. If `Classes` is `'auto'`, then the function sets the classes to `categorical(1:N)`, where `N` is the number of classes.

Data Types: `char` | `categorical` | `string` | `cell`

Output Arguments

`net` — Pretrained Keras network
`SeriesNetwork` object | `DAGNetwork` object

Pretrained Keras network, returned as one of the following:

- If the Keras network is of type `Sequential`, then `net` is a `SeriesNetwork` object.
- If the Keras network is of type `Model`, then `net` is a `DAGNetwork` object.

Limitations

- `importKerasNetwork` supports TensorFlow-Keras versions as follows:
  - The function fully supports TensorFlow-Keras versions up to 2.2.4.
  - The function offers limited support for TensorFlow-Keras versions 2.2.5 to 2.4.0.

More About

Supported Keras Layers

`importKerasNetwork` supports the following TensorFlow-Keras layer types for conversion into built-in MATLAB layers, with some limitations.

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`Add`	`additionLayer`
`Activation`, with activation names: • `'elu'` • `'relu'` • `'linear'` • `'softmax'` • `'sigmoid'` • `'swish'` • `'tanh'`	Layers: • `eluLayer` • `reluLayer` or `clippedReluLayer` • None • `softmaxLayer` • `sigmoidLayer` • `swishLayer` • `tanhLayer`

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
Advanced activations: • `ELU` • `Softmax` • `ReLU` • `LeakyReLU` • `PReLu*`	Layers: • `eluLayer` • `softmaxLayer` • `reluLayer`, `clippedReluLayer`, or `leakyReluLayer` • `leakyReluLayer` • `nnet.keras.layer.PreluLayer`
`AveragePooling1D`	`averagePooling1dLayer` with `PaddingValue` specified as `'mean'`
`AveragePooling2D`	`averagePooling2dLayer` with `PaddingValue` specified as `'mean'`
`BatchNormalization`	`batchNormalizationLayer`
`Bidirectional(LSTM(__))`	`bilstmLayer`
`Concatenate`	`depthConcatenationLayer`
`Conv1D`	`convolution1dLayer`
`Conv2D`	`convolution2dLayer`
`Conv2DTranspose`	`transposedConv2dLayer`
`CuDNNGRU`	`gruLayer`
`CuDNNLSTM`	`lstmLayer`
`Dense`	`fullyConnectedLayer`
`DepthwiseConv2D`	`groupedConvolution2dLayer`
`Dropout`	`dropoutLayer`
`Embedding`	`wordEmbeddingLayer`
`Flatten`	`nnet.keras.layer.FlattenCStyleLayer`
`GlobalAveragePooling1D`	`globalAveragePooling1dLayer`
`GlobalAveragePooling2D`	`globalAveragePooling2dLayer`
`GlobalMaxPool1D`	`globalMaxPooling1dLayer`
`GlobalMaxPool2D`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`Input`	`imageInputLayer`, `sequenceInputLayer`, or `featureInputLayer`
`LSTM`	`lstmLayer`
`MaxPool1D`	`maxPooling1dLayer`
`MaxPool2D`	`maxPooling2dLayer`
`Multiply`	`multiplicationLayer`
`SeparableConv2D`	`groupedConvolution2dLayer` or `convolution2dLayer`

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`TimeDistributed`	`sequenceFoldingLayer` before the wrapped layer, and `sequenceUnfoldingLayer` after the wrapped layer
`UpSampling2D`	`resize2dLayer`
`UpSampling3D`	`resize3dLayer`
`ZeroPadding1D`	`nnet.keras.layer.ZeroPadding1DLayer`
`ZeroPadding2D`	`nnet.keras.layer.ZeroPadding2DLayer`

* For a PReLU layer, `importKerasNetwork` replaces a vector-valued scaling parameter with the average of the vector elements. You can change the parameter back to a vector after import. For an example, see “Import Keras PReLU Layer” on page 1-785.

Supported Keras Loss Functions

`importKerasNetwork` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`
- `binary_crossentropy`

Use Imported Network on GPU

`importKerasNetwork` does not execute on a GPU. However, `importKerasNetwork` imports a pretrained neural network for deep learning as a `DAGNetwork` or `SeriesNetwork` object, which you can use on a GPU.

- You can make predictions with the imported network on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function.
- You can make predictions with the imported network on either a CPU or GPU by using `predict`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. If the network has multiple outputs, specify the name-value argument `ReturnCategorical` as `true`.
- You can train the imported network on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- If the network contains a layer that Deep Learning Toolbox Converter for TensorFlow Models does not support (see “Supported Keras Layers” on page 1-798), then `importKerasNetwork` returns an error message. In this case, you can still use `importKerasLayers` to import the network architecture and weights.

- You can import a Keras network with multiple inputs and multiple outputs (MIMO). Use `importKerasNetwork` if the network includes input size information for the inputs and loss information for the outputs. Otherwise, use `importKerasLayers`. The `importKerasLayers` function inserts placeholder layers for the inputs and outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively. The workflow for importing MIMO Keras networks is the same as the workflow for importing MIMO ONNX networks. For an example, see "Import and Assemble ONNX Network with Multiple Outputs" on page 1-834. To learn about a deep learning network with multiple inputs and multiple outputs, see "Multiple-Input and Multiple-Output Networks".
- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see "Preprocess Images for Deep Learning".

Alternative Functionality

- Use `importKerasNetwork` or `importKerasLayers` to import a TensorFlow-Keras network in HDF5 or JSON format. If the TensorFlow network is in the saved model format, use `importTensorFlowNetwork` or `importTensorFlowLayers`.
- If you import a custom TensorFlow-Keras layer or if the software cannot convert a TensorFlow-Keras layer into an equivalent built-in MATLAB layer, you can use `importTensorFlowNetwork` or `importTensorFlowLayers`, which try to generate a custom layer. For example, `importTensorFlowNetwork` and `importTensorFlowLayers` generate a custom layer when you import a TensorFlow-Keras `Lambda` layer.

Compatibility Considerations

`'ClassNames'` option will be removed
Not recommended starting in R2018b

`'ClassNames'` will be removed. Use `'Classes'` instead. To update your code, replace all instances of `'ClassNames'` with `'Classes'`. There are some differences between the corresponding properties in classification output layers that require additional updates to your code.

The `ClassNames` property of a classification output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

References

[1] *Keras: The Python Deep Learning library*. https://keras.io.

See Also

`importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `exportONNXNetwork` | `importONNXLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics

“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Deploy Imported Network with MATLAB Compiler”

Introduced in R2017b

importONNXFunction

Import pretrained ONNX network as a function

Syntax

```
params = importONNXFunction(modelfile,NetworkFunctionName)
```

Description

`params = importONNXFunction(modelfile,NetworkFunctionName)` imports an ONNX (Open Neural Network Exchange) network from the file `modelfile` and returns an `ONNXParameters` object (`params`) that contains the network parameters. The function also creates a model function with the name specified by `NetworkFunctionName` that contains the network architecture. For more information about the network function, see “Imported ONNX Model Function” on page 1-815.

Use the `ONNXParameters` object and the `NetworkFunctionName` model function to perform common deep learning tasks, such as image and sequence data classification, transfer learning, object detection, and image segmentation. `importONNXFunction` is useful when you cannot import the network using the `importONNXNetwork` function (for example, `importONNXFunction` can import YOLOv3) or if you want to define your own custom training loop (for more details, see “Train Network Using Custom Training Loop” on page 1-593).

This function requires 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.

Examples

Import ONNX Network with Unsupported Operators as a Function

Import an ONNX network as a function. The network contains ONNX operators that are not supported by Deep Learning Toolbox layers. You can use the imported model function for deep learning tasks, such as prediction and transfer learning.

Download and install the Deep Learning Toolbox Converter for ONNX Model Format support package. You can enter `importONNXFunction` at the command line to check if the support package is installed. If it is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

Specify the file to import as `shufflenet` with operator set 9 from the ONNX Model Zoo. `shufflenet` is a convolutional neural network that is trained on images from the ImageNet database.

```
modelfile = 'shufflenet-9.onnx';
```

A recommended practice is to try to import the network by using `importONNXNetwork`. If `importONNXNetwork` is unable to import the network because some of the network layers are not supported, you can import the network as layers by using `importONNXLayers`, or as a function by using `importONNXFunction`.

Import the `shufflenet` network as layers. The software generates placeholder layers in place of the unsupported layers.

```
lgraph = importONNXLayers(modelfile,'OutputLayerType','classification');
```

```
Warning: Unable to import some ONNX operators, because they are not supported. They have been rep

4 operators(s)    :    Average pooling layer in ONNX file does not include padding in the average
32 operators(s)    :    The Reshape operator is supported only when it performs a flattening oper
16 operators(s)    :    The operator 'Transpose' is not supported.

To import the ONNX network as a function, which can support most ONNX operators, call importONNXF
```

Find the placeholder layers and display the number of placeholder layers.

```
indPlaceholderLayers = findPlaceholderLayers(lgraph);
numel(indPlaceholderLayers)
```

```
ans = 48
```

You must replace the 48 placeholder layers to use `lgraph` for deep learning tasks, such as prediction.

Instead, import the network as a function to generate a model function that you can readily use for deep learning tasks.

```
params = importONNXFunction(modelfile,'shufflenetFcn')
```

```
OpsetVersion = 9
A function 'shufflenetFcn' containing the imported ONNX network has been saved to the current dir
To learn how to use this function, type: help shufflenetFcn

params =
  ONNXParameters with properties:

             Learnables: [1×1 struct]
          Nonlearnables: [1×1 struct]
                  State: [1×1 struct]
          NumDimensions: [1×1 struct]
    NetworkFunctionName: 'shufflenetFcn'
```

`importONNXFunction` returns the `ONNXParameters` object `params`, which contains the network parameters, and the model function `shufflnetFcn`, which contains the network architecture. `importONNXFunction` saves `shufflenetFcn` in the current folder. You can open the model function to view or edit the network architecture by using `open shufflenetFcn`.

Predict Using Imported ONNX Function

Import an ONNX network as a function, and use the pretrained network to predict the class label of an input image.

Specify the file to import as `shufflenet` with operator set 9 from the ONNX Model Zoo. `shufflenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

```
modelfile = 'shufflenet-9.onnx';
```

Import the pretrained ONNX network as a function by using `importONNXFunction`, which returns the `ONNXParameters` object `params`. This object contains the network parameters. The function also creates a new model function in the current folder that contains the network architecture. Specify the name of the model function as `shufflenetFcn`.

```
params = importONNXFunction(modelfile,'shufflenetFcn');
```

```
A function containing the imported ONNX network has been saved to the file shufflenetFcn.m.
To learn how to use this function, type: help shufflenetFcn.
```

Read the image you want to classify and display the size of the image. The image is 792-by-1056 pixels and has three color channels (RGB).

```
I = imread('peacock.jpg');
size(I)
```

```
ans = 1×3

         792        1056           3
```

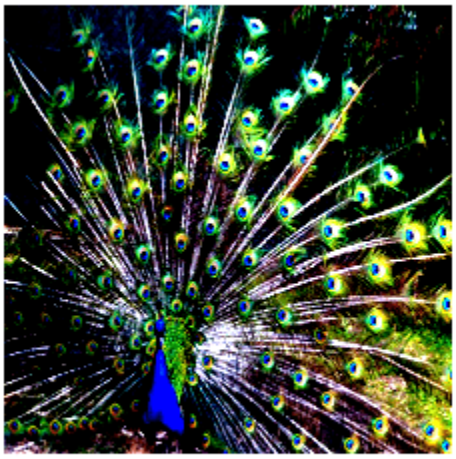
Resize the image to the input size of the network. Show the image.

```
I = imresize(I,[224 224]);
imshow(I)
```

The inputs to `shufflenet` require further preprocessing (for more details, see ShuffleNet in ONNX Model Zoo). Rescale the image. Normalize the image by subtracting the training images mean and dividing by the training images standard deviation.

```
I = rescale(I,0,1);

meanIm = [0.485 0.456 0.406];
```

```
stdIm = [0.229 0.224 0.225];
I = (I - reshape(meanIm,[1 1 3]))./reshape(stdIm,[1 1 3]);

imshow(I)
```



Import the class names from `squeezenet`, which is also trained with images from the ImageNet database.

```
net = squeezenet;
ClassNames = net.Layers(end).ClassNames;
```

Calculate the class probabilities by specifying the image to classify `I` and the `ONNXParameters` object `params` as input arguments to the model function `shufflenetFcn`.

```
scores = shufflenetFcn(I,params);
```

Find the class index with the highest probability. Display the predicted class for the input image and the corresponding classification score.

```
indMax = find(scores==max(scores));
ClassNames(indMax)
```

```
ans = 1×1 cell array
    {'peacock'}
```

```
scoreMax = scores(indMax)
```

```
scoreMax = 0.7517
```

Train Imported ONNX Function Using Custom Training Loop

Import the `squeezenet` convolution neural network as a function and fine-tune the pretrained network with transfer learning to perform classification on a new collection of images.

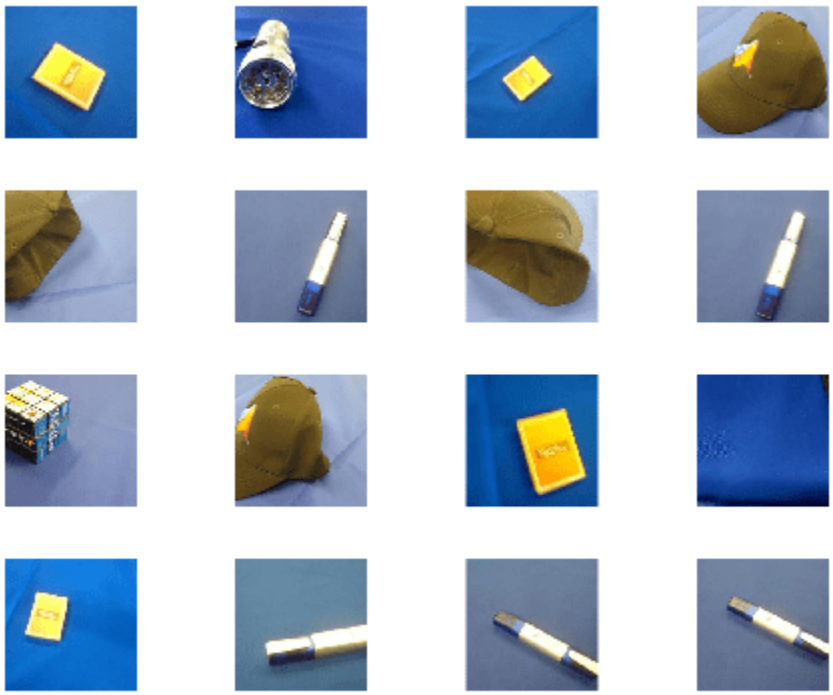
This example uses several helper functions. To view the code for these functions, see Helper Functions on page 1-0 .

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. Specify the mini-batch size.

```
unzip('MerchData.zip');
miniBatchSize = 8;
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames',...
    'ReadSize', miniBatchSize);
```

This data set is small, containing 75 training images. Display some sample images.

```
numImages = numel(imds.Labels);
idx = randperm(numImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imds,idx(i));
    imshow(I)
end
```



Extract the training set and one-hot encode the categorical classification labels.

```
XTrain = readall(imds);
XTrain = single(cat(4,XTrain{:}));
YTrain_categ = categorical(imds.Labels);
YTrain = onehotencode(YTrain_categ,2)';
```

Determine the number of classes in the data.

```
classes = categories(YTrain_categ);
numClasses = numel(classes)
```

```
numClasses = 5
```

`squeezenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

Import the pretrained `squeezenet` network as a function.

```
squeezenetONNX()
params = importONNXFunction('squeezenet.onnx','squeezenetFcn')
```

```
A function containing the imported ONNX network has been saved to the file squeezenetFcn.m.
To learn how to use this function, type: help squeezenetFcn.

params =
  ONNXParameters with properties:
```

```
            Learnables: [1×1 struct]
       Nonlearnables: [1×1 struct]
               State: [1×1 struct]
       NumDimensions: [1×1 struct]
NetworkFunctionName: 'squeezenetFcn'
```

`params` is an `ONNXParameters` object that contains the network parameters. `squeezenetFcn` is a model function that contains the network architecture. `importONNXFunction` saves `squeezenetFcn` in the current folder.

Calculate the classification accuracy of the pretrained network on the new training set.

```
accuracyBeforeTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy before transfer learning\n',accuracyBeforeTraining);
```

```
0.01 accuracy before transfer learning
```

The accuracy is very low.

Display the learnable parameters of the network by typing `params.Learnables`. These parameters, such as the weights (`W`) and bias (`B`) of convolution and fully connected layers, are updated by the network during training. Nonlearnable parameters remain constant during training.

The last two learnable parameters of the pretrained network are configured for 1000 classes.

```
conv10_W: [1×1×512×1000 dlarray]
```

```
conv10_B: [1000×1 dlarray]
```

The parameters `conv10_W` and `conv10_B` must be fine-tuned for the new classification problem. Transfer the parameters to classify five classes by initializing the parameters.

```
params.Learnables.conv10_W = rand(1,1,512,5);
params.Learnables.conv10_B = rand(5,1);
```

Freeze all the parameters of the network to convert them to nonlearnable parameters. Because you do not need to compute the gradients of the frozen layers, freezing the weights of many initial layers can significantly speed up network training.

```
params = freezeParameters(params,'all');
```

Unfreeze the last two parameters of the network to convert them to learnable parameters.

```
params = unfreezeParameters(params,'conv10_W');
params = unfreezeParameters(params,'conv10_B');
```

Now the network is ready for training. Initialize the training progress plot.

```
plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end
```

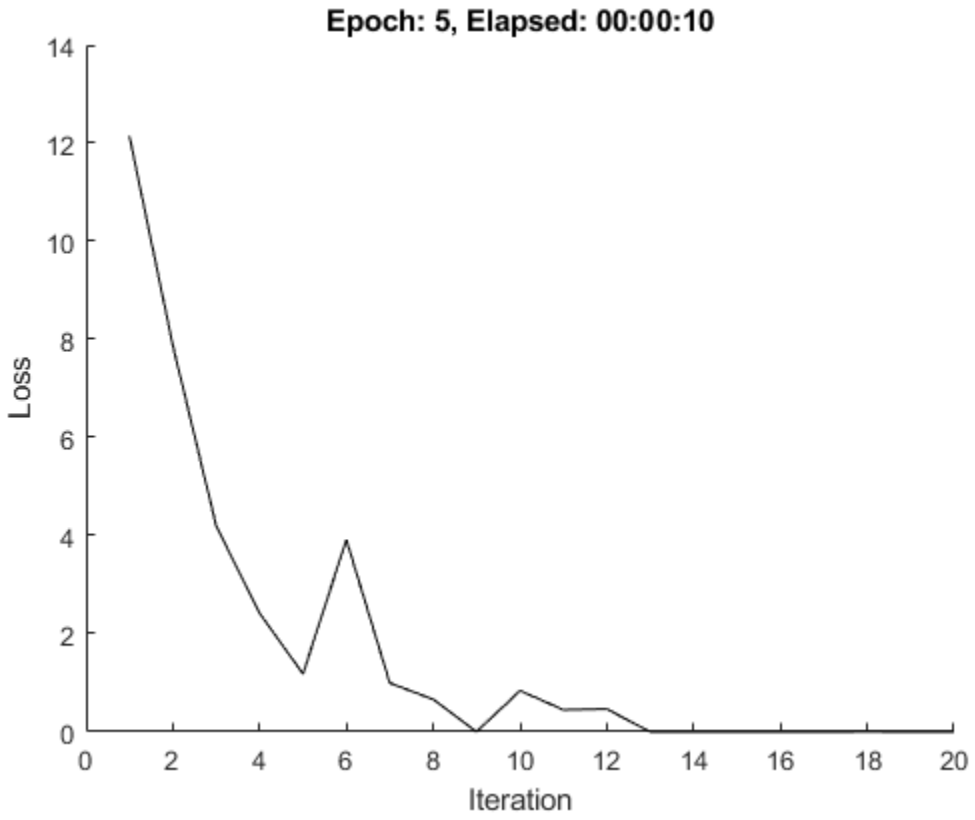
Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Train the network.

```
iteration = 0;
start = tic;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle 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.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);
        Y = YTrain(:,idx);

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

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
        params.State = state;

        % Determine the learning rate for the time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);

        % 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
```



Calculate the classification accuracy of the network after fine-tuning.

```
accuracyAfterTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy after transfer learning\n',accuracyAfterTraining);
```

```
1.00 accuracy after transfer learning
```

Helper Functions

This section provides the code of the helper functions used in this example.

The `getNetworkAccuracy` function evaluates the network performance by calculating the classification accuracy.

```
function accuracy = getNetworkAccuracy(X,Y,onnxParams)

N = size(X,4);
Ypred = squeezenetFcn(X,onnxParams,'Training',false);

[~,YIdx] = max(Y,[],1);
[~,YpredIdx] = max(Ypred,[],1);
numIncorrect = sum(abs(YIdx-YpredIdx) > 0);
accuracy = 1 - numIncorrect/N;

end
```

The `modelGradients` function calculates the loss and gradients.

```
function [grad, loss, state] = modelGradients(X,Y,onnxParams)

[y,state] = squeezenetFcn(X,onnxParams,'Training',true);
loss = crossentropy(y,Y,'DataFormat','CB');
grad = dlgradient(loss,onnxParams.Learnables);

end
```

The `squeezenetONNX` function generates an ONNX model of the `squeezenet` network.

```
function squeezenetONNX()

exportONNXNetwork(squeezenet,'squeezenet.onnx');

end
```

Sequence Classification Using Imported ONNX Function

Import an ONNX long short-term memory (LSTM) network as a function, and use the pretrained network to classify sequence data. 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 helper function `preparePermutationVector`. To view the code for this function, see Helper Function on page 1-0 .

`lstmNet` has a similar architecture to the LSTM network created in “Sequence Classification Using Deep Learning”. `lstmNet` is trained 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.

Specify `lstmNet` as the model file.

```
modelfile = 'lstmNet.onnx';
```

Import the pretrained ONNX network as a function by using `importONNXFunction`, which returns the `ONNXParameters` object `params` containing the network parameters. The function also creates a new model function in the current folder that contains the network architecture. Specify the name of the model function as `lstmnetFcn`.

```
params = importONNXFunction(modelfile,'lstmnetFcn');
```

```
A function containing the imported ONNX network has been saved to the file lstmnetFcn.m.
To learn how to use this function, type: help lstmnetFcn.
```

Load the Japanese Vowels test data. `XTest` is a cell array containing 370 sequences of dimension 12 and varying length. `YTest` is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
```

`lstmNet` was trained using mini-batches with sequences of similar length. To organize the test data 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);
```

Use `preparePermutationVector` to compute the permutation vector `inputPerm`, which permutes the dimension ordering of the input sequence data to the dimension ordering of the imported LSTM network input. You can type `help lstmnetFcn` to view the dimension ordering of the network input `SEQUENCEINPUT`.

```
inputPerm = preparePermutationVector(["FeaturesLength","SequenceLength","BatchSize"],...
    ["SequenceLength","BatchSize","FeaturesLength"]);
```

Calculate the class probabilities by specifying the sequence data to classify `XTest` and the `ONNXParameters` object `params` as input arguments to the model function `lstmnetFcn`. Customize the input dimension ordering by assigning the numeric vector `inputPerm` to the name-value argument `'InputDataPermutation'`. Return `scores` in the dimension ordering of the network output by assigning `'none'` to the name-value argument `'OutputDataPermutation'`.

```
for i = 1:length(XTest)
    scores = lstmnetFcn(XTest{i},params,'InputDataPermutation',inputPerm,'OutputDataPermutation'
    YPred(i) = find(scores==max(scores));
end
YPred = categorical(YPred');
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9514
```

Helper Function

This section provides the code of the helper function `preparePermutationVector` used in this example.

The `preparePermutationVector` function returns a permutation vector `perm`, which permutes the dimension ordering in `fromDimOrder` to the dimension ordering in `toDimOrder`. You can specify the input arguments `fromDimOrder` and `toDimOrder` as character vectors, string scalars, string arrays, cell arrays of character vectors, or numeric vectors. Both arguments must have the same type and the same unique elements. For example, if `fromDimOrder` is the character vector `'hwcn'`, `toDimOrder` can be the character vector `'nchw'` (where `h`, `w`, and `c` correspond to the height, width, and number of channels of the image, respectively, and `n` is the number of observations).

```
function perm = preparePermutationVector(fromDimOrder, toDimOrder)

% Check if both fromDimOrder and toDimOrder are vectors.
if ~isvector(fromDimOrder) || ~isvector(toDimOrder)
    error(message('nnet_cnn_onnx:onnx:FPVtypes'));
end

% Convert fromDimOrder and toDimOrder to the appropriate type.
if isstring(fromDimOrder) && isscalar(fromDimOrder)
```

```
    fromDimOrder = char(fromDimOrder);
end
if isstring(toDimOrder) && isscalar(toDimOrder)
    toDimOrder = char(toDimOrder);
end

% Check if fromDimOrder and toDimOrder have unique elements.
[fromSorted, ifrom] = unique(fromDimOrder);
[toSorted, ~, iToInv] = unique(toDimOrder);

if numel(fromSorted) ~= numel(fromDimOrder)
    error(message('nnet_cnn_onnx:onnx:FPVfromunique'));
end
if numel(toSorted) ~= numel(toDimOrder)
    error(message('nnet_cnn_onnx:onnx:FPVtounique'));
end

% Check if fromDimOrder and toDimOrder have the same number of elements.
if ~isequal(fromSorted, toSorted)
    error(message('nnet_cnn_onnx:onnx:FPVsame'));
end

% Compute the permutation vector.
perm = ifrom(iToInv);
perm = perm(:)';

end
```

Input Arguments

`modelfile` — Name of ONNX model file

character vector | string scalar

Name of the ONNX model file containing the network, specified as a character vector or string scalar. The file must be in the current folder or a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: `'shufflenet.onnx'`

`NetworkFunctionName` — Name of model function

character vector | string scalar

Name of the model function, specified as a character vector or string scalar. The function `NetworkFunctionName` contains the architecture of the imported ONNX network. The file is saved in an M-file in the current folder, or you must include a full or relative path to the file. The `NetworkFunctionName` file is required for using the network. For more information, see “Imported ONNX Model Function” on page 1-815.

Example: `'shufflenetFcn'`

Output Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, returned as an `ONNXParameters` object. `params` contains the network parameters of the imported ONNX model. Use dot notation to reference properties of `params`. For

example, `params.Learnables` displays the network learnable parameters, such as the weights of the convolution layers.

Limitations

- `importONNXFunction` supports these ONNX versions:
  - ONNX intermediate representation version 6
  - ONNX operator sets 7 to 13

More About

Imported ONNX Model Function

`importONNXFunction` creates a model function that contains the network architecture of the imported ONNX model. Specify the name `NetworkFunctionName` as an input argument to `importONNXFunction`.

Syntax

Use the following syntaxes to interface with the imported ONNX model function (`NetworkFunctionName`):

- `[Y,state] = NetworkFunctionName(X,params)` returns the output data `Y` and the updated network `state` for the input data `X`.
- `[Y,state] = NetworkFunctionName(X,params,Name,Value)` uses additional options specified by one or more name-value pair arguments.
- `[Y1,Y2,...,Yn,state] = NetworkFunctionName(X1,X2,...,Xn,params)` returns multiple output data (`Y1,Y2,...,Yn`) and the updated network `state` for the multiple input data (`X1,X2,...,Xn`).
- `[Y1,Y2,...,Yn,state] = NetworkFunctionName(X1,X2,...,Xn,params,Name,Value)` uses additional options specified by one or more name-value pair arguments for multiple inputs and outputs.

Input Arguments

Argument	Description
`X`	Input data, specified as an array or `dlarray`.
`params`	Network parameters, specified as an `ONNXParameters` object.

Name-Value Pair Arguments

Argument name	Description
`'Training'`	Training option, specified as `'false'` (default) or `'true'`. • Set value to `'false'` to use `ONNXFunction` to predict. For an example, see "Predict Using Imported ONNX Function" on page 1-804. • Set value to `'true'` to use `ONNXFunction` to train. For an example, see "Train Imported ONNX Function Using Custom Training Loop" on page 1-806.
`'InputDataPermutation'`	Permutation applied to the dimension ordering of input `X`, specified as `'auto'` (default), `'none'`, a numeric vector, or a cell array. Assign a value to the name-value pair argument `'InputDataPermutation'` to permute the input data into the dimension ordering required by the imported ONNX model. • Assign the value `'auto'` to apply an automatic permutation based on assumptions about common input data `X`. For more details, see "Automatic Input Data Permutation" on page 1-817. • Assign the value `'none'` to pass `X` in the original ordering. • Assign a numeric vector value to customize the input dimension ordering; for example, `[4 3 1 2]`. For an example, see "Sequence Classification Using Imported ONNX Function" on page 1-812. • Assign a cell array value for multiple inputs; for example, `{[3 2 1],'none'}`.

Argument name	Description
`'OutputDataPermutation'`	Permutation applied to the dimension ordering of output `Y`, specified as `'auto'` (default), `'none'`, a numeric vector, or a cell array. Assign a value to the name-value pair argument `'OutputDataPermutation'` to match the dimension ordering of the imported ONNX model. • Assign the value `'auto'` to return `Y` in Deep Learning Toolbox ordering. For more details, see “Automatic Output Data Permutation” on page 1-818. • Assign the value `'none'` to return `Y` in ONNX ordering. For an example, see “Sequence Classification Using Imported ONNX Function” on page 1-812. • Assign a numeric vector value to customize the output dimension ordering; for example, `[3 4 2 1]`. • Assign a cell array value for multiple outputs; for example, `{[3 2 1],'none'}`.

Output Arguments

Argument	Description
`Y`	Output data, returned as an array or `dlarray`. • If `X` is an array or you use `ONNXFunction` to predict, `Y` is a array. • If `X` is a `dlarray` or you use `ONNXFunction` for training, `Y` is a `dlarray`.
`state`	Updated network state, specified as a structure. The network `state` contains information remembered by the network between iterations and updated across multiple training batches.

The interpretation of input argument `X` and output argument `Y` can differ between models. For more information about the model input and output arguments, refer to `help` for the imported model function `NetworkFunctionName`, or refer to the ONNX documentation [1].

Automatic Permutation for Imported Model Function

By default, `NetworkFunctionName` automatically permutes input and output data to facilitate image classification tasks. Automatic permutation might be unsuitable for other tasks, such as object detection and time series classification.

Automatic Input Data Permutation

To automatically permute the input, `NetworkFunctionName` assumes the following based on the input dimensions specified by the imported ONNX network.

Number of ONNX Model Input Dimensions	Interpretation of Input Data	ONNX Standard Dimension Ordering	Deep Learning Toolbox Standard Dimension Ordering	Automatic Permutation of Input
4	2-D image	`NCHW` `H`, `W`, and `C` correspond to the height, width, and number of channels of the image, respectively, and `N` is the number of observations.	`HWCN` `H`, `W`, and `C` correspond to the height, width, and number of channels of the image, respectively, and `N` is the number of observations.	[4 3 1 2]

If the size of the input dimensions is a number other than 4, `NetworkFunctionName` specifies the input argument `'InputDataPermutation'` as `'none'`.

Automatic Output Data Permutation

To automatically permute the output, `NetworkFunctionName` assumes the following based on the output dimensions specified by the imported ONNX network.

Number of ONNX Model Output Dimensions	Interpretation of Output Data	ONNX Standard Dimension Ordering	Deep Learning Toolbox Standard Dimension Ordering	Automatic Permutation of Output
2	2-D image classification scores	`NK` `K` is the number of classes and `N` is the number of observations.	`KN` `K` is the number of classes and `N` is the number of observations.	`[ 2 1 ]`
4	2-D image pixel classification scores	`NKHW` `H` and `W` correspond to the height and width of the image, respectively, `K` is the number of classes, and `N` is the number of observations.	`HWKN` `H` and `W` correspond to the height and width of the image, respectively, `K` is the number of classes, and `N` is the number of observations.	`[3 4 2 1]`

If the size of the output dimensions is a number other than 2 or 4, `NetworkFunctionName` specifies the input argument `'OutputDataPermutation'` as `'none'`.

ONNX Operators That `importONNXFunction` Supports

`importONNXFunction` supports the following ONNX operators, with some limitations. Compare these operators with the operators supported by `importONNXNetwork` and `importONNXLayers` for conversion into equivalent built-in MATLAB layers.

ONNX Operators Supported by `importONNXFunction`	`importONNXNetwork` and `importONNXLayers` Support
`Abs`	No
`Add`	Yes
`And`	No
`ArgMax`	No
`AveragePool`	Yes
`BatchNormalization`	Yes
`Cast`	No
`Ceil`	No
`Clip`	Yes
`Compress`	No
`Concat`	Yes
`Constant`	Yes
`ConstantOfShape`	No
`Conv`	Yes
`ConvTranspose`	Yes
`DepthToSpace`	Yes
`Div`	Yes
`Dropout`	Yes
`Equal`	No
`Exp`	No
`Expand`	No
`Flatten`	Yes
`Floor`	No
`Gather`	No
`Gemm`	Yes
`GlobalAveragePool`	Yes
`Greater`	Yes
`Hardmax`	No
`Identity`	Yes
`If`	No
`InstanceNormalization`	Yes
`LeakyRelu`	Yes

ONNX Operators Supported by importONNXFunction	importONNXNetwork and importONNXLayers Support
Less	No
LessOrEqual	No
Log	No
Loop	No
LRN	Yes
LSTM	Yes
MatMul	Yes
MaxPool	Yes
Mul	Yes
Neg	No
NonMaxSuppression	No
NonZero	No
Not	No
OneHot	No
Or	No
Pad	No
Pow	No
PRelu	Yes
RandomUniform	No
Range	No
Reciprocal	No
ReduceMax	No
ReduceMean	No
ReduceMin	No
ReduceProd	No
ReduceSum	No
Relu	Yes
Reshape	Yes
Resize	Yes
RoiAlign	No
Round	No
Scan	No
Scatter	No
ScatterElements	No
SequenceAt	No
Shape	No

ONNX Operators Supported by `importONNXFunction`	`importONNXNetwork` and `importONNXLayers` Support
`Sigmoid`	Yes
`Slice`	No
`Softmax`	Yes
`SpaceToDepth`	Yes
`Split`	No
`SplitToSequence`	No
`Sqrt`	No
`Squeeze`	No
`Sub`	Yes
`Sum`	Yes
`Tanh`	Yes
`Tile`	No
`TopK`	No
`Transpose`	No
`Unsqueeze`	No
`Upsample`	No
`Where`	No

Tips

- Refer to the ONNX documentation for each model to see the required preprocessing of the network inputs. For example, you need to resize (using `imresize`), `rescale`, and normalize the input images to networks trained with the ImageNet dataset (such as AlexNet, GoogleNet, ShuffleNet, and SqueezeNet).

Alternative Functionality

`importONNXFunction` is useful when you cannot import a pretrained ONNX network by using `importONNXNetwork`. If you want to generate code for a pretrained network, use `importONNXLayers`. Find and replace the generated placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively. Then, use `assembleNetwork` to return a `DAGNetwork` object. You can generate code for a trained `DAGNetwork`. For more information on the import functions that best suit different scenarios, see “Select Function to Import ONNX Pretrained Network”.

References

[1] *Open Neural Network Exchange*. https://github.com/onnx/.

[2] *ONNX*. https://onnx.ai/.

See Also

`importONNXNetwork` | `importONNXLayers` | `ONNXParameters`

Topics

“Make Predictions Using Model Function”
“Train Network Using Custom Training Loop”
“Pretrained Deep Neural Networks”
“Select Function to Import ONNX Pretrained Network”

Introduced in R2020b

importONNXLayers

Import layers from ONNX network

Syntax

```
lgraph = importONNXLayers(modelfile)
lgraph = importONNXLayers(modelfile,Name=Value)
```

Description

`lgraph = importONNXLayers(modelfile)` imports the layers and weights of a pretrained ONNX (Open Neural Network Exchange) network from the file `modelfile`. The function returns `lgraph` as a `LayerGraph` object compatible with a `DAGNetwork` or `dlnetwork` object.

`importONNXLayers` requires the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then `importONNXLayers` provides a download link.

Note By default, `importONNXLayers` tries to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. For a list of operators for which the software supports conversion, see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers” on page 1-839.

`importONNXLayers` saves the generated custom layers in the package `+modelfile`.

`importONNXLayers` does not automatically generate a custom layer for each ONNX operator that is not supported for conversion into a built-in MATLAB layer. For more information on how to handle unsupported layers, see “Tips” on page 1-844.

`lgraph = importONNXLayers(modelfile,Name=Value)` imports the layers and weights from an ONNX network with additional options specified by one or more name-value arguments. For example, `OutputLayerType="classification"` imports a layer graph compatible with a `DAGNetwork` object, with a classification output layer appended to the end of the first output branch of the imported network architecture.

Examples

Download and Install Deep Learning Toolbox Converter for ONNX Model Format

Download and install the Deep Learning Toolbox Converter for ONNX Model Format support package.

Type `importONNXLayers` at the command line.

```
importONNXLayers
```

If Deep Learning Toolbox Converter for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support

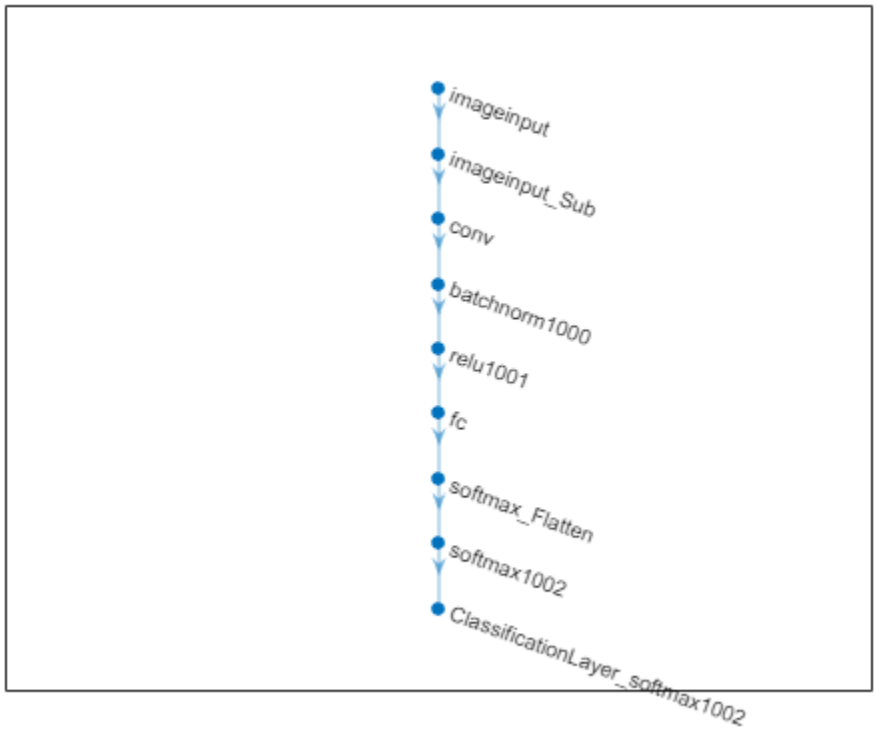
package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file `"simplenet.onnx"` at the command line. If the support package is installed, then the function returns a `LayerGraph` object.

```
modelfile = "simplenet.onnx";
lgraph = importONNXLayers(modelfile)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [9×1 nnet.cnn.layer.Layer]
    Connections: [8×2 table]
     InputNames: {'imageinput'}
    OutputNames: {'ClassificationLayer_softmax1002'}
```

Plot the network architecture.

```
plot(lgraph)
```



Import ONNX Model as Layer Graph Compatible with `DAGNetwork`

Import a pretrained ONNX network as a `LayerGraph` object. Then, assemble the imported layers into a `DAGNetwork` object, and use the assembled network to classify an image.

Generate an ONNX model of the `squeezenet` convolution neural network.

```
squeezeNet = squeezenet;
exportONNXNetwork(squeezeNet,"squeezeNet.onnx");
```

Specify the model file and the class names.

```
modelfile = "squeezenet.onnx";
ClassNames = squeezeNet.Layers(end).Classes;
```

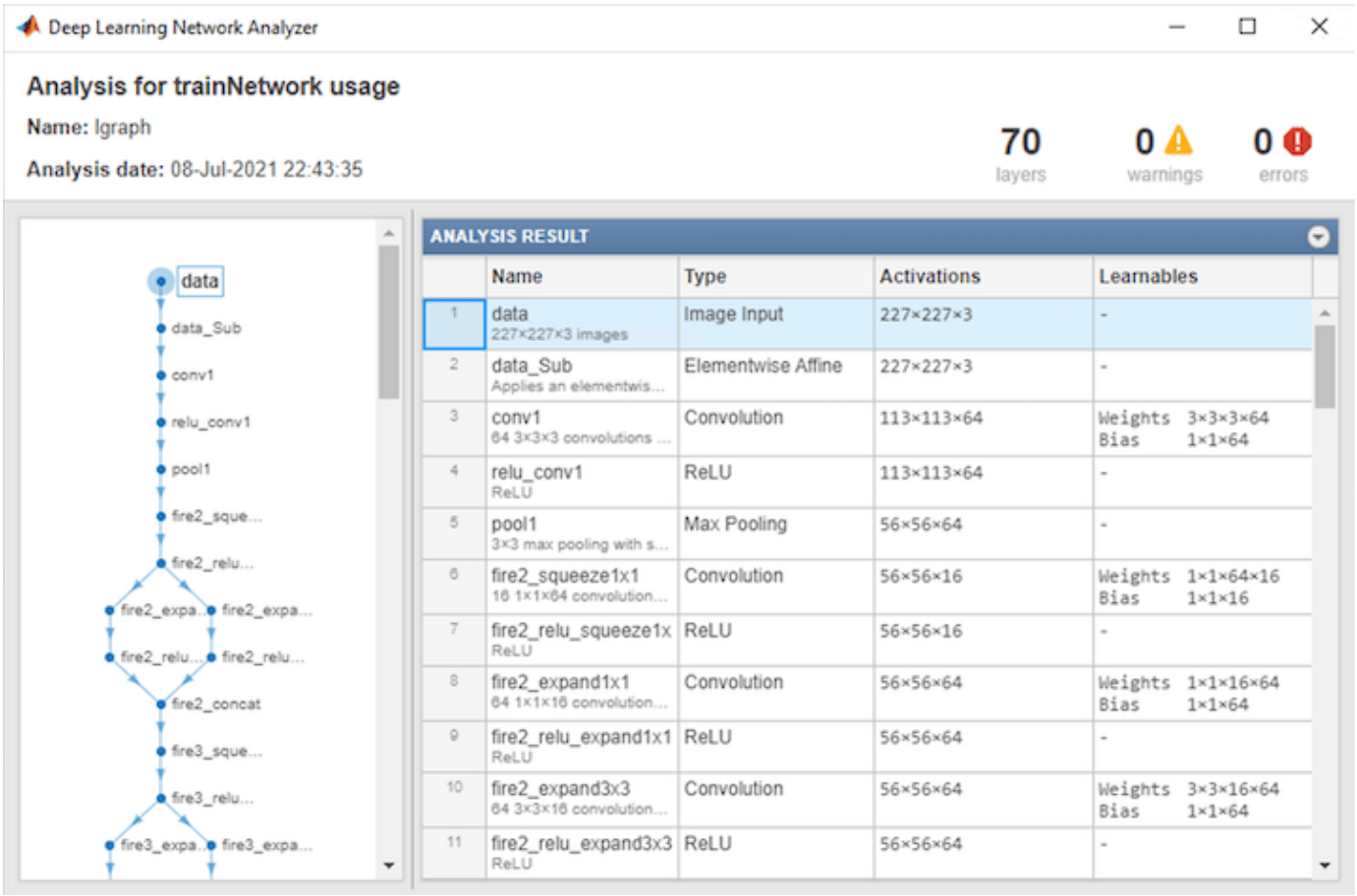
Import the layers and weights of the ONNX network. By default, `importONNXLayers` imports the network as a `LayerGraph` object compatible with a `DAGNetwork` object.

```
lgraph = importONNXLayers(modelfile)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [70×1 nnet.cnn.layer.Layer]
    Connections: [77×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_prob'}
```

Analyze the imported network architecture.

```
analyzeNetwork(lgraph)
```



Display the last layer of the imported network. The output shows that the layer graph has a `ClassificationOutputLayer` at the end of the network architecture.

```
lgraph.Layers(end)
```

```
ans =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_prob'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The classification layer does not contain the classes, so you must specify these before assembling the network. 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.

The classification layer has the name `'ClassificationLayer_prob'`. Set the classes to `ClassNames`, and then replace the imported classification layer with the new one.

```
cLayer = lgraph.Layers(end);
cLayer.Classes = ClassNames;
lgraph = replaceLayer(lgraph,'ClassificationLayer_prob',cLayer);
```

Assemble the layer graph using `assembleNetwork` to return a `DAGNetwork` object.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [70×1 nnet.cnn.layer.Layer]
    Connections: [77×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_prob'}
```

Read the image you want to classify and display the size of the image. The image is 384-by-512 pixels and has three color channels (RGB).

```
I = imread("peppers.png");
size(I)
```

```
ans = 1×3

   384   512     3
```

Resize the image to the input size of the network. Show the image.

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

Classify the image using the imported network.

```
label = classify(net,I)
```

```
label = categorical
     bell pepper
```

Import ONNX Model as Layer Graph Compatible with `dlnetwork`

Import a pretrained ONNX network as a `LayerGraph` object compatible with a `dlnetwork` object. Then, convert the layer graph to a `dlnetwork` to classify an image.

Generate an ONNX model of the `squeezenet` convolution neural network.

```
squeezeNet = squeezenet;
exportONNXNetwork(squeezeNet,"squeezeNet.onnx");
```

Specify the model file and the class names.

```
modelfile = "squeezenet.onnx";
ClassNames = squeezeNet.Layers(end).Classes;
```

Import the layers and weights of the ONNX network. Specify to import the network as a `LayerGraph` object compatible with a `dlnetwork` object.

```
lgraph = importONNXLayers(modelfile,TargetNetwork="dlnetwork")
```

```
lgraph =
  LayerGraph with properties:

         Layers: [70×1 nnet.cnn.layer.Layer]
```

```
    Connections: [77×2 table]
     InputNames: {'data'}
    OutputNames: {1×0 cell}
```

Read the image you want to classify and display the size of the image. The image is 384-by-512 pixels and has three color channels (RGB).

```
I = imread("peppers.png");
size(I)
```

```
ans = 1×3

   384   512     3
```

Resize the image to the input size of the network. Show the image.

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

Convert the imported layer graph to a `dlnetwork` object.

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

Convert the image to a `dlarray`. Format the images with the dimensions `"SSCB"` (spatial, spatial, channel, batch). In this case, the batch size is 1 and you can omit it (`"SSC"`).

```
I_dlarray = dlarray(single(I),"SSCB");
```

Classify the sample image and find the predicted label.

```
prob = predict(dlnet,I_dlarray);
[~,label] = max(prob);
```

Display the classification result.

```
ClassNames(label)

ans = categorical
     bell pepper
```

Import ONNX Model as Layer Graph with Autogenerated Custom Layers

Import a pretrained ONNX network as a `LayerGraph` object, and assemble the imported layers into a `DAGNetwork` object. Then, use the `DAGNetwork` to classify an image. The imported network contains ONNX operators that are not supported for conversion into built-in MATLAB layers. The software automatically generates custom layers when you import these operators.

This example uses the helper function `findCustomLayers`. To view the code for this function, see Helper Function on page 1-0 .

Specify the file to import as `shufflenet` with operator set 9 from the ONNX Model Zoo. `shufflenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

```
modelfile = "shufflenet-9.onnx";
```

Import the layers and weights of `shufflenet`. By default, `importONNXLayers` imports the network as a `LayerGraph` object compatible with a `DAGNetwork` object. If the imported network contains ONNX operators not supported for conversion into built-in MATLAB layers, then `importONNXLayers` can automatically generate custom layers in place of these layers. `importONNXLayers` saves each generated custom layer to a separate `.m` file in the package `+shufflenet_9` in the current folder. Specify the package name by using the name-value argument `PackageName`.

```
lgraph = importONNXLayers(modelfile,PackageName="shufflenet_9")

lgraph =
  LayerGraph with properties:

         Layers: [173×1 nnet.cnn.layer.Layer]
    Connections: [188×2 table]
     InputNames: {'gpu_0_data_0'}
    OutputNames: {'ClassificationLayer_gpu_0_softmax_1'}
```

Find the indices of the automatically generated custom layers by using the helper function `findCustomLayers`, and display the custom layers.

```
ind = findCustomLayers(lgraph.Layers,'+shufflenet_9');
lgraph.Layers(ind)

ans =
  16×1 Layer array with layers:

     1   'Reshape_To_ReshapeLayer1004'   shufflenet_9.Reshape_To_ReshapeLayer1004   shufflenet_9
     2   'Reshape_To_ReshapeLayer1009'   shufflenet_9.Reshape_To_ReshapeLayer1009   shufflenet_9
     3   'Reshape_To_ReshapeLayer1014'   shufflenet_9.Reshape_To_ReshapeLayer1014   shufflenet_9
```

```
     4   'Reshape_To_ReshapeLayer1019'   shufflenet_9.Reshape_To_ReshapeLayer1019   shufflenet_9
     5   'Reshape_To_ReshapeLayer1024'   shufflenet_9.Reshape_To_ReshapeLayer1024   shufflenet_9
     6   'Reshape_To_ReshapeLayer1029'   shufflenet_9.Reshape_To_ReshapeLayer1029   shufflenet_9
     7   'Reshape_To_ReshapeLayer1034'   shufflenet_9.Reshape_To_ReshapeLayer1034   shufflenet_9
     8   'Reshape_To_ReshapeLayer1039'   shufflenet_9.Reshape_To_ReshapeLayer1039   shufflenet_9
     9   'Reshape_To_ReshapeLayer1044'   shufflenet_9.Reshape_To_ReshapeLayer1044   shufflenet_9
    10   'Reshape_To_ReshapeLayer1049'   shufflenet_9.Reshape_To_ReshapeLayer1049   shufflenet_9
    11   'Reshape_To_ReshapeLayer1054'   shufflenet_9.Reshape_To_ReshapeLayer1054   shufflenet_9
    12   'Reshape_To_ReshapeLayer1059'   shufflenet_9.Reshape_To_ReshapeLayer1059   shufflenet_9
    13   'Reshape_To_ReshapeLayer1064'   shufflenet_9.Reshape_To_ReshapeLayer1064   shufflenet_9
    14   'Reshape_To_ReshapeLayer1069'   shufflenet_9.Reshape_To_ReshapeLayer1069   shufflenet_9
    15   'Reshape_To_ReshapeLayer1074'   shufflenet_9.Reshape_To_ReshapeLayer1074   shufflenet_9
    16   'Reshape_To_ReshapeLayer1079'   shufflenet_9.Reshape_To_ReshapeLayer1079   shufflenet_9
```

The classification layer does not contain the classes, so you must specify these before assembling the network. 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.

Import the class names from `squeezenet`, which is also trained with images from the ImageNet database.

```
SqueezeNet = squeezenet;
classNames = SqueezeNet.Layers(end).ClassNames;
```

The classification layer `cLayer` is the final layer of `lgraph`. Set the classes to `classNames` and then replace the imported classification layer with the new one.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_gpu_0_softmax_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
cLayer.Classes = classNames;
lgraph = replaceLayer(lgraph,lgraph.Layers(end).Name,cLayer);
```

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: [173×1 nnet.cnn.layer.Layer]
    Connections: [188×2 table]
     InputNames: {'gpu_0_data_0'}
    OutputNames: {'ClassificationLayer_gpu_0_softmax_1'}
```

Read the image you want to classify and display the size of the image. The image is 792-by-1056 pixels and has three color channels (RGB).

```
I = imread("peacock.jpg");
size(I)
```

```
ans = 1×3

         792        1056           3
```

Resize the image to the input size of the network. Show the image.

```
I = imresize(I,[224 224]);
imshow(I)
```

The inputs to `shufflenet` require further preprocessing (for details, see ShuffleNet in ONNX Model Zoo). Rescale the image. Normalize the image by subtracting the mean of the training images and dividing by the standard deviation of the training images.

```
I = rescale(I,0,1);

meanIm = [0.485 0.456 0.406];
stdIm = [0.229 0.224 0.225];
I = (I - reshape(meanIm,[1 1 3]))./reshape(stdIm,[1 1 3]);
```

Classify the image using the imported network.

```
label = classify(net,I)
```

```
label = categorical
     peacock
```

Helper Function

This section provides the code of the helper function `findCustomLayers` used in this example. `findCustomLayers` returns the `indices` of the custom layers that `importONNXLayers` automatically generates.

```
function indices = findCustomLayers(layers,PackageName)

s = what(['.\' PackageName]);

indices = zeros(1,length(s.m));
for i = 1:length(layers)
    for j = 1:length(s.m)
        if strcmpi(class(layers(i)),[PackageName(2:end) '.' s.m{j}(1:end-2)])
            indices(j) = i;
        end
    end
end

end
```

Import ONNX Model as Layer Graph with Placeholder Layers

Import an ONNX long short-term memory (LSTM) network as a layer graph, and then find and replace the placholder layers. 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.

`lstmNet` has a similar architecture to the LSTM network created in “Sequence Classification Using Deep Learning”. `lstmNet` is trained to recognize the speaker given time series data representing two Japanese vowels spoken in succession.

Specify `lstmNet` as the model file.

```
modelfile = "lstmNet.onnx";
```

Import the layers and weights of the ONNX network. By default, `importONNXLayers` imports the network as a `LayerGraph` object compatible with a `DAGNetwork` object.

```
lgraph = importONNXLayers("lstmNet.onnx")
```

```
Warning: Unable to import some ONNX operators, because they are not supported. They have been re

1 operators(s)    :    Unable to create an output layer for the ONNX network output 'softmax1001
                        If you know its format, pass it using the 'OutputDataFormats' argument.

To import the ONNX network as a function, use importONNXFunction.

lgraph =
  LayerGraph with properties:

         Layers: [6×1 nnet.cnn.layer.Layer]
    Connections: [5×2 table]
     InputNames: {'sequenceinput'}
    OutputNames: {1×0 cell}
```

`importONNXLayers` displays a warning and inserts a placeholder layer for the output layer.

You can check for placeholder layers by viewing the `Layers` property of `lgraph` or by using the `findPlaceholderLayers` function.

```
lgraph.Layers
```

```
ans =
  6×1 Layer array with layers:

     1   'sequenceinput'                 Sequence Input                          Sequence input wi
     2   'lstm1000'                      LSTM                                    LSTM with 100 hid
     3   'fc_MatMul'                     Fully Connected                         9 fully connected
     4   'fc_Add'                        Elementwise Affine                      Applies an elemen
     5   'Flatten_To_SoftmaxLayer1005'   lstmNet.Flatten_To_SoftmaxLayer1005   lstmNet.Flatten_T
     6   'OutputLayer_softmax1001'       PLACEHOLDER LAYER                       Placeholder for '
```

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  PlaceholderLayer with properties:

        Name: 'OutputLayer_softmax1001'
    ONNXNode: [1×1 struct]
     Weights: []

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Create an output layer to replace the placeholder layer. First, create a classification layer with the name `OutputLayer_softmax1001`. If you do not specify the classes, then the software automatically sets them to `1`, `2`, ..., `N`, where `N` is the number of classes. In this case, the class data is a categorical vector of labels "1","2",..."9", which correspond to nine speakers.

```
outputLayer = classificationLayer('Name','OutputLayer_softmax1001');
```

Replace the placeholder layers with `outputLayer` by using the `replaceLayer` function.

```
lgraph = replaceLayer(lgraph,'OutputLayer_softmax1001',outputLayer);
```

Display the `Layers` property of the layer graph to confirm the replacement.

```
lgraph.Layers
```

```
ans =
  6×1 Layer array with layers:

     1   'sequenceinput'                 Sequence Input                          Sequence input wi
     2   'lstm1000'                      LSTM                                    LSTM with 100 hid
     3   'fc_MatMul'                     Fully Connected                         9 fully connected
     4   'fc_Add'                        Elementwise Affine                      Applies an elemen
     5   'Flatten_To_SoftmaxLayer1005'   lstmNet.Flatten_To_SoftmaxLayer1005   lstmNet.Flatten_T
     6   'OutputLayer_softmax1001'       Classification Output                   crossentropyex
```

Alternatively, define the output layer when you import the layer graph by using the `OutputLayerType` or `OutputDataFormats` option. Check if the imported layer graphs have placeholder layers by using `findPlaceholderLayers`.

```
lgraph1 = importONNXLayers("lstmNet.onnx",OutputLayerType="classification");
findPlaceholderLayers(lgraph1)
```

```
ans =
  0×1 Layer array with properties:
```

```
lgraph2 = importONNXLayers("lstmNet.onnx",OutputDataFormats="BC");
findPlaceholderLayers(lgraph2)
```

```
ans =
  0×1 Layer array with properties:
```

The imported layer graphs `lgraph1` and `lgraph2` do not have placeholder layers.

Import and Assemble ONNX Network with Multiple Outputs

Import an ONNX network that has multiple outputs by using `importONNXLayers`, and then assemble the imported layer graph into a `DAGNetwork` object.

Specify the network file from which to import layers and weights.

```
modelfile = "digitsMIMO.onnx";
```

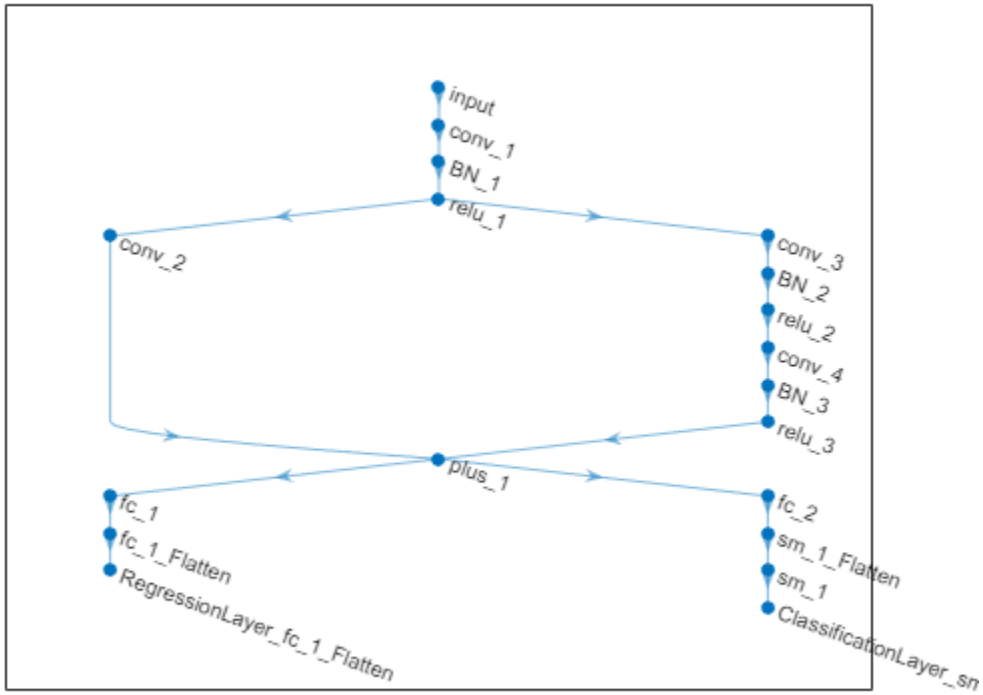
Import the layers and weights from `modelfile`. The network in `digitsMIMO.onnx` has two output layers: one classification layer (`ClassificationLayer_sm_1`) to classify digits and one regression layer (`RegressionLayer_fc_1_Flatten`) to compute the mean squared error for the predicted angles of the digits.

```
lgraph = importONNXLayers(modelfile)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [19×1 nnet.cnn.layer.Layer]
    Connections: [19×2 table]
     InputNames: {'input'}
    OutputNames: {'ClassificationLayer_sm_1'  'RegressionLayer_fc_1_Flatten'}
```

Plot the layer graph using `plot`, and display the layers of `lgraph`.

```
plot(lgraph)
```




```
lgraph.Layers

ans =
  19×1 Layer array with layers:

     1   'input'                          Image Input             28×28×1 images
     2   'conv_1'                         Convolution             16 5×5×1 convolutions with stri
     3   'BN_1'                           Batch Normalization     Batch normalization with 16 cha
     4   'relu_1'                         ReLU                    ReLU
     5   'conv_2'                         Convolution             32 1×1×16 convolutions with str
     6   'conv_3'                         Convolution             32 3×3×16 convolutions with str
     7   'BN_2'                           Batch Normalization     Batch normalization with 32 cha
     8   'relu_2'                         ReLU                    ReLU
     9   'conv_4'                         Convolution             32 3×3×32 convolutions with str
    10   'BN_3'                           Batch Normalization     Batch normalization with 32 cha
    11   'relu_3'                         ReLU                    ReLU
    12   'plus_1'                         Addition                Element-wise addition of 2 inpu
    13   'fc_1'                           Convolution             1 14×14×32 convolutions with st
    14   'fc_2'                           Convolution             10 14×14×32 convolutions with s
    15   'sm_1_Flatten'                   ONNX Flatten            Flatten activations into 1-D as
    16   'sm_1'                           Softmax                 softmax
    17   'fc_1_Flatten'                   ONNX Flatten            Flatten activations into 1-D as
    18   'ClassificationLayer_sm_1'       Classification Output   crossentropyex
    19   'RegressionLayer_fc_1_Flatten'   Regression Output       mean-squared-error
```

The classification layer does not contain the classes, so you must specify these before assembling the network. 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. Specify the classes of cLayer as 0, 1, ..., 9. Then, replace the imported classification layer with the new one.

```
ClassNames = string(0:9);
cLayer = lgraph.Layers(18);
cLayer.Classes = ClassNames;
lgraph = replaceLayer(lgraph,"ClassificationLayer_sm_1",cLayer);
```

Assemble the layer graph using assembleNetwork. The function returns a DAGNetwork object that is ready to use for prediction.

```
assembledNet = assembleNetwork(lgraph)
```

```
assembledNet =
  DAGNetwork with properties:

         Layers: [19×1 nnet.cnn.layer.Layer]
    Connections: [19×2 table]
     InputNames: {'input'}
    OutputNames: {'ClassificationLayer_sm_1'  'RegressionLayer_fc_1_Flatten'}
```

Input Arguments

modelfile — Name of ONNX model file

character vector | string scalar

Name of the ONNX model file containing the network, specified as a character vector or string scalar. The file must be in the current folder or in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: "cifarResNet.onnx"

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: importONNXLayers(modelfile,TargetNetwork="dagnetwork",GenerateCustomLayers=true,PackageName="CustomLayers") imports the network layers from modelfile as a layer graph compatible with a DAGNetwork object and saves the automatically generated custom layers in the package +CustomLayers in the current folder.

GenerateCustomLayers — Option for custom layer generation

true or 1 (default) | false or 0

Option for custom layer generation, specified as a numeric or logical 1 (true) or 0 (false). If you set GenerateCustomLayers to true, importONNXLayers tries to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. importONNXLayers saves each generated custom layer to a separate .m file in +PackageName. To view or edit a custom layer, open the associated .m file. For more information on custom layers, see "Deep Learning Custom Layers".

Example: GenerateCustomLayers=false

`PackageName` — Name of custom layers package
character vector | string scalar

Name of the package in which `importONNXLayers` saves custom layers, specified as a character vector or string scalar. `importONNXLayers` saves the custom layers package `+PackageName` in the current folder. If you do not specify `PackageName`, then `importONNXLayers` saves the custom layers in a package named `+modelfile` in the current folder. For more information on packages, see "Packages Create Namespaces".

Example: `PackageName="shufflenet_9"`

Example: `PackageName="CustomLayers"`

`TargetNetwork` — Target type of Deep Learning Toolbox network
`"dagnetwork"` (default) | `"dlnetwork"`

Target type of Deep Learning Toolbox network for imported network architecture, specified as `"dagnetwork"` or `"dlnetwork"`. The function `importONNXLayers` imports the network architecture as a `LayerGraph` object compatible with a `DAGNetwork` or `dlnetwork` object.

- If you specify `TargetNetwork` as `"dagnetwork"`, the imported `lgraph` must include input and output layers specified by the ONNX model or that you specify using the name-value arguments `InputDataFormats`, `OutputDataFormats`, or `OutputLayerType`.
- If you specify `TargetNetwork` as `"dlnetwork"`, `importONNXLayers` appends a `CustomOutputLayer` at the end of each output branch of `lgraph`, and might append a `CustomInputLayer` at the beginning of an input branch. The function appends a `CustomInputLayer` if the input data formats or input image sizes are not known. For network-specific information on the data formats of these layers, see the properties of the `CustomInputLayer` and `CustomOutputLayer` objects. For information on how to interpret Deep Learning Toolbox input and output data formats, see "Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers" on page 1-841.

Example: `TargetNetwork="dlnetwork"` imports a `LayerGraph` object compatible with a `dlnetwork` object.

`InputDataFormats` — Data format of network inputs
character vector | string scalar | string array

Data format of the network inputs, specified as a character vector, string scalar, or string array. `importONNXLayers` tries to interpret the input data formats from the ONNX file. The name-value argument `InputDataFormats` is useful when `importONNXLayers` cannot derive the input data formats.

Set `InputDataFomats` to a data format in the ordering of an ONNX input tensor. For example, if you specify `InputDataFormats` as `"BSSC"`, the imported network has one `imageInputLayer` input. For more information on how `importONNXLayers` interprets the data format of ONNX input tensors and how to specify `InputDataFormats` for different Deep Learning Toolbox input layers, see "Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers" on page 1-841.

If you specify an empty data format (`[]` or `""`), `importONNXLayers` automatically interprets the input data format.

Example: `InputDataFormats='BSSC'`

Example: `InputDataFormats="BSSC"`

Example: `InputDataFormats=["BCSS","","BC"]`

Example: `InputDataFormats={'BCSS',[],'BC'}`

Data Types: `char` | `string` | `cell`

`OutputDataFormats` — Data format of network outputs
character vector | string scalar | string array

Data format of the network outputs, specified as a character vector, string scalar, or string array. `importONNXLayers` tries to interpret the output data formats from the ONNX file. The name-value argument `OutputDataFormats` is useful when `importONNXLayers` cannot derive the output data formats.

Set `OutputDataFormats` to a data format in the ordering of an ONNX output tensor. For example, if you specify `OutputDataFormats` as `"BC"`, the imported network has one `classificationLayer` output. For more information on how `importONNXLayers` interprets the data format of ONNX output tensors and how to specify `OutputDataFormats` for different Deep Learning Toolbox output layers, see “Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers” on page 1-841.

If you specify an empty data format (`[]` or `""`), `importONNXLayers` automatically interprets the output data format.

Example: `OutputDataFormats='BC'`

Example: `OutputDataFormats="BC"`

Example: `OutputDataFormats=["BCSS","","BC"]`

Example: `OutputDataFormats={'BCSS',[],'BC'}`

Data Types: `char` | `string` | `cell`

`ImageInputSize` — Size of input image for first network input
vector of two or three numerical values

Size of the input image for the first network input, specified as a vector of three or four numerical values corresponding to `[height,width,channels]` for 2-D images and `[height,width,depth,channels]` for 3-D images. The network uses this information only when the ONNX model in `modelfile` does not specify the input size.

Example: `ImageInputSize=[28 28 1]` for a 2-D grayscale input image

Example: `ImageInputSize=[224 224 3]` for a 2-D color input image

Example: `ImageInputSize=[28 28 36 3]` for a 3-D color input image

`OutputLayerType` — Layer type for first network output
`"classification"` | `"regression"` | `"pixelclassification"`

Layer type for the first network output, specified as `"classification"`, `"regression"`, or `"pixelclassification"`. The function `importONNXLayers` appends a `ClassificationOutputLayer`, `RegressionOutputLayer`, or `pixelClassificationLayer` object to the end of the first output branch of the imported network architecture. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox. If the ONNX model in `modelfile` specifies the output layer type or you specify `TargetNetwork` as `"dlnetwork"`, `importONNXLayers` ignores the name-value argument `OutputLayerType`.

Example: `OutputLayerType="regression"`

`FoldConstants` — Constant folding optimization
`"deep"` (default) | `"shallow"` | `"none"`

Constant folding optimization, specified as `"deep"`, `"shallow"`, or `"none"`. Constant folding optimizes the imported network architecture by computing operations on ONNX initializers (initial constant values) during the conversion of ONNX operators to equivalent built-in MATLAB layers.

If the ONNX network contains operators that the software cannot convert to equivalent built-in MATLAB layers (see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers” on page 1-839), then `importONNXLayers` inserts a placeholder layer in place of each unsupported layer. For more information, see “Tips” on page 1-844.

Constant folding optimization can reduce the number of placeholder layers. When you set `FoldConstants` to `"deep"`, the imported layers include the same or fewer placeholder layers, compared to when you set the argument to `"shallow"`. However, the importing time might increase. Set `FoldConstants` to `"none"` to disable the network architecture optimization.

Example: `FoldConstants="shallow"`

Output Arguments

`lgraph` — Network architecture of pretrained ONNX model
`LayerGraph` object

Network architecture of the pretrained ONNX model, returned as a `LayerGraph` object.

To use the imported layer graph for prediction, you must convert the `LayerGraph` object to a `DAGNetwork` or `dlnetwork` object. Specify the name-value argument `TargetNetwork` as `"dagnetwork"` or `"dlnetwork"` depending on the intended workflow.

- Convert a `LayerGraph` to a `DAGNetwork` by using `assembleNetwork`. On the `DAGNetwork` object, you then predict class labels using the `classify` function.
- Convert a `LayerGraph` to a `dlnetwork` by using `dlnetwork`. On the `dlnetwork` object, you then predict class labels using the `predict` function. Specify the input data as a `dlarray` using the correct data format (for more information, see the `fmt` argument of `dlarray`).

Limitations

- `importONNXLayers` supports ONNX versions as follows:
  - The function supports ONNX intermediate representation version 6.
  - The function supports ONNX operator sets 6 to 13.

Note If you import an exported network, layers of the reimported network might differ from the original network and might not be supported.

More About

ONNX Operators Supported for Conversion into Built-In MATLAB Layers

`importONNXLayers` supports the following ONNX operators for conversion into built-in MATLAB layers, with some limitations.

ONNX Operator	Deep Learning Toolbox Layer
`Add`	`additionLayer` or `nnet.onnx.layer.ElementwiseAffineLayer`
`AveragePool`	`averagePooling2dLayer`
`BatchNormalization`	`batchNormalizationLayer`
`Concat`	`concatenationLayer`
`Constant`	None (Imported as weights)
`Conv*`	`convolution2dLayer`
`ConvTranspose`	`transposedConv2dLayer`
`Dropout`	`dropoutLayer`
`Elu`	`eluLayer`
`Gemm`	`fullyConnectedLayer` if ONNX network is recurrent, otherwise `nnet.onnx.layer.FlattenLayer` followed by `convolution2dLayer`
`GlobalAveragePool`	`globalAveragePooling2dLayer`
`GlobalMaxPool`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`InstanceNormalization`	`groupNormalizationLayer` with `numGroups` specified as `"channel-wise"`
`LeakyRelu`	`leakyReluLayer`
`LRN`	`CrossChannelNormalizationLayer`
`LSTM`	`lstmLayer` or `bilstmLayer`
`MatMul`	`fullyConnectedLayer` if ONNX network is recurrent, otherwise `convolution2dLayer`
`MaxPool`	`maxPooling2dLayer`
`Mul`	`multiplicationLayer`
`Relu`	`reluLayer` or `clippedReluLayer`
`Sigmoid`	`sigmoidLayer`
`Softmax`	`softmaxLayer`
`Sum`	`additionLayer`
`Tanh`	`tanhLayer`

*If the `pads` attribute of the `Conv` operator is a vector with only two elements `[p1,p2]`, `importONNXLayers` imports `Conv` as a `convolution2dLayer` with the name-value argument `'Padding'` specified as `[p1,p2,p1,p2]`.

ONNX Operator	ONNX Importer Custom Layer
`Clip`	`nnet.onnx.layer.ClipLayer`
`Div`	`nnet.onnx.layer.ElementwiseAffineLayer`

ONNX Operator	ONNX Importer Custom Layer
`Flatten`	`nnet.onnx.layer.FlattenLayer` or `nnet.onnx.layer.Flatten3dLayer`
`Identity`	`nnet.onnx.layer.IdentityLayer`
`ImageScaler`	`nnet.onnx.layer.ElementwiseAffineLayer`
`PRelu`	`nnet.onnx.layer.PReluLayer`
`Reshape`	`nnet.onnx.layer.FlattenLayer`
`Sub`	`nnet.onnx.layer.ElementwiseAffineLayer`

ONNX Operator	Image Processing Toolbox
`DepthToSpace`	`depthToSpace2dLayer`
`Resize`	`resize2dLayer` or `resize3dLayer`
`SpaceToDepth`	`spaceToDepthLayer`
`Upsample`	`resize2dLayer` or `resize3dLayer`

Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers

`importONNXLayers` tries to interpret the data format of the ONNX network's input and output tensors, and then convert them into built-in MATLAB input and output layers. For details on the interpretation, see the tables Conversion of ONNX Input Tensors into Deep Learning Toolbox Layers and Conversion of ONNX Output Tensors into MATLAB Layers.

In Deep Learning Toolbox, each data format character must be one of these labels:

- `S` — Spatial
- `C` — Channel
- `B` — Batch observations
- `T` — Time or sequence
- `U` — Unspecified

Conversion of ONNX Input Tensors into Deep Learning Toolbox Layers

Data Formats		Data Interpretation		Deep Learning Toolbox Layer
ONNX Input Tensor	**MATLAB Input Format**	**Shape**	**Type**	
BC	CB	*c*-by-*n* array, where *c* is the number of features and *n* is the number of observations	Features	`featureInputLayer`
BCSS, BSSC, CSS, SSC	SSCB	*h*-by-*w*-by-*c*-by-*n* numeric array, where *h*, *w*, *c* and *n* are the height, width, number of channels of the images, and number of observations, respectively	2-D image	`imageInputLayer`
BCSSS, BSSSC, CSSS, SSSC	SSSCB	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, *c* and *n* are the height, width, depth, number of channels of the images, and number of image observations, respectively	3-D image	`image3dInputLayer`
TBC	CBT	*c*-by-*s*-by-*n* matrix, where *c* is the number of features of the sequence, *s* is the sequence length, and *n* is the number of sequence observations	Vector sequence	`sequenceInputLayer`

Data Formats		Data Interpretation		Deep Learning Toolbox Layer
ONNX Input Tensor	**MATLAB Input Format**	**Shape**	**Type**	
TBCSS	SSCBT	*h*-by-*w*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, *c* and *n* correspond to the height, width, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations	2-D image sequence	`sequenceInputLayer`
TBCSSS	SSSCBT	*h*-by-*w*-by-*d*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations	3-D image sequence	`sequenceInputLayer`

Conversion of ONNX Output Tensors into MATLAB Layers

Data Formats		MATLAB Layer
ONNX Output Tensor	**MATLAB Output Format**	
BC, TBC	CB, CBT	`classificationLayer`
BCSS, BSSC, CSS, SSC, BCSSS, BSSSC, CSSS, SSSC	SSCB, SSSCB	`pixelClassificationLayer`
TBCSS, TBCSSS	SSCBT, SSSCBT	`regressionLayer`

Use Imported Network Layers on GPU

`importONNXLayers` does not execute on a GPU. However, `importONNXLayers` imports the layers of a pretrained neural network for deep learning as a `LayerGraph` object, which you can use on a GPU.

- Convert the imported `LayerGraph` object to a `DAGNetwork` object by using `assembleNetwork`. On the `DAGNetwork` object, you can then predict class labels on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument

`ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function and specify the name-value argument `ReturnCategorical` as `true`.

- Convert the imported `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. On the `dlnetwork` object, you can then predict class labels on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.

  - If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
  - Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

    ```
    dlnet = dlupdate(@gpuarray,dlnet)
    ```

- You can train the imported `LayerGraph` object on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- If the imported network contains an ONNX operator not supported for conversion into a built-in MATLAB layer (see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers” on page 1-839) and `importONNXLayers` does not generate a custom layer, then `importONNXLayers` inserts a placeholder layer in place of the unsupported layer. To find the names and indices of the unsupported layers in the network, use the `findPlaceholderLayers` function. You then can replace a placeholder layer with a new layer that you define. To replace a layer, use `replaceLayer`. For an example, see “Import and Assemble ONNX Network with Multiple Outputs” on page 1-834.
- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.

Alternative Functionality

Deep Learning Toolbox Converter for ONNX Model Format provides three functions to import a pretrained ONNX network: `importONNXNetwork`, `importONNXLayers`, and `importONNXFunction`. For more information on which import function best suits different scenarios, see “Select Function to Import ONNX Pretrained Network”.

Compatibility Considerations

`ImportWeights` option has been removed
Warns starting in R2021b

`ImportWeights` has been removed. Starting in R2021b, the ONNX model weights are automatically imported. In most cases, you do not need to make any changes to your code.

- If `ImportWeights` is not set in your code, `importONNXLayers` now imports the weights.
- If `ImportWeights` is set to `true` in your code, the behavior of `importONNXLayers` remains the same.
- If `ImportWeights` is set to `false` in your code, `importONNXLayers` now ignores the name-value argument `ImportWeights` and imports the weights.

`importONNXLayers` cannot create input and output layers from ONNX file information
Behavior changed in R2021b

If you import an ONNX model as a `LayerGraph` object compatible with a `DAGNetwork` object, the imported layer graph must include input and output layers. `importONNXLayers` tries to convert the input and output ONNX tensors into built-in MATLAB layers. When importing some networks, which `importONNXLayers` could previously import with input and output built-in MATLAB layers, `importONNXLayers` might now insert placeholder layers. In this case, do one of the following to update your code:

- Specify the name-value argument `TargetNetwork` as `"dlnetwork"` to import the network as a `LayerGraph` object compatible with a `dlnetwork` object.
- Use the name-value arguments `InputDataFormats`, `OutputDataFormats`, and `OutputLayerType` to specify the imported network's inputs and outputs.
- Use `importONNXFunction` to import the network as a model function and an `ONNXParameters` object.

References

[1] *Open Neural Network Exchange*. https://github.com/onnx/.

[2] *ONNX*. https://onnx.ai/.

See Also

`importCaffeNetwork` | `importCaffeLayers` | `importKerasNetwork` | `importKerasLayers` | `importONNXNetwork` | `exportONNXNetwork` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `importONNXFunction` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

“Select Function to Import ONNX Pretrained Network”
“Multiple-Input and Multiple-Output Networks”

Introduced in R2018a

importONNXNetwork

Import pretrained ONNX network

Syntax

```
net = importONNXNetwork(modelfile)
net = importONNXNetwork(modelfile,Name=Value)
```

Description

`net = importONNXNetwork(modelfile)` imports a pretrained ONNX (Open Neural Network Exchange) network from the file `modelfile`. The function returns the network `net` as a `DAGNetwork` or `dlnetwork` object.

`importONNXNetwork` requires the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then `importONNXNetwork` provides a download link.

Note By default, `importONNXNetwork` tries to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. For a list of operators for which the software supports conversion, see "ONNX Operators Supported for Conversion into Built-In MATLAB Layers" on page 1-860.

`importONNXNetwork` saves the generated custom layers in the package `+modelfile`.

`importONNXNetwork` does not automatically generate a custom layer for each ONNX operator that is not supported for conversion into a built-in MATLAB layer. For more information on how to handle unsupported layers, see "Alternative Functionality" on page 1-865.

`net = importONNXNetwork(modelfile,Name=Value)` imports a pretrained ONNX network with additional options specified by one or more name-value arguments. For example, `OutputLayerType="classification"` imports the network as a `DAGNetwork` object with a classification output layer appended to the end of the first output branch of the imported network architecture.

Examples

Download and Install Deep Learning Toolbox Converter for ONNX Model Format

Download and install the Deep Learning Toolbox Converter for ONNX Model Format support package.

Type `importONNXNetwork` at the command line.

```
importONNXNetwork
```

If Deep Learning Toolbox Converter for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support

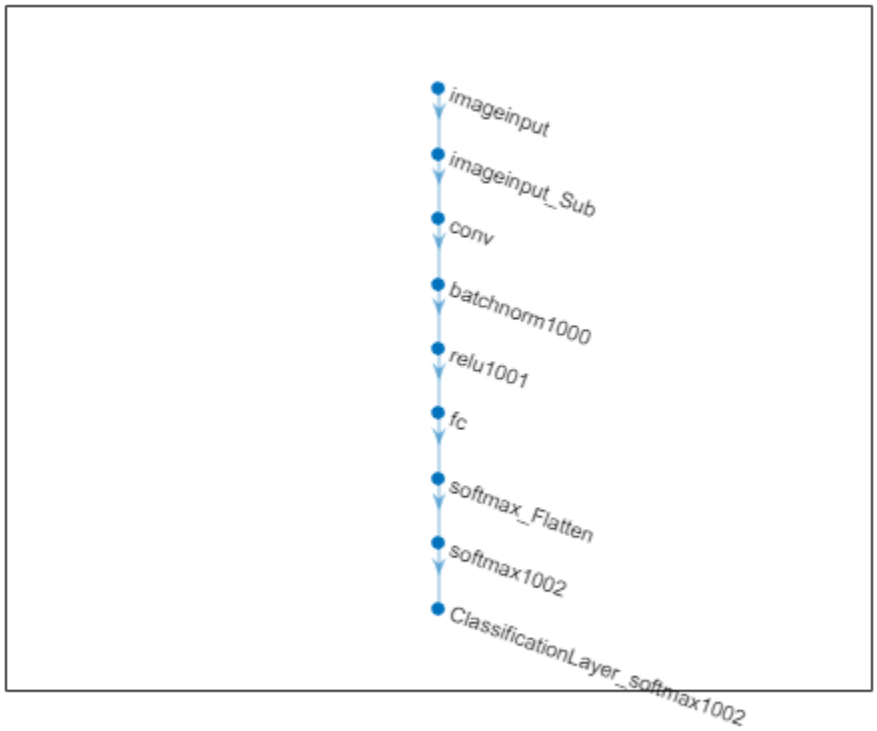
package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file `"simplenet.onnx"` at the command line. If the support package is installed, then the function returns a `DAGNetwork` object.

```
modelfile = "simplenet.onnx";
net = importONNXNetwork(modelfile)
```

```
net =
  DAGNetwork with properties:

         Layers: [9×1 nnet.cnn.layer.Layer]
    Connections: [8×2 table]
     InputNames: {'imageinput'}
    OutputNames: {'ClassificationLayer_softmax1002'}
```

Plot the network architecture.

```
plot(net)
```



Import ONNX Network as `DAGNetwork`

Import a pretrained ONNX network as a `DAGNetwork` object, and use the imported network to classify an image.

Generate an ONNX model of the `squeezenet` convolution neural network.

```
squeezeNet = squeezenet;
exportONNXNetwork(squeezeNet,"squeezeNet.onnx");
```

Specify the class names.

```
ClassNames = squeezeNet.Layers(end).Classes;
```

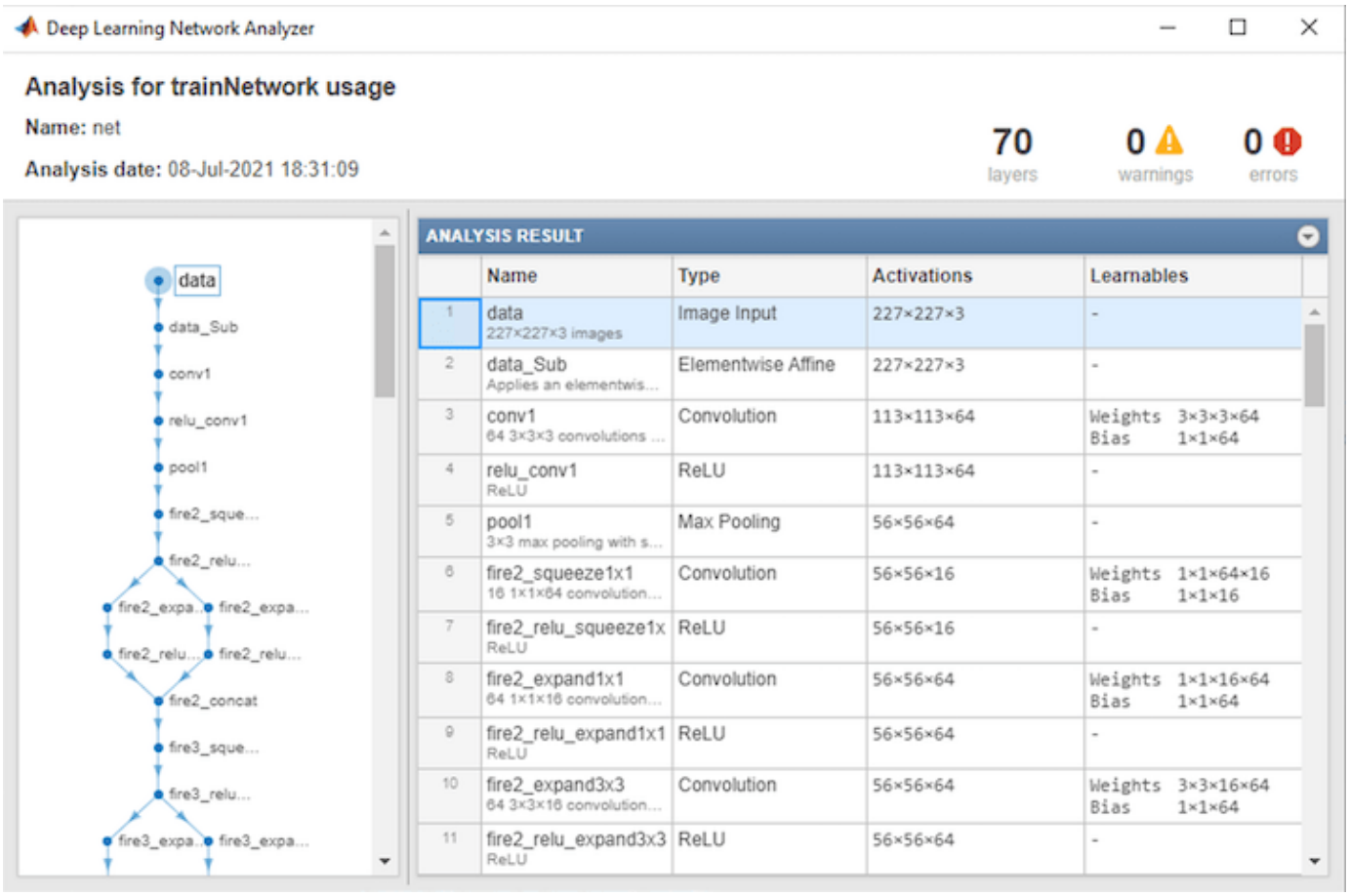
Import the pretrained `squeezeNet.onnx` model, and specify the classes. By default, `importONNXNetwork` imports the network as a `DAGNetwork` object.

```
net = importONNXNetwork("squeezeNet.onnx",Classes=ClassNames)
```

```
net =
  DAGNetwork with properties:

         Layers: [70×1 nnet.cnn.layer.Layer]
    Connections: [77×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_prob'}
```

Analyze the imported network.

```
analyzeNetwork(net)
```



Read the image you want to classify and display the size of the image. The image is 384-by-512 pixels and has three color channels (RGB).

```
I = imread("peppers.png");
size(I)
```

```
ans = 1×3

   384   512     3
```

Resize the image to the input size of the network. Show the image.

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

Classify the image using the imported network.

```
label = classify(net,I)
```

```
label = categorical
     bell pepper
```

Import ONNX Network as `dlnetwork`

Import a pretrained ONNX network as a `dlnetwork` object, and use the imported network to classify an image.

Generate an ONNX model of the `squeezenet` convolution neural network.

```
squeezeNet = squeezenet;
exportONNXNetwork(squeezeNet,"squeezeNet.onnx");
```

Specify the class names.

```
ClassNames = squeezeNet.Layers(end).Classes;
```

Import the pretrained `squeezeNet.onnx` model as a `dlnetwork` object.

```
net = importONNXNetwork("squeezeNet.onnx",TargetNetwork="dlnetwork")
```

```
net =
  dlnetwork with properties:

         Layers: [70×1 nnet.cnn.layer.Layer]
    Connections: [77×2 table]
     Learnables: [52×3 table]
          State: [0×3 table]
     InputNames: {'data'}
    OutputNames: {'probOutput'}
    Initialized: 1
```

Read the image you want to classify and display the size of the image. The image is 384-by-512 pixels and has three color channels (RGB).

```
I = imread("peppers.png");
size(I)
```

```
ans = 1×3

   384   512     3
```

Resize the image to the input size of the network. Show the image.

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

Convert the image to a `dlarray`. Format the images with the dimensions `"SSCB"` (spatial, spatial, channel, batch). In this case, the batch size is 1 and you can omit it (`"SSC"`).

```
I_dlarray = dlarray(single(I),"SSCB");
```

Classify the sample image and find the predicted label.

```
prob = predict(net,I_dlarray);
[~,label] = max(prob);
```

Display the classification result.

```
ClassNames(label)
```

```
ans = categorical
     bell pepper
```

Import ONNX Network with Autogenerated Custom Layers

Import a pretrained ONNX network as a `DAGNetwork` object, and use the imported network to classify an image. The imported network contains ONNX operators that are not supported for conversion into built-in MATLAB layers. The software automatically generates custom layers when you import these operators.

This example uses the helper function `findCustomLayers`. To view the code for this function, see Helper Function on page 1-0 .

Specify the model file to import as `shufflenet` with operator set 9 from the ONNX Model Zoo. `shufflenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

```
modelfile = "shufflenet-9.onnx";
```

Import the class names from `squeezenet`, which is also trained with images from the ImageNet database.

```
squeezeNet = squeezenet;
ClassNames = squeezeNet.Layers(end).ClassNames;
```

Import `shufflenet`. By default, `importONNXNetwork` imports the network as a `DAGNetwork` object. If the imported network contains ONNX operators not supported for conversion into built-in MATLAB layers, then `importONNXNetwork` can automatically generate custom layers in place of these operators. `importONNXNetwork` saves each generated custom layer to a separate `.m` file in the package `+shufflenet_9` in the current folder. Specify the package name by using the name-value argument `PackageName`.

```
net = importONNXNetwork(modelfile,...
    Classes=ClassNames,PackageName="shufflenet_9")
```

```
net =
  DAGNetwork with properties:
```

```
        Layers: [173×1 nnet.cnn.layer.Layer]
    Connections: [188×2 table]
     InputNames: {'gpu_0_data_0'}
    OutputNames: {'ClassificationLayer_gpu_0_softmax_1'}
```

Find the indices of the automatically generated custom layers by using the helper function `findCustomLayers`, and display the custom layers.

```
ind = findCustomLayers(net.Layers,'+shufflenet_9');
net.Layers(ind)
```

```
ans =
  16×1 Layer array with layers:

     1   'Reshape_To_ReshapeLayer1004'   shufflenet_9.Reshape_To_ReshapeLayer1004   shufflenet_9
     2   'Reshape_To_ReshapeLayer1009'   shufflenet_9.Reshape_To_ReshapeLayer1009   shufflenet_9
     3   'Reshape_To_ReshapeLayer1014'   shufflenet_9.Reshape_To_ReshapeLayer1014   shufflenet_9
     4   'Reshape_To_ReshapeLayer1019'   shufflenet_9.Reshape_To_ReshapeLayer1019   shufflenet_9
     5   'Reshape_To_ReshapeLayer1024'   shufflenet_9.Reshape_To_ReshapeLayer1024   shufflenet_9
     6   'Reshape_To_ReshapeLayer1029'   shufflenet_9.Reshape_To_ReshapeLayer1029   shufflenet_9
     7   'Reshape_To_ReshapeLayer1034'   shufflenet_9.Reshape_To_ReshapeLayer1034   shufflenet_9
     8   'Reshape_To_ReshapeLayer1039'   shufflenet_9.Reshape_To_ReshapeLayer1039   shufflenet_9
     9   'Reshape_To_ReshapeLayer1044'   shufflenet_9.Reshape_To_ReshapeLayer1044   shufflenet_9
    10   'Reshape_To_ReshapeLayer1049'   shufflenet_9.Reshape_To_ReshapeLayer1049   shufflenet_9
    11   'Reshape_To_ReshapeLayer1054'   shufflenet_9.Reshape_To_ReshapeLayer1054   shufflenet_9
    12   'Reshape_To_ReshapeLayer1059'   shufflenet_9.Reshape_To_ReshapeLayer1059   shufflenet_9
    13   'Reshape_To_ReshapeLayer1064'   shufflenet_9.Reshape_To_ReshapeLayer1064   shufflenet_9
    14   'Reshape_To_ReshapeLayer1069'   shufflenet_9.Reshape_To_ReshapeLayer1069   shufflenet_9
    15   'Reshape_To_ReshapeLayer1074'   shufflenet_9.Reshape_To_ReshapeLayer1074   shufflenet_9
    16   'Reshape_To_ReshapeLayer1079'   shufflenet_9.Reshape_To_ReshapeLayer1079   shufflenet_9
```

Read the image you want to classify and display the size of the image. The image is 792-by-1056 pixels and has three color channels (RGB).

```
I = imread("peacock.jpg");
size(I)
```

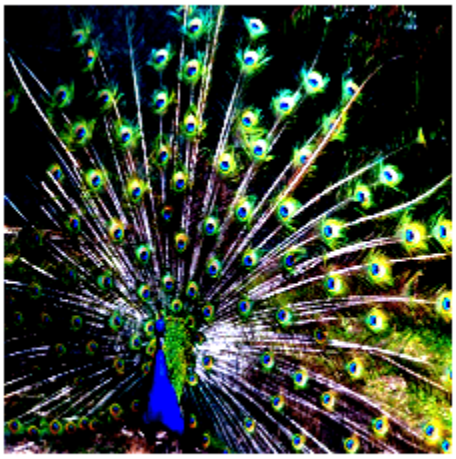
```
ans = 1×3

         792        1056           3
```

Resize the image to the input size of the network. Show the image.

```
I = imresize(I,[224 224]);
imshow(I)
```

The inputs to `shufflenet` require further preprocessing (for details, see ShuffleNet in ONNX Model Zoo). Rescale the image. Normalize the image by subtracting the mean of the training images and dividing by the standard deviation of the training images.

```
I = rescale(I,0,1);

meanIm = [0.485 0.456 0.406];
stdIm = [0.229 0.224 0.225];
I = (I - reshape(meanIm,[1 1 3]))./reshape(stdIm,[1 1 3]);

imshow(I)
```



Classify the image using the imported network.

```
label = classify(net,I)
```

```
label = categorical
     peacock
```

Helper Function

This section provides the code of the helper function `findCustomLayers` used in this example. `findCustomLayers` returns the `indices` of the custom layers that `importONNXNetwork` automatically generates.

```
function indices = findCustomLayers(layers,PackageName)

s = what(['.\' PackageName]);

indices = zeros(1,length(s.m));
for i = 1:length(layers)
    for j = 1:length(s.m)
        if strcmpi(class(layers(i)),[PackageName(2:end) '.' s.m{j}(1:end-2)])
            indices(j) = i;
        end
    end
end

end
```

Import ONNX Network with Multiple Outputs

Import an ONNX network that has multiple outputs as a `DAGNetwork` object.

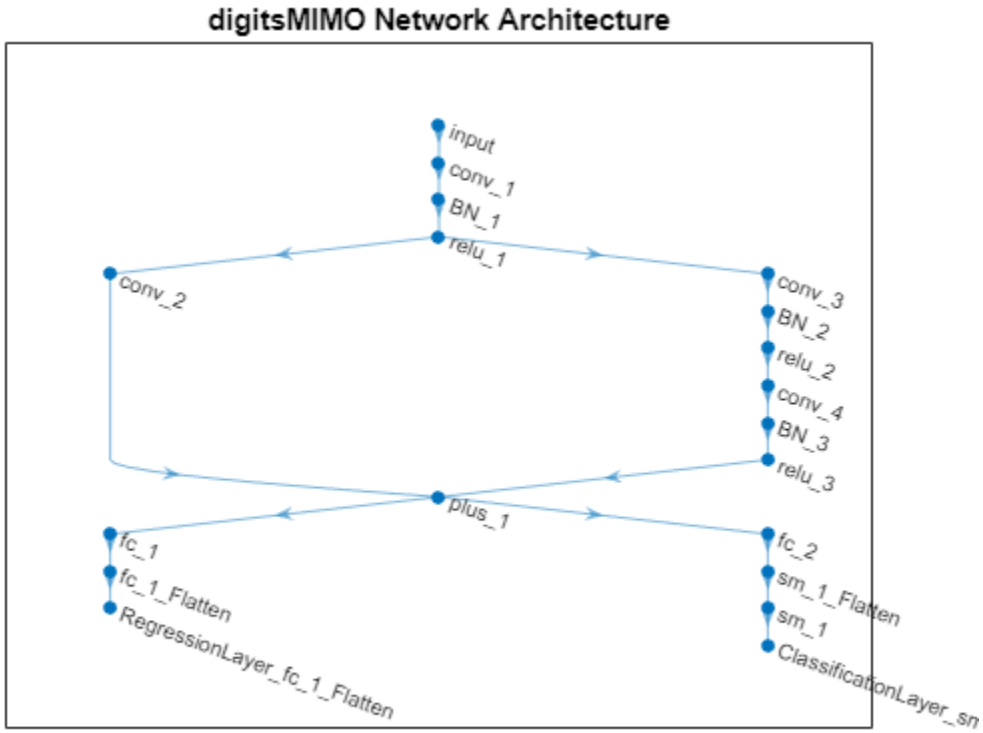
Specify the ONNX model file and import the pretrained ONNX model. By default, `importONNXNetwork` imports the network as a `DAGNetwork` object.

```
modelfile = "digitsMIMO.onnx";
net = importONNXNetwork(modelfile)
```

```
net =
  DAGNetwork with properties:

         Layers: [19×1 nnet.cnn.layer.Layer]
    Connections: [19×2 table]
     InputNames: {'input'}
    OutputNames: {'ClassificationLayer_sm_1'  'RegressionLayer_fc_1_Flatten'}
```

The network has two output layers: one classification layer (`ClassificationLayer_sm_1`) to classify digits and one regression layer (`RegressionLayer_fc_1_Flatten`) to compute the mean squared error for the predicted angles of the digits. Plot the network architecture.

```
plot(net)
title('digitsMIMO Network Architecture')
```



To make predictions using the imported network, use the `predict` function and set the `ReturnCategorical` option to `true`.

Input Arguments

`modelfile` — Name of ONNX model file

character vector | string scalar

Name of the ONNX model file containing the network, specified as a character vector or string scalar. The file must be in the current folder or in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: `"cifarResNet.onnx"`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `importONNXNetwork(modelfile,TargetNetwork="dagnetwork",GenerateCustomLayers=true,PackageName="CustomLayers")` imports the network in `modelfile` as a `DAGNetwork` object and saves the automatically generated custom layers in the package `+CustomLayers` in the current folder.

`GenerateCustomLayers` — Option for custom layer generation
`true` or `1` (default) | `false` or `0`

Option for custom layer generation, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you set `GenerateCustomLayers` to `true`, `importONNXNetwork` tries to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. `importONNXNetwork` saves each generated custom layer to a separate `.m` file in `+PackageName`. To view or edit a custom layer, open the associated `.m` file. For more information on custom layers, see "Deep Learning Custom Layers".

Example: `GenerateCustomLayers=false`

`PackageName` — Name of custom layers package
character vector | string scalar

Name of the package in which `importONNXNetwork` saves custom layers, specified as a character vector or string scalar. `importONNXNetwork` saves the custom layers package `+PackageName` in the current folder. If you do not specify `PackageName`, then `importONNXNetwork` saves the custom layers in a package named `+modelfile` in the current folder. For more information on packages, see "Packages Create Namespaces".

Example: `PackageName="shufflenet_9"`

Example: `PackageName="CustomLayers"`

`TargetNetwork` — Target type of Deep Learning Toolbox network
`"dagnetwork"` (default) | `"dlnetwork"`

Target type of Deep Learning Toolbox network, specified as `"dagnetwork"` or `"dlnetwork"`. The function `importONNXNetwork` imports the network `net` as a `DAGNetwork` or `dlnetwork` object.

- If you import the network as a `DAGNetwork` object, `net` must include input and output layers specified by the ONNX model or that you specify using the name-value arguments `InputDataFormats`, `OutputDataFormats`, or `OutputLayerType`.
- If you import the network as a `dlnetwork` object, `importONNXNetwork` appends a `CustomOutputLayer` at the end of each output branch of `net`, and might append a `CustomInputLayer` at the beginning of an input branch. The function appends a `CustomInputLayer` if the input data formats or input image sizes are not known. For network-specific information on the data formats of these layers, see the properties of the `CustomInputLayer` and `CustomOutputLayer` objects. For information on how to interpret Deep Learning Toolbox input and output data formats, see "Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers" on page 1-861.

Example: `TargetNetwork="dlnetwork"`

`InputDataFormats` — Data format of network inputs
character vector | string scalar | string array

Data format of the network inputs, specified as a character vector, string scalar, or string array. `importONNXNetwork` tries to interpret the input data formats from the ONNX file. The name-value argument `InputDataFormats` is useful when `importONNXNetwork` cannot derive the input data formats.

Set `InputDataFomats` to a data format in the ordering of an ONNX input tensor. For example, if you specify `InputDataFormats` as `"BSSC"`, the imported network has one `imageInputLayer` input. For

more information on how `importONNXNetwork` interprets the data format of ONNX input tensors and how to specify `InputDataFormats` for different Deep Learning Toolbox input layers, see “Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers” on page 1-861.

If you specify an empty data format (`[]` or `""`), `importONNXNetwork` automatically interprets the input data format.

Example: `InputDataFormats='BSSC'`

Example: `InputDataFormats="BSSC"`

Example: `InputDataFormats=["BCSS","","BC"]`

Example: `InputDataFormats={'BCSS',[],'BC'}`

Data Types: `char` | `string` | `cell`

`OutputDataFormats` — Data format of network outputs
character vector | string scalar | string array

Data format of the network outputs, specified as a character vector, string scalar, or string array. `importONNXNetwork` tries to interpret the output data formats from the ONNX file. The name-value argument `OutputDataFormats` is useful when `importONNXNetwork` cannot derive the output data formats.

Set `OutputDataFormats` to a data format in the ordering of an ONNX output tensor. For example, if you specify `OutputDataFormats` as `"BC"`, the imported network has one `classificationLayer` output. For more information on how `importONNXNetwork` interprets the data format of ONNX output tensors and how to specify `OutputDataFormats` for different Deep Learning Toolbox output layers, see “Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers” on page 1-861.

If you specify an empty data format (`[]` or `""`), `importONNXNetwork` automatically interprets the output data format.

Example: `OutputDataFormats='BC'`

Example: `OutputDataFormats="BC"`

Example: `OutputDataFormats=["BCSS","","BC"]`

Example: `OutputDataFormats={'BCSS',[],'BC'}`

Data Types: `char` | `string` | `cell`

`ImageInputSize` — Size of input image for first network input
vector of two or three numerical values

Size of the input image for the first network input, specified as a vector of three or four numerical values corresponding to `[height,width,channels]` for 2-D images and `[height,width,depth,channels]` for 3-D images. The network uses this information only when the ONNX model in `modelfile` does not specify the input size.

Example: `ImageInputSize=[28 28 1]` for a 2-D grayscale input image

Example: `ImageInputSize=[224 224 3]` for a 2-D color input image

Example: `ImageInputSize=[28 28 36 3]` for a 3-D color input image

`OutputLayerType` — Layer type for first network output
`"classification"` | `"regression"` | `"pixelclassification"`

Layer type for the first network output, specified as `"classification"`, `"regression"`, or `"pixelclassification"`. The function `importONNXNetwork` appends a `ClassificationOutputLayer`, `RegressionOutputLayer`, or `pixelClassificationLayer` object to the end of the first output branch of the imported network architecture. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox. If the ONNX model in `modelfile` specifies the output layer type or you specify `TargetNetwork` as `"dlnetwork"`, `importONNXNetwork` ignores the name-value argument `OutputLayerType`.

Example: `OutputLayerType="regression"`

`Classes` — Classes of output layer for first network output
`"auto"` (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer for the first network output, specified as a categorical vector, string array, cell array of character vectors, or `"auto"`. If `Classes` is `"auto"`, then `importONNXNetwork` sets the classes to `categorical(1:N)`, where `N` is the number of classes. If you specify a string array or cell array of character vectors `str`, then `importONNXNetwork` sets the classes of the output layer to `categorical(str,str)`. If you specify `TargetNetwork` as `"dlnetwork"`, `importONNXNetwork` ignores the name-value argument `Classes`.

Example: `Classes={'0','1','3'}`

Example: `Classes=categorical({'dog','cat'})`

Data Types: `char` | `categorical` | `string` | `cell`

`FoldConstants` — Constant folding optimization
`"deep"` (default) | `"shallow"` | `"none"`

Constant folding optimization, specified as `"deep"`, `"shallow"`, or `"none"`. Constant folding optimizes the imported network architecture by computing operations on ONNX initializers (initial constant values) during the conversion of ONNX operators to equivalent built-in MATLAB layers.

If the ONNX network contains operators that the software cannot convert to equivalent built-in MATLAB layers (see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers” on page 1-860), constant folding optimization can reduce the number of unsupported layers. When you set `FoldConstants` to `"deep"`, the network has the same or fewer unsupported layers, compared to when you set the argument to `"shallow"`. However, the network importing time might increase. Set `FoldConstants` to `"none"` to disable the network architecture optimization.

If the network still contains unsupported layers after constant folding optimization, `importONNXNetwork` returns an error. In this case, you can import the network by using `importONNXLayers` or `importONNXFunction`. For more information, see “Alternative Functionality” on page 1-865.

Example: `FoldConstants="shallow"`

Output Arguments

`net` — Pretrained ONNX network
`DAGNetwork` object | `dlnetwork` object

Pretrained ONNX network, returned as a `DAGNetwork` or `dlnetwork` object.

- Specify `TargetNetwork` as `"dagnetwork"` to import the network as a `DAGNetwork` object. On the `DAGNetwork` object, you then predict class labels by using the `classify` function.

- Specify `TargetNetwork` as `"dlnetwork"` to import the network as a `dlnetwork` object. On the `dlnetwork` object, you then predict class labels by using the `predict` function. Specify the input data as a `dlarray` using the correct data format (for more information, see the `fmt` argument of `dlarray`).

Limitations

- `importONNXNetwork` supports ONNX versions as follows:
  - The function supports ONNX intermediate representation version 6.
  - The function supports ONNX operator sets 6 to 13.

Note If you import an exported network, layers of the reimported network might differ from the original network and might not be supported.

More About

ONNX Operators Supported for Conversion into Built-In MATLAB Layers

`importONNXNetwork` supports the following ONNX operators for conversion into built-in MATLAB layers, with some limitations.

ONNX Operator	Deep Learning Toolbox Layer
`Add`	`additionLayer` or `nnet.onnx.layer.ElementwiseAffineLayer`
`AveragePool`	`averagePooling2dLayer`
`BatchNormalization`	`batchNormalizationLayer`
`Concat`	`concatenationLayer`
`Constant`	None (Imported as weights)
`Conv*`	`convolution2dLayer`
`ConvTranspose`	`transposedConv2dLayer`
`Dropout`	`dropoutLayer`
`Elu`	`eluLayer`
`Gemm`	`fullyConnectedLayer` if ONNX network is recurrent, otherwise `nnet.onnx.layer.FlattenLayer` followed by `convolution2dLayer`
`GlobalAveragePool`	`globalAveragePooling2dLayer`
`GlobalMaxPool`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`InstanceNormalization`	`groupNormalizationLayer` with `numGroups` specified as `"channel-wise"`
`LeakyRelu`	`leakyReluLayer`
`LRN`	`CrossChannelNormalizationLayer`

ONNX Operator	Deep Learning Toolbox Layer
LSTM	lstmLayer or bilstmLayer
MatMul	fullyConnectedLayer if ONNX network is recurrent, otherwise convolution2dLayer
MaxPool	maxPooling2dLayer
Mul	multiplicationLayer
Relu	reluLayer or clippedReluLayer
Sigmoid	sigmoidLayer
Softmax	softmaxLayer
Sum	additionLayer
Tanh	tanhLayer

*If the pads attribute of the Conv operator is a vector with only two elements [p1,p2], importONNXNetwork imports Conv as a convolution2dLayer with the name-value argument 'Padding' specified as [p1,p2,p1,p2].

ONNX Operator	ONNX Importer Custom Layer
Clip	nnet.onnx.layer.ClipLayer
Div	nnet.onnx.layer.ElementwiseAffineLayer
Flatten	nnet.onnx.layer.FlattenLayer or nnet.onnx.layer.Flatten3dLayer
Identity	nnet.onnx.layer.IdentityLayer
ImageScaler	nnet.onnx.layer.ElementwiseAffineLayer
PRelu	nnet.onnx.layer.PReluLayer
Reshape	nnet.onnx.layer.FlattenLayer
Sub	nnet.onnx.layer.ElementwiseAffineLayer

ONNX Operator	Image Processing Toolbox
DepthToSpace	depthToSpace2dLayer
Resize	resize2dLayer or resize3dLayer
SpaceToDepth	spaceToDepthLayer
Upsample	resize2dLayer or resize3dLayer

Conversion of ONNX Input and Output Tensors into Built-In MATLAB Layers

importONNXNetwork tries to interpret the data format of the ONNX network's input and output tensors, and then convert them into built-in MATLAB input and output layers. For details on the interpretation, see the tables Conversion of ONNX Input Tensors into Deep Learning Toolbox Layers and Conversion of ONNX Output Tensors into MATLAB Layers.

In Deep Learning Toolbox, each data format character must be one of these labels:

- S — Spatial
- C — Channel

- `B` — Batch observations
- `T` — Time or sequence
- `U` — Unspecified

Conversion of ONNX Input Tensors into Deep Learning Toolbox Layers

Data Formats		Data Interpretation		Deep Learning Toolbox Layer
ONNX Input Tensor	**MATLAB Input Format**	**Shape**	**Type**	
BC	CB	*c*-by-*n* array, where *c* is the number of features and *n* is the number of observations	Features	`featureInputLayer`
BCSS, BSSC, CSS, SSC	SSCB	*h*-by-*w*-by-*c*-by-*n* numeric array, where *h*, *w*, *c* and *n* are the height, width, number of channels of the images, and number of observations, respectively	2-D image	`imageInputLayer`
BCSSS, BSSSC, CSSS, SSSC	SSSCB	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, *c* and *n* are the height, width, depth, number of channels of the images, and number of image observations, respectively	3-D image	`image3dInputLayer`
TBC	CBT	*c*-by-*s*-by-*n* matrix, where *c* is the number of features of the sequence, *s* is the sequence length, and *n* is the number of sequence observations	Vector sequence	`sequenceInputLayer`

Data Formats		Data Interpretation		Deep Learning Toolbox Layer
ONNX Input Tensor	**MATLAB Input Format**	**Shape**	**Type**	
TBCSS	SSCBT	*h*-by-*w*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, *c* and *n* correspond to the height, width, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations	2-D image sequence	`sequenceInputLayer`
TBCSSS	SSSCBT	*h*-by-*w*-by-*d*-by-*c*-by-*s*-by-*n* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, *s* is the sequence length, and *n* is the number of image sequence observations	3-D image sequence	`sequenceInputLayer`

Conversion of ONNX Output Tensors into MATLAB Layers

Data Formats		MATLAB Layer
ONNX Output Tensor	**MATLAB Output Format**	
BC, TBC	CB, CBT	`classificationLayer`
BCSS, BSSC, CSS, SSC, BCSSS, BSSSC, CSSS, SSSC	SSCB, SSSCB	`pixelClassificationLayer`
TBCSS, TBCSSS	SSCBT, SSSCBT	`regressionLayer`

Use Imported Network on GPU

`importONNXNetwork` does not execute on a GPU. However, `importONNXNetwork` imports a pretrained neural network for deep learning as a `DAGNetwork` or `dlnetwork` object, which you can use on a GPU.

- If you import the network as a `DAGNetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `classify`. Specify the hardware requirements using

the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function for `DAGNetwork` objects.

- If you import the network as a `DAGNetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `predict`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. If the network has multiple outputs, specify the name-value argument `ReturnCategorical` as `true`.
- If you import the network as a `dlnetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.

  - If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
  - Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

    ```
    dlnet = dlupdate(@gpuarray,dlnet)
    ```
- You can train the imported network on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.

Alternative Functionality

Deep Learning Toolbox Converter for ONNX Model Format provides three functions to import a pretrained ONNX network: `importONNXNetwork`, `importONNXLayers`, and `importONNXFunction`.

If the imported network contains an ONNX operator not supported for conversion into a built-in MATLAB layer (see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers” on page 1-860) and `importONNXNetwork` does not generate a custom layer, then `importONNXNetwork` returns an error. In this case, you can still use `importONNXLayers` to import the network architecture and weights or `importONNXFunction` to import the network as an `ONNXParameters` object and a model function.

For more information on which import function best suits different scenarios, see “Select Function to Import ONNX Pretrained Network”.

Compatibility Considerations

`ClassNames` option has been removed
Errors starting in R2021b

`ClassNames` has been removed. Use `Classes` instead. To update your code, replace all instances of `ClassNames` with `Classes`.

`importONNXNetwork` cannot create input and output layers from ONNX file information
Behavior changed in R2021b

If you import an ONNX model as a `DAGNetwork` object, the imported network must include input and output layers. `importONNXNetwork` tries to convert the input and output ONNX tensors into built-in MATLAB layers. When importing some networks, which `importONNXNetwork` could previously import with input and output built-in MATLAB layers, `importONNXNetwork` might now return an error. In this case, do one of the following to update your code:

- Specify the name-value argument `TargetNetwork` as `"dlnetwork"` to import the network as a `dlnetwork` object.
- Use the name-value arguments `InputDataFormats`, `OutputDataFormats`, and `OutputLayerType` to specify the imported network's inputs and outputs.
- Use `importONNXLayers` to import the network as a layer graph with placeholder layers.
- Use `importONNXFunction` to import the network as a model function and an `ONNXParameters` object.

References

[1] *Open Neural Network Exchange*. https://github.com/onnx/.

[2] *ONNX*. https://onnx.ai/.

See Also

`importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `importONNXLayers` | `exportONNXNetwork` | `importONNXFunction` | `importTensorFlowNetwork` | `importTensorFlowLayers`

Topics
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Select Function to Import ONNX Pretrained Network”
“Deploy Imported Network with MATLAB Compiler”
“Multiple-Input and Multiple-Output Networks”

Introduced in R2018a

importTensorFlowLayers

Import layers from TensorFlow network

Syntax

```
lgraph = importTensorFlowLayers(modelFolder)
lgraph = importTensorFlowLayers(modelFolder,Name,Value)
```

Description

`lgraph = importTensorFlowLayers(modelFolder)` returns the layers of a TensorFlow network from the folder `modelFolder`, which contains the model in the saved model format (compatible only with TensorFlow 2). The function can import TensorFlow networks created with the TensorFlow-Keras sequential or functional API. `importTensorFlowLayers` imports the layers defined in the `saved_model.pb` file and the learned weights contained in the `variables` subfolder, and returns `lgraph` as a `LayerGraph` object.

`importTensorFlowLayers` requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, then `importTensorFlowLayers` provides a download link.

Note `importTensorFlowLayers` tries to generate a custom layer when you import a custom TensorFlow layer or when the software cannot convert a TensorFlow layer into an equivalent built-in MATLAB layer. For a list of layers for which the software supports conversion, see “TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers” on page 1-875.

`importTensorFlowLayers` saves the generated custom layers and the associated TensorFlow operators in the package `+modelFolder`.

`importTensorFlowLayers` does not automatically generate a custom layer for each TensorFlow layer that is not supported for conversion into a built-in MATLAB layer. For more information on how to handle unsupported layers, see “Tips” on page 1-879.

`lgraph = importTensorFlowLayers(modelFolder,Name,Value)` imports the layers and weights from a TensorFlow network with additional options specified by one or more name-value arguments. For example, `'OutputLayerType','classification'` appends a classification output layer to the end of the imported network architecture.

Examples

Import TensorFlow Network as Layer Graph Compatible with `DAGNetwork`

Import a pretrained TensorFlow network in the saved model format as a `LayerGraph` object. Then, assemble the imported layers into a `DAGNetwork` object, and use the assembled network to classify an image.

Specify the model folder.

```
if ~exist('digitsDAGnet','dir')
    unzip('digitsDAGnet.zip')
end
modelFolder = './digitsDAGnet';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import the layers and weights of a TensorFlow network in the saved model format. By default, `importTensorFlowLayers` imports the network as a `LayerGraph` object compatible with a `DAGNetwork` object. Specify the output layer type for an image classification problem.

```
lgraph = importTensorFlowLayers(modelFolder,'OutputLayerType','classification')
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation.

lgraph =
  LayerGraph with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Display the last layer of the imported network. The output shows that `importTensorFlowLayers` appends a `ClassificationOutputLayer` to the end of the network architecture.

```
lgraph.Layers(end)
```

```
ans =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The classification layer does not contain the classes, so you must specify these before assembling the network. 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.

The classification layer has the name `'ClassificationLayer_activation_1'`. Set the classes to `classNames` and then replace the imported classification layer with the new one.

```
cLayer = lgraph.Layers(end);
cLayer.Classes = classNames;
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble the layer graph using `assembleNetwork` to return a `DAGNetwork` object.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Read the image you want to classify.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
```

Classify the image using the imported network.

```
label = classify(net,I);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' classNames{label}])
```

Import TensorFlow Network as Layer Graph Compatible with `dlnetwork`

Import a pretrained TensorFlow network in the saved model format as a `LayerGraph` object compatible with a `dlnetwork` object. Then, convert the `LayerGraph` object to a `dlnetwork` to classify an image.

Specify the model folder.

```
if ~exist('digitsDAGnet','dir')
    unzip('digitsDAGnet.zip')
end
modelFolder = './digitsDAGnet';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import the TensorFlow network as layers compatible with a `dlnetwork` object.

```
lgraph = importTensorFlowLayers(modelFolder,'TargetNetwork','dlnetwork')
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation.

lgraph =
  LayerGraph with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [12×2 table]
     InputNames: {'input_1'}
    OutputNames: {1×0 cell}
```

Read the image you want to classify and display the size of the image. The image is a grayscale (one-channel) image with size 28-by-28 pixels.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
size(I)
```

```
ans = 1×2

    28    28
```

Convert the imported layer graph to a `dlnetwork` object.

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

Display the input size of the network. In this case, the image size matches the network input size. If they do not match, you must resize the image by using `imresize(I, netInputSize(1:2))`.

```
dlnet.Layers(1).InputSize
```

```
ans = 1×3

    28    28     1
```

Convert the image to a `dlarray`. Format the images with the dimensions `'SSCB'` (spatial, spatial, channel, batch). In this example, the batch size is 1 and you can omit it (`'SSC'`).

```
I_dlarray = dlarray(single(I),'SSCB');
```

Classify the sample image and find the predicted label.

```
prob = predict(dlnet,I_dlarray);
[~,label] = max(prob);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' classNames{label}])
```

Import TensorFlow Network as Layer Graph with Autogenerated Custom Layers

Import a pretrained TensorFlow network in the saved model format as a `LayerGraph` object. Then, assemble the imported layers into a `DAGNetwork` object. The imported network contains layers that are not supported for conversion into built-in MATLAB layers. The software automatically generates custom layers when you import these layers.

This example uses the helper function `findCustomLayers`. To view the code for this function, see Helper Function on page 1-0 .

Specify the model folder.

```
if ~exist('digitsDAGnetwithnoise','dir')
    unzip('digitsDAGnetwithnoise.zip')
end
modelFolder = './digitsDAGnetwithnoise';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import the layers and weights of a TensorFlow network in the saved model format. By default, `importTensorFlowLayers` imports the network as a `LayerGraph` object compatible with a `DAGNetwork` object. Specify the output layer type for an image classification problem.

```
lgraph = importTensorFlowLayers(modelFolder,'OutputLayerType','classification');
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation.
```

If the imported network contains layers not supported for conversion into built-in MATLAB layers, then `importTensorFlowLayers` can automatically generate custom layers in place of these layers. `importTensorFlowLayers` saves each generated custom layer to a separate `.m` file in the package `+digitsDAGnetwithnoise` in the current folder.

Find the indices of the automatically generated custom layers, using the helper function `findCustomLayers`, and display the custom layers.

```
ind = findCustomLayers(lgraph.Layers,'+digitsDAGnetwithnoise');
lgraph.Layers(ind)
```

```
ans =
  2×1 Layer array with layers:
```

```
     1   'gaussian_noise_1'   GaussianNoise   digitsDAGnetwithnoise.kGaussianNoise1Layer3766
     2   'gaussian_noise_2'   GaussianNoise   digitsDAGnetwithnoise.kGaussianNoise2Layer3791
```

The classification layer does not contain the classes, so you must specify these before assembling the network. 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.

The classification layer has the name `'ClassificationLayer_activation_1'`. Set the classes to `classNames` and then replace the imported classification layer with the new one.

```
cLayer = lgraph.Layers(end);
cLayer.Classes = classNames;
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

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'}
```

Helper Function

This section provides the code of the helper function `findCustomLayers` used in this example. `findCustomLayers` returns the `indices` of the custom layers that `importTensorFlowNetwork` automatically generates.

```
function indices = findCustomLayers(layers,PackageName)

s = what(['.\' PackageName]);

indices = zeros(1,length(s.m));
for i = 1:length(layers)
    for j = 1:length(s.m)
        if strcmpi(class(layers(i)),[PackageName(2:end) '.' s.m{j}(1:end-2)])
            indices(j) = i;
        end
    end
end

end
```

Input Arguments

`modelFolder` — Name of TensorFlow model folder

character vector | string scalar

Name of the folder containing the TensorFlow model, specified as a character vector or string scalar. `modelFolder` must be in the current folder, or you must include a full or relative path to the folder.

`modelFolder` must contain the file `saved_model.pb`, and the subfolder `variables`. It can also contain the subfolders `assets` and `assets.extra`.

- The file `saved_model.pb` contains the layer graph architecture and training options (for example, optimizer, losses, and metrics).
- The subfolder `variables` contains the weights learned by the pretrained TensorFlow network. By default, `importTensorFlowLayers` imports the weights.
- The subfolder `assets` contains supplementary files (for example, vocabularies), which the layer graph can use. `importTensorFlowLayers` does not import the files in `assets`.
- The subfolder `assets.extra` contains supplementary files (for example, information for users), which coexist with the layer graph.

Example: `'MobileNet'`

Example: `'./MobileNet'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importTensorFlowLayers(modelFolder,'TargetNetwork','dagnetwork','OutputLayerType','classification')` imports the network layers and weights from `modelFolder`, saves the automatically generated custom layers in the package `+modelFolder` in the current folder, specifies that the imported layers are compatible with a `DAGNetwork` object, and appends a classification output layer to the end of the imported layers.

`PackageName` — Name of custom layers package

character vector | string scalar

Name of the package in which `importTensorFlowLayers` saves custom layers, specified as a character vector or string scalar. `importTensorFlowLayers` saves the custom layers package `+PackageName` in the current folder. If you do not specify `'PackageName'`, then `importTensorFlowLayers` saves the custom layers in a package named `+modelFolder` in the current folder. For more information on packages, see “Packages Create Namespaces”.

`importTensorFlowLayers` tries to generate a custom layer when you import a custom TensorFlow layer or when the software cannot convert a TensorFlow layer into an equivalent built-in MATLAB layer. `importTensorFlowLayers` saves each generated custom layer to a separate `.m` file in `+PackageName`. To view or edit a custom layer, open the associated `.m` file. For more information on custom layers, see “Deep Learning Custom Layers”.

The package `+PackageName` can also contain the subpackage `+ops`. This subpackage contains MATLAB functions corresponding to TensorFlow operators (see “Supported TensorFlow Operators” on page 1-877) that are used in the automatically generated custom layers. `importTensorFlowLayers` saves the associated MATLAB function for each operator in a separate `.m` file in the subpackage `+ops`. The object functions of `dlnetwork`, such as the `predict` function, use these operators when interacting with the custom layers.

Example: `'PackageName','MobileNet'`

Example: `'PackageName','CustomLayers'`

`TargetNetwork` — Target type of Deep Learning Toolbox network
`'dagnetwork'` (default) | `'dlnetwork'`

Target type of Deep Learning Toolbox network for imported network architecture, specified as `'dagnetwork'` or `'dlnetwork'`.

- If you specify `'TargetNetwork'` as `'dagnetwork'`, the imported network architecture is compatible with a `DAGNetwork` object. In this case, the imported `lgraph` must include an output layer specified by the TensorFlow saved model loss function or the name-value argument `'OutputLayerType'`.
- If you specify `'TargetNetwork'` as `'dlnetwork'`, the imported network architecture is compatible with a `dlnetwork` object. In this case, the imported `lgraph` does not include an output layer.

Example: `'TargetNetwork','dlnetwork'` imports a `LayerGraph` object compatible with a `dlnetwork` object.

`OutputLayerType` — Type of output layer
`'classification'` | `'regression'` | `'pixelclassification'`

Type of output layer that `importTensorFlowLayers` appends to the end of the imported network architecture, specified as `'classification'`, `'regression'`, or `'pixelclassification'`. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox.

- If you specify `'TargetNetwork'` as `'dagnetwork'` and the saved model in `modelFolder` does not specify a loss function, you must assign a value to the name-value argument `'OutputLayerType'`. A `DAGNetwork` object must have an output layer.
- If you specify `'TargetNetwork'` as `'dlnetwork'`, `importTensorFlowLayers` ignores the name-value argument `'OutputLayerType'`. A `dlnetwork` object does not have an output layer.

Example: `'OutputLayerType','classification'`

`ImageInputSize` — Size of input images
vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to `[height,width]` for grayscale images and `[height,width,channels]` for color images, respectively. The network uses this information when the `saved_model.pb` file in `modelFolder` does not specify the input size.

Example: `'ImageInputSize',[28 28]`

`Verbose` — Indicator to display import progress information
`true` or `1` (default) | `false` or `0`

Indicator to display import progress information in the command window, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `'Verbose','true'`

Output Arguments

`lgraph` — Network architecture
`LayerGraph` object

Network architecture, returned as a `LayerGraph` object.

To use the imported layer graph for prediction, you must convert the `LayerGraph` object to a `DAGNetwork` or `dlnetwork` object. Specify the name-value argument `'TargetNetwork'` as `'dagnetwork'` or `'dlnetwork'` depending on the intended workflow.

- Convert a `LayerGraph` to a `DAGNetwork` by using `assembleNetwork`. On the `DAGNetwork` object, you then predict class labels using the `classify` function.
- Convert a `LayerGraph` to a `dlnetwork` by using `dlnetwork`. On the `dlnetwork` object, you then predict class labels using the `predict` function. Specify the input data as a `dlarray` using the correct data format (for more information, see the `fmt` argument of `dlarray`).

Limitations

- `importTensorFlowLayers` supports TensorFlow versions v2.0, v2.1, v.2.2, and v2.3.

More About

TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers

`importTensorFlowLayers` supports the following TensorFlow-Keras layer types for conversion into built-in MATLAB layers, with some limitations.

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`Add`	`additionLayer`
`Activation`, with activation names: • `'elu'` • `'relu'` • `'linear'` • `'softmax'` • `'sigmoid'` • `'swish'` • `'tanh'`	Layers: • `eluLayer` • `reluLayer` or `clippedReluLayer` • None • `softmaxLayer` • `sigmoidLayer` • `swishLayer` • `tanhLayer`
Advanced activations: • `ELU` • `Softmax` • `ReLU` • `LeakyReLU` • `PReLu*`	Layers: • `eluLayer` • `softmaxLayer` • `reluLayer`, `clippedReluLayer`, or `leakyReluLayer` • `leakyReluLayer` • `nnet.keras.layer.PreluLayer`
`AveragePooling1D`	`averagePooling1dLayer` with `PaddingValue` specified as `'mean'`
`AveragePooling2D`	`averagePooling2dLayer` with `PaddingValue` specified as `'mean'`

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`BatchNormalization`	`batchNormalizationLayer`
`Bidirectional(LSTM(__))`	`bilstmLayer`
`Concatenate`	`depthConcatenationLayer`
`Conv1D`	`convolution1dLayer`
`Conv2D`	`convolution2dLayer`
`Conv2DTranspose`	`transposedConv2dLayer`
`CuDNNGRU`	`gruLayer`
`CuDNNLSTM`	`lstmLayer`
`Dense`	`fullyConnectedLayer`
`DepthwiseConv2D`	`groupedConvolution2dLayer`
`Dropout`	`dropoutLayer`
`Embedding`	`wordEmbeddingLayer`
`Flatten`	`nnet.keras.layer.FlattenCStyleLayer`
`GlobalAveragePooling1D`	`globalAveragePooling1dLayer`
`GlobalAveragePooling2D`	`globalAveragePooling2dLayer`
`GlobalMaxPool1D`	`globalMaxPooling1dLayer`
`GlobalMaxPool2D`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`Input`	`imageInputLayer`, `sequenceInputLayer`, or `featureInputLayer`
`LSTM`	`lstmLayer`
`MaxPool1D`	`maxPooling1dLayer`
`MaxPool2D`	`maxPooling2dLayer`
`Multiply`	`multiplicationLayer`
`SeparableConv2D`	`groupedConvolution2dLayer` or `convolution2dLayer`
`TimeDistributed`	`sequenceFoldingLayer` before the wrapped layer, and `sequenceUnfoldingLayer` after the wrapped layer
`UpSampling2D`	`resize2dLayer`
`UpSampling3D`	`resize3dLayer`
`ZeroPadding1D`	`nnet.keras.layer.ZeroPadding1DLayer`
`ZeroPadding2D`	`nnet.keras.layer.ZeroPadding2DLayer`

* For a PReLU layer, `importTensorFlowLayers` replaces a vector-valued scaling parameter with the average of the vector elements. You can change the parameter back to a vector after import. For an example, see “Import Keras PReLU Layer” on page 1-785.

Supported TensorFlow-Keras Loss Functions

`importTensorFlowLayers` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`
- `binary_crossentropy`

Supported TensorFlow Operators

`importTensorFlowLayers` supports the following TensorFlow operators for conversion into MATLAB functions with `dlarray` support.

TensorFlow Operator	Corresponding MATLAB Function
`Add`	`tfAdd`
`AddN`	`tfAddN`
`AddV2`	`tfAdd`
`AvgPool`	`tfAvgPool`
`BatchMatMulV2`	`tfBatchMatMulV2`
`BiasAdd`	`tfBiasAdd`
`BroadcastTo`	`tfBroadcastTo`
`Cast`	`tfCast`
`ConcatV2`	`tfCat`
`Const`	None (translated to weights in custom layer)
`Conv2D`	`tfConv2D`
`DepthToSpace`	`depthToSpace`
`DepthwiseConv2dNative`	`tfDepthwiseConv2D`
`Exp`	`exp`
`FusedBatchNormV3`	`tfBatchnorm`
`GatherV2`	`tfGather`
`Identity`	None (translated to value assignment in custom layer)
`IdentityN`	`tfIdentityN`
`L2Loss`	`tfL2Loss`
`LeakyRelu`	`leakyrelu`
`Less`	`lt`, `<`
`Log`	`log`
`MatMul`	`tfMatMul`
`MaxPool`	`tfMaxPool`
`Maximum`	`tfMaximum`
`Mean`	`tfMean`

TensorFlow Operator	Corresponding MATLAB Function
`Minimum`	`tfMinimum`
`MirrorPad`	`tfMirrorPad`
`Mul`	`tfMul`
`Neg`	`minus`, `-`
`Pack`	`tfStack`
`Pad`	`tfPad`
`PadV2`	`tfPad`
`PartitionedCall`	None (translated to function in custom layer methods)
`Pow`	`power`, `.^`
`Prod`	`tfProd`
`RandomStandardNormal`	`tfRandomStandardNormal`
`Range`	`tfRange`
`ReadVariableOp`	None (translated to value assignment in custom layer)
`RealDiv`	`tfDiv`
`Relu`	`relu`
`Relu6`	`relu` and `min`
`Reshape`	`tfReshape`
`ResizeNearestNeighbor`	`dlresize`
`Rsqrt`	`sqrt`
`Shape`	`tfShape`
`Sigmoid`	`sigmoid`
`Softmax`	`softmax`
`SpaceToDepth`	`spaceToDepth`
`Square`	`.^2`
`Sqrt`	`sqrt`
`SquaredDifference`	`tfMul` or `tfSub`
`Squeeze`	`tfSqueeze`
`StatefulPartitionedCall`	None (translated to function in custom layer methods)
`StopGradient`	`tfStopGradient`
`StridedSlice`	`tfStridedSlice` or `tfSqueeze`
`Sub`	`tfSub`
`Tanh`	`tanh`
`Tile`	`tfTile`
`Transpose`	`permute`

For more information on functions that operate on `dlarray` objects, see “List of Functions with dlarray Support”.

Use Imported Network Layers on GPU

`importTensorFlowLayers` does not execute on a GPU. However, `importTensorFlowLayers` imports the layers of a pretrained neural network for deep learning as a `LayerGraph` object, which you can use on a GPU.

- Convert the imported `LayerGraph` object to a `DAGNetwork` object by using `assembleNetwork`. On the `DAGNetwork` object, you can then predict class labels on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function and specify the name-value argument `ReturnCategorical` as `true`.
- Convert the imported `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. On the `dlnetwork` object, you can then predict class labels on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.
  - If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
  - Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

    ```
    dlnet = dlupdate(@gpuarray,dlnet)
    ```
- You can train the imported `LayerGraph` object on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- If the imported network contains a layer not supported for conversion into a built-in MATLAB layer (see “TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers” on page 1-875) and `importTensorFlowLayers` does not automatically generate a custom layer, then `importTensorFlowLayers` inserts a placeholder layer in place of the unsupported layer. To find the names and indices of the unsupported layers in the network, use the `findPlaceholderLayers` function. You then can replace a placeholder layer with a new layer that you define. To replace a layer, use `replaceLayer`.
- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.
  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.

- The members of the package `+PackageName` (custom layers and TensorFlow operators) are not accessible if the package parent folder is not on the MATLAB path. For more information, see "Packages and the MATLAB Path".

Alternative Functionality

Use `importTensorFlowNetwork` or `importTensorFlowLayers` to import a TensorFlow network in the saved model format [2]. Alternatively, if the network is in HDF5 or JSON format, use `importKerasNetwork` or `importKerasLayers` to import the network.

References

[1] *TensorFlow*. https://www.tensorflow.org/.

[2] *Using the SavedModel format*. https://www.tensorflow.org/guide/saved_model.

See Also

`importKerasLayers` | `importKerasNetwork` | `importONNXNetwork` | `importONNXLayers` | `exportONNXNetwork` | `importCaffeLayers` | `importCaffeNetwork` | `importONNXFunction` | `layerGraph` | `assembleNetwork` | `findPlaceholderLayers` | `replaceLayer` | `importTensorFlowNetwork`

Topics
"Deep Learning in MATLAB"
"Pretrained Deep Neural Networks"
"Train Deep Learning Model in MATLAB"
"Assemble Network from Pretrained Keras Layers"
"Define Custom Deep Learning Layers"
"Make Predictions Using dlnetwork Object" on page 1-477

Introduced in R2021a

importTensorFlowNetwork

Import pretrained TensorFlow network

Syntax

```
net = importTensorFlowNetwork(modelFolder)
net = importTensorFlowNetwork(modelFolder,Name,Value)
```

Description

`net = importTensorFlowNetwork(modelFolder)` imports a pretrained TensorFlow network from the folder `modelFolder`, which contains the model in the saved model format (compatible only with TensorFlow 2). The function can import TensorFlow networks created with the TensorFlow-Keras sequential or functional API. `importTensorFlowNetwork` imports the layers defined in the `saved_model.pb` file and the learned weights contained in the `variables` subfolder, and returns the network `net` as a `DAGNetwork` or `dlnetwork` object.

`importTensorFlowNetwork` requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, then `importTensorFlowNetwork` provides a download link.

Note `importTensorFlowNetwork` tries to generate a custom layer when you import a custom TensorFlow layer or when the software cannot convert a TensorFlow layer into an equivalent built-in MATLAB layer. For a list of layers for which the software supports conversion, see “TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers” on page 1-891.

`importTensorFlowNetwork` saves the generated custom layers and the associated TensorFlow operators in the package `+modelFolder`.

`importTensorFlowNetwork` does not automatically generate a custom layer for each TensorFlow layer that is not supported for conversion into a built-in MATLAB layer. For more information on how to handle unsupported layers, see “Tips” on page 1-895.

`net = importTensorFlowNetwork(modelFolder,Name,Value)` imports the pretrained TensorFlow network with additional options specified by one or more name-value arguments. For example, `'OutputLayerType','classification'` imports the network as a `DAGNetwork` with a classification output layer appended to the end of the imported network architecture.

Examples

Import TensorFlow Network as `DAGNetwork` to Classify Image

Import a pretrained TensorFlow network in the saved model format as a `DAGNetwork` object, and use the imported network to classify an image.

Specify the model folder.

```
if ~exist('digitsDAGnet','dir')
    unzip('digitsDAGnet.zip')
end
modelFolder = './digitsDAGnet';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

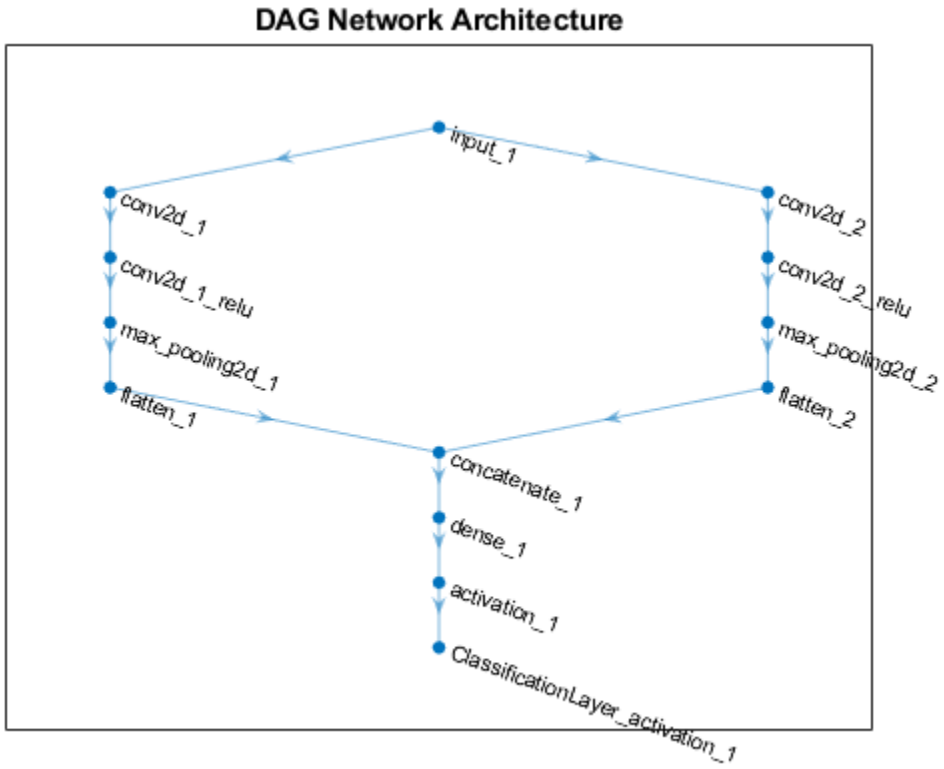
Import a TensorFlow network in the saved model format. By default, `importTensorFlowNetwork` imports the network as a `DAGNetwork` object. Specify the output layer type for an image classification problem.

```
net = importTensorFlowNetwork(modelFolder,'OutputLayerType','classification','Classes',classNames
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation. Assembling network...
Import finished.

net =
  DAGNetwork with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Plot the network architecture.

```
plot(net)
title('DAG Network Architecture')
```



Read the image you want to classify and display the size of the image. The image is a grayscale (one-channel) image with size 28-by-28 pixels.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
size(I)
```

```
ans = 1×2

    28    28
```

Display the input size of the network. In this case, the image size matches the network input size. If they do not match, you must resize the image by using `imresize(I, netInputSize(1:2))`.

```
net.Layers(1).InputSize
```

```
ans = 1×3

    28    28     1
```

Classify the image using the pretrained network.

```
label = classify(net,I);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' char(label)])
```

Import TensorFlow Network as `dlnetwork` to Classify Image

Import a pretrained TensorFlow Network in the saved model format as a `dlnetwork` object, and use the imported network to predict class labels.

Specify the model folder.

```
if ~exist('digitsDAGnet','dir')
    unzip('digitsDAGnet.zip')
end
modelFolder = './digitsDAGnet';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import a TensorFlow network in the saved model format as a `dlnetwork` object.

```
net = importTensorFlowNetwork(modelFolder,'TargetNetwork','dlnetwork')
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation. Assembling network...
Import finished.

net =
  dlnetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [12×2 table]
     Learnables: [6×3 table]
          State: [0×3 table]
     InputNames: {'input_1'}
    OutputNames: {'activation_1'}
    Initialized: 1
```

Read the image you want to classify and display the size of the image. The image is a grayscale (one-channel) image with size 28-by-28 pixels.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
size(I)
```

```
ans = 1×2

    28    28
```

Display the input size of the network. In this case, the image size matches the network input size. If they do not match, you must resize the image by using `imresize(I, netInputSize(1:2))`.

```
netInputSize = net.Layers(1).InputSize
```

```
netInputSize = 1×3

    28    28     1
```

Convert the image to a `dlarray`. Format the images with the dimensions `'SSCB'` (spatial, spatial, channel, batch). In this case, the batch size is 1 and you can omit it (`'SSC'`).

```
I_dlarray = dlarray(single(I),'SSCB');
```

Classify the sample image and find the predicted label.

```
prob = predict(net,I_dlarray);
[~,label] = max(prob);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' classNames{label}])
```

Import TensorFlow Network with Autogenerated Custom Layers

Import a pretrained TensorFlow network in the saved model format as a `DAGNetwork` object, and use the imported network to classify an image. The imported network contains layers that are not supported for conversion into built-in MATLAB layers. The software automatically generates custom layers when you import these layers.

This example uses the helper function `findCustomLayers`. To view the code for this function, see Helper Function on page 1-0 .

Specify the model folder.

```
if ~exist('digitsDAGnetwithnoise','dir')
    unzip('digitsDAGnetwithnoise.zip')
end
modelFolder = './digitsDAGnetwithnoise';
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

Import a TensorFlow network in the saved model format. By default, `importTensorFlowNetwork` imports the network as a `DAGNetwork` object. Specify the output layer type for an image classification problem.

```
net = importTensorFlowNetwork(modelFolder,'OutputLayerType','classification','Classes',classNames
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation. Assembling network...
Import finished.
```

If the imported network contains layers not supported for conversion into built-in MATLAB layers, then `importTensorFlowNetwork` can automatically generate custom layers in place of these layers. `importTensorFlowNetwork` saves each generated custom layer to a separate `.m` file in the package `+digitsDAGnetwithnoise` in the current folder.

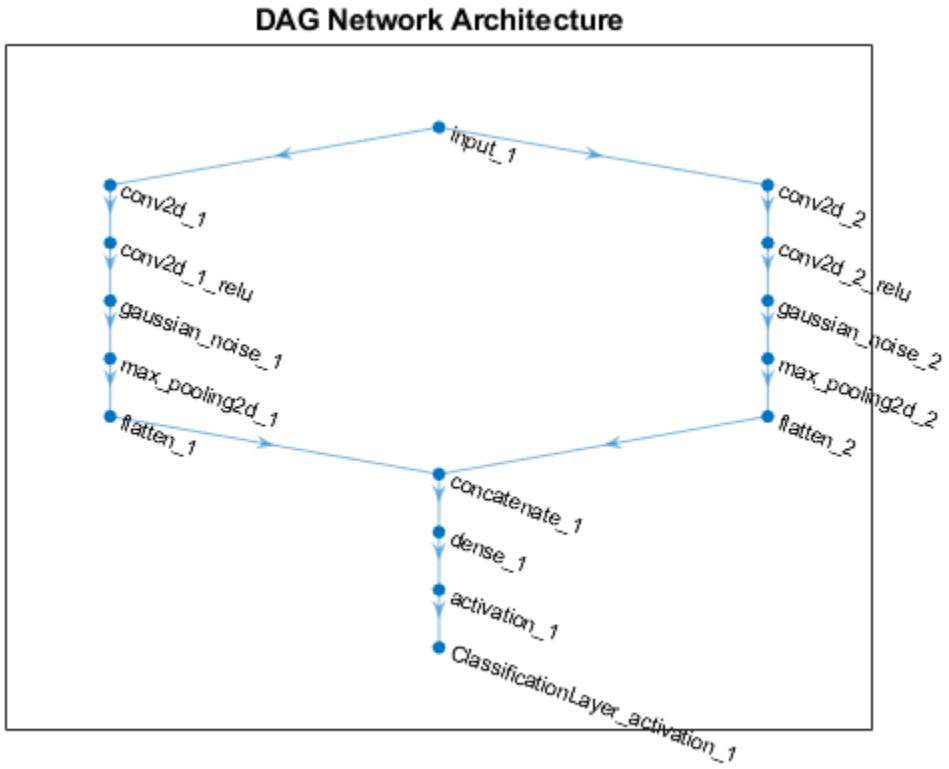
Find the indices of the automatically generated custom layers using the helper function `findCustomLayers`, and display the custom layers.

```
ind = findCustomLayers(net.Layers,'+digitsDAGnetwithnoise');
net.Layers(ind)
```

```
ans =
  2×1 Layer array with layers:

     1   'gaussian_noise_1'   GaussianNoise   digitsDAGnetwithnoise.kGaussianNoise1Layer3766
     2   'gaussian_noise_2'   GaussianNoise   digitsDAGnetwithnoise.kGaussianNoise2Layer3791
```

Plot the network architecture.

```
plot(net)
title('DAG Network Architecture')
```



Read the image you want to classify.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
```

Classify the image using the pretrained network.

```
label = classify(net,I);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' char(label)])
```

Helper Function

This section provides the code of the helper function `findCustomLayers` used in this example. `findCustomLayers` returns the `indices` of the custom layers that `importTensorFlowNetwork` automatically generates.

```
function indices = findCustomLayers(layers,PackageName)

s = what(['.\' PackageName]);

indices = zeros(1,length(s.m));
for i = 1:length(layers)
    for j = 1:length(s.m)
        if strcmpi(class(layers(i)),[PackageName(2:end) '.' s.m{j}(1:end-2)])
            indices(j) = i;
        end
    end
end

end
```

Input Arguments

`modelFolder` — Name of TensorFlow model folder

character vector | string scalar

Name of the folder containing the TensorFlow model, specified as a character vector or string scalar. `modelFolder` must be in the current folder, or you must include a full or relative path to the folder. `modelFolder` must contain the file `saved_model.pb`, and the subfolder `variables`. It can also contain the subfolders `assets` and `assets.extra`.

- The file `saved_model.pb` contains the layer graph architecture and training options (for example, optimizer, losses, and metrics).
- The subfolder `variables` contains the weights learned by the pretrained TensorFlow network. By default, `importTensorFlowNetwork` imports the weights.
- The subfolder `assets` contains supplementary files (for example, vocabularies), which the layer graph can use. `importTensorFlowNetwork` does not import the files in `assets`.
- The subfolder `assets.extra` contains supplementary files (for example, information for users), which coexist with the layer graph.

Example: `'MobileNet'`

Example: `'./MobileNet'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importTensorFlowNetwork(modelFolder,'TargetNetwork','dagnetwork','OutputLayerType','classification')` imports a network from `modelFolder` as a `DAGNetwork` object, saves the automatically generated custom layers in the package `+modelFolder` in the current folder, and appends a classification output layer to the end of the imported network architecture.

PackageName — Name of custom layers package
character vector | string scalar

Name of the package in which `importTensorFlowNetwork` saves custom layers, specified as a character vector or string scalar. `importTensorFlowNetwork` saves the custom layers package `+PackageName` in the current folder. If you do not specify `'PackageName'`, then `importTensorFlowNetwork` saves the custom layers in a package named `+modelFolder` in the current folder. For more information on packages, see "Packages Create Namespaces".

`importTensorFlowNetwork` tries to generate a custom layer when you import a custom TensorFlow layer or when the software cannot convert a TensorFlow layer into an equivalent built-in MATLAB layer. `importTensorFlowNetwork` saves each generated custom layer to a separate `.m` file in `+PackageName`. To view or edit a custom layer, open the associated `.m` file. For more information on custom layers, see "Deep Learning Custom Layers".

The package `+PackageName` can also contain the subpackage `+ops`. This subpackage contains MATLAB functions corresponding to TensorFlow operators (see "Supported TensorFlow Operators" on page 1-892) that are used in the automatically generated custom layers. `importTensorFlowNetwork` saves the associated MATLAB function for each operator in a separate `.m` file in the subpackage `+ops`. The object functions of `dlnetwork`, such as the `predict` function, use these operators when interacting with the custom layers.

Example: `'PackageName','MobileNet'`

Example: `'PackageName','CustomLayers'`

TargetNetwork — Target type of Deep Learning Toolbox network
`'dagnetwork'` (default) | `'dlnetwork'`

Target type of Deep Learning Toolbox network, specified as `'dagnetwork'` or `'dlnetwork'`.

- Specify `'TargetNetwork'` as `'dagnetwork'` to import the network as a `DAGNetwork` object. In this case, `net` must include an output layer specified by the TensorFlow saved model loss function or the name-value argument `'OutputLayerType'`.
- Specify `'TargetNetwork'` as `'dlnetwork'` to import the network as a `dlnetwork` object. In this case, `net` does not include an output layer.

Example: `'TargetNetwork','dlnetwork'`

OutputLayerType — Type of output layer
`'classification'` | `'regression'` | `'pixelclassification'`

Type of output layer that `importTensorFlowNetwork` appends to the end of the imported network architecture, specified as `'classification'`, `'regression'`, or `'pixelclassification'`. Appending a `pixelClassificationLayer` object requires Computer Vision Toolbox.

- If you specify `'TargetNetwork'` as `'dagnetwork'` and the saved model in `modelFolder` does not specify a loss function, you must assign a value to the name-value argument `'OutputLayerType'`. A `DAGNetwork` object must have an output layer.
- If you specify `'TargetNetwork'` as `'dlnetwork'`, `importTensorFlowNetwork` ignores the name-value argument `'OutputLayerType'`. A `dlnetwork` object does not have an output layer.

Example: `'OutputLayerType','classification'`

`ImageInputSize` — Size of input images
vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to `[height,width]` for grayscale images and `[height,width,channels]` for color images, respectively. The network uses this information when the `saved_model.pb` file in `modelFolder` does not specify the input size.

Example: `'ImageInputSize',[28 28]`

`Classes` — Classes of the output layer
'auto' (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. If you specify a string array or cell array of character vectors `str`, then `importTensorFlowNetwork` sets the classes of the output layer to `categorical(str,str)`. If `Classes` is `'auto'`, then `importTensorFlowNetwork` sets the classes to `categorical(1:N)`, where `N` is the number of classes.

- If you specify `'TargetNetwork'` as `'dagnetwork'`, `importTensorFlowNetwork` stores information on classes in the output layer of the `DAGNetwork` object.
- If you specify `'TargetNetwork'` as `'dlnetwork'`, `importTensorFlowNetwork` ignores the name-value argument `'Classes'`. A `dlnetwork` object does not have an output layer to store information on classes.

Example: `'Classes',{'0','1','3'}`

Example: `'Classes',categorical({'dog','cat'})`

Data Types: `char` | `categorical` | `string` | `cell`

`Verbose` — Indicator to display import progress information
`true` or `1` (default) | `false` or `0`

Indicator to display import progress information in the command window, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `'Verbose','true'`

Output Arguments

`net` — Pretrained TensorFlow network
`DAGNetwork` object | `dlnetwork` object

Pretrained TensorFlow network, returned as a `DAGNetwork` or `dlnetwork` object.

- Specify `'TargetNetwork` as `'dagnetwork'` to import the network as a `DAGNetwork` object. On the `DAGNetwork` object, you then predict class labels by using the `classify` function.
- Specify `'TargetNetwork` as `'dlnetwork'` to import the network as a `dlnetwork` object. On the `dlnetwork` object, you then predict class labels by using the `predict` function. Specify the input data as a `dlarray` using the correct data format (for more information, see the `fmt` argument of `dlarray`).

Limitations

- `importTensorFlowNetwork` supports TensorFlow versions v2.0, v2.1, v.2.2, and v2.3.

More About

TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers

`importTensorFlowNetwork` supports the following TensorFlow-Keras layer types for conversion into built-in MATLAB layers, with some limitations.

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`Add`	`additionLayer`
`Activation`, with activation names: • `'elu'` • `'relu'` • `'linear'` • `'softmax'` • `'sigmoid'` • `'swish'` • `'tanh'`	Layers: • `eluLayer` • `reluLayer` or `clippedReluLayer` • None • `softmaxLayer` • `sigmoidLayer` • `swishLayer` • `tanhLayer`
Advanced activations: • `ELU` • `Softmax` • `ReLU` • `LeakyReLU` • `PReLu*`	Layers: • `eluLayer` • `softmaxLayer` • `reluLayer`, `clippedReluLayer`, or `leakyReluLayer` • `leakyReluLayer` • `nnet.keras.layer.PreluLayer`
`AveragePooling1D`	`averagePooling1dLayer` with `PaddingValue` specified as `'mean'`
`AveragePooling2D`	`averagePooling2dLayer` with `PaddingValue` specified as `'mean'`
`BatchNormalization`	`batchNormalizationLayer`
`Bidirectional(LSTM(__))`	`bilstmLayer`
`Concatenate`	`depthConcatenationLayer`
`Conv1D`	`convolution1dLayer`
`Conv2D`	`convolution2dLayer`
`Conv2DTranspose`	`transposedConv2dLayer`
`CuDNNGRU`	`gruLayer`
`CuDNNLSTM`	`lstmLayer`
`Dense`	`fullyConnectedLayer`

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
`DepthwiseConv2D`	`groupedConvolution2dLayer`
`Dropout`	`dropoutLayer`
`Embedding`	`wordEmbeddingLayer`
`Flatten`	`nnet.keras.layer.FlattenCStyleLayer`
`GlobalAveragePooling1D`	`globalAveragePooling1dLayer`
`GlobalAveragePooling2D`	`globalAveragePooling2dLayer`
`GlobalMaxPool1D`	`globalMaxPooling1dLayer`
`GlobalMaxPool2D`	`globalMaxPooling2dLayer`
`GRU`	`gruLayer`
`Input`	`imageInputLayer`, `sequenceInputLayer`, or `featureInputLayer`
`LSTM`	`lstmLayer`
`MaxPool1D`	`maxPooling1dLayer`
`MaxPool2D`	`maxPooling2dLayer`
`Multiply`	`multiplicationLayer`
`SeparableConv2D`	`groupedConvolution2dLayer` or `convolution2dLayer`
`TimeDistributed`	`sequenceFoldingLayer` before the wrapped layer, and `sequenceUnfoldingLayer` after the wrapped layer
`UpSampling2D`	`resize2dLayer`
`UpSampling3D`	`resize3dLayer`
`ZeroPadding1D`	`nnet.keras.layer.ZeroPadding1DLayer`
`ZeroPadding2D`	`nnet.keras.layer.ZeroPadding2DLayer`

* For a PReLU layer, `importTensorFlowNetwork` replaces a vector-valued scaling parameter with the average of the vector elements. You can change the parameter back to a vector after import. For an example, see “Import Keras PReLU Layer” on page 1-785.

Supported TensorFlow-Keras Loss Functions

`importTensorFlowNetwork` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`
- `binary_crossentropy`

Supported TensorFlow Operators

`importTensorFlowNetwork` supports the following TensorFlow operators for conversion into MATLAB functions with `dlarray` support.

TensorFlow Operator	Corresponding MATLAB Function
`Add`	`tfAdd`
`AddN`	`tfAddN`
`AddV2`	`tfAdd`
`AvgPool`	`tfAvgPool`
`BatchMatMulV2`	`tfBatchMatMulV2`
`BiasAdd`	`tfBiasAdd`
`BroadcastTo`	`tfBroadcastTo`
`Cast`	`tfCast`
`ConcatV2`	`tfCat`
`Const`	None (translated to weights in custom layer)
`Conv2D`	`tfConv2D`
`DepthToSpace`	`depthToSpace`
`DepthwiseConv2dNative`	`tfDepthwiseConv2D`
`Exp`	`exp`
`FusedBatchNormV3`	`tfBatchnorm`
`GatherV2`	`tfGather`
`Identity`	None (translated to value assignment in custom layer)
`IdentityN`	`tfIdentityN`
`L2Loss`	`tfL2Loss`
`LeakyRelu`	`leakyrelu`
`Less`	`lt`, `<`
`Log`	`log`
`MatMul`	`tfMatMul`
`MaxPool`	`tfMaxPool`
`Maximum`	`tfMaximum`
`Mean`	`tfMean`
`Minimum`	`tfMinimum`
`MirrorPad`	`tfMirrorPad`
`Mul`	`tfMul`
`Neg`	`minus`, `-`
`Pack`	`tfStack`
`Pad`	`tfPad`
`PadV2`	`tfPad`
`PartitionedCall`	None (translated to function in custom layer methods)
`Pow`	`power`, `.^`

TensorFlow Operator	Corresponding MATLAB Function
`Prod`	`tfProd`
`RandomStandardNormal`	`tfRandomStandardNormal`
`Range`	`tfRange`
`ReadVariableOp`	None (translated to value assignment in custom layer)
`RealDiv`	`tfDiv`
`Relu`	`relu`
`Relu6`	`relu` and `min`
`Reshape`	`tfReshape`
`ResizeNearestNeighbor`	`dlresize`
`Rsqrt`	`sqrt`
`Shape`	`tfShape`
`Sigmoid`	`sigmoid`
`Softmax`	`softmax`
`SpaceToDepth`	`spaceToDepth`
`Square`	`.^2`
`Sqrt`	`sqrt`
`SquaredDifference`	`tfMul` or `tfSub`
`Squeeze`	`tfSqueeze`
`StatefulPartitionedCall`	None (translated to function in custom layer methods)
`StopGradient`	`tfStopGradient`
`StridedSlice`	`tfStridedSlice` or `tfSqueeze`
`Sub`	`tfSub`
`Tanh`	`tanh`
`Tile`	`tfTile`
`Transpose`	`permute`

For more information on functions that operate on `dlarray` objects, see “List of Functions with dlarray Support”.

Use Imported Network on GPU

`importTensorFlowNetwork` does not execute on a GPU. However, `importTensorFlowNetwork` imports a pretrained neural network for deep learning as a `DAGNetwork` or `dlnetwork` object, which you can use on a GPU.

- If you import the network as a `DAGNetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `classify`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For networks with multiple outputs, use the `predict` function for `DAGNetwork` objects.

- If you import the network as a `DAGNetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `predict`. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. If the network has multiple outputs, specify the name-value argument `ReturnCategorical` as `true`.
- If you import the network as a `dlnetwork` object, you can make predictions with the imported network on either a CPU or GPU by using `predict`. The function `predict` executes on the GPU if either the input data or network parameters are stored on the GPU.

  - If you use `minibatchqueue` to process and manage the mini-batches of input data, the `minibatchqueue` object converts the output to a GPU array by default if a GPU is available.
  - Use `dlupdate` to convert the learnable parameters of a `dlnetwork` object to GPU arrays.

    ```
    dlnet = dlupdate(@gpuarray,dlnet)
    ```
- You can train the imported network on either a CPU or GPU by using `trainNetwork`. To specify training options, including options for the execution environment, use the `trainingOptions` function. Specify the hardware requirements using the name-value argument `ExecutionEnvironment`. For more information on how to accelerate training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Tips

- If the imported network contains a layer not supported for conversion into a built-in MATLAB layer (see “TensorFlow-Keras Layers Supported for Conversion into Built-In MATLAB Layers” on page 1-891) and `importTensorFlowNetwork` does not generate a custom layer, then `importTensorFlowNetwork` returns an error. In this case, you can still use `importTensorFlowLayers` to import the network architecture.
- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. The most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

  - To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
  - To convert images from RGB to BGR format, use `flip`. For example, `flip(image,3)`.

  For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.
- The members of the package `+PackageName` (custom layers and TensorFlow operators) are not accessible if the package parent folder is not on the MATLAB path. For more information, see “Packages and the MATLAB Path”.

Alternative Functionality

Use `importTensorFlowNetwork` or `importTensorFlowLayers` to import a TensorFlow network in the saved model format [2]. Alternatively, if the network is in HDF5 or JSON format, use `importKerasNetwork` or `importKerasLayers` to import the network.

References

[1] *TensorFlow*. https://www.tensorflow.org/.

[2] *Using the SavedModel format*. https://www.tensorflow.org/guide/saved_model.

See Also

`importKerasLayers` | `importKerasNetwork` | `importONNXNetwork` | `importONNXLayers` | `exportONNXNetwork` | `importCaffeLayers` | `importCaffeNetwork` | `importONNXFunction` | `importTensorFlowLayers`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Train Deep Learning Model in MATLAB”
“Define Custom Deep Learning Layers”
“Make Predictions Using dlnetwork Object” on page 1-477

Introduced in R2021a

inceptionresnetv2

Pretrained Inception-ResNet-v2 convolutional neural network

Syntax

```
net = inceptionresnetv2
```

Description

Inception-ResNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 164 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the Inception-ResNet-v2 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Inception-ResNet-v2.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Inception-ResNet-v2 instead of GoogLeNet.

`net = inceptionresnetv2` returns a pretrained Inception-ResNet-v2 network.

This function requires the Deep Learning Toolbox Model *for Inception-ResNet-v2 Network* support package. If this support package is not installed, then the function provides a download link.

Examples

Load Inception-ResNet-v2 Network

Download and install the Deep Learning Toolbox Model *for Inception-ResNet-v2 Network* support package.

Type `inceptionresnetv2` at the command line.

```
inceptionresnetv2
```

If the Deep Learning Toolbox Model *for Inception-ResNet-v2 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `inceptionresnetv2` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
net = inceptionresnetv2
```

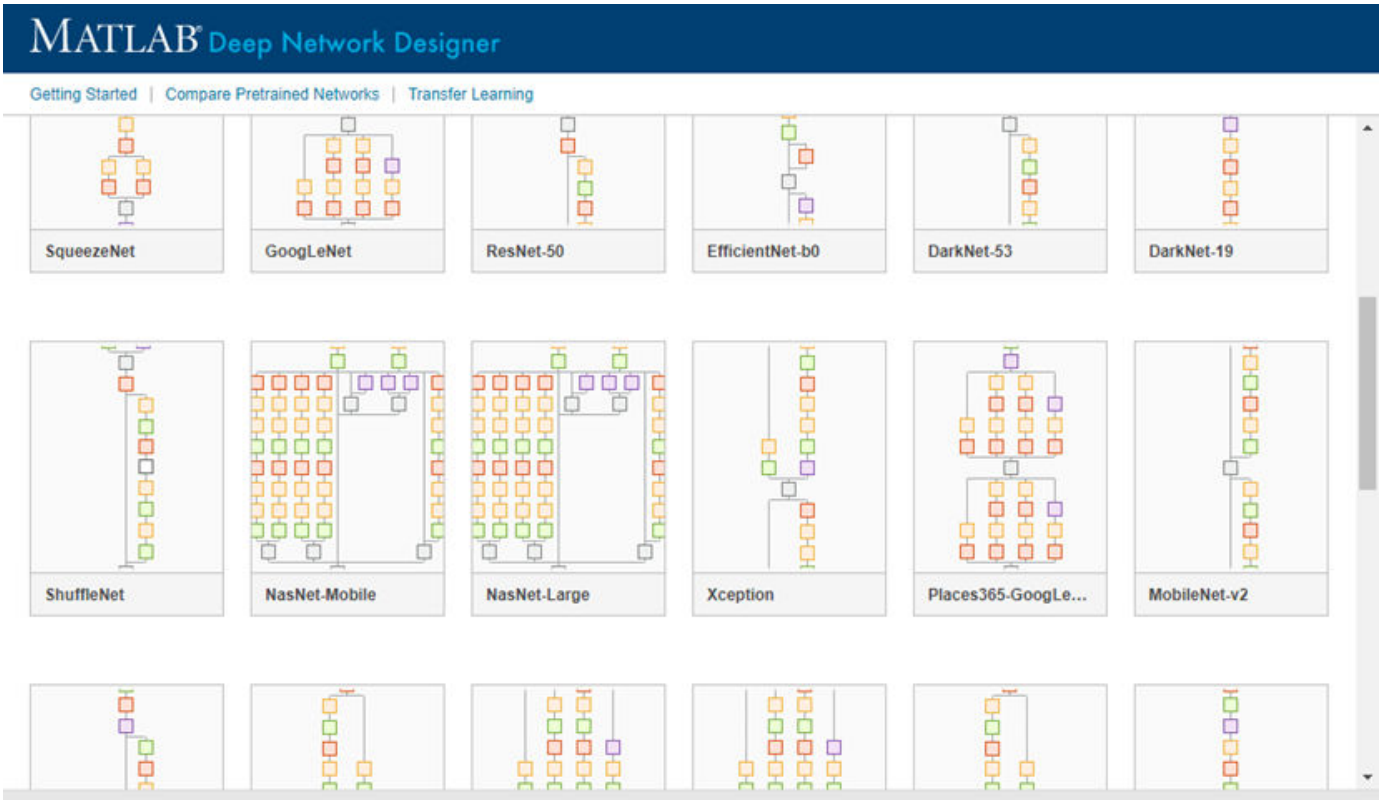
```
net =

  DAGNetwork with properties:
```

```
        Layers: [825×1 nnet.cnn.layer.Layer]
    Connections: [922×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(inceptionresnetv2)
```

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



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

Output Arguments

net — Pretrained Inception-ResNet-v2 convolutional neural network

`DAGNetwork` object

Pretrained Inception-ResNet-v2 convolutional neural network, returned as a `DAGNetwork` object.

References

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

[2] Szegedy, Christian, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning." In *AAAI*, vol. 4, p. 12. 2017.

[3] https://keras.io/api/applications/inceptionresnetv2/

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = inceptionresnetv2` or by passing the `inceptionresnetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionresnetv2')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = inceptionresnetv2` or by passing the `inceptionresnetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionresnetv2')`

For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `resnet18` | `resnet50` | `resnet101` | `inceptionv3` | `densenet201` | `squeezenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `importKerasLayers` | `importKerasNetwork`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b

inceptionv3

Inception-v3 convolutional neural network

Syntax

```
net = inceptionv3
net = inceptionv3('Weights','imagenet')

lgraph = inceptionv3('Weights','none')
```

Description

Inception-v3 is a convolutional neural network that is 48 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the Inception-v3 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Inception-v3.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Inception-v3 instead of GoogLeNet.

`net = inceptionv3` returns an Inception-v3 network trained on the ImageNet database.

This function 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('Weights','imagenet')` returns an Inception-v3 network trained on the ImageNet database. This syntax is equivalent to `net = inceptionv3`.

`lgraph = inceptionv3('Weights','none')` returns the untrained Inception-v3 network architecture. The untrained model does not require the support package.

Examples

Download Inception-v3 Support Package

Download and install the Deep Learning Toolbox Model *for Inception-v3 Network* support package.

Type `inceptionv3` at the command line.

```
inceptionv3
```

If the Deep Learning Toolbox Model *for Inception-v3 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `inceptionv3` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

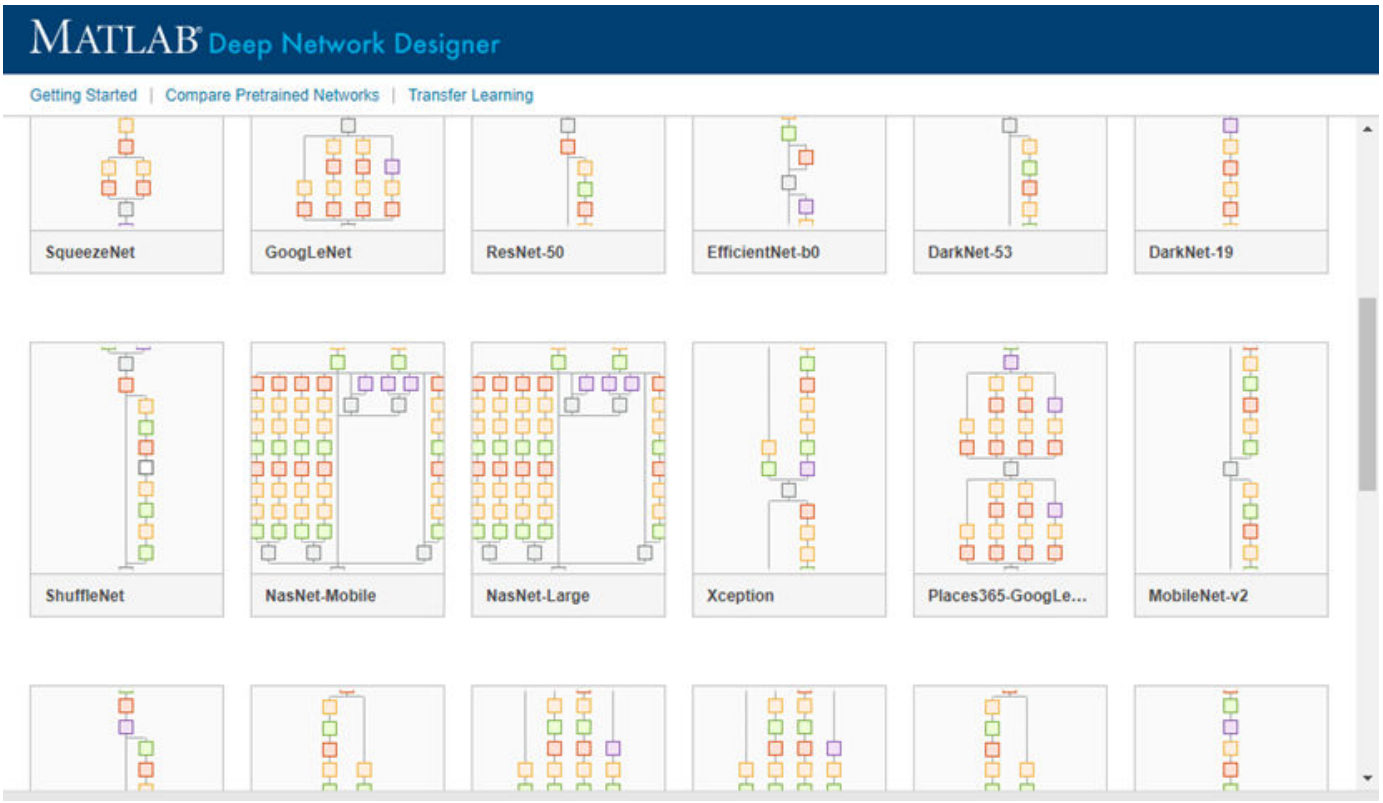
```
inceptionv3
```

```
ans =

  DAGNetwork with properties:

         Layers: [316×1 nnet.cnn.layer.Layer]
    Connections: [350×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(inceptionv3)
```

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



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

Output Arguments

net — Pretrained Inception-v3 convolutional neural network

`DAGNetwork` object

Pretrained Inception-v3 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained Inception-v3 convolutional neural network architecture
`LayerGraph` object

Untrained Inception-v3 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826. 2016.

[3] https://keras.io/api/applications/inceptionv3/

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = inceptionv3` or by passing the `inceptionv3` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionv3')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `inceptionv3('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = inceptionv3` or by passing the `inceptionv3` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionv3')`.

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `inceptionv3('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `resnet18` | `resnet50` | `trainNetwork` | `inceptionresnetv2` | `squeezenet` | `layerGraph` | `DAGNetwork` | `densenet201`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b

initialize

Initialize learnable and state parameters of a `dlnetwork`

Syntax

```
dlnet = initialize(dlnet)
dlnet = initialize(dlnet,dlX1,...,dlXn)
```

Description

Tip Most `dlnetwork` objects are initialized by default. You only need to manually initialize a `dlnetwork` if it is uninitialized. You can check if a network is initialized using the `Initialized` property of the `dlnetwork` object.

`dlnet = initialize(dlnet)` initializes any unset learnable parameters and state values of `dlnet` based on the input sizes defined by the network input layers. Any learnable or state parameters that already contain values remain unchanged.

A network with unset, empty values for learnable and state parameters is *uninitialized*. You must initialize an uninitialized `dlnetwork` before you can use it. By default, `dlnetwork` objects are constructed with initial parameters and do not need initializing.

`dlnet = initialize(dlnet,dlX1,...,dlXn)` initializes any unset learnable parameters and state values of `dlnet` based on the example network inputs `dlX1,...,dlXn`. Use this syntax when the network has inputs that are not connected to an input layer.

Examples

Initialize `dlnetwork` with Input Layer

Use the `initialize` function to initialize a `dlnetwork` object that contains an input layer.

Define the network layers.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')
    convolution2dLayer(5,20,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];
```

Create an uninitialized `dlnetwork`. Set the `Initialize` name-value option to false.

```
dlnet = dlnetwork(layers,'Initialize',false);
```

Examine the learnable parameters of the convolution layer.

```
dlnet.Learnables.Value(dlnet.Learnables.Layer=='conv')
```

```
ans =
    {0×0 double}
    {0×0 double}
```

Because the network is not initialized, the learnable parameters of the convolution layer are empty.

Initialize the learnable parameters of the network with initial values.

```
dlnet = initialize(dlnet);
```

Check the learnable parameters of the convolution layer after initialization.

```
dlnet.Learnables.Value(dlnet.Learnables.Layer=='conv')
```

```
ans =
    {5×5×1×20 dlarray}
    {1×1×20   dlarray}
```

The learnable parameters of the convolution layer are now initialized with initial values of appropriate size based on the size of the input data.

Check that the network is initialized and ready for training.

```
dlnet.Initialized
```

```
ans =
   1
```

Initialize `dlnetwork` with Input Layer and Unconnected Input

Use the `initialize` function to initialize a multi-input `dlnetwork` object that contains one input layer and one unconnected input.

Define the network architecture. Construct a network with two branches. The network takes two inputs, with one input per branch. The first branch contains an input layer, while the second branch does not. Connect the branches using an addition layer.

```
numFilters = 24;
inputSize = [64 64 3];

layersBranch1 = [
    imageInputLayer(inputSize,'Normalization','None','Name','in')
    convolution2dLayer(3,6*numFilters,'Padding','same','Stride',2,'Name','conv1Branch1')
    groupNormalizationLayer('all-channels','Name','gn1Branch1')
    reluLayer('Name','relu1Branch1')
    convolution2dLayer(3,numFilters,'Padding','same','Name','conv2Branch1')
    groupNormalizationLayer('channel-wise','Name','gn2Branch1')
    additionLayer(2,'Name','add')
    reluLayer('Name','reluCombined')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

layersBranch2 = [
    convolution2dLayer(1,numFilters,'Name','convBranch2')
    groupNormalizationLayer('all-channels','Name','gnBranch2')];
```

```
lgraph = layerGraph(layersBranch1);
lgraph = addLayers(lgraph,layersBranch2);
lgraph = connectLayers(lgraph,'gnBranch2','add/in2');
```

Convert the network to a `dlnetwork`. To construct the `dlnetwork` object without initial values of learnable and state parameters, set the `Initialize` name-value option to false..

```
dlnet = dlnetwork(lgraph,'Initialize',false);
```

Examine the learnable parameters of the second group normalization layer in the first branch of the network.

```
dlnet.Learnables.Value(dlnet.Learnables.Layer=='gn2Branch1')
```

```
ans =
    {0×0 double}
    {0×0 double}
```

Because the network is not initialized, the learnable parameters of the layer are empty.

Inspect the order of the network inputs.

```
dlnet.InputNames
```

```
ans = 1×2 cell
'in'          'convBranch2'
```

Create example input data with the same size and format as typical network inputs. Use an example input of size 64-by-64 with 3 channels for the input to the input layer `in`. Use an input of size 64-by-64 with 18 channels for the unconnected input to the layer `convBranch2`.

```
dlX1 = dlarray(rand(inputSize),"SSCB");
dlX2 = dlarray(rand([32 32 18]),"SSCB");
```

Initialize the learnable parameters of the network using the example inputs.

```
dlnet = initialize(dlnet,dlX1,dlX2);
```

Check the learnable parameters of the convolution layer after initialization.

```
dlnet.Learnables.Value(dlnet.Learnables.Layer=='gn2Branch1')
```

```
ans =
    {1×1×24 dlarray}
    {1×1×24 dlarray}
```

The learnable parameters of the convolution layer are now initialized with initial values of appropriate size based on the size of the input data.

Use the `Initialized` property of the network to check that the network is initialized and ready for training.

```
dlnet.Initialized
```

```
ans =
   1
```

Input Arguments

dlnet — Uninitialized network

`dlnetwork` object

Uninitialized network, specified as a `dlnetwork` object.

dlX1,...,dlXn — Example network inputs

`dlarray` object

Example network inputs, specified as `dlarray` objects.

Example inputs must be formatted `dlarray` objects. Provide example inputs in the same order as the order specified by the `InputNames` property of the input network.

Output Arguments

dlnet — Initialized network

`dlnetwork` object

Initialized network, returned as a `dlnetwork` object.

See Also

`dlarray` | `dlnetwork`

Topics

“Deep Learning Network Composition”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Specify Training Options in Custom Training Loop”

Introduced in R2021a

instancenorm

Normalize across each channel for each observation independently

Syntax

```
dlY = instancenorm(dlX,offset,scaleFactor)
dlY = instancenorm(dlX,offset,scaleFactor,'DataFormat',FMT)
dlY = instancenorm( ___ Name,Value)
```

Description

The instance normalization operation normalizes the input data across each channel for each observation independently. To improve the convergence of training the convolutional neural network and reduce the sensitivity to network hyperparameters, use instance normalization between convolution and nonlinear operations such as `relu`.

After normalization, the operation shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

The `instancenorm` function applies the layer normalization operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply instance normalization within a `layerGraph` object or `Layer` array, use `instanceNormalizationLayer`.

`dlY = instancenorm(dlX,offset,scaleFactor)` applies the instance normalization operation to the input data `dlX` and transforms using the specified offset and scale factor.

The function normalizes over grouped subsets of the `'S'` (spatial), `'T'` (time), and `'U'` (unspecified) dimensions of `dlX` for each observation in the `'C'` (channel) and `'B'` (batch) dimensions, independently.

For unformatted input data, use the `'DataFormat'` option.

`dlY = instancenorm(dlX,offset,scaleFactor,'DataFormat',FMT)` applies the instance normalization operation to the unformatted `dlarray` object `dlX` with format specified by `FMT` using any of the previous syntaxes. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D image input with format `'SSCB'` (spatial, spatial, channel, batch).

`dlY = instancenorm( ___ Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'Epsilon',3e-5` sets the variance offset to `3e-5`.

Examples

Apply Instance Normalization

Create randomized input data with two spatial, one channel, and one observation dimension.

```
width = 12;
height = 12;
channels = 6;
numObservations = 16;
X = randn(width,height,channels,numObservations);
dlX = dlarray(X,'SSCB');
```

Create the learnable parameters.

```
offset = dlarray(zeros(channels,1));
scaleFactor = dlarray(ones(channels,1));
```

Calculate the instance normalization.

```
dlZ = instancenorm(dlX,offset,scaleFactor);
```

View the size and format of the normalized data.

```
size(dlZ)
```

```
ans = 1×4

    12    12     6    16
```

```
dims(dlZ)
```

```
ans =
'SSCB'
```

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `dlX` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `scaleFactor` or `offset` must be a `dlarray` object.

`dlX` must have a `'C'` (channel) dimension.

`offset` — Offset

`dlarray` | numeric array

Offset β, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`.

If `offset` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).

`scaleFactor` — Scale factor
`dlarray` | numeric array

Scale factor γ, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`.

If `scaleFactor` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Epsilon',3e-5` sets the variance offset to `3e-5`.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`Epsilon` — Variance offset
`1e-5` (default) | numeric scalar

Variance offset for preventing divide-by-zero errors, specified as the comma-separated pair consisting of `'Epsilon'` and a numeric scalar greater than or equal to `1e-5`.

Data Types: `single` | `double`

Output Arguments

`dlY` — Normalized data
`dlarray`

Normalized data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Algorithms

The instance normalization operation normalizes the elements x_i of the input by first calculating the mean μ_I and variance σ_I^2 over the spatial and time dimensions for each channel in each observation independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow instance normalization, the instance normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `offset`
  - `scaleFactor`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`relu` | `fullyconnect` | `dlconv` | `dlarray` | `dlgradient` | `dlfeval` | `batchnorm` | `layernorm` | `groupnorm`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2021a

instanceNormalizationLayer

Instance normalization layer

Description

An instance normalization layer normalizes a mini-batch of data across each channel for each observation independently. To improve the convergence of training the convolutional neural network and reduce the sensitivity to network hyperparameters, use instance normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

After normalization, the layer scales the input with a learnable scale factor γ and shifts it by a learnable offset β.

Creation

Syntax

```
layer = instanceNormalizationLayer
layer = instanceNormalizationLayer(Name,Value)
```

Description

`layer = instanceNormalizationLayer` creates an instance normalization layer.

`layer = instanceNormalizationLayer(Name,Value)` creates an instance normalization layer and sets the optional `Epsilon`, “Parameters and Initialization” on page 1-914, “Learning Rate and Regularization” on page 1-915, and `Name` properties using one or more name-value arguments. You can specify multiple name-value arguments. Enclose each property name in quotes.

Example: `instanceNormalizationLayer('Name','instancenorm')` creates an instance normalization layer with the name `'instancenorm'`

Properties

Instance Normalization

`Epsilon` — Constant to add to mini-batch variances
`1e-5` (default) | numeric scalar

Constant to add to the mini-batch variances, specified as a numeric scalar equal to or larger than `1e-5`.

The layer adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`NumChannels` — Number of input channels
`'auto'` (default) | positive integer

Number of input channels, specified as `'auto'` or a positive integer.

This property is always equal to the number of channels of the input to the layer. If `NumChannels` is `'auto'`, then the software automatically determines the correct value for the number of channels at training time.

Parameters and Initialization

`ScaleInitializer` — Function to initialize channel scale factors
`'ones'` (default) | `'narrow-normal'` | function handle

Function to initialize the channel scale factors, specified as one of the following:

- `'ones'` – Initialize the channel scale factors with ones.
- `'zeros'` – Initialize the channel scale factors with zeros.
- `'narrow-normal'` – Initialize the channel scale factors by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel scale factors with a custom function. If you specify a function handle, then the function must be of the form `scale = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel scale factors when the `Scale` property is empty.

Data Types: `char` | `string` | `function_handle`

`OffsetInitializer` — Function to initialize channel offsets
`'zeros'` (default) | `'ones'` | `'narrow-normal'` | function handle

Function to initialize the channel offsets, specified as one of the following:

- `'zeros'` – Initialize the channel offsets with zeros.
- `'ones'` – Initialize the channel offsets with ones.
- `'narrow-normal'` – Initialize the channel offsets by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel offsets with a custom function. If you specify a function handle, then the function must be of the form `offset = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel offsets when the `Offset` property is empty.

Data Types: `char` | `string` | `function_handle`

`Scale` — Channel scale factors
`[]` (default) | numeric array

Channel scale factors γ, specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network, if `Scale` is nonempty, then `trainNetwork` uses the `Scale` property as the initial value. If `Scale` is empty, then `trainNetwork` uses the initializer specified by `ScaleInitializer`.

At training time, `Scale` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

`Offset` — Channel offsets
`[ ]` (default) | numeric array

Channel offsets β, specified as a numeric array.

The channel offsets are learnable parameters. When you train a network, if `Offset` is nonempty, then `trainNetwork` uses the `Offset` property as the initial value. If `Offset` is empty, then `trainNetwork` uses the initializer specified by `OffsetInitializer`.

At training time, `Offset` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

Learning Rate and Regularization

`ScaleLearnRateFactor` — Learning rate factor for scale factors
1 (default) | nonnegative scalar

Learning rate factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the scale factors in a layer. For example, if `ScaleLearnRateFactor` is 2, then the learning rate for the scale factors in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`OffsetLearnRateFactor` — Learning rate factor for offsets
1 (default) | nonnegative scalar

Learning rate factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the offsets in a layer. For example, if `OffsetLearnRateFactor` is 2, then the learning rate for the offsets in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`ScaleL2Factor` — L_2 regularization factor for scale factors
1 (default) | nonnegative scalar

L_2 regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

`OffsetL2Factor` — L_2 regularization factor for offsets
1 (default) | nonnegative scalar

L_2 regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Instance Normalization Layer

Create an instance normalization layer with the name `'instancenorm'`.

```
layer = instanceNormalizationLayer('Name','instancenorm')
```

```
layer =
  InstanceNormalizationLayer with properties:

           Name: 'instancenorm'
    NumChannels: 'auto'

   Hyperparameters
        Epsilon: 1.0000e-05

   Learnable Parameters
         Offset: []
          Scale: []

  Show all properties
```

Include an instance normalization layer in a `Layer` array.

```
layers = [
    imageInputLayer([28 28 3])
    convolution2dLayer(5,20)
    instanceNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  8x1 Layer array with layers:

     1   ''   Image Input             28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0
     3   ''   Instance Normalization   Instance normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  (
     6   ''   Fully Connected          10 fully connected layer
     7   ''   Softmax                 softmax
     8   ''   Classification Output    crossentropyex
```

Algorithms

The instance normalization operation normalizes the elements x_i of the input by first calculating the mean μ_I and variance σ_I^2 over the spatial and time dimensions for each channel in each observation independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow instance normalization, the instance normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \hat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

See Also

`trainNetwork` | `trainingOptions` | `reluLayer` | `convolution2dLayer` | `fullyConnectedLayer` | `batchNormalizationLayer` | `groupNormalizationLayer` | `layerNormalizationLayer`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2021a

isdlarray

Determine whether input is `dlarray`

Syntax

```
TF = isdlarray(X)
```

Description

`TF = isdlarray(X)` returns logical `1` (`true`) if `X` is a `dlarray`, and logical `0` (`false`) otherwise. You can use this function with an `if` statement to avoid executing code that expects `dlarray` input.

Examples

Determine if Array is `dlarray`

Create an array of random numbers.

```
X = rand(3,3);
```

Create a `dlarray` from `X`.

```
dlX = dlarray(X);
```

Use the function `isdlarray` to verify that `dlX` is a `dlarray`

```
isdlarray(dlX)
```

```
ans =
   1
```

Verify that `X` is not a `dlarray`

```
isdlarray(X)
```

```
ans =
   0
```

Input Arguments

X — Input variable

workspace variable

Input variable, specified as a workspace variable. `X` can be any data type.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `extractdata`

Topics

“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”

Introduced in R2020b

isequal

Check equality of deep learning layer graphs or networks

Syntax

```
tf = isequal(net1,net2)
tf = isequal(net1,...,netN)
```

Description

`tf = isequal(net1,net2)` checks the equality of `SeriesNetwork`, `DAGNetwork`, `LayerGraph`, or `dlnetwork` objects `net1` and `net2`.

- If both inputs are `SeriesNetwork` or `DAGNetwork` objects, or one of each, then the function returns `1` (true) when the properties and architectures match. Otherwise, the function returns `0` (false).
- If both inputs are `LayerGraph` objects, then the function returns `1` (true) when the properties and architectures match. Otherwise, the function returns `0` (false).
- If both inputs are `dlnetwork` objects, then the function returns `1` (true) when the properties and architectures match. Otherwise, the function returns `0` (false).
- For other combinations, the function returns `0` (false).

`tf = isequal(net1,...,netN)` checks equality of the `N` networks or layer graphs `net1`, ..., `netN`.

Examples

Check if Layer Graphs are Equal

Create two instances of SqueezeNet layer graphs.

```
lgraph1 = squeezenet('Weights','none');
lgraph2 = squeezenet('Weights','none');
```

Check if the layer graphs are equal using the `isequal` function.

```
tf = isequal(lgraph1,lgraph2)
```

```
tf = logical
   1
```

Check if Networks are Equal

Create two instances of a pretrained SqueezeNet network.

```
net1 = squeezenet;
net2 = squeezenet;
```

Check if the networks are equal using the `isequal` function.

```
tf = isequal(net1,net2)
```

```
tf = logical
   1
```

Input Arguments

net — Network or layer graph

`SeriesNetwork` object | `DAGNetwork` object | `dlnetwork` object | `LayerGraph` object

Network or layer graph, specified as a `SeriesNetwork`, `DAGNetwork`, `dlnetwork` , or `LayerGraph` object.

See Also

`isequaln` | `trainNetwork` | `SeriesNetwork` | `DAGNetwork` | `analyzeNetwork` | `assembleNetwork` | `dlnetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Train Convolutional Neural Network for Regression”
“Sequence Classification Using Deep Learning”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2021a

isequaln

Check equality of deep learning layer graphs or networks ignoring `NaN` values

Syntax

```
tf = isequaln(net1,net2)
tf = isequaln(net1,...,netN)
```

Description

`tf = isequaln(net1,net2)` checks the equality of `SeriesNetwork`, `DAGNetwork`, `LayerGraph`, or `dlnetwork` objects `net1` and `net2`.

- If both inputs are `SeriesNetwork` or `DAGNetwork` objects, or one of each, then the function returns `1` (true) when the properties and architectures match, ignoring `NaN` values. Otherwise, the function returns `0` (false).
- If both inputs are `LayerGraph` objects, then the function returns `1` (true) when the properties and architectures match, ignoring `NaN` values. Otherwise, the function returns `0` (false).
- If both inputs are `dlnetwork` objects, then the function returns `1` (true) when the properties and architectures match, ignoring `NaN` values. Otherwise, the function returns `0` (false).
- For other combinations, the function returns `0` (false).

`tf = isequaln(net1,...,netN)` checks equality of the `N` networks or layer graphs `net1`, ..., `netN`, ignoring `NaN` values.

Examples

Check if Layer Graphs are Equal Ignoring NaN Values

Create two instances of SqueezeNet layer graphs.

```
lgraph1 = squeezenet('Weights','none');
lgraph2 = squeezenet('Weights','none');
```

Check if the layer graphs are equal ignoring `NaN` values using the `isequaln` function.

```
tf = isequaln(lgraph1,lgraph2)
```

```
tf = logical
   1
```

Check if Networks are Equal Ignoring NaN Values

Create two instances of a pretrained SqueezeNet network.

```
net1 = squeezenet;
net2 = squeezenet;
```

Check if the networks are equal ignoring `NaN` values using the `isequaln` function.

```
tf = isequaln(net1,net2)
```

```
tf = logical
   1
```

Input Arguments

net — Network or layer graph

`SeriesNetwork` object | `DAGNetwork` object | `dlnetwork` object | `LayerGraph` object

Network or layer graph, specified as a `SeriesNetwork`, `DAGNetwork`, `dlnetwork` , or `LayerGraph` object.

See Also

`isequal` | `trainNetwork` | `SeriesNetwork` | `DAGNetwork` | `analyzeNetwork` | `assembleNetwork` | `dlnetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Train Convolutional Neural Network for Regression”
“Sequence Classification Using Deep Learning”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2021a

l1loss

L_1 loss for regression tasks

Syntax

```
loss = l1loss(dlY,targets)
loss = l1loss(dlY,targets,weights)
loss = l1loss(___,DataFormat=FMT)
loss = l1loss(___,Name=Value)
```

Description

The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean absolute error (MAE).

The `l1loss` function calculates the L_1 loss using `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

`loss = l1loss(dlY,targets)` computes the MAE loss for the predictions `dlY` and the target values `targets`. The input `dlY` must be a formatted `dlarray`. The output `loss` is an unformatted `dlarray` scalar.

`loss = l1loss(dlY,targets,weights)` computes the weighted L_1 loss using the weight values `weights`. The output `loss` is an unformatted `dlarray` scalar.

`loss = l1loss(___,DataFormat=FMT)` computes the loss for the unformatted `dlarray` object `dlY` and the target values with the format specified by `FMT`. Use this syntax with any of the input arguments in previous syntaxes.

`loss = l1loss(___,Name=Value)` specifies additional options using one or more name-value arguments. For example, `l1loss(dlY,targets,Reduction="none")` computes the L_1 loss without reducing the output to a scalar.

Examples

Mean Absolute Error Loss

Create an array of predictions for 12 observations over 10 responses.

```
numResponses = 10;
numObservations = 12;

Y = rand(numResponses,numObservations);
dlY = dlarray(Y,'CB');
```

View the size and format of the predictions.

```
size(dlY)
```

```
ans = 1×2

    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create an array of random targets.

```
targets = rand(numResponses,numObservations);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the mean absolute error (MAE) loss between the predictions and the targets using the `l1loss` function.

```
loss = l1loss(dlY,targets)
```

```
loss =
  1x1 dlarray

    3.1679
```

Masked Mean Absolute Error for Padded Sequences

Create arrays of predictions and targets for 12 sequences of varying lengths over 10 responses.

```
numResponses = 10;
numObservations = 12;
maxSequenceLength = 15;

sequenceLengths = randi(maxSequenceLength,[1 numObservations]);

Y = cell(numObservations,1);
targets = cell(numObservations,1);

for i = 1:numObservations
    Y{i} = rand(numResponses,sequenceLengths(i));
    targets{i} = rand(numResponses,sequenceLengths(i));
end
```

View the cell arrays of predictions and targets.

```
Y
```

```
Y=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

```
targets
```

```
targets=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

Pad the prediction and target sequences in the second dimension using the `padsequences` function and also return the corresponding mask.

```
[Y,mask] = padsequences(Y,2);
targets = padsequences(targets,2);
```

Convert the padded sequences to `dlarray` with the format `"CTB"` (channel, time, batch). Because formatted `dlarray` objects automatically permute the dimensions of the underlying data, keep the order consistent by also converting the targets and mask to formatted `dlarray` objects with the format `"CTB"` (channel, batch, time).

```
dlY = dlarray(Y,"CTB");
targets = dlarray(targets,"CTB");
mask = dlarray(mask,"CTB");
```

View the sizes of the prediction scores, targets, and mask.

```
size(dlY)
```

```
ans = 1×3

    10    12    15
```

```
size(targets)
```

```
ans = 1×3

    10    12    15


size(mask)

ans = 1×3

    10    12    15
```

Compute the mean absolute error (MAE) between the predictions and the targets. To prevent the loss values calculated from padding from contributing to the loss, set the `Mask` option to the mask returned by the `padsequences` function.

```
loss = l1loss(dlY,targets,Mask=mask)

loss =
  1x1 dlarray

   32.6172
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `dlY` is a numeric array, `targets` must be a `dlarray`.

`targets` — Target responses

`dlarray` | numeric array

Target responses, specified as a formatted or unformatted `dlarray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `dlY`.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`weights` — Weights

`dlarray` | numeric array

Weights, specified as a formatted or unformatted `dlarray` or a numeric array.

If `weights` is a vector and `dlY` has two or more nonsingleton dimensions, then `weights` must be a formatted `dlarray`, where the dimension label of the nonsingleton dimension is either `"C"` (channel) or `"B"` (batch) and has a size that matches the size of the corresponding dimension in `dlY`.

If `weights` is a formatted `dlarray` with two or more nonsingleton dimensions, then its format must match the format of `dlY`.

If `weights` is not a formatted `dlarray` and has two or more nonsingleton dimensions, then its size must match the size of `dlY` and the function uses the same format as `dlY`. Alternatively, to specify the weights format, use the `WeightsFormat` option.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `loss = l1loss(dlY,targets,Reduction="none")` specifies to compute the L_1 loss without reducing the output to a scalar

`Mask` — Mask indicating which elements to include for loss computation

`dlarray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the input data for loss computation when the corresponding value in the mask is 1 and 0, respectively.

The default value is a logical array of ones with the same size as `dlY`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

`Reduction` — Mode for reducing array of loss values

`"sum"` (default) | `"none"`

Mode for reducing the array of loss values, specified as one of the following:

- `"sum"` — Sum all of the elements in the array of loss values. In this case, the output `loss` is scalar.
- `"none"` — Do not reduce the array of loss values. In this case, the output `loss` is an unformatted `dlarray` object with the same size as `dlY`.

`NormalizationFactor` — Divisor for normalizing reduced loss
`"batch-size"` (default) | `"all-elements"` | `"mask-included"` | `"none"`

Divisor for normalizing the reduced loss when `Reduction` is `"sum"`, specified as one of the following:

- `"batch-size"` — Normalize the loss by dividing it by the number of observations in `dlX`.
- `"all-elements"` — Normalize the loss by dividing it by the number of elements of `dlX`.
- `"mask-included"` — Normalize the loss by dividing the loss values by the number of included elements specified by the mask for each observation independently. To use this option, you must specify a mask using the `Mask` option.
- `"none"` — Do not normalize the loss.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`WeightsFormat` — Dimension order of weights
character vector | string scalar

Dimension order of the weights, specified as a character vector or string scalar that provides a label for each dimension of the weights.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `WeightsFormat` when `weights` is a numeric vector and `dlY` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `dlY` are vectors, then default value of `WeightsFormat` is the same as the format of `dlY`.

Data Types: `char` | `string`

Output Arguments

`loss` — L_1 loss

`dlarray`

L_1 loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `dlY`.

The size of `loss` depends on the `Reduction` option.

Algorithms

L1 Loss

The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean absolute error (MAE).

For each element Y_j of the input, the `l1loss` function computes the corresponding element-wise loss values using

$$\text{loss}_j = \left|Y_j - T_j\right|,$$

where Y_j is a predicted value and T_j is the corresponding target value.

To reduce the loss values to a scalar, the function then reduces the element-wise loss using the formula

$$\text{loss} = -\frac{1}{N}\sum_j m_j w_j \text{loss}_j,$$

where N is the normalization factor, m_j is the mask value for element j, and w_j is the weight value for element j.

If you do not opt to reduce the loss, then the function applies the mask and the weights to the loss values directly:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlY`
  - `targets`
  - `weights`
  - `Mask`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `crossentropy` | `l2loss` | `huber` | `mse` | `ctc`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2021b

l2loss

L_2 loss for regression tasks

Syntax

```
loss = l2loss(dlY,targets)
loss = l2loss(dlY,targets,weights)
loss = l2loss( ___ ,DataFormat=FMT)
loss = l2loss( ___ ,Name=Value)
```

Description

The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean squared error (MSE).

The `l2loss` function calculates the L_2 loss using `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

`loss = l2loss(dlY,targets)` computes MSE loss for the predictions `dlY` and the target values `targets`. The input `dlY` must be a formatted `dlarray`. The output `loss` is an unformatted `dlarray` scalar.

`loss = l2loss(dlY,targets,weights)` computes the weighted L_2 loss using the weight values `weights`. The output `loss` is an unformatted `dlarray` scalar.

`loss = l2loss( ___ ,DataFormat=FMT)` computes the loss for the unformatted `dlarray` object `dlY` and the target values with the format specified by `FMT`. Use this syntax with any of the input arguments in previous syntaxes.

`loss = l2loss( ___ ,Name=Value)` specifies additional options using one or more name-value arguments. For example, `l2loss(dlY,targets,Reduction="none")` computes the L_2 loss without reducing the output to a scalar.

Examples

Mean Squared Error Loss

Create an array of predictions for 12 observations over 10 responses.

```
numResponses = 10;
numObservations = 12;
Y = rand(numResponses,numObservations);
dlY = dlarray(Y,'CB');
```

View the size and format of the predictions.

```
size(dlY)
```

```
ans = 1×2

    10    12
```

```
dims(dlY)
```

```
ans =
'CB'
```

Create an array of random targets.

```
targets = rand(numResponses,numObservations);
```

View the size of the targets.

```
size(targets)
```

```
ans = 1×2

    10    12
```

Compute the mean squared error (MSE) loss between the predictions and the targets using the `l2loss` function.

```
loss = l2loss(dlY,targets)
```

```
loss =
  1x1 dlarray

    1.4748
```

Masked Mean Squared Error for Padded Sequences

Create arrays of predictions and targets for 12 sequences of varying lengths over 10 responses.

```
numResponses = 10;
numObservations = 12;
maxSequenceLength = 15;

sequenceLengths = randi(maxSequenceLength,[1 numObservations]);

Y = cell(numObservations,1);
targets = cell(numObservations,1);

for i = 1:numObservations
    Y{i} = rand(numResponses,sequenceLengths(i));
    targets{i} = rand(numResponses,sequenceLengths(i));
end
```

View the cell arrays of predictions and targets.

```
Y
```

```
Y=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

```
targets
```

```
targets=12×1 cell array
    {10x13 double}
    {10x14 double}
    {10x2  double}
    {10x14 double}
    {10x10 double}
    {10x2  double}
    {10x5  double}
    {10x9  double}
    {10x15 double}
    {10x15 double}
    {10x3  double}
    {10x15 double}
```

Pad the prediction and target sequences in the second dimension using the `padsequences` function and also return the corresponding mask.

```
[Y,mask] = padsequences(Y,2);
targets = padsequences(targets,2);
```

Convert the padded sequences to `dlarray` with the format `"CTB"` (channel, time, batch). Because formatted `dlarray` objects automatically permute the dimensions of the underlying data, keep the order consistent by also converting the targets and mask to formatted `dlarray` objects with the format `"CTB"` (channel, batch, time).

```
dlY = dlarray(Y,"CTB");
targets = dlarray(targets,"CTB");
mask = dlarray(mask,"CTB");
```

View the sizes of the prediction scores, targets, and mask.

```
size(dlY)
```

```
ans = 1×3

    10    12    15
```

```
size(targets)
```

```
ans = 1×3
```

```
    10    12    15
```

```
size(mask)
```

```
ans = 1×3

    10    12    15
```

Compute the mean squared error (MSE) between the predictions and the targets. To prevent the loss values calculated from padding from contributing to the loss, set the `Mask` option to the mask returned by the `padsequences` function.

```
loss = l2loss(dlY,targets,Mask=mask)
```

```
loss =
  1x1 dlarray

   16.3668
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `dlY` is a numeric array, `targets` must be a `dlarray`.

`targets` — Target responses

`dlarray` | numeric array

Target responses, specified as a formatted or unformatted `dlarray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `dlY`.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`weights` — Weights

`dlarray` | numeric array

Weights, specified as a formatted or unformatted `dlarray` or a numeric array.

If `weights` is a vector and `dlY` has two or more nonsingleton dimensions, then `weights` must be a formatted `dlarray`, where the dimension label of the nonsingleton dimension is either `"C"` (channel) or `"B"` (batch) and has a size that matches the size of the corresponding dimension in `dlY`.

If `weights` is a formatted `dlarray` with two or more nonsingleton dimensions, then its format must match the format of `dlY`.

If `weights` is not a formatted `dlarray` and has two or more nonsingleton dimensions, then its size must match the size of `dlY` and the function uses the same format as `dlY`. Alternatively, to specify the weights format, use the `WeightsFormat` option.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `loss = l2loss(dlY,targets,Reduction="none")` specifies to compute the L_2 loss without reducing the output to a scalar

`Mask` — Mask indicating which elements to include for loss computation

`dlarray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `dlarray` object, a logical array, or a numeric array with the same size as `dlY`.

The function includes and excludes elements of the input data for loss computation when the corresponding value in the mask is 1 and 0, respectively.

The default value is a logical array of ones with the same size as `dlY`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have this order: `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), and `"U"` (unspecified). For example, `dlarray` objects automatically permute the dimensions of data with format `"TSCSBS"` to have format `"SSSCBT"`.

To ensure that the dimensions of `dlY` and the mask are consistent, when `dlY` is a formatted `dlarray`, also specify the mask as a formatted `dlarray`.

`Reduction` — Mode for reducing array of loss values

`"sum"` (default) | `"none"`

Mode for reducing the array of loss values, specified as one of the following:

- `"sum"` — Sum all of the elements in the array of loss values. In this case, the output `loss` is scalar.
- `"none"` — Do not reduce the array of loss values. In this case, the output `loss` is an unformatted `dlarray` object with the same size as `dlY`.

`NormalizationFactor` — Divisor for normalizing reduced loss

`"batch-size"` (default) | `"all-elements"` | `"mask-included"` | `"none"`

Divisor for normalizing the reduced loss when `Reduction` is `"sum"`, specified as one of the following:

- `"batch-size"` — Normalize the loss by dividing it by the number of observations in `dlX`.
- `"all-elements"` — Normalize the loss by dividing it by the number of elements of `dlX`.
- `"mask-included"` — Normalize the loss by dividing the loss values by the number of included elements specified by the mask for each observation independently. To use this option, you must specify a mask using the `Mask` option.
- `"none"` — Do not normalize the loss.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`WeightsFormat` — Dimension order of weights
character vector | string scalar

Dimension order of the weights, specified as a character vector or string scalar that provides a label for each dimension of the weights.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `WeightsFormat` when `weights` is a numeric vector and `dlY` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `dlY` are vectors, then default value of `WeightsFormat` is the same as the format of `dlY`.

Data Types: `char` | `string`

Output Arguments

`loss` — L_2 loss

`dlarray`

L_2 loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `dlY`.

The size of `loss` depends on the `Reduction` option.

Algorithms

L2 Loss

The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean squared error (MSE).

For each element Y_j of the input, the `l2loss` function computes the corresponding element-wise loss values using

$$\text{loss}_j = \left(Y_j - T_j\right)^2,$$

where Y_j is a predicted value and T_j is the corresponding target value.

To reduce the loss values to a scalar, the function then reduces the element-wise loss using the formula

$$\text{loss} = -\frac{1}{N}\sum_j m_j w_j \text{loss}_j,$$

where N is the normalization factor, m_j is the mask value for element j, and w_j is the weight value for element j.

If you do not opt to reduce the loss, then the function applies the mask and the weights to the loss values directly:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:

- `dlY`
- `targets`
- `weights`
- `Mask`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `crossentropy` | `l1loss` | `huber` | `mse` | `ctc`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2021b

Layer

Network layer for deep learning

Description

Layers that define the architecture of neural networks for deep learning.

Creation

For a list of deep learning layers in MATLAB, see “List of Deep Learning Layers”. 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.

Alternatively, you can import layers from Caffe, Keras, and ONNX using `importCaffeLayers`, `importKerasLayers`, and `importONNXLayers` respectively.

To learn how to create your own custom layers, see “Define Custom Deep Learning Layers”.

Object Functions

trainNetwork Train deep learning neural network

Examples

Construct Network Architecture

Define a convolutional neural network architecture for classification with one convolutional layer, a ReLU layer, and a fully connected layer.

```
layers = [ ...
    imageInputLayer([28 28 3])
    convolution2dLayer([5 5],10)
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  6x1 Layer array with layers:

     1   ''   Image Input             28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             10 5x5 convolutions with stride [1  1] and padding [0  0  0
     3   ''   ReLU                    ReLU
     4   ''   Fully Connected         10 fully connected layer
     5   ''   Softmax                 softmax
     6   ''   Classification Output   crossentropyex
```

`layers` is a `Layer` object.

Alternatively, you can create the layers individually and then concatenate them.

```
input = imageInputLayer([28 28 3]);
conv = convolution2dLayer([5 5],10);
relu = reluLayer;
fc = fullyConnectedLayer(10);
sm = softmaxLayer;
co = classificationLayer;

layers = [ ...
    input
    conv
    relu
    fc
    sm
    co]
```

```
layers =
  6x1 Layer array with layers:

     1   ''   Image Input             28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             10 5x5 convolutions with stride [1  1] and padding [0  0  0
     3   ''   ReLU                    ReLU
     4   ''   Fully Connected         10 fully connected layer
     5   ''   Softmax                 softmax
     6   ''   Classification Output   crossentropyex
```

Access Layers and Properties in Layer Array

Define a convolutional neural network architecture for classification with one convolutional layer, a ReLU layer, and a fully connected layer.

```
layers = [ ...
    imageInputLayer([28 28 3])
    convolution2dLayer([5 5],10)
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Display the image input layer by selecting the first layer.

```
layers(1)
```

```
ans =
  ImageInputLayer with properties:

                      Name: ''
                 InputSize: [28 28 3]

   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                      Mean: []
```

View the input size of the image input layer.

```
layers(1).InputSize
```

```
ans = 1×3

    28    28     3
```

Display the stride for the convolutional layer.

```
layers(2).Stride
```

```
ans = 1×2

     1     1
```

Access the bias learn rate factor for the fully connected layer.

```
layers(4).BiasLearnRateFactor
```

```
ans = 1
```

Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

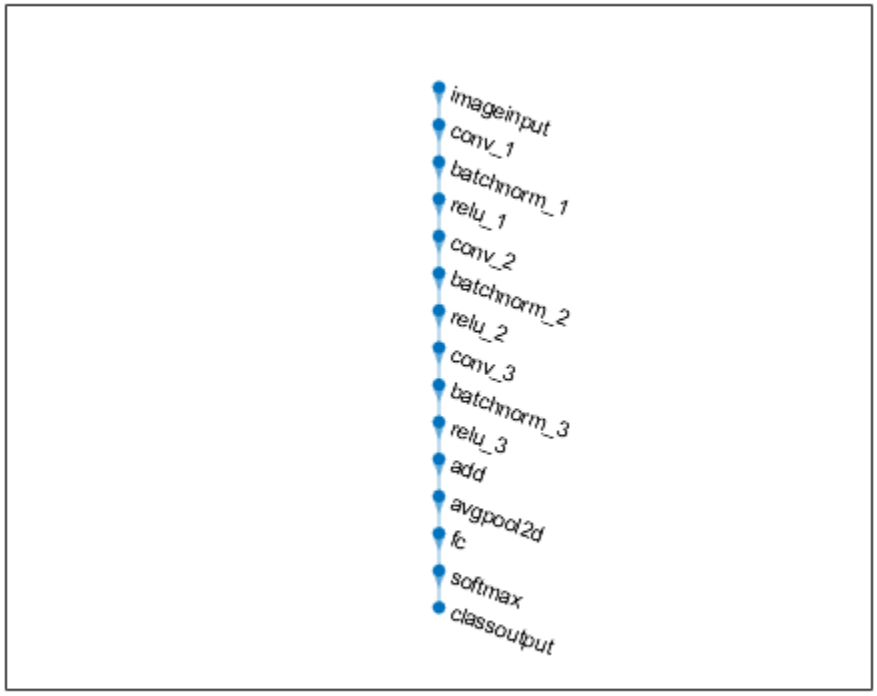
- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(5,16,'Padding','same')
    batchNormalizationLayer
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

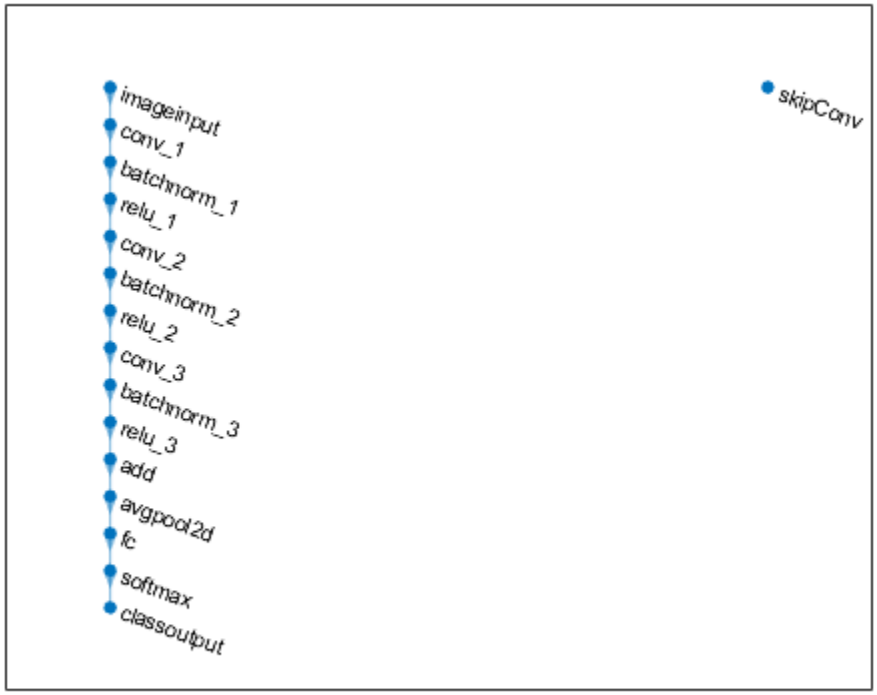
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



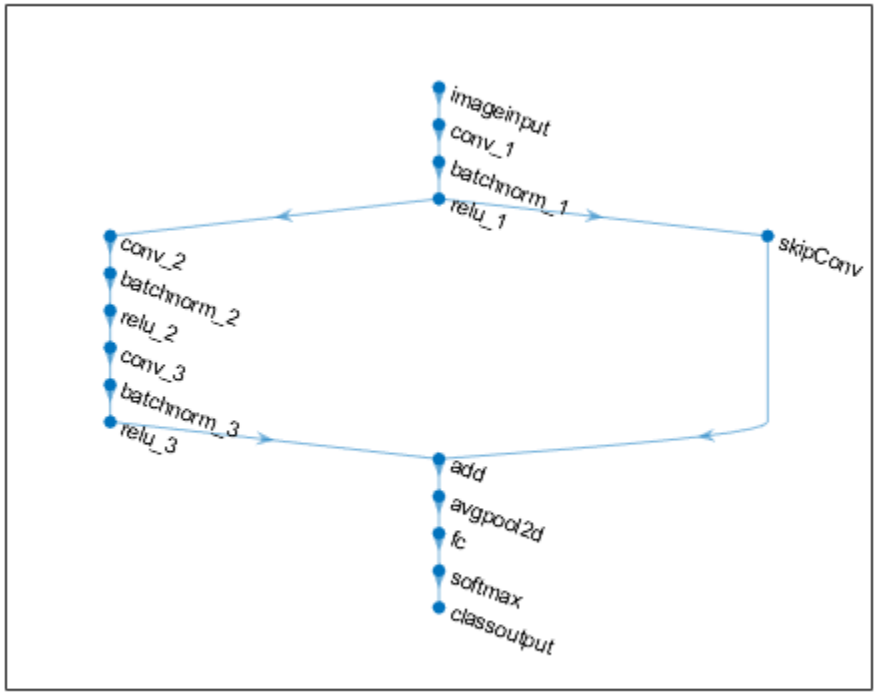
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 1-by-1 convolutional layer. To check that the layer is in the graph, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` layer to the `'add'` layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named `'in1'` and `'in2'`. The third ReLU layer is already connected to the `'in1'` input. Connect the `'relu_1'` layer to the `'skipConv'` layer and the `'skipConv'` layer to the `'in2'` input of the `'add'` layer. The addition layer now sums the outputs of the third ReLU layer and the `'skipConv'` layer. To check that the layers are connected correctly, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

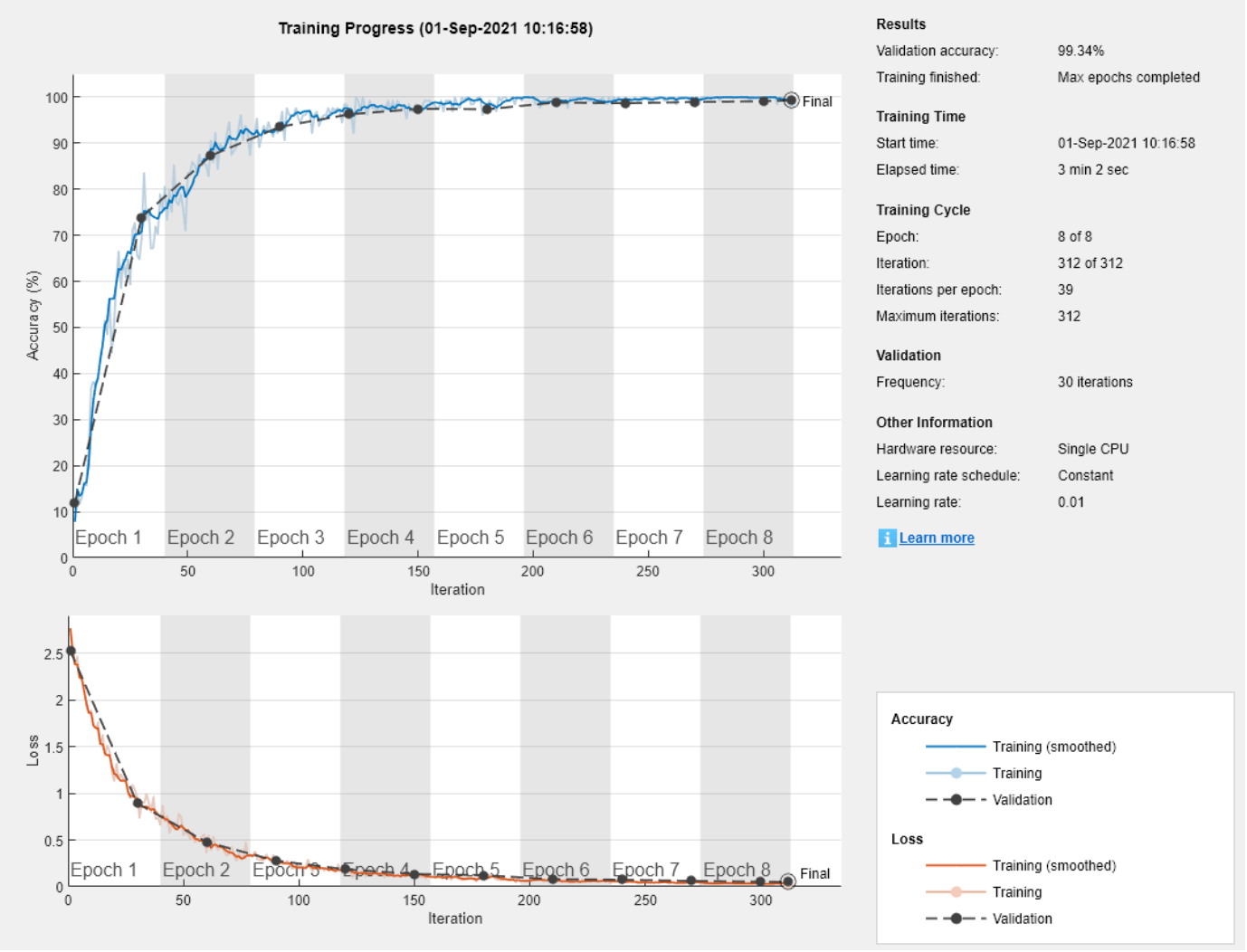


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



Display the properties of the trained network. The network is a `DAGNetwork` object.

```
net
```

```
net =
  DAGNetwork with properties:

         Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9934
```

See Also

`importCaffeLayers` | `trainNetwork` | `LayerGraph` | `Layer` | `importKerasLayers` | `assembleNetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”

Introduced in R2016a

layerGraph

Graph of network layers for deep learning

Description

A layer graph specifies the architecture of a deep learning network with a more complex graph structure in which layers can have inputs from multiple layers and outputs to multiple layers. Networks with this structure are called directed acyclic graph (DAG) networks. After you create a `layerGraph` object, you can use the object functions to plot the graph and modify it by adding, removing, connecting, and disconnecting layers. To train the network, use the layer graph as input to the `trainNetwork` function.

Creation

Syntax

```
lgraph = layerGraph
lgraph = layerGraph(layers)
lgraph = layerGraph(net)
lgraph = layerGraph(dlnet)
```

Description

`lgraph = layerGraph` creates an empty layer graph that contains no layers. You can add layers to the empty graph by using the `addLayers` function.

`lgraph = layerGraph(layers)` creates a layer graph from an array of network layers and sets the `Layers` property. The layers in `lgraph` are connected in the same sequential order as in `layers`.

`lgraph = layerGraph(net)` extracts the layer graph of a `SeriesNetwork` or `DAGNetwork` object. For example, you can extract the layer graph of a pretrained network to perform transfer learning.

`lgraph = layerGraph(dlnet)` extracts the layer graph of a `dlnetwork`. Use this syntax to use a `dlnetwork` with the `trainNetwork` function or **Deep Network Designer**.

Input Arguments

`net` — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

`dlnet` — Network

`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

For `dlnetwork` input, the software extracts the numeric data from the learnable parameters and converts it to single precision.

Properties

`Layers` — Network layers
`Layer` array

This property is read-only.

Network layers, specified as a `Layer` array.

`Connections` — Layer connections
table

This property is read-only.

Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, `Source`, specifies the source of each connection. The second column, `Destination`, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form `'layerName/IOName'`, where `'IOName'` is the name of the layer input or output.

Data Types: `table`

`InputNames` — Network input layer names
cell array of character vectors

This property is read-only.

Network input layer names, specified as a cell array of character vectors.

Data Types: `cell`

`OutputNames` — Network output layer names
cell array

Network output layer names, specified as a cell array of character vectors.

Data Types: `cell`

Object Functions

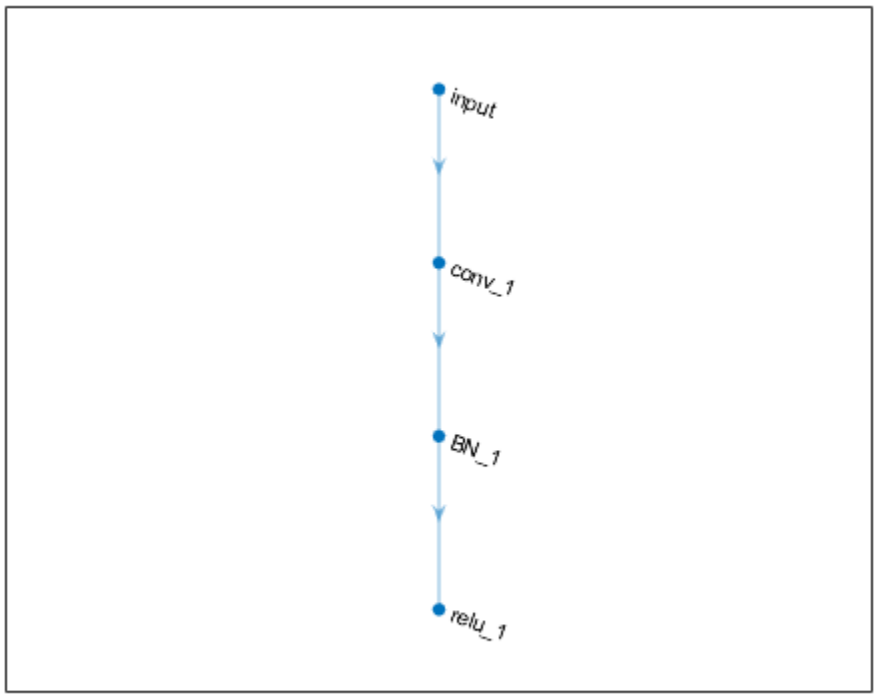
addLayers	Add layers to layer graph
removeLayers	Remove layers from layer graph
replaceLayer	Replace layer in layer graph
connectLayers	Connect layers in layer graph
disconnectLayers	Disconnect layers in layer graph
plot	Plot neural network layer graph

Examples

Add Layers to Layer Graph

Create an empty layer graph and an array of layers. Add the layers to the layer graph and plot the graph. `addLayers` connects the layers sequentially.

```
lgraph = layerGraph;

layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];

lgraph = addLayers(lgraph,layers);
figure
plot(lgraph)
```



Create Layer Graph from an Array of Layers

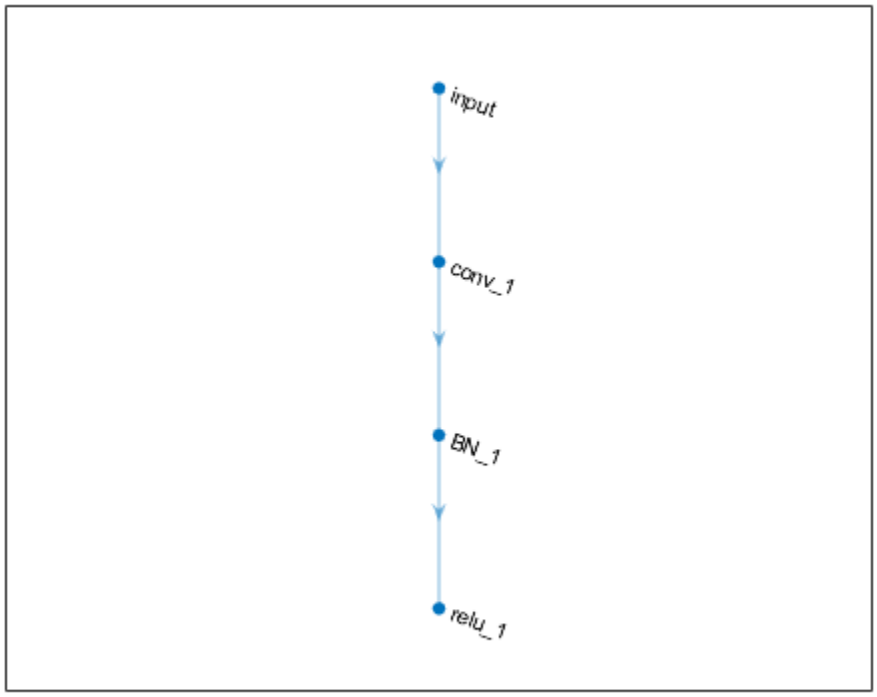
Create an array of layers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
```

```
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];
```

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



Extract Layer Graph of DAG Network

Load a pretrained SqueezeNet network. You can use this trained network for classification and prediction.

```
net = squeezenet;
```

To modify the network structure, first extract the structure of the DAG network by using `layerGraph`. You can then use the object functions of `LayerGraph` to modify the network architecture.

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

```
lgraph =
  LayerGraph with properties:
```

```
     Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
 InputNames: {'data'}
OutputNames: {'ClassificationLayer_predictions'}
```

Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

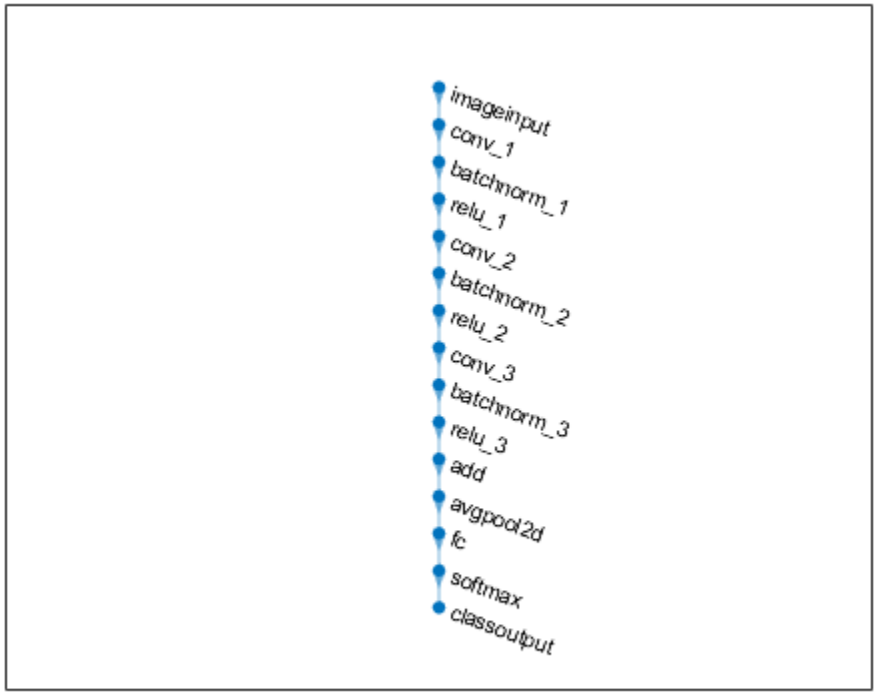
- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(5,16,'Padding','same')
    batchNormalizationLayer
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

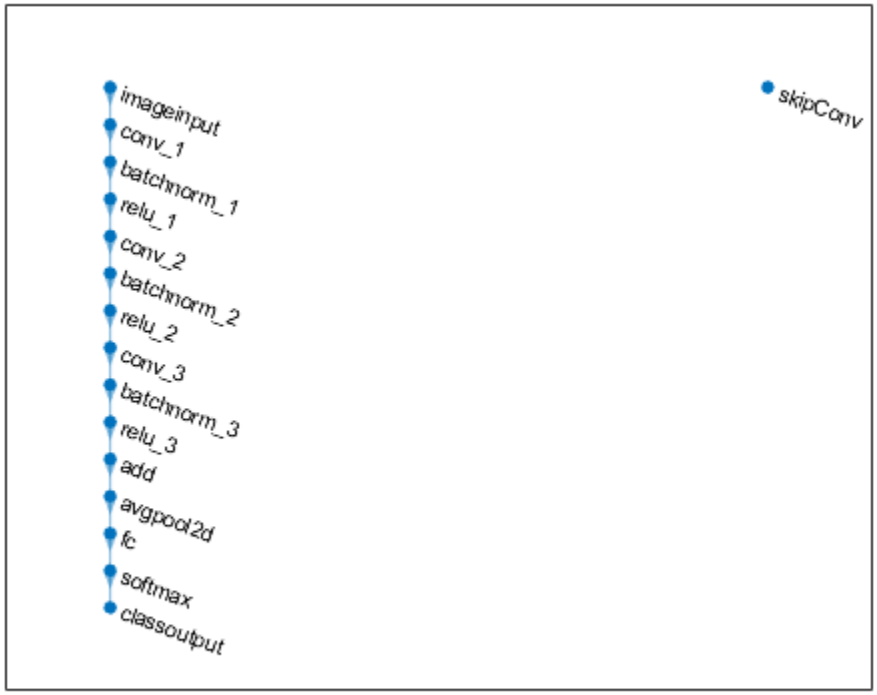
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

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



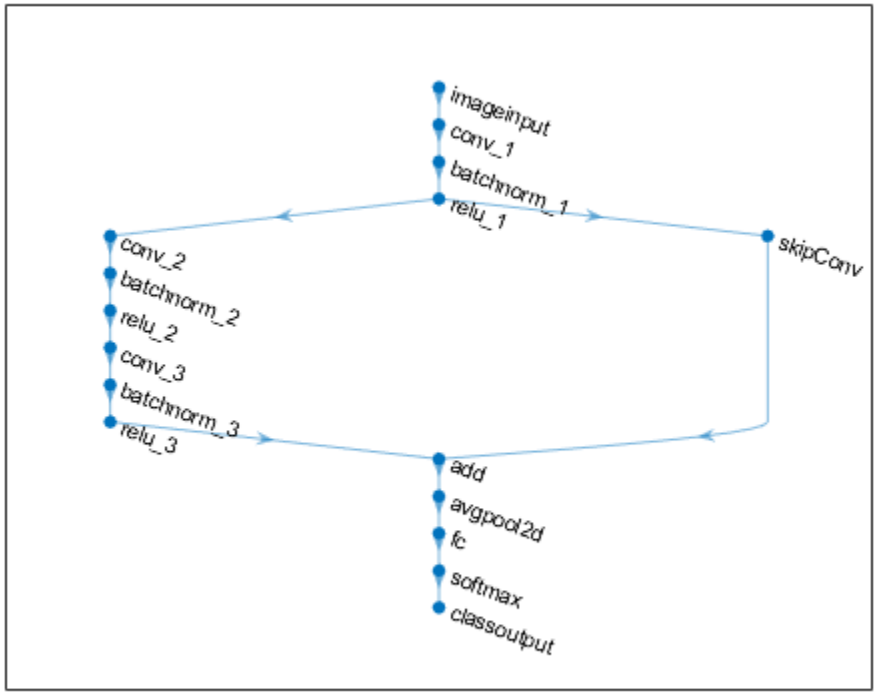
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 1-by-1 convolutional layer. To check that the layer is in the graph, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` layer to the `'add'` layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named `'in1'` and `'in2'`. The third ReLU layer is already connected to the `'in1'` input. Connect the `'relu_1'` layer to the `'skipConv'` layer and the `'skipConv'` layer to the `'in2'` input of the `'add'` layer. The addition layer now sums the outputs of the third ReLU layer and the `'skipConv'` layer. To check that the layers are connected correctly, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

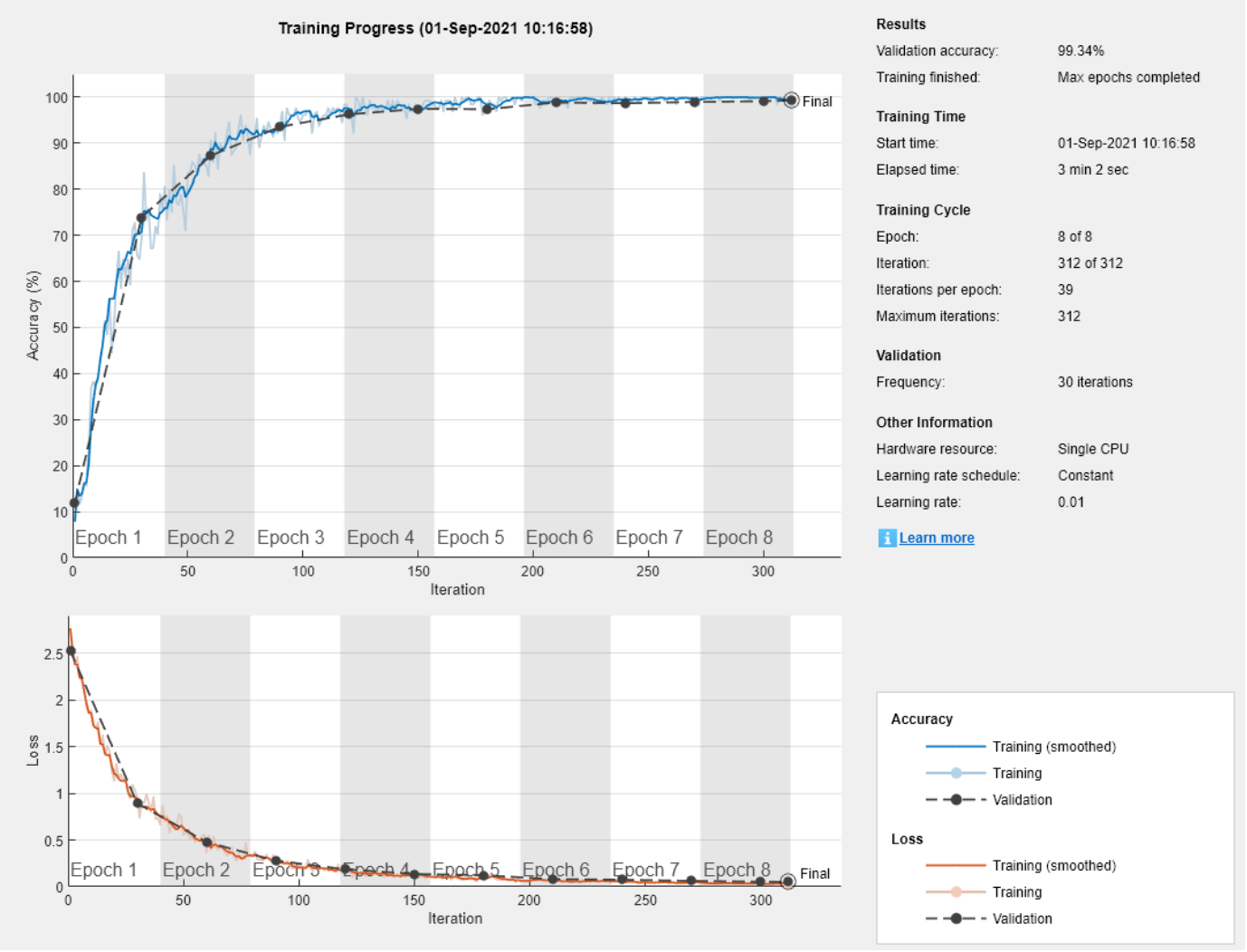


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



Display the properties of the trained network. The network is a `DAGNetwork` object.

```
net
```

```
net =
  DAGNetwork with properties:

         Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9934
```

See Also

`trainNetwork` | `DAGNetwork` | `addLayers` | `removeLayers` | `connectLayers` | `disconnectLayers` | `plot` | `googlenet` | `resnet18` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `additionLayer` | `replaceLayer` | `depthConcatenationLayer` | `inceptionv3` | `analyzeNetwork` | `assembleNetwork` | **Deep Network Designer**

Topics

“Create Simple Deep Learning Network for Classification”
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”

Introduced in R2017b

layernorm

Normalize data across all channels for each observation independently

Syntax

```
dlY = layernorm(dlX,offset,scaleFactor)
dlY = layernorm(dlX,offset,scaleFactor,'DataFormat',FMT)
[dlY] = layernorm( ___ ,Name,Value)
```

Description

The layer normalization operation normalizes the input data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization after the learnable operations, such as LSTM and fully connect operations.

After normalization, the operation shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

The `layernorm` function applies the layer normalization operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply layer normalization within a `layerGraph` object or `Layer` array, use `layerNormalizationLayer`.

`dlY = layernorm(dlX,offset,scaleFactor)` applies the layer normalization operation to the input data `dlX` and transforms it using the specified offset and scale factor.

The function normalizes over the `'S'` (spatial), `'T'` (time), `'C'` (channel), and `'U'` (unspecified) dimensions of `dlX` for each observation in the `'B'` (batch) dimension, independently.

For unformatted input data, use the `'DataFormat'` option.

`dlY = layernorm(dlX,offset,scaleFactor,'DataFormat',FMT)` applies the layer normalization operation to the unformatted `dlarray` object `dlX` with the format specified by `FMT`. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D image input with the format `'SSCB'` (spatial, spatial, channel, batch).

To specify the format of the scale and offset, use the `'ScaleFormat'` and `'OffsetFormat'` options, respectively.

`[dlY] = layernorm( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'Epsilon',1e-4` sets the epsilon value to `1e-4`.

Examples

Apply Layer Normalization

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 10 channels. Specify the format `'CBT'` (channel, batch, time).

```
numChannels = 10;
miniBatchSize = 128;
sequenceLength = 100;

X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×3

    10   128   100
```

```
dims(dlX)
```

```
ans =
'CBT'
```

For per-observation channel-wise layer normalization, initialize the offset and scale with a vector of zeros and ones, respectively.

```
offset = zeros(numChannels,1);
scaleFactor = ones(numChannels,1);
```

Apply the layer normalization operation using the `layernorm` function.

```
dlY = layernorm(dlX,offset,scaleFactor);
```

View the size and the format of the output `dlY`.

```
size(dlY)
```

```
ans = 1×3

    10   128   100
```

```
dims(dlY)
```

```
ans =
'CBT'
```

Apply Element-Wise Layer Normalization

To perform element-wise layer normalization, specify an offset and scale factor with the same size as the input data.

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 10 channels. Specify the format `'CBT'` (channel, batch, time).

```
numChannels = 10;
miniBatchSize = 128;
sequenceLength = 100;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×3

    10   128   100
```

```
dims(dlX)
```

```
ans =
'CBT'
```

For element-wise layer normalization, initialize the offset and scale with an array of zeros and ones, respectively.

```
offset = zeros(numChannels,sequenceLength);
scaleFactor = ones(numChannels,sequenceLength);
```

Apply the layer normalization operation using the `layernorm` function. Specify the offset and scale formats as `'CT'` (channel, time) using the `'OffsetFormat'` and `'ScaleFormat'` options, respectively.

```
dlY = layernorm(dlX,offset,scaleFactor,'OffsetFormat','CT','ScaleFormat','CT');
```

View the size and the format of the output `dlY`.

```
size(dlY)
```

```
ans = 1×3

    10   128   100
```

```
dims(dlY)
```

```
ans =
'CBT'
```

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `dlX` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `'DataFormat'` option. If `dlX` is a numeric array, then either `scaleFactor` or `offset` must be a `dlarray` object.

`dlX` must have a `'C'` (channel) dimension.

`offset` — Offset
`dlarray` | numeric array

Offset β, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

The size and format of the offset depends on the type of transformation.

Task	Description
Channel-wise transformation	Array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`. For channel-wise transformation, if `offset` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).
Element-wise transformation	Array with a `'C'` (channel) dimension with the same size as the `'C'` (channel) dimension of the input `dlX` and zero or the same number of `'S'` (spatial), `'T'` (time), and `'U'` (unspecified) dimensions of the input `dlX`. Each dimension must have size 1 or have sizes matching the corresponding dimensions in the input `dlX`. For any repeated dimensions, for example, multiple `'S'` (spatial) dimensions, the sizes must match the corresponding dimensions in `dlX` or must all be singleton. The software automatically expands any singleton dimensions to match the size of a single observation in the input `dlX`. For element-wise transformation, if `offset` is a numeric array or an unformatted `dlarray`, then you must specify the offset format using the `'OffsetFormat'` option.

`scaleFactor` — Scale factor
`dlarray` | numeric array

Scale factor γ, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

The size and format of the offset depends on the type of transformation.

Task	Description
Channel-wise transformation	Array with one nonsingleton dimension with size matching the size of the `'C'` (channel) dimension of the input `dlX`. For channel-wise transformation, if `scaleFactor` is a formatted `dlarray` object, then the nonsingleton dimension must have label `'C'` (channel).
Element-wise transformation	Array with a `'C'` (channel) dimension with the same size as the `'C'` (channel) dimension of the input `dlX` and zero or the same number of `'S'` (spatial), `'T'` (time), and `'U'` (unspecified) dimensions of the input `dlX`. Each dimension must have size 1 or have sizes matching the corresponding dimensions in the input `dlX`. For any repeated dimensions, for example, multiple `'S'` (spatial) dimensions, the sizes must match the corresponding dimensions in `dlX` or must all be singleton. The software automatically expands any singleton dimensions to match the size of a single observation in the input `dlX`. For element-wise transformation, if `scaleFactor` is a numeric array or an unformatted `dlarray`, then you must specify the scale format using the `'ScaleFormat'` option.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Epsilon',1e-4` sets the variance offset value to `1e-4`.

`DataFormat` — Dimension order of unformatted data

character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)

- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`Epsilon` — Variance offset
`1e-5` (default) | numeric scalar

Variance offset for preventing divide-by-zero errors, specified as the comma-separated pair consisting of `'Epsilon'` and a numeric scalar greater than or equal to `1e-5`.

Data Types: `single` | `double`

`ScaleFormat` — Dimension order of unformatted scale factor
character vector | string scalar

Dimension order of the unformatted scale factor, specified as the comma-separated pair consisting of `'ScaleFormat'` and a character vector or string scalar.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

For layer normalization, the scale factor must have a `"C"` (channel) dimension. You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the label `"T"` (time) at most once. The scale factor must not have a `"B"` (batch) dimension.

You must specify `'ScaleFormat'` for element-wise normalization when `scaleFactor` is a numeric array or an a unformatted `dlarray`.

Example: `'ScaleFormat',"SSCB"`

Data Types: `char` | `string`

`OffsetFormat` — Dimension order of unformatted offset
character vector | string scalar

Dimension order of the unformatted offset, specified as the comma-separated pair consisting of `'OffsetFormat'` and a character vector or string scalar.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel

- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

For layer normalization, the offset must have a `"C"` (channel) dimension. You can specify multiple dimensions labeled `"S"` or `'U'`. You can use the label `"T"` (time) at most once. The offset must not have a `"B"` (batch) dimension.

You must specify `'OffsetFormat'` for element-wise normalization when `offset` is a numeric array or an unformatted `dlarray`.

Example: `'OffsetFormat',"SSCB"`

Data Types: `char` | `string`

Output Arguments

`dlY` — Normalized data

`dlarray`

Normalized data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension labels as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Algorithms

The layer normalization operation normalizes the elements x_i of the input by first calculating the mean μ_L and variance σ_L^2 over the spatial, time, and channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow layer normalization, the layer normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

References

[1] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. “Layer Normalization.” Preprint, submitted July 21, 2016. https://arxiv.org/abs/1607.06450.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `offset`
  - `scaleFactor`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`relu` | `fullyconnect` | `dlconv` | `dlarray` | `dlgradient` | `dlfeval` | `groupnorm` | `batchnorm`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Update Batch Normalization Statistics Using Model Function”
“Train Network Using Model Function”
“Train Network with Multiple Outputs”
“List of Functions with dlarray Support”

Introduced in R2021a

layerNormalizationLayer

Layer normalization layer

Description

A layer normalization layer normalizes a mini-batch of data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization layers after the learnable layers, such as LSTM and fully connected layers.

After normalization, the layer scales the input with a learnable scale factor γ and shifts it by a learnable offset β.

Creation

Syntax

```
layer = layerNormalizationLayer
layer = layerNormalizationLayer(Name,Value)
```

Description

`layer = layerNormalizationLayer` creates a layer normalization layer.

`layer = layerNormalizationLayer(Name,Value)` sets the optional `Epsilon`, “Parameters and Initialization” on page 1-968, “Learning Rate and Regularization” on page 1-969, and `Name` properties using one or more name-value arguments. For example, `layerNormalizationLayer('Name','layernorm')` creates a layer normalization layer with name `'layernorm'`.

Properties

Layer Normalization

`Epsilon` — Constant to add to mini-batch variances
`1e-5` (default) | numeric scalar

Constant to add to the mini-batch variances, specified as a numeric scalar equal to or larger than `1e-5`.

The layer adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`NumChannels` — Number of input channels
`'auto'` (default) | positive integer

Number of input channels, specified as `'auto'` or a positive integer.

This property is always equal to the number of channels of the input to the layer. If `NumChannels` is `'auto'`, then the software automatically determines the correct value for the number of channels at training time.

Parameters and Initialization

`ScaleInitializer` — Function to initialize channel scale factors

`'ones'` (default) | `'narrow-normal'` | function handle

Function to initialize the channel scale factors, specified as one of the following:

- `'ones'` – Initialize the channel scale factors with ones.
- `'zeros'` – Initialize the channel scale factors with zeros.
- `'narrow-normal'` – Initialize the channel scale factors by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel scale factors with a custom function. If you specify a function handle, then the function must be of the form `scale = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel scale factors when the `Scale` property is empty.

Data Types: `char` | `string` | `function_handle`

`OffsetInitializer` — Function to initialize channel offsets

`'zeros'` (default) | `'ones'` | `'narrow-normal'` | function handle

Function to initialize the channel offsets, specified as one of the following:

- `'zeros'` – Initialize the channel offsets with zeros.
- `'ones'` – Initialize the channel offsets with ones.
- `'narrow-normal'` – Initialize the channel offsets by independently sampling from a normal distribution with a mean of zero and standard deviation of 0.01.
- Function handle – Initialize the channel offsets with a custom function. If you specify a function handle, then the function must be of the form `offset = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel offsets when the `Offset` property is empty.

Data Types: `char` | `string` | `function_handle`

`Scale` — Channel scale factors

`[]` (default) | numeric array

Channel scale factors γ, specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network, if `Scale` is nonempty, then `trainNetwork` uses the `Scale` property as the initial value. If `Scale` is empty, then `trainNetwork` uses the initializer specified by `ScaleInitializer`.

At training time, `Scale` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`

- For feature or sequence input, a numeric array of size `NumChannels`-by-1

`Offset` — Channel offsets
`[ ]` (default) | numeric array

Channel offsets β, specified as a numeric array.

The channel offsets are learnable parameters. When you train a network, if `Offset` is nonempty, then `trainNetwork` uses the `Offset` property as the initial value. If `Offset` is empty, then `trainNetwork` uses the initializer specified by `OffsetInitializer`.

At training time, `Offset` is one of the following:

- For 2-D image input, a numeric array of size 1-by-1-by-`NumChannels`
- For 3-D image input, a numeric array of size 1-by-1-by-1-by-`NumChannels`
- For feature or sequence input, a numeric array of size `NumChannels`-by-1

Learning Rate and Regularization

`ScaleLearnRateFactor` — Learning rate factor for scale factors
1 (default) | nonnegative scalar

Learning rate factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the scale factors in a layer. For example, if `ScaleLearnRateFactor` is 2, then the learning rate for the scale factors in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`OffsetLearnRateFactor` — Learning rate factor for offsets
1 (default) | nonnegative scalar

Learning rate factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the offsets in a layer. For example, if `OffsetLearnRateFactor` is 2, then the learning rate for the offsets in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

`ScaleL2Factor` — L_2 regularization factor for scale factors
1 (default) | nonnegative scalar

L_2 regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

`OffsetL2Factor` — L_2 regularization factor for offsets
1 (default) | nonnegative scalar

L_2 regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is `2`, then the L_2 regularization for the offsets in the layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Layer Normalization Layer

Create a layer normalization layer with the name `'layernorm'`.

```
layer = layerNormalizationLayer('Name','layernorm')
```

```
layer =
  LayerNormalizationLayer with properties:

           Name: 'layernorm'
    NumChannels: 'auto'

   Hyperparameters
        Epsilon: 1.0000e-05

   Learnable Parameters
         Offset: []
          Scale: []

  Show all properties
```

Include a layer normalization layer in a `Layer` array.

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

```
layers =
  11x1 Layer array with layers:

     1   ''   Image Input             32x32x3 images with 'zerocenter' normalization
     2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [1  1  1
     3   ''   Layer Normalization     Layer normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [1  1  1
     7   ''   Layer Normalization     Layer normalization
     8   ''   ReLU                    ReLU
     9   ''   Fully Connected         10 fully connected layer
    10   ''   Softmax                 softmax
    11   ''   Classification Output   crossentropyex
```

Algorithms

The layer normalization operation normalizes the elements x_i of the input by first calculating the mean μ_L and variance σ_L^2 over the spatial, time, and channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}},$$

where ϵ is a constant that improves numerical stability when the variance is very small.

To allow for the possibility that inputs with zero mean and unit variance are not optimal for the operations that follow layer normalization, the layer normalization operation further shifts and scales the activations using the transformation

$$y_i = \gamma \widehat{x}_i + \beta,$$

where the offset β and scale factor γ are learnable parameters that are updated during network training.

References

[1] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. “Layer Normalization.” Preprint, submitted July 21, 2016. https://arxiv.org/abs/1607.06450.

See Also

`batchNormalizationLayer` | `trainNetwork` | `trainingOptions` | `reluLayer` | `convolution2dLayer` | `groupNormalizationLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2021a

leakyrelu

Apply leaky rectified linear unit activation

Syntax

```
dlY = leakyrelu(dlX)
dlY = leakyrelu(dlX,scaleFactor)
```

Description

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.

This operation is equivalent to

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

Note This function applies the leaky ReLU operation to `dlarray` data. If you want to apply leaky ReLU activation within a `layerGraph` object or `Layer` array, use the following layer:

- `leakyReluLayer`

`dlY = leakyrelu(dlX)` computes the leaky ReLU activation of the input `dlX` by applying a threshold operation. All values in `dlX` less than zero are multiplied by a default scale factor of `0.01`.

`dlY = leakyrelu(dlX,scaleFactor)` specifies the scale factor for the leaky ReLU operation.

Examples

Apply Leaky ReLU Activation

Use the `leakyrelu` function to scale negative values in the input data.

Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```
height = 12;
width = 12;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the leaky ReLU activation using a scale factor of `0.05` for the negative values in the input.

```
dlY = leakyrelu(dlX,0.05);
```

Input Arguments

dlX — Input data

dlarray

Input data, specified as a formatted dlarray or an unformatted dlarray.

Data Types: single | double

scaleFactor — Scale factor for negative inputs

0.01 (default) | numeric scalar

Scale factor for negative inputs, specified as a numeric scalar. The default value is 0.01.

Data Types: single | double

Output Arguments

dlY — Leaky ReLU activations

dlarray

Leaky ReLU activations, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension format as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument dlX is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

dlarray | dlconv | batchnorm | relu | dlgradient | dlfeval

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

leakyReluLayer

Leaky Rectified Linear Unit (ReLU) layer

Description

A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.

This operation is equivalent to:

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

Creation

Syntax

```
layer = leakyReluLayer
layer = leakyReluLayer(scale)
layer = leakyReluLayer(___,'Name',Name)
```

Description

`layer = leakyReluLayer` returns a leaky ReLU layer.

`layer = leakyReluLayer(scale)` returns a leaky ReLU layer with a scalar multiplier for negative inputs equal to `scale`.

`layer = leakyReluLayer(___,'Name',Name)` returns a leaky ReLU layer and sets the optional `Name` property.

Properties

Leaky ReLU

`Scale` — Scalar multiplier for negative input values
0.01 (default) | numeric scalar

Scalar multiplier for negative input values, specified as a numeric scalar.

Example: `0.4`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Leaky ReLU Layer

Create a leaky ReLU layer with the name `'leaky1'` and a scalar multiplier for negative inputs equal to 0.1.

```
layer = leakyReluLayer(0.1,'Name','leaky1')
```

```
layer =
  LeakyReLULayer with properties:

     Name: 'leaky1'

   Hyperparameters
```

```
    Scale: 0.1000
```

Include a leaky ReLU layer in a `Layer` array.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    leakyReluLayer

    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    leakyReluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  11x1 Layer array with layers:

     1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [0  0  0
     3   ''   Batch Normalization     Batch normalization
     4   ''   Leaky ReLU              Leaky ReLU with scale 0.01
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [0  0  0
     7   ''   Batch Normalization     Batch normalization
     8   ''   Leaky ReLU              Leaky ReLU with scale 0.01
     9   ''   Fully Connected         10 fully connected layer
    10   ''   Softmax                 softmax
    11   ''   Classification Output   crossentropyex
```

References

[1] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." In *Proc. ICML*, vol. 30, no. 1. 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `reluLayer` | `clippedReluLayer` | `swishLayer`

Topics

“Create Simple Deep Learning Network for Classification”

“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2017b

lstm

Long short-term memory

Syntax

```
dlY = lstm(dlX,H0,C0,weights,recurrentWeights,bias)
[dlY,hiddenState,cellState] = lstm(dlX,H0,C0,weights,recurrentWeights,bias)
[___] = lstm(___,'DataFormat',FMT)
```

Description

The long short-term memory (LSTM) operation allows a network to learn long-term dependencies between time steps in time series and sequence data.

Note This function applies the deep learning LSTM operation to `dlarray` data. If you want to apply an LSTM operation within a `layerGraph` object or `Layer` array, use the following layer:

- `lstmLayer`

`dlY = lstm(dlX,H0,C0,weights,recurrentWeights,bias)` applies a long short-term memory (LSTM) calculation to input `dlX` using the initial hidden state `H0`, initial cell state `C0`, and parameters `weights`, `recurrentWeights`, and `bias`. The input `dlX` must be a formatted `dlarray`. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`, except for any `'S'` dimensions.

The `lstm` function updates the cell and hidden states using the hyperbolic tangent function (tanh) as the state activation function. The `lstm` function uses the sigmoid function given by $\sigma(x) = (1 + e^{-x})^{-1}$ as the gate activation function.

`[dlY,hiddenState,cellState] = lstm(dlX,H0,C0,weights,recurrentWeights,bias)` also returns the hidden state and cell state after the LSTM operation.

`[___] = lstm(___,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`, except for any `'S'` dimensions.

Examples

Apply LSTM Operation to Sequence Data

Perform an LSTM operation using three hidden units.

Create the input sequence data as 32 observations with 10 channels and a sequence length of 64

```
numFeatures = 10;
numObservations = 32;
sequenceLength = 64;
```

```
X = randn(numFeatures,numObservations,sequenceLength);
dlX = dlarray(X,'CBT');
```

Create the initial hidden and cell states with three hidden units. Use the same initial hidden state and cell state for all observations.

```
numHiddenUnits = 3;
H0 = zeros(numHiddenUnits,1);
C0 = zeros(numHiddenUnits,1);
```

Create the learnable parameters for the LSTM operation.

```
weights = dlarray(randn(4*numHiddenUnits,numFeatures),'CU');
recurrentWeights = dlarray(randn(4*numHiddenUnits,numHiddenUnits),'CU');
bias = dlarray(randn(4*numHiddenUnits,1),'C');
```

Perform the LSTM calculation

```
[dlY,hiddenState,cellState] = lstm(dlX,H0,C0,weights,recurrentWeights,bias);
```

View the size and dimensions of `dlY`.

```
size(dlY)
```

```
ans = 1×3

     3    32    64
```

```
dlY.dims
```

```
ans =
'CBT'
```

View the size of `hiddenState` and `cellState`.

```
size(hiddenState)
```

```
ans = 1×2

     3    32
```

```
size(cellState)
```

```
ans = 1×2

     3    32
```

Check that the output `hiddenState` is the same as the last time step of output `dlY`.

```
if extractdata(dlY(:,:,end)) == hiddenState
   disp("The hidden state and the last time step are equal.");
else
   disp("The hidden state and the last time step are not equal.")
end
```

```
The hidden state and the last time step are equal.
```

You can use the hidden state and cell state to keep track of the state of the LSTM operation and input further sequential data.

Input Arguments

`dlX` — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using `'DataFormat',FMT`. If `dlX` is a numeric array, at least one of `H0`, `C0`, `weights`, `recurrentWeights`, or `bias` must be a `dlarray`.

`dlX` must contain a sequence dimension labeled `'T'`. If `dlX` has any spatial dimensions labeled `'S'`, they are flattened into the `'C'` channel dimension. If `dlX` does not have a channel dimension, then one is added. If `dlX` has any unspecified dimensions labeled `'U'`, they must be singleton.

Data Types: `single` | `double`

`H0` — Initial hidden state vector

`dlarray` | numeric array

Initial hidden state vector, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `H0` is a formatted `dlarray`, it must contain a channel dimension labeled `'C'` and optionally a batch dimension labeled `'B'` with the same size as the `'B'` dimension of `dlX`. If `H0` does not have a `'B'` dimension, the function uses the same hidden state vector for each observation in `dlX`.

The size of the `'C'` dimension determines the number of hidden units. The size of the `'C'` dimension of `H0` must be equal to the size of the `'C'` dimensions of `C0`.

If `H0` is a not a formatted `dlarray`, the size of the first dimension determines the number of hidden units and must be the same size as the first dimension or the `'C'` dimension of `C0`.

Data Types: `single` | `double`

`C0` — Initial cell state vector

`dlarray` | numeric array

Initial cell state vector, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `C0` is a formatted `dlarray`, it must contain a channel dimension labeled `'C'` and optionally a batch dimension labeled `'B'` with the same size as the `'B'` dimension of `dlX`. If `C0` does not have a `'B'` dimension, the function uses the same cell state vector for each observation in `dlX`.

The size of the `'C'` dimension determines the number of hidden units. The size of the `'C'` dimension of `C0` must be equal to the size of the `'C'` dimensions of `H0`.

If `C0` is a not a formatted `dlarray`, the size of the first dimension determines the number of hidden units and must be the same size as the first dimension or the `'C'` dimension of `H0`.

Data Types: `single` | `double`

weights — Weights
dlarray | numeric array

Weights, specified as a formatted dlarray, an unformatted dlarray, or a numeric array.

Specify weights as a matrix of size 4*NumHiddenUnits-by-InputSize, where NumHiddenUnits is the size of the 'C' dimension of both C0 and H0, and InputSize is the size of the 'C' dimension of dlX multiplied by the size of each 'S' dimension of dlX, where present.

If weights is a formatted dlarray, it must contain a 'C' dimension of size 4*NumHiddenUnits and a 'U' dimension of size InputSize.

Data Types: single | double

recurrentWeights — Recurrent weights
dlarray | numeric array

Recurrent weights, specified as a formatted dlarray, an unformatted dlarray, or a numeric array.

Specify recurrentWeights as a matrix of size 4*NumHiddenUnits-by-NumHiddenUnits, where NumHiddenUnits is the size of the 'C' dimension of both C0 and H0.

If recurrentWeights is a formatted dlarray, it must contain a 'C' dimension of size 4*NumHiddenUnits and a 'U' dimension of size NumHiddenUnits.

Data Types: single | double

bias — Bias
dlarray vector | numeric vector

Bias, specified as a formatted dlarray, an unformatted dlarray, or a numeric array.

Specify bias as a vector of length 4*NumHiddenUnits, where NumHiddenUnits is the size of the 'C' dimension of both C0 and H0.

If bias is a formatted dlarray, the nonsingleton dimension must be labeled with 'C'.

Data Types: single | double

FMT — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string FMT that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', and 'T' at most once.

You must specify 'DataFormat',FMT when the input data is not a formatted dlarray.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`dlY` — LSTM output

`dlarray`

LSTM output, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`, except for any `'S'` dimensions. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

The size of the `'C'` dimension of `dlY` is the same as the number of hidden units, specified by the size of the `'C'` dimension of `H0` or `C0`.

`hiddenState` — Hidden state vector

`dlarray` | numeric array

Hidden state vector for each observation, returned as a `dlarray` or a numeric array with the same data type as `H0`.

If the input `H0` is a formatted `dlarray`, then the output `hiddenState` is a formatted `dlarray` with the format `'CB'`.

`cellState` — Cell state vector

`dlarray` | numeric array

Cell state vector for each observation, returned as a `dlarray` or a numeric array. `cellState` is returned with the same data type as `C0`.

If the input `C0` is a formatted `dlarray`, the output `cellState` is returned as a formatted `dlarray` with the format `'CB'`.

Limitations

- `functionToLayerGraph` does not support the `lstm` function. If you use `functionToLayerGraph` with a function that contains the `lstm` operation, the resulting `LayerGraph` contains placeholder layers.

More About

Long Short-Term Memory

The LSTM operation allows a network to learn long-term dependencies between time steps in time series and sequence data. For more information, see the definition of Long Short-Tem Memory Layer on page 1-1000 on the `lstmLayer` reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `H0`
  - `C0`
  - `weights`
  - `recurrentWeights`
  - `bias`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `fullyconnect` | `softmax` | `dlgradient` | `dlfeval` | `gru`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Sequence-to-Sequence Translation Using Attention”
“Multilabel Text Classification Using Deep Learning”
“List of Functions with dlarray Support”

Introduced in R2019b

lstmLayer

Long short-term memory (LSTM) layer

Description

An LSTM layer learns long-term dependencies between time steps in time series and sequence data.

The layer performs additive interactions, which can help improve gradient flow over long sequences during training.

Creation

Syntax

```
layer = lstmLayer(numHiddenUnits)
layer = lstmLayer(numHiddenUnits,Name,Value)
```

Description

`layer = lstmLayer(numHiddenUnits)` creates an LSTM layer and sets the `NumHiddenUnits` property.

`layer = lstmLayer(numHiddenUnits,Name,Value)` sets additional `OutputMode`, “Activations” on page 1-986, “State” on page 1-987, “Parameters and Initialization” on page 1-987, “Learning Rate and Regularization” on page 1-990, and `Name` properties using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.

Properties

LSTM

`NumHiddenUnits` — Number of hidden units

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data. This value can vary from a few dozen to a few thousand.

The hidden state does not limit the number of time steps that are processed in an iteration. To split your sequences into smaller sequences for training, use the `'SequenceLength'` option in `trainingOptions`.

Example: 200

`OutputMode` — Output mode
`'sequence'` (default) | `'last'`

Output mode, specified as one of the following:

- `'sequence'` – Output the complete sequence.
- `'last'` – Output the last time step of the sequence.

`HasStateInputs` — Flag for state inputs to layer
`0` (false) (default) | `1` (true)

Flag for state inputs to the layer, specified as `0` (false) or `1` (true).

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

`HasStateOutputs` — Flag for state outputs from layer
`0` (false) (default) | `1` (true)

Flag for state outputs from the layer, specified as `0` (false) or `1` (true).

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

`InputSize` — Input size
`'auto'` (default) | positive integer

Input size, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically assigns the input size at training time.

Example: 100

Activations

`StateActivationFunction` — Activation function to update the cell and hidden state
`'tanh'` (default) | `'softsign'`

Activation function to update the cell and hidden state, specified as one of the following:

- `'tanh'` – Use the hyperbolic tangent function (tanh).
- `'softsign'` – Use the softsign function $\text{softsign}(x) = \frac{x}{1 + |x|}$.

The layer uses this option as the function σ_c in the calculations to update the cell and hidden state. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1000.

`GateActivationFunction` — Activation function to apply to the gates

`'sigmoid'` (default) | `'hard-sigmoid'`

Activation function to apply to the gates, specified as one of the following:

- `'sigmoid'` – Use the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$.
- `'hard-sigmoid'` – Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if} -2.5 \le x \le 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}.$$

The layer uses this option as the function σ_g in the calculations for the layer gates.

State

`CellState` — Cell state

numeric vector

Cell state to use in the layer operation, specified as a `NumHiddenUnits`-by-1 numeric vector. This value corresponds to the initial cell state when data is passed to the layer.

After setting this property manually, calls to the `resetState` function set the cell state to this value.

If `HasStateInputs` is `true`, then the `CellState` property must be empty.

`HiddenState` — Hidden state

numeric vector

Hidden state to use in the layer operation, specified as a `NumHiddenUnits`-by-1 numeric vector. This value corresponds to the initial hidden state when data is passed to the layer.

After setting this property manually, calls to the `resetState` function set the hidden state to this value.

If `HasStateInputs` is `true`, then the `HiddenState` property must be empty.

Parameters and Initialization

`InputWeightsInitializer` — Function to initialize input weights

`'glorot'` (default) | `'he'` | `'orthogonal'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the input weights, specified as one of the following:

- `'glorot'` – Initialize the input weights with the Glorot initializer [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(InputSize + numOut)`, where `numOut = 4*NumHiddenUnits`.
- `'he'` – Initialize the input weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance `2/InputSize`.

- `'orthogonal'` – Initialize the input weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [6]
- `'narrow-normal'` – Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the input weights with zeros.
- `'ones'` – Initialize the input weights with ones.
- Function handle – Initialize the input weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the input weights.

The layer only initializes the input weights when the `InputWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`RecurrentWeightsInitializer` — Function to initialize recurrent weights
`'orthogonal'` (default) | `'glorot'` | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the recurrent weights, specified as one of the following:

- `'orthogonal'` – Initialize the recurrent weights with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. [6]
- `'glorot'` – Initialize the recurrent weights with the Glorot initializer [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = NumHiddenUnits` and `numOut = 4*NumHiddenUnits`.
- `'he'` – Initialize the recurrent weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance `2/NumHiddenUnits`.
- `'narrow-normal'` – Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the recurrent weights with zeros.
- `'ones'` – Initialize the recurrent weights with ones.
- Function handle – Initialize the recurrent weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the recurrent weights.

The layer only initializes the recurrent weights when the `RecurrentWeights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'unit-forget-gate'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'unit-forget-gate'` – Initialize the forget gate bias with ones and the remaining biases with zeros.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'ones'` – Initialize the bias with ones.

- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`InputWeights` — Input weights
[ ] (default) | matrix

Input weights, specified as a matrix.

The input weight matrix is a concatenation of the four input weight matrices for the components (gates) in the LSTM layer. The four matrices are concatenated vertically in the following order:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

The input weights are learnable parameters. When training a network, if `InputWeights` is nonempty, then `trainNetwork` uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then `trainNetwork` uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is a `4*NumHiddenUnits`-by-`InputSize` matrix.

`RecurrentWeights` — Recurrent weights
[ ] (default) | matrix

Recurrent weights, specified as a matrix.

The recurrent weight matrix is a concatenation of the four recurrent weight matrices for the components (gates) in the LSTM layer. The four matrices are vertically concatenated in the following order:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

The recurrent weights are learnable parameters. When training a network, if `RecurrentWeights` is nonempty, then `trainNetwork` uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then `trainNetwork` uses the initializer specified by `RecurrentWeightsInitializer`.

At training time `RecurrentWeights` is a `4*NumHiddenUnits`-by-`NumHiddenUnits` matrix.

`Bias` — Layer biases
[ ] (default) | numeric vector

Layer biases for the LSTM layer, specified as a numeric vector.

The bias vector is a concatenation of the four bias vectors for the components (gates) in the LSTM layer. The four vectors are concatenated vertically in the following order:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a `4*NumHiddenUnits`-by-1 numeric vector.

Learning Rate and Regularization

`InputWeightsLearnRateFactor` — Learning rate factor for input weights
1 (default) | numeric scalar | 1-by-4 numeric vector

Learning rate factor for the input weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input weights of the layer. For example, if `InputWeightsLearnRateFactor` is 2, then the learning rate factor for the input weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the four individual matrices in `InputWeights`, specify a 1-by-4 vector. The entries of `InputWeightsLearnRateFactor` correspond to the learning rate factor of the following:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`RecurrentWeightsLearnRateFactor` — Learning rate factor for recurrent weights
1 (default) | numeric scalar | 1-by-4 numeric vector

Learning rate factor for the recurrent weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the recurrent weights of the layer. For example, if `RecurrentWeightsLearnRateFactor` is 2, then the learning rate for the recurrent weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the four individual matrices in `RecurrentWeights`, specify a 1-by-4 vector. The entries of `RecurrentWeightsLearnRateFactor` correspond to the learning rate factor of the following:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`BiasLearnRateFactor` — Learning rate factor for biases

1 (default) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the biases, specified as a nonnegative scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

To control the value of the learning rate factor for the four individual vectors in `Bias`, specify a 1-by-4 vector. The entries of `BiasLearnRateFactor` correspond to the learning rate factor of the following:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

To specify the same value for all the vectors, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`InputWeightsL2Factor` — L2 regularization factor for input weights

1 (default) | numeric scalar | 1-by-4 numeric vector

L2 regularization factor for the input weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the L2 regularization factor for the input weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `InputWeights`, specify a 1-by-4 vector. The entries of `InputWeightsL2Factor` correspond to the L2 regularization factor of the following:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`RecurrentWeightsL2Factor` — L2 regularization factor for recurrent weights
1 (default) | numeric scalar | 1-by-4 numeric vector

L2 regularization factor for the recurrent weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the L2 regularization factor for the recurrent weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `RecurrentWeights`, specify a 1-by-4 vector. The entries of `RecurrentWeightsL2Factor` correspond to the L2 regularization factor of the following:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`BiasL2Factor` — L2 regularization factor for biases
0 (default) | nonnegative scalar | 1-by-4 numeric vector

L2 regularization factor for the biases, specified as a nonnegative scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual vectors in `Bias`, specify a 1-by-4 vector. The entries of `BiasL2Factor` correspond to the L2 regularization factor of the following:

1. Input gate
2. Forget gate

3 Cell candidate

4 Output gate

To specify the same value for all the vectors, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs

`1` | `3`

Number of inputs of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

Data Types: `double`

`InputNames` — Input names

`{'in'}` | `{'in','hidden','cell'}`

Input names of the layer.

If the `HasStateInputs` property is `0` (false), then the layer has one input with name `'in'`, which corresponds to the input data. In this case, the layer uses the `HiddenState` and `CellState` properties for the layer operation.

If the `HasStateInputs` property is `1` (true), then the layer has three inputs with names `'in'`, `'hidden'`, and `'cell'`, which correspond to the input data, hidden state, and cell state respectively. In this case, the layer uses the values passed to these inputs for the layer operation. If `HasStateInputs` is `1` (true), then the `HiddenState` and `CellState` properties must be empty.

`NumOutputs` — Number of outputs

`1` | `3`

Number of outputs of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` | `{'out','hidden','cell'}`

Output names of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with names `'out'`, `'hidden'`, and `'cell'`, which correspond to the output data, hidden state, and cell state, respectively. In this case, the layer also outputs the state values computed during the layer operation.

Examples

Create LSTM Layer

Create an LSTM layer with the name `'lstm1'` and 100 hidden units.

```
layer = lstmLayer(100,'Name','lstm1')
```

```
layer = 
  LSTMLayer with properties:

                      Name: 'lstm1'
                 InputNames: {'in'}
                OutputNames: {'out'}
                  NumInputs: 1
                 NumOutputs: 1
             HasStateInputs: 0
            HasStateOutputs: 0

   Hyperparameters
                  InputSize: 'auto'
             NumHiddenUnits: 100
                 OutputMode: 'sequence'
    StateActivationFunction: 'tanh'
     GateActivationFunction: 'sigmoid'

   Learnable Parameters
               InputWeights: []
           RecurrentWeights: []
                       Bias: []

   State Parameters
                HiddenState: []
                  CellState: []

  Show all properties
```

Include an LSTM layer in a `Layer` array.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits)
    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
```

Train Network for Sequence Classification

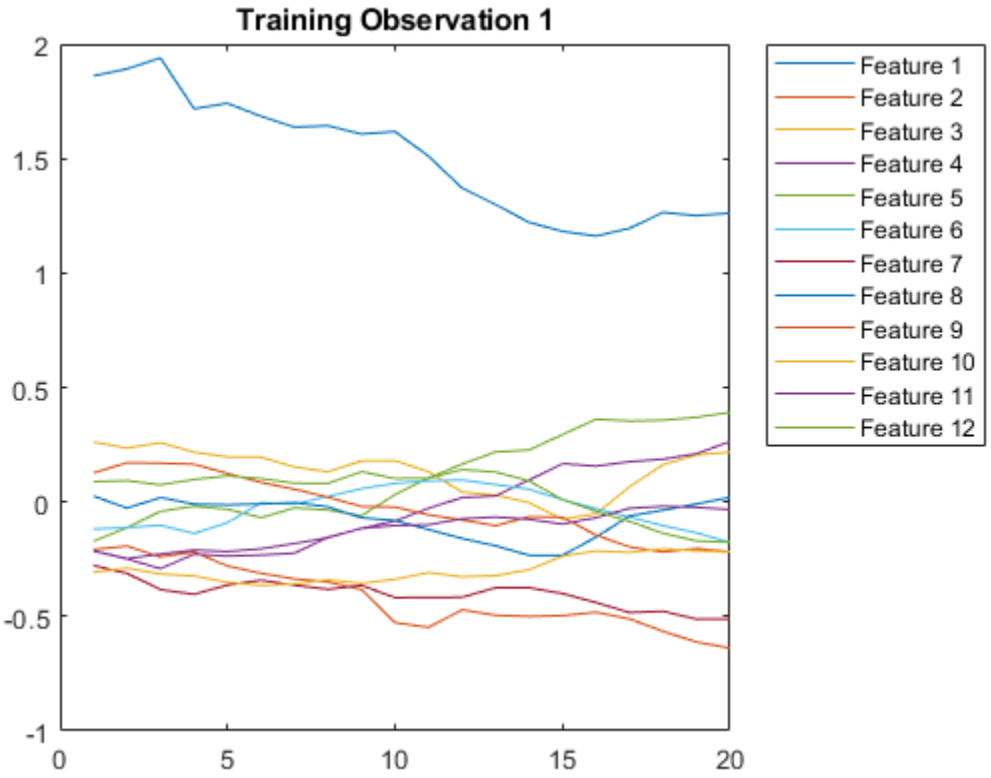
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with 12 features corresponding to LPC cepstrum coefficients. `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;
```

Visualize the first time series in a plot. Each line corresponds to a feature.

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



Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 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.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    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
```

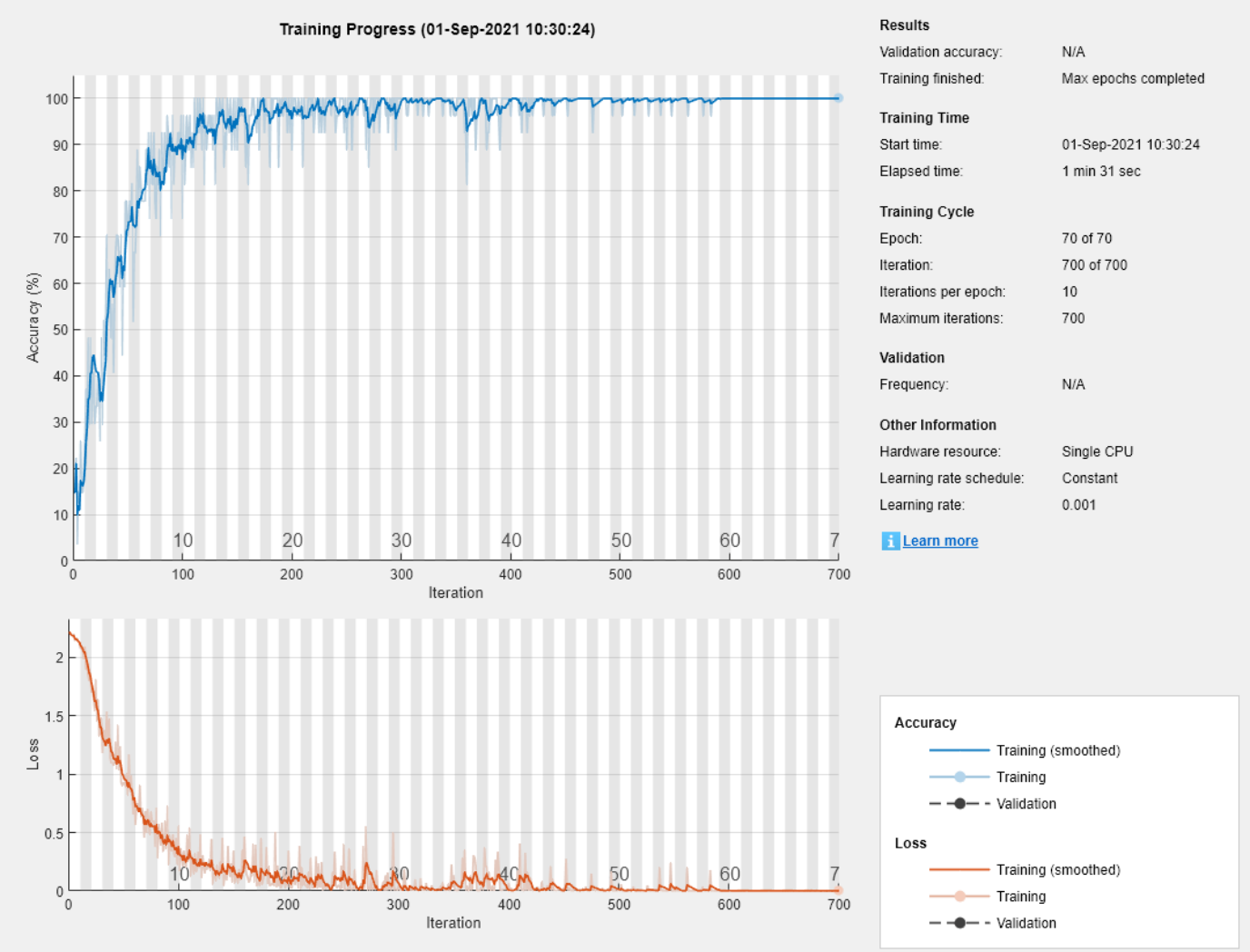
Specify the training options. Specify the solver as `'adam'` and `'GradientThreshold'` as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 70.

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).

```
maxEpochs = 70;
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',1, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the LSTM network with the specified training options.

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



Load the test set and classify the sequences into speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Classify the test data. Specify the same mini-batch size used for training.

```
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9541
```

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”.

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”.

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];
```

Algorithms

Long Short-Term Memory Layer

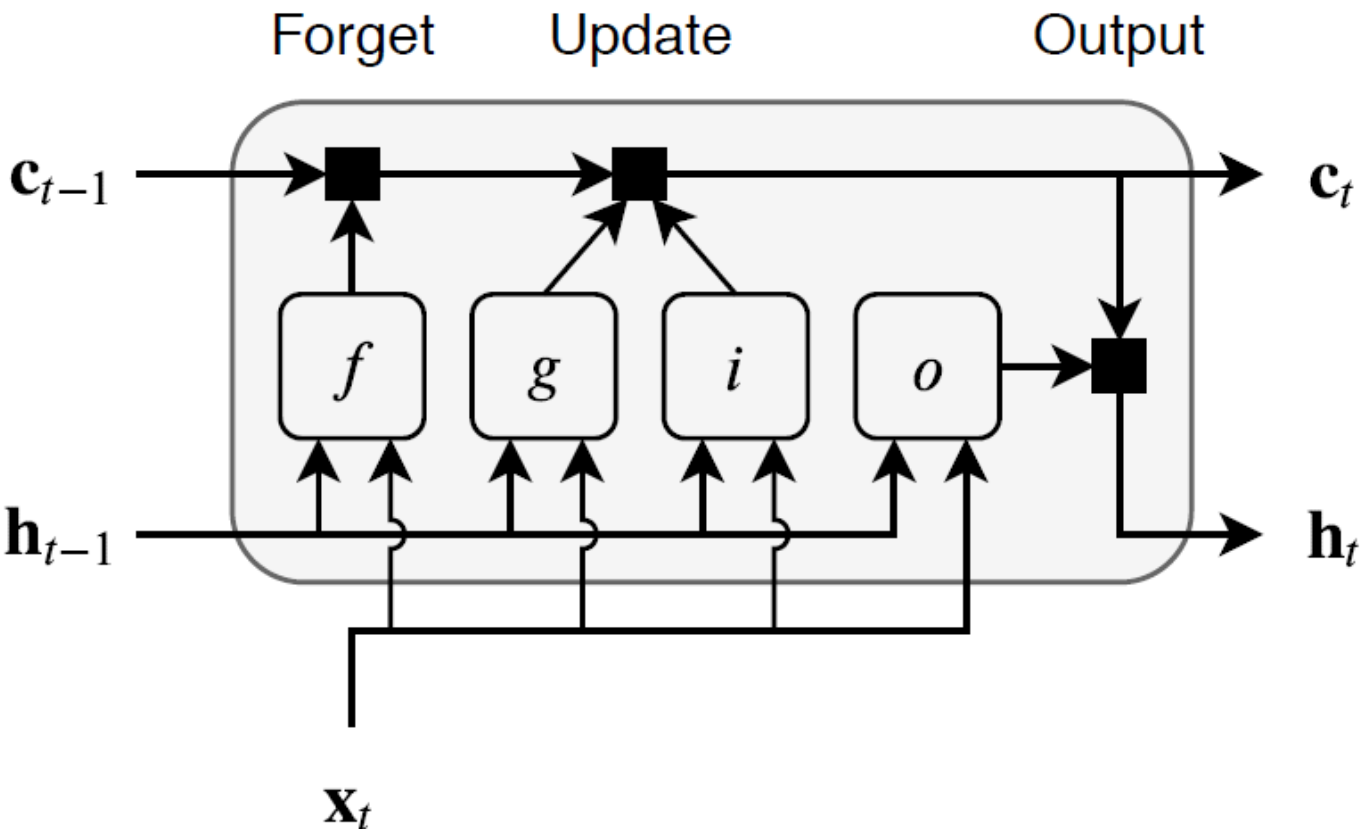
An LSTM layer learns long-term dependencies between time steps in time series and sequence data.

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.

Layer Input and Output Formats

Layers in a layer array or layer graph pass data specified as formatted `dlarray` objects.

You can interact with these `dlarray` objects in automatic differentiation workflows such as when developing a custom layer, using a `functionLayer` object, or using the `forward` and `predict` functions with `dlnetwork` objects.

This table shows the supported input formats of a `LSTMLayer` object and the corresponding output format. If the output of the layer is passed to a custom layer that does not inherit from the `nnet.layer.Formattable` class, or a `FunctionLayer` object with the `Formattable` option set to `false`, then the layer receives an unformatted `dlarray` object with dimensions ordered corresponding to the formats outlined in this table.

Input Format	`OutputMode`	Output Format
`"CBT"` (channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)

In `dlnetwork` objects, `LSTMLayer` objects also support the following input and output format combinations.

Input Format	`OutputMode`	Output Format
`"SCBT"` (spatial, channel, batch)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)
`"SSCBT"` (spatial, spatial, channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)
`"SSSCBT"` (spatial, spatial, spatial, channel, batch, time)	`"sequence"`	`"CBT"` (channel, batch, time)
	`"last"`	`"CB"` (channel, batch)

To use these input formats in `trainNetwork` workflows, first convert the data to `"CBT"` (channel, batch, time) format using `flattenLayer`.

If the `HasStateInputs` property is `1` (true), then the layer has two additional inputs with names `'hidden'` and `'cell'`, which correspond to the hidden state and cell state, respectively. These additional inputs expect input format `"CB"` (channel, batch).

If the `HasStateOutputs` property is `1` (true), then the layer has two additional outputs with names `'hidden'` and `'cell'`, which correspond to the hidden state and cell state, respectively. These additional outputs have output format `"CB"` (channel, batch).

Compatibility Considerations

Default input weights initialization is Glorot

Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer input weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer input weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'InputWeightsInitializer'` option of the layer to `'narrow-normal'`.

Default recurrent weights initialization is orthogonal

Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer recurrent weights of this layer with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer recurrent weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'RecurrentWeightsInitializer'` option of the layer to `'narrow-normal'`.

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

[3] Hochreiter, S, and J. Schmidhuber, 1997. Long short-term memory. *Neural computation*, 9(8), pp.1735–1780.

[4] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[5] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

[6] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." *arXiv preprint arXiv:1312.6120* (2013).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When generating code with Intel MKL-DNN:

- The `StateActivationFunction` property must be set to `'tanh'`.
- The `GateActivationFunction` property must be set to `'sigmoid'`.
- The `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `StateActivationFunction` property must be set to `'tanh'`.
- For GPU code generation, the `GateActivationFunction` property must be set to `'sigmoid'`.
- The `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer` | **Deep Network Designer**

Topics
“Sequence Classification Using Deep Learning”
“Sequence Classification Using 1-D Convolutions”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Classify Videos Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2017b

maxpool

Pool data to maximum value

Syntax

```
dlY = maxpool(dlX,poolsize)
[dlY,indx,inputSize] = maxpool(dlX,poolsize)
dlY = maxpool(dlX,'global')
___ = maxpool( ___ ,'DataFormat',FMT)
___ = maxpool( ___ ,Name,Value)
```

Description

The maximum pooling operation performs downsampling by dividing the input into pooling regions and computing the maximum value of each region.

The `maxpool` function applies the maximum pooling operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply maximum pooling within a `layerGraph` object or `Layer` array, use one of the following layers:

- `maxPooling2dLayer`
- `maxPooling3dLayer`

`dlY = maxpool(dlX,poolsize)` applies the maximum pooling operation to the formatted `dlarray` object `dlX`. The function downsamples the input by dividing it into regions defined by `poolsize` and calculating the maximum value of the data in each region. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`.

The function, by default, pools over up to three dimensions of `dlX` labeled `'S'` (spatial). To pool over dimensions labeled `'T'` (time), specify a pooling region with a `'T'` dimension using the `'PoolFormat'` option.

For unformatted input data, use the `'DataFormat'` option.

`[dlY,indx,inputSize] = maxpool(dlX,poolsize)` also returns the linear indices of the maximum value within each pooled region and the size of the input feature map `dlX` for use with the `maxunpool` function.

`dlY = maxpool(dlX,'global')` computes the global maximum over the spatial dimensions of the input `dlX`. This syntax is equivalent to setting `poolsize` in the previous syntaxes to the size of the `'S'` dimensions of `dlX`.

`___ = maxpool(___,'DataFormat',FMT)` applies the maximum pooling operation to the unformatted `dlarray` object `dlX` with format specified by `FMT` using any of the previous syntaxes. The output `dlY` is an unformatted `dlarray` object with dimensions in the same order as `dlX`. For example, `'DataFormat','SSCB'` specifies data for 2-D maximum pooling with format `'SSCB'` (spatial, spatial, channel, batch).

`___ = maxpool(___,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'PoolFormat','T'` specifies a pooling region for 1-D pooling with format `'T'` (time).

Examples

Perform 2-D Maximum Pooling

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Apply 2-D maximum pooling with 2-by-2 pooling windows using the `maxpool` function.

```
poolSize = [2 2];
dlY = maxpool(dlX,poolSize);
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

    27    27     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Perform 2-D Global Maximum Pooling

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)
```

```
ans =
'SSCB'
```

Apply 2-D global maximum pooling using the `maxpool` function by specifying the `'global'` option.

```
dlY = maxpool(dlX,'global');
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×4

     1     1     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Perform 1-D Maximum Pooling

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 12 channels. Specify the format `'CBT'` (channel, batch, time).

```
miniBatchSize = 128;
sequenceLength = 100;
numChannels = 12;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)

ans = 1×3

    12   128   100


dims(dlX)

ans =
'CBT'
```

Apply 1-D maximum pooling with pooling regions of size 2 with a stride of 2 using the `maxpool` function by specifying the `'PoolFormat'` and `'Stride'` options.

```
poolSize = 2;
dlY = maxpool(dlX,poolSize,'PoolFormat','T','Stride',2);
```

View the size and format of the output.

```
size(dlY)

ans = 1×3

    12   128    50


dims(dlY)

ans =
'CBT'
```

Unpool 2-D Maximum Pooled Data

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)

ans = 1×4

    28    28     3   128


dims(dlX)

ans =
'SSCB'
```

Pool the data to maximum values over pooling regions of size 2 using a stride of 2.

```
[dlY,indx,dataSize] = maxpool(dlX,2,'Stride',2);
```

View the size and format of the pooled data.

```
size(dlY)
```

```
ans = 1×4

    14    14     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

View the data size.

```
dataSize
```

```
dataSize = 1×4

    28    28     3   128
```

Unpool the data using the indices and data size from the `maxpool` operation.

```
dlY = maxunpool(dlY,indx,dataSize);
```

View the size and format of the unpooled data.

```
size(dlY)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlY)
```

```
ans =
'SSCB'
```

Unpool 1-D Maximum Pooled Data

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 12 channels. Specify the format `'CBT'` (channel, batch, time).

```
miniBatchSize = 128;
sequenceLength = 100;
numChannels = 12;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)

ans = 1×3

    12   128   100


dims(dlX)

ans =
'CBT'
```

Apply 1-D maximum pooling with pooling regions of size 2 with a stride of 2 using the `maxpool` function by specifying the `'PoolFormat'` and `'Stride'` options.

```
poolSize = 2;
[dlY,indx,dataSize] = maxpool(dlX,poolSize,'PoolFormat','T','Stride',2);
```

View the size and format of the output.

```
size(dlY)

ans = 1×3

    12   128    50


dims(dlY)

ans =
'CBT'
```

Unpool the data using the indices and data size from the `maxpool` operation.

```
dlY = maxunpool(dlY,indx,dataSize);
```

View the size and format of the unpooled data.

```
size(dlY)

ans = 1×3

    12   128   100


dims(dlY)

ans =
'CBT'
```

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted or unformatted `dlarray` object.

If `dlX` is an unformatted `dlarray`, then you must specify the format using the `'DataFormat'` option.

The function, by default, pools over up to three dimensions of `dlX` labeled `'S'` (spatial). To pool over dimensions labeled `'T'` (time), specify a pooling region with a `'T'` dimension using the `'PoolFormat'` option.

`poolsize` — Size of pooling regions
positive integer | vector of positive integers

Size of the pooling regions, specified as a numeric scalar or numeric vector.

To pool using a pooling region with edges of the same size, specify `poolsize` as a scalar. The pooling regions have the same size along all dimensions specified by `'PoolFormat'`.

To pool using a pooling region with edges of different sizes, specify `poolsize` as a vector, where `poolsize(i)` is the size of corresponding dimension in `'PoolFormat'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Stride',2` specifies the stride of the pooling regions as 2.

`DataFormat` — Dimension order of unformatted data
character vector | string scalar

Dimension order of unformatted input data, specified as a character vector or string scalar `FMT` that provides a label for each dimension of the data.

When you specify the format of a `dlarray` object, each character provides a label for each dimension of the data and must be one of the following:

- `"S"` — Spatial
- `"C"` — Channel
- `"B"` — Batch (for example, samples and observations)
- `"T"` — Time (for example, time steps of sequences)
- `"U"` — Unspecified

You can specify multiple dimensions labeled `"S"` or `"U"`. You can use the labels `"C"`, `"B"`, and `"T"` at most once.

You must specify `DataFormat` when the input data is not a formatted `dlarray`.

Data Types: `char` | `string`

`PoolFormat` — Dimension order of pooling region
character vector | string scalar

Dimension order of the pooling region, specified as the comma-separated pair consisting of `'PoolFormat'` and a character vector or string scalar that provides a label for each dimension of the pooling region.

The default value of 'PoolFormat' depends on the task:

Task	Default
1-D pooling	'S' (spatial)
2-D pooling	'SS' (spatial, spatial)
3-D pooling	'SSS' (spatial, spatial, spatial)

The format must have either no 'S' (spatial) dimensions, or as many 'S' (spatial) dimensions as the input data.

The function, by default, pools over up to three dimensions of dlX labeled 'S' (spatial). To pool over dimensions labeled 'T' (time), specify a pooling region with a 'T' dimension using the 'PoolFormat' option.

Example: 'PoolFormat','T'

Stride — Step size for traversing input data
1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1. If 'Stride' is less than poolsize in any dimension, then the pooling regions overlap.

The Stride parameter is not supported for global pooling using the 'global' option.

Example: 'Stride',3

Data Types: single | double

Padding — Size of padding applied to edges of data
0 (default) | 'same' | numeric scalar | numeric vector | numeric matrix

Size of padding applied to edges of data, specified as the comma-separated pair consisting of 'Padding' and one of the following:

- 'same' — Padding size is set so that the output size is the same as the input size when the stride is 1. More generally, the output size of each spatial dimension is ceil(inputSize/stride), where inputSize is the size of the input along a spatial dimension.
- Numeric scalar — The same amount of padding is applied to both ends of all spatial dimensions.
- Numeric vector — A different amount of padding is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the size of padding applied to the start and the end along the ith spatial dimension.
- Numeric matrix — A different amount of padding is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-d, where d is the number of spatial dimensions of the input data. The element (1,d) specifies the size of padding applied to the start of spatial dimension d. The element (2,d) specifies the size of padding applied to the end of spatial dimension d. For example, in 2-D, the format is [top, left; bottom, right].

The `'Padding'` parameter is not supported for global pooling using the `'global'` option.

Example: `'Padding','same'`

Data Types: `single` | `double`

Output Arguments

`dlY` — Pooled data

`dlarray`

Pooled data, returned as a `dlarray` with the same underlying data type as `dlX`.

If the input data `dlX` is a formatted `dlarray`, then `dlY` has the same format as `dlX`. If the input data is not a formatted `dlarray`, then `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

`indx` — Indices of maximum values

`dlarray`

Indices of maximum values in each pooled region, returned as a `dlarray`. Each value in `indx` represents the location of the corresponding maximum value in `dlY`, given as a linear index of the values in `dlX`.

If `dlX` is a formatted `dlarray`, `indx` has the same size and format as the output `dlY`.

If `dlX` is not a formatted `dlarray`, `indx` is an unformatted `dlarray`. In that case, `indx` is returned with the following dimension order: all `'S'` dimensions, followed by `'C'`, `'B'`, and `'T'` dimensions, then all `'U'` dimensions. The size of `indx` matches the size of `dlY` when `dlY` is permuted to match the previously stated dimension order.

Use the `indx` output with the `maxunpool` function to unpool the output of `maxpool`.

`indx` output is not supported when using the `'global'` option.

`inputSize` — Size of input feature map

numeric vector

Size of the input feature map, returned as a numeric vector.

Use the `inputSize` output with the `maxunpool` function to unpool the output of `maxpool`.

`inputSize` output is not supported when using the `'global'` option.

More About

Maximum Pooling

The `maxpool` function pools the input data to maximum values. For more information, see the definition of “Max Pooling Layer” on page 1-1026 on the `maxPooling2dLayer` reference page.

Compatibility Considerations

`maxpool` indices output argument changes shape and data type

Behavior changed in R2020a

Starting in R2020a, the data type and shape of the indices output argument of the `maxpool` function are changed. The `maxpool` function outputs the indices of the maximum values as a `dlarray` with the same shape and format as the pooled data, instead of a numeric vector.

The indices output of `maxpool` remains compatible with the indices input of `maxunpool`. The `maxunpool` function accepts the indices of the maximum values as a `dlarray` with the same shape and format as the input data. To prevent errors, use only the indices output of the `maxpool` function as the indices input to the `maxunpool` function.

To reproduce the previous behavior and obtain the indices output as a numeric vector, use the following code:

```
[dlY,indx,inputSize] = maxpool(dlY,poolsize);
indx = extractdata(indx);
indx = reshape(indx,[],1);
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`avgpool` | `dlconv` | `maxunpool` | `dlgradient` | `dlfeval` | `dlarray`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

maxPooling1dLayer

1-D max pooling layer

Description

A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

Creation

Syntax

```
layer = maxPooling1dLayer(poolSize)
layer = maxPooling1dLayer(poolSize,Name=Value)
```

Description

`layer = maxPooling1dLayer(poolSize)` creates a 1-D max pooling layer and sets the `PoolSize` property.

`layer = maxPooling1dLayer(poolSize,Name=Value)` also specifies the padding or sets the `Stride` and `Name` properties using one or more optional name-value arguments. For example, `maxPooling1dLayer(3,Padding=1,Stride=2)` creates a 1-D max pooling layer with a pool size of 3, a stride of 2, and padding of size 1 on both the left and right of the input.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `maxPooling1dLayer(2,Padding=1)` creates a 1-D max pooling layer with a pool size of 3 and padding of size 1 on the left and right of the layer input.

`Padding` — Padding to apply to input
`[0 0]` (default) | `"same"` | nonnegative integer | vector of nonnegative integers

Padding to apply to the input, specified as one of the following:

- `"same"` — Apply padding such that the output size is `ceil(inputSize/stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same size as the input.
- Nonnegative integer `sz` — Add padding of size `sz` to both ends of the input.
- Vector `[l r]` of nonnegative integers — Add padding of size `l` to the left and `r` to the right of the input.

Example: `Padding=[2 1]` adds padding of size 2 to the left and size 1 to the right.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

Properties

Max Pooling

`PoolSize` — Width of pooling regions

positive integer

Width of the pooling regions, specified as a positive integer.

The width of the pooling regions `PoolSize` must be greater than or equal to the padding dimensions `PaddingSize`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Stride` — Step size for traversing input

`1` (default) | positive integer

Step size for traversing the input, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PaddingSize` — Size of padding

`[0 0]` (default) | vector of two nonnegative integers

Size of padding to apply to each side of the input, specified as a vector `[l r]` of two nonnegative integers, where `l` is the padding applied to the left and `r` is the padding applied to the right.

When you create a layer, use the `Padding` name-value argument to specify the padding size.

Data Types: `double`

`PaddingMode` — Method to determine padding size

`'manual'` (default) | `'same'`

This property is read-only.

Method to determine padding size, specified as one of the following:

- `'manual'` – Pad using the integer or vector specified by `Padding`.
- `'same'` – Apply padding such that the output size is `ceil(inputSize/Stride)`, where `inputSize` is the length of the input. When `Stride` is `1`, the output is the same as the input.

To specify the layer padding, use the `Padding` name-value argument.

Data Types: `char`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create 1-D Max Pooling Layer

Create a 1-D max pooling layer with a pool size of 3.

```
layer = maxPooling1dLayer(3);
```

Include a 1-D max pooling layer in a layer array.

```
layers = [
    sequenceInputLayer(12)
    convolution1dLayer(11,96)
    reluLayer
    maxPooling1dLayer(3)
    convolution1dLayer(11,96)
    reluLayer
    globalMaxPooling1dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
  10x1 Layer array with layers:

     1   ''   Sequence Input           Sequence input with 12 dimensions
     2   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
     3   ''   ReLU                     ReLU
     4   ''   1-D Max Pooling          Max pooling with pool size 3, stride 1, and padding [0  0
     5   ''   Convolution              96 11 convolutions with stride 1 and padding [0  0]
     6   ''   ReLU                     ReLU
     7   ''   1-D Global Max Pooling   1-D global max pooling
     8   ''   Fully Connected          10 fully connected layer
     9   ''   Softmax                  softmax
    10   ''   Classification Output    crossentropyex
```

Algorithms

1-D Max Pooling Layer

A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region. The layer pools the input by moving the pooling regions along the input horizontally.

The dimension that the layer pools over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps), the layer pools over the spatial dimension.

See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Topics

“Sequence Classification Using 1-D Convolutions”
“Sequence-to-Sequence Classification Using 1-D Convolutions”
“Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”

“Sequence-to-Sequence Regression Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Long Short-Term Memory Networks”
“List of Deep Learning Layers”
“Deep Learning Tips and Tricks”

Introduced in R2021b

maxPooling2dLayer

Max pooling layer

Description

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

Creation

Syntax

```
layer = maxPooling2dLayer(poolSize)
layer = maxPooling2dLayer(poolSize,Name,Value)
```

Description

`layer = maxPooling2dLayer(poolSize)` creates a max pooling layer and sets the `PoolSize` property.

`layer = maxPooling2dLayer(poolSize,Name,Value)` sets the optional `Stride`, `Name`, and `HasUnpoolingOutputs` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `maxPooling2dLayer(2,'Stride',3)` creates a max pooling layer with pool size `[2 2]` and stride `[3 3]`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input and to set the `Stride`, `Name`, and `HasUnpoolingOutputs` properties. Enclose names in single quotes.

Example: `maxPooling2dLayer(2,'Stride',3)` creates a max pooling layer with pool size `[2 2]` and stride `[3 3]`.

`Padding` — Input edge padding
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If

the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Vector `[a b]` of nonnegative integers — Add padding of size `a` to the top and bottom of the input and padding of size `b` to the left and right.
- Vector `[t b l r]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, and `r` to the right of the input.

Example: `'Padding',1` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Max Pooling

`PoolSize` — Dimensions of pooling regions

vector of two positive integers

Dimensions of the pooling regions, specified as a vector of two positive integers `[h w]`, where `h` is the height and `w` is the width. When creating the layer, you can specify `PoolSize` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 1]` specifies pooling regions of height 2 and width 1.

`Stride` — Step size for traversing input

`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector of two positive integers `[a b]`, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`PaddingSize` — Size of padding

`[0 0 0 0]` (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector `[t b l r]` of four nonnegative integers, where `t` is the padding applied to the top, `b` is the padding applied to the bottom, `l` is the padding applied to the left, and `r` is the padding applied to the right.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 1 2 2]` adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the `'Padding'` value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height or width of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

`Padding` — Size of padding
`[0 0]` (default) | vector of two nonnegative integers

Note `Padding` property will be removed in a future release. Use `PaddingSize` instead. When creating a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector `[a b]` of two nonnegative integers, where `a` is the padding applied to the top and bottom of the input data and `b` is the padding applied to the left and right.

Example: `[1 1]` adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

`HasUnpoolingOutputs` — Flag for outputs to unpooling layer
`false` (default) | `true`

Flag for outputs to unpooling layer, specified as `true` or `false`.

If the `HasUnpoolingOutputs` value equals `false`, then the max pooling layer has a single output with the name `'out'`.

To use the output of a max pooling layer as the input to a max unpooling layer, set the `HasUnpoolingOutputs` value to `true`. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- `'indices'` — Indices of the maximum value in each pooled region.
- `'size'` — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.

For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs

1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names

`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs

1 (default) | 3

Number of outputs of the layer.

If the `HasUnpoolingOutputs` value equals `false`, then the max pooling layer has a single output with the name `'out'`.

To use the output of a max pooling layer as the input to a max unpooling layer, set the `HasUnpoolingOutputs` value to `true`. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- `'indices'` — Indices of the maximum value in each pooled region.
- `'size'` — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.

For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

Data Types: `double`

`OutputNames` — Output names

`{'out'}` (default) | `{'out','indices','size'}`

Output names of the layer.

If the `HasUnpoolingOutputs` value equals `false`, then the max pooling layer has a single output with the name `'out'`.

To use the output of a max pooling layer as the input to a max unpooling layer, set the `HasUnpoolingOutputs` value to `true`. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- `'indices'` — Indices of the maximum value in each pooled region.
- `'size'` — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.

For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

Data Types: `cell`

Examples

Create Max Pooling Layer with Nonoverlapping Pooling Regions

Create a max pooling layer with nonoverlapping pooling regions.

```
layer = maxPooling2dLayer(2,'Stride',2)
```

```
layer =
  MaxPooling2DLayer with properties:

                  Name: ''
    HasUnpoolingOutputs: 0
             NumOutputs: 1
            OutputNames: {'out'}

   Hyperparameters
               PoolSize: [2 2]
                 Stride: [2 2]
            PaddingMode: 'manual'
            PaddingSize: [0 0 0 0]
```

The height and the width of the rectangular regions (pool size) are both 2. The pooling regions do not overlap because the step size for traversing the images vertically and horizontally (stride) is also `[2 2]`.

Include a max pooling layer with nonoverlapping regions in a `Layer` array.

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

```
    softmaxLayer
    classificationLayer]

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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Max Pooling Layer with Overlapping Pooling Regions

Create a max pooling layer with overlapping pooling regions.

```
layer = maxPooling2dLayer([3 2],'Stride',2)

layer =
  MaxPooling2DLayer with properties:

                   Name: ''
    HasUnpoolingOutputs: 0
             NumOutputs: 1
            OutputNames: {'out'}

   Hyperparameters
              PoolSize: [3 2]
                Stride: [2 2]
           PaddingMode: 'manual'
           PaddingSize: [0 0 0 0]
```

This layer creates pooling regions of size [3 2] and takes the maximum of the six elements in each region. The pooling regions overlap because there are stride dimensions `Stride` that are less than the respective pooling dimensions `PoolSize`.

Include a max pooling layer with overlapping pooling regions in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer([3 2],'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

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  0
     3   ''   ReLU                    ReLU
```

```
4   ''   Max Pooling             3x2 max pooling with stride [2  2] and padding [0  0  0  0]
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex
```

More About

Max Pooling Layer

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

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.

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* [1]. 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.

References

[1] 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`averagePooling2dLayer` | `globalAveragePooling2dLayer` | `convolution2dLayer` | `maxUnpooling2dLayer`

Topics

“Create Simple Deep Learning Network for Classification”

“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

maxPooling3dLayer

3-D max pooling layer

Description

A 3-D max pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the maximum of each region.

Creation

Syntax

```
layer = maxPooling3dLayer(poolSize)
layer = maxPooling3dLayer(poolSize,Name,Value)
```

Description

`layer = maxPooling3dLayer(poolSize)` creates a 3-D max pooling layer and sets the `PoolSize` property.

`layer = maxPooling3dLayer(poolSize,Name,Value)` sets the optional `Stride` and `Name` properties using name-value pairs. To specify input padding, use the `'Padding'` name-value pair argument. For example, `maxPooling3dLayer(2,'Stride',3)` creates a 3-D max pooling layer with pool size `[2 2 2]` and stride `[3 3 3]`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input and to set the `Stride` and `Name` properties. Enclose names in single quotes.

Example: `maxPooling3dLayer(2,'Stride',3)` creates a 3-D max pooling layer with pool size `[2 2 2]` and stride `[3 3 3]`.

`Padding` — Input edge padding
`0` (default) | array of nonnegative integers | `'same'`

Input edge padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to

the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Three-element vector `[a b c]` of nonnegative integers — Add padding of size `a` to the top and bottom, padding of size `b` to the left and right, and padding of size `c` to the front and back of the input.
- 2-by-3 matrix `[t l f;b r k]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, `r` to the right, `f` to the front, and `k` to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: `'Padding',1` adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Max Pooling

`PoolSize` — Dimensions of pooling regions

vector of three positive integers

Dimensions of the pooling regions, specified as a vector of three positive integers `[h w d]`, where `h` is the height, `w` is the width, and `d` is the depth. When creating the layer, you can specify `PoolSize` as a scalar to use the same value for all three dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 1 1]` specifies pooling regions of height 2, width 1, and depth 1.

`Stride` — Step size for traversing input

`[1 1 1]` (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector `[a b c]` of three positive integers, where `a` is the vertical step size, `b` is the horizontal step size, and `c` is the step size along the depth direction. When creating the layer, you can specify `Stride` as a scalar to use the same value for step sizes in all three directions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: `[2 3 1]` specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

`PaddingSize` — Size of padding

`[0 0 0;0 0 0]` (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix `[t l f;b r k]` of nonnegative integers, where `t` and `b` are the padding applied to the top and bottom in the vertical direction, `l` and `r` are the padding applied to the left and right in the horizontal direction, and `f` and `k` are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the `'Padding'` name-value pair argument to specify the padding size.

Example: `[1 2 4;1 2 4]` adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

`PaddingMode` — Method to determine padding size
`'manual'` (default) | `'same'`

Method to determine padding size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `PaddingMode` based on the 'Padding' value you specify when creating a layer.

- If you set the `'Padding'` option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to `'manual'`.
- If you set the `'Padding'` option to `'same'`, then the software automatically sets `PaddingMode` to `'same'` and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Max Pooling 3-D Layer with Nonoverlapping Pooling Regions

Create a max pooling 3-D layer with nonoverlapping pooling regions.

```
layer = maxPooling3dLayer(2,'Stride',2)
```

```
layer =
  MaxPooling3DLayer with properties:

           Name: ''
     NumOutputs: 1
    OutputNames: {'out'}

   Hyperparameters
       PoolSize: [2 2 2]
         Stride: [2 2 2]
    PaddingMode: 'manual'
    PaddingSize: [2x3 double]
```

The height, width, and depth of the cuboidal regions (pool size) are 2. The step size for traversing the images (stride) is 2 in all dimensions. The pooling regions do not overlap because the stride is greater than or equal to the corresponding pool size in all dimensions.

Include a max pooling layer with nonoverlapping regions in a `Layer` array.

```
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    maxPooling3dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
```

```
    softmaxLayer
    classificationLayer]

layers =
  7x1 Layer array with layers:

     1   ''   3-D Image Input         28x28x28x3 images with 'zerocenter' normalization
     2   ''   Convolution             20 5x5x5 convolutions with stride [1  1  1] and padding [0
     3   ''   ReLU                    ReLU
     4   ''   3-D Max Pooling         2x2x2 max pooling with stride [2  2  2] and padding [0  0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Max Pooling 3-D Layer with Overlapping Pooling Regions

Create a max pooling 3-D layer with overlapping pooling regions and padding for the top and bottom of the input.

```
layer = maxPooling3dLayer([3 2 2],'Stride',2,'Padding',[1 0 0])

layer =
  MaxPooling3DLayer with properties:

           Name: ''
     NumOutputs: 1
    OutputNames: {'out'}

   Hyperparameters
       PoolSize: [3 2 2]
         Stride: [2 2 2]
    PaddingMode: 'manual'
    PaddingSize: [2x3 double]
```

This layer creates pooling regions of size 3-by-2-by-2 and takes the maximum of the twelve elements in each region. The stride is 2 in all dimensions. The pooling regions overlap because there are stride dimensions `Stride` that are less than the respective pooling dimensions `PoolSize`.

More About

3-D Max Pooling Layer

A 3-D max pooling layer extends the functionality of a max pooling layer to a third dimension, depth. A max pooling layer performs down-sampling by dividing the input into rectangular or cuboidal pooling regions, and computing the maximum of each region. To learn more, see the definition of max pooling layer on page 1-1026 on the `maxPooling2dLayer` reference page.

See Also

`maxPooling2dLayer` | `globalAveragePooling3dLayer` | `convolution3dLayer` | `averagePooling3dLayer`

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a

maxunpool

Unpool the output of a maximum pooling operation

Syntax

```
dlY = maxunpool(dlX,indx,outputSize)
dlY = maxunpool(dlX,indx,outputSize,'DataFormat',FMT)
```

Description

The maximum unpooling operation unpools the output of a maximum pooling operation by upsampling and padding with zeros.

The `maxunpool` function applies the maximum unpooling operation to `dlarray` data. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Note To apply maximum unpooling within a `layerGraph` object or `Layer` array, use `maxUnpooling2dLayer`.

`dlY = maxunpool(dlX,indx,outputSize)` upsamples the spatial or time dimensions of input data `dlX` to match the size `outputSize`. The data is padded with zeros between the locations of maximum values specified by `indx`. The input `dlX` is a formatted `dlarray` with dimension labels. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`.

`dlY = maxunpool(dlX,indx,outputSize,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`.

Examples

Unpool 2-D Maximum Pooled Data

Create a formatted `dlarray` object containing a batch of 128 28-by-28 images with 3 channels. Specify the format `'SSCB'` (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
dlX = dlarray(X,'SSCB');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×4

    28    28     3   128
```

```
dims(dlX)

ans =
'SSCB'
```

Pool the data to maximum values over pooling regions of size 2 using a stride of 2.

```
[dlY,indx,dataSize] = maxpool(dlX,2,'Stride',2);
```

View the size and format of the pooled data.

```
size(dlY)

ans = 1×4

    14    14     3   128
```

```
dims(dlY)

ans =
'SSCB'
```

View the data size.

```
dataSize

dataSize = 1×4

    28    28     3   128
```

Unpool the data using the indices and data size from the `maxpool` operation.

```
dlY = maxunpool(dlY,indx,dataSize);
```

View the size and format of the unpooled data.

```
size(dlY)

ans = 1×4

    28    28     3   128
```

```
dims(dlY)

ans =
'SSCB'
```

Unpool 1-D Maximum Pooled Data

Create a formatted `dlarray` object containing a batch of 128 sequences of length 100 with 12 channels. Specify the format `'CBT'` (channel, batch, time).

```
miniBatchSize = 128;
sequenceLength = 100;
numChannels = 12;
X = rand(numChannels,miniBatchSize,sequenceLength);
dlX = dlarray(X,'CBT');
```

View the size and format of the input data.

```
size(dlX)
```

```
ans = 1×3

    12   128   100
```

```
dims(dlX)
```

```
ans =
'CBT'
```

Apply 1-D maximum pooling with pooling regions of size 2 with a stride of 2 using the `maxpool` function by specifying the `'PoolFormat'` and `'Stride'` options.

```
poolSize = 2;
[dlY,indx,dataSize] = maxpool(dlX,poolSize,'PoolFormat','T','Stride',2);
```

View the size and format of the output.

```
size(dlY)
```

```
ans = 1×3

    12   128    50
```

```
dims(dlY)
```

```
ans =
'CBT'
```

Unpool the data using the indices and data size from the `maxpool` operation.

```
dlY = maxunpool(dlY,indx,dataSize);
```

View the size and format of the unpooled data.

```
size(dlY)
```

```
ans = 1×3

    12   128   100
```

```
dims(dlY)
```

```
ans =
'CBT'
```

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted or unformatted `dlarray` object.

If `dlX` is an unformatted `dlarray`, then you must specify the format using the `'DataFormat'` option.

The function, unpools the `'S'` (spatial) and `'T'` dimensions of the data to have sizes given by `outputSize`.

`indx` — Indices of maximum values

`dlarray`

Indices of maximum values in each pooled region, specified as a `dlarray`.

Use the indices output of the `maxpool` function as the `indx` input to `maxunpool`.

`outputSize` — Size of output feature map

numeric array

Size of the output feature map, specified as a numeric array.

Use the size output of the `maxpool` function as the `outputSize` input to `maxunpool`.

`FMT` — Dimension order of unformatted data

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

dlY — Unpooled data

`dlarray`

Unpooled data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, then `dlY` has the same dimension format as `dlX`. If the input data is not a formatted `dlarray`, then `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`maxpool` | `dlarray` | `dlgradient` | `dlfeval`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

maxUnpooling2dLayer

Max unpooling layer

Description

A 2-D max unpooling layer unpools the output of a 2-D max pooling layer.

Creation

Syntax

```
layer = maxUnpooling2dLayer
layer = maxUnpooling2dLayer('Name',name)
```

Description

`layer = maxUnpooling2dLayer` creates a max unpooling layer.

`layer = maxUnpooling2dLayer('Name',name)` sets the `Name` property. To create a network containing a max unpooling layer you must specify a layer name.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
3 (default)

Number of inputs of the layer.

There are three inputs to this layer:

- `'in'` — Input feature map to unpool.
- `'indices'` — Indices of the maximum value in each pooled region. This is output by the max pooling layer.
- `'size'` — Output size of unpooled feature map. This is output by the max pooling layer.

Use the input names when connecting or disconnecting the max unpooling layer to other layers using `connectLayers` or `disconnectLayers`, respectively.

Data Types: `double`

`InputNames` — Input names
`{'in','indices','size'}` (default)

Input names of the layer.

There are three inputs to this layer:

- `'in'` — Input feature map to unpool.
- `'indices'` — Indices of the maximum value in each pooled region. This is output by the max pooling layer.
- `'size'` — Output size of unpooled feature map. This is output by the max pooling layer.

Use the input names when connecting or disconnecting the max unpooling layer to other layers using `connectLayers` or `disconnectLayers`, respectively.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Max Unpooling Layer

Create a max unpooling layer that unpools the output of a max pooling layer.

```
layer = maxUnpooling2dLayer
```

```
layer =
  MaxUnpooling2DLayer with properties:

         Name: ''
    NumInputs: 3
   InputNames: {'in'  'indices'  'size'}
```

Unpool Max Pooling Layer

Create a max pooling layer, and set the `'HasUnpoolingOutputs'` property as `true`. This property gives the max pooling layer two additional outputs, `'indices'` and `'size'`, which enables unpooling the layer. Also create a max unpooling layer.

```
layers = [
    maxPooling2dLayer(2,'Stride',2,'Name','mpool','HasUnpoolingOutputs',true)
    maxUnpooling2dLayer('Name','unpool');
]
```

```
layers =
  2x1 Layer array with layers:

     1   'mpool'    Max Pooling     2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     2   'unpool'   Max Unpooling   Max Unpooling
```

Sequentially connect layers by adding them to a `layerGraph`. This step connects the `'out'` output of the max pooling layer to the `'in'` input of the max unpooling layer.

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

```
lgraph =
  LayerGraph with properties:

         Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [1x2 table]
     InputNames: {1x0 cell}
    OutputNames: {1x0 cell}
```

Unpool the output of the max pooling layer, by connecting the max pooling layer outputs to the max unpooling layer inputs.

```
lgraph = connectLayers(lgraph,'mpool/indices','unpool/indices');
lgraph = connectLayers(lgraph,'mpool/size','unpool/size');
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`maxPooling2dLayer` | `connectLayers` | `disconnectLayers` | `layerGraph` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”

Introduced in R2017b

minibatchqueue

Create mini-batches for deep learning

Description

Use a `minibatchqueue` object to create, preprocess, and manage mini-batches of data for training using custom training loops.

A `minibatchqueue` object iterates over a datastore to provide data in a suitable format for training using custom training loops. The object prepares a queue of mini-batches that are preprocessed on demand. Use a `minibatchqueue` object to automatically convert your data to `dlarray` or `gpuArray`, convert data to a different precision, or apply a custom function to preprocess your data. You can prepare your data in parallel in the background.

During training, you can manage your data using the `minibatchqueue` object. You can shuffle the data at the start of each training epoch using the `shuffle` function and collect data from the queue for each training iteration using the `next` function. You can check if any data is left in the queue using the `hasdata` function, and `reset` the queue when it is empty.

Creation

Syntax

```
mbq = minibatchqueue(ds)
mbq = minibatchqueue(ds,numOutputs)
mbq = minibatchqueue( ___ ,Name,Value)
```

Description

`mbq = minibatchqueue(ds)` creates a `minibatchqueue` object from the input datastore `ds`. The mini-batches in `mbq` have the same number of variables as the results of `read` on the input datastore.

`mbq = minibatchqueue(ds,numOutputs)` creates a `minibatchqueue` object from the input datastore `ds` and sets the number of variables in each mini-batch. Use this syntax when you use `MiniBatchFcn` to specify a mini-batch preprocessing function that has a different number of outputs than the number of variables of the input datastore `ds`.

`mbq = minibatchqueue( ___ ,Name,Value)` sets one or more properties using name-value options. For example, `minibatchqueue(ds, "MiniBatchSize",64,"PartialMiniBatches","discard")` sets the size of the returned mini-batches to 64 and discards any mini-batches with fewer than 64 observations.

Input Arguments

`ds` — Input datastore
datastore | custom datastore

Input datastore, specified as a MATLAB datastore or a custom datastore.

For more information about datastores for deep learning, see “Datastores for Deep Learning”.

`numOutputs` — Number of mini-batch variables
positive integer

Number of mini-batch variables, specified as a positive integer. By default, the number of mini-batch variables is equal to the number of variables of the input datastore.

You can determine the number of variables of the input datastore by examining the output of `read(ds)`. If your datastore returns a table, the number of variables is the number of variables of the table. If your datastore returns a cell array, the number of variables is the size of the second dimension of the cell array.

If you use the `MiniBatchFcn` name-value argument to specify a mini-batch preprocessing function that returns a different number of variables than the input datastore, you must set `numOutputs` to match the number of outputs of the function.

Example: 2

Properties

`MiniBatchSize` — Size of mini-batches
128 (default) | positive integer

This property is read-only.

Size of mini-batches returned by the `next` function, specified as a positive integer. The default value is 128.

Example: 256

`PartialMiniBatch` — Return or discard incomplete mini-batches
`"return"` (default) | `"discard"`

Return or discard incomplete mini-batches, specified as `"return"` or `"discard"`.

If the total number of observations is not exactly divisible by `MiniBatchSize`, the final mini-batch returned by the `next` function can have fewer than `MiniBatchSize` observations. This property specifies how any partial mini-batches are treated, using the following options:

- `"return"` — A mini-batch can contain fewer than `MiniBatchSize` observations. All data is returned.

  `"discard"` — All mini-batches must contain exactly `MiniBatchSize` observations. Some data can be discarded from the queue if there is not enough for a complete mini-batch.

Set `PartialMiniBatch` to `"discard"` if you require that all of your mini-batches are the same size.

Example: `"discard"`

Data Types: `char` | `string`

`MiniBatchFcn` — Mini-batch preprocessing function
`"collate"` (default) | function handle

This property is read-only.

Mini-batch preprocessing function, specified as `"collate"` or a function handle.

The default value of `MiniBatchFcn` is `"collate"`. This function concatenates the mini-batch variables into arrays.

Use a function handle to a custom function to preprocess mini-batches for custom training. Doing so is recommended for one-hot encoding classification labels, padding sequence data, calculating average images, and so on. You must specify a custom function if your data consists of cell arrays containing arrays of different sizes.

If you specify a custom mini-batch preprocessing function, the function must concatenate each batch of output variables into an array after preprocessing and return each variable as a separate function output. The function must accept at least as many inputs as the number of variables of the underlying datastore. The inputs are passed to the custom function as *N*-by-1 cell arrays, where *N* is the number of observations in the mini-batch. The function can return as many variables as required. If the function specified by `MiniBatchFcn` returns a different number of outputs than inputs, specify `numOutputs` as the number of outputs of the function.

The following actions are not recommended inside the custom function. To reproduce the desired behavior, instead, set the corresponding property when you create the `minibatchqueue` object.

Action	**Recommended Property**
Cast variable to different data type.	`OutputCast`
Move data to GPU.	`OutputEnvironment`
Convert data to `dlarray`.	`OutputAsDlarray`
Apply data format to `dlarray` variable.	`MiniBatchFormat`

Example: `@myCustomFunction`

Data Types: `char` | `string` | `function_handle`

`DispatchInBackground` — Preprocess mini-batches in the background in a parallel pool
`false` or `0` (default) | `true` or `1`

Preprocess mini-batches in the background in a parallel pool, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Using this option requires Parallel Computing Toolbox. The input datastore `ds` must be partitionable. Custom datastores must implement the `matlab.io.datastore.Partitionable` class.

Use this option when your mini-batches require heavy preprocessing. This option uses a parallel pool to prepare mini-batches in the background while you use mini-batches during training.

Workers in the pool process mini-batches by applying the function specified by `MiniBatchFcn`. Further processing, including applying the effects of the `OutputCast`, `OutputEnvironment`, `OutputAsDlarray`, and `MiniBatchFormat`, does not occur on the workers.

When `DispatchInBackground` is set to `true`, the software opens a local parallel pool using the current settings, if a local pool is not currently open. Non-local pools are not supported. The pool opens the first time you call `next`.

Example: `true`

Data Types: `logical`

`OutputCast` — Data type of each mini-batch variable
`'single'` (default) | `'double'` | `'int8'` | `'int16'` | `'int32'` | `'int64'` | `'uint8'` | `'uint16'` | `'uint32'` | `'uint64'` | `'logical'` | `'char'` | cell array

This property is read-only.

Data type of each mini-batch variable, specified as `'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'logical'`, or `'char'`, or a cell array of these values, or an empty vector.

If you specify `OutputCast` as an empty vector, the data type of each mini-batch variable is unchanged. To specify a different data type for each mini-batch variable, specify a cell array containing an entry for each mini-batch variable. The order of the elements of this cell array must match the order in which the mini-batch variables are returned. This order is the same order in which the variables are returned from the function specified by `MiniBatchFcn`. If you do not specify a custom function for `MiniBatchFcn`, it is the same order in which the variables are returned by the underlying datastore.

You must make sure that the value of `OutputCast` does not conflict with the values of the `OutputAsDlarray` or `OutputEnvironment` properties. If you specify `OutputAsDlarray` as `true` or `1`, check that the data type specified by `OutputCast` is supported by `dlarray`. If you specify `OutputEnvironment` as `"gpu"` or `"auto"` and a supported GPU is available, check that the data type specified by `OutputCast` is supported by `gpuArray`.

Example: `{'single','single','logical'}`

Data Types: `char` | `string`

`OutputAsDlarray` — Flag to convert mini-batch variable to `dlarray`
`true` or `1` (default) | `false` or `0` | vector of logical values

This property is read-only.

Flag to convert mini-batch variable to `dlarray`, specified as a numeric or logical `1` (`true`) or `0` (`false`) or as a vector of numeric or logical values.

To specify a different value for each output, specify a vector containing an entry for each mini-batch variable. The order of the elements of this vector must match the order in which the mini-batch variable are returned. This order is the same order in which the variables are returned from the function specified by `MiniBatchFcn`. If you do not specify a custom function for `MiniBatchFcn`, it is the same order in which the variables are returned by the underlying datastore.

Variables that are converted to `dlarray` have the underlying data type specified by the `OutputCast` property.

Example: `[1,1,0]`

Data Types: `logical`

`MiniBatchFormat` — Data format of mini-batch variables
`''` (default) | character vector | cell array

This property is read-only.

Data format of mini-batch variables, specified as a character vector or a cell array of character vectors.

The mini-batch format is applied to `dlarray` variables only. Non-`dlarray` mini-batch variables must have a `MiniBatchFormat` of `''`.

To avoid an error when you have a mix of `dlarray` and non-`dlarray` variables, you must specify a value for each output by providing a cell array containing an entry for each mini-batch variable. The order of the elements of this cell array must match the order in which the mini-batch variables are returned. This is the same order in which the variables are returned from the function specified by `MiniBatchFcn`. If you do not specify a custom function for `MiniBatchFcn`, it is the same order in which the variables are returned by the underlying datastore.

Example: `{'SSCB', ''}`

Data Types: `char` | `string`

`OutputEnvironment` — Hardware resource for mini-batch variables
`'auto'` (default) | `'gpu'` | `'cpu'` | cell array

Hardware resource for mini-batch variables returned using the `next` function, specified as one of the following values:

- `'auto'` — Return mini-batch variables on the GPU if one is available. Otherwise, return mini-batch variables on the CPU.
- `'gpu'` — Return mini-batch variables on the GPU.
- `'cpu'` — Return mini-batch variables on the CPU

To return only specific variables on the GPU, specify `OutputEnvironment` as a cell array containing an entry for each mini-batch variable. The order of the elements of this cell array must match the order the mini-batch variable are returned. This order is the same order as the variables are returned from the function specified by `MiniBatchFcn`. If you do not specify a custom `MiniBatchFcn`, it is the same order as the variables are returned by the underlying datastore.

Using a GPU requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose the `'gpu'` option and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

Example: `{'gpu','cpu'}`

Data Types: `char` | `string`

Object Functions

hasdata	Determine if minibatchqueue can return mini-batch
next	Obtain next mini-batch of data from minibatchqueue
partition	Partition minibatchqueue
reset	Reset minibatchqueue to start of data
shuffle	Shuffle data in minibatchqueue

Examples

Prepare Mini-Batches for Custom Training Loop

Use a `minibatchqueue` object to automatically prepare mini-batches of images and classification labels for training in a custom training loop.

Create a datastore. Calling `read` on `auimds` produces a table with two variables: `input`, containing the image data, and `response`, containing the corresponding classification labels.

```
auimds = augmentedImageDatastore([100 100],digitDatastore);
A = read(auimds);
head(A,2)
```

```
ans =
          input          response
    _________________    ________

    {100×100 uint8}        0
    {100×100 uint8}        0
```

Create a `minibatchqueue` object from `auimds`. Set the `MiniBatchSize` property to `256`.

The `minibatchqueue` object has two output variables: the images and classification labels from the `input` and `response` variables of `auimds`, respectively. Set the `minibatchqueue` object to return the images as a formatted `dlarray` on the GPU. The images are single-channel black-and-white images. Add a singleton channel dimension by applying the format `'SSBC'` to the batch. Return the labels as a non-`dlarray` on the CPU.

```
mbq = minibatchqueue(auimds,...
    'MiniBatchSize',256,...
    'OutputAsDlarray',[1,0],...
    'MiniBatchFormat',{'SSBC',''},...
    'OutputEnvironment',{'gpu','cpu'})
```

Use the `next` function to obtain mini-batches from `mbq`.

```
[X,Y] = next(mbq);
```

Create Mini-Batches Using Custom Preprocessing Function

Preprocess data using a `minibatchqueue` with a custom mini-batch preprocessing function. The custom function rescales the incoming image data between 0 and 1 and calculates the average image.

Unzip the data and create a datastore.

```
unzip("MerchData.zip");
imds = imageDatastore("MerchData", ...
    "IncludeSubfolders",true, ...
    "LabelSource",'foldernames');
```

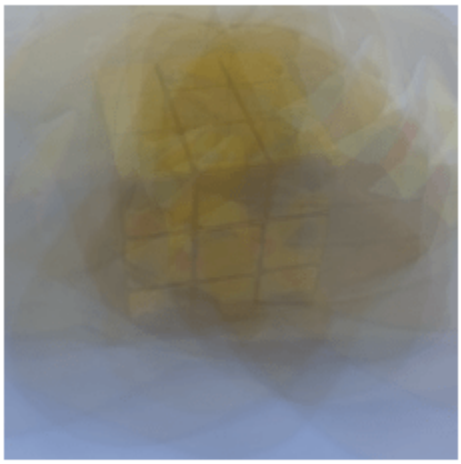
Create a `minibatchqueue` that preprocesses data using the custom function `preprocessMiniBatch` defined at the end of this example. The custom function concatenates the image data into a numeric array, rescales the image between 0 and 1, and calculates the average of the batch of images. The function returns the rescaled batch of images and the average image. Set the number of outputs to 2, to match the number of outputs of the function.

```
mbq = minibatchqueue(imds,2,...
    'MiniBatchSize',16,...
    'MiniBatchFcn',@preprocessMiniBatch,...
    'OutputAsDlarray',0)
```

```
mbq =
minibatchqueue with 2 outputs and properties:

   Mini-batch creation:
           MiniBatchSize: 16
        PartialMiniBatch: 'return'
            MiniBatchFcn: @preprocessMiniBatch
    DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'  'single'}
         OutputAsDlarray: [0 0]
         MiniBatchFormat: {''  ''}
       OutputEnvironment: {'auto'  'auto'}
```

Obtain a mini-batch and display the average of the images in the mini-batch.

```
[X,averageImage] = next(mbq);
imshow(averageImage)
```



```
function [X,averageImage] = preprocessMiniBatch(XCell)
    X = cat(4,XCell{:});

    X = rescale(X,"InputMin",0,"InputMax",255);
    averageImage = mean(X,4);

end
```

Use `minibatchqueue` in Custom Training Loop

Train a network using `minibatchqueue` to manage the processing of mini-batches.

Load Training Data

Load the digits training data and store the data in a datastore. Create a datastore for the images and one for the labels using `arrayDatastore`. Then, combine the datastores to produce a single datastore to use with `minibatchqueue`.

```
[XTrain,YTrain] = digitTrain4DArrayData;
dsX = arrayDatastore(XTrain,'IterationDimension',4);
dsY = arrayDatastore(YTrain);

dsTrain = combine(dsX,dsY);
```

Determine the number of unique classes in the label data.

```
classes = categories(YTrain);
numClasses = numel(classes);
```

Define Network

Define the network and specify the average image value 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')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of the example. The function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```
numEpochs = 10;
miniBatchSize = 128;
```

Visualize the training progress in a plot.

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

Create the `minibatchqueue`

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

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

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

Train Network

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches while data is still available in the `minibatchqueue`. Update the network parameters using the `adamupdate` function. At the end of each epoch, display the training progress.

Initialize the training progress plot.

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

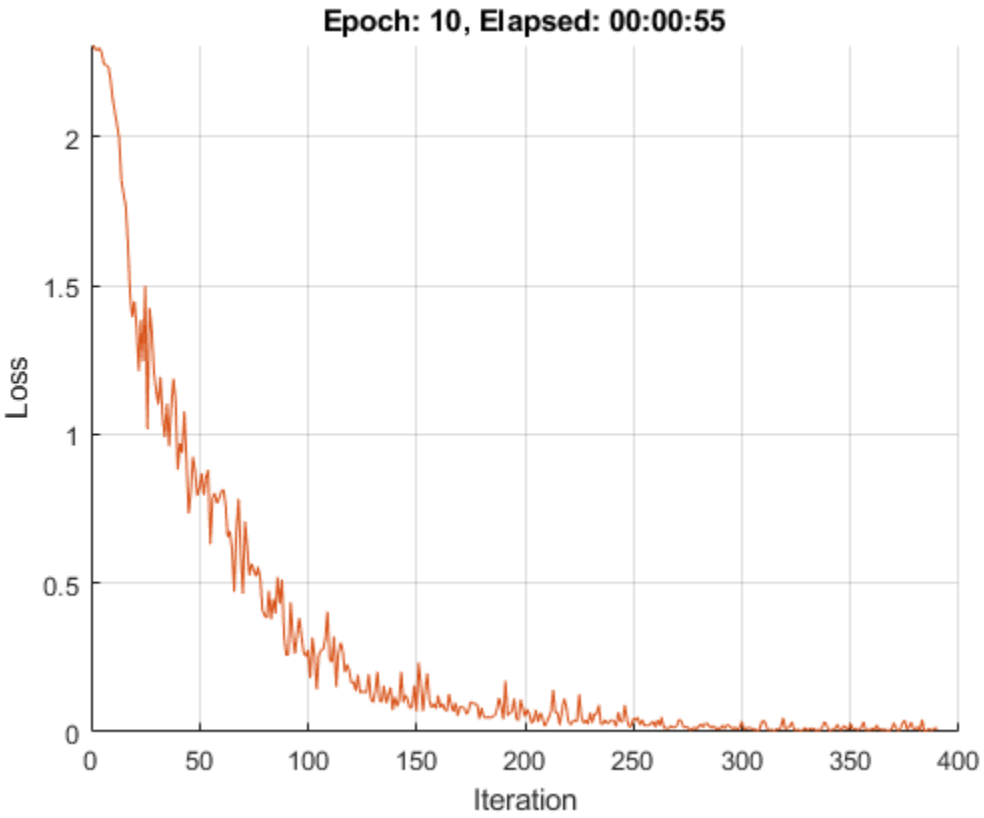
Initialize the average gradients and squared average gradients.

```
averageGrad = [];
averageSqGrad = [];
```

Train the network.

```
iteration = 0;
start = tic;

for epoch = 1:numEpochs
    % Shuffle data.
    shuffle (mbq);

    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,Y] = next(mbq);

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients helper function.
```

```
            [grad,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

            % Update the network parameters using the Adam optimizer.
            [dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,averageGrad,averageSqGrad,itera

            % 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
```



Model Gradients Function

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

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)
    dlYPred = forward(dlnet,dlX);
    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);
```

```
end
```

Mini-Batch Preprocessing Function

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

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

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)
    % Extract image data from the cell array and concatenate over fourth
    % dimension to add a third singleton dimension, as the channel
    % dimension.
    X = cat(4,XCell{:});

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

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

end
```

See Also

`datastore` | `dlfeval` | `dlarray` | `dlnetwork`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Generative Adversarial Network (GAN)”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

mobilenetv2

MobileNet-v2 convolutional neural network

Syntax

```
net = mobilenetv2
net = mobilenetv2('Weights','imagenet')

lgraph = mobilenetv2('Weights','none')
```

Description

MobileNet-v2 is a convolutional neural network that is 53 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the MobileNet-v2 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with MobileNet-v2.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load MobileNet-v2 instead of GoogLeNet.

`net = mobilenetv2` returns a MobileNet-v2 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for MobileNet-v2 Network* support package. If this support package is not installed, then the function provides a download link.

`net = mobilenetv2('Weights','imagenet')` returns a MobileNet-v2 network trained on the ImageNet data set. This syntax is equivalent to `net = mobilenetv2`.

`lgraph = mobilenetv2('Weights','none')` returns the untrained MobileNet-v2 network architecture. The untrained model does not require the support package.

Examples

Download MobileNet-v2 Support Package

Download and install the Deep Learning Toolbox Model *for MobileNet-v2 Network* support package.

Type `mobilenetv2` at the command line.

```
mobilenetv2
```

If the Deep Learning Toolbox Model *for MobileNet-v2 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `mobilenetv2` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

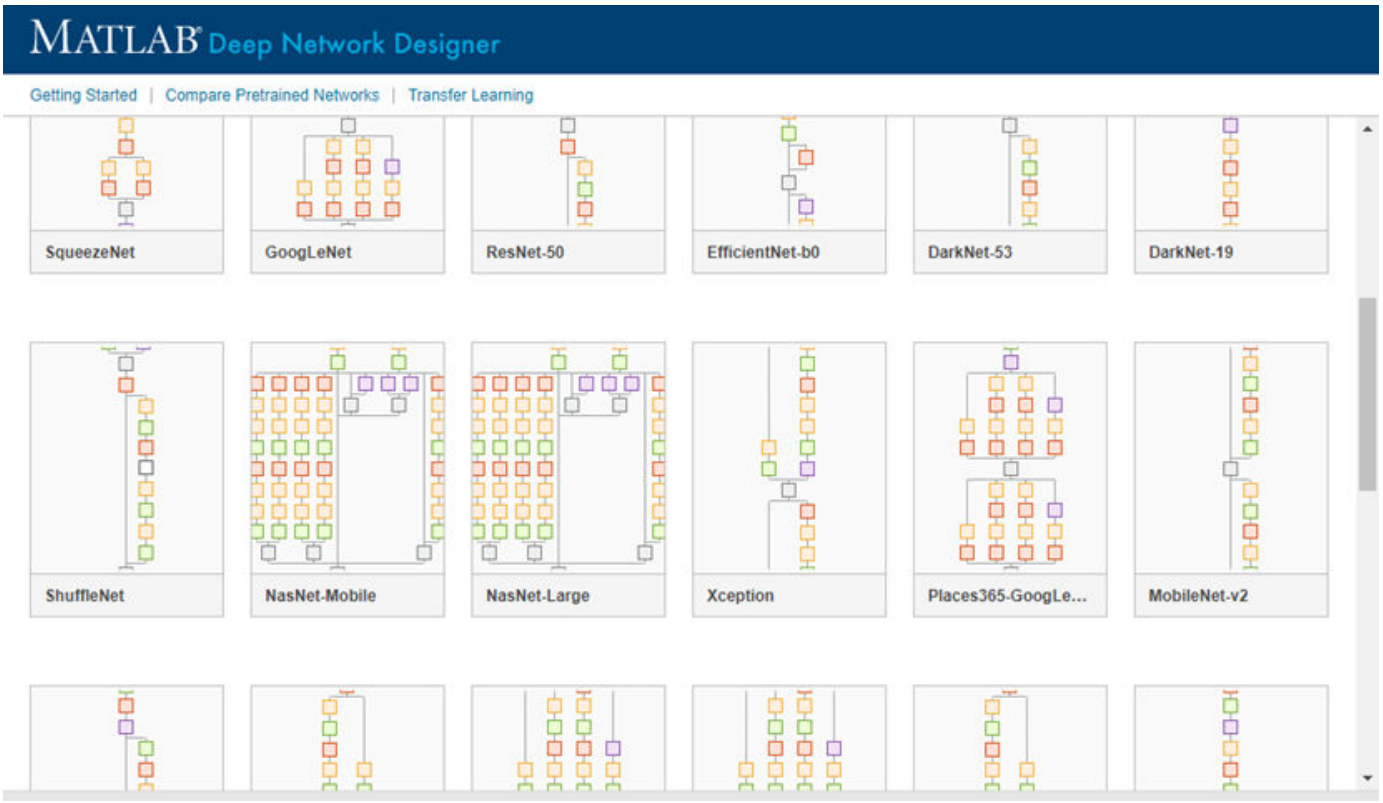
```
mobilenetv2
```

```
ans =

  DAGNetwork with properties:

         Layers: [155×1 nnet.cnn.layer.Layer]
    Connections: [164×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(mobilenetv2)
```

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



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

Output Arguments

net — Pretrained MobileNet-v2 convolutional neural network

`DAGNetwork` object

Pretrained MobileNet-v2 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained MobileNet-v2 convolutional neural network architecture
`LayerGraph` object

Untrained MobileNet-v2 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.C. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4510-4520). IEEE.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = mobilenetv2` or by passing the `mobilenetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('mobilenetv2')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `mobilenetv2('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = mobilenetv2` or by passing the `mobilenetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('mobilenetv2')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `mobilenetv2('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a

mse

Half mean squared error

Syntax

```
loss = mse(dlY,targets)
loss = mse(dlY,targets,'DataFormat',FMT)
```

Description

The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression tasks.

The loss is calculated using the following formula

$$\text{loss} = \frac{1}{2N}\sum_{i=1}^{M}(X_i - T_i)^2$$

where X_i is the network prediction, T_i is the target value, M is the total number of responses in X (across all observations), and N is the total number of observations in X.

Note This function computes the half mean squared error loss between predictions and targets stored as `dlarray` data. If you want to calculate the half mean squared error loss within a `layerGraph` object or `Layer` array for use with `trainNetwork`, use the following layer:

- `regressionLayer`

`loss = mse(dlY,targets)` computes the half mean squared error loss between the predictions `dlY` and the target values `targets` for regression problems. The input `dlY` must be a formatted `dlarray`. The output `loss` is an unformatted `dlarray` scalar.

`loss = mse(dlY,targets,'DataFormat',FMT)` also specifies the dimension format `FMT` when `dlY` is not a formatted `dlarray`.

Examples

Find Half Mean Squared Error Between Predicted and Target Values

The half mean squared error evaluates how well the network predictions correspond to the target values.

Create the input predictions as a single observation of random values with a height and width of six and a single channel.

```
height = 6;
width = 6;
```

```
channels = 1;
observations = 1;

Y = rand(height,width,channels,observations);
dlY = dlarray(Y,'SSCB')
```

Create the target values as a numeric array with the same dimension order as the input data `dlY`.

```
targets = ones(height,width,channels,observations);
```

Compute the half mean squared error between the predictions and the targets.

```
loss = mse(dlY,targets)
```

```
loss =

  1x1 dlarray

    5.2061
```

Input Arguments

`dlY` — Predictions

`dlarray` | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `dlY` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `dlY` is a numeric array, `targets` must be a `dlarray`.

`targets` — Target responses

`dlarray` | numeric array

Target responses, specified as a formatted or unformatted `dlarray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `dlY`.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `dlY`, or the same as `DataFormat` if `dlY` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `dlY` or the value of `DataFormat` to `targets`.

Tip Formatted `dlarray` objects automatically permute the dimensions of the underlying data to have order `"S"` (spatial), `"C"` (channel), `"B"` (batch), `"T"` (time), then `"U"` (unspecified). To ensure that the dimensions of `dlY` and `targets` are consistent, when `dlY` is a formatted `dlarray`, also specify `targets` as a formatted `dlarray`.

`FMT` — Dimension order of unformatted data

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`loss` — Half mean squared error loss

`dlarray` scalar

Half mean squared error loss, returned as an unformatted `dlarray` scalar. The output `loss` has the same underlying data type as the input `dlY`.

More About

Half Mean Squared Error Loss

The `mse` function computes the half-mean-squared-error loss for regression problems. For more information, see the definition of "Regression Output Layer" on page 1-1161 on the `RegressionOutputLayer` reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlY`
  - `targets`

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `sigmoid` | `crossentropy` | `huber`

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network with Multiple Outputs”
“List of Functions with dlarray Support”

Introduced in R2019b

multiplicationLayer

Multiplication layer

Description

A multiplication layer multiplies inputs from multiple neural network layers element-wise.

Specify the number of inputs to the layer when you create it. The inputs to the layer have the names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer by using `connectLayers` or `disconnectLayers`, respectively. The size of the inputs to the multiplication layer must be either same across all dimensions or same across at least one dimension with other dimensions as singleton dimensions.

Creation

Syntax

```
layer = multiplicationLayer(numInputs)
layer = multiplicationLayer(numInputs,'Name',name)
```

Description

`layer = multiplicationLayer(numInputs)` creates a multiplication layer that multiplies `numInputs` inputs element-wise. This function also sets the `NumInputs` property.

`layer = multiplicationLayer(numInputs,'Name',name)` also sets the `Name` property.

Properties

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer greater than or equal to 2.

The inputs have the names `'in1','in2',...,'inN'`, where `N` is `NumInputs`. For example, if `NumInputs` is 3, then the inputs have the names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer using the `connectLayers` or `disconnectLayers` functions.

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`InputNames` — Input Names
`{'in1','in2',…,'inN'}` (default)

Input names, specified as `{'in1','in2',...,'inN'}`, where `N` is the number of inputs of the layer.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create and Connect Multiplication Layer

Create a multiplication layer with two inputs and the name `'mul_1'`.

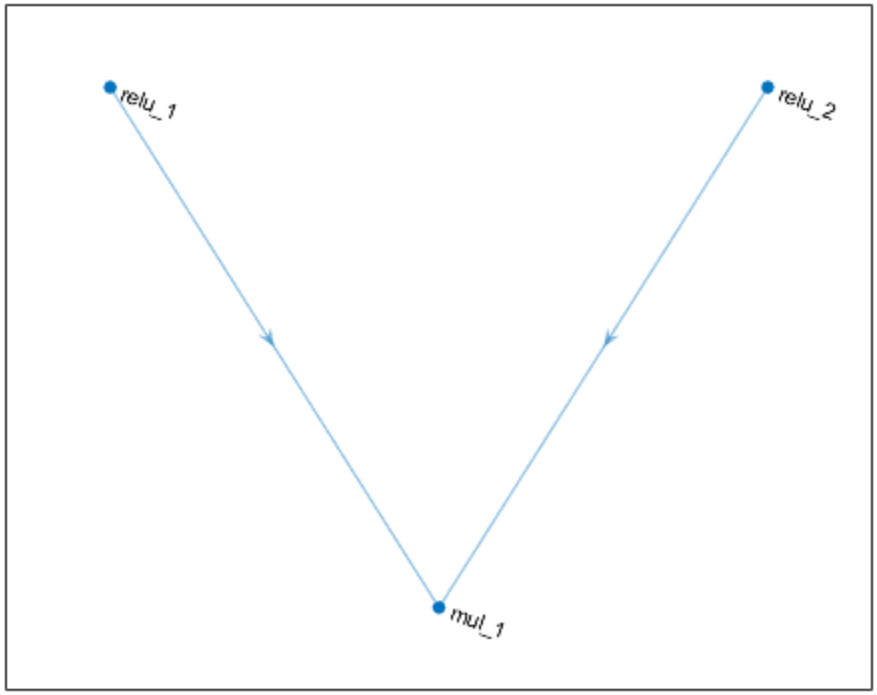
```
mul = multiplicationLayer(2,'Name','mul_1')
```

```
mul =
  MultiplicationLayer with properties:

         Name: 'mul_1'
    NumInputs: 2
   InputNames: {'in1'  'in2'}

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Create two ReLU layers and connect them to the multiplication layer. The multiplication layer multiplies the outputs from the ReLU layers.

```
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph();
lgraph = addLayers(lgraph,relu_1);
```

```
lgraph = addLayers(lgraph,relu_2);
lgraph = addLayers(lgraph,mul);

lgraph = connectLayers(lgraph,'relu_1','mul_1/in1');
lgraph = connectLayers(lgraph,'relu_2','mul_1/in2');

plot(lgraph);
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `layerGraph` | `additionLayer` | `concatenationLayer`

Topics

“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2020b

nasnetlarge

Pretrained NASNet-Large convolutional neural network

Syntax

```
net = nasnetlarge
```

Description

NASNet-Large is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 331-by-331. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the NASNet-Large model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with NASNet-Large.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load NASNet-Large instead of GoogLeNet.

`net = nasnetlarge` returns a pretrained NASNet-Large convolutional neural network.

This function requires the *Deep Learning Toolbox Model for NASNet-Large Network* support package. If this support package is not installed, then the function provides a download link.

Examples

Download NASNet-Large Support Package

Download and install the *Deep Learning Toolbox Model for NASNet-Large Network* support package.

Type `nasnetlarge` at the command line.

```
nasnetlarge
```

If the *Deep Learning Toolbox Model for NASNet-Large Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `nasnetlarge` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

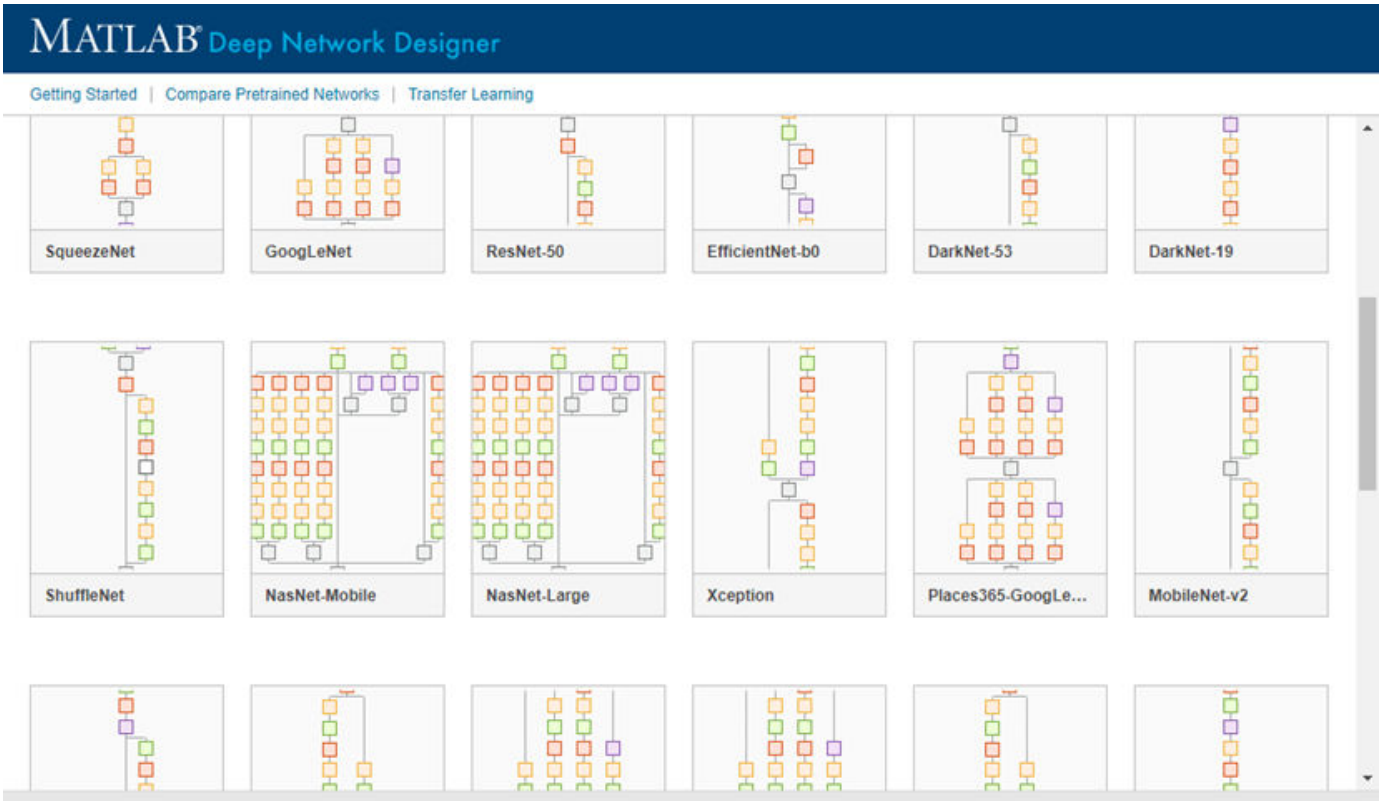
```
nasnetlarge
```

```
ans =

  DAGNetwork with properties:

         Layers: [1244×1 nnet.cnn.layer.Layer]
    Connections: [1463×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(nasnetlarge)
```

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



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

Transfer Learning with NASNet-Large

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the NASNet-Large network instead of GoogLeNet.

```
net = nasnetlarge
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

Output Arguments

net — Pretrained NASNet-Large convolutional neural network
`DAGNetwork` object

Pretrained NASNet-Large convolutional neural network, returned as a `DAGNetwork` object.

References

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

[2] Zoph, Barret, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. "Learning Transferable Architectures for Scalable Image Recognition ." *arXiv preprint arXiv:1707.07012* 2, no. 6 (2017).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = nasnetlarge` or by passing the `nasnetlarge` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetlarge')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = nasnetlarge` or by passing the `nasnetlarge` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetlarge')`

For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also
Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201` | `nasnetmobile` | `shufflenet`

Topics
“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”

“Train Residual Network for Image Classification”

Introduced in R2019a

nasnetmobile

Pretrained NASNet-Mobile convolutional neural network

Syntax

```
net = nasnetmobile
```

Description

NASNet-Mobile is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the NASNet-Mobile model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with NASNet-Mobile.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load NASNet-Mobile instead of GoogLeNet.

`net = nasnetmobile` returns a pretrained NASNet-Mobile convolutional neural network.

This function requires the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package. If this support package is not installed, then the function provides a download link.

Examples

Download NASNet-Mobile Support Package

Download and install the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package.

Type `nasnetmobile` at the command line.

```
nasnetmobile
```

If the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `nasnetmobile` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
nasnetmobile
```

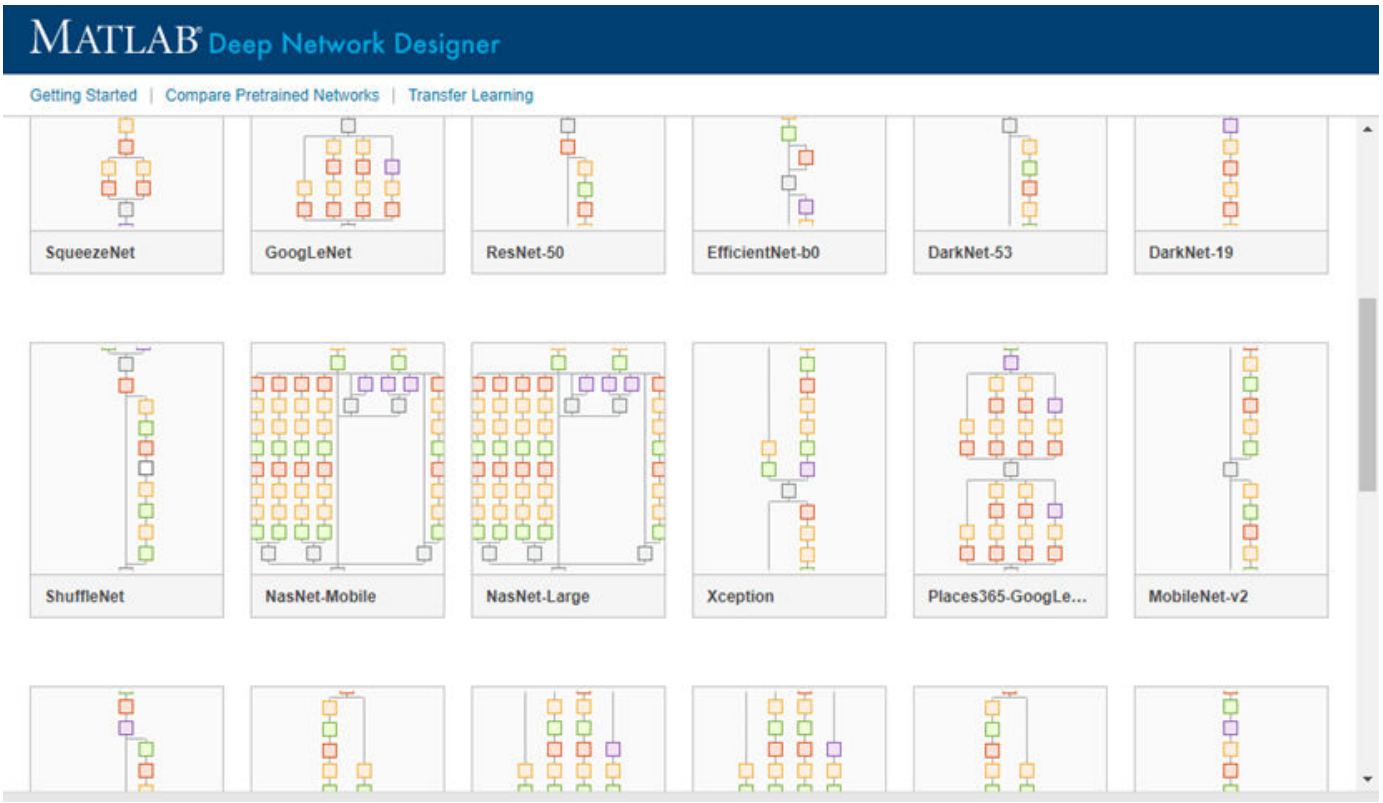
```
ans =

  DAGNetwork with properties:
```

```
        Layers: [914×1 nnet.cnn.layer.Layer]
    Connections: [1073×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(nasnetmobile)
```

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



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

Transfer Learning with NASNet-Mobile

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the NASNet-Mobile network instead of GoogLeNet.

```
net = nasnetmobile
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

Output Arguments

net — Pretrained NASNet-Mobile convolutional neural network
`DAGNetwork` object

Pretrained NASNet-Mobile convolutional neural network, returned as a `DAGNetwork` object.

References

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

[2] Zoph, Barret, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. "Learning Transferable Architectures for Scalable Image Recognition ." *arXiv preprint arXiv:1707.07012* 2, no. 6 (2017).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = nasnetmobile` or by passing the `nasnetmobile` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetmobile')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = nasnetmobile` or by passing the `nasnetmobile` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetmobile')`

For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201` | `nasnetlarge` | `shufflenet`

Topics
“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”

“Train Residual Network for Image Classification”

Introduced in R2019a

next

Obtain next mini-batch of data from minibatchqueue

Syntax

```
[x1,...,xN] = next(mbq)
```

Description

`[x1,...,xN] = next(mbq)` returns a mini-batch of data prepared using the `minibatchqueue` object `mbq`. The function returns as many variables as the number of outputs of `mbq`.

Examples

Obtain Mini-Batch

Create a `minibatchqueue` object and obtain a mini-batch.

Create a `minibatchqueue` object from a datastore. Set the `MiniBatchSize` property to `2`.

```
auimds = augmentedImageDatastore([100 100],digitDatastore);
mbq = minibatchqueue(auimds,'MiniBatchSize',2,"MiniBatchFormat",{'SSBC','BC'})
```

```
mbq =
minibatchqueue with 2 outputs and properties:

   Mini-batch creation:
           MiniBatchSize: 2
         PartialMiniBatch: 'return'
             MiniBatchFcn: 'collate'
    DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'  'single'}
         OutputAsDlarray: [1 1]
         MiniBatchFormat: {'SSBC'  'BC'}
       OutputEnvironment: {'auto'  'auto'}
```

Use `next` to obtain a mini-batch. `mbq` has two outputs.

```
[X,Y] = next(mbq);
```

`X` is a mini-batch containing two images from the datastore. `Y` contains the classification labels of those images. Check the size and data format of the mini-batch variables.

```
size(X)
dims(X)
size(Y)
dims(Y)
```

```
ans = 1×4
   100   100     1     2
```

```
ans = 'SSCB'
ans = 1×2
     1     2
ans = 'CB'
```

Input Arguments

`mbq` — Queue of mini-batches

`minibatchqueue`

Queue of mini-batches, specified as a `minibatchqueue` object.

Output Arguments

`[x1,...,xN]` — Mini-batch

numeric array | cell array

Mini-batch, returned as a numeric array or cell array.

The number and type of variables returned by `next` depends on the configuration of `mbq`. The function returns as many variables as the number of outputs of `mbq`.

See Also

`hasdata` | `reset` | `minibatchqueue`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Generative Adversarial Network (GAN)”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

occlusionSensitivity

Explain network predictions by occluding the inputs

Syntax

```
scoreMap = occlusionSensitivity(net,X,label)
activationMap = occlusionSensitivity(net,X,layer,channel)
___ = occlusionSensitivity( ___ ,Name,Value)
```

Description

`scoreMap = occlusionSensitivity(net,X,label)` computes a map of the change in classification score for the classes specified by `label` when parts of the input data `X` are occluded with a mask. The change in classification score is relative to the original data without occlusion. The occluding mask is moved across the input data, giving a change in classification score for each mask location. Use an occlusion map to identify the parts of your input data that most impact the classification score. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified classification label. The network must contain a `classificationLayer`.

`activationMap = occlusionSensitivity(net,X,layer,channel)` computes a map of the change in total activation for the specified layer and channel when parts of the input data `X` are occluded with a mask. The change in activation score is relative to the original data without occlusion. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation, obtained by summing over all spatial dimensions for that channel.

`___ = occlusionSensitivity( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, `'Stride',50` sets the stride of the occluding mask to 50 pixels.

Examples

Visualize Which Parts of an Image Influence Classification Score

Import the pretrained network GoogLeNet.

```
net = googlenet;
```

Import the image and resize to match the input size for the network.

```
X = imread("sherlock.jpg");

inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Display the image.

```
imshow(X)
```

Classify the image to get the class label.

```
label = classify(net,X)
```

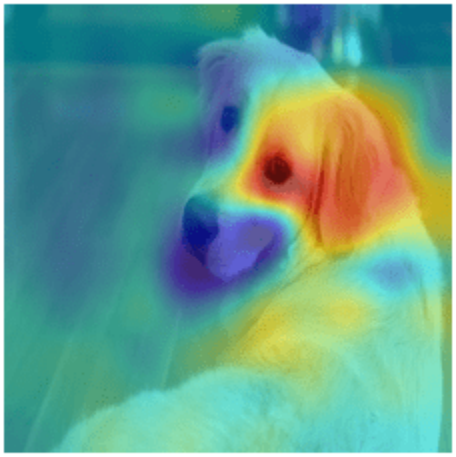
```
label = categorical
     golden retriever
```

Use `occlusionSensitivity` to determine which parts of the image positively influence the classification result.

```
scoreMap = occlusionSensitivity(net,X,label);
```

Plot the result over the original image with transparency to see which areas of the image affect the classification score.

```
figure
imshow(X)
hold on
imagesc(scoreMap,'AlphaData',0.5);
colormap jet
```



The red parts of the map show the areas which have a positive contribution to the specified label. The dog's left eye and ear strongly influence the network's prediction of `golden retriever`.

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”.

Input Arguments

`net` — Trained network
`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` object or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function. For more information about pretrained networks, see “Pretrained Deep Neural Networks”.

`net` must contain a single input layer. The input layer must be an `imageInputLayer`.

`X` — Observation to occlude
numeric array

Observation to occlude, specified as a numeric array. You can calculate the occlusion map of one observation at a time. For example, specify a single image to understand which parts of that image affect classification results.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`label` — Class label used to calculate change in classification score
categorical array | character array | string array

Class label used to calculate change in classification score, specified as a categorical, a character array, or a string array.

If you specify `label` as a vector, the software calculates the change in classification score for each class label independently. In that case, `scoreMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `label`.

Data Types: `char` | `string` | `categorical`

`layer` — Layer used to calculate change in activation

character vector | string scalar

Layer used to calculate change in activation, specified as a character vector or a string scalar. Specify `layer` as the name of the layer in `net` for which you want to compute the change in activations.

Data Types: `char` | `string`

`channel` — Channel used to calculate change in activation

numeric index | vector of numeric indices

Channel used to calculate change in activation, specified as scalar or vector of channel indices. The possible choices for `channel` depend on the selected layer. For example, for convolutional layers, the `NumFilters` property specifies the number of output channels. You can use `analyzeNetwork` to inspect the network and find out the number of output channels for each layer.

If `channel` is specified as a vector, the change in total activation for each specified channel is calculated independently. In that case, `activationMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `channel`.

The function computes the change in total activation due to occlusion. The total activation is computed by summing over all spatial dimensions of the activation of that channel. The occlusion map corresponds to the difference between the total activation of the original data with no occlusion and the total activation for the occluded data. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaskSize',75,'OutputUpsampling','nearest'` uses an occluding mask with size 75 pixels along each side, and uses nearest-neighbor interpolation to upsample the output to the same size as the input data

`MaskSize` — Size of occluding mask

`'auto'` (default) | vector | scalar

Size of occluding mask, specified as the comma-separated pair consisting of `'MaskSize'` and one of the following.

- `'auto'` — Use a mask size of 20% of the input size, rounded to the nearest integer.
- A vector of the form `[h w]`— Use a rectangular mask with height `h` and width `w`.

- A scalar — Use a square mask with height and width equal to the specified value.

Example: `'MaskSize',[50 60]`

`Stride` — Step size for traversing mask across input data
`'auto'` (default) | vector | scalar

Step size for traversing the mask across the input data, specified as the comma-separated pair consisting of `'Stride'` and one of the following.

- `'auto'` — Use a stride of 10% of the input size, rounded to the nearest integer.
- A vector of the form `[a b]`— Use a vertical stride of `a` and a horizontal stride of `b`.
- A scalar — Use a stride of the specified value in both the vertical and horizontal directions.

Example: `'Stride',30`

`MaskValue` — Replacement value of occluded region
`'auto'` (default) | scalar | vector

Replacement value of occluded region, specified as the comma-separated pair consisting of `'MaskValue'` and one of the following.

- `'auto'` — Replace occluded pixels with the channel-wise mean of the input data.
- A scalar — Replace occluded pixels with the specified value.
- A vector — Replace occluded pixels with the value specified for each channel. The vector must contain the same number of elements as the number of output channels of the layer.

Example: `'MaskValue',0.5`

`OutputUpsampling` — Output upsampling method
`'bicubic'` (default) | `'nearest'` | `'none'`

Output upsampling method, specified as the comma-separated pair consisting of `'OutputUpsampling'` and one of the following.

- `'bicubic'` — Use bicubic interpolation to produce a smooth map the same size as the input data.
- `'nearest'` — Use nearest-neighbor interpolation expand the map to the same size as the input data. The map indicates the resolution of the occlusion computation with respect to the size of the input data.
- `'none'` — Use no upsampling. The map can be smaller than the input data.

If `'OutputUpsampling'` is `'bicubic'` or `'nearest'`, the computed map is upsampled to the size of the input data using the `imresize` function.

Example: `'OutputUpsampling','nearest'`

`MaskClipping` — Edge handling of the occluding mask
`'on'` (default) | `'off'`

Edge handling of the occluding mask, specified as the comma-separated pair consisting of `'MaskClipping'` and one of the following.

- `'on'` — Place the center of the first mask at the top-left corner of the input data. Masks at the edges of the data are not full size.

- `'off'` — Place the top-left corner of the first mask at the top-left corner of the input data. Masks are always full size. If the values of the `MaskSize` and `Stride` options mean that some masks extend past the boundaries of the data, those masks are excluded.

For non-image input data, you can ensure you always occlude the same amount of input data using the option `'MaskClipping','off'`. For example, for word embeddings data, you can ensure the same number of words are occluded at each point.

Example: `'MaskClipping','off'`

`MiniBatchSize` — Size of mini-batch
128 (default) | positive integer

Size of the mini-batch to use to compute the map of change in classification score, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a positive integer.

A mini-batch is a subset of the set of occluded images as the mask is moved across the input image. All occluded images are used to calculate the map; the mini-batch determines the number of images that are passed to the network at once. Larger mini-batch sizes lead to faster computation, at the cost of more memory.

Example: `'MiniBatchSize',256`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'cpu'` | `'gpu'`

Hardware resource for computing map, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following.

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.

The GPU option requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose the `'ExecutionEnvironment','gpu'` option and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

Example: `'ExecutionEnvironment','gpu'`

Output Arguments

`scoreMap` — Map of change of classification score
numeric matrix | numeric array

Map of change of classification score, returned as a numeric matrix or a numeric array. The change in classification score is calculated relative to the original input data without occlusion. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified classification label.

If `label` is specified as a vector, the change in classification score for each class label is calculated independently. In that case, `scoreMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `label`.

`activationMap` — Map of change of total activation
numeric matrix | numeric array

Map of change of total activation, returned as a numeric matrix or a numeric array.

The function computes the change in total activation due to occlusion. The total activation is computed by summing over all spatial dimensions of the activation of that channel. The occlusion map corresponds to the difference between the total activation of the original data with no occlusion and the total activation for the occluded data. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation.

If `channels` is specified as a vector, the change in total activation for each specified channel is calculated independently. In that case, `activationMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `channel`.

See Also

`activations` | `classify` | `imageLIME` | `gradCAM`

Topics
“Understand Network Predictions Using Occlusion”
“Grad-CAM Reveals the Why Behind Deep Learning Decisions”
“Understand Network Predictions Using LIME”
“Investigate Network Predictions Using Class Activation Mapping”
“Visualize Features of a Convolutional Neural Network”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2019b

onehotdecode

Decode probability vectors into class labels

Syntax

```
A = onehotdecode(B,classes,featureDim)
A = onehotdecode(B,classes,featureDim,typename)
```

Description

`A = onehotdecode(B,classes,featureDim)` decodes each probability vector in `B` to the most probable class label from the labels specified by `classes`. `featureDim` specifies the dimension along which the probability vectors are defined. The function decodes the probability vectors into class labels by matching the position of the highest value in the vector with the class label in the corresponding position in `classes`. Each probability vector in `A` is replaced with the value of `classes` that corresponds to the highest value in the probability vector.

`A = onehotdecode(B,classes,featureDim,typename)` decodes each probability vector in `B` to the most probable class label and returns the result with data type `typename`. Use this syntax to obtain decoded class labels with a specific data type.

Examples

Encode and Decode Labels

Use the `onehotencode` and `onehotdecode` functions to encode a set of labels into probability vectors and decode them back into labels.

Create a vector of categorical labels.

```
colorsOriginal = ["red" "blue" "red" "green" "yellow" "blue"];
colorsOriginal = categorical(colorsOriginal)
```

```
colorsOriginal = 1x6 categorical
     red      blue      red      green      yellow      blue
```

Determine the classes in the categorical vector.

```
classes = categories(colorsOriginal);
```

One-hot encode the labels into probability vectors by using the `onehotencode` function. Encode the probability vectors into the first dimension.

```
colorsEncoded = onehotencode(colorsOriginal,1)
```

```
colorsEncoded = 4×6

     0     1     0     0     0     1
     0     0     0     1     0     0
     1     0     1     0     0     0
```

```
     0     0     0     0     1     0
```

Use `onehotdecode` to decode the probability vectors.

```
colorsDecoded = onehotdecode(colorsEncoded,classes,1)
```

```
colorsDecoded = 1x6 categorical
     red      blue      red      green      yellow      blue
```

The decoded labels match the original labels.

Decode Probability Vectors into Most Probable Classes

Use onehotdecode to decode a set of probability vectors into the most probable class for each observation.

Create a set of 10 random probability vectors. The vectors express the probability that an observation belongs to one of five classes.

```
numObs = 10;
numClasses = 5;

prob = rand(numObs,numClasses);

tot = sum(prob,2);
prob = prob./tot;
```

Define the set of five classes.

```
classes = ["Red" "Yellow" "Green" "Blue" "Purple"];
```

Decode the probabilities into the most probable classes. The probability vectors are encoded into the second dimension, so specify the dimension containing encoded probabilities as 2. Obtain the most probable classes as a vector of strings.

```
result = onehotdecode(prob,classes,2,"string")
```

```
result = 10x1 string
    "Red"
    "Yellow"
    "Yellow"
    "Green"
    "Yellow"
    "Blue"
    "Green"
    "Yellow"
    "Red"
    "Red"
```

Input Arguments

`B` — Probability vectors

numeric array

Probability vectors to decode, specified as a numeric array.

Values in `B` must be between `0` and `1`. If a probability vector in `B` contains `NaN` values, the function decodes that observation to the class with the largest probability that is not `NaN`. If an observation contains only `NaN` values, the function decodes that observation to the first class label in `classes`.

Data Types: `single` | `double`

`classes` — Classes

cell array | string vector | numeric vector | character array

Classes, specified as a cell array of character vectors, a string vector, a numeric vector, or a two-dimensional character array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `string` | `cell` | `char`

`featureDim` — Dimension containing probability vectors

positive integer

Dimension containing probability vectors, specified as a positive integer.

Use `featureDim` to specify the dimension in `B` that contains the probability vectors. The function replaces each vector in `B` along the specified dimension with the element of `classes` in the same position as the highest value along the vector.

The dimension of `B` specified by `featureDim` must have length equal to the number of classes specified by `classes`.

`typename` — Data type of decoded labels

`'categorical'` (default) | character vector | string scalar

Data type of decoded labels, specified as a character vector or a string scalar.

Valid values of `typename` are `'categorical'`, `'string'`, and numeric types such as `'single'` and `'int64'`. If you specify a numeric type, `classes` must be a numeric vector.

Example: `'double'`

Data Types: `char` | `string`

Output Arguments

`A` — Decoded class labels

categorical array (default) | string array | numeric array

Decoded class labels, returned as a categorical array, a string array, or a numeric array.

See Also

`onehotencode` | `categories`

Topics
“Train Network Using Custom Training Loop”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

onehotencode

Encode data labels into one-hot vectors

Syntax

```
B = onehotencode(A,featureDim)
tblB = onehotencode(tblA)
___ = onehotencode( ___ ,typename)
___ = onehotencode( ___ ,'ClassNames',classes)
```

Description

`B = onehotencode(A,featureDim)` encodes data labels in categorical array `A` into a one-hot encoded array `B`. The function replaces each element of `A` with a numeric vector of length equal to the number of unique classes in `A` along the dimension specified by `featureDim`. The vector contains a `1` in the position corresponding to the class of the label in `A`, and a `0` in every other position. Any `<undefined>` values are encoded to `NaN` values.

`tblB = onehotencode(tblA)` encodes categorical data labels in table `tblA` into a table of one-hot encoded numeric values. The function replaces the single variable of `tblA` with as many variables as the number of unique classes in `tblA`. Each row in `tblB` contains a `1` in the variable corresponding to the class of the label in `tlbA`, and a `0` in all other variables.

`___ = onehotencode( ___ ,typename)` encodes the labels into numeric values of data type `typename`. Use this syntax with any of the input and output arguments in previous syntaxes.

`___ = onehotencode( ___ ,'ClassNames',classes)` also specifies the names of the classes to use for encoding. Use this syntax when `A` or `tblA` does not contain categorical values, when you want to exclude any class labels from being encoded, or when you want to encode the vector elements in a specific order. Any label in `A` or `tblA` of a class that does not exist in `classes` is encoded to a vector of `NaN` values.

Examples

One-Hot Encode a Vector of Labels

Encode a categorical vector of class labels into one-hot vectors representing the labels.

Create a column vector of labels, where each row of the vector represents a single observation. Convert the labels to a categorical array.

```
labels = ["red"; "blue"; "red"; "green"; "yellow"; "blue"];
labels = categorical(labels);
```

View the order of the categories.

```
categories(labels)
```

```
ans = 4x1 cell
    {'blue'  }
```

```
    {'green' }
    {'red'   }
    {'yellow'}
```

Encode the labels into one-hot vectors. Expand the labels into vectors in the second dimension to encode the classes.

```
labels = onehotencode(labels,2)
```

```
labels = 6×4

     0     0     1     0
     1     0     0     0
     0     0     1     0
     0     1     0     0
     0     0     0     1
     1     0     0     0
```

Each observation in `labels` is now a row vector with a `1` in the position corresponding to the category of the class label and `0` in all other positions. The function encodes the labels in the same order as the categories, such that a `1` in position `1` represents the first category in the list, in this case, `'blue'`.

One-Hot Encode Table

One-hot encode a table of categorical values.

Create a table of categorical data labels. Each row in the table holds a single observation.

```
color = ["blue"; "red"; "blue"; "green"; "yellow"; "red"];
color = categorical(color);
color = table (color);
```

One-hot encode the table of class labels.

```
color = onehotencode(color)
```

```
color=6×4 table
    blue    green    red    yellow
    ____    _____    ___    ______

     1        0       0       0
     0        0       1       0
     1        0       0       0
     0        1       0       0
     0        0       0       1
     0        0       1       0
```

Each column of the table represents a class. The function encodes the data labels with a `1` in the column of the corresponding class, and `0` everywhere else.

One-Hot Encode Subset of Classes

If not all classes in the data are relevant, encode the data labels using only a subset of the classes.

Create a row vector of data labels, where each column of the vector represents a single observation

```
pets = ["dog" "fish" "cat" "dog" "cat" "bird"];
```

Define the list of classes to encode. These classes are a subset of those present in the observations.

```
animalClasses = ["bird"; "cat"; "dog"];
```

One-hot encode the observations into the first dimension. Specify the classes to encode.

```
encPets = onehotencode(pets,1,"ClassNames",animalClasses)
```

```
encPets = 3×6

     0   NaN     0     0     0     1
     0   NaN     1     0     1     0
     1   NaN     0     1     0     0
```

Observations of a class not present in the list of classes to encode are encoded to a vector of `NaN` values.

One-Hot Encode Image for Semantic Segmentation

Use `onehotencode` to encode a matrix of class labels, such as a semantic segmentation of an image.

Define a simple 15-by-15 pixel segmentation matrix of class labels.

```
A = "blue";
B = "green";
C = "black";

A = repmat(A,8,15);
B = repmat(B,7,5);
C = repmat(C,7,5);

seg = [A;B C B];
```

Convert the segmentation matrix into a categorical array.

```
seg = categorical(seg);
```

One-hot encode the segmentation matrix into an array of type `single`. Expand the encoded labels into the third dimension.

```
encSeg = onehotencode(seg,3,"single");
```

Check the size of the encoded segmentation.

```
size(encSeg)
```

```
ans = 1×3
```

```
    15    15     3
```

The three possible classes of the pixels in the segmentation matrix are encoded as vectors in the third dimension.

One-Hot Encode Table with Several Variables

If your data is a table that contains several types of class variables, you can encode each variable separately.

Create a table of observations of several types of categorical data.

```
color = ["blue"; "red"; "blue"; "green"; "yellow"; "red"];
color = categorical(color);

pets = ["dog"; "fish"; "cat"; "dog"; "cat"; "bird"];
pets = categorical(pets);

location = ["USA"; "CAN"; "CAN"; "USA"; "AUS"; "USA"];
location = categorical(location);

data = table(color,pets,location)
```

```
data=6×3 table
    color     pets    location
    ______    ____    ________

    blue      dog       USA
    red       fish      CAN
    blue      cat       CAN
    green     dog       USA
    yellow    cat       AUS
    red       bird      USA
```

Use a `for`-loop to one-hot encode each table variable and append it to a new table containing the encoded data.

```
encData = table();

for i=1:width(data)
 encData = [encData onehotencode(data(:,i))];
end

encData
```

```
encData=6×11 table
    blue    green    red    yellow    bird    cat    dog    fish    AUS    CAN    USA
    ____    _____    ___    ______    ____    ___    ___    ____    ___    ___    ___

     1        0       0       0        0       0      1      0       0      0      1
     0        0       1       0        0       0      0      1       0      1      0
     1        0       0       0        0       1      0      0       0      1      0
     0        1       0       0        0       0      1      0       0      0      1
```

```
     0     0     0     1     0     1     0     0     1     0     0
     0     0     1     0     1     0     0     0     0     0     1
```

Each row of `encData` encodes the three different categorical classes for each observation.

Input Arguments

`A` — Array of data labels

categorical array | numeric array | string array

Array of data labels to encode, specified as a categorical array, a numeric array, or a string array.

- If `A` is a categorical array, the elements of the one-hot encoded vectors match the same order in `categories(A)`.
- If A is not a categorical array, you must specify the classes to encode using the `'ClassNames'` name-value argument. The function encodes the vectors in the order that the classes appear in `classes`.
- If `A` contains undefined values or values not present in `classes`, the function encodes those values as a vector of `NaN` values. `typename` must be `'double'` or `'single'`.

Data Types: `categorical` | `numeric` | `string`

`tblA` — Table of data labels

table

Table of data labels to encode, specified as a table. The table must contain a single variable and one row for each observation. Each entry must contain a categorical scalar, a numeric scalar, or a string scalar.

- If `tblA` contains categorical values, the elements of the one-hot encoded vectors match the order of the categories; for example, the same order as `categories(tbl(1,n))`.
- If `tblA` does not contain categorical values, you must specify the classes to encode using the `'ClassNames'` name-value argument. The function encodes the vectors in the order that the classes appear in `classes`.
- If `tblA` contains undefined values or values not present in `classes`, the function encodes those values as `NaN` values. `typename` must be `'double'` or `'single'`.

Data Types: `table`

`featureDim` — Dimension to expand

positive integer

Dimension to expand to encode the labels, specified as a positive integer.

`featureDim` must specify a singleton dimension of `A`, or be larger than `n` where `n` is the number of dimensions of `A`.

`typename` — Data type of encoded labels

`'double'` (default) | character vector | string scalar

Data type of the encoded labels, specified as a character vector or a string scalar.

- If the classification label input is a categorical array, a numeric array, or a string array, then the encoded labels are returned as an array of data type `typename`.
- If the classification label input is a table, then the encoded labels are returned as a table where each entry has data type `typename`.

Valid values of `typename` are floating point, signed and unsigned integer, and logical types.

Example: `'int64'`

Data Types: `char` | `string`

`classes` — Classes to encode
cell array | string vector | numeric vector | character array

Classes to encode, specified as a cell array of character vectors, a string vector, a numeric vector, or a two-dimensional character array.

- If the input `A` or `tblA` does not contain categorical values, then you must specify `classes`. You can also use the `classes` argument to exclude any class labels from being encoded, or to encode the vector elements in a specific order.
- If `A` or `tblA` contains undefined values or values not present in `classes`, the function encodes those values to a vector of `NaN` values. `typename` must be `'double'` or `'single'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `string` | `cell`

Output Arguments

`B` — Encoded labels
numeric array

Encoded labels, returned as a numeric array.

`tblB` — Encoded labels
table

Encoded labels, returned as a table.

Each row of `tblB` contains the one-hot encoded label for a single observation, in the same order as in `tblA`. Each row contains a `1` in the variable corresponding to the class of the label in `tlbA`, and a `0` in all other variables.

See Also
`categorical` | `onehotdecode` | `minibatchqueue`

Topics
“Train Network Using Custom Training Loop”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

padsequences

Pad or truncate sequence data to same length

Syntax

```
XPad = padsequences(X,paddingDim)
[XPad,mask] = padsequences(X,paddingDim)
[ ___ ] = padsequences(X,paddingDim,Name,Value)
```

Description

`XPad = padsequences(X,paddingDim)` pads the sequences in the cell array `X` along the dimension specified by `paddingDim`. The function adds padding at the end of each sequence to match the size of the longest sequence in `X`. The padded sequences are concatenated and the function returns `XPad` as an array.

`[XPad,mask] = padsequences(X,paddingDim)` additionally returns a logical array representing the positions of original sequence data in `XPad`, . The position of values of `true` or `1` in `mask` correspond to the positions of original sequence data in `XPad`; values of `false` or `0` correspond to padded values.

`[ ___ ] = padsequences(X,paddingDim,Name,Value)` specifies options using one or more name-value arguments in addition to the input and output arguments in previous syntaxes. For example, `'PaddingValue','left'` adds padding to the beginning of the original sequence.

Examples

Pad Sequence Data to Same Length

Pad sequence data ready for training.

Load the sequence data.

```
s = japaneseVowelsTrainData;
```

The preprocessed data contains 270 observations each with 12 sequence features. The length of the observations varies between 7 and 26 time steps.

Pad the data with zeros to the same length as the longest sequence. The function applies on the right side of the data. Specify the dimension containing the time steps as the padding dimension. For this example, the dimension is `2`.

```
sPad = padsequences(s,2);
```

Examine the size of the padded sequences.

```
size(sPad)
```

```
ans = 1×3
```

```
    12    26   270
```

Pad or Truncate Both Sides of Sequence Data

Use `padsequences` to extend or cut each sequence to a fixed length by adding or removing data at both ends of the sequence, depending on the length of the original sequence.

Load the sequence data.

```
s = japaneseVowelsTrainData;
```

The preprocessed data contains 270 observations each with 12 sequence features. The length of the observations varies between 7 and 26 time steps.

Process the data so that each sequence is exactly 14 time steps. For shorter sequences, padding is required, while longer sequences need to be truncated. Pad or truncate at both sides of the data. For the padded sequences, apply symmetric padding so that the padded values are mirror reflections of the original sequence values.

```
[sPad,mask] = padsequences(s,2,'Length',14,'Direction','both','PaddingValue','symmetric');
```

Compare some of the padded sequences with the original sequence. Each observation contains 12 features so extract a single feature to compare.

Extract the first feature of the 74th observation. This sequence is shorter than 14 time steps.

```
s{74}(1,:)
```

```
ans = 1×9

    0.6691    0.5291    0.3820    0.3107    0.2546    0.1942    0.0931   -0.0179   -0.1081
```

```
sPad(1,:,74)
```

```
ans = 1×14

    0.5291    0.6691    0.6691    0.5291    0.3820    0.3107    0.2546    0.1942    0.0931   -0.0
```

```
mask(1,:,74)
```

```
ans = 1×14 logical array

   0   0   1   1   1   1   1   1   1   1   1   0   0   0
```

The function centers the sequence and pads at both ends by reflecting the values at the ends of the sequence. The mask shows the location of the original sequence values.

Extract the first feature of the 28th observation. This sequence is longer than 14 time steps.

```
s{28}(1,:)
```

```
ans = 1×16
```

```
    1.1178    1.0772    1.2365    1.4858    1.6191    1.4893    1.2791    1.4692    1.5592    1.5
```

```
sPad(1,:,28)
```

```
ans = 1×14
```

```
    1.0772    1.2365    1.4858    1.6191    1.4893    1.2791    1.4692    1.5592    1.5112    1.5
```

```
mask(1,:,28)
```

```
ans = 1×14 logical array
```

```
   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

The function centers the sequence and truncates at both ends. The mask shows that all data in the resulting sequence is part of the original sequence.

Pad Mini-Batches of Sequences for Custom Training Loop

Use the `padsequences` function in conjunction with `minibatchqueue` to prepare and preprocess sequence data ready for training using a custom training loop.

The example uses the human activity recognition training data. The data contains six 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 the training data. Combine the data and labels into a single datastore.

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

dsXTrain = arrayDatastore(s.XTrain,'OutputType','same');
dsYTrain = arrayDatastore(s.YTrain,'OutputType','same');

dsTrain = combine(dsXTrain,dsYTrain);
```

Use `minibatchqueue` to process the mini-batches of sequence data. Define a custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to pad the sequence data and labels, and one-hot encode the label sequences. To also return the mask of the padded data, specify three output variables for the `minibatchqueue` object.

```
miniBatchSize = 2;
mbq = minibatchqueue(dsTrain,3,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch);
```

Check the size of the mini-batches.

```
[X,Y,mask] = next(mbq);
size(X)
```

```
ans = 1×3
```

```
           3       64480           2
```

```
size(mask)
```

```
ans = 1×3
```

```
           3       64480           2
```

Each mini-batch has two observations. The function pads the sequences to the same size as the longest sequence in the mini-batch. The mask is the same size as the padded sequences, and shows the location of the original data values in the padded sequence data.

```
size(Y)
```

```
ans = 1×3
```

```
           5       64480           2
```

The padded labels are one-hot encoded into numeric data ready for training.

```
function [xPad,yPad,mask] = preprocessMiniBatch(X,Y)
    [xPad,mask] = padsequences(X,2);
    yPad = padsequences(Y,2);
    yPad = onehotencode(yPad,1);
end
```

Input Arguments

X — Sequences to pad

cell vector

Sequences to pad, specified as a cell vector of numeric or categorical arrays.

Data Types: `cell`

paddingDim — Dimension along which to pad

positive integer

Dimension along which to pad input sequence data, specified as a positive integer.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `padsequences(X,'Length','shortest','Direction','both')` truncates the sequences at each end, to match the length of the shortest input sequence.

Length — Length of padded sequences

`'longest'` (default) | `'shortest'` | positive integer

Length of padded sequences, specified as one of the following:

- `'longest'` — Pad each input sequence to the same length as the longest input sequence.
- `'shortest'` — Truncate each input sequence to the same length as the shortest input sequence.
- Positive integer — Pad or truncate each input sequence to the specified length.

Example: `padsequences(X,'Length','shortest')`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`Direction` — Direction of padding or truncation
`'right'` (default) | `'left'` | `'both'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate at the end of each original sequence.
- `'left'` — Pad or truncate at the beginning of each original sequence.
- `'both'` — Pad or truncate at the beginning and end of each original sequence. Half the required padding or truncation is applied to each end of the sequence.

Example: `padsequences(X,'Direction','both')`

Data Types: `char` | `string`

`PaddingValue` — Value used to pad input
`'auto'` (default) | `'symmetric'` | numeric scalar | categorical scalar

Value used to pad input, specified as one of the following:

- `'auto'` — Determine the [adding value automatically depending on the data type of the input sequences. Numeric sequences are padded with `0`. Categorical sequences are padded with `<undefined>`.
- `'symmetric'` — Pad each sequence with a mirror reflection of itself.
- Numeric scalar — Pad each sequence with the specified numeric value.
- Categorical scalar — Pad each sequence with the specified categorical value.

Example: `padsequences(X,'PaddingValue','symmetric')`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

`UniformOutput` — Flag to return padded data as uniform array
`true` or `1` (default) | `false` or `0`

Flag to return padded data as a uniform array, specified as a numeric or logical `1` (`true`) or `0` (`false`). When you set the value to `0`, `XPad` is returned as a cell vector with the same size and underlying data type as the input `X`.

Example: `padsequences(X,'UniformOutput',0)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

XPad — Padded sequence data

numeric array | categorical array | cell vector

Padded sequence data, returned as a numeric array, categorical array, or a cell vector of numeric or categorical arrays.

If you set the `UniformOutput` name-value option to `true` or `1`, the function concatenates the padded sequences over the last dimension. The last dimension of `XPad` has the same size as the number of sequences in input `X`. `XPad` is an array with `N + 1` dimensions, where `N` is the number of dimensions of the sequence arrays in `X`. `XPad` has the same data type as the arrays in input `X`.

If you set the `UniformOutput` name-value option to `false` or `0`, the function returns the padded sequences as a cell vector with the same size and underlying data type as the input `X`.

mask — Position of original sequence data

logical array | cell vector

Position of original sequence data in the padded sequences, returned as a logical array or as a cell vector of logical arrays.

`mask` has the same size and data type as `XPad`. Values of `1` in `mask` correspond to positions of original sequence values in `XPad`. Values of `0` correspond to padded values.

Use `mask` to excluded padded values from loss calculations using the `"Mask"` name-value option in the `crossentropy` function.

See Also

`crossentropy` | `onehotencode` | `minibatchqueue`

Topics

“Sequence-to-Sequence Classification Using 1-D Convolutions”
“Long Short-Term Memory Networks”

Introduced in R2021a

partition

Partition minibatchqueue

Syntax

```
submbq = partition(mbq,numParts,indx)
```

Description

`submbq = partition(mbq,numParts,indx)` partitions the `minibatchqueue` object `mbq` into `numParts` parts and returns the partition corresponding to the index `indx`. The properties of `submbq` are the same as the properties of `mbq`.

The output `minibatchqueue` object has access only to the partition of data it is given when it is created. Using `reset` with `submbq` resets the `minibatchqueue` object to the start of the data partition. Using `shuffle` with `submbq` shuffles only the partitioned data. If you want to shuffle the data across multiple partitions, you must shuffle the original `minibatchqueue` object and then re-partition.

Examples

Partition `minibatchqueue`

Use the partition function to divide a `minibatchqueue` object into three parts.

Create a `minibatchqueue` object from a datastore.

```
ds = digitDatastore;
mbq = minibatchqueue(ds)
```

```
mbq =
minibatchqueue with 1 output and properties:

   Mini-batch creation:
           MiniBatchSize: 128
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
     DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

Partition the `minibatchqueue` object into three parts and return the first partition.

```
sub1 = partition(mbq,3,1)
```

```
sub1 =
minibatchqueue with 1 output and properties:
```

```
   Mini-batch creation:
           MiniBatchSize: 128
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
    DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

`sub1` contains approximately the first third of the data in `mbq`.

Partition `minibatchqueue` in Parallel

Use the `partition` function to divide a `minibatchqueue` object into three parts.

Create a `minibatchqueue` object from a datastore.

```
ds = digitDatastore;
mbq = minibatchqueue(ds)
```

```
mbq =
minibatchqueue with 1 output and properties:

   Mini-batch creation:
           MiniBatchSize: 128
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
    DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

Partition the `minibatchqueue` object into three parts on three workers in a parallel pool. Iterate over the data on each worker.

```
numWorkers = 3;
p = parpool('local',numWorkers);
parfor i=1:3
    submbq = partition(mbq,3,i);
    while hasdata(submbq)
        data = next(submbq);
    end
end
```

Each worker has access to a subset of the data in the original `minibatchqueue` object.

Input Arguments

`mbq` — Queue of mini-batches

`minibatchqueue`

Queue of mini-batches, specified as a `minibatchqueue` object.

`numParts` — Number of partitions

numeric scalar

Number of partitions, specified as a numeric scalar.

`indx` — Partition index

numeric scalar

Partition index, specified as a numeric scalar.

Output Arguments

`submbq` — Output `minibatchqueue`

`minibatchqueue`

Output `minibatchqueue`, specified as a `minibatchqueue` object. `submbq` contains a subset of the data in `mbq`. The properties of `submbq` are the same as the properties of `mbq`.

See Also

`shuffle` | `reset` | `minibatchqueue` | `next`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2020b

ONNXParameters

Parameters of imported ONNX network for deep learning

Description

`ONNXParameters` contains the parameters (such as weights and bias) of an imported ONNX (Open Neural Network Exchange) network. Use `ONNXParameters` to perform tasks such as transfer learning.

Creation

Create an `ONNXParameters` object by using `importONNXFunction`.

Properties

`Learnables` — Parameters updated during network training
structure

Parameters updated during network training, specified as a structure. For example, the weights of convolution and fully connected layers are parameters that the network learns during training. To prevent `Learnables` parameters from being updated during training, convert them to `Nonlearnables` by using `freezeParameters`. Convert frozen parameters back to `Learnables` by using `unfreezeParameters`.

Add a new parameter to `params.Learnables` by using `addParameter`. Remove a parameter from `params.Learnables` by using `removeParameter`.

Access the fields of the structure `Learnables` by using dot notation. For example, `params.Learnables.conv1_W` could display the weights of the first convolution layer. Initialize the weights for transfer learning by entering `params.Learnables.conv1_W = rand([1000,4096])`. For more details about assigning a new value and parameter naming, see “Tips” on page 1-1110.

`Nonlearnables` — Parameters unchanged during network training
structure

Parameters unchanged during network training, specified as a structure. For example, padding and stride are parameters that stay constant during training.

Add a new parameter to `params.Nonlearnables` by using `addParameter`. Remove a parameter from `params.Nonlearnables` by using `removeParameter`.

Access the fields of the structure `Nonlearnables` by using dot notation. For example, `params.Nonlearnables.conv1_Padding` could display the padding of the first convolution layer. For more details about parameter naming, see “Tips” on page 1-1110.

`State` — Network state
structure

Network state, specified as a structure. The network `State` contains information remembered by the network between iterations and updated across multiple training batches. For example, the states of LSTM and batch normalization layers are `State` parameters.

Add a new parameter to `params.State` by using `addParameter`. Remove a parameter from `params.State` by using `removeParameter`.

Access the fields of the structure `State` by using dot notation. For example, `params.State.bn1_var` could display the variance of the first batch normalization layer. For more details about parameter naming, see “Tips” on page 1-1110.

`NumDimensions` — Number of dimensions for every parameter
structure

This property is read-only.

Number of dimensions for every parameter, specified as a structure. `NumDimensions` includes trailing singleton dimensions.

Access the fields of the structure `NumDimensions` by using dot notation. For example, `params.NumDimensions.conv1_W` could display the number of dimensions for the weights parameter of the first convolution layer.

`NetworkFunctionName` — Name of model function
character vector | string scalar

This property is read-only.

Name of the model function, specified as a character vector or string scalar. The property `NetworkFunctionName` contains the name of the function `NetworkFunctionName`, which you specify in `importONNXFunction`. The function `NetworkFunctionName` contains the architecture of the imported ONNX network.

Example: `'shufflenetFcn'`

Object Functions

addParameter	Add parameter to ONNXParameters object
freezeParameters	Convert learnable network parameters in ONNXParameters to nonlearnable
removeParameter	Remove parameter from ONNXParameters object
unfreezeParameters	Convert nonlearnable network parameters in ONNXParameters to learnable

Examples

Train Imported ONNX Function Using Custom Training Loop

Import the `squeezenet` convolution neural network as a function and fine-tune the pretrained network with transfer learning to perform classification on a new collection of images.

This example uses several helper functions. To view the code for these functions, see Helper Functions on page 1-0 .

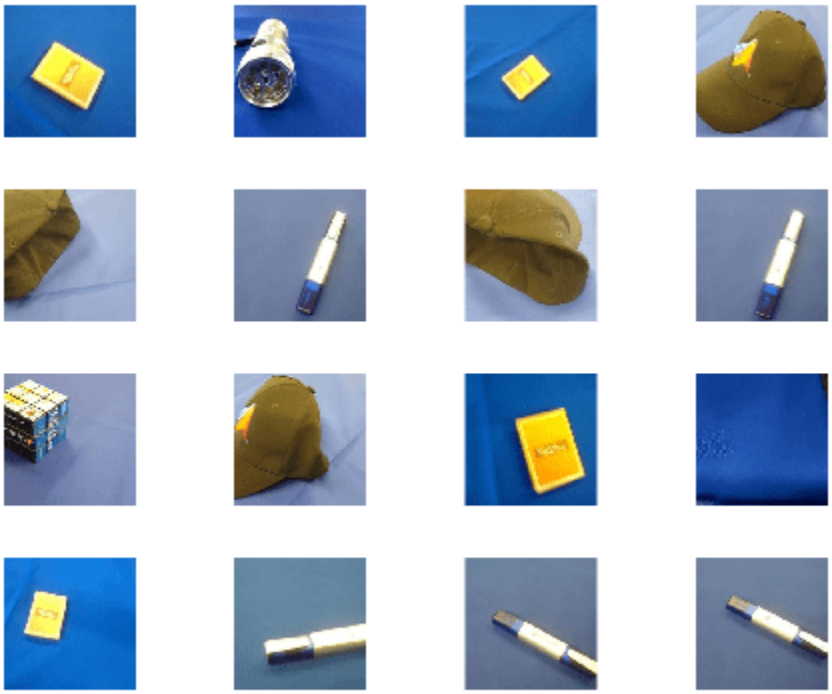
Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image

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. Specify the mini-batch size.

```
unzip('MerchData.zip');
miniBatchSize = 8;
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames',...
    'ReadSize', miniBatchSize);
```

This data set is small, containing 75 training images. Display some sample images.

```
numImages = numel(imds.Labels);
idx = randperm(numImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imds,idx(i));
    imshow(I)
end
```



Extract the training set and one-hot encode the categorical classification labels.

```
XTrain = readall(imds);
XTrain = single(cat(4,XTrain{:}));
YTrain_categ = categorical(imds.Labels);
YTrain = onehotencode(YTrain_categ,2)';
```

Determine the number of classes in the data.

```
classes = categories(YTrain_categ);
numClasses = numel(classes)
```

```
numClasses = 5
```

`squeezenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

Import the pretrained `squeezenet` network as a function.

```
squeezenetONNX()
params = importONNXFunction('squeezenet.onnx','squeezenetFcn')
```

```
A function containing the imported ONNX network has been saved to the file squeezenetFcn.m.
To learn how to use this function, type: help squeezenetFcn.

params =
  ONNXParameters with properties:

            Learnables: [1×1 struct]
          Nonlearnables: [1×1 struct]
                  State: [1×1 struct]
          NumDimensions: [1×1 struct]
    NetworkFunctionName: 'squeezenetFcn'
```

`params` is an `ONNXParameters` object that contains the network parameters. `squeezenetFcn` is a model function that contains the network architecture. `importONNXFunction` saves `squeezenetFcn` in the current folder.

Calculate the classification accuracy of the pretrained network on the new training set.

```
accuracyBeforeTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy before transfer learning\n',accuracyBeforeTraining);
```

```
0.01 accuracy before transfer learning
```

The accuracy is very low.

Display the learnable parameters of the network by typing `params.Learnables`. These parameters, such as the weights (`W`) and bias (`B`) of convolution and fully connected layers, are updated by the network during training. Nonlearnable parameters remain constant during training.

The last two learnable parameters of the pretrained network are configured for 1000 classes.

`conv10_W: [1×1×512×1000 dlarray]`

`conv10_B: [1000×1 dlarray]`

The parameters `conv10_W` and `conv10_B` must be fine-tuned for the new classification problem. Transfer the parameters to classify five classes by initializing the parameters.

```
params.Learnables.conv10_W = rand(1,1,512,5);
params.Learnables.conv10_B = rand(5,1);
```

Freeze all the parameters of the network to convert them to nonlearnable parameters. Because you do not need to compute the gradients of the frozen layers, freezing the weights of many initial layers can significantly speed up network training.

```
params = freezeParameters(params,'all');
```

Unfreeze the last two parameters of the network to convert them to learnable parameters.

```
params = unfreezeParameters(params,'conv10_W');
params = unfreezeParameters(params,'conv10_B');
```

Now the network is ready for training. Initialize the training progress plot.

```
plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end
```

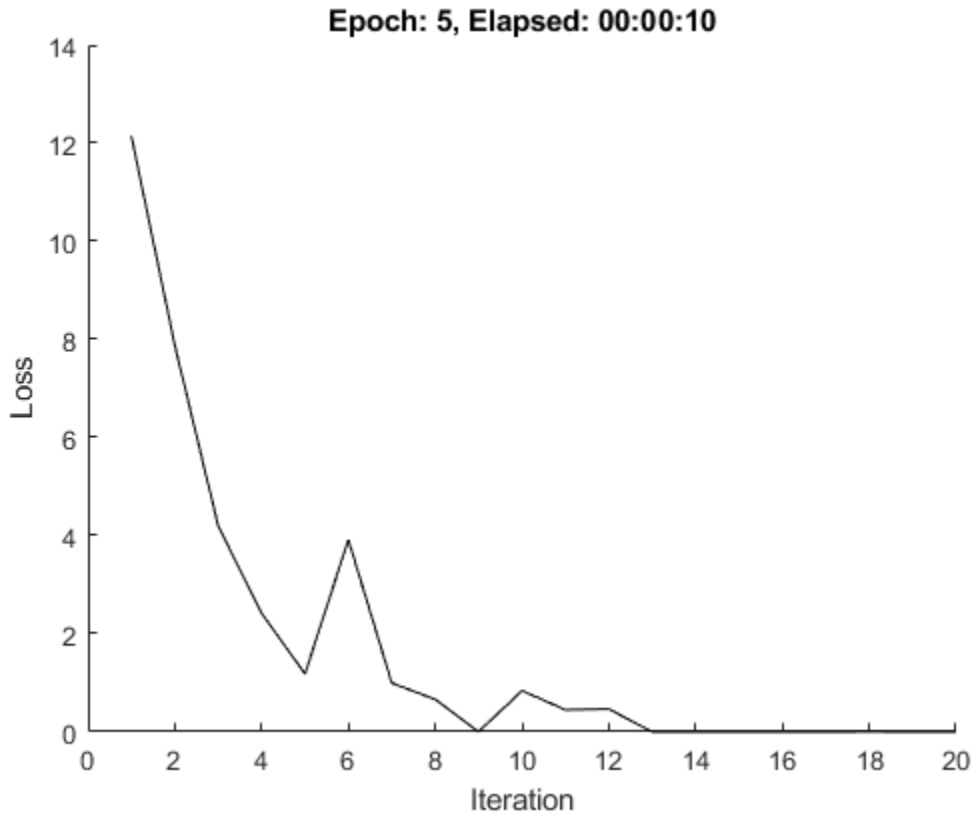
Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Train the network.

```
iteration = 0;
start = tic;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle 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.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);
        Y = YTrain(:,idx);

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

```
        end

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
        params.State = state;

        % Determine the learning rate for the time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);

        % 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
```



Calculate the classification accuracy of the network after fine-tuning.

```
accuracyAfterTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy after transfer learning\n',accuracyAfterTraining);
```

```
1.00 accuracy after transfer learning
```

Helper Functions

This section provides the code of the helper functions used in this example.

The `getNetworkAccuracy` function evaluates the network performance by calculating the classification accuracy.

```
function accuracy = getNetworkAccuracy(X,Y,onnxParams)

N = size(X,4);
Ypred = squeezenetFcn(X,onnxParams,'Training',false);

[~,YIdx] = max(Y,[],1);
[~,YpredIdx] = max(Ypred,[],1);
numIncorrect = sum(abs(YIdx-YpredIdx) > 0);
accuracy = 1 - numIncorrect/N;

end
```

The `modelGradients` function calculates the loss and gradients.

```
function [grad, loss, state] = modelGradients(X,Y,onnxParams)

[y,state] = squeezenetFcn(X,onnxParams,'Training',true);
loss = crossentropy(y,Y,'DataFormat','CB');
grad = dlgradient(loss,onnxParams.Learnables);

end
```

The `squeezenetONNX` function generates an ONNX model of the `squeezenet` network.

```
function squeezenetONNX()

exportONNXNetwork(squeezenet,'squeezenet.onnx');

end
```

Move Parameters Mislabeled by ONNX Functional Importer

Import a network saved in the ONNX format as a function, and move the mislabeled parameters by using `freeze` or `unfreeze`.

Import the pretrained `simplenet.onnx` network as a function. `simplenet` is a simple convolutional neural network trained on digit image data. For more information on how to create `simplenet`, see “Create Simple Image Classification Network”.

Import `simplenet.onnx` using `importONNXFunction`, which returns an `ONNXParameters` object that contains the network parameters. The function also creates a new model function in the current folder that contains the network architecture. Specify the name of the model function as `simplenetFcn`.

```
params = importONNXFunction('simplenet.onnx','simplenetFcn');
```

```
A function containing the imported ONNX network has been saved to the file simplenetFcn.m.
To learn how to use this function, type: help simplenetFcn.
```

`importONNXFunction` labels the parameters of the imported network as `Learnables` (parameters that are updated during training) or `Nonlearnables` (parameters that remain unchanged during training). The labeling is not always accurate. A recommended practice is to check if the parameters are assigned to the correct structure `params.Learnables` or `params.Nonlearnables`. Display the learnable and nonlearnable parameters of the imported network.

```
params.Learnables
```

```
ans = struct with fields:
    imageinput_Mean: [1×1 dlarray]
             conv_W: [5×5×1×20 dlarray]
             conv_B: [20×1 dlarray]
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
               fc_W: [24×24×20×10 dlarray]
               fc_B: [10×1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
            ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
            ConvStride1007: [2×1 dlarray]
    ConvDilationFactor1008: [2×1 dlarray]
           ConvPadding1009: [4×1 dlarray]
```

Note that `params.Learnables` contains the parameter `imageinput_Mean`, which should remain unchanged during training (see the `Mean` property of `imageInputLayer`). Convert `imageinput_Mean` to a nonlearnable parameter. The `freezeParameters` function removes the parameter `imageinput_Mean` from `param.Learnables` and adds it to `params.Nonlearnables` sequentially.

```
params = freezeParameters(params,'imageinput_Mean');
```

Display the updated learnable and nonlearnable parameters.

```
params.Learnables
```

```
ans = struct with fields:
             conv_W: [5×5×1×20 dlarray]
             conv_B: [20×1 dlarray]
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
               fc_W: [24×24×20×10 dlarray]
               fc_B: [10×1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
            ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
```

```
         ConvStride1007: [2×1 dlarray]
ConvDilationFactor1008: [2×1 dlarray]
       ConvPadding1009: [4×1 dlarray]
       imageinput_Mean: [1×1 dlarray]
```

Tips

- The following rules apply when you assign a new value to a `params.Learnables` parameter:
  - The software automatically converts the new value to a `dlarray`.
  - The new value must be compatible with the existing value of `params.NumDimensions`.
- `importONNXFunction` derives the field names of the structures `Learnables`, `Nonlearnables`, and `State` from the names in the imported ONNX model file. The field names might differ between imported networks.

See Also

`importONNXFunction`

Topics

“Make Predictions Using Model Function”
“Train Network Using Custom Training Loop”

Introduced in R2020b

partitionByIndex

Partition `augmentedImageDatastore` according to indices

Syntax

```
auimds2 = partitionByIndex(auimds,ind)
```

Description

`auimds2 = partitionByIndex(auimds,ind)` partitions a subset of observations in an augmented image datastore, `auimds`, into a new datastore, `auimds2`. The desired observations are specified by indices, `ind`.

Input Arguments

`auimds` — Augmented image datastore

`augmentedImageDatastore`

Augmented image datastore, specified as an `augmentedImageDatastore` object.

`ind` — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

`auimds2` — Output datastore

`augmentedImageDatastore` object

Output datastore, returned as an `augmentedImageDatastore` object containing a subset of files from `auimds`.

See Also

`read` | `readall` | `readByIndex`

Introduced in R2018a

PlaceholderLayer

Layer replacing an unsupported Keras or ONNX layer, or unsupported functionality from `functionToLayerGraph`

Description

`PlaceholderLayer` is a layer that `importKerasLayers` and `importONNXLayers` insert into a layer array or layer graph in place of an unsupported Keras or ONNX layer. It can also represent unsupported functionality from `functionToLayerGraph`.

Creation

Importing layers from a Keras or ONNX network that has layers that are not supported by Deep Learning Toolbox creates `PlaceholderLayer` objects. Also, when you create a layer graph using `functionToLayerGraph`, unsupported functionality leads to `PlaceholderLayer` objects.

Properties

`Name` — Layer name
character vector | string scalar

Layer name, specified as a character vector or a string scalar.

Data Types: `char` | `string`

`Description` — Layer description
character vector | string scalar

Layer description, specified as a character vector or a string scalar.

Data Types: `char` | `string`

`Type` — Layer type
character vector | string scalar

Layer type, specified as a character vector or a string scalar.

Data Types: `char` | `string`

`KerasConfiguration` — Keras configuration of layer
structure

Keras configuration of a layer, specified as a structure. The fields of the structure depend on the layer type.

Note This property only exists if the layer was created when importing a Keras network.

Data Types: `struct`

`ONNXNode` — ONNX configuration of layer

structure

ONNX configuration of a layer, specified as a structure. The fields of the structure depend on the layer type.

Note This property only exists if the layer was created when importing an ONNX network.

Data Types: `struct`

`Weights` — Imported weights

structure

Imported weights, specified as a structure.

Data Types: `struct`

Examples

Find and Explore Placeholder Layers

Specify the Keras network file to import layers from.

```
modelfile = 'digitsDAGnetwithnoise.h5';
```

Import the network architecture. The network includes some layer types that are not supported by Deep Learning Toolbox. The `importKerasLayers` function replaces each unsupported layer with a placeholder layer and returns a warning message.

```
lgraph = importKerasLayers(modelfile)
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning

lgraph =
  LayerGraph with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Display the imported layers of the network. Two placeholder layers replace the Gaussian noise layers in the Keras network.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                          Image Input             28x28x1 images
     2   'conv2d_1'                         Convolution             20 7x7 convolutions with st
     3   'conv2d_1_relu'                    ReLU                    ReLU
     4   'conv2d_2'                         Convolution             20 3x3 convolutions with st
```

```
     5   'conv2d_2_relu'                     ReLU                    ReLU
     6   'gaussian_noise_1'                  PLACEHOLDER LAYER       Placeholder for 'GaussianNo
     7   'gaussian_noise_2'                  PLACEHOLDER LAYER       Placeholder for 'GaussianNo
     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
```

Find the placeholder layers using `findPlaceholderLayers`. The output argument contains the two placeholder layers that `importKerasLayers` inserted in place of the Gaussian noise layers of the Keras network.

```
placeholders = findPlaceholderLayers(lgraph)
```

```
placeholders =
  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
```

Specify a name for each placeholder layer.

```
gaussian1 = placeholders(1);
gaussian2 = placeholders(2);
```

Display the configuration of each placeholder layer.

```
gaussian1.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000
```

```
gaussian2.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

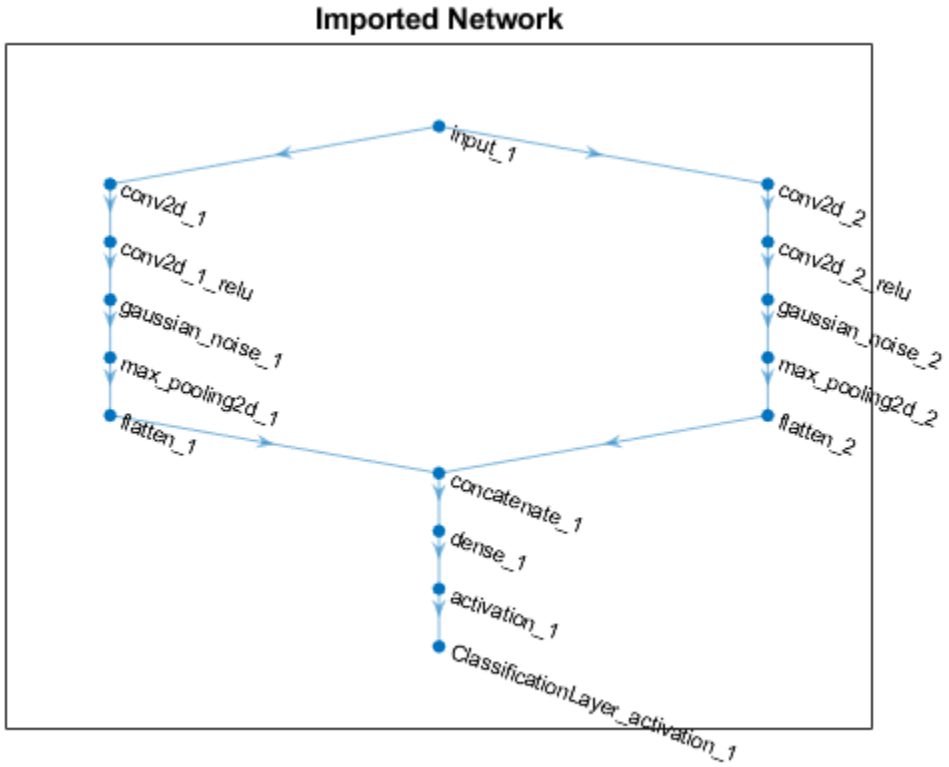
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

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



Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```
placeholderLayers.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

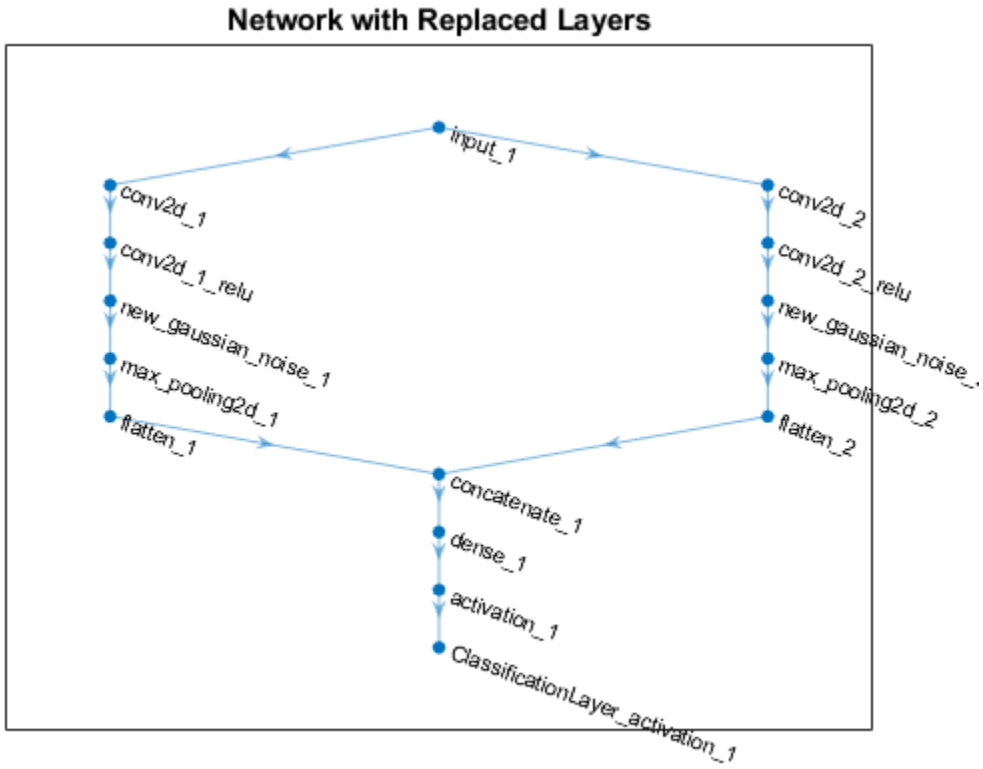
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```
figure
plot(lgraph)
title("Network with Replaced Layers")
```



Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                           Image Input             28x28x1 images
     2   'conv2d_1'                          Convolution             20 7x7x1 convolutions with
     3   'conv2d_1_relu'                     ReLU                    ReLU
     4   'conv2d_2'                          Convolution             20 3x3x1 convolutions with
     5   'conv2d_2_relu'                     ReLU                    ReLU
     6   'new_gaussian_noise_1'              Gaussian Noise          Gaussian noise with standar
     7   'new_gaussian_noise_2'              Gaussian Noise          Gaussian noise with standar
     8   'max_pooling2d_1'                   Max Pooling             2x2 max pooling with stride
     9   'max_pooling2d_2'                   Max Pooling             2x2 max pooling with stride
    10   'flatten_1'                         Keras Flatten           Flatten activations into 1-
    11   'flatten_2'                         Keras Flatten           Flatten activations into 1-
    12   'concatenate_1'                     Depth concatenation     Depth concatenation of 2 in
    13   'dense_1'                           Fully Connected         10 fully connected layer
    14   'activation_1'                      Softmax                 softmax
    15   'ClassificationLayer_activation_1'  Classification Output  crossentropyex
```

The classification layer has the name `'ClassificationLayer_activation_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

           Name: 'ClassificationLayer_activation_1'
        Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `'auto'`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9)
```

```
cLayer =
  ClassificationOutputLayer with properties:

           Name: 'ClassificationLayer_activation_1'
        Classes: [0    1    2    3    4    5    6    7    8    9]
    ClassWeights: 'none'
      OutputSize: 10

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

See Also

`importKerasLayers` | `importONNXLayers` | `findPlaceholderLayers` | `assembleNetwork` | `functionToLayerGraph` | `functionLayer`

Topics

“List of Deep Learning Layers”

“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

Introduced in R2017b

plot

Plot neural network layer graph

Syntax

```
plot(lgraph)
plot(net)
```

Description

`plot(lgraph)` plots a diagram of the layer graph `lgraph`. The `plot` function labels each layer by its name and displays all layer connections.

Tip To create an interactive network visualization and analyze the network architecture, use `deepNetworkDesigner(lgraph)`. For more information, see **Deep Network Designer**.

`plot(net)` plots a diagram of the network `net`.

Examples

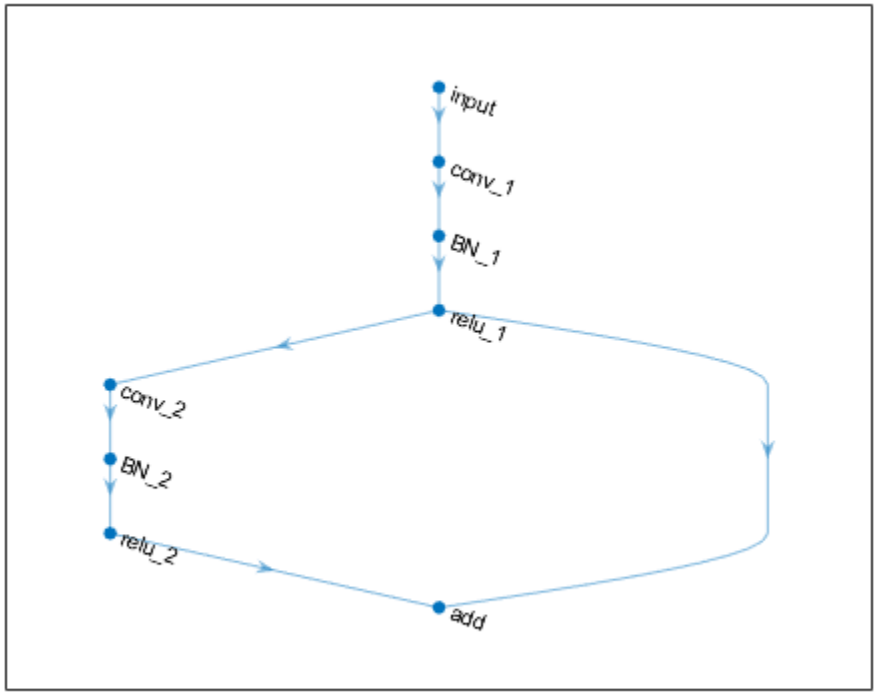
Plot Layer Graph

Create a layer graph from an array of layers. Connect the `'relu_1'` layer to the `'add'` layer.

```
layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')

    convolution2dLayer(3,16,'Padding','same','Stride',2,'Name','conv_2')
    batchNormalizationLayer('Name','BN_2')
    reluLayer('Name','relu_2')
    additionLayer(2,'Name','add')];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'relu_1','add/in2');
```

Plot the layer graph.

```
figure
plot(lgraph);
```



Plot DAG Network

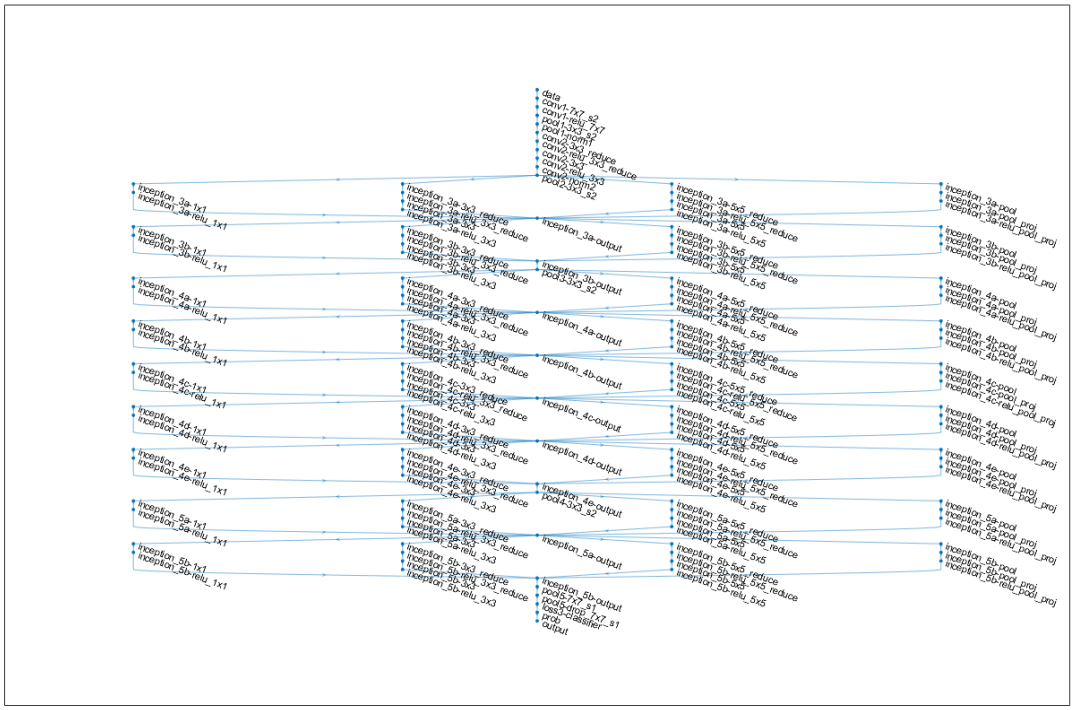
Load a pretrained GoogLeNet convolutional neural network as a `DAGNetwork` object. If the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

```
net = googlenet
```

```
net =
  DAGNetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
```

Plot the network.

```
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
plot(net)
```



Plot Series Network

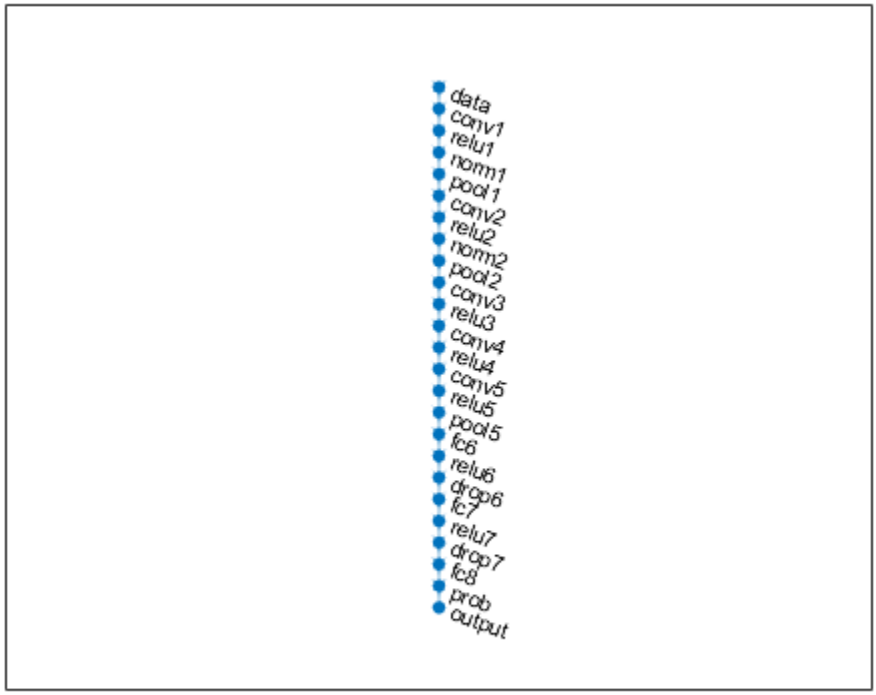
Load a pretrained AlexNet convolutional neural network as a `SeriesNetwork` object. If the Deep Learning Toolbox™ Model *for AlexNet Network* support package is not installed, then the software provides a download link.

```
net = alexnet
```

```
net =
  SeriesNetwork with properties:

         Layers: [25x1 nnet.cnn.layer.Layer]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Plot the network.

```
plot(net)
```



Input Arguments

`lgraph` — Layer graph

`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`net` — Network architecture

`SeriesNetwork` object | `DagNetwork`object

Network architecture, specified as a `SeriesNetwork` or a `DAGNetwork` object.

See Also

Deep Network Designer | `layerGraph` | `addLayers` | `removeLayers` | `replaceLayer` | `connectLayers` | `disconnectLayers` | `analyzeNetwork`

Topics

“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b

predict

Predict responses using a trained deep learning neural network

Syntax

```
YPred = predict(net,imds)
YPred = predict(net,ds)
YPred = predict(net,tbl)

YPred = predict(net,X)
YPred = predict(net,X1,...,XN)

[YPred1,...,YPredM] = predict( ___ )

YPred = predict(net,sequences)

 ___  = predict( ___ ,Name,Value)
```

Description

You can make predictions using a trained neural network for deep learning on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value pair argument.

`YPred = predict(net,imds)` predicts responses for the image data in `imds` using the trained `SeriesNetwork` or `DAGNetwork` object `net`. For `dlnetwork` input, see `predict`.

`YPred = predict(net,ds)` predicts responses for the data in the datastore `ds`.

`YPred = predict(net,tbl)` predicts responses for the data in the table `tbl`.

`YPred = predict(net,X)` predicts responses for the image or feature data in the numeric array `X`.

`YPred = predict(net,X1,...,XN)` predicts responses for the data in the numeric arrays `X1`, ..., `XN` for the mutli-input network `net`. The input `Xi` corresponds to the network input `net.InputNames(i)`.

`[YPred1,...,YPredM] = predict( ___ )` predicts responses for the `M` outputs of a multi-output network using any of the previous syntaxes. The output `YPredj` corresponds to the network output `net.OutputNames(j)`. To return categorical outputs for the classification output layers, set the `'ReturnCategorical'` option to `true`.

`YPred = predict(net,sequences)` predicts responses for the sequence or time series data in `sequences` using the trained recurrent network (for example, an LSTM or GRU network) `net`.

` ___  = predict( ___ ,Name,Value)` predicts responses with additional options specified by one or more name-value pair arguments.

Tip 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, respectively.

Examples

Predict Output Scores Using a Trained ConvNet

Load the sample data.

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

`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.

Construct the convolutional neural network architecture.

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

Set the options to default settings for the stochastic gradient descent with momentum.

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

Train the network.

```
rng('default')
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 |       10.16% |       2.3195 |          0.0100 |
|       2 |          50 |       00:00:03 |       50.78% |       1.7102 |          0.0100 |
|       3 |         100 |       00:00:06 |       63.28% |       1.1632 |          0.0100 |
|       4 |         150 |       00:00:09 |       60.16% |       1.0859 |          0.0100 |
|       6 |         200 |       00:00:13 |       68.75% |       0.8996 |          0.0100 |
|       7 |         250 |       00:00:16 |       76.56% |       0.7919 |          0.0100 |
|       8 |         300 |       00:00:20 |       73.44% |       0.8411 |          0.0100 |
|       9 |         350 |       00:00:24 |       81.25% |       0.5514 |          0.0100 |
|      11 |         400 |       00:00:27 |       90.62% |       0.4744 |          0.0100 |
|      12 |         450 |       00:00:31 |       92.19% |       0.3614 |          0.0100 |
|      13 |         500 |       00:00:35 |       94.53% |       0.3159 |          0.0100 |
|      15 |         550 |       00:00:38 |       96.09% |       0.2543 |          0.0100 |
|      16 |         600 |       00:00:42 |       92.19% |       0.2765 |          0.0100 |
|      17 |         650 |       00:00:45 |       95.31% |       0.2461 |          0.0100 |
```

```
|      18 |          700 |       00:00:48 |       99.22% |       0.1418 |          0.0100 |
|      20 |          750 |       00:00:52 |       98.44% |       0.1000 |          0.0100 |
|      21 |          800 |       00:00:55 |       98.44% |       0.1448 |          0.0100 |
|      22 |          850 |       00:00:59 |       98.44% |       0.0989 |          0.0100 |
|      24 |          900 |       00:01:02 |       96.88% |       0.1316 |          0.0100 |
|      25 |          950 |       00:01:06 |      100.00% |       0.0859 |          0.0100 |
|      26 |         1000 |       00:01:09 |      100.00% |       0.0701 |          0.0100 |
|      27 |         1050 |       00:01:12 |      100.00% |       0.0759 |          0.0100 |
|      29 |         1100 |       00:01:16 |       99.22% |       0.0663 |          0.0100 |
|      30 |         1150 |       00:01:19 |       98.44% |       0.0775 |          0.0100 |
|      30 |         1170 |       00:01:20 |       99.22% |       0.0732 |          0.0100 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Run the trained network on a test set and predict the scores.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);
```

`predict`, by default, uses a CUDA® enabled GPU with compute capability 3.0, when available. You can also choose to run `predict` on a CPU using the `'ExecutionEnvironment','cpu'` name-value pair argument.

Display the first 10 images in the test data and compare to the predictions from `predict`.

```
YTest(1:10,:)
```

```
ans = 10x1 categorical
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

```
YPred(1:10,:)
```

```
ans = 10x10 single matrix

    0.9978    0.0001    0.0008    0.0002    0.0003    0.0000    0.0004    0.0000    0.0002    0.0
    0.8883    0.0000    0.0472    0.0001    0.0000    0.0002    0.0029    0.0001    0.0014    0.0
    0.9998    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0
    0.9814    0.0000    0.0000    0.0000    0.0000    0.0000    0.0046    0.0000    0.0011    0.0
    0.9748    0.0000    0.0132    0.0003    0.0000    0.0000    0.0002    0.0004    0.0112    0.0
    0.9872    0.0000    0.0001    0.0000    0.0000    0.0000    0.0007    0.0000    0.0072    0.0
    0.9981    0.0000    0.0000    0.0000    0.0000    0.0000    0.0018    0.0000    0.0000    0.0
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0
    0.9270    0.0000    0.0046    0.0000    0.0006    0.0009    0.0001    0.0000    0.0018    0.0
    0.9328    0.0000    0.0139    0.0012    0.0001    0.0001    0.0378    0.0000    0.0110    0.0
```

`YTest` contains the digits corresponding to the images in `XTest`. The columns of `YPred` contain `predict`'s estimation of a probability that an image contains a particular digit. That is, the first

column contains the probability estimate that the given image is digit 0, the second column contains the probability estimate that the image is digit 1, the third column contains the probability estimate that the image is digit 2, and so on. You can see that `predict`'s estimation of probabilities for the correct digits are almost 1 and the probability for any other digit is almost 0. `predict` correctly estimates the first 10 observations as digit 0.

Predict Output Scores Using a Trained LSTM Network

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;
```

Make predictions on the test data.

```
YPred = predict(net,XTest);
```

View the prediction scores for the first 10 sequences.

```
YPred(1:10,:)
```

```
ans = 10x9 single matrix

    0.9918    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0006    0.0059
    0.9868    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0010    0.0105
    0.9924    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0006    0.0054
    0.9896    0.0000    0.0000    0.0000    0.0006    0.0009    0.0001    0.0007    0.0080
    0.9965    0.0000    0.0000    0.0000    0.0007    0.0009    0.0000    0.0003    0.0016
    0.9888    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0087
    0.9886    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0089
    0.9982    0.0000    0.0000    0.0000    0.0006    0.0007    0.0000    0.0001    0.0004
    0.9883    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0093
    0.9959    0.0000    0.0000    0.0000    0.0007    0.0011    0.0000    0.0004    0.0019
```

Compare these prediction scores to the labels of these sequences. The function assigns high prediction scores to the correct class.

```
YTest(1:10)

ans = 10x1 categorical
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
```

Input Arguments

net — Trained network

SeriesNetwork object | DAGNetwork object

Trained network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network (for example, by using the googlenet function) or by training your own network using trainNetwork.

imds — Image datastore

ImageDatastore object

Image datastore, specified as an ImageDatastore object.

ImageDatastore allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then ImageDatastore does not prefetch.

Tip Use augmentedImageDatastore for efficient preprocessing of images for deep learning including image resizing.

Do not use the readFcn option of imageDatastore for preprocessing or resizing as this option is usually significantly slower.

---

ds — Datastore

datastore

Datastore for out-of-memory data and preprocessing. The datastore must return data in a table or a cell array. The format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output

Single input	Table or cell array, where the first column specifies the predictors.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom datastores must output tables.
		`data = read(ds)`
		
		`data =`
		
		`  4×1 table`
		
		`  Predictors`
		`  ____________________`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

		`data = read(ds)`
		
		`data =`
		
		`  4×1 cell array`
		
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`
		`  {224×224×3 double}`

Multiple input	Cell array with at least `numInputs` columns, where `numInputs` is the number of network inputs.
	
	The first `numInputs` columns specify the predictors for each input.
	
	The order of inputs is given by the `InputNames` property of the network.
		`data = read(ds)`
		
		`data =`
		
		`  4×2 cell array`
		
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`
		`  {224×224×3 double}  {128×128×3 do`

The format of the predictors depend on the type of data.

Data	Format of Predictors
2-D image	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the image, respectively.
3-D image	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the image, respectively.
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.

Data	Format of Predictors
1-D image sequence	*h*-by-*c*-by-*s* array, where *h* and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
Features	*c*-by-1 column vector, where *c* is the number of features.

For more information, see “Datastores for Deep Learning”.

X — Image or feature data
numeric array

Image or feature data, specified as a numeric array. The size of the array depends on the type of input:

Input	Description
2-D images	A *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
3-D images	A *h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.
Features	A *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

If the array contains `NaN`s, then they are propagated through the network.

For networks with multiple inputs, you can specify multiple arrays `X1`, ..., `XN`, where `N` is the number of network inputs and the input `Xi` corresponds to the network input `net.InputNames(i)`.

sequences — Sequence or time series data
cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

`tbl` — Table of image or feature data
`table`

Table of image or feature data. Each row in the table corresponds to an observation.

The arrangement of predictors in the table columns depend on the type of input data.

Input	Predictors
Image data	• Absolute or relative file path to an image, specified as a character vector in a single column • Image specified as a 3-D numeric array Specify predictors in a single column.
Feature data	Numeric scalar. Specify predictors in the first `numFeatures` columns of the table, where `numFeatures` is the number of features of the input data.

This argument supports networks with a single input only.

Data Types: `table`

Name-Value Pair Arguments

Specify optional comma-separated pair of `Name,Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`).

Example: `'MiniBatchSize',256` specifies the mini-batch size as 256.

`MiniBatchSize` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

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, respectively.

Example: `'MiniBatchSize',256`

`Acceleration` — Performance optimization

`'auto'` (default) | `'mex'` | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'mex'` — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'none'` — Disable all acceleration.

The default option is `'auto'`. If `'auto'` is specified, MATLAB will apply a number of compatible optimizations. If you use the `'auto'` option, MATLAB does not ever generate a MEX function.

Using the `'Acceleration'` options `'auto'` and `'mex'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The `'mex'` option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The `'mex'` option is only available when you are using a GPU. MEX acceleration supports single GPU execution using the name-value option `'ExecutionEvironment','gpu'` only.

To use the `'mex'` option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The `'mex'` option does not support all layers. For a list of supported layers, see “Supported Layers” (GPU Coder). Only networks with an `imageInputLayer` are supported.

You cannot use MATLAB Compiler to deploy your network when using the `'mex'` option.

Example: `'Acceleration','mex'`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following:

- `'auto'` — Use a GPU if one is available; otherwise, use the CPU.
- `'gpu'` — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'cpu'` — Use the CPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform computation. If the pool does not have GPUs, then computation takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

The `'multi-gpu'` and `'parallel'` options do not support recurrent neural networks (RNNs) containing `lstmLayer`, `bilstmLayer`, or `gruLayer` objects.

Example: `'ExecutionEnvironment','cpu'`

`ReturnCategorical` — Option to return categorical labels
`false` (default) | `true`

Option to return categorical labels, specified as `true` or `false`.

If `ReturnCategorical` is `true`, then the function returns categorical labels for classification output layers. Otherwise, the function returns the prediction scores for classification output layers.

`SequenceLength` — Option to pad, truncate, or split input sequences
`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.

- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

Example: `'SequenceLength','shortest'`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

Output Arguments

`YPred` — Predicted scores or responses
matrix | 4-D numeric array | cell array of matrices

Predicted scores or responses, returned as a matrix, a 4-D numeric array, or a cell array of matrices. The format of `YPred` depends on the type of problem.

The following table describes the format for classification problems.

Task	Format
Image classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes
Sequence-to-label classification	
Feature classification	
Sequence-to-sequence classification	*N*-by-1 cell array of matrices, where *N* is the number of observations. The sequences are matrices with *K* rows, where *K* is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

The following table describes the format for regression problems.

Task	Format
2-D image regression	• *N*-by-*R* matrix, where *N* is the number of images and *R* is the number of responses. • *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
3-D image regression	• *N*-by-*R* matrix, where *N* is the number of images and *R* is the number of responses. • *h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.
Sequence-to-one regression	*N*-by-*R* matrix, where *N* is the number of sequences and *R* is the number of responses.
Sequence-to-sequence regression	*N*-by-1 cell array of numeric sequences, where *N* is the number of sequences. The sequences are matrices with *R* rows, where *R* is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently. For sequence-to-sequence regression tasks with one observation, `sequences` can be a matrix. In this case, `YPred` is a matrix of responses.
Feature regression	*N*-by-*R* matrix, where *N* is the number of observations and *R* is the number of responses.

For sequence-to-sequence regression problems with one observation, `sequences` can be a matrix. In this case, `YPred` is a matrix of responses.

Algorithms

If the image data contains `NaN`s, `predict` propagates them through the network. If the network has ReLU layers, these layers ignore `NaN`s. However, if the network does not have a ReLU layer, then `predict` returns NaNs as predictions.

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

Alternatives

You can compute the predicted scores and the predicted classes from a trained network using `classify`.

You can also compute the activations from a network layer using `activations`.

For sequence-to-label and sequence-to-sequence classification networks (for example, LSTM networks), you can make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - `YPred = predict(net,X)`
  - `[YPred1,...,YPredM] = predict(__)`
  - `YPred = predict(net,sequences)`
  - `__ = predict(__,Name,Value)`
- The input `X` must not have a variable size. The size must be fixed at code generation time.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.

- Only the `'MiniBatchSize'`, `'ReturnCategorical'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- If `'ReturnCategorical'` is set to `true` and you use a GCC C/C++ compiler version 8.2 or above, you might get a `-Wstringop-overflow` warning.
- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

For more information about generating code for deep learning neural networks, see “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:
  - `YPred = predict(net,X)`
  - `[YPred1,...,YPredM] = predict(__)`
  - `YPred = predict(net,sequences)`
  - `__ = predict(__,Name,Value)`
- The input `X` must not have variable size. The size must be fixed at code generation time.
- GPU code generation does not support `gpuArray` inputs to the `predict` function.
- The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences. The ARM `Compute Library` for GPU does not support recurrent networks.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'ReturnCategorical'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- GPU code generation for the `predict` function supports inputs that are defined as half-precision floating point data types. For more information, see `half`.
- If `'ReturnCategorical'` is set to `true` and you use a GCC C/C++ compiler version 8.2 or above, you might get a `-Wstringop-overflow` warning.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run computations in parallel, set the `'ExecutionEnvironment'` option to `'multi-gpu'` or `'parallel'`.

For details, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- When input data is a `gpuArray`, a cell array or table containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`activations` | `classify` | `classifyAndUpdateState` | `predictAndUpdateState`

Introduced in R2016a

predictAndUpdateState

Predict responses using a trained recurrent neural network and update the network state

Syntax

```
[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)
[updatedNet,YPred] = predictAndUpdateState( ___ ,Name,Value)
```

Description

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Specify the hardware requirements using the `'ExecutionEnvironment'` name-value pair argument.

`[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)` predicts responses for data in `sequences` using the trained recurrent neural network `recNet` and updates the network state.

This function supports recurrent neural networks only. The input `recNet` must have at least one recurrent layer.

`[updatedNet,YPred] = predictAndUpdateState( ___ ,Name,Value)` uses any of the arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. For example, `'MiniBatchSize',27` makes predictions using mini-batches of size 27.

Tip 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, respectively.

Examples

Predict and Update Network State

Predict responses using a trained recurrent neural network and update the network state.

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

```
ans =
  5x1 Layer array with layers:
```

```
     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load the test data.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Loop over the time steps in a sequence. Predict the scores of each time step and update the network state.

```
X = XTest{94};
numTimeSteps = size(X,2);
for i = 1:numTimeSteps
    v = X(:,i);
    [net,score] = predictAndUpdateState(net,v);
    scores(:,i) = score;
end
```

Plot the prediction scores. The plot shows how the prediction scores change between time steps.

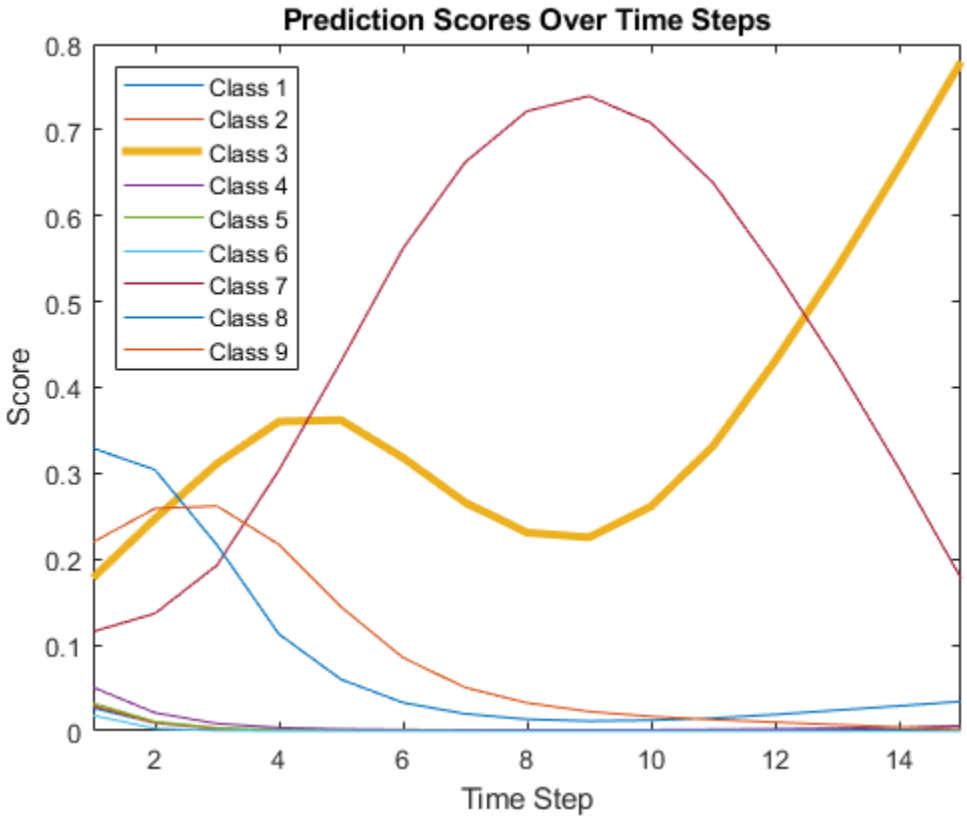
```
classNames = string(net.Layers(end).Classes);
figure
lines = plot(scores');
xlim([1 numTimeSteps])
legend("Class " + classNames,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
title("Prediction Scores Over Time Steps")
```

Highlight the prediction scores over time steps for the correct class.

```
trueLabel = YTest(94)
```

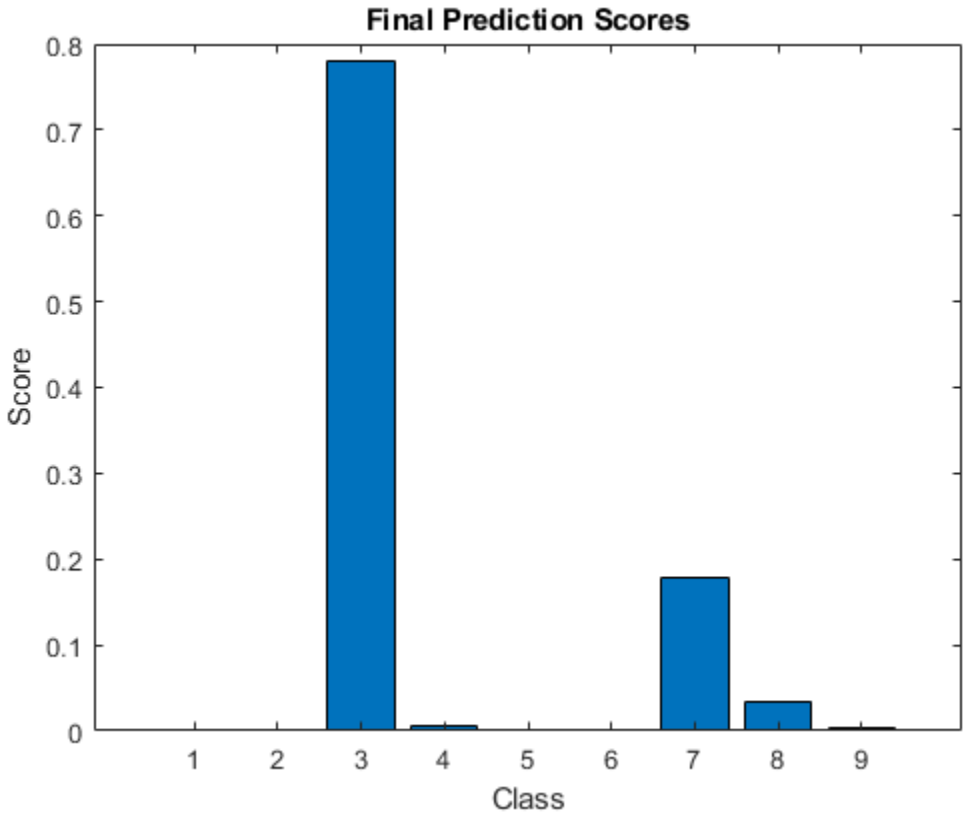
```
trueLabel = categorical
     3
```

```
lines(trueLabel).LineWidth = 3;
```



Display the final time step prediction in a bar chart.

```
figure
bar(score)
title("Final Prediction Scores")
xlabel("Class")
ylabel("Score")
```



Input Arguments

`recNet` — Trained recurrent neural network

`SeriesNetwork` object | `DAGNetwork` object

Trained recurrent neural network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function.

`recNet` is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network).

`sequences` — Sequence or time series data

cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[updatedNet, YPred] = predictAndUpdateState(recNet,C,'MiniBatchSize',27)` makes predictions using mini-batches of size 27.

`MiniBatchSize` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

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, respectively.

Example: `'MiniBatchSize',256`

`Acceleration` — Performance optimization

`'auto'` (default) | `'none'`

Performance optimization, specified as the comma-separated pair consisting of `'Acceleration'` and one of the following:

- `'auto'` — Automatically apply a number of optimizations suitable for the input network and hardware resources.
- `'none'` — Disable all acceleration.

The default option is `'auto'`.

Using the `'Acceleration'` option `'auto'` can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

Example: `'Acceleration','auto'`

`ExecutionEnvironment` — Hardware resource
`'auto'` (default) | `'gpu'` | `'cpu'`

Hardware resource, specified as the comma-separated pair consisting of `'ExecutionEnvironment'` and one of the following:

- `'auto'` — Use a GPU if one is available; otherwise, use the CPU.
- `'gpu'` — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `'cpu'` — Use the CPU.

Example: `'ExecutionEnvironment','cpu'`

`SequenceLength` — Option to pad, truncate, or split input sequences
`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: `'SequenceLength','shortest'`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

Output Arguments

`updatedNet` — Updated network
`SeriesNetwork` object | `DAGNetwork` object

Updated network. `updatedNet` is the same type of network as the input network.

`YPred` — Predicted scores or responses
matrix | cell array of matrices

Predicted scores or responses, returned as a matrix or a cell array of matrices. The format of `YPred` depends on the type of problem.

The following table describes the format for classification problems.

Task	Format
Sequence-to-label classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.
Sequence-to-sequence classification	*N*-by-1 cell array of matrices, where *N* is the number of observations. The sequences are matrices with *K* rows, where *K* is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

For sequence-to-sequence classification problems with one observation, `sequences` can be a matrix. In this case, `YPred` is a *K*-by-*S* matrix of scores, where *K* is the number of classes, and *S* is the total number of time steps in the corresponding input sequence.

The following table describes the format for regression problems.

Task	Format
Sequence-to-one regression	*N*-by-*R* matrix, where *N* is the number of observations and *R* is the number of responses.
Sequence-to-sequence regression	*N*-by-1 cell array of numeric sequences, where *N* is the number of observations. The sequences are matrices with *R* rows, where *R* is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently. For sequence-to-sequence problems with one observation, `sequences` can be a matrix. In this case, `YPred` is a matrix of responses.

Algorithms

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C++ code generation supports the following syntaxes:
  - `[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)`
  - `[updatedNet,YPred] = predictAndUpdateState(__,Name,Value)`
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.

- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.
- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation supports the following syntaxes:
  - `[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)`
  - `[updatedNet,YPred] = predictAndUpdateState(__,Name,Value)`
- GPU code generation for the `predictAndUpdateState` function is only supported for recurrent neural networks and cuDNN target library.
- GPU code generation does not support `gpuArray` inputs to the `predictAndUpdateState` function.
- The cuDNN library supports vector and 2-D image sequences.
- For vector sequence inputs, the number of features must be a constant during code generation. The sequence length can be variable sized.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- When input data is a `gpuArray`, a cell array containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `resetState` | `classify` | `predict`

Topics
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Deep Learning in MATLAB”

Introduced in R2017b

read

Read data from `augmentedImageDatastore`

Syntax

```
data = read(auimds)
[data,info] = read(auimds)
```

Description

`data = read(auimds)` returns a batch of data from an augmented image datastore, `auimds`. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(auimds)` also returns information about the extracted data, including metadata, in `info`.

Input Arguments

`auimds` — Augmented image datastore

`augmentedImageDatastore`

Augmented image datastore, specified as an `augmentedImageDatastore` object. The datastore specifies a `MiniBatchSize` number of observations in each batch, and a `numObservations` total number of observations.

Output Arguments

`data` — Output data

table

Output data, returned as a table with `MiniBatchSize` number of rows.

For the last batch of data in the datastore `auimds`, if `numObservations` is not cleanly divisible by `MiniBatchSize`, then `read` returns a partial batch containing all the remaining observations in the datastore.

`info` — Information about read data

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
`Filename`	`Filename` is a fully resolved path containing the path string, name of the file, and file extension.
`FileSize`	Total file size, in bytes. For MAT-files, `FileSize` is the total number of key-value pairs in the file.

See Also

`read (Datastore)` | `readByIndex` | `readall`

Introduced in R2018a

readByIndex

Read data specified by index from `augmentedImageDatastore`

Syntax

```
data = readByIndex(auimds,ind)
[data,info] = readByIndex(auimds,ind)
```

Description

`data = readByIndex(auimds,ind)` returns a subset of observations from an augmented image datastore, `auimds`. The desired observations are specified by indices, `ind`.

`[data,info] = readByIndex(auimds,ind)` also returns information about the observations, including metadata, in `info`.

Input Arguments

`auimds` — Augmented image datastore

`augmentedImageDatastore`

Augmented image datastore, specified as an `augmentedImageDatastore` object.

`ind` — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

`data` — Observations from datastore

table

Observations from the datastore, returned as a table with `length(ind)` number of rows.

`info` — Information about read data

structure array

Information about read data, returned as a structure array with the following fields.

Field Name	Description
`MiniBatchIndices`	Numeric vector of indices.

See Also

`read` | `readall` | `partitionByIndex`

Introduced in R2018a

recordMetrics

Package: `experiments`

Record metric values in experiment results table and training plot

Syntax

```
recordMetrics(monitor,xValue,metricName=yValue)
recordMetrics(monitor,xValue,metricName1=yValue1,...,metricNameN=yValueN)
recordMetrics(monitor,xValue,structure)
```

Description

`recordMetrics(monitor,xValue,metricName=yValue)` records the specified metric value for a trial in the **Experiment Manager** results table and training plot.

`recordMetrics(monitor,xValue,metricName1=yValue1,...,metricNameN=yValueN)` records multiple metric values for a trial.

`recordMetrics(monitor,xValue,structure)` records the metric values specified by the structure `structure`.

Examples

Track Progress, Display Information and Record Metric Values, and Produce Training Plots

Use an `experiments.Monitor` object to track the progress of the training, display information and metric values in the experiment results table, and produce training plots for custom training experiments.

Before starting the training, specify the names of the information and metric columns of the Experiment Manager results table.

```
monitor.Info = ["GradientDecayFactor","SquaredGradientDecayFactor"];
monitor.Metrics = ["TrainingLoss","ValidationLoss"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot.

```
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
```

Update the values of the gradient decay factor and the squared gradient decay factor for the trial in the results table.

```
updateInfo(monitor, ...
    GradientDecayFactor=gradientDecayFactor, ...
    SquaredGradientDecayFactor=squaredGradientDecayFactor);
```

After each iteration of the custom training loop, record the value of training and validation loss for the trial in the results table and the training plot.

```
recordMetrics(monitor,iteration, ...
    TrainingLoss=trainingLoss, ...
    ValidationLoss=validationLoss);
```

Update the training progress for the trial based on the fraction of iterations completed.

```
monitor.Progress = (iteration/numIterations) * 100;
```

Specify Metric Values by Using Structure

Use a structure to record metric values in the results table and the training plot.

```
structure.TrainingLoss = trainingLoss;
structure.ValidationLoss = validationLoss);
recordMetrics(monitor,iteration,structure);
```

Input Arguments

`monitor` — Experiment monitor
`experiments.Monitor` object

Experiment monitor for the trial, specified as an `experiments.Monitor` object. When you run a custom training experiment, Experiment Manager passes this object as the second input argument of the training function.

`xValue` — x-coordinate
numeric scalar

x-coordinate for the training plot, specified as a numeric scalar. Use this value to record the custom training loop iteration or epoch number for your data points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`metricName` — Metric name
string | character vector

Metric name, specified as a string or character vector. This name must be an element of the `Metrics` property of the `experiments.Monitor` object `monitor`.

Data Types: `char` | `string`

`yValue` — Metric value
numeric scalar

Metric value, specified as a numeric scalar. Experiment Manager uses this value as the y-coordinate for the training plot.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`structure` — Metric names and values
structure

Metric names and values, specified as a structure. Names must be elements of the `Metrics` property of the `experiments.Monitor` object `monitor` and can appear in any order in the structure.

Example: `struct(TrainingLoss=trainingLoss,ValidationLoss=validationLoss)`

Data Types: `struct`

Tips

- Both information and metric columns display values in the results table for your experiment. Additionally, the training plot shows a record of the metric values. Use information columns for text and for numerical values that you want to display in the results table but not in the training plot.
- Use the `groupSubPlot` function to define your training subplots before calling the function `recordMetrics`.

See Also

Apps
Experiment Manager

Objects
`experiments.Monitor`

Functions
`groupSubPlot` | `struct` | `updateInfo`

Introduced in R2021a

regressionLayer

Create a regression output layer

Syntax

```
layer = regressionLayer
layer = regressionLayer(Name,Value)
```

Description

A regression layer computes the half-mean-squared-error loss for regression tasks.

`layer = regressionLayer` returns a regression output layer for a neural network as a `RegressionOutputLayer` object.

Predict responses of a trained regression network using `predict`. Normalizing the responses often helps stabilizing and speeding up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

`layer = regressionLayer(Name,Value)` sets the optional `Name` and `ResponseNames` properties using name-value pairs. For example, `regressionLayer('Name','output')` creates a regression layer with the name `'output'`. Enclose each property name in single quotes.

Examples

Create Regression Output Layer

Create a regression output layer with the name `'routput'`.

```
layer = regressionLayer('Name','routput')
```

```
layer =
  RegressionOutputLayer with properties:

             Name: 'routput'
    ResponseNames: {}

   Hyperparameters
     LossFunction: 'mean-squared-error'
```

The default loss function for regression is mean-squared-error.

Include a regression output layer in a Layer array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Image Input         28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution         25 12x12 convolutions with stride [1  1] and padding [0  0  0
     3   ''   ReLU                ReLU
     4   ''   Fully Connected     1 fully connected layer
     5   ''   Regression Output   mean-squared-error
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `regressionLayer('Name','output')` creates a regression layer with the name `'output'`

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`ResponseNames` — Names of responses
`{}` (default) | cell array of character vectors | string array

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 `{}`.

Data Types: `cell`

Output Arguments

`layer` — Regression output layer
`RegressionOutputLayer` object

Regression output layer, returned as a `RegressionOutputLayer` object.

More About

Regression Output Layer

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

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

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

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

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

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

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

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

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

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

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

where S is the sequence length.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`RegressionOutputLayer` | `fullyConnectedLayer` | `classificationLayer`

Topics

"Deep Learning in MATLAB"
"Train Convolutional Neural Network for Regression"

Introduced in R2017a

RegressionOutputLayer

Regression output layer

Description

A regression layer computes the half-mean-squared-error loss for regression tasks.

Creation

Create a regression output layer using `regressionLayer`.

Properties

Regression Output

`ResponseNames` — Names of responses
`{}` (default) | cell array of character vectors | string array

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 `{}`.

Data Types: `cell`

`LossFunction` — Loss function for training
`'mean-squared-error'`

Loss function the software uses for training, specified as `'mean-squared-error'`.

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
0 (default)

Number of outputs of the layer. The layer has no outputs.

Data Types: `double`

`OutputNames` — Output names
`{}` (default)

Output names of the layer. The layer has no outputs.

Data Types: `cell`

Examples

Create Regression Output Layer

Create a regression output layer with the name `'routput'`.

```
layer = regressionLayer('Name','routput')
```

```
layer =
  RegressionOutputLayer with properties:

             Name: 'routput'
    ResponseNames: {}

   Hyperparameters
     LossFunction: 'mean-squared-error'
```

The default loss function for regression is mean-squared-error.

Include a regression output layer in a Layer array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Image Input         28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution         25 12x12 convolutions with stride [1  1] and padding [0  0  0
     3   ''   ReLU                ReLU
     4   ''   Fully Connected     1 fully connected layer
     5   ''   Regression Output   mean-squared-error
```

More About

Regression Output Layer

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

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

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

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

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

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

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

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

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

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

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

where S is the sequence length.

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

See Also

`trainNetwork` | `regressionLayer` | `classificationLayer` | `fullyConnectedLayer`

Topics

"Create Simple Deep Learning Network for Classification"
"Train Convolutional Neural Network for Regression"

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2017a

reset

Reset minibatchqueue to start of data

Syntax

```
reset(mbq)
```

Description

`reset(mbq)` resets `mbq` back to the start of the underlying datastore.

Examples

Reset `minibatchqueue` and Obtain More Mini-Batches

You can call `next` on a `minibatchqueue` object until all data is returned. When you reach the end of the data, use `reset` to reset the `minibatchqueue` object and continue obtaining mini-batches with `next`.

Create a `minibatchqueue` object from a datastore.

```
ds = digitDatastore;
mbq = minibatchqueue(ds,'MinibatchSize',256)
```

```
mbq =
minibatchqueue with 1 output and properties:

   Mini-batch creation:
           MiniBatchSize: 256
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
     DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

Iterate over all data in the `minibatchqueue` object. Use `hasdata` to check if data is still available.

```
while hasdata(mbq)
    [~] = next(mbq);
end
```

When `hasdata` returns `0` (`false`), you cannot collect a mini-batch using `next`.

```
hasdata(mbq)
```

```
ans =
   0
```

```
X = next(mbq);

Error using minibatchqueue/next (line 353)
Unable to provide a mini-batch because end of data reached. Use reset or shuffle to continue gene
```

Reset the `minibatchqueue` object. Now, `hasdata` returns `1` (`true`), and you can continue to obtain data using `next`.

```
reset(mbq);
hasdata(mbq)

ans =
   1

X = next(mbq);
```

Input Arguments

`mbq` — Queue of mini-batches

`minibatchqueue`

Queue of mini-batches, specified as a `minibatchqueue` object.

See Also

`hasdata` | `next` | `shuffle` | `minibatchqueue`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2020b

resetState

Reset the state of a recurrent neural network

Syntax

```
updatedNet = resetState(recNet)
```

Description

`updatedNet = resetState(recNet)` resets the state of a recurrent neural network (for example, an LSTM network) to the initial state.

Examples

Reset Network State

Reset the network state between sequence predictions.

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

```
ans =
  5x1 Layer array with layers:

     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load the test data.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Classify a sequence and update the network state. For reproducibility, set `rng` to `'shuffle'`.

```
rng('shuffle')
X = XTest{94};
[net,label] = classifyAndUpdateState(net,X);
label
```

```
label = categorical
     3
```

Classify another sequence using the updated network.

```
X = XTest{1};
label = classify(net,X)
```

```
label = categorical
     7
```

Compare the final prediction with the true label.

```
trueLabel = YTest(1)
```

```
trueLabel = categorical
     1
```

The updated state of the network may have negatively influenced the classification. Reset the network state and predict on the sequence again.

```
net = resetState(net);
label = classify(net,XTest{1})
```

```
label = categorical
     1
```

Input Arguments

`recNet` — Trained recurrent neural network
`SeriesNetwork` object | `DAGNetwork` object

Trained recurrent neural network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function.

`recNet` is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network). If the input network is not a recurrent network, then the function has no effect and returns the input network.

Output Arguments

`updatedNet` — Updated network
`SeriesNetwork` object | `DAGNetwork` object

Updated network. `updatedNet` is the same type of network as the input network.

If the input network is not a recurrent network, then the function has no effect and returns the input network.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation for Intel MKL-DNN target does not support the combination of `'SequenceLength','longest'`, `'SequencePaddingDirection','left'`, and `'SequencePaddingValue',0` name-value arguments.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation for the `resetState` function is only supported for recurrent neural networks and cuDNN target library.

See Also

`sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState`

Topics

“Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”

Introduced in R2017b

rmspropupdate

Update parameters using root mean squared propagation (RMSProp)

Syntax

```
[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad)
[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad)
[___] = rmspropupdate(___,learnRate,sqGradDecay,epsilon)
```

Description

Update the network learnable parameters in a custom training loop using the root mean squared propagation (RMSProp) algorithm.

Note This function applies the RMSProp optimization algorithm to update network parameters in custom training loops that use networks defined as `dlnetwork` objects or model functions. If you want to train a network defined as a `Layer` array or as a `LayerGraph`, use the following functions:

- Create a `TrainingOptionsRMSProp` object using the `trainingOptions` function.
- Use the `TrainingOptionsRMSProp` object with the `trainNetwork` function.

`[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad)` updates the learnable parameters of the network `dlnet` using the RMSProp algorithm. Use this syntax in a training loop to iteratively update a network defined as a `dlnetwork` object.

`[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad)` updates the learnable parameters in `params` using the RMSProp algorithm. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined using functions.

`[___] = rmspropupdate(___,learnRate,sqGradDecay,epsilon)` also specifies values to use for the global learning rate, square gradient decay, and small constant epsilon, in addition to the input arguments in previous syntaxes.

Examples

Update Learnable Parameters Using `rmspropupdate`

Perform a single root mean squared propagation update step with a global learning rate of `0.05` and squared gradient decay factor of `0.95`.

Create the parameters and parameter gradients as numeric arrays.

```
params = rand(3,3,4);
grad = ones(3,3,4);
```

Initialize the average squared gradient for the first iteration.

```
averageSqGrad = [];
```

Specify custom values for the global learning rate and squared gradient decay factor.

```
learnRate = 0.05;
sqGradDecay = 0.95;
```

Update the learnable parameters using `rmspropupdate`.

```
[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad,learnRate,sqGradDecay);
```

Train a Network Using `rmspropupdate`

Use `rmspropupdate` to train a network using the root mean squared propagation (RMSProp) algorithm.

Load Training Data

Load the digits training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

Define the Network

Define the network architecture and specify the average image value 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')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of the example. The function takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```
miniBatchSize = 128;
numEpochs = 20;
```

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

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

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

Visualize the training progress in a plot.

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

Train Network

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the `rmspropupdate` function. At the end of each epoch, display the training progress.

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
```

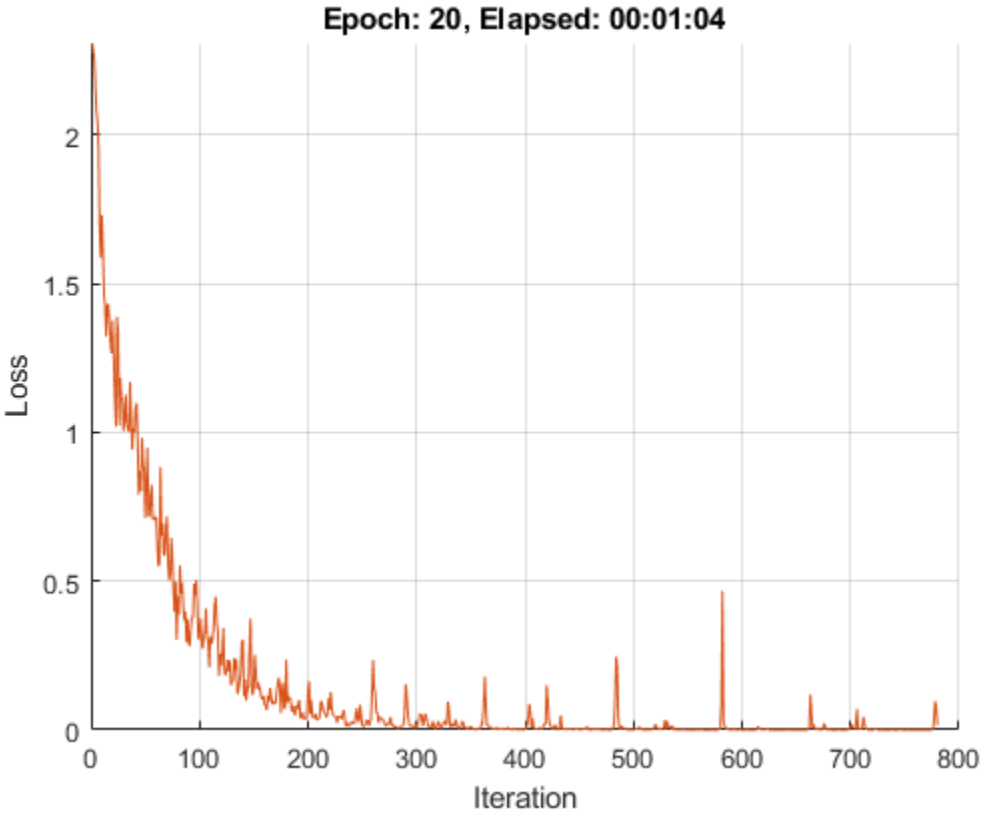
Initialize the squared average gradients.

```
averageSqGrad = [];
```

Train the network.

```
iteration = 0;
start = tic;

for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);

    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 a dlarray.
        dlX = dlarray(single(X),'SSCB');

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

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients helper function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

        % Update the network parameters using the RMSProp optimizer.
        [dlnet,averageSqGrad] = rmspropupdate(dlnet,gradients,averageSqGrad);

        % 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 the Network

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

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

To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.9860
```

Model Gradients Function

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

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

dlYPred = forward(dlnet,dlX);

loss = crossentropy(dlYPred,Y);

gradients = dlgradient(loss,dlnet.Learnables);

end
```

Input Arguments

`dlnet` — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object. `dlnet.Learnables` is a table with three variables:

- `Layer` — Layer name, specified as a string scalar.

- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

The input argument `grad` must be a table of the same form as `dlnet.Learnables`.

`params` — Network learnable parameters
`dlarray` | numeric array | cell array | structure | table

Network learnable parameters, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

If you specify `params` as a table, it must contain the following three variables.

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

You can specify `params` as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `grad` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

Data Types: `single` | `double` | `struct` | `table` | `cell`

`grad` — Gradients of loss
`dlarray` | numeric array | cell array | structure | table

Gradients of the loss, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `grad` depends on the input network or learnable parameters. The following table shows the required format for `grad` for possible inputs to `rmspropupdate`.

Input	Learnable Parameters	Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`

Input	Learnable Parameters	Gradients
	Structure	Structure with the same data types, fields, and ordering as `params`
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.

You can obtain `grad` from a call to `dlfeval` that evaluates a function that contains a call to `dlgradient`. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

`averageSqGrad` — Moving average of squared parameter gradients
`[]` | `dlarray` | numeric array | cell array | structure | table

Moving average of squared parameter gradients, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `averageSqGrad` depends on the input network or learnable parameters. The following table shows the required format for `averageSqGrad` for possible inputs to `rmspropupdate`.

Input	Learnable Parameters	Average Squared Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `averageSqGrad` must have a `Value` variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`

Input	Learnable Parameters	Average Squared Gradients
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `averageSqGrad` must have a `Value` variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.

If you specify `averageSqGrad` as an empty array, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageSqGrad` output of a previous call to `rmspropupdate` as the `averageSqGrad` input.

`learnRate` — Global learning rate
`0.001` (default) | positive scalar

Global learning rate, specified as a positive scalar. The default value of `learnRate` is `0.001`.

If you specify the network parameters as a `dlnetwork`, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

`sqGradDecay` — Squared gradient decay factor
`0.9` (default) | positive scalar between `0` and `1`.

Squared gradient decay factor, specified as a positive scalar between `0` and `1`. The default value of `sqGradDecay` is `0.9`.

`epsilon` — Small constant
`1e-8` (default) | positive scalar

Small constant for preventing divide-by-zero errors, specified as a positive scalar. The default value of `epsilon` is `1e-8`.

Output Arguments

`dlnet` — Updated network
`dlnetwork` object

Network, returned as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object.

`params` — Updated network learnable parameters
`dlarray` | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table with a `Value` variable containing the updated learnable parameters of the network.

`averageSqGrad` — Updated moving average of squared parameter gradients
`dlarray` | numeric array | cell array | structure | table

Updated moving average of squared parameter gradients, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table.

More About

RMSProp

The function uses the root mean squared propagation algorithm to update the learnable parameters. For more information, see the definition of the RMSProp algorithm under "Stochastic Gradient Descent" on page 1-1368 on the `trainingOptions` reference page.

Compatibility Considerations

`rmspropupdate` squared gradient decay factor default is `0.9`

Behavior changed in R2020a

Starting in R2020a, the default value of the squared gradient decay factor in `rmspropupdate` is `0.9`. In previous versions, the default value was `0.999`. To reproduce the previous default behavior, use one of the following syntaxes:

```
[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad,0.001,0.999)
[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad,0.001,0.999)
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.
  - `grad`
  - `averageSqGrad`
  - `params`

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`dlnetwork` | `dlarray` | `dlupdate` | `adamupdate` | `sgdmupdate` | `forward` | `dlgradient` | `dlfeval`

Topics

"Define Custom Training Loops, Loss Functions, and Networks"
"Specify Training Options in Custom Training Loop"
"Train Network Using Custom Training Loop"

Introduced in R2019b

relu

Apply rectified linear unit activation

Syntax

```
dlY = relu(dlX)
```

Description

The rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is set to zero.

This operation is equivalent to

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

Note This function applies the ReLU operation to `dlarray` data. If you want to apply ReLU activation within a `layerGraph` object or `Layer` array, use the following layer:

- `reluLayer`

`dlY = relu(dlX)` computes the ReLU activation of the input `dlX` by applying a threshold operation. All values in `dlX` that are less than zero are set to zero.

Examples

Apply ReLU Activation

Use the `relu` function to set negative values in the input data to zero.

Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```
height = 12;
width = 12;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the leaky ReLU activation.

```
dlY = relu(dlX);
```

All negative values in `dlX` are now set to `0`.

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Data Types: `single` | `double`

Output Arguments

`dlY` — ReLU activations

`dlarray`

ReLU activations, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlconv` | `batchnorm` | `leakyrelu` | `dlgradient` | `dlfeval`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

reluLayer

Rectified Linear Unit (ReLU) layer

Description

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

This operation is equivalent to

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

Creation

Syntax

```
layer = reluLayer
layer = reluLayer('Name',Name)
```

Description

`layer = reluLayer` creates a ReLU layer.

`layer = reluLayer('Name',Name)` creates a ReLU layer and sets the optional `Name` property using a name-value pair. For example, `reluLayer('Name','relu1')` creates a ReLU layer with the name `'relu1'`.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create ReLU Layer

Create a ReLU layer with the name `'relu1'`.

```
layer = reluLayer('Name','relu1')
```

```
layer =
  ReLULayer with properties:

    Name: 'relu1'
```

Include a ReLU layer in a `Layer` array.

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

```
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  0
```

```
    3   ''   ReLU                    ReLU
    4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0
    5   ''   Fully Connected         10 fully connected layer
    6   ''   Softmax                 softmax
    7   ''   Classification Output   crossentropyex
```

More About

ReLU Layer

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”.

References

[1] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807-814. 2010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `batchNormalizationLayer` | `leakyReluLayer` | `clippedReluLayer` | `swishLayer` | **Deep Network Designer**

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2016a

removeLayers

Remove layers from layer graph

Syntax

```
newlgraph = removeLayers(lgraph,layerNames)
```

Description

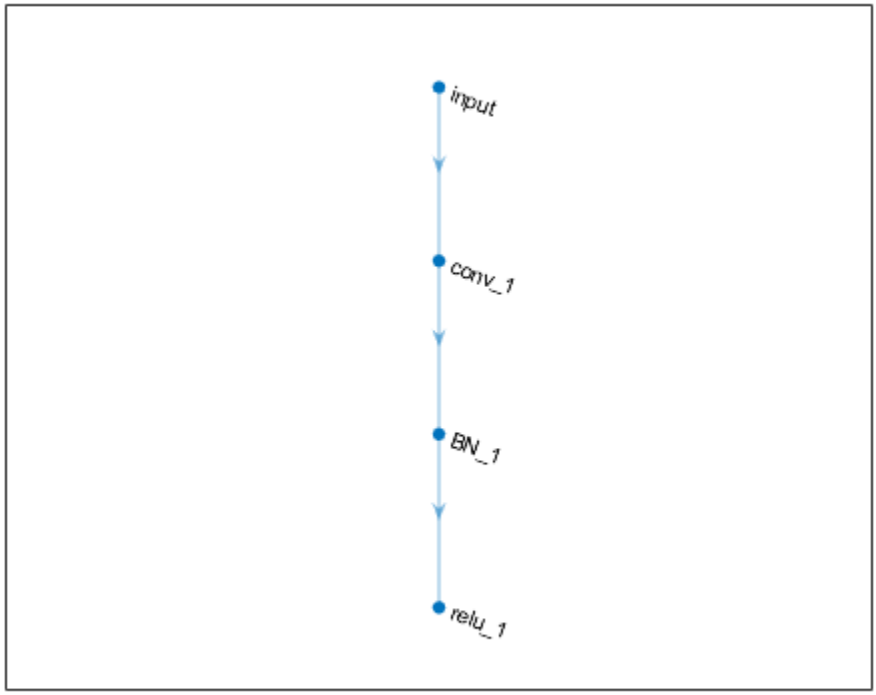
`newlgraph = removeLayers(lgraph,layerNames)` removes the layers specified by `layerNames` from the layer graph `lgraph`. The function also removes any connections to the removed layers.

Examples

Remove Layer from Layer Graph

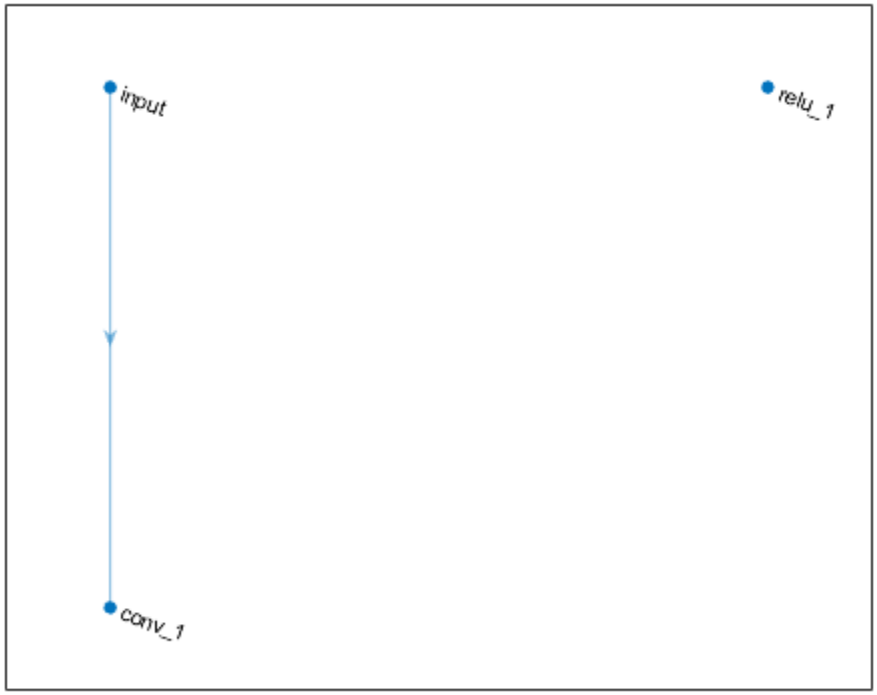
Create a layer graph from an array of layers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];

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



Remove the 'BN_1' layer and its connections.

```
lgraph = removeLayers(lgraph,'BN_1');
figure
plot(lgraph)
```



Input Arguments

`lgraph` — Layer graph

`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`layerNames` — Names of layers to remove

character vector | cell array of character vectors | string array

Names of layers to remove, specified as a character vector, a cell array of character vectors, or a string array.

To remove a single layer from the layer graph, specify the name of the layer.

To remove multiple layers, specify the layer names in an array, where each element of the array is a layer name.

Example: `'conv1'`

Example: `{'conv1','add1'}`

Output Arguments

`newlgraph` — Output layer graph
`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `addLayers` | `replaceLayer` | `connectLayers` | `disconnectLayers` | `plot` | `assembleNetwork`

Topics
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b

removeParameter

Remove parameter from `ONNXParameters` object

Syntax

```
params = removeParameter(params,name)
```

Description

`params = removeParameter(params,name)` removes the parameter specified by `name` from the `ONNXParameters` object `params`.

Examples

Remove Parameters from Imported ONNX Model Function

Import a network saved in the ONNX format as a function and modify the network parameters.

Import the pretrained `simplenet3fc.onnx` network as a function. `simplenet3fc` is a simple convolutional neural network trained on digit image data. For more information on how to create a network similar to `simplenet3fc`, see “Create Simple Image Classification Network”.

Import `simplenet3fc.onnx` using `importONNXFunction`, which returns an `ONNXParameters` object that contains the network parameters. The function also creates a new model function in the current folder that contains the network architecture. Specify the name of the model function as `simplenetFcn`.

```
params = importONNXFunction('simplenet3fc.onnx','simplenetFcn');
```

```
A function containing the imported ONNX network has been saved to the file simplenetFcn.m.
To learn how to use this function, type: help simplenetFcn.
```

Display the parameters that are updated during training (`params.Learnables`) and the parameters that remain unchanged during training (`params.Nonlearnables`).

```
params.Learnables
```

```
ans = struct with fields:
    imageinput_Mean: [1×1 dlarray]
             conv_W: [5×5×1×20 dlarray]
             conv_B: [20×1 dlarray]
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
             fc_1_W: [24×24×20×20 dlarray]
             fc_1_B: [20×1 dlarray]
             fc_2_W: [1×1×20×20 dlarray]
             fc_2_B: [20×1 dlarray]
             fc_3_W: [1×1×20×10 dlarray]
             fc_3_B: [10×1 dlarray]
```

```
params.Nonlearnables

ans = struct with fields:
             ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
             ConvStride1007: [2×1 dlarray]
    ConvDilationFactor1008: [2×1 dlarray]
           ConvPadding1009: [4×1 dlarray]
             ConvStride1010: [2×1 dlarray]
    ConvDilationFactor1011: [2×1 dlarray]
           ConvPadding1012: [4×1 dlarray]
             ConvStride1013: [2×1 dlarray]
    ConvDilationFactor1014: [2×1 dlarray]
           ConvPadding1015: [4×1 dlarray]
```

The network has parameters that represent three fully connected layers. To see the parameters of the convolutional layers `fc_1`, `fc_2`, and `fc_3`, open the model function `simplenetFcn`.

```
open simplenetFcn
```

Scroll down to the layer definitions in the function `simplenetFcn`. The code below shows the definitions for layers `fc_1`, `fc_2`, and `fc_3`.

```
% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_1] = prepareConvArgs(Vars
Vars.fc_1 = dlconv(Vars.relu1001, weights, bias, 'Stride', stride, 'DilationFactor', dilationFact

% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_2] = prepareConvArgs(Vars
Vars.fc_2 = dlconv(Vars.fc_1, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,

% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_3] = prepareConvArgs(Vars
Vars.fc_3 = dlconv(Vars.fc_2, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,
```

You can remove the parameters of the fully connected layer `fc_2` to reduce computational complexity. Check the output dimensions of the previous layer and the input dimensions of the subsequent layer before removing a middle layer from `params`. In this case, the output size of the previous layer `fc_1` is 20, and the input size of the subsequent layer `fc_3` is also 20.

Remove the parameters of layer `fc_2` by using `removeParameter`.

```
params = removeParameter(params,'fc_2_B');
params = removeParameter(params,'fc_2_W');
params = removeParameter(params,'ConvStride1010');
params = removeParameter(params,'ConvDilationFactor1011');
params = removeParameter(params,'ConvPadding1012');
```

Display the updated learnable and nonlearnable parameters.

```
params.Learnables

ans = struct with fields:
    imageinput_Mean: [1×1 dlarray]
             conv_W: [5×5×1×20 dlarray]
             conv_B: [20×1 dlarray]
```

```
    batchnorm_scale: [20×1 dlarray]
        batchnorm_B: [20×1 dlarray]
             fc_1_W: [24×24×20×20 dlarray]
             fc_1_B: [20×1 dlarray]
             fc_3_W: [1×1×20×10 dlarray]
             fc_3_B: [10×1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
            ConvStride1004: [2×1 dlarray]
    ConvDilationFactor1005: [2×1 dlarray]
           ConvPadding1006: [4×1 dlarray]
            ConvStride1007: [2×1 dlarray]
    ConvDilationFactor1008: [2×1 dlarray]
           ConvPadding1009: [4×1 dlarray]
            ConvStride1013: [2×1 dlarray]
    ConvDilationFactor1014: [2×1 dlarray]
           ConvPadding1015: [4×1 dlarray]
```

Modify the architecture of the model function to reflect the changes in `params` so you can use the network for prediction with the new parameters or retrain the network. Open the model function `simplenetFcn`. Then, remove the fully connected layer `fc_2`, and change the input data of the convolution operation `dlconv` for layer `fc_3` to `Vars.fc_1`.

```
open simplenetFcn
```

The code below shows layers `fc_1` and `fc_3`.

```
% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_1] = prepareConvArgs(Vars
Vars.fc_1 = dlconv(Vars.relu1001, weights, bias, 'Stride', stride, 'DilationFactor', dilationFact
```

```
% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_3] = prepareConvArgs(Vars
Vars.fc_3 = dlconv(Vars.fc_1, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor,
```

Input Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, specified as an `ONNXParameters` object. `params` contains the network parameters of the imported ONNX model.

`name` — Name of parameter

character vector | string scalar

Name of the parameter, specified as a character vector or string scalar.

Example: `'conv2_W'`

Example: `'conv2_Padding'`

Output Arguments

params — Network parameters
`ONNXParameters` object

Network parameters, returned as an `ONNXParameters` object. `params` contains the network parameters updated by `removeParameter`.

See Also

`importONNXFunction` | `ONNXParameters` | `addParameter`

Introduced in R2020b

replaceLayer

Replace layer in layer graph

Syntax

```
newlgraph = replaceLayer(lgraph,layerName,larray)
newlgraph = replaceLayer(lgraph,layerName,larray,'ReconnectBy',mode)
```

Description

`newlgraph = replaceLayer(lgraph,layerName,larray)` replaces the layer `layerName` in the layer graph `lgraph` with the layers in `larray`.

`replaceLayer` connects the layers in `larray` sequentially and connects `larray` into the layer graph.

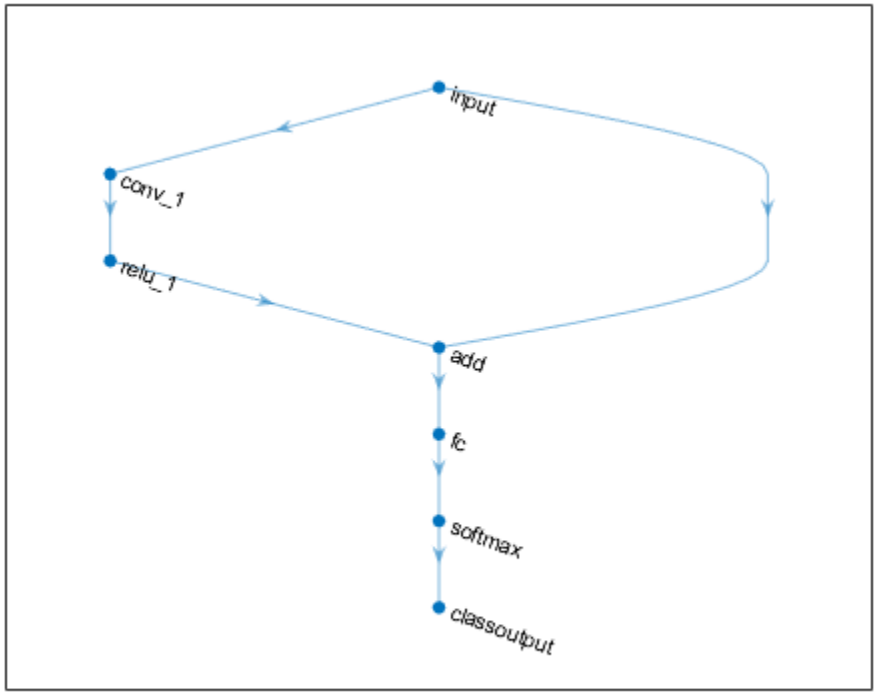
`newlgraph = replaceLayer(lgraph,layerName,larray,'ReconnectBy',mode)` additionally specifies the method of reconnecting layers.

Examples

Replace Layer in Layer Graph

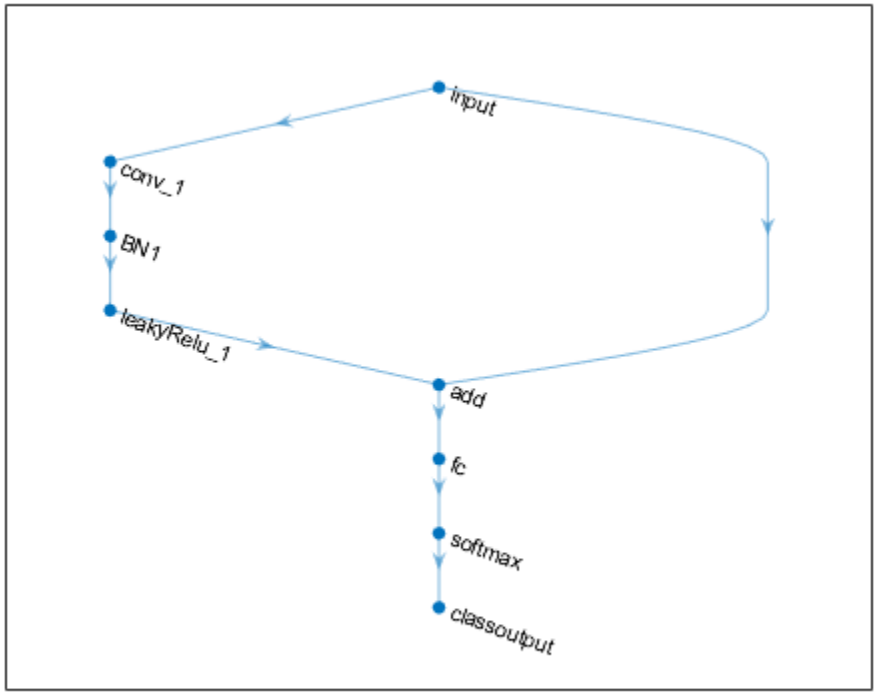
Define a simple network architecture and plot it.

```
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    reluLayer('Name','relu_1')
    additionLayer(2,'Name','add')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classoutput')];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'input','add/in2');

figure
plot(lgraph)
```



Replace the ReLU layer in the network with a batch normalization layer followed by a leaky ReLU layer.

```
larray = [batchNormalizationLayer('Name','BN1')
          leakyReluLayer('Name','leakyRelu_1','Scale',0.1)];
lgraph = replaceLayer(lgraph,'relu_1',larray);

plot(lgraph)
```



Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

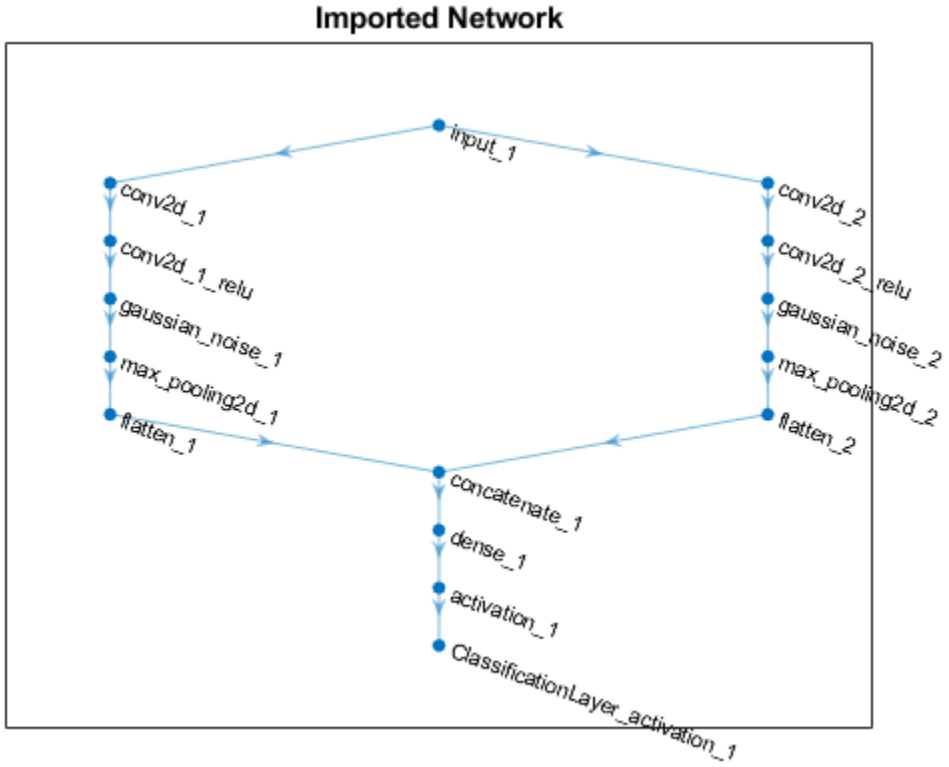
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

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



Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```
placeholderLayers.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

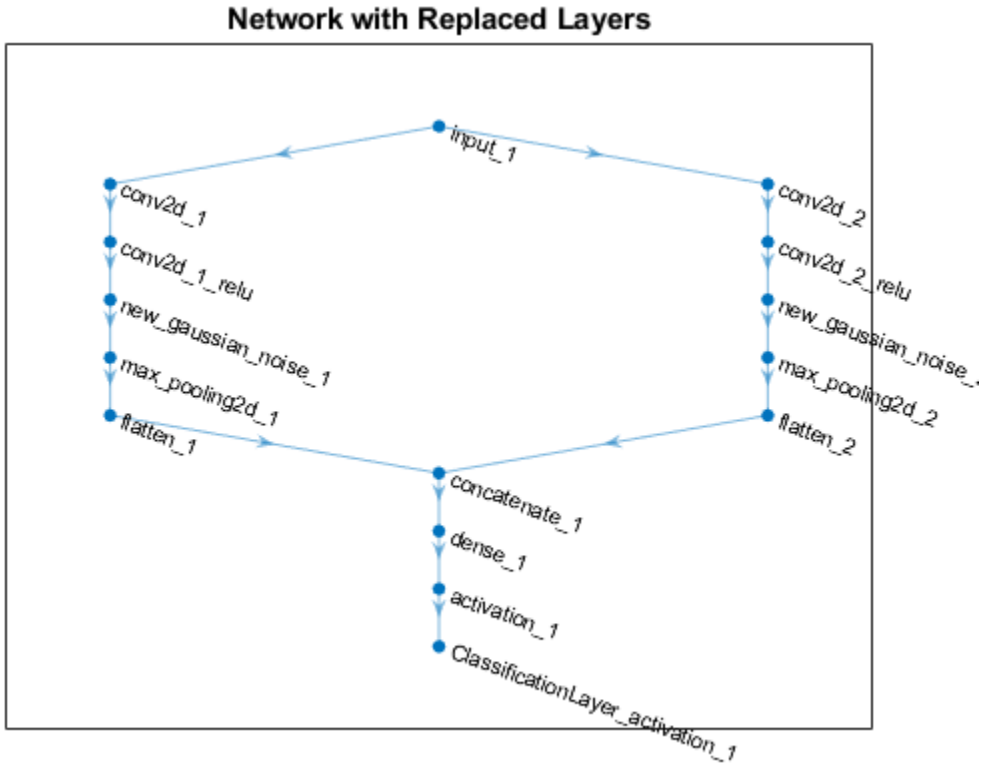
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```
figure
plot(lgraph)
title("Network with Replaced Layers")
```



Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                           Image Input             28x28x1 images
     2   'conv2d_1'                          Convolution             20 7x7x1 convolutions with
     3   'conv2d_1_relu'                     ReLU                    ReLU
     4   'conv2d_2'                          Convolution             20 3x3x1 convolutions with
     5   'conv2d_2_relu'                     ReLU                    ReLU
     6   'new_gaussian_noise_1'              Gaussian Noise          Gaussian noise with standar
     7   'new_gaussian_noise_2'              Gaussian Noise          Gaussian noise with standar
     8   'max_pooling2d_1'                   Max Pooling             2x2 max pooling with stride
     9   'max_pooling2d_2'                   Max Pooling             2x2 max pooling with stride
    10   'flatten_1'                         Keras Flatten           Flatten activations into 1-
    11   'flatten_2'                         Keras Flatten           Flatten activations into 1-
    12   'concatenate_1'                     Depth concatenation     Depth concatenation of 2 in
    13   'dense_1'                           Fully Connected         10 fully connected layer
    14   'activation_1'                      Softmax                 softmax
    15   'ClassificationLayer_activation_1'  Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_activation_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `'auto'`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: [0    1    2    3    4    5    6    7    8    9]
    ClassWeights: 'none'
      OutputSize: 10

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)

net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Input Arguments

`lgraph` — Layer graph

`LayerGraph` object

Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

`layerName` — Name of layer to replace

string scalar | character vector

Name of the layer to replace, specified as a string scalar or a character vector.

`larray` — Network layers

`Layer` array

Network layers, specified as a `Layer` array.

For a list of built-in layers, see “List of Deep Learning Layers”.

`mode` — Method to reconnect layers

`'name'` (default) | `'order'`

Method to reconnect layers specified as one of the following:

- `'name'` – Reconnect `larray` using the input and output names of the replaced layer. For each layer connected to an input of the replaced layer, reconnect the layer to the input of the same input name of `larray(1)`. For each layer connected to an output of the replaced layer, reconnect the layer to the output of the same output name of `larray(end)`.
- `'order'` – Reconnect `larray` using the order of the input names of `larray(1)` and the output names of `larray(end)`. Reconnect the layer connected to the `i`th input of the replaced layer to the `i`th input of `larray(1)`. Reconnect the layer connected to the `j`th output of the replaced layer to the `j`th output of `larray(end)`.

Data Types: `char` | `string`

Output Arguments

`newlgraph` — Output layer graph

`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`layerGraph` | `findPlaceholderLayers` | `PlaceholderLayer` | `connectLayers` | `disconnectLayers` | `addLayers` | `removeLayers` | `assembleNetwork` | `functionLayer`

Topics

"Deep Learning in MATLAB"
"Pretrained Deep Neural Networks"
"Train Residual Network for Image Classification"
"Train Deep Learning Network to Classify New Images"

Introduced in R2018b

resnet18

ResNet-18 convolutional neural network

Syntax

```
net = resnet18
net = resnet18('Weights','imagenet')

lgraph = resnet18('Weights','none')
```

Description

ResNet-18 is a convolutional neural network that is 18 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the ResNet-18 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-18.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-18 instead of GoogLeNet.

Tip To create an untrained residual network suitable for image classification tasks, use `resnetLayers`.

`net = resnet18` returns a ResNet-18 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for ResNet-18 Network* support package. If this support package is not installed, then the function provides a download link.

`net = resnet18('Weights','imagenet')` returns a ResNet-18 network trained on the ImageNet data set. This syntax is equivalent to `net = resnet18`.

`lgraph = resnet18('Weights','none')` returns the untrained ResNet-18 network architecture. The untrained model does not require the support package.

Examples

Download ResNet-18 Support Package

Download and install the Deep Learning Toolbox Model *for ResNet-18 Network* support package.

Type `resnet18` at the command line.

```
resnet18
```

If the Deep Learning Toolbox Model *for ResNet-18 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet18` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

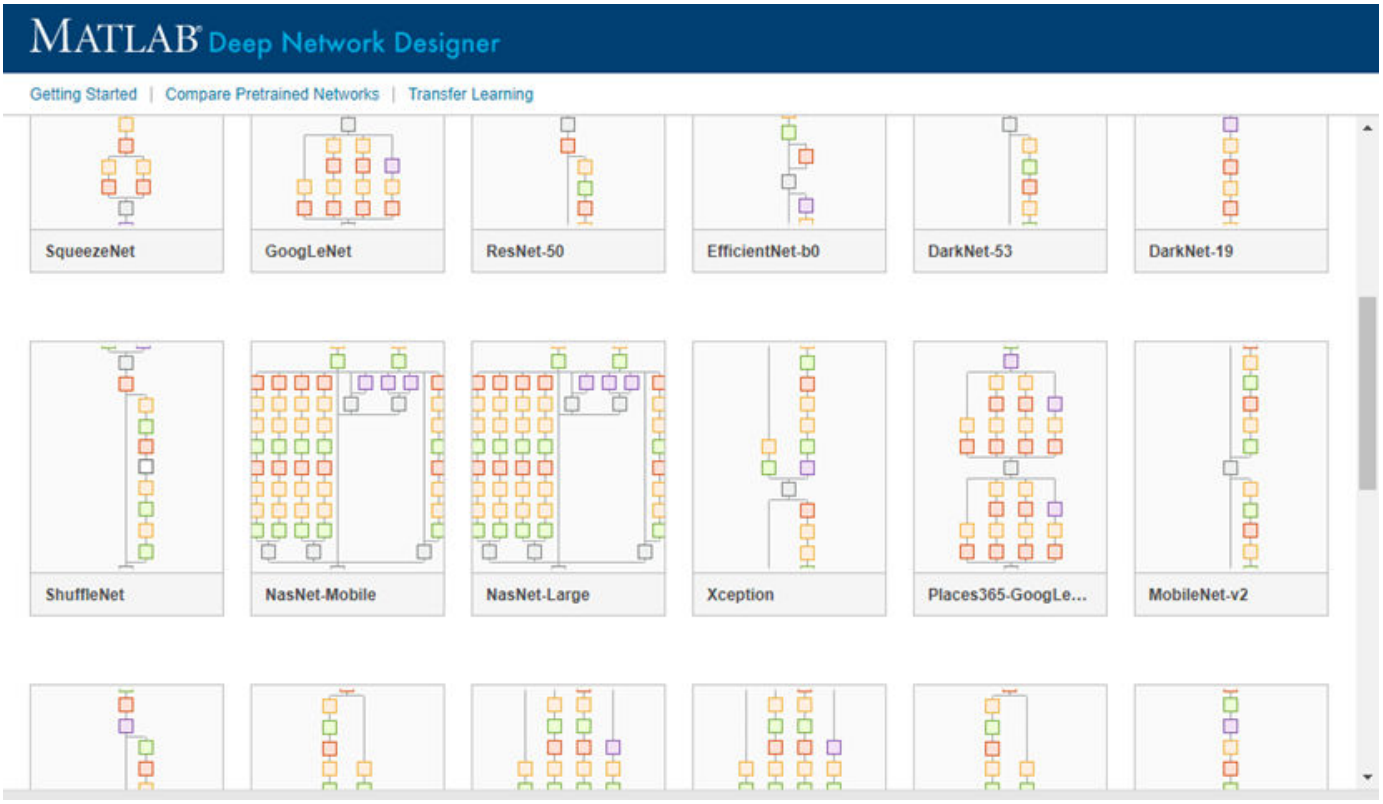
```
resnet18
```

```
ans =

  DAGNetwork with properties:

         Layers: [72×1 nnet.cnn.layer.Layer]
    Connections: [79×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(resnet18)
```

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



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

Output Arguments

net — Pretrained ResNet-18 convolutional neural network
`DAGNetwork` object

Pretrained ResNet-18 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained ResNet-18 convolutional neural network architecture
`LayerGraph` object

Untrained ResNet-18 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = resnet18` or by passing the `resnet18` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet18')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `resnet18('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = resnet18` or by passing the `resnet18` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet18')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `resnet18('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `resnetLayers` | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201`

Topics
“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a

resnet50

ResNet-50 convolutional neural network

Syntax

```
net = resnet50
net = resnet50('Weights','imagenet')

lgraph = resnet50('Weights','none')
```

Description

ResNet-50 is a convolutional neural network that is 50 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the ResNet-50 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-50.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-50 instead of GoogLeNet.

Tip To create an untrained residual network suitable for image classification tasks, use `resnetLayers`.

`net = resnet50` returns a ResNet-50 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for ResNet-50 Network* support package. If this support package is not installed, then the function provides a download link.

`net = resnet50('Weights','imagenet')` returns a ResNet-50 network trained on the ImageNet data set. This syntax is equivalent to `net = resnet50`.

`lgraph = resnet50('Weights','none')` returns the untrained ResNet-50 network architecture. The untrained model does not require the support package.

Examples

Download ResNet-50 Support Package

Download and install the Deep Learning Toolbox Model *for ResNet-50 Network* support package.

Type `resnet50` at the command line.

```
resnet50
```

If the Deep Learning Toolbox Model *for ResNet-50 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet50` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

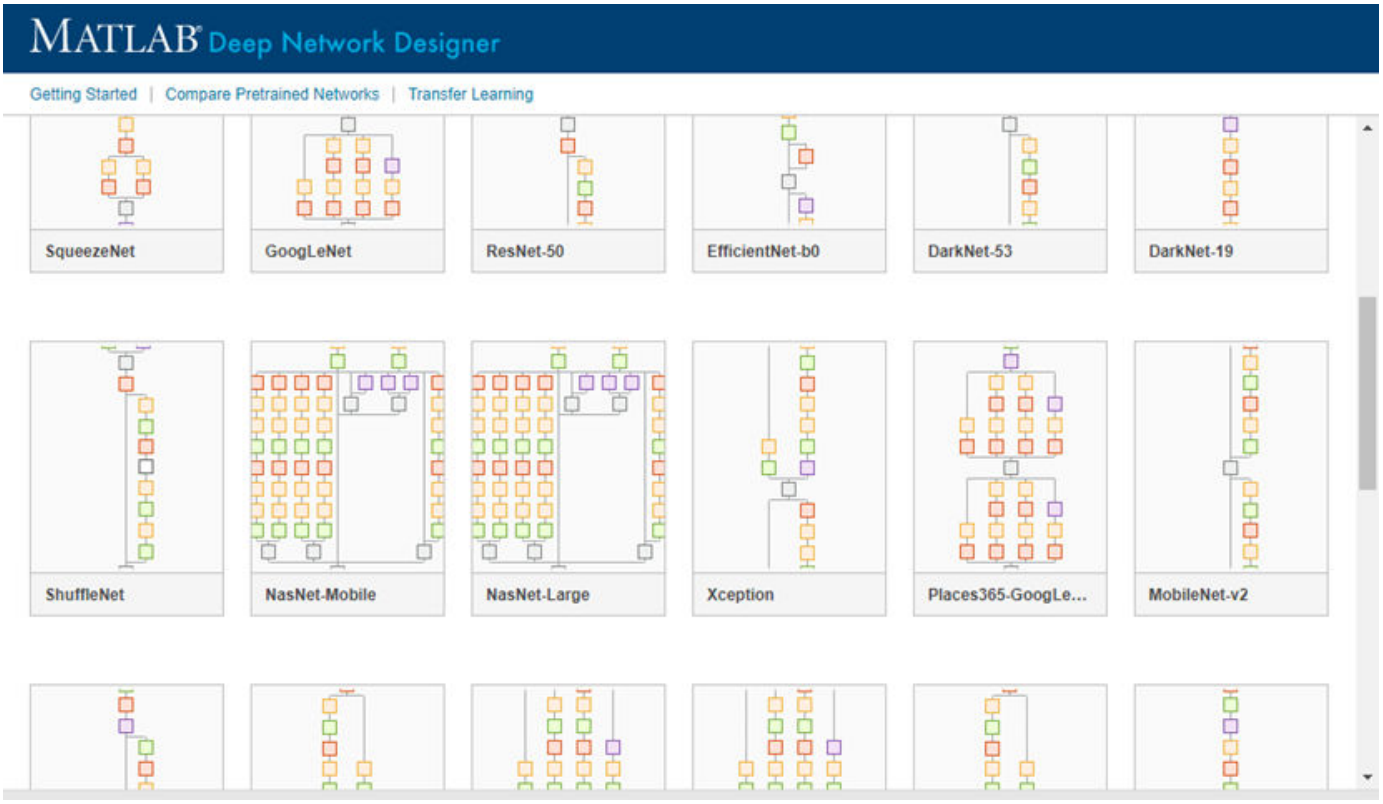
```
resnet50
```

```
ans =

  DAGNetwork with properties:

         Layers: [177×1 nnet.cnn.layer.Layer]
    Connections: [192×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(resnet50)
```

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



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

Output Arguments

net — Pretrained ResNet-50 convolutional neural network
`DAGNetwork` object

Pretrained ResNet-50 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained ResNet-50 convolutional neural network architecture
`LayerGraph` object

Untrained ResNet-50 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[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.

[3] https://keras.io/api/applications/resnet/#resnet50-function

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = resnet50` or by passing the `resnet50` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet50')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `resnet50('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = resnet50` or by passing the `resnet50` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet50')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `resnet50('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `resnetLayers` | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet18` | `resnet101` | `densenet201` | `inceptionresnetv2` | `squeezenet`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”

“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b

resnet101

ResNet-101 convolutional neural network

Syntax

```
net = resnet101
net = resnet101('Weights','imagenet')

lgraph = resnet101('Weights','none')
```

Description

ResNet-101 is a convolutional neural network that is 101 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the ResNet-101 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-101.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-101 instead of GoogLeNet.

Tip To create an untrained residual network suitable for image classification tasks, use `resnetLayers`.

`net = resnet101` returns a ResNet-101 network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for ResNet-101 Network* support package. If this support package is not installed, then the function provides a download link.

`net = resnet101('Weights','imagenet')` returns a ResNet-101 network trained on the ImageNet data set. This syntax is equivalent to `net = resnet101`.

`lgraph = resnet101('Weights','none')` returns the untrained ResNet-101 network architecture. The untrained model does not require the support package.

Examples

Download ResNet-101 Support Package

Download and install the Deep Learning Toolbox Model *for ResNet-101 Network* support package.

Type `resnet101` at the command line.

```
resnet101
```

If the Deep Learning Toolbox Model *for ResNet-101 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet101` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

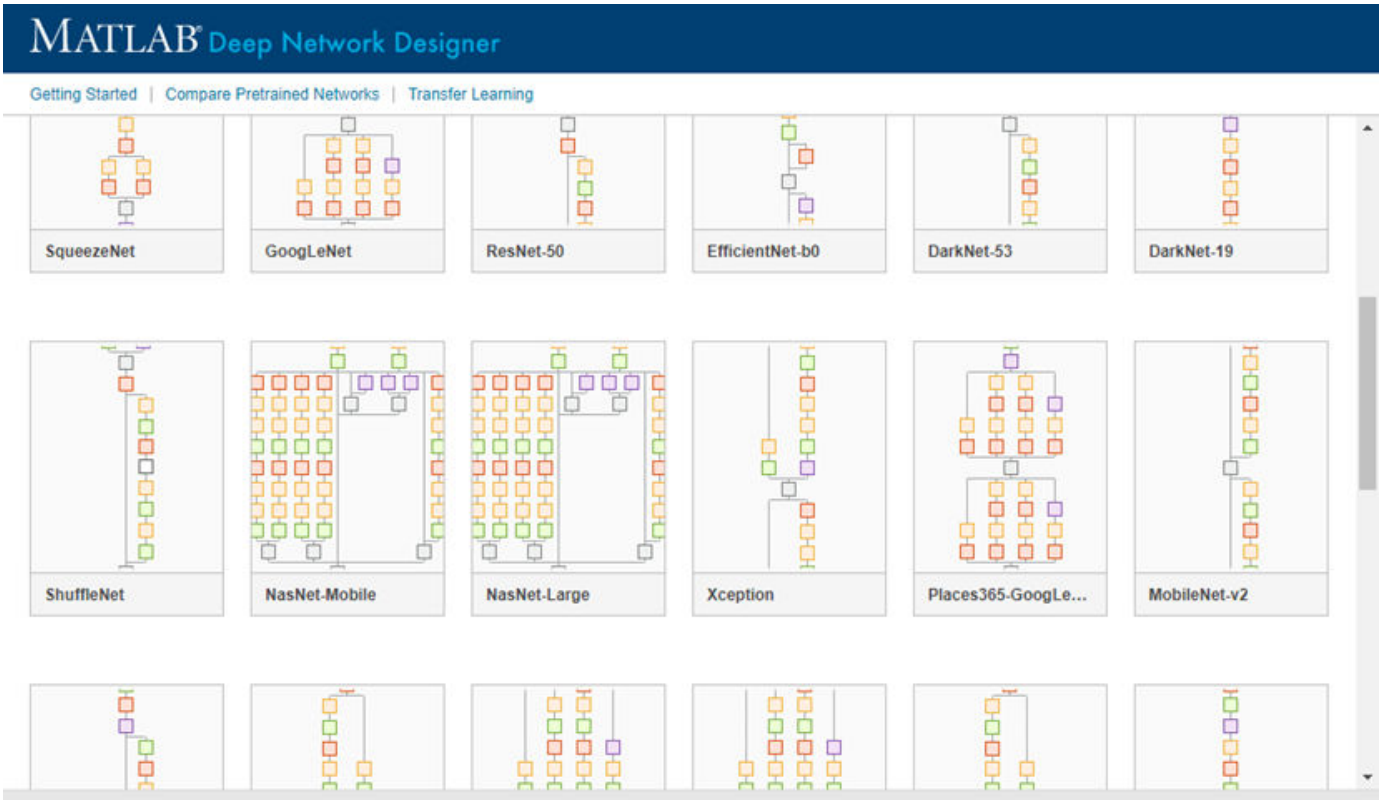
```
resnet101
```

```
ans =

  DAGNetwork with properties:

         Layers: [347×1 nnet.cnn.layer.Layer]
    Connections: [379×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(resnet101)
```

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



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

Output Arguments

net — Pretrained ResNet-101 convolutional neural network
`DAGNetwork` object

Pretrained ResNet-101 convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained ResNet-101 convolutional neural network architecture
`LayerGraph` object

Untrained ResNet-101 convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[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.

[3] https://github.com/KaimingHe/deep-residual-networks

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = resnet101` or by passing the `resnet101` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet101')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `resnet101('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = resnet101` or by passing the `resnet101` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet101')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `resnet101('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `resnetLayers` | `vgg16` | `vgg19` | `resnet18` | `resnet50` | `googlenet` | `inceptionv3` | `inceptionresnetv2` | `densenet201` | `squeezenet` | `trainNetwork` | `layerGraph` | `DAGNetwork`

Topics
“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”

“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b

resnetLayers

Create 2-D residual network

Syntax

```
lgraph = resnetLayers(inputSize,numClasses)
lgraph = resnetLayers( ___ ,Name=Value)
```

Description

`lgraph = resnetLayers(inputSize,numClasses)` creates a 2-D residual network with an image input size specified by `inputSize` and a number of classes specified by `numClasses`. A residual network consists of stacks of blocks. Each block contains deep learning layers. The network includes an image classification layer, suitable for predicting the categorical label of an input image.

To create a 3-D residual network, use `resnet3dLayers`.

`lgraph = resnetLayers( ___ ,Name=Value)` creates a residual network using one or more name-value arguments using any of the input arguments in the previous syntax. For example, `InitialNumFilters=32` specifies 32 filters in the initial convolutional layer.

Examples

Residual Network with Bottleneck

Create a residual network with a bottleneck architecture.

```
imageSize = [224 224 3];
numClasses = 10;

lgraph = resnetLayers(imageSize,numClasses)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [177x1 nnet.cnn.layer.Layer]
    Connections: [192x2 table]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Analyze the network.

```
analyzeNetwork(lgraph)
```

This network is equivalent to a ResNet-50 residual network.

Residual Network with Custom Stack Depth

Create a ResNet-101 network using a custom stack depth.

```
imageSize = [224 224 3];
numClasses = 10;

stackDepth = [3 4 23 3];
numFilters = [64 128 256 512];

lgraph = resnetLayers(imageSize,numClasses, ...
    StackDepth=stackDepth, ...
    NumFilters=numFilters)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [347x1 nnet.cnn.layer.Layer]
    Connections: [379x2 table]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Analyze the network.

```
analyzeNetwork(lgraph)
```

Train Residual Network

Create and train a residual network to classify images.

Load the digits data as in-memory numeric arrays using the `digitTrain4DArrayData` and `digitTest4DArrayData` functions.

```
[XTrain,YTrain] = digitTrain4DArrayData;
[XTest,YTest] = digitTest4DArrayData;
```

Define the residual network. The digits data contains 28-by-28 pixel images, therefore, construct a residual network with smaller filters.

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

lgraph = resnetLayers(imageSize,numClasses, ...
    InitialStride=1, ...
    InitialFilterSize=3, ...
    InitialNumFilters=16, ...
    StackDepth=[4 3 2], ...
    NumFilters=[16 32 64]);
```

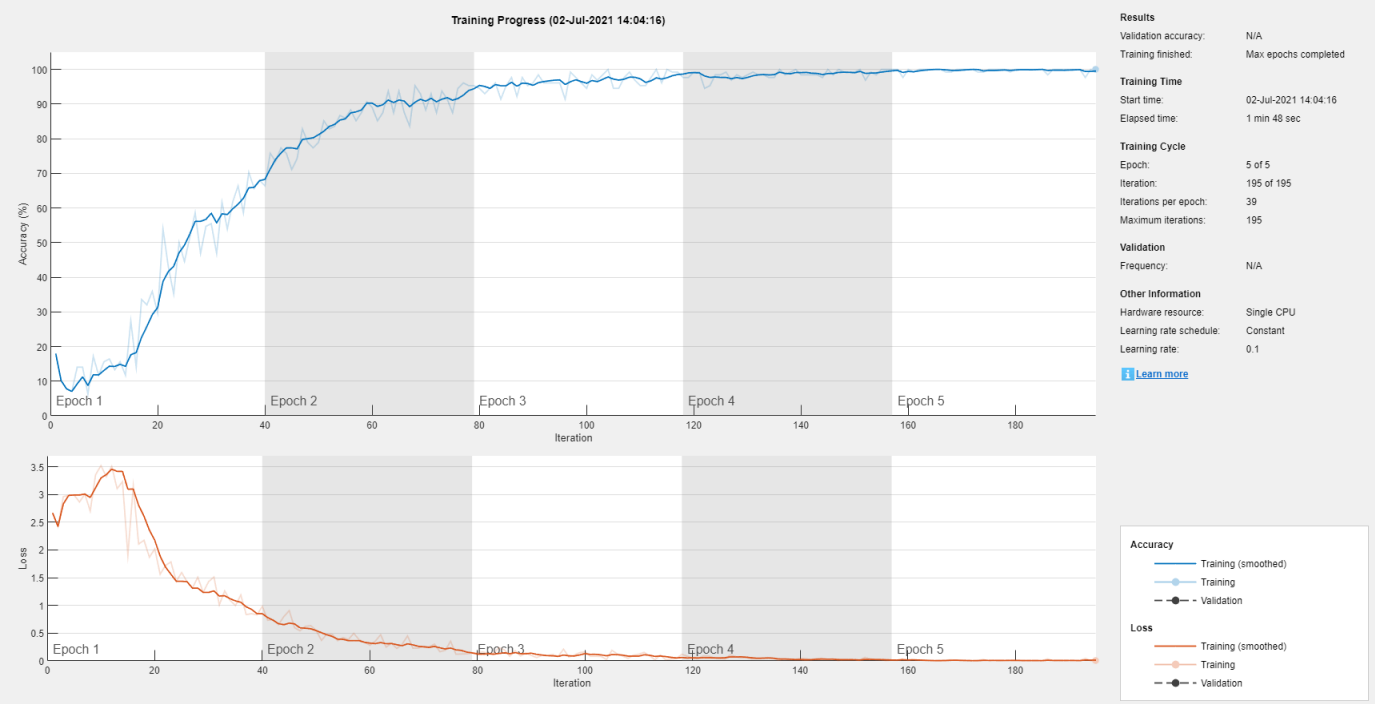
Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 5, and start the training with an initial learning rate of 0.1.

```
options = trainingOptions("sgdm", ...
    MaxEpochs=5, ...
    InitialLearnRate=0.1, ...
```

```
    Verbose=false, ...
    Plots="training-progress");
```

Train the network.

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



Test the performance of the network by evaluating the prediction accuracy of the test data. Use the `classify` function to predict the class label of each test image.

```
YPred = classify(net,XTest);
```

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

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9956
```

Convert Residual Network to `dlnetwork` Object

To train a residual network using a custom training loop, first convert it to a `dlnetwork` object.

Create a residual network.

```
lgraph = resnetLayers([224 224 3],5);
```

Remove the classification layer.

```
lgraph = removeLayers(lgraph,"output");
```

Replace the input layer with a new input layer that has `Normalization` set to `"none"`. To use an input layer with zero-center or z-score normalization, you must specify an `imageInputLayer` with nonempty value for the `Mean` property. For example, `Mean=sum(XTrain,4)`, where `XTrain` is a 4-D array containing your input data.

```
newInputLayer = imageInputLayer([224 224 3],Normalization="none");
lgraph = replaceLayer(lgraph,"input",newInputLayer);
```

Convert to a `dlnetwork`.

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

```
dlnet =
  dlnetwork with properties:

         Layers: [176x1 nnet.cnn.layer.Layer]
    Connections: [191x2 table]
     Learnables: [214x3 table]
          State: [106x3 table]
     InputNames: {'imageinput'}
    OutputNames: {'softmax'}
    Initialized: 1
```

Input Arguments

`inputSize` — Network input image size

2-element vector | 3-element vector

Network input image size, specified as one of the following:

- 2-element vector in the form [*height*, *width*].
- 3-element vector in the form [*height*, *width*, *depth*], where *depth* is the number of channels. Set *depth* to `3` for RGB images and to `1` for grayscale images. For multispectral and hyperspectral images, set *depth* to the number of channels.

The *height* and *width* values must be greater than or equal to *initialStride* * *poolingStride* * 2^D, where *D* is the number of downsampling blocks. Set the initial stride using the `InitialStride` argument. The pooling stride is `1` when the `InitialPoolingLayer` is set to `"none"`, and `2` otherwise.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`numClasses` — Number of classes

integer greater than 1

Number of classes in the image classification network, specified as an integer greater than 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `InitialFilterSize=[5,5],InitialNumFilters=32,BottleneckType="none"` specifies an initial filter size of 5-by-5 pixels, 32 initial filters, and a network architecture without bottleneck components.

Initial Layers

`InitialFilterSize` — Filter size in first convolutional layer

7 (default) | positive integer | 2-element vector of positive integers

Filter size in the first convolutional layer, specified as one of the following:

- Positive integer. The filter has equal height and width. For example, specifying `5` yields a filter of height 5 and width 5.
- 2-element vector in the form [*height*, *width*]. For example, specifying an initial filter size of `[1 5]` yields a filter of height 1 and width 5.

Example: `InitialFilterSize=[5,5]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialNumFilters` — Number of filters in first convolutional layer

64 (default) | positive integer

Number of filters in the first convolutional layer, specified as a positive integer. The number of initial filters determines the number of channels (feature maps) in the output of the first convolutional layer in the residual network.

Example: `InitialNumFilters=32`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialStride` — Stride in first convolutional layer

2 (default) | positive integer | 2-element vector of positive integers

Stride in the first convolutional layer, specified as a:

- Positive integer. The stride has equal height and width. For example, specifying `3` yields a stride of height 3 and width 3.
- 2-element vector in the form [*height*, *width*]. For example, specifying an initial stride of `[1 2]` yields a stride of height 1 and width 2.

The stride defines the step size for traversing the input vertically and horizontally.

Example: `InitialStride=[3,3]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialPoolingLayer` — First pooling layer

`"max"` (default) | `"average"` | `"none"`

First pooling layer before the initial residual block, specified as one of the following:

- `"max"` — Use a max pooling layer before the initial residual block. For more information, see `maxPooling2dLayer`.
- `"average"` — Use an average pooling layer before the initial residual block. For more information, see `averagePooling2dLayer`.

- `"none"`— Do not use a pooling layer before the initial residual block.

Example: `InitialPoolingLayer="average"`

Data Types: `char` | `string`

Network Architecture

`ResidualBlockType` — Residual block type
`"batchnorm-before-add"` (default) | `"batchnorm-after-add"`

Residual block type, specified as one of the following:

- `"batchnorm-before-add"` — Add the batch normalization layer before the addition layer in the residual blocks [1].
- `"batchnorm-after-add"` — Add the batch normalization layer after the addition layer in the residual blocks [2].

The `ResidualBlockType` argument specifies the location of the batch normalization layer in the standard and downsampling residual blocks. For more information, see "More About" on page 1-1218.

Example: `ResidualBlockType="batchnorm-after-add"`

Data Types: `char` | `string`

`BottleneckType` — Block bottleneck type
`"downsample-first-conv"` (default) | `"none"`

Block bottleneck type, specified as one of the following:

- `"downsample-first-conv"` — Use bottleneck residual blocks that perform downsampling in the first convolutional layer of the downsampling residual blocks, using a stride of 2. A bottleneck residual block consists of three convolutional layers: a 1-by-1 layer for downsampling the channel dimension, a 3-by-3 convolutional layer, and a 1-by-1 layer for upsampling the channel dimension.

  The number of filters in the final convolutional layer is four times that in the first two convolutional layers. For more information, see "`NumFilters`" on page 1-0 .
- `"none"` — Do not use bottleneck residual blocks. The residual blocks consist of two 3-by-3 convolutional layers.

A bottleneck block performs a 1-by-1 convolution before the 3-by-3 convolution to reduce the number of channels by a factor of four. Networks with and without bottleneck blocks will have a similar level of computational complexity, but the total number of features propagating in the residual connections is four times larger when you use bottleneck units. Therefore, using a bottleneck increases the efficiency of the network [1]. For more information on the layers in each residual block, see "More About" on page 1-1218.

Example: `BottleneckType="none"`

Data Types: `char` | `string`

`StackDepth` — Number of residual blocks in each stack
`[3 4 6 3]` (default) | vector of positive integers

Number of residual blocks in each stack, specified as a vector of positive integers. For example, if the stack depth is `[3 4 6 3]`, the network has four stacks, with three blocks, four blocks, six blocks, and three blocks.

Specify the number of filters in the convolutional layers of each stack using the `NumFilters` argument. The `StackDepth` value must have the same number of elements as the `NumFilters` value.

Example: `StackDepth=[9 12 69 9]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumFilters` — Number of filters in convolutional layers of each stack
`[64 128 256 512]` (default) | vector of positive integers

Number of filters in the convolutional layers of each stack, specified as a vector of positive integers.

- When you set `BottleneckType` to `"downsample-first-conv"`, the first two convolutional layers in each block of each stack have the same number of filters, set by the `NumFilters` value. The final convolutional layer has four times the number of filters in the first two convolutional layers.

  For example, suppose you set `NumFilters` to `[4 5]` and `BottleneckType` to `"downsample-first-conv"`. In the first stack, the first two convolutional layers in each block have 4 filters and the final convolutional layer in each block has 16 filters. In the second stack, the first two convolutional layers in each block have 5 filters and the final convolutional layer has 20 filters.
- When you set `BottleneckType` to `"none"`, the convolutional layers in each stack have the same number of filters, set by the `NumFilters` value.

The `NumFilters` value must have the same number of elements as the `StackDepth` value.

The `NumFilters` value determines the layers on the residual connection in the initial residual block. There is a convolutional layer on the residual connection if one of the following conditions is met:

- `BottleneckType="downsample-first-conv"`(default) and `InitialNumFilters` is not equal to four times the first element of `NumFilters`.
- `BottleneckType="none"` and `InitialNumFilters` is not equal to the first element of `NumFilters`.

For more information about the layers in each residual block, see “More About” on page 1-1218.

Example: `NumFilters=[32 64 126 256]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Normalization` — Data normalization
`"zerocenter"` (default) | `"zscore"`

Data normalization to apply every time data is forward-propagated through the input layer, specified as one of the following:

- `"zerocenter"` — Subtract the mean. The mean is calculated at training time.
- `"zscore"` — Subtract the mean and divide by the standard deviation. The mean and standard deviation are calculated at training time.

Example: `Normalization="zscore"`

Data Types: `char` | `string`

Output Arguments

`lgraph` — Residual network

`layerGraph` object

Residual network, returned as a `layerGraph` object.

More About

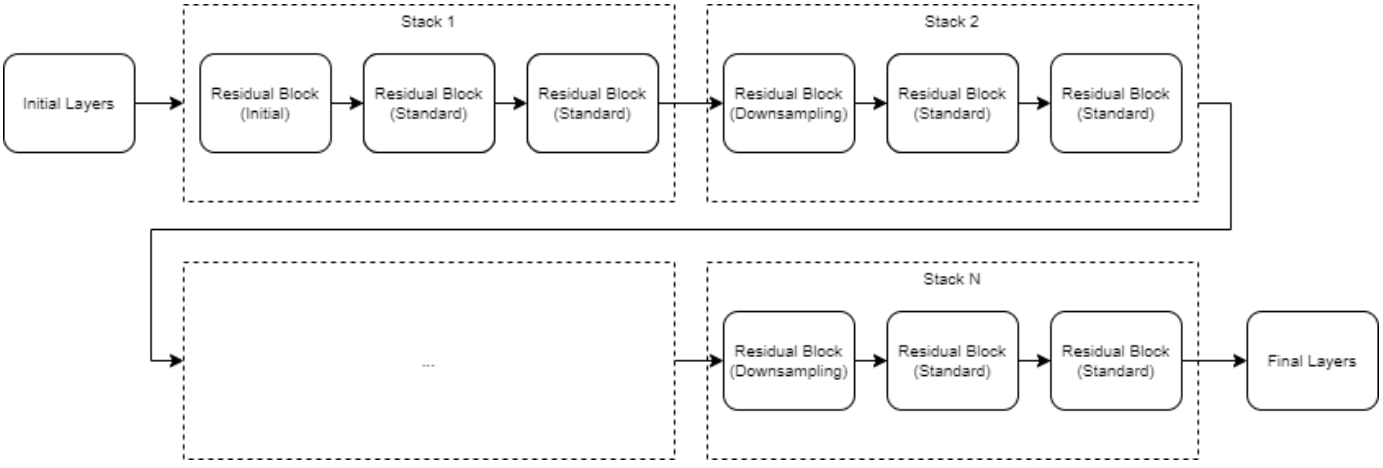
Residual Network

Residual networks (ResNets) are a type of deep network consisting of building blocks that have *residual connections* (also known as *skip* or *shortcut* connections). These connections allow the input to skip the convolutional units of the main branch, thus providing a simpler path through the network. By allowing the parameter gradients to flow more easily from the output layer to the earlier layers of the network, residual connections help mitigate the problem of vanishing gradients during early training.

The structure of a residual network is flexible. The key component is the inclusion of the residual connections within *residual blocks*. A group of residual blocks is called a *stack*. A ResNet architecture consists of initial layers, followed by stacks containing residual blocks, and then the final layers. A network has three types of residual blocks:

- Initial residual block — This block occurs at the start of the first stack. The layers in the residual connection of the initial residual block determine if the block preserves the activation sizes or performs downsampling.
- Standard residual block — This block occurs multiple times in each stack, after the first downsampling residual block. The standard residual block preserves the activation sizes.
- Downsampling residual block — This block occurs once, at the start of each stack. The first convolutional unit in the downsampling block downsamples the spatial dimensions by a factor of two.

A typical stack has a downsampling residual block, followed by `m` standard residual blocks, where `m` is greater than or equal to one. The first stack is the only stack that begins with an initial residual block.



The initial, standard, and downsampling residual blocks can be *bottleneck* or nonbottleneck blocks. Bottleneck blocks perform a 1-by-1 convolution before the 3-by-3 convolution, to reduce the number of channels by a factor of four. Networks with and without bottleneck blocks have a similar level of computational complexity, but the total number of features propagating in the residual connections is four times larger when you use the bottleneck units. Therefore, using bottleneck blocks increases the efficiency of the network.

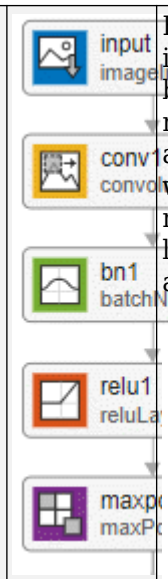
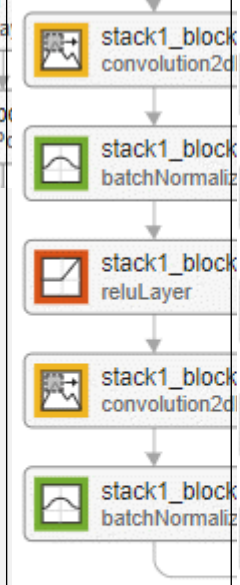
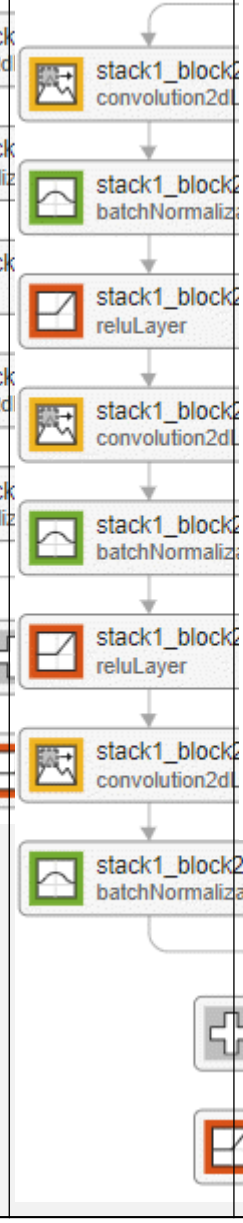
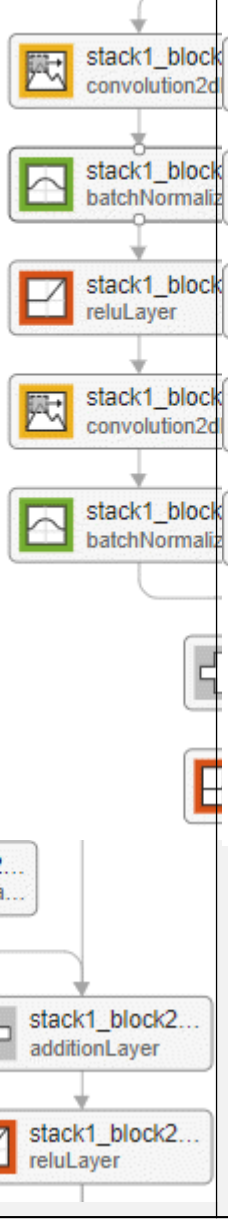
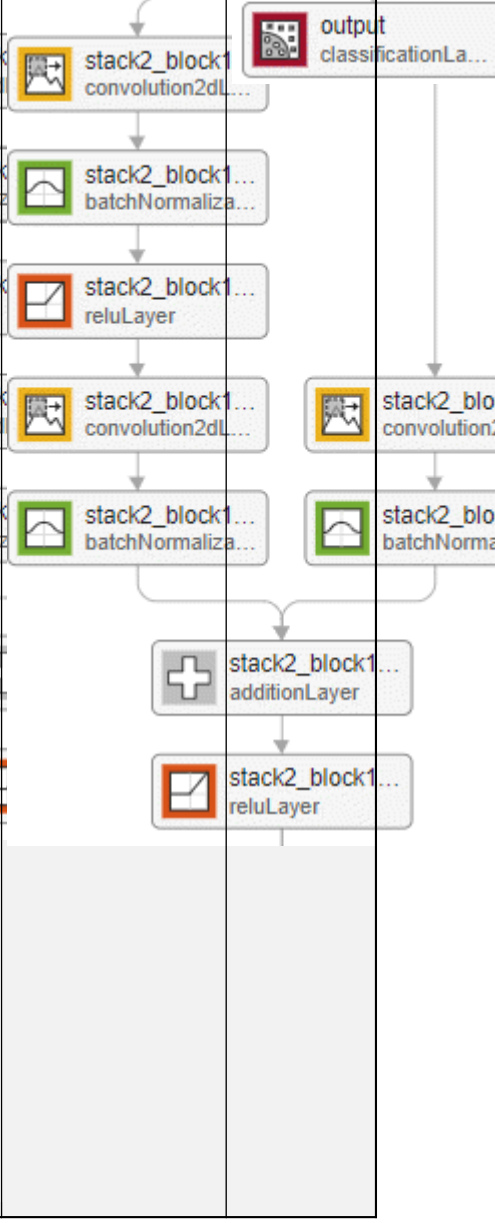
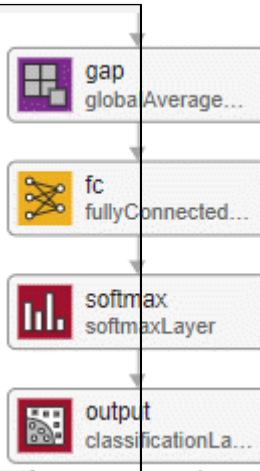
The layers inside each block are determined by the type of block and the options you set.

Block Layers

Name	Initial Layers	Initial Residual Block	Standard Residual Block (`BottleneckType="downsample-first-conv"`)	Standard Residual Block (`BottleneckType="none"`)	Downsampling Residual Block	Final Layers

Description	A residual network starts with the following layers, in order: • `imageInputLayer` • `convolution2dLayer` • `batchNormalizationLayer` • `reluLayer` • (Optional) Pooling layer (either max, average, or none) Set the optional pooling layer using the `InitialPoolingLayer` argument.	The main branch of the initial residual block has the same layers as a standard residual block. The `InitialNumFilters` and `NumFilters` values determine the layers on the residual connection. The residual connection has a convolutional layer with `[1,1]` filter and `[1,1]` stride if one of the following conditions is met: • `BottleneckType="downsample-first-conv"`(default) and `InitialNumFilters` is not equal to four times the first element of `NumFilters`. • `BottleneckType="none"` and `InitialNumFilters` is not equal to the first element of `NumFilters`. If `ResidualBlockType` is set	The standard residual block with bottleneck units has the following layers, in order: • `convolution2dLayer` with `[1,1]` filter and `[1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution2dLayer` with `[3,3]` filter and `[1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution2dLayer` with `[1,1]` filter and `[1,1]` stride • `batchNormalizationLayer` • `additionLayer` • `reluLayer` The standard block has a residual connection from the output of the previous block to the addition layer. Set the position of the addition layer using the	The standard residual block without bottleneck units has the following layers, in order: • `convolution2dLayer` with `[3,3]` filter and `[1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution2dLayer` with `[3,3]` filter and `[1,1]` stride • `batchNormalizationLayer` • `additionLayer` • `reluLayer` The standard block has a residual connection from the output of the previous block to the addition layer. Set the position of the addition layer using the `ResidualBlockType` argument.	The downsampling residual block is the same as the standard block (either with or without the bottleneck) but with a stride of `[2,2]` in the first convolutional layer and additional layers on the residual connection. The layers on the residual connection depend on the `ResidualBlockType` value. • When `ResidualBlockType` is set to `"batchnorm-before-add"`, the second branch contains a `convolution2dLayer` with `[1,1]` filter and `[2,2]` stride, and a `batchNormalizationLayer`. • When `ResidualBlockType` is set to `"batchnorm-after-add"`, the second branch	A residual network ends with the following layers, in order: • `globalAveragePooling2dLayer` • `fullyConnectedLayer` • `softmaxLayer` • `classificationLayer`

		to `"batchnorm-before-add"`, the residual connection will also have a batch normalization layer.	`ResidualBlockType` argument.		contains a `convolution2dLayer` with `[1,1]` filter and `[2,2]` stride. The downsampling block halves the height and width of the input, and increases the number of channels.	

Example Visualization		Example of an initial residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	Example of the standard residual block for a network with a bottleneck and with the batch normalization layer before the addition layer. 	Example of the standard residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	Example of a downsampling residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	

The convolution and fully connected layer weights are initialized using the He weight initialization method [3]. For more information, see `convolution2dLayer`.

Tips

- When working with small images, set the `InitialPoolingLayer` option to `"none"` to remove the initial pooling layer and reduce the amount of downsampling.
- Residual networks are usually named ResNet-*X*, where *X* is the *depth* of the network. The depth of a network is defined as the largest number of sequential convolutional or fully connected layers on a path from the input layer to the output layer. You can use the following formula to compute the depth of your network:

$$\text{depth} = \begin{cases} 1 + 2\sum_{i=1}^{N} s_i + 1 & \text{If no bottleneck} \\ 1 + 3\sum_{i=1}^{N} s_i + 1 & \text{If bottleneck} \end{cases},$$

  where s_i is the depth of stack i.

  Networks with the same depth can have different network architectures. For example, you can create a ResNet-14 architecture with or without a bottleneck:

```
resnet14Bottleneck = resnetLayers([224 224 3],10, ...
StackDepth=[2 2], ...
NumFilters=[64 128]);

resnet14NoBottleneck = resnetLayers([224 224 3],10, ...
BottleneckType="none", ...
StackDepth=[2 2 2], ...
NumFilters=[64 128 256]);
```

  The relationship between bottleneck and nonbottleneck architectures also means that a network with a bottleneck will have a different depth than a network without a bottleneck.

```
resnet50Bottleneck = resnetLayers([224 224 3],10);

resnet34NoBottleneck = resnetLayers([224 224 3],10, ...
BottleneckType="none");
```

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” Preprint, submitted December 10, 2015. https://arxiv.org/abs/1512.03385.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Identity Mappings in Deep Residual Networks.” Preprint, submitted July 25, 2016. https://arxiv.org/abs/1603.05027.

[3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

You can use the residual network for code generation. First, create the network using the `resnetLayers` function. Then, use the `trainNetwork` function to train the network. After training and evaluating the network, you can generate code for the `DAGNetwork` object by using GPU Coder™.

See Also

`resnet3dLayers` | `trainNetwork` | `trainingOptions` | `dlnetwork` | `resnet18` | `resnet50` | `resnet101`

Topics

“Train Residual Network for Image Classification”
“Pretrained Deep Neural Networks”
“Train Network Using Custom Training Loop”

Introduced in R2021b

resnet3dLayers

Create 3-D residual network

Syntax

```
lgraph = resnet3dLayers(inputSize,numClasses)
lgraph = resnet3dLayers( ___ ,Name=Value)
```

Description

`lgraph = resnet3dLayers(inputSize,numClasses)` creates a 3-D residual network with an image input size specified by `inputSize` and a number of classes specified by `numClasses`. A residual network consists of stacks of blocks. Each block contains deep learning layers. The network includes an image classification layer, suitable for predicting the categorical label of an input image.

To create a 2-D residual network, use `resnetLayers`.

`lgraph = resnet3dLayers( ___ ,Name=Value)` creates a residual network using one or more name-value arguments using any of the input arguments in the previous syntax. For example, `InitialNumFilters=32` specifies 32 filters in the initial convolutional layer.

Examples

3-D Residual Network with Bottleneck

Create a 3-D residual network with a bottleneck architecture.

```
imageSize = [224 224 64 3];
numClasses = 10;

lgraph = resnet3dLayers(imageSize,numClasses)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [177x1 nnet.cnn.layer.Layer]
    Connections: [192x2 table]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Analyze the network.

```
analyzeNetwork(lgraph)
```

This network is equivalent to a 3-D ResNet-50 residual network.

3-D Residual Network with Custom Stack Depth

Create a 3-D ResNet-101 network using a custom stack depth.

```
imageSize = [224 224 64 3];
numClasses = 10;

stackDepth = [3 4 23 3];
numFilters = [64 128 256 512];

lgraph = resnet3dLayers(imageSize,numClasses, ...
    StackDepth=stackDepth, ...
    NumFilters=numFilters)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [347x1 nnet.cnn.layer.Layer]
    Connections: [379x2 table]
     InputNames: {'input'}
    OutputNames: {'output'}
```

Analyze the network.

```
analyzeNetwork(lgraph)
```

Input Arguments

`inputSize` — Network input image size

3-element vector | 4-element vector

Network input image size, specified as one of the following:

- 3-element vector in the form [*height*, *width*, *depth*]
- 4-element vector in the form [*height*, *width*, *depth*, *channel*] where *channel* denotes the number of image channels.

The *height*, *width*, and *depth* values must be greater than or equal to *initialStride* * *poolingStride* * 2^D, where D is the number of downsampling blocks. Set the initial stride using the `InitialStride` argument. The pooling stride is `1` when the `InitialPoolingLayer` is set to `"none"`, and `2` otherwise.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`numClasses` — Number of classes

integer greater than 1

Number of classes in the image classification network, specified as an integer greater than 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `InitialFilterSize=[5,5,5],InitialNumFilters=32,BottleneckType="none"` specifies an initial filter size of 5-by-5-by-5 pixels, 32 initial filters, and a network architecture without bottleneck components.

Initial Layers

`InitialFilterSize` — Filter size in first convolutional layer
7 (default) | positive integer | 3-element vector of positive integers

Filter size in the first convolutional layer, specified as one of the following:

- Positive integer. The filter has equal height, width, and depth. For example, specifying `5` yields a filter of height 5, width 5, and depth 5.
- 3-element vector in the form [*height*, *width*, *depth*]. For example, specifying an initial filter size of `[1 5 2]` yields a filter of height 1, width 5, and depth 2.

Example: `InitialFilterSize=[5,5,5]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialNumFilters` — Number of filters in first convolutional layer
64 (default) | positive integer

Number of filters in the first convolutional layer, specified as a positive integer. The number of initial filters determines the number of channels (feature maps) in the output of the first convolutional layer in the residual network.

Example: `InitialNumFilters=32`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialStride` — Stride in first convolutional layer
2 (default) | positive integer | 3-element vector of positive integers

Stride in the first convolutional layer, specified as a:

- Positive integer. The stride has equal height, width, and depth. For example, specifying `3` yields a stride of height 3, width 3, and depth 3.
- 3-element vector in the form [*height*, *width*, *depth*]. For example, specifying an initial stride of `[1 2 2]` yields a stride of height 1, width 2, and depth 2.

The stride defines the step size for traversing the input in three dimensions.

Example: `InitialStride=[3,3,3]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialPoolingLayer` — First pooling layer
`"max"` (default) | `"average"` | `"none"`

First pooling layer before the initial residual block, specified as one of the following:

- `"max"` — Use a max pooling layer before the initial residual block. For more information, see `maxPooling3dLayer`.
- `"average"` — Use an average pooling layer before the initial residual block. For more information, see `globalAveragePooling3dLayer`.
- `"none"`— Do not use a pooling layer before the initial residual block.

Example: `InitialPoolingLayer="average"`

Data Types: `char` | `string`

Network Architecture

`ResidualBlockType` — Residual block type
`"batchnorm-before-add"` (default) | `"batchnorm-after-add"`

Residual block type, specified as one of the following:

- `"batchnorm-before-add"` — Add the batch normalization layer before the addition layer in the residual blocks [1].
- `"batchnorm-after-add"` — Add the batch normalization layer after the addition layer in the residual blocks [2].

The `ResidualBlockType` argument specifies the location of the batch normalization layer in the standard and downsampling residual blocks. For more information, see “More About” on page 1-1231.

Example: `ResidualBlockType="batchnorm-after-add"`

Data Types: `char` | `string`

`BottleneckType` — Block bottleneck type
`"downsample-first-conv"` (default) | `"none"`

Block bottleneck type, specified as one of the following:

- `"downsample-first-conv"` — Use bottleneck residual blocks that perform downsampling in the first convolutional layer of the downsampling residual blocks, using a stride of 2. A bottleneck residual block consists of three convolutional layers: a 1-by-1-by-1 layer for downsampling the channel dimension, a 3-by-3-by-3 convolutional layer, and a 1-by-1-by-1 layer for upsampling the channel dimension.

  The number of filters in the final convolutional layer is four times that in the first two convolutional layers. For more information, see “NumFilters” on page 1-0 .
- `"none"` — Do not use bottleneck residual blocks. The residual blocks consist of two 3-by-3-by-3 convolutional layers.

A bottleneck block performs a 1-by-1-by-1 convolution before the 3-by-3-by-3 convolution to reduce the number of channels by a factor of four. Networks with and without bottleneck blocks will have a similar level of computational complexity, but the total number of features propagating in the residual connections is four times larger when you use bottleneck units. Therefore, using a bottleneck increases the efficiency of the network [1]. For more information on the layers in each residual block, see “More About” on page 1-1231.

Example: `BottleneckType="none"`

Data Types: `char` | `string`

StackDepth — Number of residual blocks in each stack
`[3 4 6 3]` (default) | vector of positive integers

Number of residual blocks in each stack, specified as a vector of positive integers. For example, if the stack depth is `[3 4 6 3]`, the network has four stacks, with three blocks, four blocks, six blocks, and three blocks.

Specify the number of filters in the convolutional layers of each stack using the `NumFilters` argument. The `StackDepth` value must have the same number of elements as the `NumFilters` value.

Example: `StackDepth=[9 12 69 9]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

NumFilters — Number of filters in convolutional layers of each stack
`[64 128 256 512]` (default) | vector of positive integers

Number of filters in the convolutional layers of each stack, specified as a vector of positive integers.

- When you set `BottleneckType` to `"downsample-first-conv"`, the first two convolutional layers in each block of each stack have the same number of filters, set by the `NumFilters` value. The final convolutional layer has four times the number of filters in the first two convolutional layers.

  For example, suppose you set `NumFilters` to `[4 5]` and `BottleneckType` to `"downsample-first-conv"`. In the first stack, the first two convolutional layers in each block have 4 filters and the final convolutional layer in each block has 16 filters. In the second stack, the first two convolutional layers in each block have 5 filters and the final convolutional layer has 20 filters.
- When you set `BottleneckType` to `"none"`, the convolutional layers in each stack have the same number of filters, set by the `NumFilters` value.

The `NumFilters` value must have the same number of elements as the `StackDepth` value.

The `NumFilters` value determines the layers on the residual connection in the initial residual block. There is a convolutional layer on the residual connection if one of the following conditions is met:

- `BottleneckType="downsample-first-conv"`(default) and `InitialNumFilters` is not equal to four times the first element of `NumFilters`.
- `BottleneckType="none"` and `InitialNumFilters` is not equal to the first element of `NumFilters`.

For more information about the layers in each residual block, see “More About” on page 1-1231.

Example: `NumFilters=[32 64 126 256]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Normalization — Data normalization
`"zerocenter"` (default) | `"zscore"`

Data normalization to apply every time data is forward-propagated through the input layer, specified as one of the following:

- `"zerocenter"` — Subtract the mean. The mean is calculated at training time.

- `"zscore"` — Subtract the mean and divide by the standard deviation. The mean and standard deviation are calculated at training time.

Example: `Normalization="zscore"`

Data Types: `char` | `string`

Output Arguments

`lgraph` — 3-D residual network

`layerGraph` object

3-D residual network, returned as a `layerGraph` object.

More About

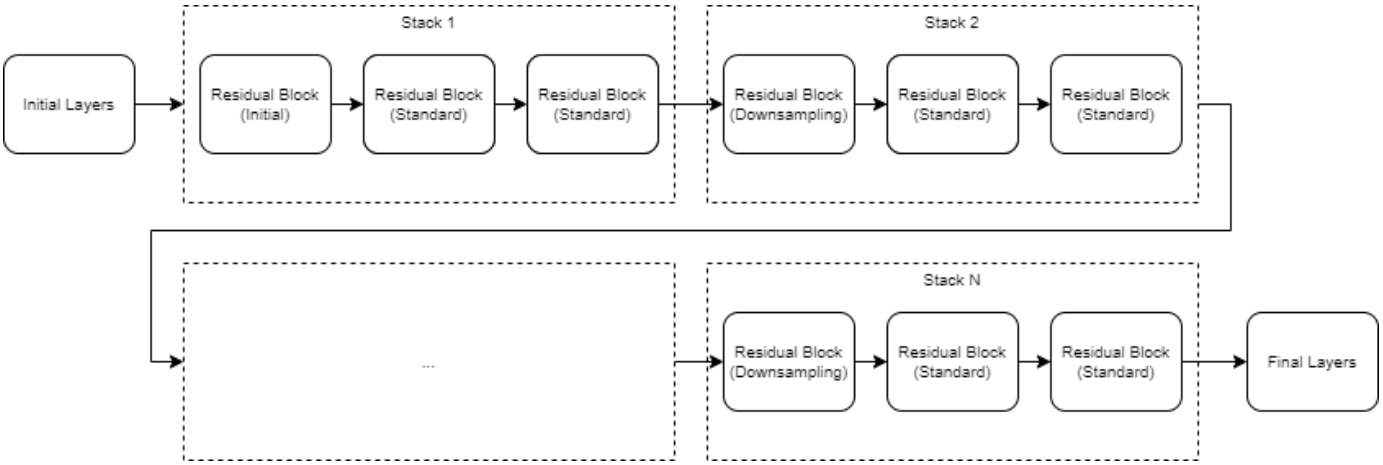
Residual Network

Residual networks (ResNets) are a type of deep network consisting of building blocks that have *residual connections* (also known as *skip* or *shortcut* connections). These connections allow the input to skip the convolutional units of the main branch, thus providing a simpler path through the network. By allowing the parameter gradients to flow more easily from the output layer to the earlier layers of the network, residual connections help mitigate the problem of vanishing gradients during early training.

The structure of a residual network is flexible. The key component is the inclusion of the residual connections within *residual blocks*. A group of residual blocks is called a *stack*. A ResNet architecture consists of initial layers, followed by stacks containing residual blocks, and then the final layers. A network has three types of residual blocks:

- Initial residual block — This block occurs at the start of the first stack. The layers in the residual connection of the initial residual block determine if the block preserves the activation sizes or performs downsampling.
- Standard residual block — This block occurs multiple times in each stack, after the first downsampling residual block. The standard residual block preserves the activation sizes.
- Downsampling residual block — This block occurs once, at the start of each stack. The first convolutional unit in the downsampling block downsamples the spatial dimensions by a factor of two.

A typical stack has a downsampling residual block, followed by `m` standard residual blocks, where `m` is greater than or equal to one. The first stack is the only stack that begins with an initial residual block.



The initial, standard, and downsampling residual blocks can be *bottleneck* or nonbottleneck blocks. Bottleneck blocks perform a 1-by-1-by-1 convolution before the 3-by-3-by-3 convolution, to reduce the number of channels by a factor of four. Networks with and without bottleneck blocks have a similar level of computational complexity, but the total number of features propagating in the residual connections is four times larger when you use the bottleneck units. Therefore, using bottleneck blocks increases the efficiency of the network.

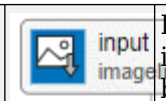
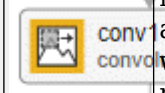
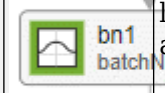
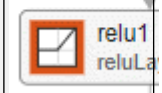
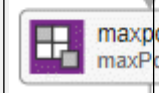
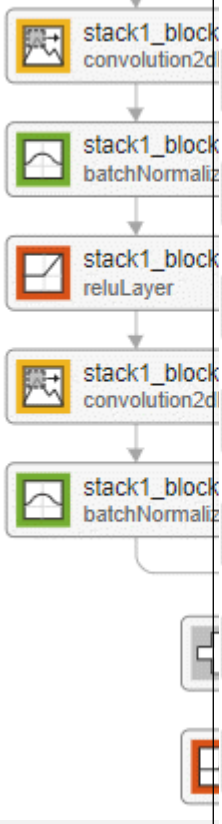
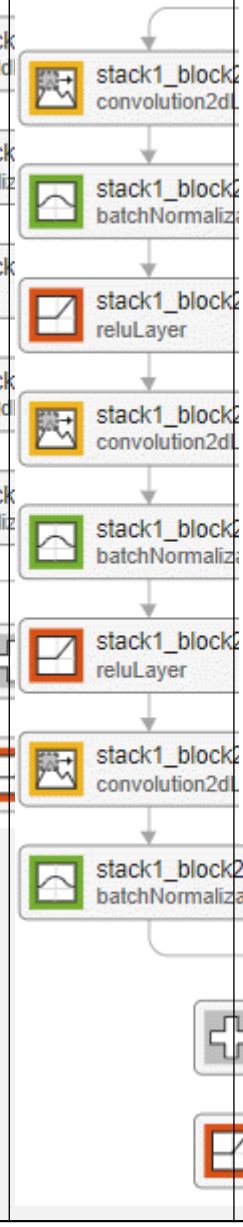
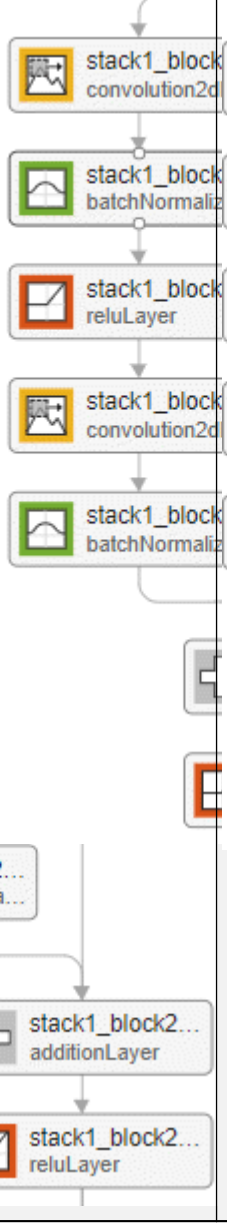
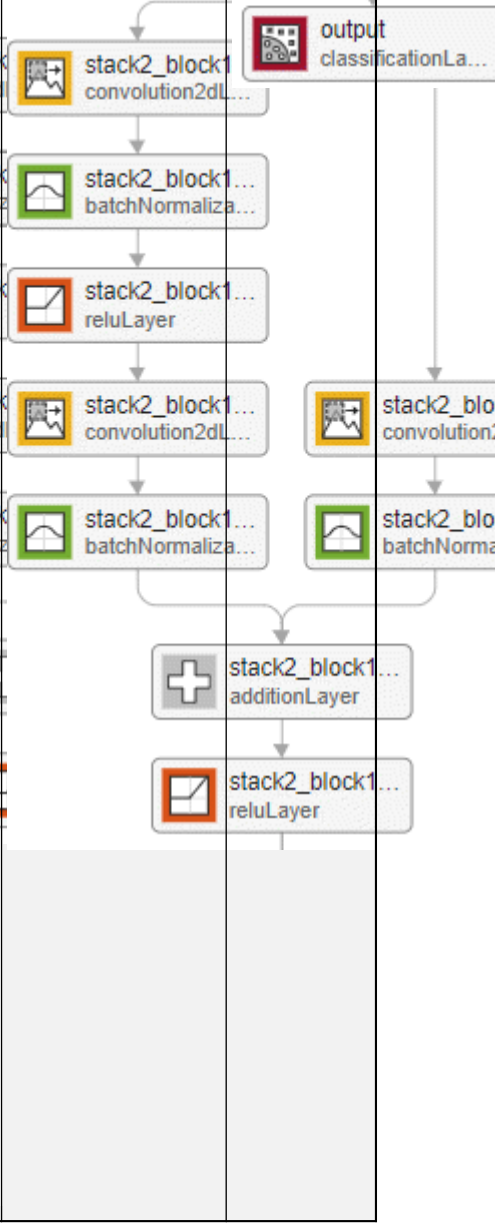
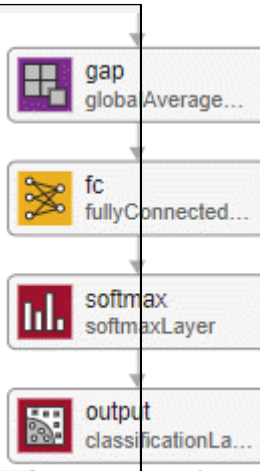
The layers inside each block are determined by the type of block and the options you set.

Block Layers

Name	Initial Layers	Initial Residual Block	Standard Residual Block (`BottleneckType="downsample-first-conv"`)	Standard Residual Block (`BottleneckType="none"`)	Downsampling Residual Block	Final Layers

Description	A residual network starts with the following layers, in order: • `image3dInputLayer` • `convolution3dLayer` • `batchNormalizationLayer` • `reluLayer` • (Optional) Pooling layer (either max, average, or none) Set the optional pooling layer using the `InitialPoolingLayer` argument.	The main branch of the initial residual block has the same layers as a standard residual block. The `InitialNumFilters` and `NumFilters` values determine the layers on the residual connection. The residual connection has a convolutional layer with `[1,1,1]` filter and `[1,1,1]` stride if one of the following conditions is met: • `BottleneckType="downsample-first-conv"`(default) and `InitialNumFilters` is not equal to four times the first element of `NumFilters`. • `BottleneckType="none"` and `InitialNumFilters` is not equal to the first element of `NumFilters`. If `ResidualBlock`	The standard residual block with bottleneck units has the following layers, in order: • `convolution3dLayer` with `[1,1,1]` filter and `[1,1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution3dLayer` with `[3,3,3]` filter and `[1,1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution3dLayer` with `[1,1,1]` filter and `[1,1,1]` stride • `batchNormalizationLayer` • `additionLayer` • `reluLayer` The standard block has a residual connection from the output of the previous block to	The standard residual block without bottleneck units has the following layers, in order: • `convolution3dLayer` with `[3,3,3]` filter and `[1,1,1]` stride • `batchNormalizationLayer` • `reluLayer` • `convolution3dLayer` with `[3,3,3]` filter and `[1,1,1]` stride • `batchNormalizationLayer` • `additionLayer` • `reluLayer` The standard block has a residual connection from the output of the previous block to the addition layer. Set the position of the addition layer using the `ResidualBlockType` argument.	The downsampling residual block is the same as the standard block (either with or without the bottleneck) but with a stride of `[2,2,2]` in the first convolutional layer and additional layers on the residual connection. The layers on the residual connection depend on the `ResidualBlockType` value. • When `ResidualBlockType` is set to `"batchnorm-before-add"`, the second branch contains a `convolution3dLayer` with `[1,1,1]` filter and `[2,2,2]` stride, and a `batchNormalizationLayer`. • When `ResidualBlockType` is set to `"batchnorm-after-add"`, the second	A residual network ends with the following layers, in order: • `globalAveragePooling3dLayer` • `fullyConnectedLayer` • `softmaxLayer` • `classificationLayer`

		`Type` is set to`"batchnorm-before-add"`, the residual connection will also have a batch normalization layer.	the addition layer. Set the position of the addition layer using the `ResidualBlock Type` argument.		branch contains a `convolution3dLayer` with `[1,1,1]` filter and `[2,2,2]` stride. The downsampling block halves the height and width of the input, and increases the number of channels.	

Example Visualization	    	Example of an initial residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	Example of the standard residual block for a network with a bottleneck and with the batch normalization layer before the addition layer. 	Example of the standard residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	Example of a downsampling residual block for a network without a bottleneck and with the batch normalization layer before the addition layer. 	

The convolution and fully connected layer weights are initialized using the He weight initialization method [3]. For more information, see `convolution3dLayer`.

Tips

- When working with small images, set the `InitialPoolingLayer` option to `"none"` to remove the initial pooling layer and reduce the amount of downsampling.
- Residual networks are usually named ResNet-*X*, where *X* is the *depth* of the network. The depth of a network is defined as the largest number of sequential convolutional or fully connected layers on a path from the input layer to the output layer. You can use the following formula to compute the depth of your network:

$$\text{depth} = \begin{cases} 1 + 2\sum_{i=1}^{N} s_i + 1 & \text{If no bottleneck} \\ 1 + 3\sum_{i=1}^{N} s_i + 1 & \text{If bottleneck} \end{cases},$$

  where s_i is the depth of stack i.

  Networks with the same depth can have different network architectures. For example, you can create a 3-D ResNet-14 architecture with or without a bottleneck:

```
resnet14Bottleneck = resnet3dLayers([224 224 64 3],10, ...
StackDepth=[2 2], ...
NumFilters=[64 128]);

resnet14NoBottleneck = resnet3dLayers([224 224 64 3],10, ...
BottleneckType="none", ...
StackDepth=[2 2 2], ...
NumFilters=[64 128 256]);
```

  The relationship between bottleneck and nonbottleneck architectures also means that a network with a bottleneck will have a different depth than a network without a bottleneck.

```
resnet50Bottleneck = resnet3dLayers([224 224 64 3],10);

resnet34NoBottleneck = resnet3dLayers([224 224 64 3],10, ...
BottleneckType="none");
```

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” Preprint, submitted December 10, 2015. https://arxiv.org/abs/1512.03385.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Identity Mappings in Deep Residual Networks.” Preprint, submitted July 25, 2016. https://arxiv.org/abs/1603.05027.

[3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

You can use the residual network for code generation. First, create the network using the `resnet3dLayers` function. Then, use the `trainNetwork` function to train the network. After training and evaluating the network, you can generate code for the `DAGNetwork` object by using GPU Coder™.

See Also

`resnetLayers` | `trainNetwork` | `trainingOptions` | `dlnetwork` | `resnet18` | `resnet50` | `resnet101`

Topics

“Train Residual Network for Image Classification”
“Pretrained Deep Neural Networks”
“Train Network Using Custom Training Loop”

Introduced in R2021b

sequenceFoldingLayer

Sequence folding layer

Description

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.

To use a sequence folding layer, you must connect the `miniBatchSize` output to the `miniBatchSize` input of the corresponding sequence unfolding layer. For an example, see “Create Network for Video Classification” on page 1-1240.

Creation

Syntax

```
layer = sequenceFoldingLayer
layer = sequenceFoldingLayer('Name',Name)
```

Description

`layer = sequenceFoldingLayer` creates a sequence folding layer.

`layer = sequenceFoldingLayer('Name',Name)` creates a sequence folding layer and sets the optional `Name` property using a name-value pair. For example, `sequenceFoldingLayer('Name','fold1')` creates a sequence folding layer with the name `'fold1'`. Enclose the property name in single quotes.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
2 (default)

Number of outputs of the layer.

The layer has two outputs:

- `'out'` – Output feature map corresponding to reshaped input.
- `'miniBatchSize'` – Size of the mini-batch passed into the layer. This output must be connected to the `'miniBatchSize'` input of the corresponding sequence unfolding layer.

Data Types: `double`

`OutputNames` — Output names
`{'out','miniBatchSize'}` (default)

Output names of the layer.

The layer has two outputs:

- `'out'` – Output feature map corresponding to reshaped input.
- `'miniBatchSize'` – Size of the mini-batch passed into the layer. This output must be connected to the `'miniBatchSize'` input of the corresponding sequence unfolding layer.

Data Types: `cell`

Examples

Create Sequence Folding Layer

Create a sequence folding layer with name the `'fold1'`.

```
layer = sequenceFoldingLayer('Name','fold1')
```

```
layer =
  SequenceFoldingLayer with properties:

           Name: 'fold1'
     NumOutputs: 2
    OutputNames: {'out'  'miniBatchSize'}
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of `[28 28 1]`.
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

```
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')
```

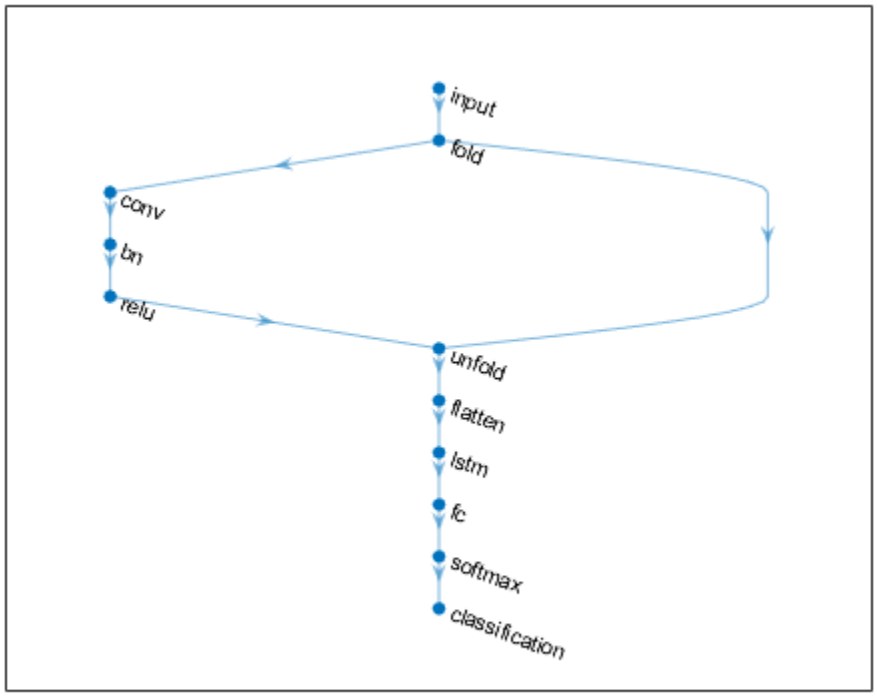
```
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')];
```

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');
```

View the final network architecture using the `plot` function.

```
figure
plot(lgraph)
```



Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `resetState` | `flattenLayer` | `sequenceUnfoldingLayer` | `sequenceInputLayer`

Topics

“Classify Videos Using Deep Learning”

“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2019a

sequenceInputLayer

Sequence input layer

Description

A sequence input layer inputs sequence data to a network.

Creation

Syntax

```
layer = sequenceInputLayer(inputSize)
layer = sequenceInputLayer(inputSize,Name,Value)
```

Description

`layer = sequenceInputLayer(inputSize)` creates a sequence input layer and sets the `InputSize` property.

`layer = sequenceInputLayer(inputSize,Name,Value)` sets the optional `MinLength`, `Normalization`, `Mean`, and `Name` properties using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

Sequence Input

`InputSize` — Size of input
positive integer | vector of positive integers

Size of the input, specified as a positive integer or a vector of positive integers.

- For vector sequence input, `InputSize` is a scalar corresponding to the number of features.
- For 1-D image sequence input, `InputSize` is vector of two elements `[h c]`, where `h` is the image height and `c` is the number of channels of the image.
- For 2-D image sequence input, `InputSize` is vector of three elements `[h w c]`, where `h` is the image height, `w` is the image width, and `c` is the number of channels of the image.
- For 3-D image sequence input, `InputSize` is vector of four elements `[h w d c]`, where `h` is the image height, `w` is the image width, `d` is the image depth, and `c` is the number of channels of the image.

To specify the minimum sequence length of the input data, use the `MinLength` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MinLength` — Minimum sequence length of input data
1 (default) | positive integer

Minimum sequence length of input data, specified as a positive integer. When training or making predictions with the network, if the input data has fewer than `MinLength` time steps, then the software throws an error.

When you create a network that downsamples data in the time dimension, you must take care that the network supports your training data and any data for prediction. Some deep learning layers require that the input has a minimum sequence length. For example, a 1-D convolution layer requires that the input has at least as many time steps as the filter size.

As time series of sequence data propagates through a network, the sequence length can change. For example, downsampling operations such as 1-D convolutions can output data with fewer time steps than its input. This means that downsampling operations can cause later layers in the network to throw an error because the data has a shorter sequence length than the minimum length required by the layer.

When you train or assemble a network, the software automatically checks that sequences of length 1 can propagate through the network. Some networks might not support sequences of length 1, but can successfully propagate sequences of longer lengths. To check that a network supports propagating your training and expected prediction data, set the `MinLength` property to a value less than or equal to the minimum length of your data and the expected minimum length of your prediction data.

Tip To prevent convolution and pooling layers from changing the size of the data, set the `Padding` option of the layer to `"same"` or `"causal"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Normalization` — Data normalization

`'none'` (default) | `'zerocenter'` | `'zscore'` | `'rescale-symmetric'` | `'rescale-zero-one'` | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by `Mean`.
- `'zscore'` — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form `Y = func(X)`, where `X` is the input data and the output `Y` is the normalized data.

Tip The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

If the input data contains padding, then the layer ignored padding values when normalizing the input data.

Data Types: `char` | `string` | `function_handle`

`NormalizationDimension` — Normalization dimension
`'auto'` (default) | `'channel'` | `'element'` | `'all'`

Normalization dimension, specified as one of the following:

- `'auto'` – If the training option is `false` and you specify any of the normalization statistics (`Mean`, `StandardDeviation`, `Min`, or `Max`), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- `'channel'` – Channel-wise normalization.
- `'element'` – Element-wise normalization.
- `'all'` – Normalize all values using scalar statistics.

Data Types: `char` | `string`

`Mean` — Mean for zero-center and z-score normalization
`[]` (default) | numeric array | numeric scalar

Mean for zero-center and z-score normalization, specified as a numeric array, or empty.

- For vector sequence input, `Mean` must be a `InputSize`-by-1 vector of means per channel, a numeric scalar, or `[]`.
- For 2-D image sequence input, `Mean` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-`InputSize(3)` array of means per channel, a numeric scalar, or `[]`.
- For 3-D image sequence input, `Mean` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-1-by-`InputSize(4)` array of means per channel, a numeric scalar, or `[]`.

If you specify the `Mean` property, then `Normalization` must be `'zerocenter'` or `'zscore'`. If `Mean` is `[]`, then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StandardDeviation` — Standard deviation
`[]` (default) | numeric array | numeric scalar

Standard deviation used for z-score normalization, specified as a numeric array, a numeric scalar, or empty.

- For vector sequence input, `StandardDeviation` must be a `InputSize`-by-1 vector of standard deviations per channel, a numeric scalar, or `[]`.
- For 2-D image sequence input, `StandardDeviation` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-`InputSize(3)` array of standard deviations per channel, a numeric scalar, or `[]`.
- For 3-D image sequence input, `StandardDeviation` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-1-by-`InputSize(4)` array of standard deviations per channel, or a numeric scalar.

If you specify the `StandardDeviation` property, then `Normalization` must be `'zscore'`. If `StandardDeviation` is `[]`, then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Min` — Minimum value for rescaling
`[]` (default) | numeric array | numeric scalar

Minimum value for rescaling, specified as a numeric array, or empty.

- For vector sequence input, `Min` must be a `InputSize`-by-1 vector of means per channel or a numeric scalar.
- For 2-D image sequence input, `Min` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-`InputSize(3)` array of minima per channel, or a numeric scalar.
- For 3-D image sequence input, `Min` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-1-by-`InputSize(4)` array of minima per channel, or a numeric scalar.

If you specify the `Min` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Min` is `[]`, then the software calculates the minima at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Max` — Maximum value for rescaling
`[]` (default) | numeric array | numeric scalar

Maximum value for rescaling, specified as a numeric array, or empty.

- For vector sequence input, `Max` must be a `InputSize`-by-1 vector of means per channel or a numeric scalar.
- For 2-D image sequence input, `Max` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-`InputSize(3)` array of maxima per channel, a numeric scalar, or `[]`.
- For 3-D image sequence input, `Max` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-1-by-`InputSize(4)` array of maxima per channel, a numeric scalar, or `[]`.

If you specify the `Max` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Max` is `[]`, then the software calculates the maxima at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

`InputNames` — Input names
`{}` (default)

Input names of the layer. The layer has no inputs.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Sequence Input Layer

Create a sequence input layer with the name `'seq1'` and an input size of 12.

```
layer = sequenceInputLayer(12,'Name','seq1')
```

```
layer =
  SequenceInputLayer with properties:

                     Name: 'seq1'
                InputSize: 12
                MinLength: 1

   Hyperparameters
             Normalization: 'none'
    NormalizationDimension: 'auto'
```

Include a sequence input layer in a `Layer` array.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
```

```
layers = [ ...
    sequenceInputLayer(inputSize)
    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
```

Create Sequence Input Layer for Image Sequences

Create a sequence input layer for sequences of 224-224 RGB images with the name `'seq1'`.

```
layer = sequenceInputLayer([224 224 3], 'Name', 'seq1')

layer =
  SequenceInputLayer with properties:

                      Name: 'seq1'
                 InputSize: [224 224 3]
                 MinLength: 1

   Hyperparameters
             Normalization: 'none'
    NormalizationDimension: 'auto'
```

Train Network for Sequence Classification

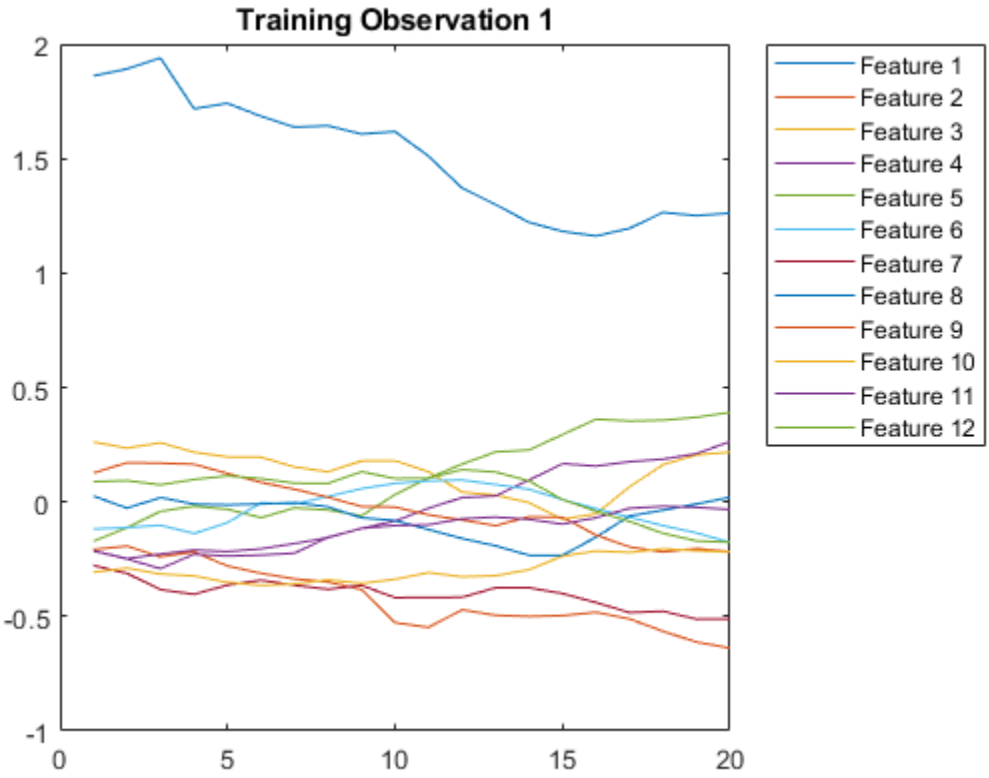
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with 12 features corresponding to LPC cepstrum coefficients. `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;
```

Visualize the first time series in a plot. Each line corresponds to a feature.

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



Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 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.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    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
```

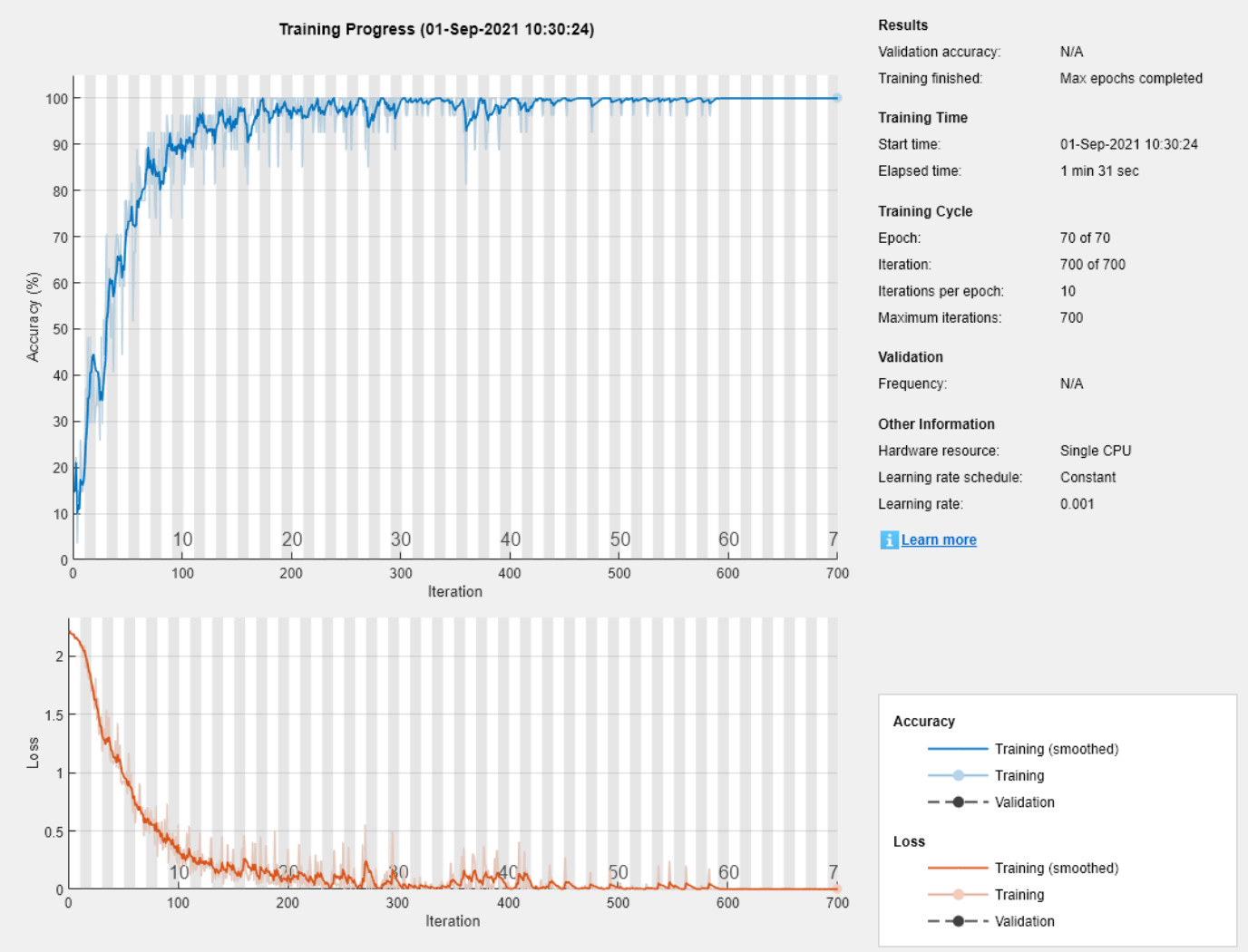
Specify the training options. Specify the solver as `'adam'` and `'GradientThreshold'` as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 70.

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).

```
maxEpochs = 70;
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',1, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the LSTM network with the specified training options.

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



Load the test set and classify the sequences into speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Classify the test data. Specify the same mini-batch size used for training.

```
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9541
```

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”.

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”.

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];
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of `[28 28 1]`.
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

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

layers = [ ...
```

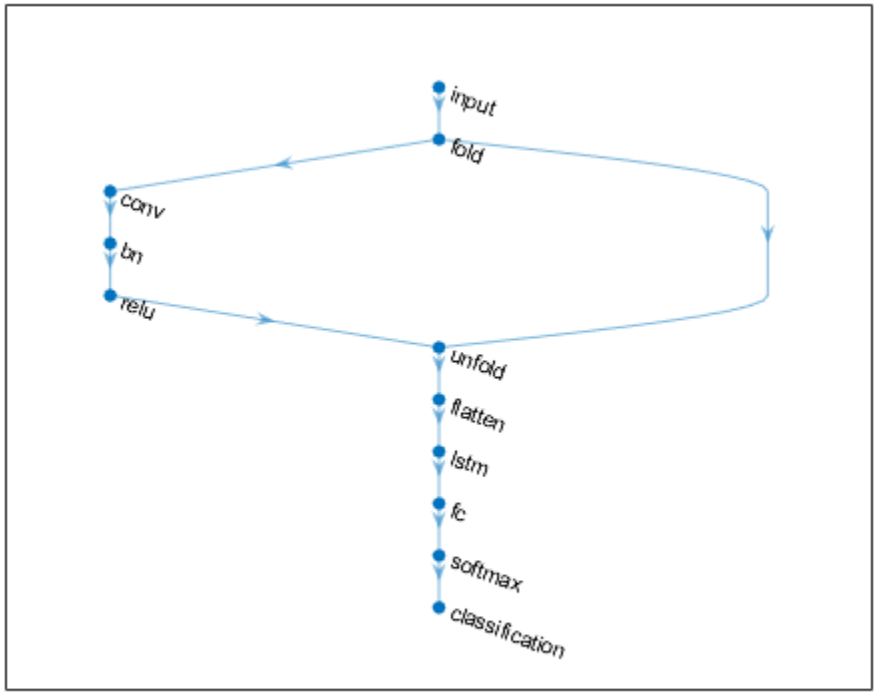
```
    sequenceInputLayer(inputSize,'Name','input')

    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');
```

View the final network architecture using the `plot` function.

```
figure
plot(lgraph)
```



Compatibility Considerations

`sequenceInputLayer`, by default, uses channel-wise normalization for zero-center normalization
Behavior change in future release

Starting in R2019b, `sequenceInputLayer`, by default, uses channel-wise normalization for zero-center normalization. In previous versions, this layer uses element-wise normalization. To reproduce this behavior, set the `NormalizationDimension` option of this layer to `'element'`.

`sequenceInputLayer` ignores padding values when normalizing
Behavior changed in R2020a

Starting in R2020a, `sequenceInputLayer` objects ignore padding values in the input data when normalizing. This means that the `Normalization` option in the `sequenceInputLayer` now makes training invariant to data operations, for example, `'zerocenter'` normalization now implies that the training results are invariant to the mean of the data.

If you train on padded sequences, then the calculated normalization factors may be different in earlier versions and can produce different results.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- For vector sequence inputs, the number of features must be a constant during code generation.
- Code generation does not support `'Normalization'` specified using a function handle.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

To generate CUDA or C++ code by using GPU Coder, you must first construct and train a deep neural network. Once the network is trained and evaluated, you can configure the code generator to generate code and deploy the convolutional neural network on platforms that use NVIDIA or ARM GPU processors. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

For this layer, you can generate code that takes advantage of the NVIDIA CUDA deep neural network library (cuDNN), or the NVIDIA TensorRT high performance inference library.

- The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences.
- For vector sequence inputs, the number of features must be a constant during code generation.
- For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.
- Code generation does not support `'Normalization'` specified using a function handle.

See Also

`lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `resetState` | `sequenceFoldingLayer` | `flattenLayer` | `sequenceUnfoldingLayer` | **Deep Network Designer** | `featureInputLayer`

Topics

“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Classify Videos Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2017b

sequenceUnfoldingLayer

Sequence unfolding layer

Description

A sequence unfolding layer restores the sequence structure of the input data after sequence folding.

To use a sequence unfolding layer, you must connect the `miniBatchSize` output of the corresponding sequence folding layer to the `miniBatchSize` input of the sequence unfolding layer. For an example, see “Create Network for Video Classification” on page 1-1259.

Creation

Syntax

```
layer = sequenceUnfoldingLayer
layer = sequenceUnfoldingLayer('Name',Name)
```

Description

`layer = sequenceUnfoldingLayer` creates a sequence unfolding layer.

`layer = sequenceUnfoldingLayer('Name',Name)` creates a sequence unfolding layer and sets the optional `Name` property using a name-value pair. For example, `sequenceUnfoldingLayer('Name','unfold1')` creates a sequence unfolding layer with the name `'unfold1'`. Enclose the property name in single quotes.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
2 (default)

Number of inputs of the layer.

This layer has two inputs:

- `'in'` – Input feature map.

- `'miniBatchSize'` – Size of the mini-batch from the corresponding sequence folding layer. This output must be connected to the `'miniBatchSize'` output of the corresponding sequence folding layer.

Data Types: `double`

`InputNames` — Input names
`{'in','miniBatchSize'}` (default)

Input names of the layer.

This layer has two inputs:

- `'in'` – Input feature map.
- `'miniBatchSize'` – Size of the mini-batch from the corresponding sequence folding layer. This output must be connected to the `'miniBatchSize'` output of the corresponding sequence folding layer.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Sequence Unfolding Layer

Create a sequence unfolding layer with the name `'unfold1'`.

```
layer = sequenceUnfoldingLayer('Name','unfold1')
```

```
layer =
  SequenceUnfoldingLayer with properties:

         Name: 'unfold1'
    NumInputs: 2
   InputNames: {'in'  'miniBatchSize'}
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of `[28 28 1]`.
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

```
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')
```

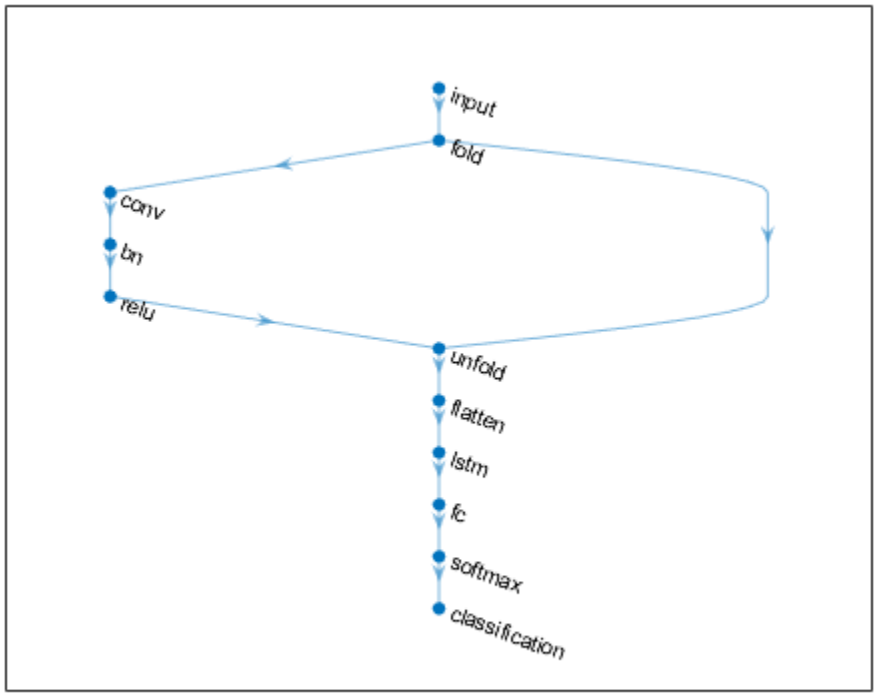
```
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')];
```

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');
```

View the final network architecture using the `plot` function.

```
figure
plot(lgraph)
```



Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `resetState` | `sequenceFoldingLayer` | `flattenLayer` | `sequenceInputLayer`

Topics

“Classify Videos Using Deep Learning”

“Classify Videos Using Deep Learning”
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Long Short-Term Memory Networks”
“Visualize Activations of LSTM Network”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2019a

SeriesNetwork

Series network for deep learning

Description

A series network is a neural network for deep learning with layers arranged one after the other. It has a single input layer and a single output layer.

Creation

There are several ways to create a `SeriesNetwork` object:

- Load a pretrained network using `alexnet`, `darknet19`, `vgg16`, or `vgg19`. For an example, see “Load Pretrained AlexNet Convolutional Neural Network” on page 1-1264.
- Train or fine-tune a network using `trainNetwork`. For an example, see “Train Network for Image Classification” on page 1-1265.
- Import a pretrained network from TensorFlow-Keras, Caffe, or the ONNX (Open Neural Network Exchange) model format.
  - For a Keras model, use `importKerasNetwork`. For an example, see “Import and Plot Keras Network” on page 1-794.
  - For a Caffe model, use `importCaffeNetwork`. For an example, see “Import Caffe Network” on page 1-772.
  - For an ONNX model, use `importONNXNetwork`. For an example, see “Import ONNX Network as DAGNetwork” on page 1-848.
- Assemble a deep learning network from pretrained layers using the `assembleNetwork` function.

Note To learn about other pretrained networks, such as `googlenet` and `resnet50`, see “Pretrained Deep Neural Networks”.

Properties

`Layers` — Network layers
`Layer` array

This property is read-only.

Network layers, specified as a `Layer` array.

`InputNames` — Network input layer names
cell array of character vectors

This property is read-only.

Network input layer names, specified as a cell array of character vectors.

Data Types: `cell`

`OutputNames` — Network output layer names
cell array

Network output layer names, specified as a cell array of character vectors.

Data Types: `cell`

Object Functions

activations	Compute deep learning network layer activations
classify	Classify data using a trained deep learning neural network
predict	Predict responses using a trained deep learning neural network
predictAndUpdateState	Predict responses using a trained recurrent neural network and update the network state
classifyAndUpdateState	Classify data using a trained recurrent neural network and update the network state
resetState	Reset the state of a recurrent neural network
plot	Plot neural network layer graph

Examples

Load Pretrained AlexNet Convolutional Neural Network

Load a pretrained AlexNet convolutional neural network and examine the layers and classes.

Load the pretrained AlexNet network using `alexnet`. The output `net` is a `SeriesNetwork` object.

```
net = alexnet
```

```
net =
  SeriesNetwork with properties:

    Layers: [25×1 nnet.cnn.layer.Layer]
```

Using the `Layers` property, view the network architecture. The network comprises of 25 layers. There are 8 layers with learnable weights: 5 convolutional layers, and 3 fully connected layers.

```
net.Layers
```

```
ans =
  25x1 Layer array with layers:

     1   'data'     Image Input                   227x227x3 images with 'zerocenter' normalizatio
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride [4  4] and
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] and padding
     6   'conv2'    Grouped Convolution           2 groups of 128 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] and padding
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride [1  1] and
    11   'relu3'    ReLU                          ReLU
```

```
    12   'conv4'    Grouped Convolution          2 groups of 192 3x3x192 convolutions with stric
    13   'relu4'    ReLU                         ReLU
    14   'conv5'    Grouped Convolution          2 groups of 128 3x3x192 convolutions with stric
    15   'relu5'    ReLU                         ReLU
    16   'pool5'    Max Pooling                  3x3 max pooling with stride [2  2] and padding
    17   'fc6'      Fully Connected              4096 fully connected layer
    18   'relu6'    ReLU                         ReLU
    19   'drop6'    Dropout                      50% dropout
    20   'fc7'      Fully Connected              4096 fully connected layer
    21   'relu7'    ReLU                         ReLU
    22   'drop7'    Dropout                      50% dropout
    23   'fc8'      Fully Connected              1000 fully connected layer
    24   'prob'     Softmax                      softmax
    25   'output'   Classification Output        crossentropyex with 'tench' and 999 other class
```

You can view the names of the classes learned by the network by viewing the `Classes` property of the classification output layer (the final layer). View the first 10 classes by selecting the first 10 elements.

```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical array
     tench
     goldfish
     great white shark
     tiger shark
     hammerhead
     electric ray
     stingray
     cock
     hen
     ostrich
```

Import Layers from Caffe Network

Specify the example file `'digitsnet.prototxt'` to import.

```
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```
layers = importCaffeLayers(protofile)
```

```
layers =

  1x7 Layer array with layers:

     1   'testdata'   Image Input             28x28x1 images
     2   'conv1'      Convolution            20 5x5x1 convolutions with stride [1  1] and padding [0  0]
     3   'relu1'      ReLU                   ReLU
     4   'pool1'      Max Pooling            2x2 max pooling with stride [2  2] and padding [0  0]
     5   'ip1'        Fully Connected        10 fully connected layer
     6   'loss'       Softmax                softmax
     7   'output'     Classification Output  crossentropyex with 'class1', 'class2', and 8 other classes
```

Train Network for Image Classification

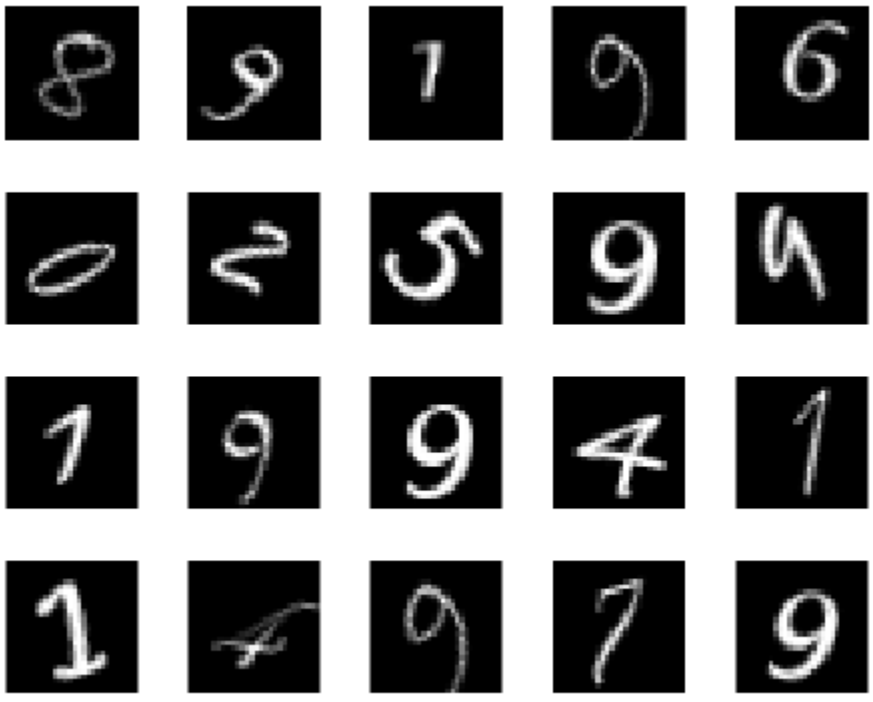
Load the data as an `ImageDatastore` object.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

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.

Display some of the images in the datastore.

```
figure
numImages = 10000;
perm = randperm(numImages,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
    drawnow;
end
```



Divide the datastore so that each category in the training set has 750 images and the testing set has the remaining images from each label.

```
numTrainingFiles = 750;
[imdsTrain,imdsTest] = splitEachLabel(imds,numTrainingFiles,'randomize');
```

`splitEachLabel` splits the image files in `digitData` into two new datastores, `imdsTrain` and `imdsTest`.

Define the convolutional neural network architecture.

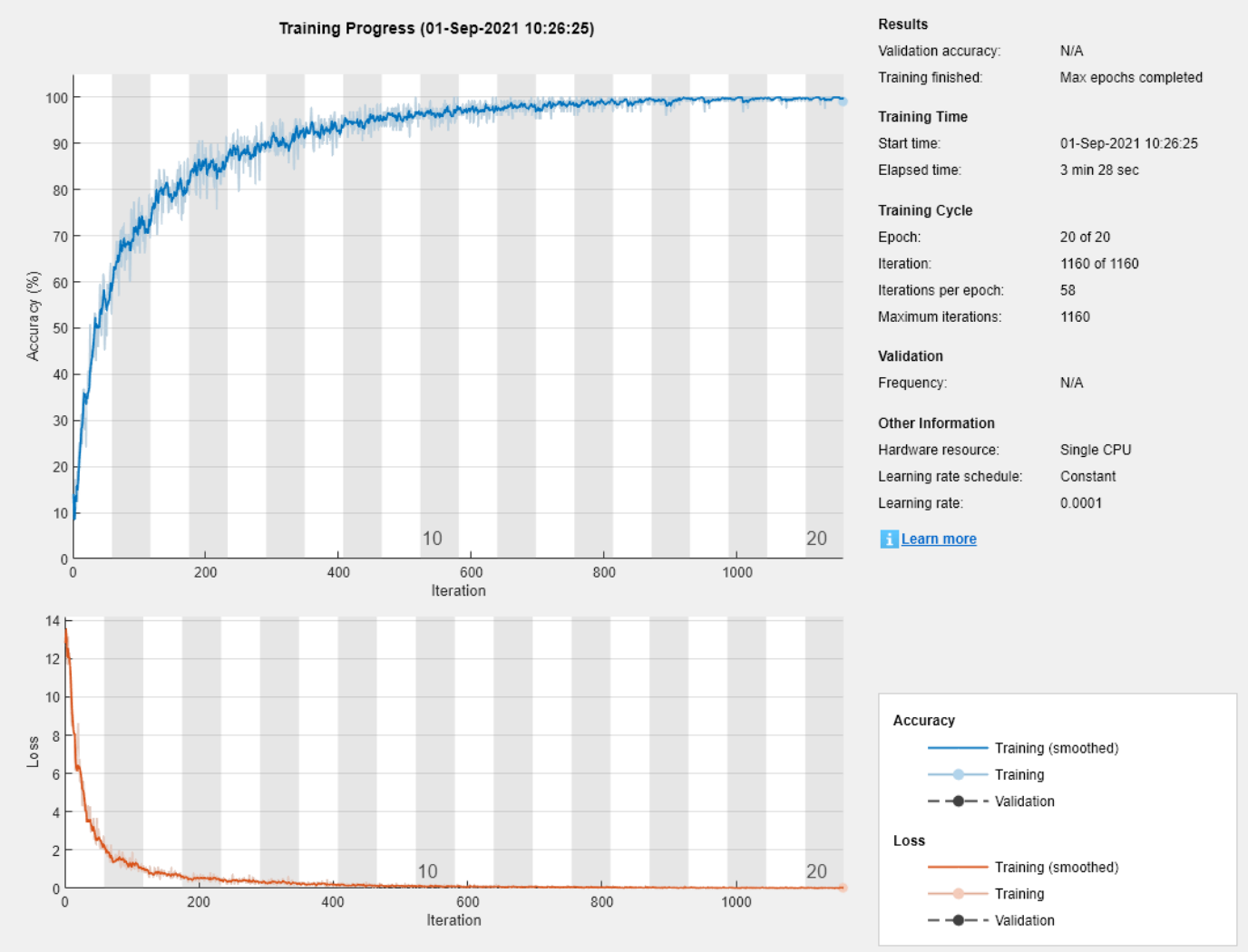
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.0001.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',20,...
    'InitialLearnRate',1e-4, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(imdsTrain,layers,options);
```



Run the trained network on the test set, which was not used to train the network, and predict the image labels (digits).

```
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy. The accuracy is the ratio of the number of true labels in the test data matching the classifications from `classify` to the number of images in the test data.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9412
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, `classifyAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, `classifyAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (GPU Coder).

See Also

`alexnet` | `vgg16` | `vgg19` | `darknet19` | `importCaffeNetwork` | `trainNetwork` | `trainingOptions` | `DAGNetwork` | `analyzeNetwork` | `assembleNetwork` | `plot` | `classify` | `predict`

Topics
"Create Simple Deep Learning Network for Classification"
"Train Convolutional Neural Network for Regression"
"Sequence Classification Using Deep Learning"
"Deep Learning in MATLAB"
"Specify Layers of Convolutional Neural Network"
"Define Custom Deep Learning Layers"
"Long Short-Term Memory Networks"

Introduced in R2016a

setL2Factor

Package: nnet.cnn.layer

Set L2 regularization factor of layer learnable parameter

Syntax

```
layer = setL2Factor(layer,parameterName,factor)
layerUpdated = setL2Factor(layer,parameterPath,factor)

dlnetUpdated = setL2Factor(dlnet,layerName,parameterName,factor)
dlnetUpdated = setL2Factor(dlnet,parameterPath,factor)
```

Description

`layer = setL2Factor(layer,parameterName,factor)` sets the L2 regularization factor of the parameter with the name `parameterName` in `layer` to `factor`.

For built-in layers, you can set the L2 regularization factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `layer = setL2Factor(layer,'Weights',factor)` is equivalent to `layer.WeightL2Factor = factor`.

`layerUpdated = setL2Factor(layer,parameterPath,factor)` sets the L2 regularization factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a `dlnetwork` object in a custom layer.

`dlnetUpdated = setL2Factor(dlnet,layerName,parameterName,factor)` sets the L2 regularization factor of the parameter with the name `parameterName` in the layer with name `layerName` for the specified `dlnetwork` object.

`dlnetUpdated = setL2Factor(dlnet,parameterPath,factor)` sets the L2 regularization factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a nested layer.

Examples

Set and Get L2 Regularization Factor of Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a layer.

Create a layer array containing the custom layer `preluLayer`, attached to this is example as a supporting file. To access this layer, open this example as a live script.

Create a layer array including a custom layer `preluLayer`.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
```

```
    preluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the L2 regularization factor of the `Alpha` learnable parameter of the `preluLayer` to 2.

```
layers(4) = setL2Factor(layers(4),"Alpha",2);
```

View the updated L2 regularization factor.

```
factor = getL2Factor(layers(4),"Alpha")
```

```
factor = 2
```

Set and Get L2 Regularization Factor of Nested Layer Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a nested layer.

Create a residual block layer using the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
numFilters = 64;
layer = residualBlockLayer(numFilters)
```

```
layer =
  residualBlockLayer with properties:

       Name: ''

   Learnable Parameters
    Network: [1x1 dlnetwork]

   State Parameters
    No properties.

  Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     2   'gn1'     Group Normalization   Group normalization
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     5   'gn2'     Group Normalization   Group normalization
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the L2 regularization factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setL2Factor` function.

```
factor = 2;
layer = setL2Factor(layer,'Network/conv1/Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(layer,'Network/conv1/Weights')
```

```
factor = 2
```

Set and Get L2 Regularization Factor of `dlnetwork` Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a `dlnetwork` object.

Create a `dlnetwork` object.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')
    convolution2dLayer(5,20,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

lgraph = layerGraph(layers);

dlnet = dlnetwork(lgraph);
```

Set the L2 regularization factor of the `'Weights'` learnable parameter of the convolution layer to 2 using the `setL2Factor` function.

```
factor = 2;
dlnet = setL2Factor(dlnet,'conv','Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(dlnet,'conv','Weights')
```

```
factor = 2
```

Set and Get L2 Regularization Factor of Nested `dlnetwork` Learnable Parameter

Set and get the L2 regularization factor of a learnable parameter of a nested layer in a `dlnetwork` object.

Create a `dlnetwork` object containing the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
inputSize = [224 224 3];
numFilters = 32;
numClasses = 5;

layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    convolution2dLayer(7,numFilters,'Stride',2,'Padding','same','Name','conv')
    groupNormalizationLayer('all-channels','Name','gn')
    reluLayer('Name','relu')
    maxPooling2dLayer(3,'Stride',2,'Name','max')
    residualBlockLayer(numFilters,'Name','res1')
    residualBlockLayer(numFilters,'Name','res2')
    residualBlockLayer(2*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res3')
    residualBlockLayer(2*numFilters,'Name','res4')
    residualBlockLayer(4*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res5')
    residualBlockLayer(4*numFilters,'Name','res6')
    globalAveragePooling2dLayer('Name','gap')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')];

dlnet = dlnetwork(layers);
```

The `Learnables` property of the `dlnetwork` object is a table that contains the learnable parameters of the network. The table includes parameters of nested layers in separate rows. View the learnable parameters of the layer `"res1"`.

```
learnables = dlnet.Learnables;
idx = learnables.Layer == "res1";
learnables(idx,:)
```

```
ans=8×3 table
    Layer            Parameter                 Value
    _____    ____________________________    ____________________

    "res1"    "Network/conv1/Weights"    {3x3x32x32 dlarray}
    "res1"    "Network/conv1/Bias"       {1x1x32    dlarray}
    "res1"    "Network/gn1/Offset"       {1x1x32    dlarray}
    "res1"    "Network/gn1/Scale"        {1x1x32    dlarray}
    "res1"    "Network/conv2/Weights"    {3x3x32x32 dlarray}
    "res1"    "Network/conv2/Bias"       {1x1x32    dlarray}
    "res1"    "Network/gn2/Offset"       {1x1x32    dlarray}
    "res1"    "Network/gn2/Scale"        {1x1x32    dlarray}
```

For the layer `"res1"`, set the L2 regularization factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setL2Factor` function.

```
factor = 2;
dlnet = setL2Factor(dlnet,'res1/Network/conv1/Weights',factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(dlnet,'res1/Network/conv1/Weights')
```

```
factor = 2
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name
character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

`factor` — L2 regularization factor
nonnegative scalar

L2 regularization factor for the parameter, specified as a nonnegative scalar.

The software multiplies this factor with the global L2 regularization factor to determine the L2 regularization factor for the specified parameter. For example, if `factor` is 2, then the L2 regularization for the specified parameter is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

`parameterPath` — Path to parameter in nested layer
string scalar | character vector

Path to parameter in nested layer, specified as a string scalar or a character vector. A nested layer is a custom layer that itself defines a layer graph as a learnable parameter.

If the input to `setL2Factor` is a nested layer, then the parameter path has the form `"propertyName/layerName/parameterName"`, where:

- `propertyName` is the name of the property containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"propertyName1/layerName1/.../propertyNameN/layerNameN/parameterName"`, where `propertyName1` and `layerName1` correspond to the layer in the input to the `setL2Factor` function, and the subsequent parts correspond to the deeper levels.

Example: For layer input to `setL2Factor`, the path `"Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`.

If the input to `setL2Factor` is a `dlnetwork` object and the desired parameter is in a nested layer, then the parameter path has the form `"layerName1/propertyName/layerName/parameterName"`, where:

- `layerName1` is the name of the layer in the input `dlnetwork` object
- `propertyName` is the property of the layer containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"layerName1/propertyName1/.../layerNameN/propertyNameN/layerName/parameterName"`, where `layerName1` and `propertyName1` correspond to the layer in the input to the `setL2Factor` function, and the subsequent parts correspond to the deeper levels.

Example: For `dlnetwork` input to `setL2Factor`, the path `"res1/Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`, where `layer` is the layer with name `"res1"` in the input network `dlnet`.

Data Types: `char` | `string`

`dlnet` — Network for custom training loops
`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`layerName` — Layer name
string scalar | character vector

Layer name, specified as a string scalar or a character vector.

Data Types: `char` | `string`

Output Arguments

`layerUpdated` — Updated layer
`Layer` object

Updated layer, returned as a `Layer`.

`dlnetUpdated` — Updated network
`dlnetwork` object

Updated network, returned as a `dlnetwork`.

See Also

`setLearnRateFactor` | `getLearnRateFactor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b

setLearnRateFactor

Package: `nnet.cnn.layer`

Set learn rate factor of layer learnable parameter

Syntax

```
layerUpdated = setLearnRateFactor(layer,parameterName,factor)
layerUpdated = setLearnRateFactor(layer,parameterPath,factor)

dlnetUpdated = setLearnRateFactor(dlnet,layerName,parameterName,factor)
dlnetUpdated = setLearnRateFactor(dlnet,parameterPath,factor)
```

Description

`layerUpdated = setLearnRateFactor(layer,parameterName,factor)` sets the learn rate factor of the parameter with the name `parameterName` in `layer` to `factor`.

For built-in layers, you can set the learn rate factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `layer = setLearnRateFactor(layer,'Weights',factor)` is equivalent to `layer.WeightLearnRateFactor = factor`.

`layerUpdated = setLearnRateFactor(layer,parameterPath,factor)` sets the learn rate factor of the parameter specified by the path `parameterPath`. Use this syntax when the parameter is in a `dlnetwork` object in a custom layer.

`dlnetUpdated = setLearnRateFactor(dlnet,layerName,parameterName,factor)` sets the learn rate factor of the parameter with the name `parameterName` in the layer with name `layerName` for the specified `dlnetwork` object.

`dlnetUpdated = setLearnRateFactor(dlnet,parameterPath,factor)` sets the learn rate factor of the parameter specifiedby the path `parameterPath`. Use this syntax when the parameter is in a nested layer.

Examples

Set and Get Learning Rate Factor of Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a custom PReLU layer.

Create a layer array containing the custom layer `preluLayer`, attached to this is example as a supporting file. To access this layer, open this example as a live script.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    preluLayer(20)
```

```
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the learn rate factor of the `Alpha` learnable parameter of the `preluLayer` to 2.

```
layers(4) = setLearnRateFactor(layers(4),"Alpha",2);
```

View the updated learn rate factor.

```
factor = getLearnRateFactor(layers(4),"Alpha")
```

```
factor = 2
```

Set and Get Learning Rate Factor of Nested Layer Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a nested layer.

Create a residual block layer using the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
numFilters = 64;
layer = residualBlockLayer(numFilters)
```

```
layer =
  residualBlockLayer with properties:

       Name: ''

   Learnable Parameters
    Network: [1x1 dlnetwork]

   State Parameters
    No properties.

  Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     2   'gn1'     Group Normalization   Group normalization
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           64 3x3 convolutions with stride [1  1] and padding 'same
     5   'gn2'     Group Normalization   Group normalization
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the learning rate factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setLearnRateFactor` function.

```
factor = 2;
layer = setLearnRateFactor(layer,'Network/conv1/Weights',factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(layer,'Network/conv1/Weights')
```

```
factor = 2
```

Set and Get Learn Rate Factor of `dlnetwork` Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a `dlnetwork` object.

Create a `dlnetwork` object.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')
    convolution2dLayer(5,20,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','sm')];

lgraph = layerGraph(layers);

dlnet = dlnetwork(lgraph);
```

Set the learn rate factor of the `'Weights'` learnable parameter of the convolution layer to 2 using the `setLearnRateFactor` function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'conv','Weights',factor);
```

Get the updated learn rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(dlnet,'conv','Weights')
```

```
factor = 2
```

Set and Get Learning Rate Factor of Nested `dlnetwork` Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a nested layer in a `dlnetwork` object.

Create a `dlnetwork` object containing the custom layer `residualBlockLayer` attached to this example as a supporting file. To access this file, open this example as a Live Script.

```
inputSize = [224 224 3];
numFilters = 32;
numClasses = 5;

layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    convolution2dLayer(7,numFilters,'Stride',2,'Padding','same','Name','conv')
    groupNormalizationLayer('all-channels','Name','gn')
    reluLayer('Name','relu')
    maxPooling2dLayer(3,'Stride',2,'Name','max')
    residualBlockLayer(numFilters,'Name','res1')
    residualBlockLayer(numFilters,'Name','res2')
    residualBlockLayer(2*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res3')
    residualBlockLayer(2*numFilters,'Name','res4')
    residualBlockLayer(4*numFilters,'Stride',2,'IncludeSkipConvolution',true,'Name','res5')
    residualBlockLayer(4*numFilters,'Name','res6')
    globalAveragePooling2dLayer('Name','gap')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')];

dlnet = dlnetwork(layers);
```

View the layers of the nested network in the layer `'res1'`.

```
dlnet.Layers(6).Network.Layers
```

```
ans =
  7x1 Layer array with layers:

     1   'conv1'   Convolution           32 3x3x32 convolutions with stride [1  1] and padding 's
     2   'gn1'     Group Normalization   Group normalization with 32 channels split into 1 groups
     3   'relu1'   ReLU                  ReLU
     4   'conv2'   Convolution           32 3x3x32 convolutions with stride [1  1] and padding 's
     5   'gn2'     Group Normalization   Group normalization with 32 channels split into 32 group
     6   'add'     Addition              Element-wise addition of 2 inputs
     7   'relu2'   ReLU                  ReLU
```

Set the learning rate factor of the learnable parameter `'Weights'` of the layer `'conv1'` to 2 using the `setLearnRateFactor` function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'res1/Network/conv1/Weights',factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(dlnet,'res1/Network/conv1/Weights')
```

```
factor = 2
```

Freeze Learnable Parameters of `dlnetwork` Object

Load a pretrained network.

```
net = squeezenet;
```

Convert the network to a layer graph, remove the output layer, and convert it to a `dlnetwork` object.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,'ClassificationLayer_predictions');
dlnet = dlnetwork(lgraph);
```

The `Learnables` property of the `dlnetwork` object is a table that contains the learnable parameters of the network. The table includes parameters of nested layers in separate rows. View the first few rows of the learnables table.

```
learnables = dlnet.Learnables;
head(learnables)
```

```
ans=8×3 table
          Layer           Parameter           Value
    ____________________    _________    ____________________

    "conv1"                 "Weights"    {3x3x3x64  dlarray}
    "conv1"                 "Bias"       {1x1x64    dlarray}
    "fire2-squeeze1x1"      "Weights"    {1x1x64x16 dlarray}
    "fire2-squeeze1x1"      "Bias"       {1x1x16    dlarray}
    "fire2-expand1x1"       "Weights"    {1x1x16x64 dlarray}
    "fire2-expand1x1"       "Bias"       {1x1x64    dlarray}
    "fire2-expand3x3"       "Weights"    {3x3x16x64 dlarray}
    "fire2-expand3x3"       "Bias"       {1x1x64    dlarray}
```

To freeze the learnable parameters of the network, loop over the learnable parameters and set the learn rate to 0 using the `setLearnRateFactor` function.

```
factor = 0;

numLearnables = size(learnables,1);
for i = 1:numLearnables
    layerName = learnables.Layer(i);
    parameterName = learnables.Parameter(i);

    dlnet = setLearnRateFactor(dlnet,layerName,parameterName,factor);
end
```

To use the updated learn rate factors when training, you must pass the dlnetwork object to the update function in the custom training loop. For example, use the command

```
[dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity);
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name

character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

`factor` — Learning rate factor

nonnegative scalar

Learning rate factor for the parameter, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the specified parameter. For example, if `factor` is 2, then the learning rate for the specified parameter is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

`parameterPath` — Path to parameter in nested layer
string scalar | character vector

Path to parameter in nested layer, specified as a string scalar or a character vector. A nested layer is a custom layer that itself defines a layer graph as a learnable parameter.

If the input to `setLearnRateFactor` is a nested layer, then the parameter path has the form `"propertyName/layerName/parameterName"`, where:

- `propertyName` is the name of the property containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"propertyName1/layerName1/.../propertyNameN/layerNameN/parameterName"`, where `propertyName1` and `layerName1` correspond to the layer in the input to the `setLearnRateFactor` function, and the subsequent parts correspond to the deeper levels.

Example: For layer input to `setLearnRateFactor`, the path `"Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`.

If the input to `setLearnRateFactor` is a `dlnetwork` object and the desired parameter is in a nested layer, then the parameter path has the form `"layerName1/propertyName/layerName/parameterName"`, where:

- `layerName1` is the name of the layer in the input `dlnetwork` object
- `propertyName` is the property of the layer containing a `dlnetwork` object
- `layerName` is the name of the layer in the `dlnetwork` object
- `parameterName` is the name of the parameter

If there are multiple levels of nested layers, then specify each level using the form `"layerName1/propertyName1/.../layerNameN/propertyNameN/layerName/parameterName"`, where `layerName1` and `propertyName1` correspond to the layer in the input to the `setLearnRateFactor` function, and the subsequent parts correspond to the deeper levels.

Example: For `dlnetwork` input to `setLearnRateFactor`, the path `"res1/Network/conv1/Weights"` specifies the `"Weights"` parameter of the layer with name `"conv1"` in the `dlnetwork` object given by `layer.Network`, where `layer` is the layer with name `"res1"` in the input network `dlnet`.

Data Types: `char` | `string`

`dlnet` — Network for custom training loops
`dlnetwork` object

Network for custom training loops, specified as a `dlnetwork` object.

`layerName` — Layer name
string scalar | character vector

Layer name, specified as a string scalar or a character vector.

Data Types: `char` | `string`

Output Arguments

`layerUpdated` — Updated layer
`Layer` object

Updated layer, returned as a `Layer`.

`dlnetUpdated` — Updated network
`dlnetwork` object

Updated network, returned as a `dlnetwork`.

See Also

`setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b

sgdmupdate

Update parameters using stochastic gradient descent with momentum (SGDM)

Syntax

```
[dlnet,vel] = sgdmupdate(dlnet,grad,vel)
[params,vel] = sgdmupdate(params,grad,vel)
[___] = sgdmupdate(___,learnRate,momentum)
```

Description

Update the network learnable parameters in a custom training loop using the stochastic gradient descent with momentum (SGDM) algorithm.

Note This function applies the SGDM optimization algorithm to update network parameters in custom training loops that use networks defined as `dlnetwork` objects or model functions. If you want to train a network defined as a `Layer` array or as a `LayerGraph`, use the following functions:

- Create a `TrainingOptionsSGDM` object using the `trainingOptions` function.
- Use the `TrainingOptionsSGDM` object with the `trainNetwork` function.

`[dlnet,vel] = sgdmupdate(dlnet,grad,vel)` updates the learnable parameters of the network `dlnet` using the SGDM algorithm. Use this syntax in a training loop to iteratively update a network defined as a `dlnetwork` object.

`[params,vel] = sgdmupdate(params,grad,vel)` updates the learnable parameters in `params` using the SGDM algorithm. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined using functions.

`[___] = sgdmupdate(___,learnRate,momentum)` also specifies values to use for the global learning rate and momentum, in addition to the input arguments in previous syntaxes.

Examples

Update Learnable Parameters Using `sgdmupdate`

Perform a single SGDM update step with a global learning rate of `0.05` and momentum of `0.95`.

Create the parameters and parameter gradients as numeric arrays.

```
params = rand(3,3,4);
grad = ones(3,3,4);
```

Initialize the parameter velocities for the first iteration.

```
vel = [];
```

Specify custom values for the global learning rate and momentum.

```
learnRate = 0.05;
momentum = 0.95;
```

Update the learnable parameters using `sgdmupdate`.

```
[params,vel] = sgdmupdate(params,grad,vel,learnRate,momentum);
```

Train Network Using `sgdmupdate`

Use `sgdmupdate` to train a network using the SGDM algorithm.

Load Training Data

Load the digits training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

Define Network

Define the network architecture and specify the average image value 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')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

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

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of the example. The function takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```
miniBatchSize = 128;
numEpochs = 20;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

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

Visualize the training progress in a plot.

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

Train Network

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the `sgdmupdate` function. At the end of each epoch, display the training progress.

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
```

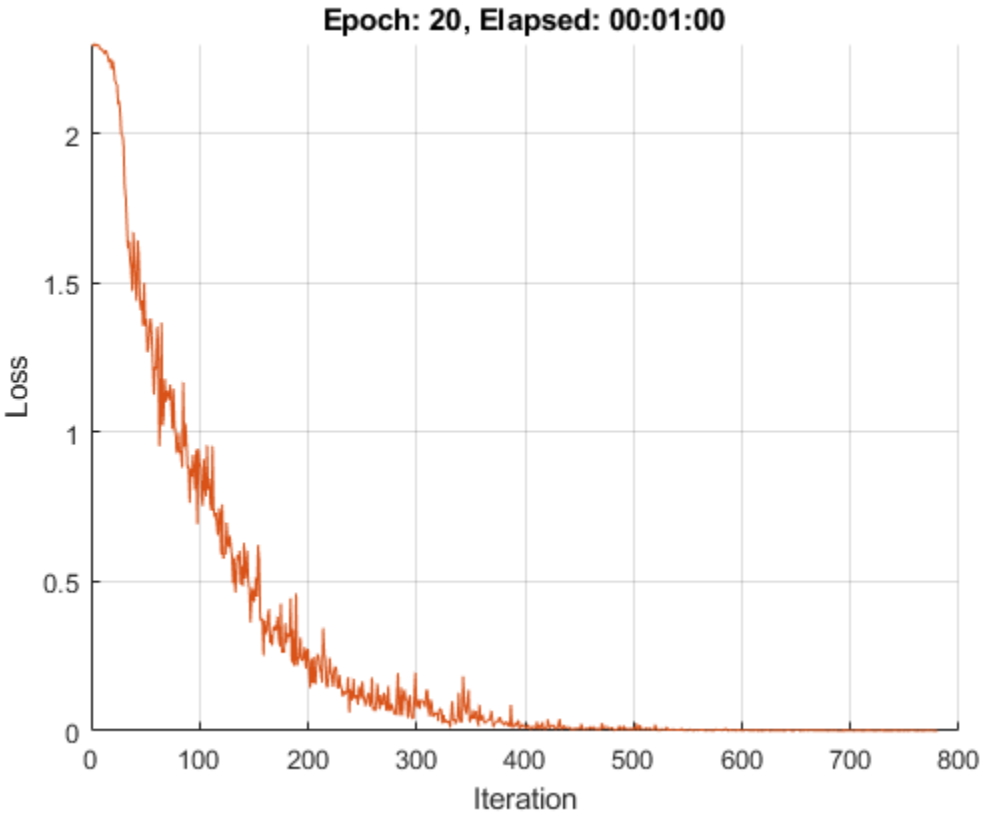
Initialize the velocity parameter.

```
vel = [];
```

Train the network.

```
iteration = 0;
start = tic;

for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);

    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 a dlarray.
        dlX = dlarray(single(X),'SSCB');
```

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

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients helper function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

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

        % 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 the Network

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

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

To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.9916
```

Model Gradients Function

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

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

    dlYPred = forward(dlnet,dlX);

    loss = crossentropy(dlYPred,Y);

    gradients = dlgradient(loss,dlnet.Learnables);

end
```

Input Arguments

`dlnet` — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object. `dlnet.Learnables` is a table with three variables:

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

The input argument `grad` must be a table of the same form as `dlnet.Learnables`.

`params` — Network learnable parameters

`dlarray` | numeric array | cell array | structure | table

Network learnable parameters, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

If you specify `params` as a table, it must contain the following three variables.

- `Layer` — Layer name, specified as a string scalar.
- `Parameter` — Parameter name, specified as a string scalar.
- `Value` — Value of parameter, specified as a cell array containing a `dlarray`.

You can specify `params` as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `grad` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

Data Types: `single` | `double` | `struct` | `table` | `cell`

`grad` — Gradients of the loss
`dlarray` | numeric array | cell array | structure | table

Gradients of the loss, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `grad` depends on the input network or learnable parameters. The following table shows the required format for `grad` for possible inputs to `sgdmupdate`.

Input	Learnable Parameters	Gradients
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`

Input	Learnable Parameters	Gradients
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `grad` must have a `Value` variable consisting of cell arrays that contain the gradient of each learnable parameter.

You can obtain `grad` from a call to `dlfeval` that evaluates a function that contains a call to `dlgradient`. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

`vel` — Parameter velocities
`[]` | `dlarray` | numeric array | cell array | structure | table

Parameter velocities, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `vel` depends on the input network or learnable parameters. The following table shows the required format for `vel` for possible inputs to `sgdmpdate`.

Input	Learnable Parameters	Velocities
`dlnet`	Table `dlnet.Learnables` containing `Layer`, `Parameter`, and `Value` variables. The `Value` variable consists of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data type, variables, and ordering as `dlnet.Learnables`. `vel` must have a `Value` variable consisting of cell arrays that contain the velocity of each learnable parameter.
`params`	`dlarray`	`dlarray` with the same data type and ordering as `params`
	Numeric array	Numeric array with the same data type and ordering as `params`
	Cell array	Cell array with the same data types, structure, and ordering as `params`
	Structure	Structure with the same data types, fields, and ordering as `params`
	Table with `Layer`, `Parameter`, and `Value` variables. The `Value` variable must consist of cell arrays that contain each learnable parameter as a `dlarray`.	Table with the same data types, variables, and ordering as `params`. `vel` must have a `Value` variable consisting of cell arrays that contain the velocity of each learnable parameter.

If you specify `vel` as an empty array, the function assumes no previous velocities and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `vel` output of a previous call to `sgdmupdate` as the `vel` input.

learnRate — Global learning rate
0.01 (default) | positive scalar

Learning rate, specified as a positive scalar. The default value of learnRate is 0.01.

If you specify the network parameters as a dlnetwork object, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

momentum — Momentum
0.9 (default) | positive scalar between 0 and 1

Momentum, specified as a positive scalar between 0 and 1. The default value of momentum is 0.9.

Output Arguments

dlnet — Updated network
dlnetwork object

Network, returned as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object.

params — Updated network learnable parameters
dlarray | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a dlarray, a numeric array, a cell array, a structure, or a table with a Value variable containing the updated learnable parameters of the network.

vel — Updated parameter velocities
dlarray | numeric array | cell array | structure | table

Updated parameter velocities, returned as a dlarray, a numeric array, a cell array, a structure, or a table.

More About

Stochastic Gradient Descent with Momentum

The function uses the stochastic gradient descent with momentum algorithm to update the learnable parameters. For more information, see the definition of the stochastic gradient descent with momentum algorithm under “Stochastic Gradient Descent” on page 1-1368 on the trainingOptions reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.

- `grad`
- `params`

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlnetwork` | `dlarray` | `dlupdate` | `rmspropupdate` | `adamupdate` | `forward` | `dlgradient` | `dlfeval`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”

Introduced in R2019b

shuffle

Shuffle data in `augmentedImageDatastore`

Syntax

```
auimds2 = shuffle(auimds)
```

Description

`auimds2 = shuffle(auimds)` returns an `augmentedImageDatastore` object containing a random ordering of the data from augmented image datastore `auimds`.

Input Arguments

`auimds` — Augmented image datastore

`augmentedImageDatastore`

Augmented image datastore, specified as an `augmentedImageDatastore` object.

Output Arguments

`auimds2` — Output datastore

`augmentedImageDatastore` object

Output datastore, returned as an `augmentedImageDatastore` object containing randomly ordered files from `auimds`.

See Also

`read` | `readByIndex` | `readall`

Introduced in R2018a

shuffle

Shuffle data in minibatchqueue

Syntax

`shuffle(mbq)`

Description

`shuffle(mbq)` resets the data held in `mbq` and shuffles it into a random order. After shuffling, the next function returns different mini-batches. Use this syntax to reset and shuffle your data after each training epoch in a custom training loop.

Examples

Differences Between `shuffle` and `reset`

The `shuffle` function resets and shuffles the `minibatchqueue` object so that you can obtain data from it in a random order. By contrast, the `reset` function resets the `minibatchqueue` object to the start of the underlying datastore.

Create a `minibatchqueue` object from a datastore.

```
ds = digitDatastore;
mbq = minibatchqueue(ds,'MinibatchSize',256)
```

```
mbq =
minibatchqueue with 1 output and properties:

   Mini-batch creation:
           MiniBatchSize: 256
        PartialMiniBatch: 'return'
            MiniBatchFcn: 'collate'
     DispatchInBackground: 0

   Outputs:
              OutputCast: {'single'}
         OutputAsDlarray: 1
         MiniBatchFormat: {''}
       OutputEnvironment: {'auto'}
```

Obtain the first mini-batch of data.

```
X1 = next(mbq);
```

Iterate over the rest of the data in the `minibatchqueue` object. Use `hasdata` to check if data is still available.

```
while hasdata(mbq)
    next(mbq);
end
```

Shuffle the `minibatchqueue` object and obtain the first mini-batch after the queue is shuffled.

```
shuffle(mbq);
X2 = next(mbq);
```

Iterate over the remaining data again.

```
while hasdata(mbq)
    next(mbq);
end
```

Reset the `minibatchqueue` object and obtain the first mini-batch after the queue is reset.

```
reset(mbq);
X3 = next(mbq);
```

Check whether the mini-batches obtained after resetting or shuffling the `minibatchqueue` object are the same as the first mini-batch after the `minibatchqueue` object is created.

```
isequal(X1,X2)
isequal(X1,X3)
```

```
ans =
   0
ans =
   1
```

The `reset` function returns the `minibatchqueue` object to the start of the underlying data, so that the `next` function returns mini-batches in the same order each time. By contrast, the `shuffle` function shuffles the underlying data and produces randomized mini-batches.

Input Arguments

`mbq` — Queue of mini-batches

`minibatchqueue`

Queue of mini-batches, specified as a `minibatchqueue` object.

See Also

`hasdata` | `next` | `minibatchqueue` | `reset`

Topics

“Train Deep Learning Model in MATLAB”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Generative Adversarial Network (GAN)”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2020b

shufflenet

Pretrained ShuffleNet convolutional neural network

Syntax

```
net = shufflenet
```

Description

ShuffleNet is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the ShuffleNet model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ShuffleNet.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ShuffleNet instead of GoogLeNet.

`net = shufflenet` returns a pretrained ShuffleNet convolutional neural network.

This function requires the *Deep Learning Toolbox Model for ShuffleNet Network* support package. If this support package is not installed, then the function provides a download link.

Examples

Download ShuffleNet Support Package

Download and install the *Deep Learning Toolbox Model for ShuffleNet Network* support package.

Type `shufflenet` at the command line.

```
shufflenet
```

If the *Deep Learning Toolbox Model for ShuffleNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `shufflenet` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

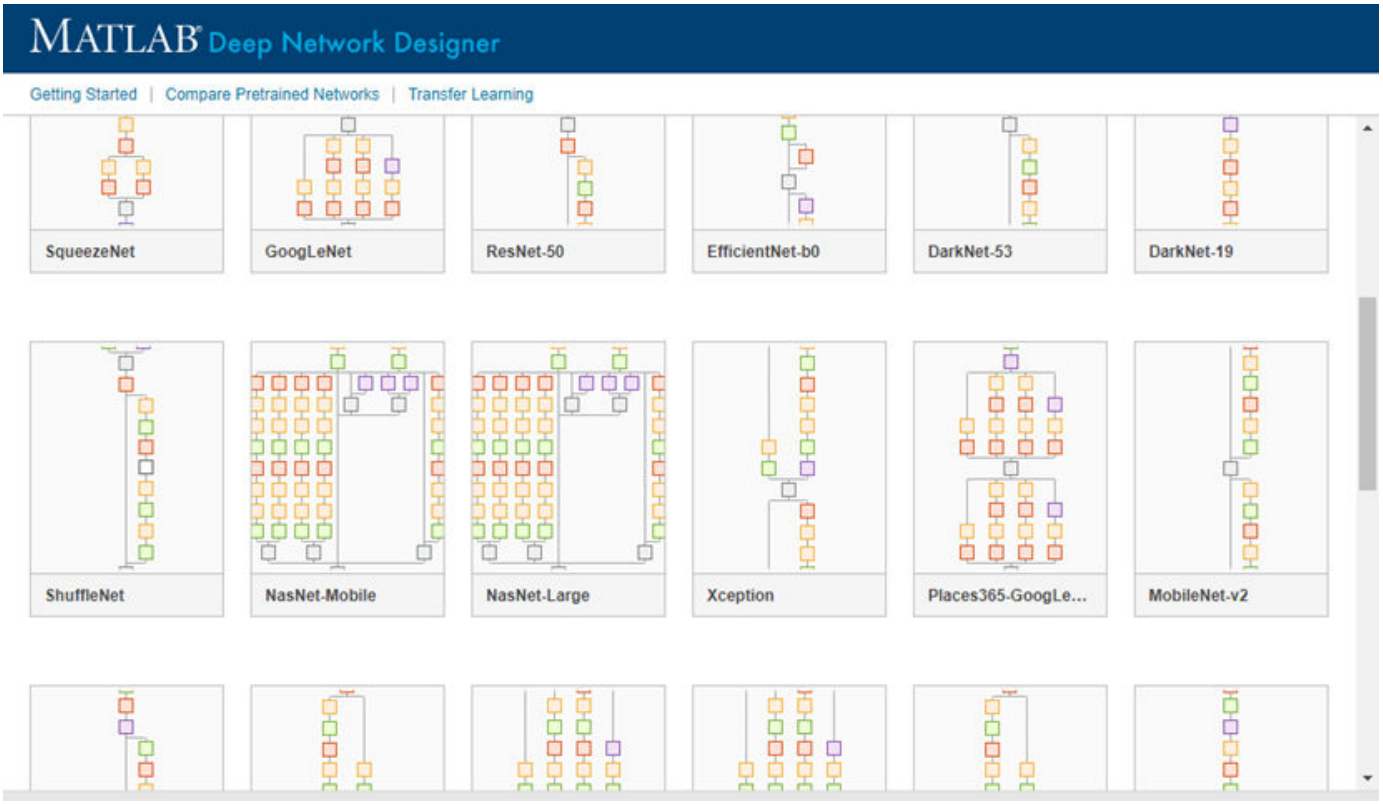
```
shufflenet
```

```
ans =

  DAGNetwork with properties:

         Layers: [173×1 nnet.cnn.layer.Layer]
    Connections: [188×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(shufflenet)
```

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



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

Transfer Learning with ShuffleNet

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the ShuffleNet network instead of GoogLeNet.

```
net = shufflenet
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

Output Arguments

net — Pretrained ShuffleNet convolutional neural network

`DAGNetwork` object

Pretrained ShuffleNet convolutional neural network, returned as a `DAGNetwork` object.

References

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

[2] Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." *arXiv preprint arXiv:1707.01083v2* (2017).

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201` | `nasnetmobile` | `nasnetlarge`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a

sigmoid

Apply sigmoid activation

Syntax

```
dlY = sigmoid(dlX)
```

Description

The sigmoid activation operation applies the sigmoid function to the input data.

This operation is equivalent to

$$f(x) = \frac{1}{1 + e^{-x}}.$$

Note This function applies the sigmoid operation to `dlarray` data. If you want to apply sigmoid within a `layerGraph` object or `Layer` array, use the following layer:

- `sigmoidLayer`

`dlY = sigmoid(dlX)` computes the sigmoid activation of the input `dlX` by applying the sigmoid transfer function. All values in `dlY` are between `0` and `1`.

Examples

Apply Sigmoid Activation

Use the `sigmoid` function to set all values in the input data to a value between `0` and `1`.

Create the input data as a single observation of random values with a height and width of seven and 32 channels.

```
height = 7;
width = 7;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the sigmoid activation.

```
dlY = sigmoid(dlX);
```

All values in `dlY` now range between `0` and `1`.

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Data Types: `single` | `double`

Output Arguments

`dlY` — Sigmoid activations

`dlarray`

Sigmoid activations, returned as a `dlarray`. All values in `dlY` are between `0` and `1`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `dlgradient` | `dlfeval` | `softmax` | `crossentropy` | `huber` | `mse`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“List of Functions with dlarray Support”

Introduced in R2019b

sigmoidLayer

Sigmoid layer

Description

A sigmoid layer applies a sigmoid function to the input such that the output is bounded in the interval (0,1).

Tip To use the sigmoid layer for binary or multilabel classification problems, create a custom binary cross-entropy loss output layer or use a custom training loop.

Creation

Syntax

```
layer = sigmoidLayer
layer = sigmoidLayer('Name',Name)
```

Description

`layer = sigmoidLayer` creates a sigmoid layer.

`layer = sigmoidLayer('Name',Name)` creates a sigmoid layer and sets the optional `Name` property using a name-value pair argument. For example, `sigmoidLayer('Name','sig1')` creates a sigmoid layer with the name `'sig1'`. Enclose the property name in single quotes.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Sigmoid Layer

Create a sigmoid layer with the name `'sig1'`.

```
layer = sigmoidLayer('Name', 'sig1')
```

```
layer =
  SigmoidLayer with properties:

    Name: 'sig1'

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

More About

Sigmoid Layer

A sigmoid layer applies a sigmoid function to the input such that the output is bounded in the interval (0,1).

This operation is equivalent to

$$f(x) = \frac{1}{1 + e^{-x}}.$$

A multilabel classification problem can be thought of as a binary classification problem, where each class is considered independently of other classes as either present or not present. Solving this type of problem requires the sigmoid activation function, where for any sample x_n the posterior probability of class C_k is

$$\mathrm{P}(C_k|x_n) = \frac{1}{1 + e^{-a_k}}.$$

The value a_k is the weighted sum of all the units that are connected to class k. Performing multilabel classification requires a sigmoid layer followed by a custom binary cross-entropy loss layer.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `convolution2dLayer` | `tanhLayer` | `softmaxLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2020b

softmax

Apply softmax activation to channel dimension

Syntax

```
dlY = softmax(dlX)
dlY = softmax(dlX,'DataFormat',FMT)
```

Description

The softmax activation operation applies the softmax function to the channel dimension of the input data.

The softmax function normalizes the value of the input data across the channel dimension such that it sums to one. You can regard the output of the softmax function as a probability distribution.

Note This function applies the softmax operation to `dlarray` data. If you want to apply softmax within a `layerGraph` object or `Layer` array, use the following layer:

- `softmaxLayer`

`dlY = softmax(dlX)` computes the softmax activation of the input `dlX` by applying the softmax transfer function to the channel dimension of the input data. All values in `dlY` are between `0` and `1`, and sum to `1`. The input `dlX` must be a formatted `dlarray`. The output `dlY` is a formatted `dlarray` with the same dimension format as `dlX`.

`dlY = softmax(dlX,'DataFormat',FMT)` also specifies dimension format `FMT` when `dlX` is not a formatted `dlarray`. The output `dlY` is an unformatted `dlarray` with the same dimension order as `dlX`.

Examples

Apply Softmax Activation

Use the `softmax` function to set all values in the input data to values between `0` and `1` that sum to `1` over all channels.

Create the input classification data as two observations of random variables. The data can be in any of 10 categories.

```
numCategories = 10;
observations = 2;

X = rand(numCategories,observations);
dlX = dlarray(X,'CB');
```

Compute the `softmax` activation.

```
dlY = softmax(dlX);
totalProb = sum(dlY,1)
```

```
dlY =

  10(C) x 2(B) dlarray

    0.1151    0.0578
    0.1261    0.1303
    0.0579    0.1285
    0.1270    0.0802
    0.0959    0.1099
    0.0562    0.0569
    0.0673    0.0753
    0.0880    0.1233
    0.1328    0.1090
    0.1337    0.1288
totalProb =

  1(C) x 2(B) dlarray

    1.0000    1.0000
```

All values in `dlY` range between `0` and `1`. The values over all channels sum to `1` for each observation.

Input Arguments

`dlX` — Input data

`dlarray`

Input data, specified as a formatted `dlarray` or an unformatted `dlarray`. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using `'DataFormat',FMT`.

`dlX` must contain a `'C'` channel dimension.

Data Types: `single` | `double`

`FMT` — Dimension order of unformatted data

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string `FMT` that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, and `'T'` at most once.

You must specify `'DataFormat',FMT` when the input data is not a formatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

Output Arguments

`dlY` — Softmax activations

`dlarray`

Softmax activations, returned as a `dlarray`. All values in `dlY` are between `0` and `1`. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted `dlarray`, `dlY` has the same dimension format as `dlX`. If the input data is not a formatted `dlarray`, `dlY` is an unformatted `dlarray` with the same dimension order as the input data.

More About

Softmax Activation

The `softmax` function normalizes the input across the channel dimension, such that it sums to one. For more information, see the definition of “Softmax Layer” on page 1-1309 on the `softmaxLayer` reference page.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `batchnorm` | `fullyconnect` | `relu` | `crossentropy` | `dlgradient` | `dlfeval` | `huber` | `l1loss` | `l2loss`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Make Predictions Using Model Function”
“Train Network with Multiple Outputs”
“List of Functions with dlarray Support”

Introduced in R2019b

softmaxLayer

Softmax layer

Description

A softmax layer applies a softmax function to the input.

Creation

Syntax

```
layer = softmaxLayer
layer = softmaxLayer('Name',Name)
```

Description

`layer = softmaxLayer` creates a softmax layer.

`layer = softmaxLayer('Name',Name)` creates a softmax layer and sets the optional `Name` property using a name-value pair. For example, `softmaxLayer('Name','sm1')` creates a softmax layer with the name `'sm1'`. Enclose the property name in single quotes.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Softmax Layer

Create a softmax layer with the name `'sm1'`.

```
layer = softmaxLayer('Name','sm1')
```

```
layer =
  SoftmaxLayer with properties:

    Name: 'sm1'
```

Include a softmax layer in a `Layer` array.

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

```
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  0
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

More About

Softmax Layer

A softmax layer applies a softmax function to the input.

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

The output unit activation function is the softmax function:

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

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

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

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

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

The softmax function is also known as the *normalized exponential* and can be considered the multi-class generalization of the logistic sigmoid function [1].

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `classificationLayer` | `convolution2dLayer` | `fullyConnectedLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a

sortClasses

Package: `mlearnlib.graphics.chart`

Sort classes of confusion matrix chart

Syntax

```
sortClasses(cm,order)
```

Description

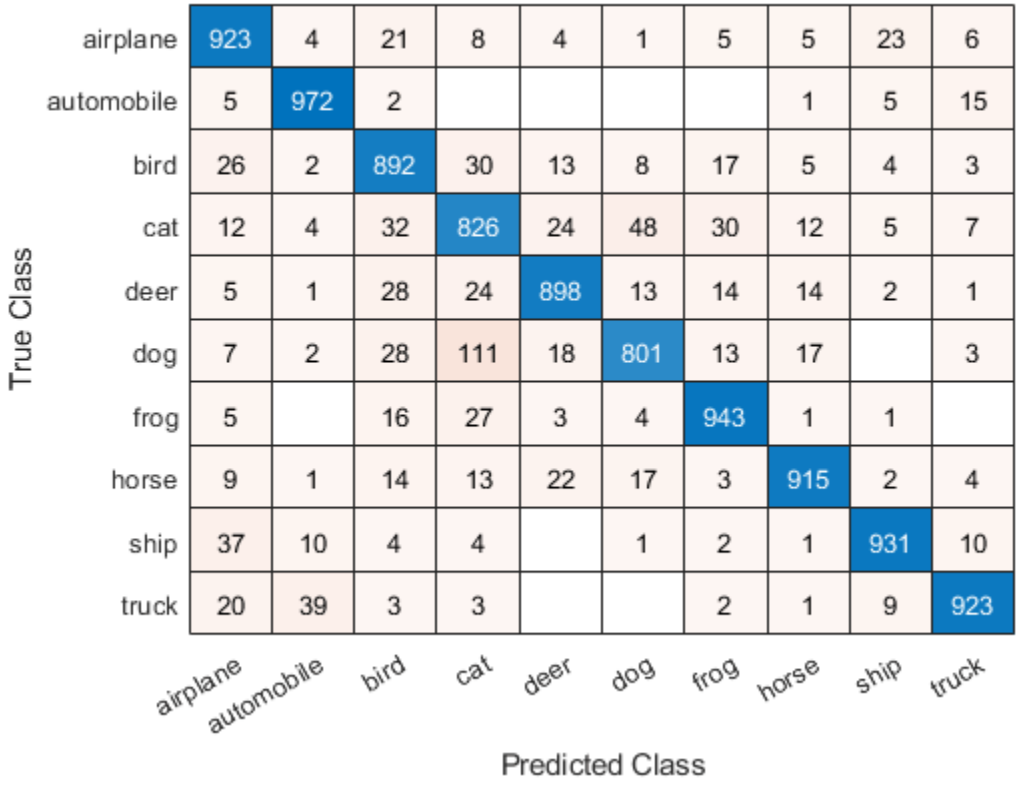
`sortClasses(cm,order)` sorts the classes of the confusion matrix chart `cm` in the order specified by `order`. You can sort the classes in their natural order, by the values along the diagonal of the confusion matrix, or in fixed order that you specify.

Examples

Sort Classes in a Fixed Order

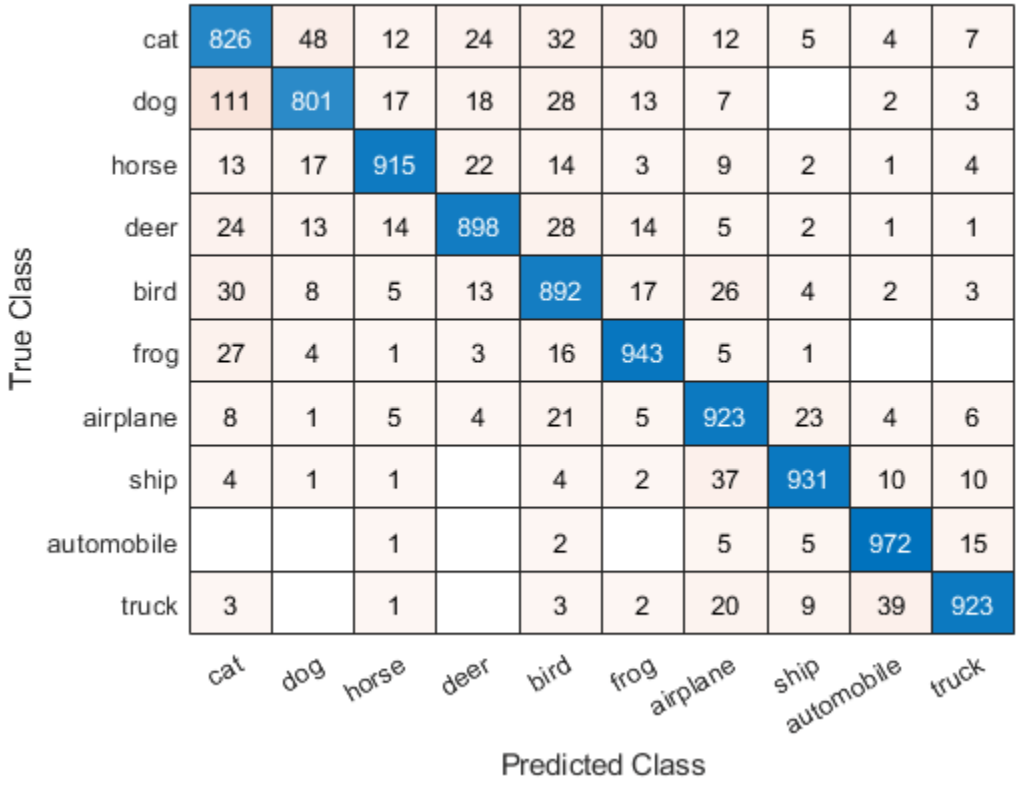
Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network. Create a confusion matrix chart.

```
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
cm = confusionchart(trueLabels,predictedLabels);
```



Reorder the classes of the confusion matrix chart so that the classes are in a fixed order.

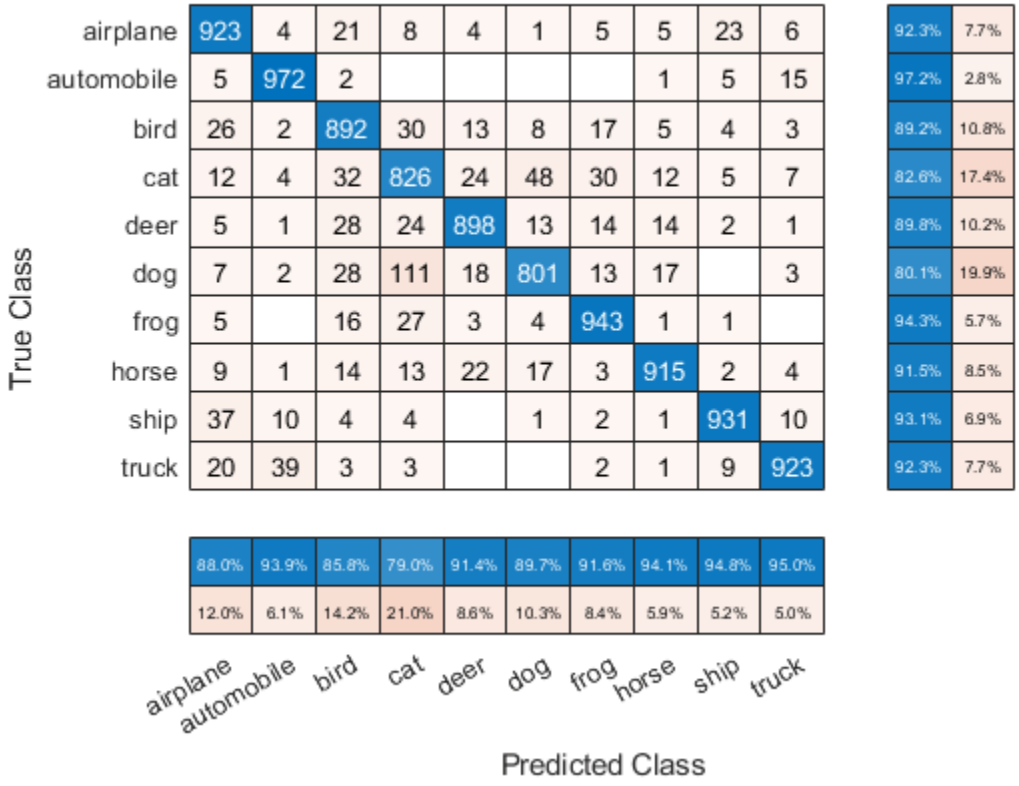
```
sortClasses(cm, ...
    ["cat" "dog" "horse" "deer" "bird" "frog", ...
    "airplane" "ship" "automobile" "truck"])
```



Sort Classes by Precision or Recall

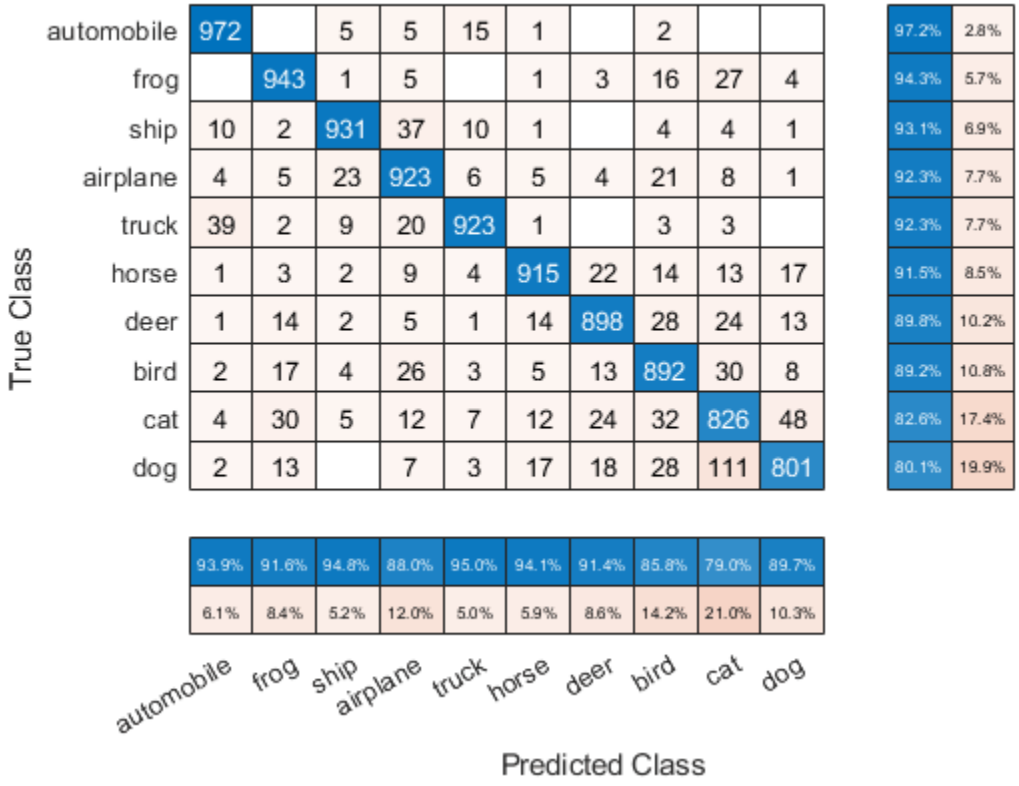
Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network. Create a confusion matrix chart with column and row summaries

```
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
cm = confusionchart(trueLabels,predictedLabels, ...
    'ColumnSummary','column-normalized', ...
    'RowSummary','row-normalized');
```



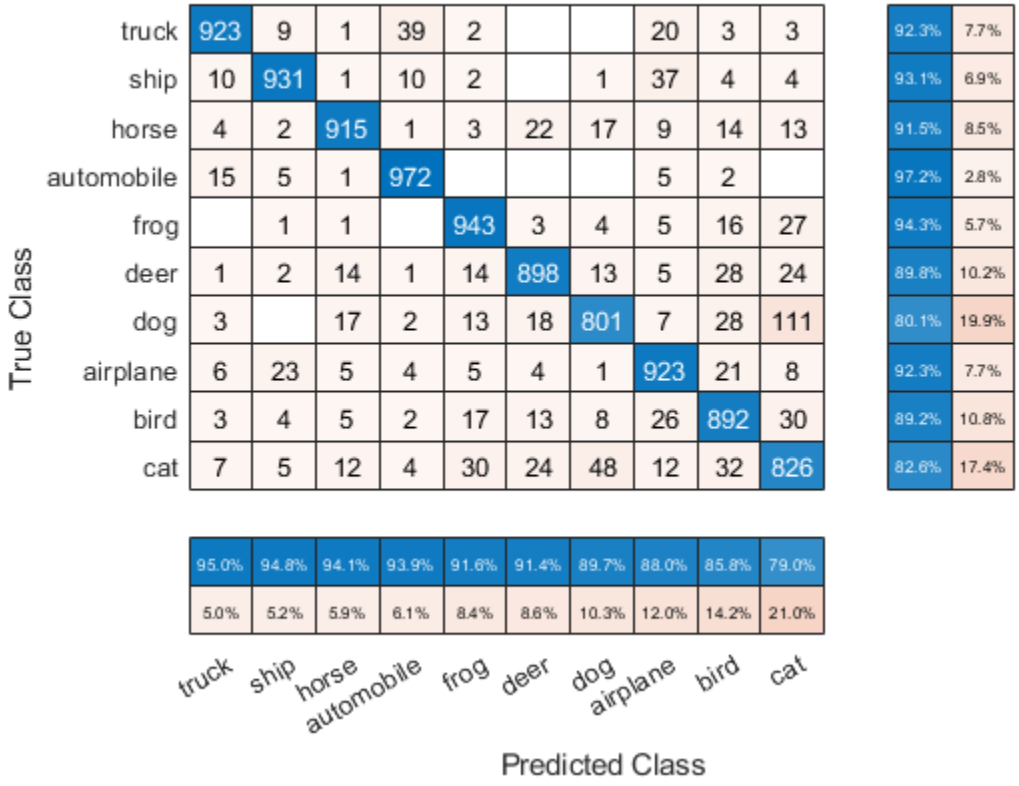
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the row summaries to the right are decreasing.

```
cm.Normalization = 'row-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```



To sort the classes by class-wise precision (positive predictive value), normalize the cell values across each column, that is, by the number of observations that have the same predicted class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the column summaries at the bottom are decreasing.

```
cm.Normalization = 'column-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```



Input Arguments

`cm` — Confusion matrix chart

`ConfusionMatrixChart` object

Confusion matrix chart, specified as a `ConfusionMatrixChart` object. To create a confusion matrix chart, use `confusionchart`,

`order` — Order in which to sort classes

`'auto'` | `'ascending-diagonal'` | `'descending-diagonal'` | array

Order in which to sort the classes of the confusion matrix chart, specified as one of these values:

- `'auto'` — Sorts the classes into their natural order as defined by the `sort` function. For example, if the class labels of the confusion matrix chart are a string vector, then sort alphabetically. If the class labels are an ordinal categorical vector, then use the order of the class labels.
- `'ascending-diagonal'` — Sort the classes so that the values along the diagonal of the confusion matrix increase from top left to bottom right.
- `'descending-diagonal'` — Sort the classes so that the values along the diagonal of the confusion matrix decrease from top left to bottom right.
- `'cluster'` (Requires Statistics and Machine Learning Toolbox) — Sort the classes to cluster similar classes. You can customize clustering by using the `pdist`, `linkage`, and `optimalleaforder` functions. For details, see “Sort Classes to Cluster Similar Classes” (Statistics and Machine Learning Toolbox).

- Array — Sort the classes in a unique order specified by a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. The array must be a permutation of the `ClassLabels` property of the confusion matrix chart.

Example: `sortClasses(cm,'ascending-diagonal')`

Example: `sortClasses(cm,["owl","cat","toad"])`

See Also

Functions
`categorical` | `confusionchart`

Properties
ConfusionMatrixChart Properties

Topics
“Deep Learning in MATLAB”

Introduced in R2018b

squeezenet

SqueezeNet convolutional neural network

Syntax

```
net = squeezenet
net = squeezenet('Weights','imagenet')

lgraph = squeezenet('Weights','none')
```

Description

SqueezeNet is a convolutional neural network that is 18 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. This function returns a SqueezeNet v1.1 network, which has similar accuracy to SqueezeNet v1.0 but requires fewer floating-point operations per prediction [3]. The network has an image input size of 227-by-227. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the SqueezeNet network. For an example, see “Classify Image Using SqueezeNet” on page 1-1333.

You can retrain a SqueezeNet network to perform a new task using transfer learning. For an example, see “Interactive Transfer Learning Using SqueezeNet” on page 1-1319.

`net = squeezenet` returns a SqueezeNet network trained on the ImageNet data set.

`net = squeezenet('Weights','imagenet')` returns a SqueezeNet network trained on the ImageNet data set. This syntax is equivalent to `net = squeezenet`.

`lgraph = squeezenet('Weights','none')` returns the untrained SqueezeNet network architecture.

Examples

Load SqueezeNet Network

Load a pretrained SqueezeNet network.

```
net = squeezenet
```

```
net =

  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
```

This function returns a `DAGNetwork` object.

SqueezeNet is included within Deep Learning Toolbox. To load other networks, use functions such as `googlenet` to get links to download pretrained networks from the Add-On Explorer.

Interactive Transfer Learning Using SqueezeNet

This example shows how to fine-tune a pretrained SqueezeNet network to classify a new collection of images. This process is called transfer learning and is usually much faster and easier than training a new network, because you can apply learned features to a new task using a smaller number of training images. To prepare a network for transfer learning interactively, use Deep Network Designer.

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*).

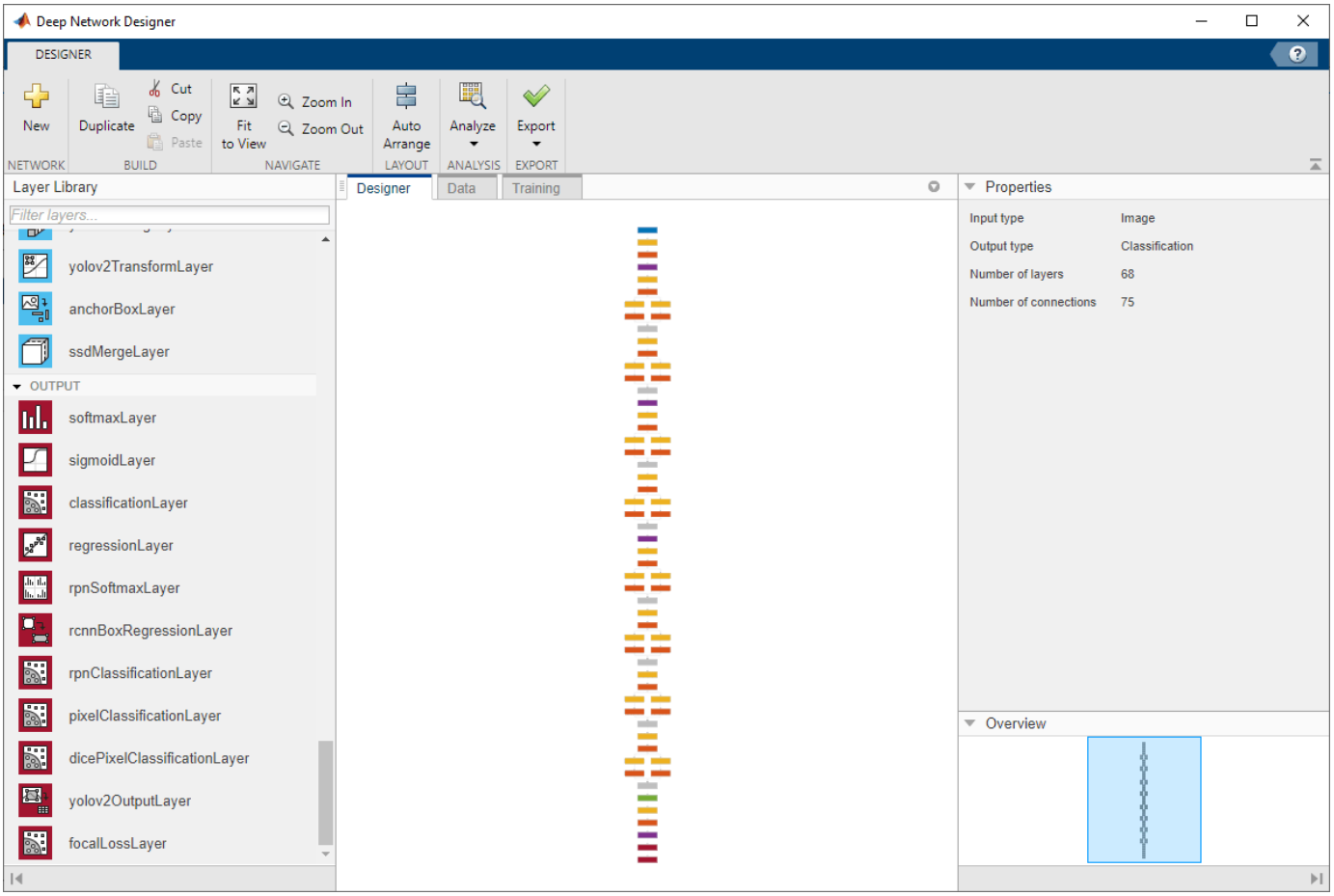
```
unzip("MerchData.zip");
```

Open SqueezeNet in Deep Network Designer

Open Deep Network Designer with SqueezeNet.

```
deepNetworkDesigner(squeezenet);
```

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



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

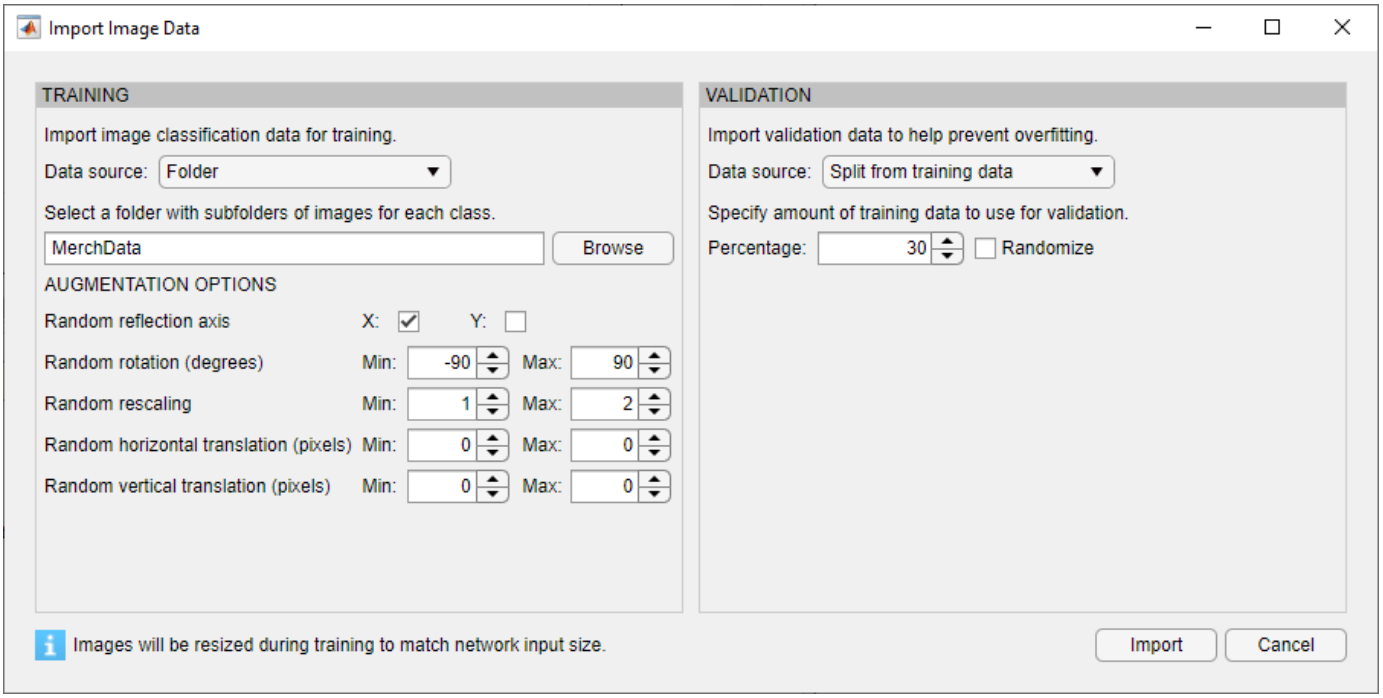
Import Data

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

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

Divide the data into 70% training data and 30% validation data.

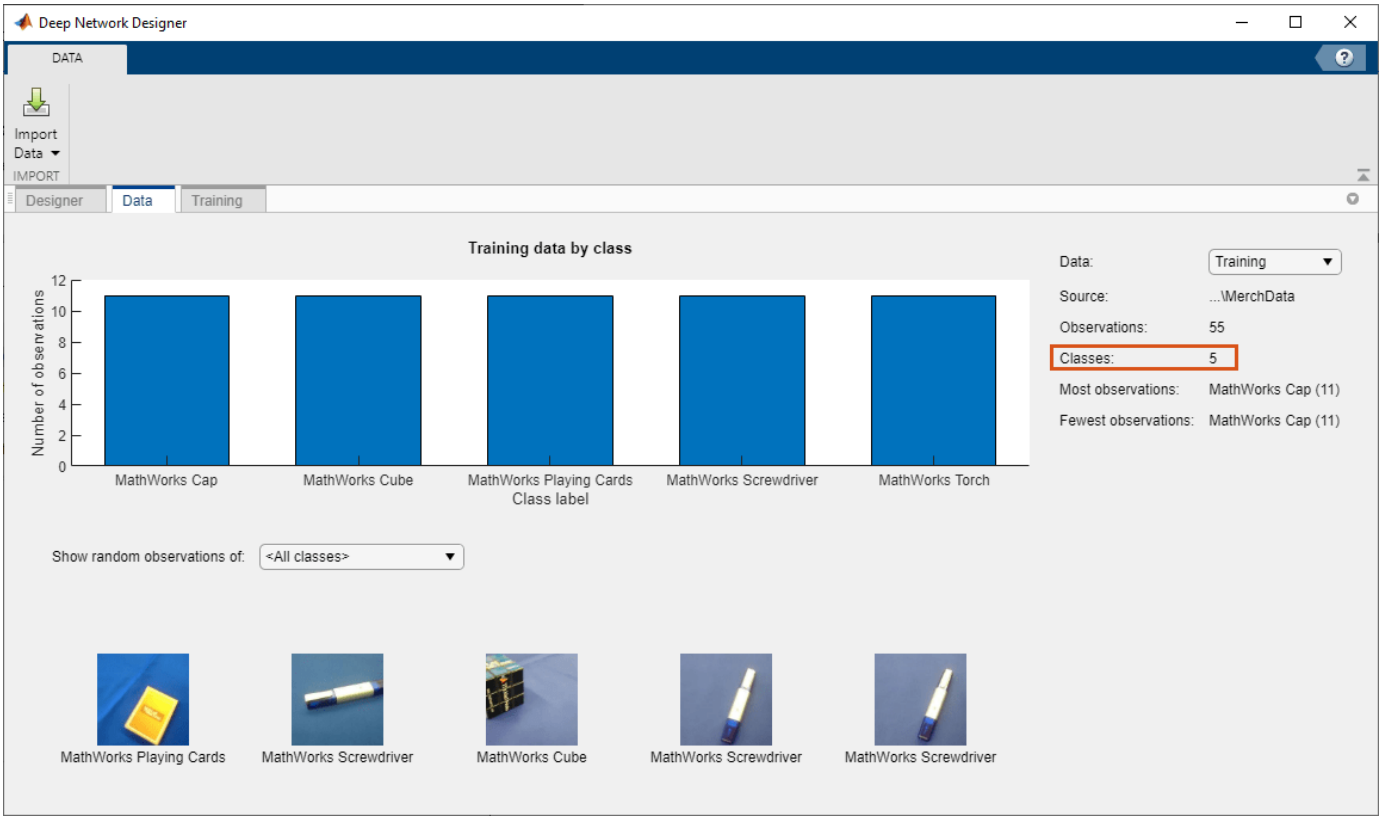
Specify augmentation operations to perform on the training images. For this example, apply a random reflection in the x-axis, a random rotation from the range [-90,90] degrees, and a random rescaling from the range [1,2]. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.



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

Visualize Data

Using Deep Network Designer, you can visually inspect the distribution of the training and validation data in the **Data** tab. You can also view random observations and their labels as a simple check before training. You can see that, in this example, there are five classes in the data set.



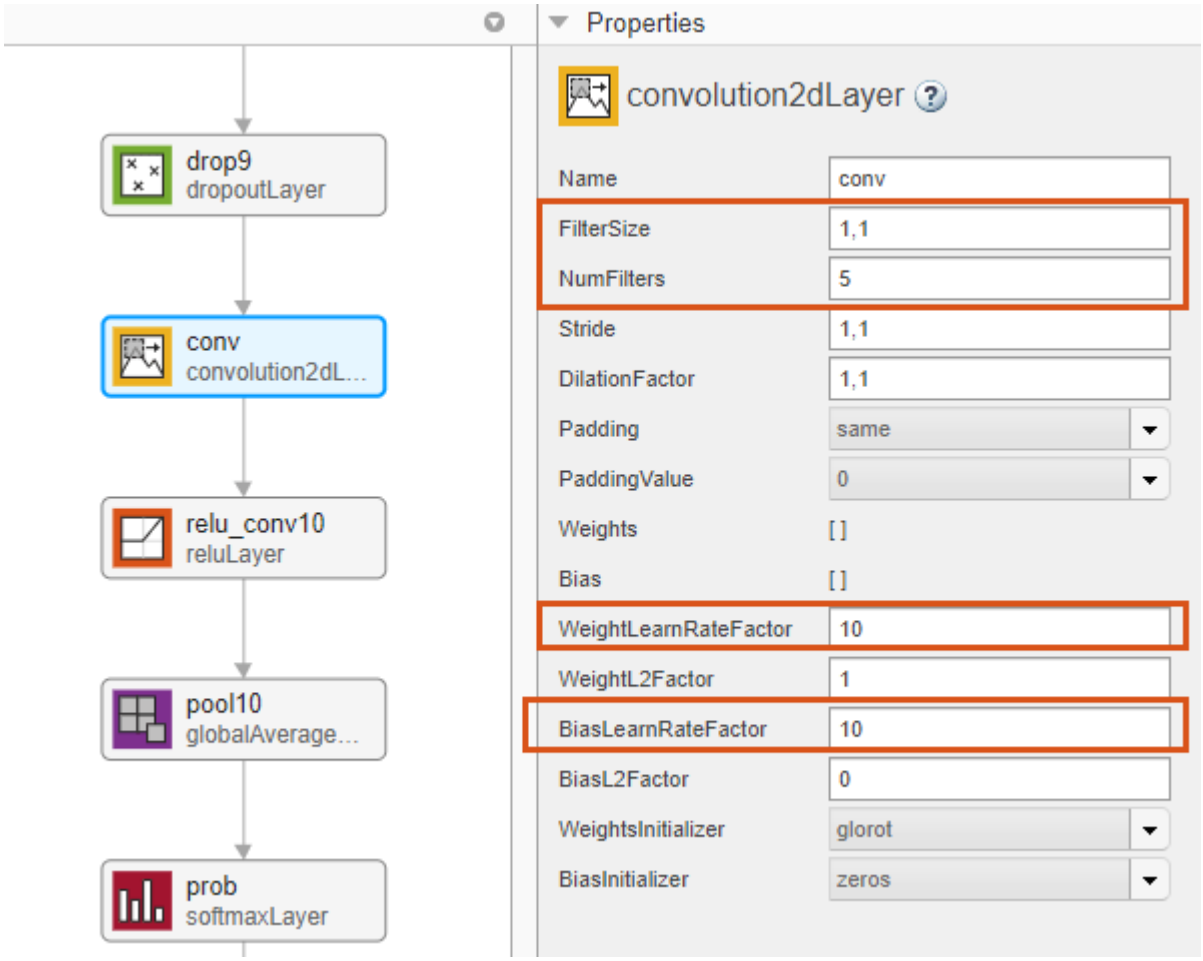
Edit Network for Transfer Learning

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, `'conv10'` and `'ClassificationLayer_predictions'` in SqueezeNet, 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.

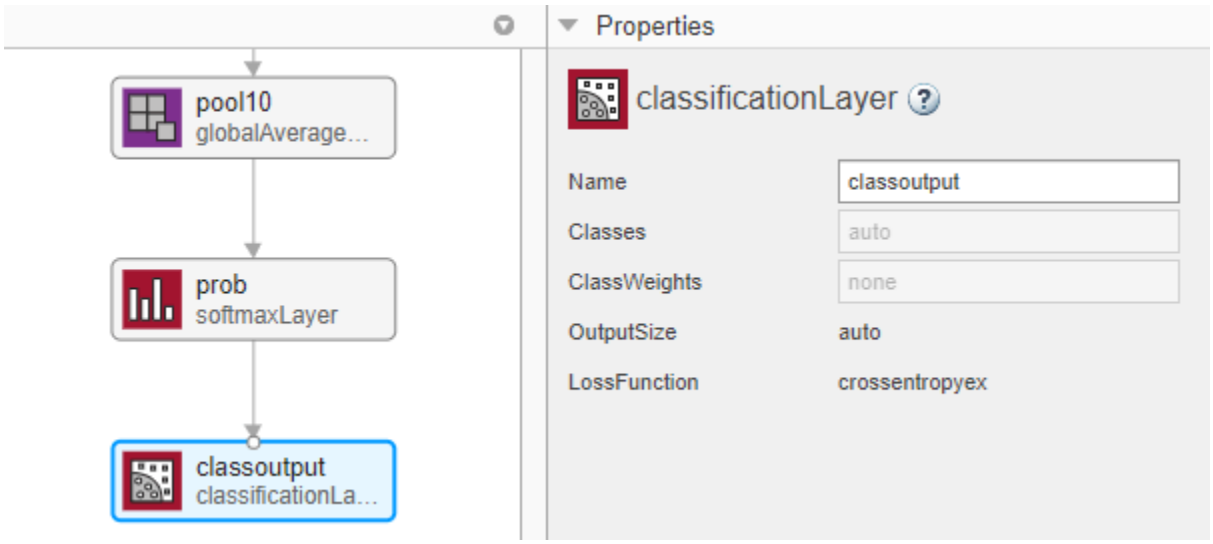
In most networks, the last layer with learnable weights is a fully connected layer. In some networks, such as SqueezeNet, the last learnable layer is the final convolutional layer instead. In this case, replace the convolutional layer with a new convolutional layer with the number of filters equal to the number of classes.

In the **Designer** pane, drag a new `convolution2dLayer` onto the canvas. To match the original convolutional layer, set `FilterSize` to `1,1`. Change `NumFilters` to the number of classes in the new data, in this example, `5`.

Change the learning rates so that learning is faster in the new layer than in the transferred layers by setting `WeightLearnRateFactor` and `BiasLearnRateFactor` to `10`. Delete the last 2-D convolutional layer and connect your new layer instead.

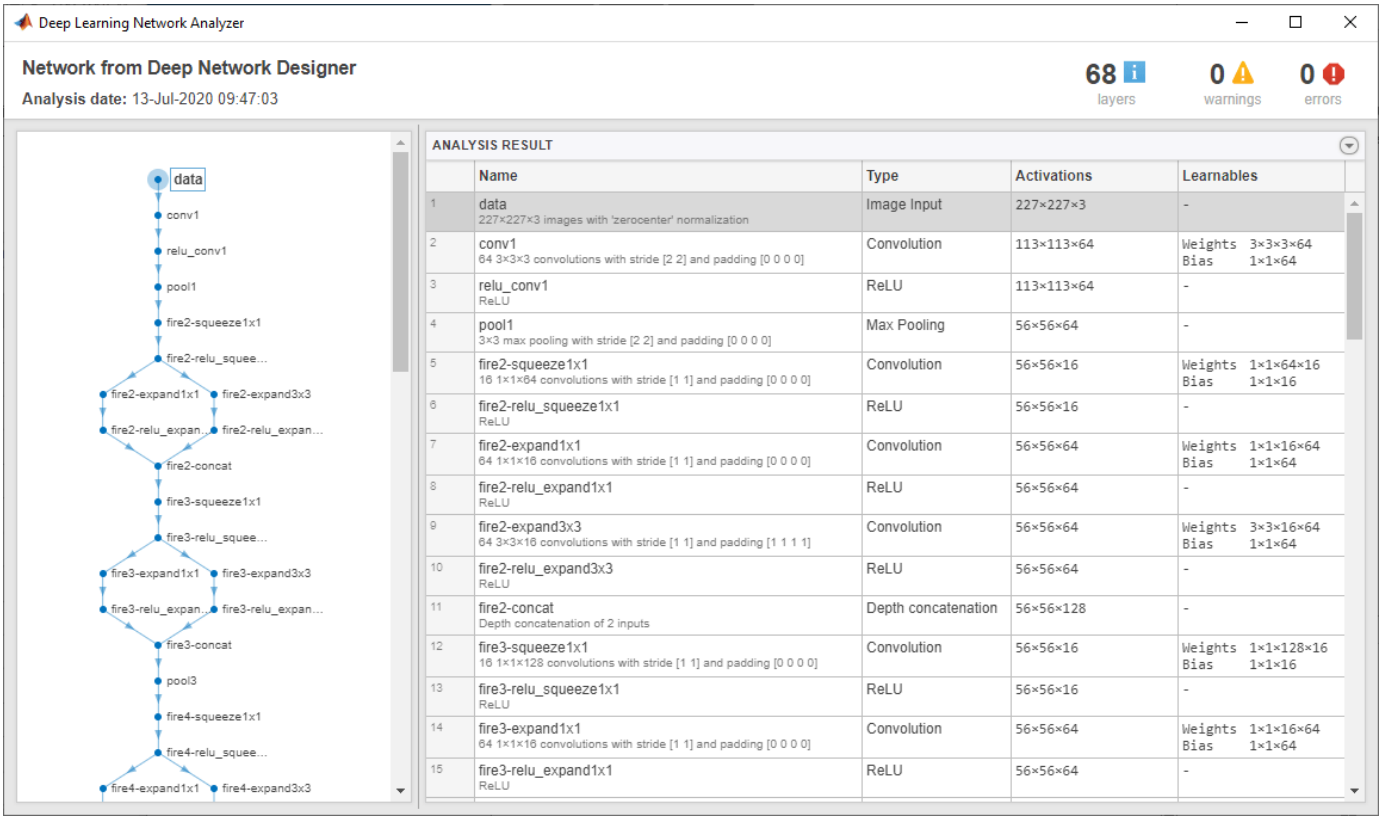


Replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original output layer and connect your new layer instead.



Check Network

To make sure your edited network is ready for training, click **Analyze**, and ensure the Deep Learning Network Analyzer reports zero errors.

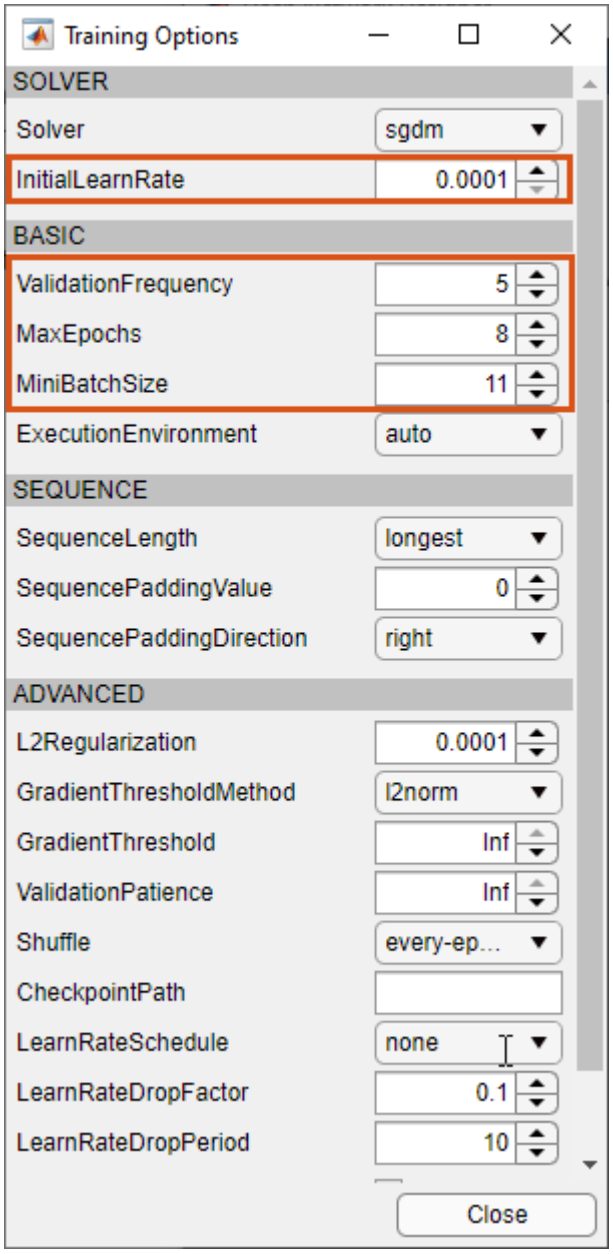


Train Network

Specify training options. Select the **Training** tab and click **Training Options**.

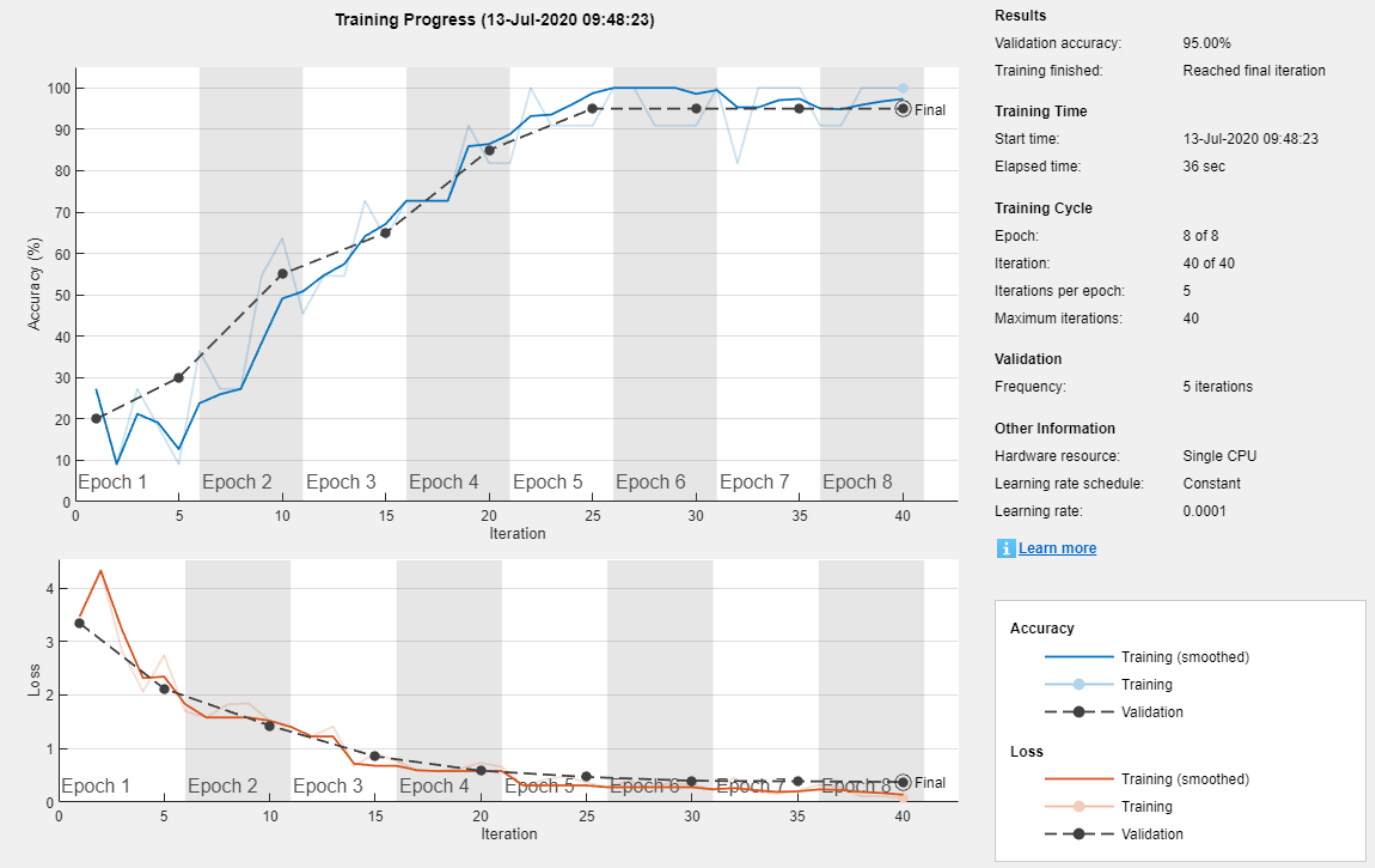
- Set the initial learn rate to a small value to slow down learning in the transferred layers.
- Specify the validation frequency so that the accuracy on the validation data is calculated once every epoch.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data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

For this example, set **InitialLearnRate** to `0.0001`, **ValidationFrequency** to `5`, and **MaxEpochs** to `8`. As there are 55 observations, set **MiniBatchSize** to `11`.



To 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: [3.3420 NaN NaN NaN 2.1187 NaN NaN NaN NaN 1.4291 NaN NaN NaN NaN 0
        ValidationAccuracy: [20 NaN NaN NaN 30 NaN NaN NaN NaN 55.0000 NaN NaN NaN NaN 65 NaN Na
            BaseLearnRate: [1×40 double]
       FinalValidationLoss: 0.3749
   FinalValidationAccuracy: 95
```

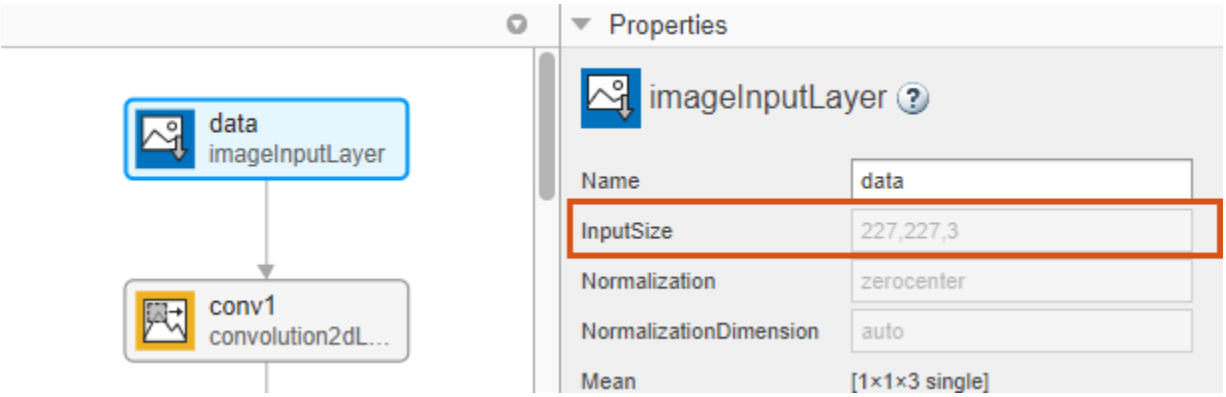
You can also generate MATLAB code, which recreates the network and the training options used. On the **Training** tab, select **Export** > **Generate Code for Training**. Examine the MATLAB code to learn how to programmatically prepare the data for training, create the network architecture, and train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

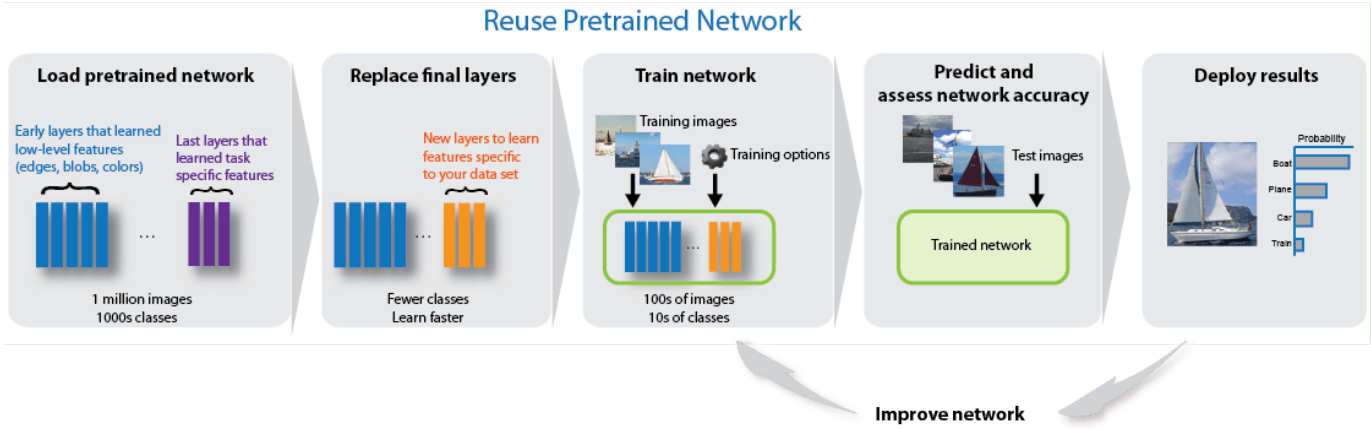
```
[YPred,probs] = classify(trainedNetwork_1,I);
imshow(I)
label = YPred;
title(string(label) + ", " + num2str(100*max(probs),3) + "%");
```

Programmatic Transfer Learning Using SqueezeNet

This example shows how to fine-tune a pretrained SqueezeNet convolutional neural network to perform classification on a new collection of images.

SqueezeNet 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');
```

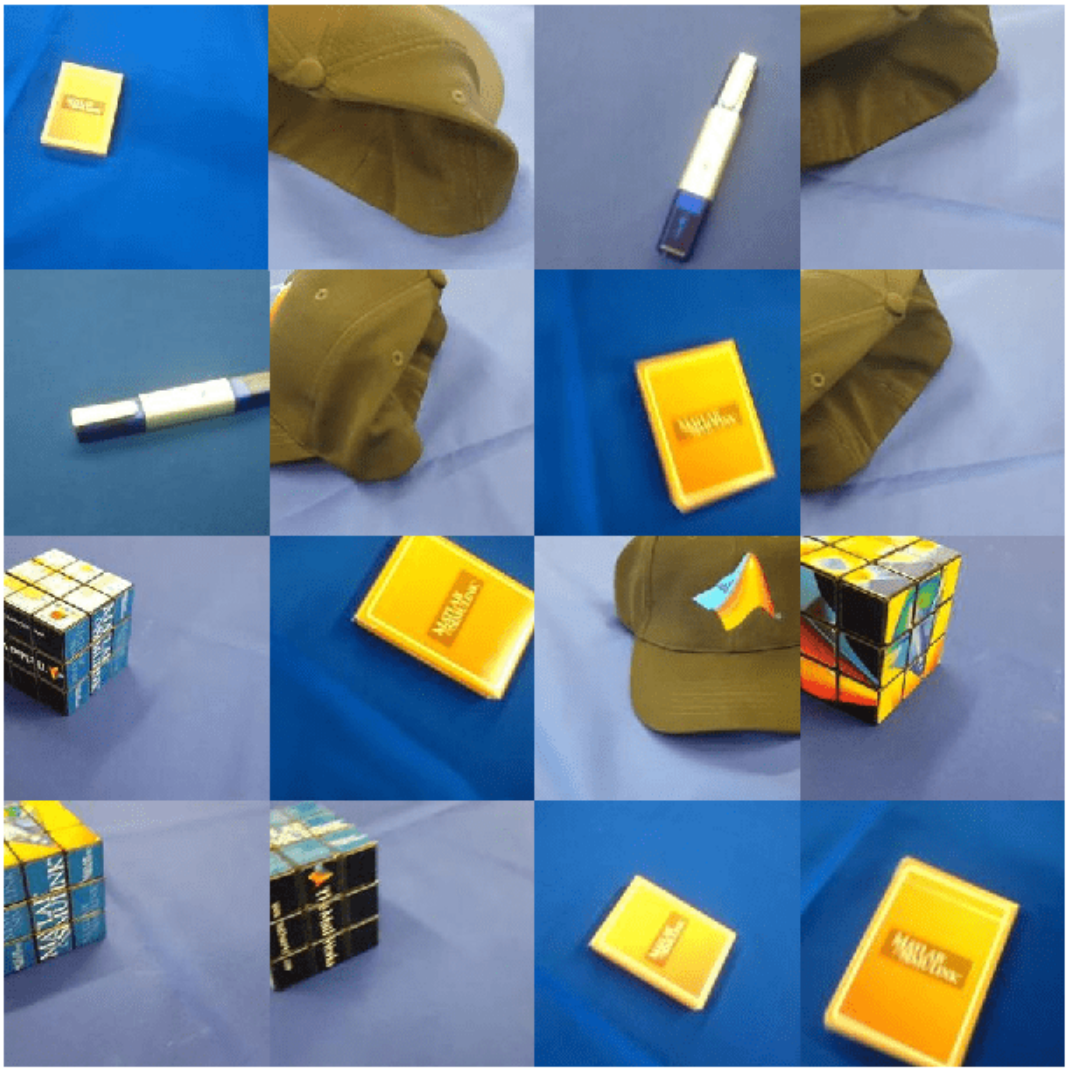
Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[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);

I = imtile(imds, 'Frames', idx);
```

```
figure
imshow(I)
```



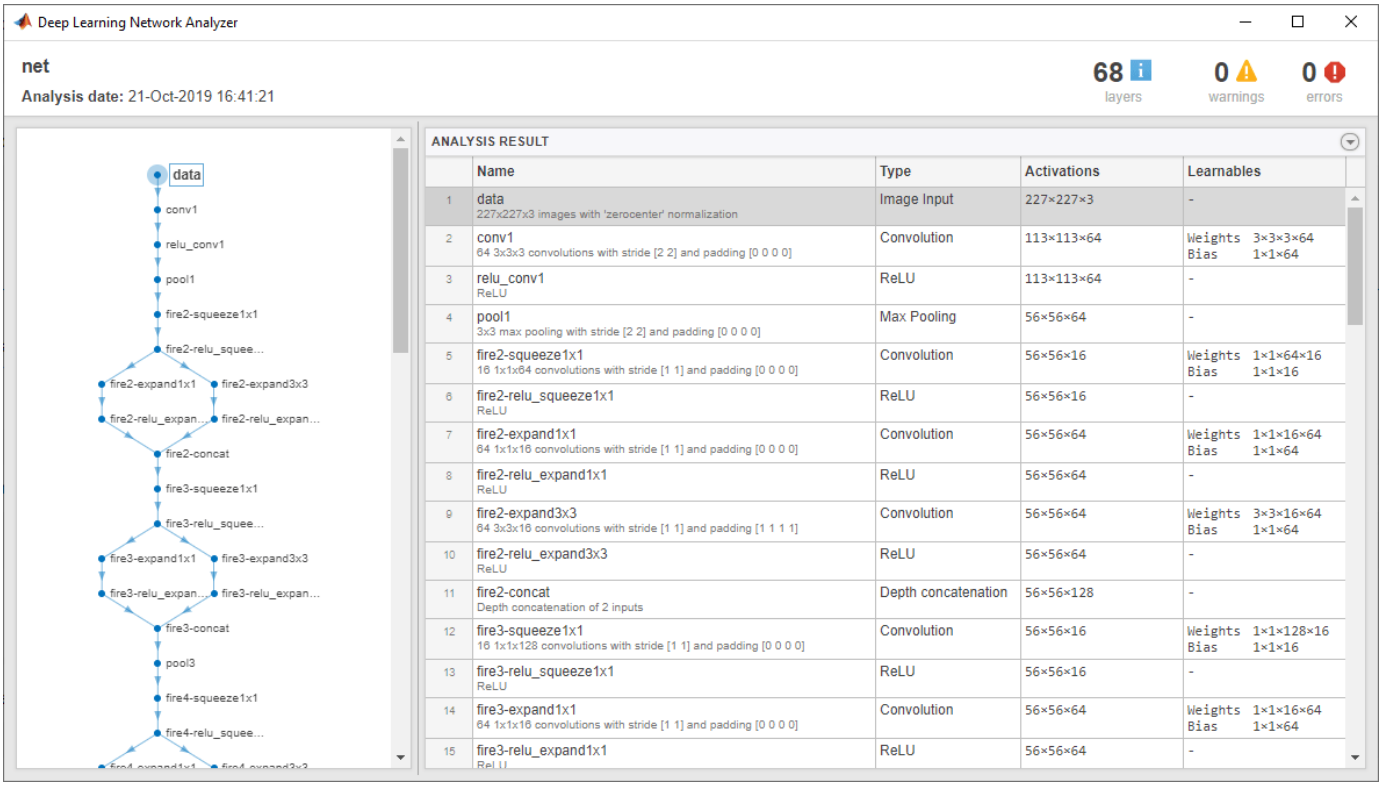
Load Pretrained Network

Load the pretrained SqueezeNet neural network.

```
net = squeezenet;
```

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 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, `'conv10'` and `'ClassificationLayer_predictions'` in SqueezeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Extract the layer graph from the trained network.

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

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   'conv10'                            Convolution             1000 1x1x512 convolutions wi
     2   'ClassificationLayer_predictions'   Classification Output   crossentropyex with 'tench'
```

In most networks, the last layer with learnable weights is a fully connected layer. 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 layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the convolutional layer.

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

```
numClasses = 5
```

```
newConvLayer =  convolution2dLayer([1, 1],numClasses,'WeightLearnRateFactor',10,'BiasLearnRateFa
lgraph = replaceLayer(lgraph,'conv10',newConvLayer);
```

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.

```
newClassificatonLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassificatonLayer);
```

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);
```

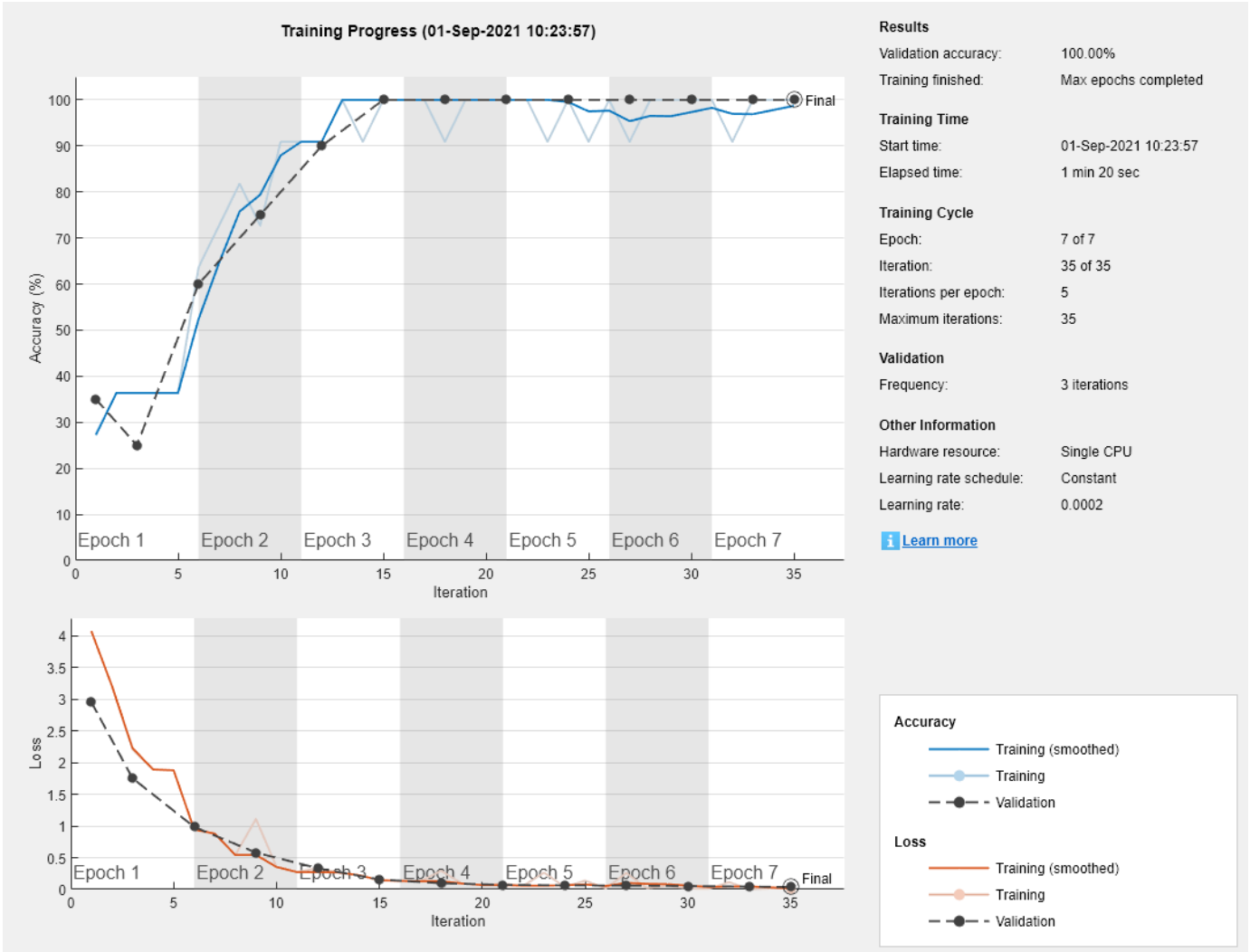
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the convolutional 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 to be 11 so that in each epoch you consider all of the data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',11, ...
```

```
'MaxEpochs',7, ...
'InitialLearnRate',2e-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. This requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



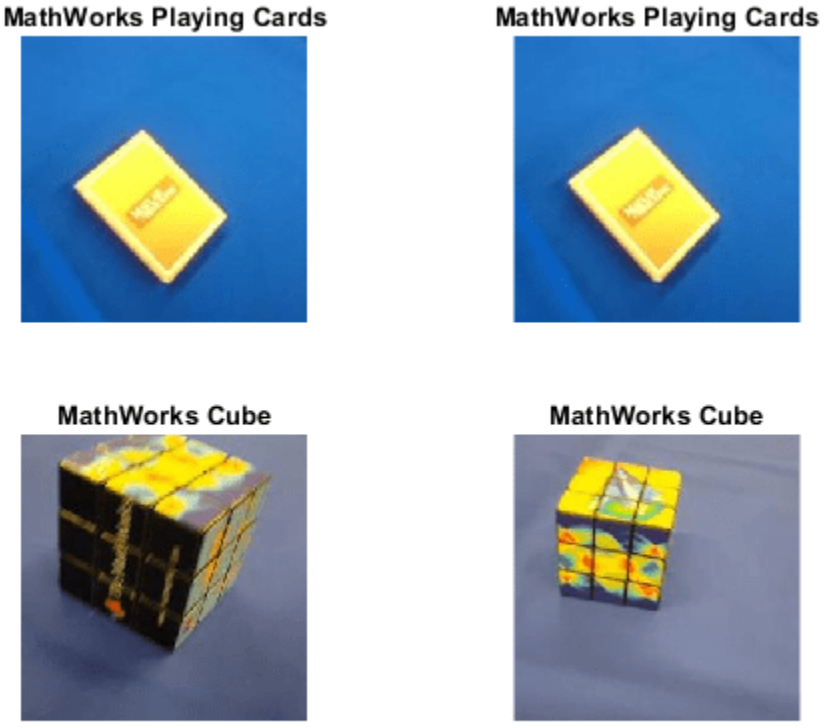
Classify Validation Images

Classify the validation images using the fine-tuned network.

```
[YPred,scores] = classify(netTransfer,augimdsValidation);
```

Display four sample validation images with their predicted labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label));
end
```



Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

```
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 1
```

For tips on improving classification accuracy, see “Deep Learning Tips and Tricks”.

Classify Image Using SqueezeNet

Read, resize, and classify an image using SqueezeNet.

First, load a pretrained SqueezeNet model.

```
net = squeezenet;
```

Read the image using `imread`.

```
I = imread('peppers.png');
figure
imshow(I)
```

The pretrained model requires the image size to be the same as the input size of the network. Determine the input size of the network using the `InputSize` property of the first layer of the network.

```
sz = net.Layers(1).InputSize
```

```
sz = 1×3

   227   227     3
```

Resize the image to the input size of the network.

```
I = imresize(I,sz(1:2));
figure
imshow(I)
```

Classify the image using `classify`.

```
label = classify(net,I)
```

```
label = categorical
     bell pepper
```

Show the image and classification result together.

```
figure
imshow(I)
title(label)
```

Feature Extraction Using SqueezeNet

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

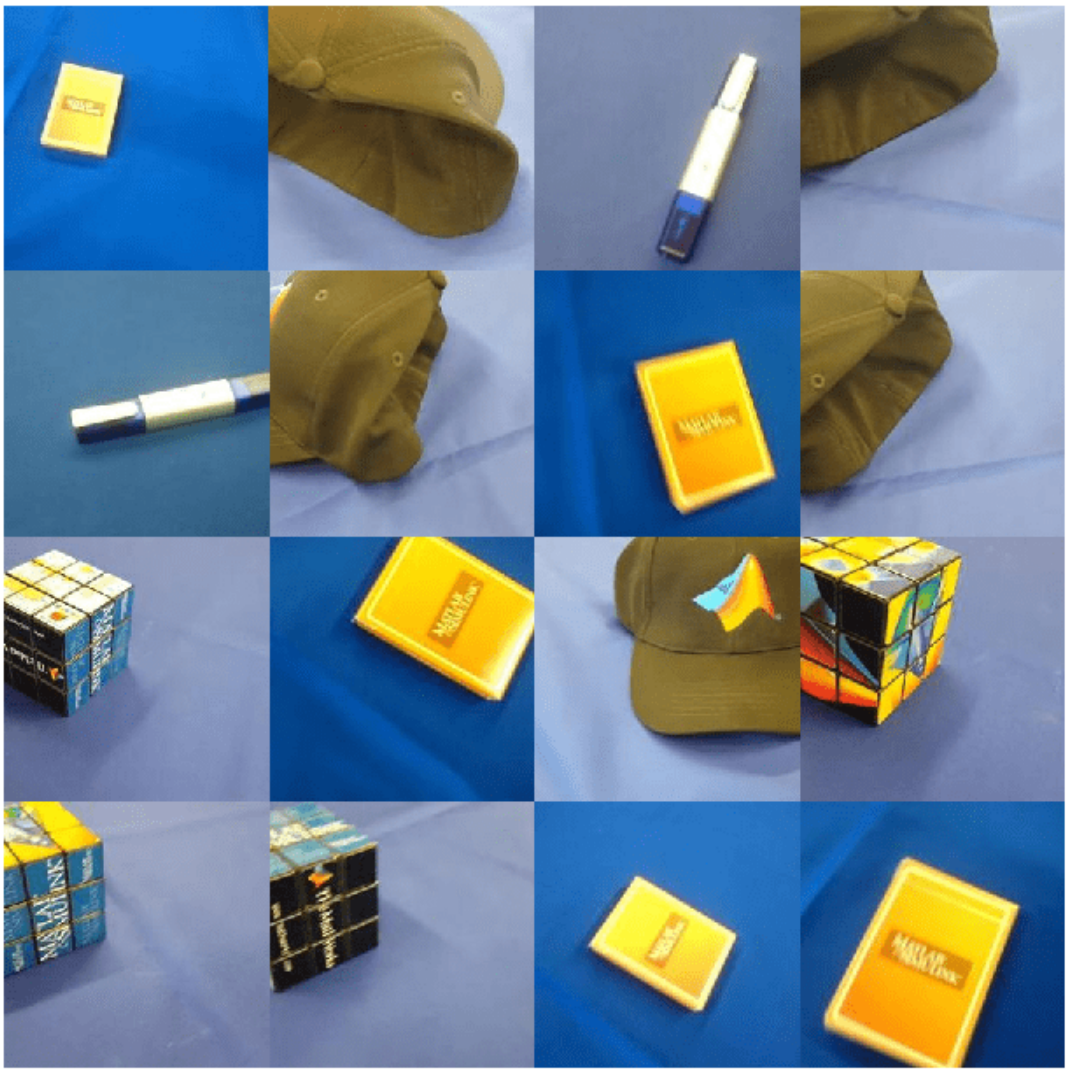
Unzip and load the sample images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore lets you store large image data, including data that does not fit in memory. Split the data into 70% training and 30% test data.

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

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

[imdsTrain,imdsTest] = splitEachLabel(imds,0.7,'randomized');
```

This very small data set now has 55 training images and 20 validation images. Display some sample images.

```
numImagesTrain = numel(imdsTrain.Labels);
idx = randperm(numImagesTrain,16);

I = imtile(imds, 'Frames', idx);
```

```
figure
imshow(I)
```



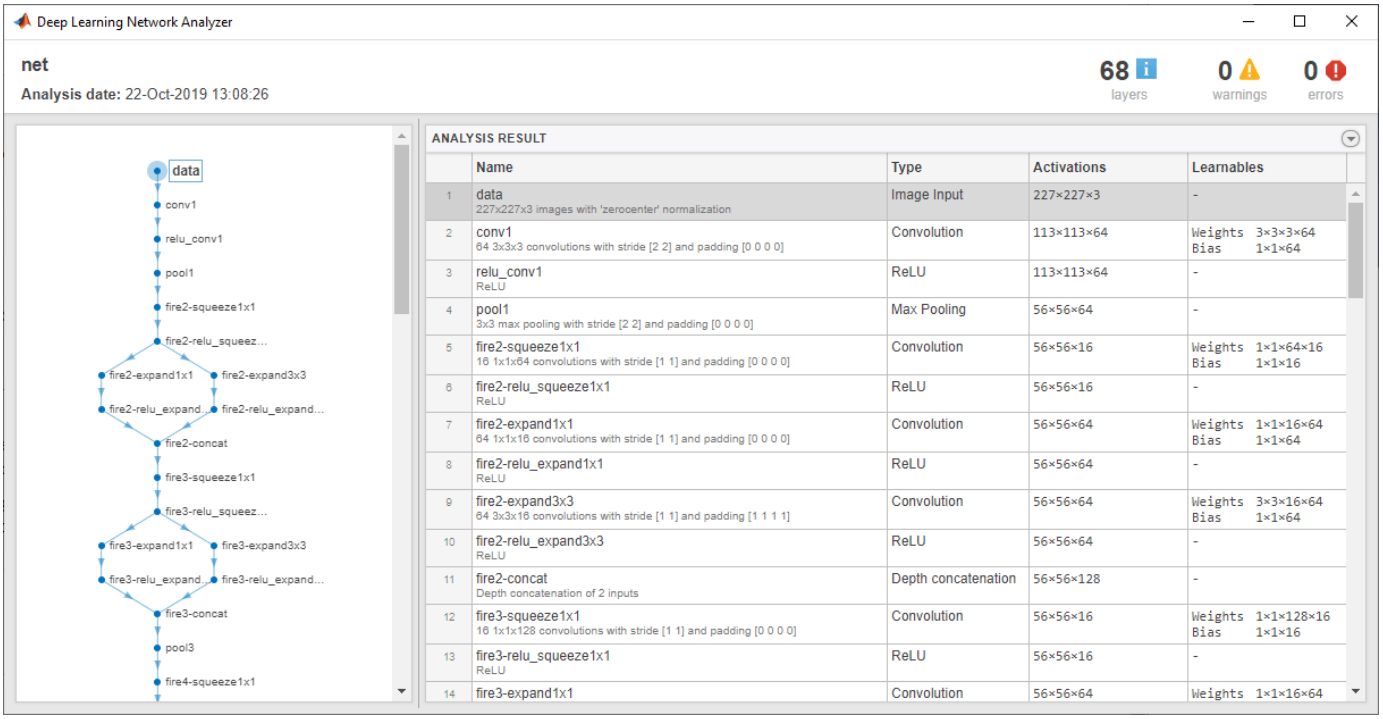
Load Pretrained Network

Load a pretrained SqueezeNet network. SqueezeNet is trained on more than a million images and can classify images into 1000 object categories, for example, keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = squeezenet;
```

Analyze the network architecture.

```
analyzeNetwork(net)
```



	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 64 3x3x3 convolutions with stride [2 2] and padding [0 0 0 0]	Convolution	113×113×64	Weights 3×3×3×64 Bias 1×1×64
3	relu_conv1 ReLU	ReLU	113×113×64	-
4	pool1 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	56×56×64	-
5	fire2-squeeze1x1 16 1x1x64 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	56×56×16	Weights 1×1×64×16 Bias 1×1×16
6	fire2-relu_squeeze1x1 ReLU	ReLU	56×56×16	-
7	fire2-expand1x1 64 1x1x16 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	56×56×64	Weights 1×1×16×64 Bias 1×1×64
8	fire2-relu_expand1x1 ReLU	ReLU	56×56×64	-
9	fire2-expand3x3 64 3x3x16 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56×56×64	Weights 3×3×16×64 Bias 1×1×64
10	fire2-relu_expand3x3 ReLU	ReLU	56×56×64	-
11	fire2-concat Depth concatenation of 2 inputs	Depth concatenation	56×56×128	-
12	fire3-squeeze1x1 16 1x1x128 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	56×56×16	Weights 1×1×128×16 Bias 1×1×16
13	fire3-relu_squeeze1x1 ReLU	ReLU	56×56×16	-
14	fire3-expand1x1	Convolution	56×56×64	Weights 1×1×16×64

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
```

Extract Image Features

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 average pooling layer `'pool10'`. To get a lower level representation of the images, use an earlier layer in the network.

The network requires input images of size 227-by-227-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);

layer = 'pool10';
featuresTrain = activations(net,augimdsTrain,layer,'OutputAs','rows');
featuresTest = activations(net,augimdsTest,layer,'OutputAs','rows');
```

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).

```
mdl = fitcecoc(featuresTrain,YTrain);
```

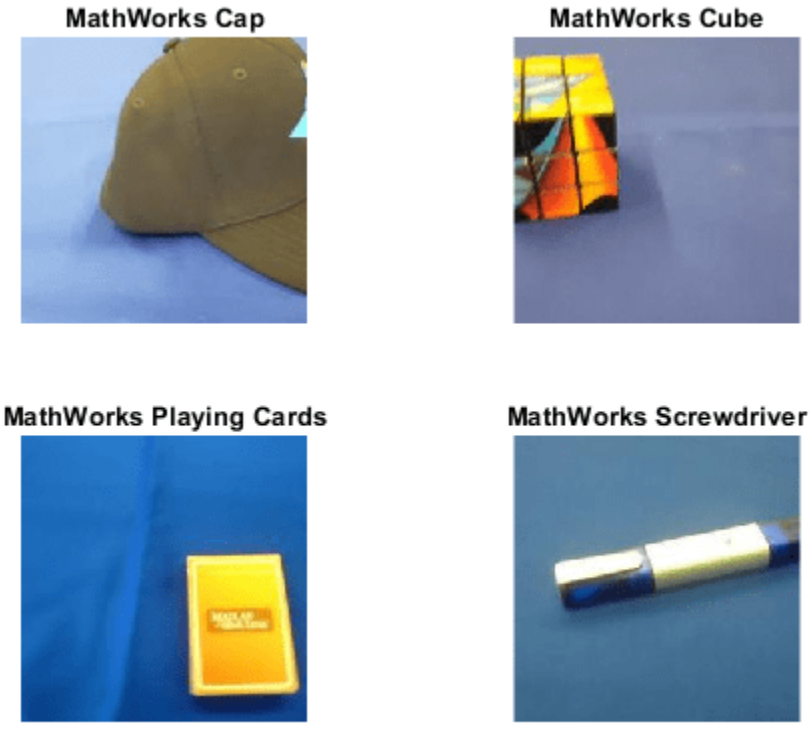
Classify Test Images

Classify the test images using the trained SVM model and the features extracted from the test images.

```
YPred = predict(mdl,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(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
```

This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

Output Arguments

net — Pretrained SqueezeNet convolutional neural network

`DAGNetwork` object

Pretrained SqueezeNet convolutional neural network, returned as a `DAGNetwork` object.

lgraph — Untrained SqueezeNet convolutional neural network architecture

`LayerGraph` object

Untrained SqueezeNet convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Iandola, Forrest N., Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size." Preprint, submitted November 4, 2016. https://arxiv.org/abs/1602.07360.

[3] Iandola, Forrest N. "SqueezeNet." https://github.com/forresti/SqueezeNet.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, load the network by passing the `squeezenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('squeezenet')`.

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `squeezenet('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = squeezenet` or by passing the `squeezenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('squeezenet')`.

For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).

- The syntax `squeezenet('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `resnet18` | `resnet50` | `resnet101` | `inceptionv3` | `inceptionresnetv2` | `densenet201` | `trainNetwork` | `layerGraph` | `DAGNetwork`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a

stripdims

Remove `dlarray` data format

Syntax

```
dlY = stripdims(dlX)
```

Description

`dlY = stripdims(dlX)` returns the `dlarray` `dlX` without any dimension labels. `dlY` is an unformatted `dlarray`.

Examples

Remove Data Format from `dlarray`

Create a formatted `dlarray`.

```
dlX = dlarray(randn(3,2,1,2),'SSTU')
```

```
dlX =
  3(S) x 2(S) x 1(T) x 2(U) dlarray


(:,:,1,1) =

    0.5377    0.8622
    1.8339    0.3188
   -2.2588   -1.3077


(:,:,1,2) =

   -0.4336    2.7694
    0.3426   -1.3499
    3.5784    3.0349
```

Create an array that is the same as `dlX` but has no dimension labels.

```
y = stripdims(dlX)
```

```
y =
  3x2x1x2 dlarray


(:,:,1,1) =

    0.5377    0.8622
    1.8339    0.3188
   -2.2588   -1.3077
```

```
(:,:,1,2) =

   -0.4336    2.7694
    0.3426   -1.3499
    3.5784    3.0349
```

Input Arguments

`dlX` — Input `dlarray`

`dlarray` object

Input `dlarray`, specified as a `dlarray` object.

Example: `dlX = dlarray(randn(3,4),'ST')`

Output Arguments

`dlY` — Unformatted `dlarray`

unformatted `dlarray` object

Unformatted `dlarray`, returned as an unformatted `dlarray` object that is the same as the input array `dlX`, but without any dimension labels. If `dlX` is unformatted, then `dlY = dlX`.

Tips

- Use `stripdims` to ensure that a `dlarray` behaves like a numeric array of the same size, without any special behavior due to dimension labels.
- `ndims(dlX)` can decrease after a `stripdims` call because the function removes trailing singleton dimensions.

```
dlX = dlarray(ones(3,2), 'SCB');
ndims(dlX)

ans =

     3

dlX = stripdims(dlX);
ndims(dlX)

ans =

     2
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`dims` | `finddim` | `dlarray`

Introduced in R2019b

swishLayer

Swish layer

Description

A swish activation layer applies the swish function on the layer inputs.

The swish operation is given by $f(x) = \frac{x}{1 + e^{-x}}$.

Creation

Syntax

```
layer = swishLayer
layer = swishLayer('Name',Name)
```

Description

`layer = swishLayer` creates a swish layer.

`layer = swishLayer('Name',Name)` creates a swish layer and sets the optional `Name` property using a name-value argument. For example, `swishLayer('Name','swish1')` creates a swish layer with the name `'swish1'`.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Swish Layer

Create a swish layer with the name `'swish1'`.

```
layer = swishLayer('Name','swish1')
```

```
layer =
  SwishLayer with properties:

    Name: 'swish1'

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Include a swish layer in a `Layer` array.

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

```
layers =
  8x1 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  0
3   ''   Batch Normalization     Batch normalization
4   ''   Swish                   Swish
5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
6   ''   Fully Connected         10 fully connected layer
7   ''   Softmax                 softmax
8   ''   Classification Output   crossentropyex
```

More About

Swish Layer

A swish activation layer applies the swish function on the layer inputs. The swish operation is given by $f(x) = \frac{x}{1 + e^{-x}}$. The swish layer does not change the size of its input.

Activation layers such as swish layers improve the training accuracy for some applications and usually follow convolution and normalization layers. Other nonlinear activation layers perform different operations. For a list of activation layers, see “Activation Layers”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
`reluLayer` | `trainNetwork` | `batchNormalizationLayer` | `leakyReluLayer` | `clippedReluLayer`

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2021a

tanhLayer

Hyperbolic tangent (tanh) layer

Description

A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.

Creation

Syntax

```
layer = tanhLayer
layer = tanhLayer('Name',Name)
```

Description

`layer = tanhLayer` creates a hyperbolic tangent layer.

`layer = tanhLayer('Name',Name)` additionally specifies the optional `Name` property. For example, `tanhLayer('Name','tanh1')` creates a tanh layer with the name `'tanh1'`.

Properties

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Hyperbolic Tangent Layer

Create a hyperbolic tangent (tanh) layer with the name `'tanh1'`.

```
layer = tanhLayer('Name','tanh1')
```

```
layer =
  TanhLayer with properties:

    Name: 'tanh1'

   Learnable Parameters
    No properties.

   State Parameters
    No properties.

  Show all properties
```

Include a tanh layer in a `Layer` array.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    tanhLayer

    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    tanhLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  11x1 Layer array with layers:

     1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [0  0  0
     3   ''   Batch Normalization     Batch normalization
     4   ''   Tanh                    Hyperbolic tangent
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [0  0  0
     7   ''   Batch Normalization     Batch normalization
     8   ''   Tanh                    Hyperbolic tangent
     9   ''   Fully Connected         10 fully connected layer
    10   ''   Softmax                 softmax
    11   ''   Classification Output   crossentropyex
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`trainNetwork` | `batchNormalizationLayer` | `leakyReluLayer` | `clippedReluLayer` | `reluLayer` | `swishLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Activation Layers”
“List of Deep Learning Layers”

Introduced in R2019a

trainingOptions

Options for training deep learning neural network

Syntax

```
options = trainingOptions(solverName)
options = trainingOptions(solverName,Name,Value)
```

Description

`options = trainingOptions(solverName)` returns training options for the optimizer specified by `solverName`. To train a network, use the training options as an input argument to the `trainNetwork` function.

`options = trainingOptions(solverName,Name,Value)` returns training options with additional options specified by one or more name-value pair arguments.

Examples

Specify Training Options

Create a set of options for training a network using stochastic gradient descent with momentum. Reduce the learning rate by a factor of 0.2 every 5 epochs. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Turn on the training progress plot.

```
options = trainingOptions('sgdm', ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.2, ...
    'LearnRateDropPeriod',5, ...
    'MaxEpochs',20, ...
    'MiniBatchSize',64, ...
    'Plots','training-progress')
```

```
options =
  TrainingOptionsSGDM with properties:

                      Momentum: 0.9000
                 InitialLearnRate: 0.0100
                LearnRateSchedule: 'piecewise'
              LearnRateDropFactor: 0.2000
              LearnRateDropPeriod: 5
                 L2Regularization: 1.0000e-04
          GradientThresholdMethod: 'l2norm'
                GradientThreshold: Inf
                        MaxEpochs: 20
                    MiniBatchSize: 64
                          Verbose: 1
                  VerboseFrequency: 50
                    ValidationData: []
```

```
          ValidationFrequency: 50
           ValidationPatience: Inf
                      Shuffle: 'once'
               CheckpointPath: ''
          ExecutionEnvironment: 'auto'
                    WorkerLoad: []
                     OutputFcn: []
                         Plots: 'training-progress'
               SequenceLength: 'longest'
          SequencePaddingValue: 0
      SequencePaddingDirection: 'right'
          DispatchInBackground: 0
       ResetInputNormalization: 1
  BatchNormalizationStatistics: 'population'
                 OutputNetwork: 'last-iteration'
```

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”.

For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

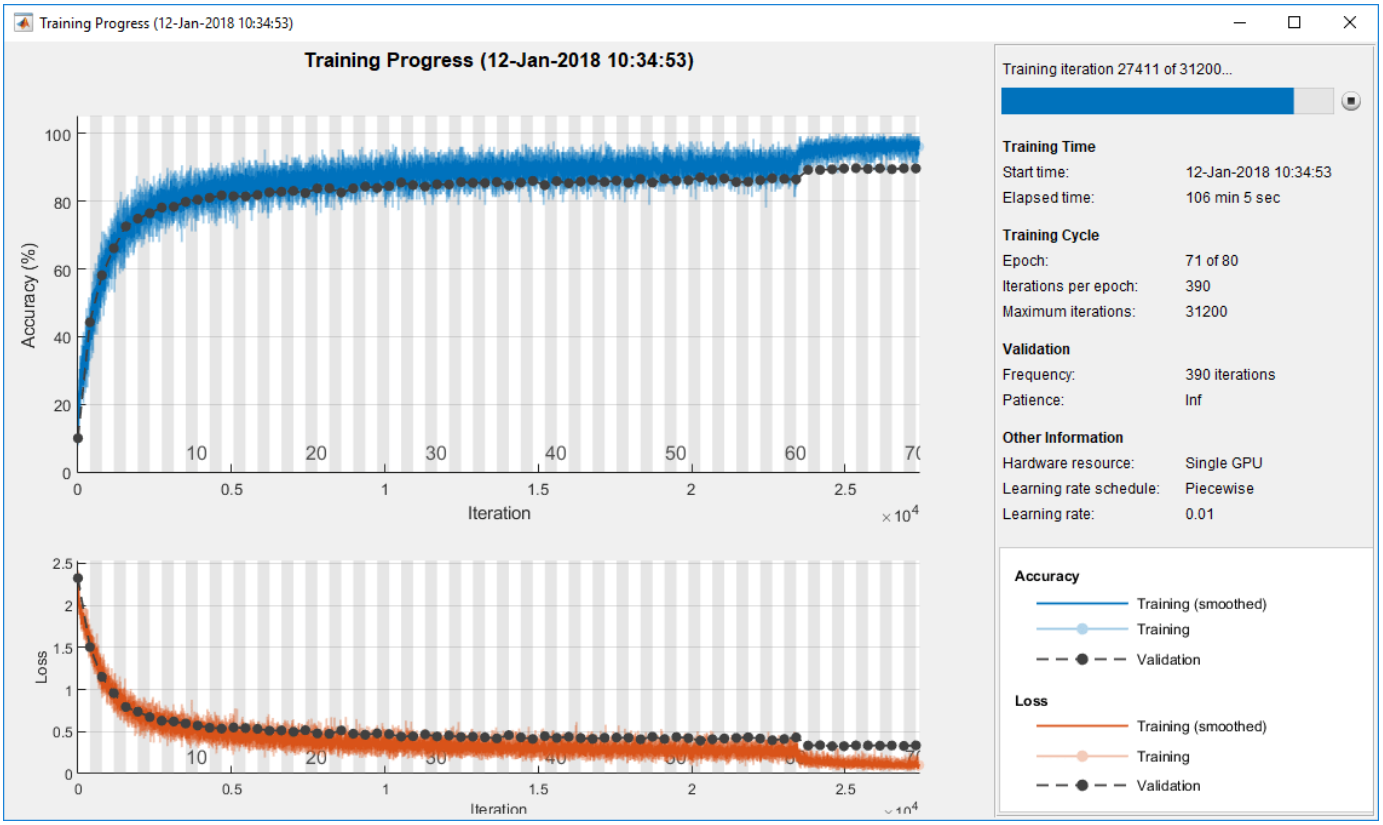
The figure marks each training **Epoch** using a shaded background. An epoch is a full pass through the entire data set.

During training, you can stop training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving. After you click the stop button, it can take a while for the training to complete. Once training is complete, `trainNetwork` returns the trained network.

When training finishes, view the **Results** showing the finalized validation accuracy and the reason that training finished. If the `'OutputNetwork'` training option is set to `'last-iteration'` (default), the finalized metrics correspond to the last training iteration. If the `'OutputNetwork'` training option is set to `'best-validation-loss'`, the finalized metrics correspond to the iteration with the lowest validation loss. The iteration from which the final validation metrics are calculated is labeled **Final** in the plots.

If your network contains batch normalization layers, then the final validation metrics can be different to the validation metrics evaluated during training. This is because the mean and variance statistics used for batch normalization can be different after training completes. For example, if the `'BatchNormalizationStatisics'` training option is `'population'`, then after training, the software finalizes the batch normalization statistics by passing through the training data once more and uses the resulting mean and variance. If the `'BatchNormalizationStatisics'` training option is `'moving'`, then the software approximates the statistics during training using a running estimate and uses the latest values of the statistics.

On the right, view information about the training time and settings. To learn more about training options, see “Set Up Parameters and Train Convolutional Neural Network”.



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.

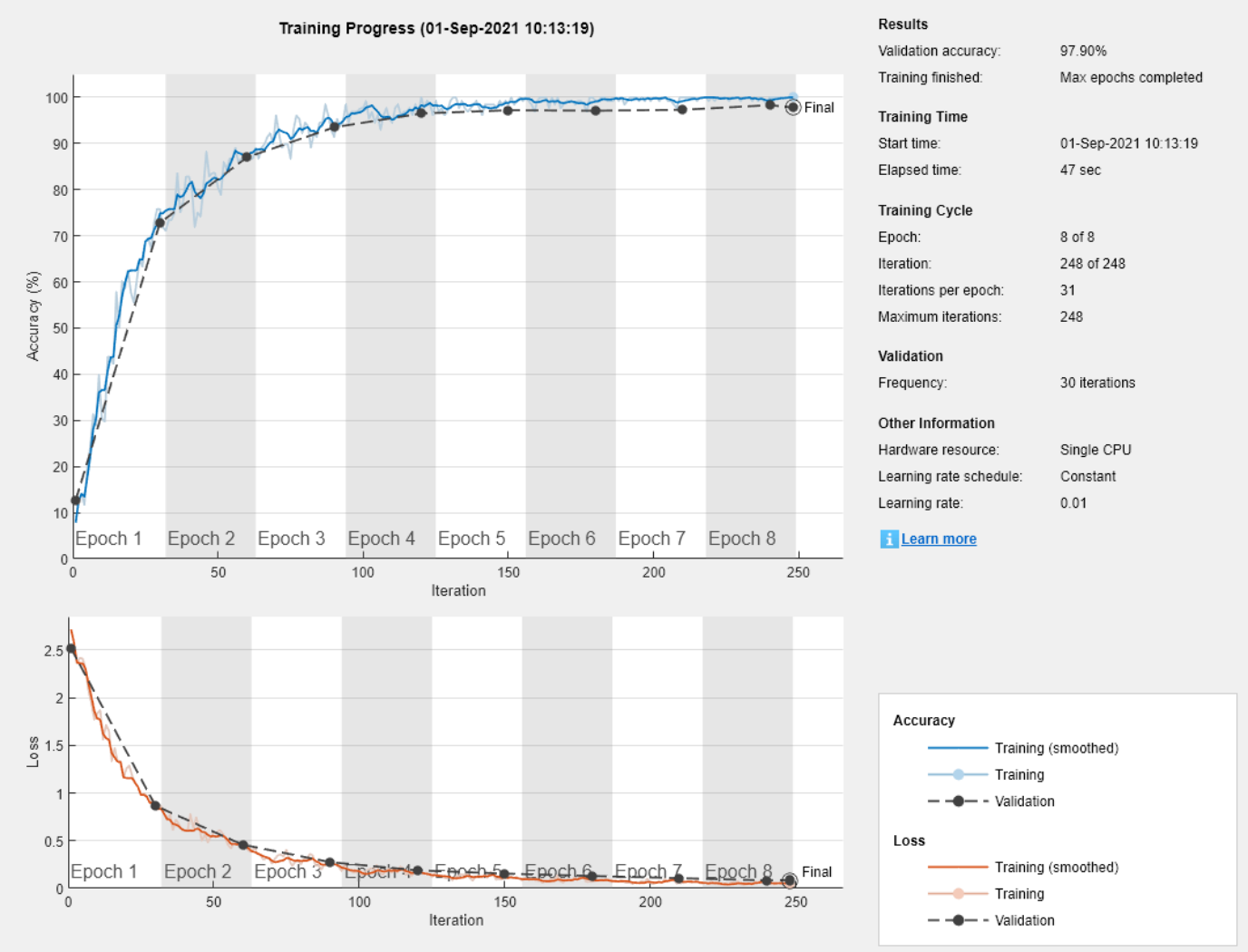
```
layers = [
    imageInputLayer([28 28 1])

    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);
```



Input Arguments

`solverName` — Solver for training network

`'sgdm'` | `'rmsprop'` | `'adam'`

Solver for training network, specified as one of the following:

- `'sgdm'` — Use the stochastic gradient descent with momentum (SGDM) optimizer. You can specify the momentum value using the `'Momentum'` name-value pair argument.
- `'rmsprop'`— Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the `'SquaredGradientDecayFactor'` name-value pair argument.
- `'adam'`— Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the `'GradientDecayFactor'` and `'SquaredGradientDecayFactor'` name-value pair arguments, respectively.

For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1368.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InitialLearnRate',0.03,'L2Regularization',0.0005,'LearnRateSchedule','piecewise'` specifies the initial learning rate as 0.03 and the L_2 regularization factor as 0.0005, and instructs the software to drop the learning rate every given number of epochs by multiplying with a certain factor.

Plots and Display

`Plots` — Plots to display during network training
`'none'` (default) | `'training-progress'`

Plots to display during network training, specified as the comma-separated pair consisting of `'Plots'` and one of the following:

- `'none'` — Do not display plots during training.
- `'training-progress'`— Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network. For more information on the training progress plot, see “Monitor Deep Learning Training Progress” on page 1-1352.

Example: `'Plots','training-progress'`

`Verbose` — Indicator to display training progress information
`1` (true) (default) | `0` (false)

Indicator to display training progress information in the command window, specified as `1` (true) or `0` (false).

The verbose output displays the following information:

Classification Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch Accuracy`	Classification accuracy on the mini-batch.
`Validation Accuracy`	Classification accuracy on the validation data. If you do not specify validation data, then the function does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes.
`Validation Loss`	Loss on the validation data. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes. If you do not specify validation data, then the function does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

Regression Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch RMSE`	Root-mean-squared-error (RMSE) on the mini-batch.
`Validation RMSE`	RMSE on the validation data. If you do not specify validation data, then the software does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error.
`Validation Loss`	Loss on the validation data. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error. If you do not specify validation data, then the software does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

When training stops, the verbose output displays the reason for stopping.

To specify validation data, use the `ValidationData` training option.

Data Types: `logical`

`VerboseFrequency` — Frequency of verbose printing
50 (default) | positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as the comma-separated pair consisting of `'VerboseFrequency'` and a positive integer. This option only has an effect when the `'Verbose'` value equals `true`.

If you validate the network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Example: `'VerboseFrequency',100`

Mini-Batch Options

`MaxEpochs` — Maximum number of epochs
30 (default) | positive integer

Maximum number of epochs to use for training, specified as the comma-separated pair consisting of `'MaxEpochs'` and a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

Example: `'MaxEpochs',20`

`MiniBatchSize` — Size of mini-batch
128 (default) | positive integer

Size of the mini-batch to use for each training iteration, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights. See “Stochastic Gradient Descent” on page 1-1368.

Example: `'MiniBatchSize',256`

`Shuffle` — Option for data shuffling
`'once'` (default) | `'never'` | `'every-epoch'`

Option for data shuffling, specified as the comma-separated pair consisting of `'Shuffle'` and one of the following:

- `'once'` — Shuffle the training and validation data once before training.
- `'never'` — Do not shuffle the data.
- `'every-epoch'` — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then `trainNetwork` discards the training data that does not fit into the final complete mini-batch of each epoch. To avoid discarding the same data every epoch, set the `'Shuffle'` value to `'every-epoch'`.

Example: `'Shuffle','every-epoch'`

Validation

`ValidationData` — Data to use for validation during training
datastore | table | cell array

Data to use for validation during training, specified as a datastore, a table, or a cell array containing the validation predictors and responses.

You can specify validation predictors and responses using the same formats supported by the `trainNetwork` function. You can specify the validation data as a datastore, table, or the cell array `{predictors,responses}`, where `predictors` contains the validation predictors and `responses` contains the validation responses.

For more information, see the `images`, `sequences`, and `features` input arguments of the `trainNetwork` function.

During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the `ValidationFrequency` training option. 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` training option.

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.

The validation data is shuffled according to the `Shuffle` training option. If `Shuffle` is `'every-epoch'`, then the validation data is shuffled before each network validation.

`ValidationFrequency` — Frequency of network validation
50 (default) | positive integer

Frequency of network validation in number of iterations, specified as the comma-separated pair consisting of `'ValidationFrequency'` and a positive integer.

The `'ValidationFrequency'` value is the number of iterations between evaluations of validation metrics. To specify validation data, use the `'ValidationData'` name-value pair argument.

Example: `'ValidationFrequency',20`

`ValidationPatience` — Patience of validation stopping
`Inf` (default) | positive integer

Patience of validation stopping of network training, specified as a positive integer or `Inf`.

`ValidationPatience` specifies 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. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned network depends on the `OutputNetwork` training option. To return the network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

`OutputNetwork` — Network to return when training completes
`'last-iteration'` (default) | `'best-validation-loss'`

Network to return when training completes, specified as one of the following:

- `'last-iteration'` – Return the network corresponding to the last training iteration.
- `'best-validation-loss'` – Return the network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify `'ValidationData'`.

Solver Options

`InitialLearnRate` — Initial learning rate
0.001 | 0.01 | positive scalar

Initial learning rate used for training, specified as the comma-separated pair consisting of `'InitialLearnRate'` and a positive scalar. The default value is 0.01 for the `'sgdm'` solver and 0.001 for the `'rmsprop'` and `'adam'` solvers. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Example: `'InitialLearnRate',0.03`

Data Types: `single` | `double`

`LearnRateSchedule` — Option for dropping learning rate during training
`'none'` (default) | `'piecewise'`

Option for dropping the learning rate during training, specified as the comma-separated pair consisting of `'LearnRateSchedule'` and one of the following:

- `'none'` — The learning rate remains constant throughout training.
- `'piecewise'` — The software updates the learning rate every certain number of epochs by multiplying with a certain factor. Use the `LearnRateDropFactor` name-value pair argument to

specify the value of this factor. Use the `LearnRateDropPeriod` name-value pair argument to specify the number of epochs between multiplications.

Example: `'LearnRateSchedule','piecewise'`

`LearnRateDropPeriod` — Number of epochs for dropping the learning rate
10 (default) | positive integer

Number of epochs for dropping the learning rate, specified as the comma-separated pair consisting of `'LearnRateDropPeriod'` and a positive integer. This option is valid only when the value of `LearnRateSchedule` is `'piecewise'`.

The software multiplies the global learning rate with the drop factor every time the specified number of epochs passes. Specify the drop factor using the `LearnRateDropFactor` name-value pair argument.

Example: `'LearnRateDropPeriod',3`

`LearnRateDropFactor` — Factor for dropping the learning rate
0.1 (default) | scalar from 0 to 1

Factor for dropping the learning rate, specified as the comma-separated pair consisting of `'LearnRateDropFactor'` and a scalar from 0 to 1. This option is valid only when the value of `LearnRateSchedule` is `'piecewise'`.

`LearnRateDropFactor` is a multiplicative factor to apply to the learning rate every time a certain number of epochs passes. Specify the number of epochs using the `LearnRateDropPeriod` name-value pair argument.

Example: `'LearnRateDropFactor',0.1`

Data Types: `single` | `double`

`L2Regularization` — Factor for L_2 regularization
0.0001 (default) | nonnegative scalar

Factor for L_2 regularization (weight decay), specified as the comma-separated pair consisting of `'L2Regularization'` and a nonnegative scalar. For more information, see “L2 Regularization” on page 1-1370.

You can specify a multiplier for the L_2 regularization for network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Example: `'L2Regularization',0.0005`

Data Types: `single` | `double`

`Momentum` — Contribution of previous step
0.9 (default) | scalar from 0 to 1

Contribution of the parameter update step of the previous iteration to the current iteration of stochastic gradient descent with momentum, specified as the comma-separated pair consisting of `'Momentum'` and a scalar from 0 to 1. A value of 0 means no contribution from the previous step, whereas a value of 1 means maximal contribution from the previous step.

To specify the `'Momentum'` value, you must set `solverName` to be `'sgdm'`. The default value works well for most problems. For more information about the different solvers, see "Stochastic Gradient Descent" on page 1-1368.

Example: `'Momentum',0.95`

Data Types: `single` | `double`

`GradientDecayFactor` — Decay rate of gradient moving average
0.9 (default) | nonnegative scalar less than 1

Decay rate of gradient moving average for the Adam solver, specified as the comma-separated pair consisting of `'GradientDecayFactor'` and a nonnegative scalar less than 1. The gradient decay rate is denoted by β_1 in [4].

To specify the `'GradientDecayFactor'` value, you must set `solverName` to be `'adam'`. The default value works well for most problems. For more information about the different solvers, see "Stochastic Gradient Descent" on page 1-1368.

Example: `'GradientDecayFactor',0.95`

Data Types: `single` | `double`

`SquaredGradientDecayFactor` — Decay rate of squared gradient moving average
0.9 | 0.999 | nonnegative scalar less than 1

Decay rate of squared gradient moving average for the Adam and RMSProp solvers, specified as the comma-separated pair consisting of `'SquaredGradientDecayFactor'` and a nonnegative scalar less than 1. The squared gradient decay rate is denoted by β_2 in [4].

To specify the `'SquaredGradientDecayFactor'` value, you must set `solverName` to be `'adam'` or `'rmsprop'`. Typical values of the decay rate are 0.9, 0.99, and 0.999, corresponding to averaging lengths of 10, 100, and 1000 parameter updates, respectively. The default value is 0.999 for the Adam solver. The default value is 0.9 for the RMSProp solver.

For more information about the different solvers, see "Stochastic Gradient Descent" on page 1-1368.

Example: `'SquaredGradientDecayFactor',0.99`

Data Types: `single` | `double`

`Epsilon` — Denominator offset
10^{-8} (default) | positive scalar

Denominator offset for Adam and RMSProp solvers, specified as the comma-separated pair consisting of `'Epsilon'` and a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.

To specify the `'Epsilon'` value, you must set `solverName` to be `'adam'` or `'rmsprop'`. The default value works well for most problems. For more information about the different solvers, see "Stochastic Gradient Descent" on page 1-1368.

Example: `'Epsilon',1e-6`

Data Types: `single` | `double`

`ResetInputNormalization` — Option to reset input layer normalization
`true` (default) | `false`

Option to reset input layer normalization, specified as one of the following:

- `true` – Reset the input layer normalization statistics and recalculate them at training time.
- `false` – Calculate normalization statistics at training time when they are empty.

`BatchNormalizationStatistics` — Mode to evaluate statistics in batch normalization layers

`'population'` (default) | `'moving'`

Mode to evaluate the statistics in batch normalization layers, specified as one of the following:

- `'population'` – Use the population statistics. After training, the software finalizes the statistics by passing through the training data once more and uses the resulting mean and variance.
- `'moving'` – Approximate the statistics during training using a running estimate given by update steps

  $$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu$$

  $$\sigma^{2*} = \lambda_{\sigma^2}\widehat{\sigma^2} + (1-\lambda_{\sigma^2})\sigma^2$$

  where μ^* and σ^{2*} denote the updated mean and variance, respectively, λ_μ and λ_{σ^2} denote the mean and variance decay values, respectively, $\widehat{\mu}$ and $\widehat{\sigma^2}$ denote the mean and variance of the layer input, respectively, and μ and σ^2 denote the latest values of the moving mean and variance values, respectively. After training, the software uses the most recent value of the moving mean and variance statistics. This option supports CPU and single GPU training only.

Gradient Clipping

`GradientThreshold` — Gradient threshold

`Inf` (default) | positive scalar

Gradient threshold, specified as the comma-separated pair consisting of `'GradientThreshold'` and `Inf` or a positive scalar. If the gradient exceeds the value of `GradientThreshold`, then the gradient is clipped according to `GradientThresholdMethod`.

Example: `'GradientThreshold',6`

`GradientThresholdMethod` — Gradient threshold method

`'l2norm'` (default) | `'global-l2norm'` | `'absolute-value'`

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as the comma-separated pair consisting of `'GradientThresholdMethod'` and one of the following:

- `'l2norm'` — If the L_2 norm of the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the gradient so that the L_2 norm equals `GradientThreshold`.
- `'global-l2norm'` — If the global L_2 norm, L, is larger than `GradientThreshold`, then scale all gradients by a factor of `GradientThreshold`/L. The global L_2 norm considers all learnable parameters.
- `'absolute-value'` — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the partial derivative to have magnitude equal to `GradientThreshold` and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-1370.

Example: `'GradientThresholdMethod','global-l2norm'`

Sequence Options

`SequenceLength` — Option to pad, truncate, or split input sequences
`'longest'` (default) | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

Example: `'SequenceLength','shortest'`

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Example: `'SequencePaddingValue',-1`

Hardware Options

`ExecutionEnvironment` — Hardware resource for training network

`'auto'` (default) | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training network, specified as one of the following:

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Datastores used for multi-GPU training or parallel training must be partitionable. For more information, see “Use Datastore for Parallel Training and Background Dispatching”.

If you use the `'multi-gpu'` option with a partitionable input datastore and the `'DispatchInBackground'` option, then the software starts a parallel pool with size equal to the default pool size. Workers with unique GPUs perform training computation. The remaining workers are used for background dispatch.

Example: `'ExecutionEnvironment','cpu'`

`WorkerLoad` — Parallel worker load division

scalar from 0 to 1 | positive integer | numeric vector

Parallel worker load division between GPUs or CPUs, specified as the comma-separated pair consisting of `'WorkerLoad'` and one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for network training computation. If you train the network using data in a mini-batch datastore with background dispatch enabled, then the remaining workers fetch and preprocess data in the background.

- Positive integer — Number of workers on each machine to use for network training computation. If you train the network using data in a mini-batch datastore with background dispatch enabled, then the remaining workers fetch and preprocess data in the background.
- Numeric vector — Network training load for each worker in the parallel pool. For a vector `W`, worker `i` gets a fraction `W(i)/sum(W)` of the work (number of examples per mini-batch). If you train a network using data in a mini-batch datastore with background dispatch enabled, then you can assign a worker load of 0 to use that worker for fetching data in the background. The specified vector must contain one value per worker in the parallel pool.

If the parallel pool has access to GPUs, then workers without a unique GPU are never used for training computation. The default for pools with GPUs is to use all workers with a unique GPU for training computation, and the remaining workers for background dispatch. If the pool does not have access to GPUs and CPUs are used for training, then the default is to use one worker per machine for background data dispatch.

`DispatchInBackground` — Use background dispatch
`false` (default) | `true`

Use background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as `false` or `true`. Background dispatch requires Parallel Computing Toolbox.

`DispatchInBackground` is only supported for datastores that are partitionable. For more information, see “Use Datastore for Parallel Training and Background Dispatching”.

Checkpoints

`CheckpointPath` — Path for saving checkpoint networks
`''` (default) | character vector

Path for saving the checkpoint networks, specified as the comma-separated pair consisting of `'CheckpointPath'` and a character vector.

- If you do not specify a path (that is, you use the default `''`), then the software does not save any checkpoint networks.
- If you specify a path, then `trainNetwork` saves checkpoint networks to this path after every epoch and assigns a unique name to each network. You can then load any checkpoint network and resume training from that network.

  If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint networks. If the path you specify does not exist, then `trainingOptions` returns an error.

For more information about saving network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Example: `'CheckpointPath','C:\Temp\checkpoint'`

Data Types: `char`

`OutputFcn` — Output functions
function handle | cell array of function handles

Output functions to call during training, specified as the comma-separated pair consisting of `'OutputFcn'` and a function handle or cell array of function handles. `trainNetwork` calls the

specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

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. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training”.

Data Types: `function_handle` | `cell`

Output Arguments

`options` — Training options

`TrainingOptionsSGDM` | `TrainingOptionsRMSProp` | `TrainingOptionsADAM`

Training options, returned as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object. To train a neural network, use the training options as an input argument to the `trainNetwork` function.

If `solverName` equals `'sgdm'`, `'rmsprop'`, or `'adam'`, then the training options are returned as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object, respectively.

You can edit training option properties of `TrainingOptionsSGDM`, `TrainingOptionsADAM`, and `TrainingOptionsRMSProp` objects directly. For example, to change the mini-batch size after using the `trainingOptions` function, you can edit the `MiniBatchSize` property directly:

```
options = trainingOptions('sgdm');
options.MiniBatchSize = 64;
```

Tips

- 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”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

  If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Deep Learning Network for Custom Training Loops”.

Algorithms

Initial Weights and Biases

For convolutional and fully connected layers, the initialization for the weights and biases are given by the `WeightsInitializer` and `BiasInitializer` properties of the layers, respectively. For examples showing how to change the initialization for the weights and biases, see “Specify Initial Weights and Biases in Convolutional Layer” on page 1-326 and “Specify Initial Weights and Biases in Fully Connected Layer” on page 1-616.

Stochastic Gradient Descent

The standard gradient descent algorithm updates the network parameters (weights and biases) to minimize the loss function by taking small steps at each iteration in the direction of the negative gradient of the loss,

$$\theta_{\ell+1} = \theta_{\ell} - \alpha \nabla E(\theta_{\ell}),$$

where $\ell$is the iteration number, $\alpha > 0$ is the learning rate, θ is the parameter vector, and $E(\theta)$ is the loss function. In the standard gradient descent algorithm, the gradient of the loss function, $\nabla E(\theta)$, is evaluated using the entire training set, and the standard gradient descent algorithm uses the entire data set at once.

By contrast, at each iteration the *stochastic* gradient descent algorithm evaluates the gradient and updates the parameters using a subset of the training data. A different subset, called a mini-batch, is used at each iteration. The full pass of the training algorithm over the entire training set using mini-batches is one *epoch*. Stochastic gradient descent is stochastic because the parameter updates computed using a mini-batch is a noisy estimate of the parameter update that would result from using the full data set. You can specify the mini-batch size and the maximum number of epochs by using the `'MiniBatchSize'` and `'MaxEpochs'` name-value pair arguments, respectively.

Stochastic Gradient Descent with Momentum

The stochastic gradient descent algorithm can oscillate along the path of steepest descent towards the optimum. Adding a momentum term to the parameter update is one way to reduce this oscillation [2]. The stochastic gradient descent with momentum (SGDM) update is

$$\theta_{\ell+1} = \theta_{\ell} - \alpha \nabla E(\theta_{\ell}) + \gamma(\theta_{\ell} - \theta_{\ell-1}),$$

where γ determines the contribution of the previous gradient step to the current iteration. You can specify this value using the `'Momentum'` name-value pair argument. To train a neural network using the stochastic gradient descent with momentum algorithm, specify `solverName` as `'sgdm'`. To

specify the initial value of the learning rate α, use the `'InitialLearnRate'` name-value pair argument. You can also specify different learning rates for different layers and parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

RMSProp

Stochastic gradient descent with momentum uses a single learning rate for all the parameters. Other optimization algorithms seek to improve network training by using learning rates that differ by parameter and can automatically adapt to the loss function being optimized. RMSProp (root mean square propagation) is one such algorithm. It keeps a moving average of the element-wise squares of the parameter gradients,

$$v_\ell = \beta_2 v_{\ell-1} + (1-\beta_2)[\nabla E(\theta_\ell)]^2$$

β_2 is the decay rate of the moving average. Common values of the decay rate are 0.9, 0.99, and 0.999. The corresponding averaging lengths of the squared gradients equal *1/(1-β_2)*, that is, 10, 100, and 1000 parameter updates, respectively. You can specify β_2 by using the `'SquaredGradientDecayFactor'` name-value pair argument. The RMSProp algorithm uses this moving average to normalize the updates of each parameter individually,

$$\theta_{\ell+1} = \theta_\ell - \frac{\alpha \nabla E(\theta_\ell)}{\sqrt{v_\ell} + \epsilon}$$

where the division is performed element-wise. Using RMSProp effectively decreases the learning rates of parameters with large gradients and increases the learning rates of parameters with small gradients. ε is a small constant added to avoid division by zero. You can specify ε by using the `'Epsilon'` name-value pair argument, but the default value usually works well. To use RMSProp to train a neural network, specify `solverName` as `'rmsprop'`.

Adam

Adam (derived from *adaptive moment estimation*) [4] uses a parameter update that is similar to RMSProp, but with an added momentum term. It keeps an element-wise moving average of both the parameter gradients and their squared values,

$$m_\ell = \beta_1 m_{\ell-1} + (1-\beta_1)\nabla E(\theta_\ell)$$

$$v_\ell = \beta_2 v_{\ell-1} + (1-\beta_2)[\nabla E(\theta_\ell)]^2$$

You can specify the β_1 and β_2 decay rates using the `'GradientDecayFactor'` and `'SquaredGradientDecayFactor'` name-value pair arguments, respectively. Adam uses the moving averages to update the network parameters as

$$\theta_{\ell+1} = \theta_\ell - \frac{\alpha m_l}{\sqrt{v_l} + \epsilon}$$

If gradients over many iterations are similar, then using a moving average of the gradient enables the parameter updates to pick up momentum in a certain direction. If the gradients contain mostly noise, then the moving average of the gradient becomes smaller, and so the parameter updates become smaller too. You can specify ε by using the `'Epsilon'` name-value pair argument. The default value usually works well, but for certain problems a value as large as 1 works better. To use Adam to train a neural network, specify `solverName` as `'adam'`. The full Adam update also includes a mechanism to correct a bias the appears in the beginning of training. For more information, see [4].

Specify the learning rate α for all optimization algorithms using the `'InitialLearnRate'` name-value pair argument. The effect of the learning rate is different for the different optimization algorithms, so the optimal learning rates are also different in general. You can also specify learning rates that differ by layers and by parameter. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Gradient Clipping

If the gradients increase in magnitude exponentially, then the training is unstable and can diverge within a few iterations. This "gradient explosion" is indicated by a training loss that goes to `NaN` or `Inf`. Gradient clipping helps prevent gradient explosion by stabilizing the training at higher learning rates and in the presence of outliers [3]. Gradient clipping enables networks to be trained faster, and does not usually impact the accuracy of the learned task.

There are two types of gradient clipping.

- Norm-based gradient clipping rescales the gradient based on a threshold, and does not change the direction of the gradient. The `'l2norm'` and `'global-l2norm'` values of `GradientThresholdMethod` are norm-based gradient clipping methods.
- Value-based gradient clipping clips any partial derivative greater than the threshold, which can result in the gradient arbitrarily changing direction. Value-based gradient clipping can have unpredictable behavior, but sufficiently small changes do not cause the network to diverge. The `'absolute-value'` value of `GradientThresholdMethod` is a value-based gradient clipping method.

For examples, see “Time Series Forecasting Using Deep Learning” and “Sequence-to-Sequence Classification Using Deep Learning”.

L_2 Regularization

Adding a regularization term for the weights to the loss function $E(\theta)$ is one way to reduce overfitting [1], [2]. The regularization term is also called *weight decay*. The loss function with the regularization term takes the form

$$E_R(\theta) = E(\theta) + \lambda\Omega(w),$$

where w is the weight vector, λ is the regularization factor (coefficient), and the regularization function $\Omega(w)$ is

$$\Omega(w) = \frac{1}{2}w^T w\,.$$

Note that the biases are not regularized [2]. You can specify the regularization factor λ by using the `'L2Regularization'` name-value pair argument. You can also specify different regularization factors for different layers and parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

The loss function that the software uses for network training includes the regularization term. However, the loss value displayed in the command window and training progress plot during training is the loss on the data only and does not include the regularization term.

Compatibility Considerations

`'ValidationPatience'` training option default is `Inf`
Behavior changed in R2018b

Starting in R2018b, the default value of the `'ValidationPatience'` training option is `Inf`, which means that automatic stopping via validation is turned off. This behavior prevents the training from stopping before sufficiently learning from the data.

In previous versions, the default value is `5`. To reproduce this behavior, set the `'ValidationPatience'` option to `5`.

Different file name for checkpoint networks
Behavior changed in R2018b

Starting in R2018b, when saving checkpoint networks, the software assigns file names beginning with `net_checkpoint_`. In previous versions, the software assigns file names beginning with `convnet_checkpoint_`.

If you have code that saves and loads checkpoint networks, then update your code to load files with the new name.

References

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

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

[3] Pascanu, R., T. Mikolov, and Y. Bengio. "On the difficulty of training recurrent neural networks". *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28(3), 2013, pp. 1310–1318.

[4] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

See Also

`trainNetwork` | `analyzeNetwork` | **Deep Network Designer**

Topics
“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2016a

TrainingOptionsADAM

Training options for Adam optimizer

Description

Training options for Adam (adaptive moment estimation) optimizer, including learning rate information, L_2 regularization factor, and mini-batch size.

Creation

Create a `TrainingOptionsADAM` object using `trainingOptions` and specifying `'adam'` as the `solverName` input argument.

Properties

Plots and Display

`Plots` — Plots to display during network training
`'none'` | `'training-progress'`

Plots to display during network training, specified as one of the following:

- `'none'` — Do not display plots during training.
- `'training-progress'` — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

`Verbose` — Indicator to display training progress information
`1` (true) (default) | `0` (false)

Indicator to display training progress information in the command window, specified as `1` (true) or `0` (false).

The verbose output displays the following information:

Classification Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch Accuracy`	Classification accuracy on the mini-batch.
`Validation Accuracy`	Classification accuracy on the validation data. If you do not specify validation data, then the function does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes.
`Validation Loss`	Loss on the validation data. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes. If you do not specify validation data, then the function does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

Regression Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch RMSE`	Root-mean-squared-error (RMSE) on the mini-batch.
`Validation RMSE`	RMSE on the validation data. If you do not specify validation data, then the software does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error.
`Validation Loss`	Loss on the validation data. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error. If you do not specify validation data, then the software does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

When training stops, the verbose output displays the reason for stopping.

To specify validation data, use the `ValidationData` training option.

Data Types: `logical`

`VerboseFrequency` — Frequency of verbose printing
positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the `Verbose` value equals `true`.

If you validate the network during training, then `trainNetwork` prints to the command window every time validation occurs.

Mini-Batch Options

`MaxEpochs` — Maximum number of epochs
positive integer

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

`MiniBatchSize` — Size of mini-batch
positive integer

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

`Shuffle` — Option for data shuffling
`'once'` | `'never'` | `'every-epoch'`

Option for data shuffling, specified as one of the following:

- `'once'` — Shuffle the training and validation data once before training.
- `'never'` — Do not shuffle the data.
- `'every-epoch'` — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then `trainNetwork` discards the training data that does not fit into the final complete mini-batch of each epoch. Set the `Shuffle` value to `'every-epoch'` to avoid discarding the same data every epoch.

Validation

`ValidationData` — Data to use for validation during training
datastore | table | cell array

Data to use for validation during training, specified as a datastore, a table, or a cell array containing the validation predictors and responses.

You can specify validation predictors and responses using the same formats supported by the `trainNetwork` function. You can specify the validation data as a datastore, table, or the cell array `{predictors,responses}`, where `predictors` contains the validation predictors and `responses` contains the validation responses.

For more information, see the `images`, `sequences`, and `features` input arguments of the `trainNetwork` function.

During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the `ValidationFrequency` training option. 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` training option.

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.

The validation data is shuffled according to the `Shuffle` training option. If `Shuffle` is `'every-epoch'`, then the validation data is shuffled before each network validation.

`ValidationFrequency` — Frequency of network validation
positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics.

`ValidationPatience` — Patience of validation stopping
`Inf` (default) | positive integer

Patience of validation stopping of network training, specified as a positive integer or `Inf`.

`ValidationPatience` specifies 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. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned network depends on the `OutputNetwork` training option. To return the network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

`OutputNetwork` — Network to return when training completes
`'last-iteration'` (default) | `'best-validation-loss'`

Network to return when training completes, specified as one of the following:

- `'last-iteration'` – Return the network corresponding to the last training iteration.
- `'best-validation-loss'` – Return the network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify `'ValidationData'`.

Solver Options

`InitialLearnRate` — Initial learning rate
positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.

`LearnRateScheduleSettings` — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure. `LearnRateScheduleSettings` has the field `Method`, which specifies the type of method for adjusting the learning rate. The possible methods are:

- `'none'` — The learning rate is constant throughout training.
- `'piecewise'` — The learning rate drops periodically during training.

If `Method` is `'piecewise'`, then `LearnRateScheduleSettings` contains two more fields:

- `DropRateFactor` — The multiplicative factor by which the learning rate drops during training
- `DropPeriod` — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using `trainingOptions`.

Data Types: `struct`

`L2Regularization` — Factor for L_2 regularizer
nonnegative scalar

Factor for L_2 regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the L_2 regularizer for network layers with learnable parameters.

`GradientDecayFactor` — Decay rate of gradient moving average
scalar from 0 to 1

Decay rate of gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1368.

`SquaredGradientDecayFactor` — Decay rate of squared gradient moving average
scalar from 0 to 1

Decay rate of squared gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1368.

`Epsilon` — Denominator offset
positive scalar

Denominator offset, specified as a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.

`ResetInputNormalization` — Option to reset input layer normalization
`true` (default) | `false`

Option to reset input layer normalization, specified as one of the following:

- `true` – Reset the input layer normalization statistics and recalculate them at training time.
- `false` – Calculate normalization statistics at training time when they are empty.

`BatchNormalizationStatistics` — Mode to evaluate statistics in batch normalization layers
`'population'` (default) | `'moving'`

Mode to evaluate the statistics in batch normalization layers, specified as one of the following:

- `'population'` – Use the population statistics. After training, the software finalizes the statistics by passing through the training data once more and uses the resulting mean and variance.
- `'moving'` – Approximate the statistics during training using a running estimate given by update steps

  $$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu$$

  $$\sigma^{2*} = \lambda_{\sigma^2}\widehat{\sigma^2} + (1-\lambda_{\sigma^2})\sigma^2$$

  where μ^* and σ^{2*} denote the updated mean and variance, respectively, λ_μ and λ_{σ^2} denote the mean and variance decay values, respectively, $\widehat{\mu}$ and $\widehat{\sigma^2}$ denote the mean and variance of the layer input, respectively, and μ and σ^2 denote the latest values of the moving mean and variance values, respectively. After training, the software uses the most recent value of the moving mean and variance statistics. This option supports CPU and single GPU training only.

Gradient Clipping

`GradientThreshold` — Gradient threshold
positive scalar | `Inf`

Positive threshold for the gradient, specified as positive scalar or `Inf`. When the gradient exceeds the value of `GradientThreshold`, then the gradient is clipped according to `GradientThresholdMethod`.

`GradientThresholdMethod` — Gradient threshold method
`'l2norm'` | `'global-l2norm'` | `'absolutevalue'`

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- `'l2norm'` — If the L_2 norm of the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the gradient so that the L_2 norm equals `GradientThreshold`.
- `'global-l2norm'` — If the global L_2 norm, L, is larger than `GradientThreshold`, then scale all gradients by a factor of `GradientThreshold`/L. The global L_2 norm considers all learnable parameters.
- `'absolute-value'` — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the partial derivative to have magnitude equal to `GradientThreshold` and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-1370.

Sequence Options

`SequenceLength` — Option to pad or truncate sequences
`'longest'` | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad sequences
scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Hardware Options

`ExecutionEnvironment` — Hardware resource for training network
`'auto'` | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training network, specified as one of the following:

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see "Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud".

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Specify the execution environment using `trainingOptions`.

Data Types: `char` | `string`

`WorkerLoad` — Parallel worker load division
scalar from 0 to 1 | positive integer | numeric vector

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the `ExecutionEnvironment` value equals `'multi-gpu'` or `'parallel'`.

Checkpoints

`CheckpointPath` — Path for saving checkpoint networks
character vector

Path where checkpoint networks are saved, specified as a character vector.

Data Types: `char`

`OutputFcn` — Output functions
function handle | cell array of function handles

Output functions to call during training, specified as a function handle or cell array of function handles. `trainNetwork` calls the specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

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. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training” .

Data Types: `function_handle` | `cell`

Examples

Create Training Options for the Adam Optimizer

Create a set of options for training a neural network using the Adam optimizer. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Specify the learning rate and the decay rate of the moving average of the squared gradient. Turn on the training progress plot.

```
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',20, ...
    'MiniBatchSize',64, ...
    'Plots','training-progress')
```

```
options =
  TrainingOptionsADAM with properties:

            GradientDecayFactor: 0.9000
        SquaredGradientDecayFactor: 0.9900
                           Epsilon: 1.0000e-08
                  InitialLearnRate: 3.0000e-04
                 LearnRateSchedule: 'none'
                LearnRateDropFactor: 0.1000
                LearnRateDropPeriod: 10
                   L2Regularization: 1.0000e-04
           GradientThresholdMethod: 'l2norm'
                 GradientThreshold: Inf
                         MaxEpochs: 20
                     MiniBatchSize: 64
                           Verbose: 1
                   VerboseFrequency: 50
                     ValidationData: []
                ValidationFrequency: 50
                 ValidationPatience: Inf
                           Shuffle: 'once'
                    CheckpointPath: ''
              ExecutionEnvironment: 'auto'
                        WorkerLoad: []
                         OutputFcn: []
                             Plots: 'training-progress'
                    SequenceLength: 'longest'
              SequencePaddingValue: 0
          SequencePaddingDirection: 'right'
              DispatchInBackground: 0
           ResetInputNormalization: 1
      BatchNormalizationStatistics: 'population'
                   OutputNetwork: 'last-iteration'
```

References

[1] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2018a

TrainingOptionsRMSProp

Training options for RMSProp optimizer

Description

Training options for RMSProp (root mean square propagation) optimizer, including learning rate information, L_2 regularization factor, and mini-batch size.

Creation

Create a `TrainingOptionsRMSProp` object using `trainingOptions` and specifying `'rmsprop'` as the `solverName` input argument.

Properties

Plots and Display

`Plots` — Plots to display during network training
`'none'` | `'training-progress'`

Plots to display during network training, specified as one of the following:

- `'none'` — Do not display plots during training.
- `'training-progress'` — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

`Verbose` — Indicator to display training progress information
`1` (true) (default) | `0` (false)

Indicator to display training progress information in the command window, specified as `1` (true) or `0` (false).

The verbose output displays the following information:

Classification Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch Accuracy`	Classification accuracy on the mini-batch.
`Validation Accuracy`	Classification accuracy on the validation data. If you do not specify validation data, then the function does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes.
`Validation Loss`	Loss on the validation data. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes. If you do not specify validation data, then the function does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

Regression Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch RMSE`	Root-mean-squared-error (RMSE) on the mini-batch.
`Validation RMSE`	RMSE on the validation data. If you do not specify validation data, then the software does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error.
`Validation Loss`	Loss on the validation data. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error. If you do not specify validation data, then the software does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

When training stops, the verbose output displays the reason for stopping.

To specify validation data, use the `ValidationData` training option.

Data Types: `logical`

`VerboseFrequency` — Frequency of verbose printing
positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the `Verbose` value equals `true`.

If you validate the network during training, then `trainNetwork` prints to the command window every time validation occurs.

Mini-Batch Options

`MaxEpochs` — Maximum number of epochs
positive integer

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

`MiniBatchSize` — Size of mini-batch
positive integer

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

`Shuffle` — Option for data shuffling
`'once'` | `'never'` | `'every-epoch'`

Option for data shuffling, specified as one of the following:

- `'once'` — Shuffle the training and validation data once before training.
- `'never'` — Do not shuffle the data.
- `'every-epoch'` — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then `trainNetwork` discards the training data that does not fit into the final complete mini-batch of each epoch. Set the `Shuffle` value to `'every-epoch'` to avoid discarding the same data every epoch.

Validation

`ValidationData` — Data to use for validation during training
datastore | table | cell array

Data to use for validation during training, specified as a datastore, a table, or a cell array containing the validation predictors and responses.

You can specify validation predictors and responses using the same formats supported by the `trainNetwork` function. You can specify the validation data as a datastore, table, or the cell array `{predictors,responses}`, where `predictors` contains the validation predictors and `responses` contains the validation responses.

For more information, see the `images`, `sequences`, and `features` input arguments of the `trainNetwork` function.

During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the `ValidationFrequency` training option. 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` training option.

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.

The validation data is shuffled according to the `Shuffle` training option. If `Shuffle` is `'every-epoch'`, then the validation data is shuffled before each network validation.

`ValidationFrequency` — Frequency of network validation
positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics.

`ValidationPatience` — Patience of validation stopping
`Inf` (default) | positive integer

Patience of validation stopping of network training, specified as a positive integer or `Inf`.

`ValidationPatience` specifies 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. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned network depends on the `OutputNetwork` training option. To return the network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

`OutputNetwork` — Network to return when training completes
`'last-iteration'` (default) | `'best-validation-loss'`

Network to return when training completes, specified as one of the following:

- `'last-iteration'` – Return the network corresponding to the last training iteration.
- `'best-validation-loss'` – Return the network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify `'ValidationData'`.

Solver Options

`InitialLearnRate` — Initial learning rate
positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.

`LearnRateScheduleSettings` — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure. `LearnRateScheduleSettings` has the field `Method`, which specifies the type of method for adjusting the learning rate. The possible methods are:

- `'none'` — The learning rate is constant throughout training.
- `'piecewise'` — The learning rate drops periodically during training.

If `Method` is `'piecewise'`, then `LearnRateScheduleSettings` contains two more fields:

- `DropRateFactor` — The multiplicative factor by which the learning rate drops during training
- `DropPeriod` — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using `trainingOptions`.

Data Types: `struct`

`L2Regularization` — Factor for L_2 regularizer
nonnegative scalar

Factor for L_2 regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the L_2 regularizer for network layers with learnable parameters.

`SquaredGradientDecayFactor` — Decay rate of squared gradient moving average
scalar from 0 to 1

Decay rate of squared gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1368.

`Epsilon` — Denominator offset
positive scalar

Denominator offset, specified as a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.

`ResetInputNormalization` — Option to reset input layer normalization
`true` (default) | `false`

Option to reset input layer normalization, specified as one of the following:

- `true` – Reset the input layer normalization statistics and recalculate them at training time.
- `false` – Calculate normalization statistics at training time when they are empty.

`BatchNormalizationStatistics` — Mode to evaluate statistics in batch normalization layers
`'population'` (default) | `'moving'`

Mode to evaluate the statistics in batch normalization layers, specified as one of the following:

- `'population'` – Use the population statistics. After training, the software finalizes the statistics by passing through the training data once more and uses the resulting mean and variance.
- `'moving'` – Approximate the statistics during training using a running estimate given by update steps

  $$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu$$

  $$\sigma^{2*} = \lambda_{\sigma^2}\widehat{\sigma^2} + (1-\lambda_{\sigma^2})\sigma^2$$

  where μ^* and σ^{2*} denote the updated mean and variance, respectively, λ_μ and λ_{σ^2} denote the mean and variance decay values, respectively, $\widehat{\mu}$ and $\widehat{\sigma^2}$ denote the mean and variance of the layer input, respectively, and μ and σ^2 denote the latest values of the moving mean and variance values, respectively. After training, the software uses the most recent value of the moving mean and variance statistics. This option supports CPU and single GPU training only.

Gradient Clipping

`GradientThreshold` — Gradient threshold
positive scalar | `Inf`

Positive threshold for the gradient, specified as positive scalar or `Inf`. When the gradient exceeds the value of `GradientThreshold`, then the gradient is clipped according to `GradientThresholdMethod`.

`GradientThresholdMethod` — Gradient threshold method
`'l2norm'` | `'global-l2norm'` | `'absolutevalue'`

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- `'l2norm'` — If the L_2 norm of the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the gradient so that the L_2 norm equals `GradientThreshold`.
- `'global-l2norm'` — If the global L_2 norm, L, is larger than `GradientThreshold`, then scale all gradients by a factor of `GradientThreshold`/L. The global L_2 norm considers all learnable parameters.
- `'absolute-value'` — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the partial derivative to have magnitude equal to `GradientThreshold` and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-1370.

Sequence Options

`SequenceLength` — Option to pad or truncate sequences
`'longest'` | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad sequences
scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Hardware Options

`ExecutionEnvironment` — Hardware resource for training network
`'auto'` | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training network, specified as one of the following:

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see "Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud".

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Specify the execution environment using `trainingOptions`.

Data Types: `char` | `string`

`WorkerLoad` — Parallel worker load division
scalar from 0 to 1 | positive integer | numeric vector

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the `ExecutionEnvironment` value equals `'multi-gpu'` or `'parallel'`.

Checkpoints

`CheckpointPath` — Path for saving checkpoint networks

character vector

Path where checkpoint networks are saved, specified as a character vector.

Data Types: `char`

`OutputFcn` — Output functions

function handle | cell array of function handles

Output functions to call during training, specified as a function handle or cell array of function handles. `trainNetwork` calls the specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

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. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training” .

Data Types: `function_handle` | `cell`

Examples

Create Training Options for the RMSProp Optimizer

Create a set of options for training a neural network using the RMSProp optimizer. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration.

Specify the learning rate and the decay rate of the moving average of the squared gradient. Turn on the training progress plot.

```
options = trainingOptions('rmsprop', ...
    'InitialLearnRate',3e-4, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',20, ...
    'MiniBatchSize',64, ...
    'Plots','training-progress')
```

```
options =
  TrainingOptionsRMSProp with properties:

      SquaredGradientDecayFactor: 0.9900
                         Epsilon: 1.0000e-08
                InitialLearnRate: 3.0000e-04
               LearnRateSchedule: 'none'
             LearnRateDropFactor: 0.1000
             LearnRateDropPeriod: 10
                L2Regularization: 1.0000e-04
        GradientThresholdMethod: 'l2norm'
              GradientThreshold: Inf
                      MaxEpochs: 20
                  MiniBatchSize: 64
                        Verbose: 1
                VerboseFrequency: 50
                  ValidationData: []
             ValidationFrequency: 50
              ValidationPatience: Inf
                         Shuffle: 'once'
                 CheckpointPath: ''
           ExecutionEnvironment: 'auto'
                     WorkerLoad: []
                      OutputFcn: []
                          Plots: 'training-progress'
                 SequenceLength: 'longest'
           SequencePaddingValue: 0
       SequencePaddingDirection: 'right'
           DispatchInBackground: 0
         ResetInputNormalization: 1
   BatchNormalizationStatistics: 'population'
                 OutputNetwork: 'last-iteration'
```

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2018a

TrainingOptionsSGDM

Training options for stochastic gradient descent with momentum

Description

Training options for stochastic gradient descent with momentum, including learning rate information, L_2 regularization factor, and mini-batch size.

Creation

Create a `TrainingOptionsSGDM` object using `trainingOptions` and specifying `'sgdm'` as the `solverName` input argument.

Properties

Plots and Display

`Plots` — Plots to display during network training
`'none'` | `'training-progress'`

Plots to display during network training, specified as one of the following:

- `'none'` — Do not display plots during training.
- `'training-progress'` — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

`Verbose` — Indicator to display training progress information
`1` (true) (default) | `0` (false)

Indicator to display training progress information in the command window, specified as `1` (true) or `0` (false).

The verbose output displays the following information:

Classification Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch Accuracy`	Classification accuracy on the mini-batch.
`Validation Accuracy`	Classification accuracy on the validation data. If you do not specify validation data, then the function does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes.
`Validation Loss`	Loss on the validation data. If the output layer is a `ClassificationOutputLayer` object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes. If you do not specify validation data, then the function does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

Regression Networks

Field	Description
`Epoch`	Epoch number. An epoch corresponds to a full pass of the data.
`Iteration`	Iteration number. An iteration corresponds to a mini-batch.
`Time Elapsed`	Time elapsed in hours, minutes, and seconds.
`Mini-batch RMSE`	Root-mean-squared-error (RMSE) on the mini-batch.
`Validation RMSE`	RMSE on the validation data. If you do not specify validation data, then the software does not display this field.
`Mini-batch Loss`	Loss on the mini-batch. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error.
`Validation Loss`	Loss on the validation data. If the output layer is a `RegressionOutputLayer` object, then the loss is the half-mean-squared-error. If you do not specify validation data, then the software does not display this field.
`Base Learning Rate`	Base learning rate. The software multiplies the learn rate factors of the layers by this value.

When training stops, the verbose output displays the reason for stopping.

To specify validation data, use the `ValidationData` training option.

Data Types: `logical`

`VerboseFrequency` — Frequency of verbose printing
positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the `Verbose` value equals `true`.

If you validate the network during training, then `trainNetwork` prints to the command window every time validation occurs.

Mini-Batch Options

`MaxEpochs` — Maximum number of epochs
positive integer

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

`MiniBatchSize` — Size of mini-batch
positive integer

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

`Shuffle` — Option for data shuffling
`'once'` | `'never'` | `'every-epoch'`

Option for data shuffling, specified as one of the following:

- `'once'` — Shuffle the training and validation data once before training.
- `'never'` — Do not shuffle the data.
- `'every-epoch'` — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then `trainNetwork` discards the training data that does not fit into the final complete mini-batch of each epoch. Set the `Shuffle` value to `'every-epoch'` to avoid discarding the same data every epoch.

Validation

`ValidationData` — Data to use for validation during training
datastore | table | cell array

Data to use for validation during training, specified as a datastore, a table, or a cell array containing the validation predictors and responses.

You can specify validation predictors and responses using the same formats supported by the `trainNetwork` function. You can specify the validation data as a datastore, table, or the cell array `{predictors,responses}`, where `predictors` contains the validation predictors and `responses` contains the validation responses.

For more information, see the `images`, `sequences`, and `features` input arguments of the `trainNetwork` function.

During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the `ValidationFrequency` training option. 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` training option.

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.

The validation data is shuffled according to the `Shuffle` training option. If `Shuffle` is `'every-epoch'`, then the validation data is shuffled before each network validation.

`ValidationFrequency` — Frequency of network validation
positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics.

`ValidationPatience` — Patience of validation stopping
`Inf` (default) | positive integer

Patience of validation stopping of network training, specified as a positive integer or `Inf`.

`ValidationPatience` specifies 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. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned network depends on the `OutputNetwork` training option. To return the network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

`OutputNetwork` — Network to return when training completes
`'last-iteration'` (default) | `'best-validation-loss'`

Network to return when training completes, specified as one of the following:

- `'last-iteration'` – Return the network corresponding to the last training iteration.
- `'best-validation-loss'` – Return the network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify `'ValidationData'`.

Solver Options

`InitialLearnRate` — Initial learning rate
positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.

`LearnRateScheduleSettings` — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure. `LearnRateScheduleSettings` has the field `Method`, which specifies the type of method for adjusting the learning rate. The possible methods are:

- `'none'` — The learning rate is constant throughout training.
- `'piecewise'` — The learning rate drops periodically during training.

If `Method` is `'piecewise'`, then `LearnRateScheduleSettings` contains two more fields:

- `DropRateFactor` — The multiplicative factor by which the learning rate drops during training
- `DropPeriod` — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using `trainingOptions`.

Data Types: `struct`

`L2Regularization` — Factor for L_2 regularizer
nonnegative scalar

Factor for L_2 regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the L_2 regularizer for network layers with learnable parameters.

`Momentum` — Contribution of previous gradient step
scalar from 0 to 1

Contribution of the gradient step from the previous iteration to the current iteration of the training, specified as a scalar value from 0 to 1. A value of 0 means no contribution from the previous step, whereas a value of 1 means maximal contribution from the previous step. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1368.

`BatchNormalizationStatistics` — Mode to evaluate statistics in batch normalization layers
`'population'` (default) | `'moving'`

Mode to evaluate the statistics in batch normalization layers, specified as one of the following:

- `'population'` – Use the population statistics. After training, the software finalizes the statistics by passing through the training data once more and uses the resulting mean and variance.
- `'moving'` – Approximate the statistics during training using a running estimate given by update steps

  $$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu$$

  $$\sigma^{2*} = \lambda_{\sigma^2}\widehat{\sigma^2} + (1-\lambda_{\sigma^2})\sigma^2$$

  where μ^* and σ^{2*} denote the updated mean and variance, respectively, λ_μ and λ_{σ^2} denote the mean and variance decay values, respectively, $\widehat{\mu}$ and $\widehat{\sigma^2}$ denote the mean and variance of the layer input, respectively, and μ and σ^2 denote the latest values of the moving mean and variance values, respectively. After training, the software uses the most recent value of the moving mean and variance statistics. This option supports CPU and single GPU training only.

Gradient Clipping

`GradientThreshold` — Gradient threshold
positive scalar | `Inf`

Positive threshold for the gradient, specified as positive scalar or `Inf`. When the gradient exceeds the value of `GradientThreshold`, then the gradient is clipped according to `GradientThresholdMethod`.

`GradientThresholdMethod` — Gradient threshold method
`'l2norm'` | `'global-l2norm'` | `'absolutevalue'`

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- `'l2norm'` — If the L_2 norm of the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the gradient so that the L_2 norm equals `GradientThreshold`.
- `'global-l2norm'` — If the global L_2 norm, L, is larger than `GradientThreshold`, then scale all gradients by a factor of `GradientThreshold`/L. The global L_2 norm considers all learnable parameters.
- `'absolute-value'` — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than `GradientThreshold`, then scale the partial derivative to have magnitude equal to `GradientThreshold` and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-1370.

`ResetInputNormalization` — Option to reset input layer normalization
`true` (default) | `false`

Option to reset input layer normalization, specified as one of the following:

- `true` – Reset the input layer normalization statistics and recalculate them at training time.
- `false` – Calculate normalization statistics at training time when they are empty.

Sequence Options

`SequenceLength` — Option to pad or truncate sequences
`'longest'` | `'shortest'` | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- `'longest'` — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences 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, try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingDirection` — Direction of padding or truncation
`'right'` (default) | `'left'`

Direction of padding or truncation, specified as one of the following:

- `'right'` — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- `'left'` — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

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'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see "Sequence Padding, Truncation, and Splitting".

`SequencePaddingValue` — Value to pad sequences
scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is `'longest'` or a positive integer. Do not pad sequences with `NaN`, because doing so can propagate errors throughout the network.

Hardware Options

`ExecutionEnvironment` — Hardware resource for training network
`'auto'` | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training network, specified as one of the following:

- `'auto'` — Use a GPU if one is available. Otherwise, use the CPU.
- `'cpu'` — Use the CPU.
- `'gpu'` — Use the GPU.
- `'multi-gpu'` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- `'parallel'` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

`'gpu'`, `'multi-gpu'`, and `'parallel'` options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Specify the execution environment using `trainingOptions`.

Data Types: `char` | `string`

`WorkerLoad` — Parallel worker load division
scalar from 0 to 1 | positive integer | numeric vector

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the `ExecutionEnvironment` value equals `'multi-gpu'` or `'parallel'`.

Checkpoints

`CheckpointPath` — Path for saving checkpoint networks
character vector

Path where checkpoint networks are saved, specified as a character vector.

Data Types: `char`

`OutputFcn` — Output functions
function handle | cell array of function handles

Output functions to call during training, specified as a function handle or cell array of function handles. `trainNetwork` calls the specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

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. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training” .

Data Types: `function_handle` | `cell`

Examples

Specify Training Options

Create a set of options for training a network using stochastic gradient descent with momentum. Reduce the learning rate by a factor of 0.2 every 5 epochs. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Turn on the training progress plot.

```
options = trainingOptions('sgdm', ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.2, ...
    'LearnRateDropPeriod',5, ...
    'MaxEpochs',20, ...
    'MiniBatchSize',64, ...
    'Plots','training-progress')

options =
  TrainingOptionsSGDM with properties:

                      Momentum: 0.9000
                 InitialLearnRate: 0.0100
                LearnRateSchedule: 'piecewise'
              LearnRateDropFactor: 0.2000
              LearnRateDropPeriod: 5
                 L2Regularization: 1.0000e-04
          GradientThresholdMethod: 'l2norm'
                GradientThreshold: Inf
                        MaxEpochs: 20
                    MiniBatchSize: 64
                          Verbose: 1
                  VerboseFrequency: 50
                    ValidationData: []
               ValidationFrequency: 50
                ValidationPatience: Inf
                          Shuffle: 'once'
                   CheckpointPath: ''
             ExecutionEnvironment: 'auto'
                       WorkerLoad: []
                        OutputFcn: []
                            Plots: 'training-progress'
                   SequenceLength: 'longest'
              SequencePaddingValue: 0
          SequencePaddingDirection: 'right'
              DispatchInBackground: 0
           ResetInputNormalization: 1
    BatchNormalizationStatistics: 'population'
                  OutputNetwork: 'last-iteration'
```

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2016a

trainNetwork

Train deep learning neural network

Syntax

```
net = trainNetwork(images,layers,options)
net = trainNetwork(images,responses,layers,options)

net = trainNetwork(sequences,layers,options)
net = trainNetwork(sequences,responses,layers,options)

net = trainNetwork(features,layers,options)
net = trainNetwork(features,responses,layers,options)

[net,info] = trainNetwork( ___ )
```

Description

For classification and regression tasks, you can train various types of neural networks using the `trainNetwork` function.

For example, you can train:

- a convolutional neural network (ConvNet, CNN) for image data
- a recurrent neural network (RNN) such as a long short-term memory (LSTM) or a gated recurrent unit (GRU) network for sequence and time-series data
- a multilayer perceptron (MLP) network for numeric feature data

You can train on either a CPU or a GPU. For image classification and image regression, you can train a single network in parallel using multiple GPUs or a local or remote parallel pool. Training on a GPU or in parallel requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). To specify training options, including options for the execution environment, use the `trainingOptions` function.

When training a neural network, you can specify the predictors and responses as a single input or in two separate inputs.

`net = trainNetwork(images,layers,options)` trains the neural network specified by `layers` for image classification and regression tasks using the images and responses specified by `images` and the training options defined by `options`.

`net = trainNetwork(images,responses,layers,options)` trains using the images specified by `images` and responses specified by `responses`.

`net = trainNetwork(sequences,layers,options)` trains a neural network for sequence or time-series classification and regression tasks (for example, an LSTM or GRU network) using the sequences and responses specified by `sequences`.

`net = trainNetwork(sequences,responses,layers,options)` trains using the sequences specified by `sequences` and responses specified by `responses`.

`net = trainNetwork(features,layers,options)` trains a neural network for feature classification or regression tasks (for example, a multilayer perceptron (MLP) network) using the feature data and responses specified by `features`.

`net = trainNetwork(features,responses,layers,options)` trains using the feature data specified by `features` and responses specified by `responses`.

`[net,info] = trainNetwork( ___ )` also returns information on the training using any of the previous syntaxes.

Examples

Train Network for Image Classification

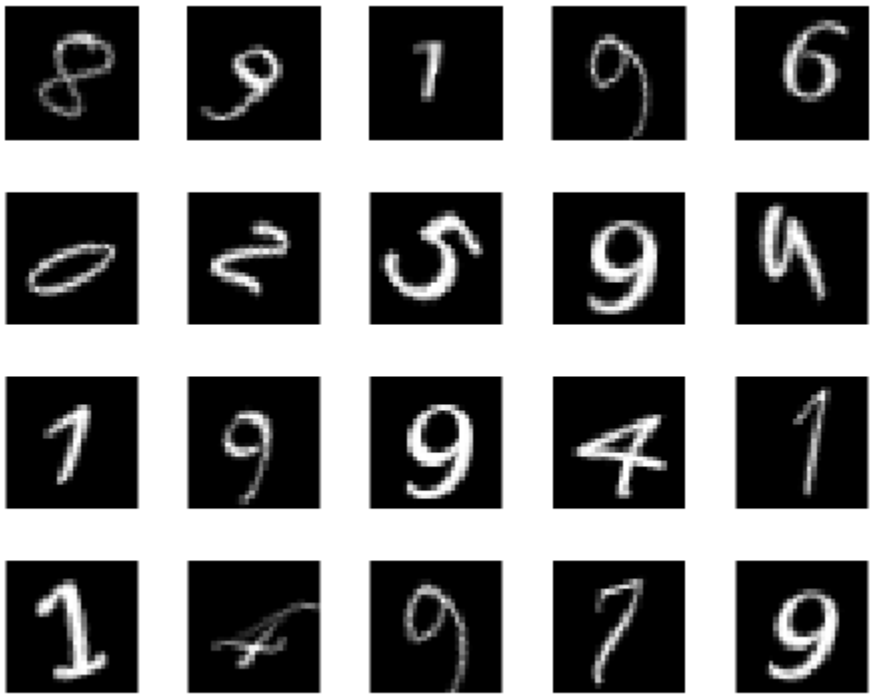
Load the data as an `ImageDatastore` object.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

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.

Display some of the images in the datastore.

```
figure
numImages = 10000;
perm = randperm(numImages,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
    drawnow;
end
```



Divide the datastore so that each category in the training set has 750 images and the testing set has the remaining images from each label.

```
numTrainingFiles = 750;
[imdsTrain,imdsTest] = splitEachLabel(imds,numTrainingFiles,'randomize');
```

`splitEachLabel` splits the image files in `digitData` into two new datastores, `imdsTrain` and `imdsTest`.

Define the convolutional neural network architecture.

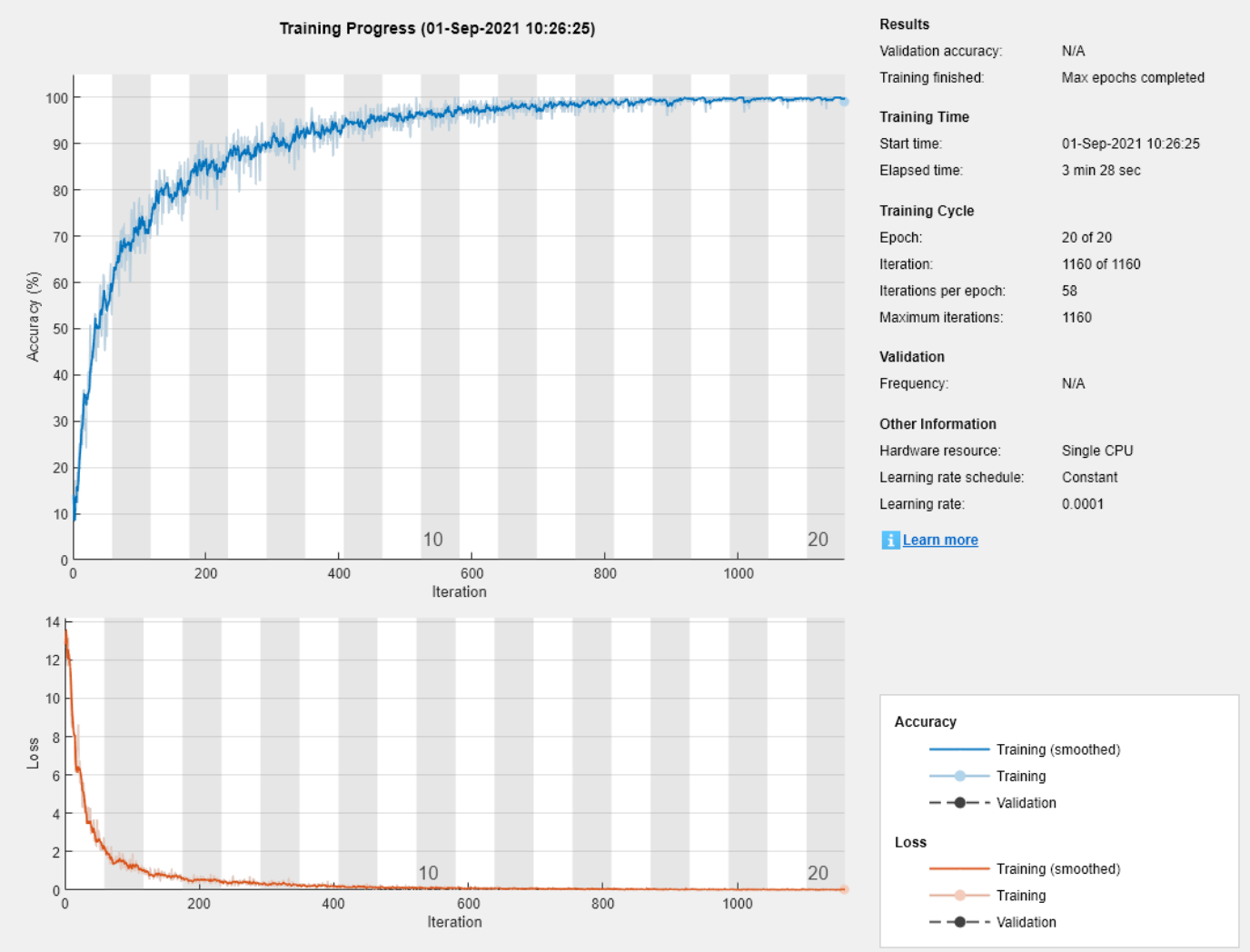
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.0001.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',20,...
    'InitialLearnRate',1e-4, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(imdsTrain,layers,options);
```



Run the trained network on the test set, which was not used to train the network, and predict the image labels (digits).

```
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy. The accuracy is the ratio of the number of true labels in the test data matching the classifications from `classify` to the number of images in the test data.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9412
```

Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

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

`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 width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

```
idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,:,idx);
XTrain(:,:,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Create an `imageDataAugmenter` object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

```
imageAugmenter = imageDataAugmenter( ...
    'RandRotation',[-20,20], ...
    'RandXTranslation',[-3 3], ...
    'RandYTranslation',[-3 3])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-20 20]
           RandScale: [1 1]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [-3 3]
    RandYTranslation: [-3 3]
```

Create an `augmentedImageDatastore` object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

Specify the convolutional neural network architecture.

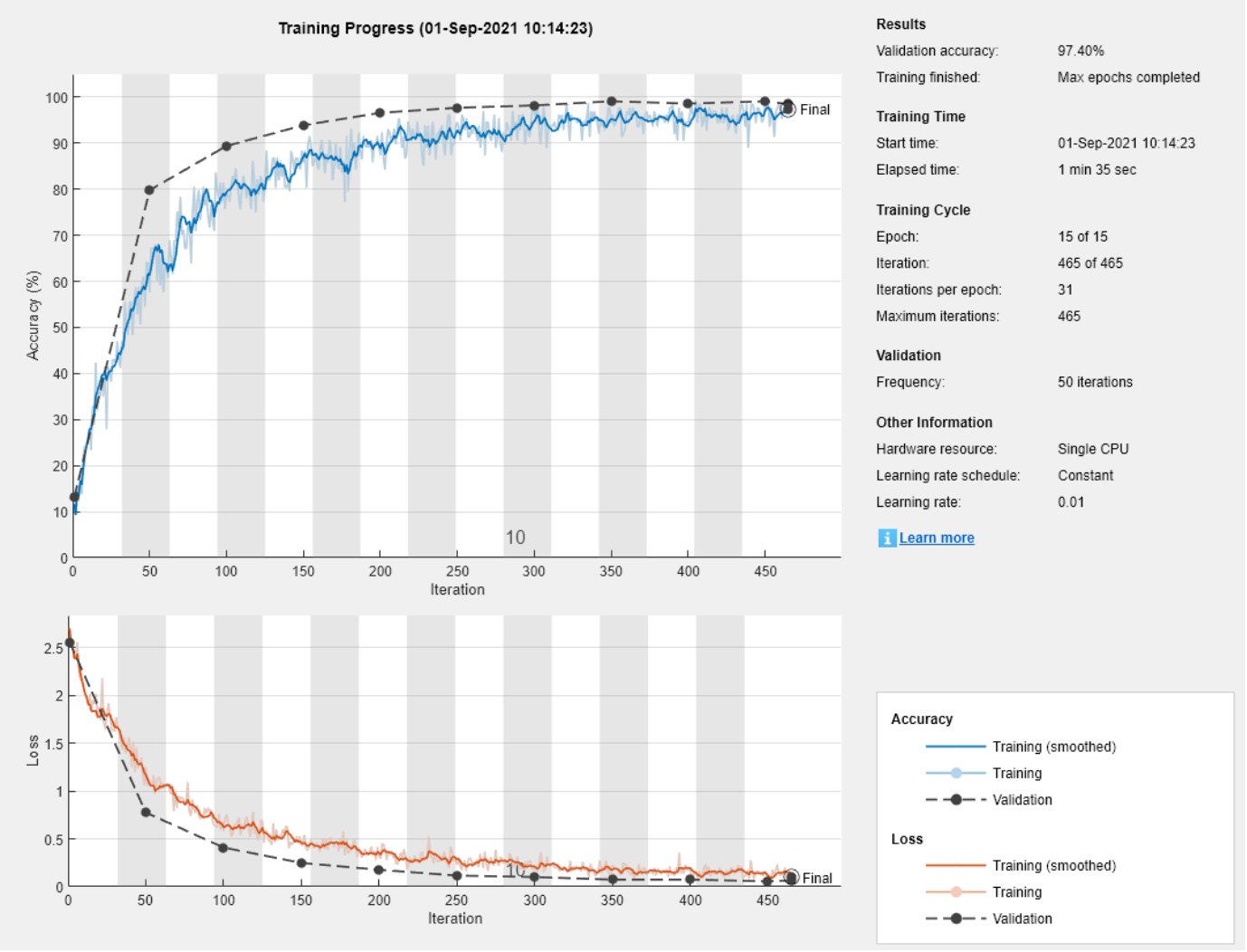
```
layers = [
    imageInputLayer(imageSize)

    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 training options for stochastic gradient descent with momentum.

```
opts = trainingOptions('sgdm', ...
    'MaxEpochs',15, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation});
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```
net = trainNetwork(augimds,layers,opts);
```



Train Network for Image Regression

Load the sample data, which consists of synthetic images of handwritten digits. The third output contains the corresponding angles in degrees by which each image has been rotated.

Load the training images as 4-D arrays using `digitTrain4DArrayData`. The output `XTrain` is a 28-by-28-by-1-by-5000 array, where:

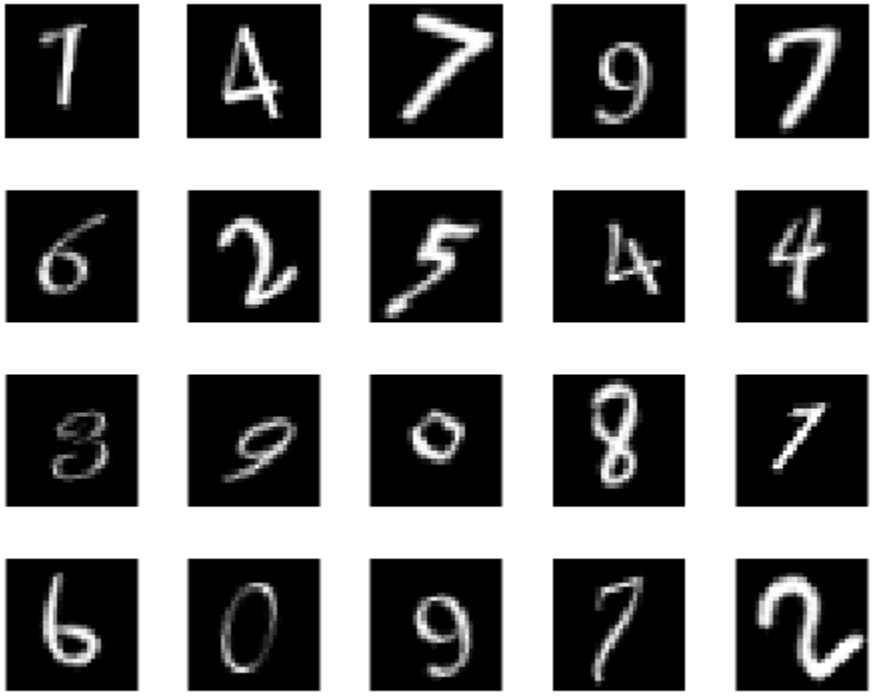
- 28 is the height and width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` contains the rotation angles in degrees.

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

Display 20 random training images using `imshow`.

```
figure
numTrainImages = numel(YTrain);
idx = randperm(numTrainImages,20);
for i = 1:numel(idx)
    subplot(4,5,i)
    imshow(XTrain(:,:,:,idx(i)))
    drawnow;
end
```



Specify the convolutional neural network architecture. For regression problems, include a regression layer at the end of the network.

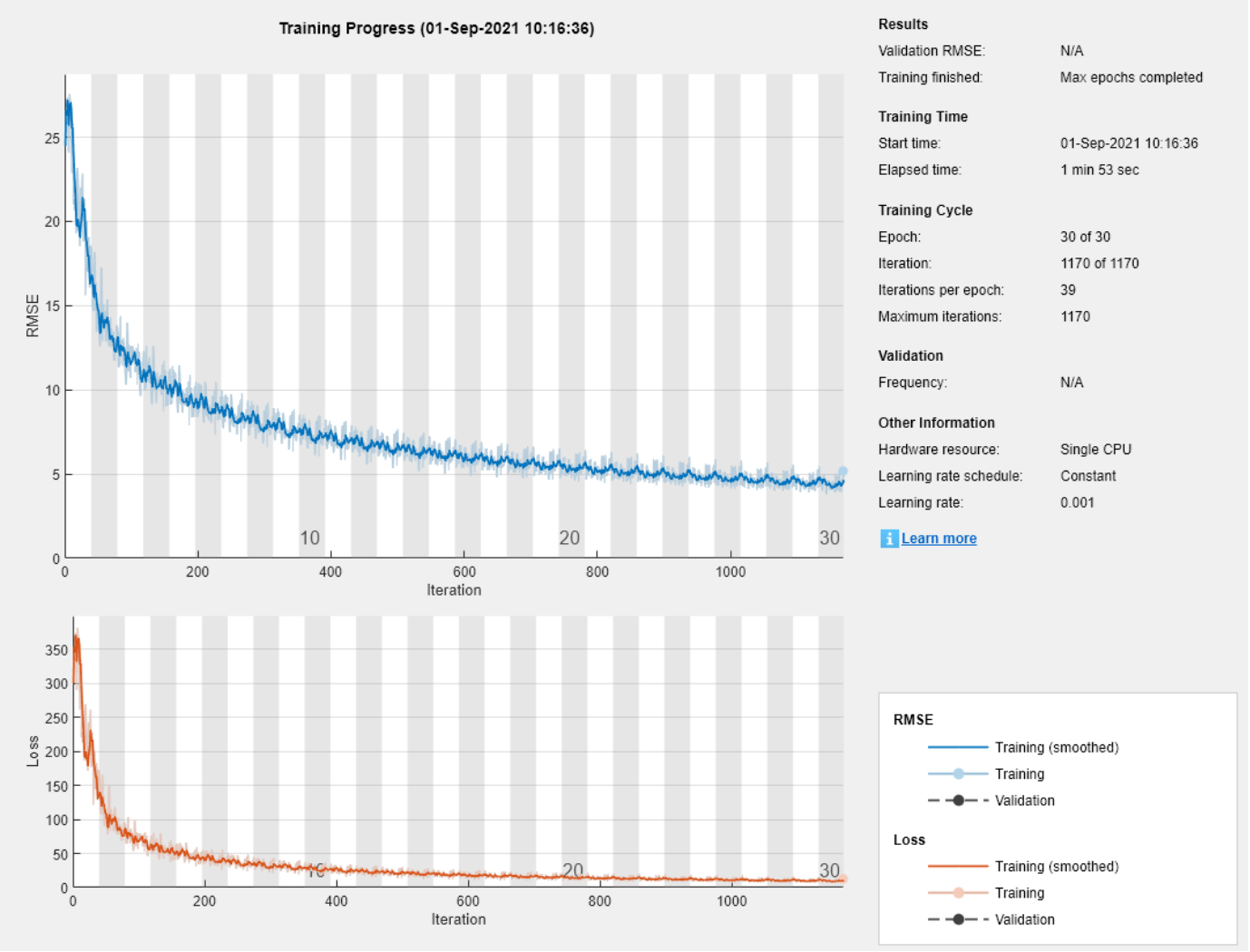
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer];
```

Specify the network training options. Set the initial learn rate to 0.001.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

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



Test the performance of the network by evaluating the prediction accuracy of the test data. Use `predict` to predict the angles of rotation of the validation images.

```
[XTest,~,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);
```

Evaluate the performance of the model by calculating the root-mean-square error (RMSE) of the predicted and actual angles of rotation.

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

```
rmse = single
    6.0709
```

Train Network for Sequence Classification

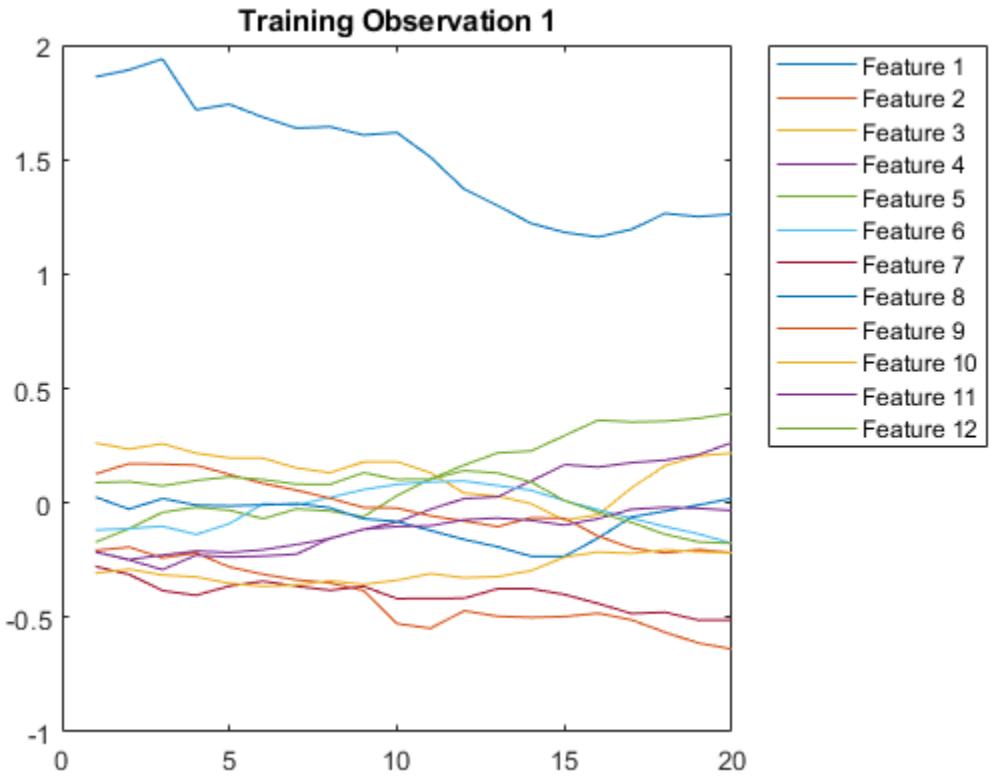
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with 12 features corresponding to LPC cepstrum coefficients. `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;
```

Visualize the first time series in a plot. Each line corresponds to a feature.

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



Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 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.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
```

```
layers = [ ...
    sequenceInputLayer(inputSize)
    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
```

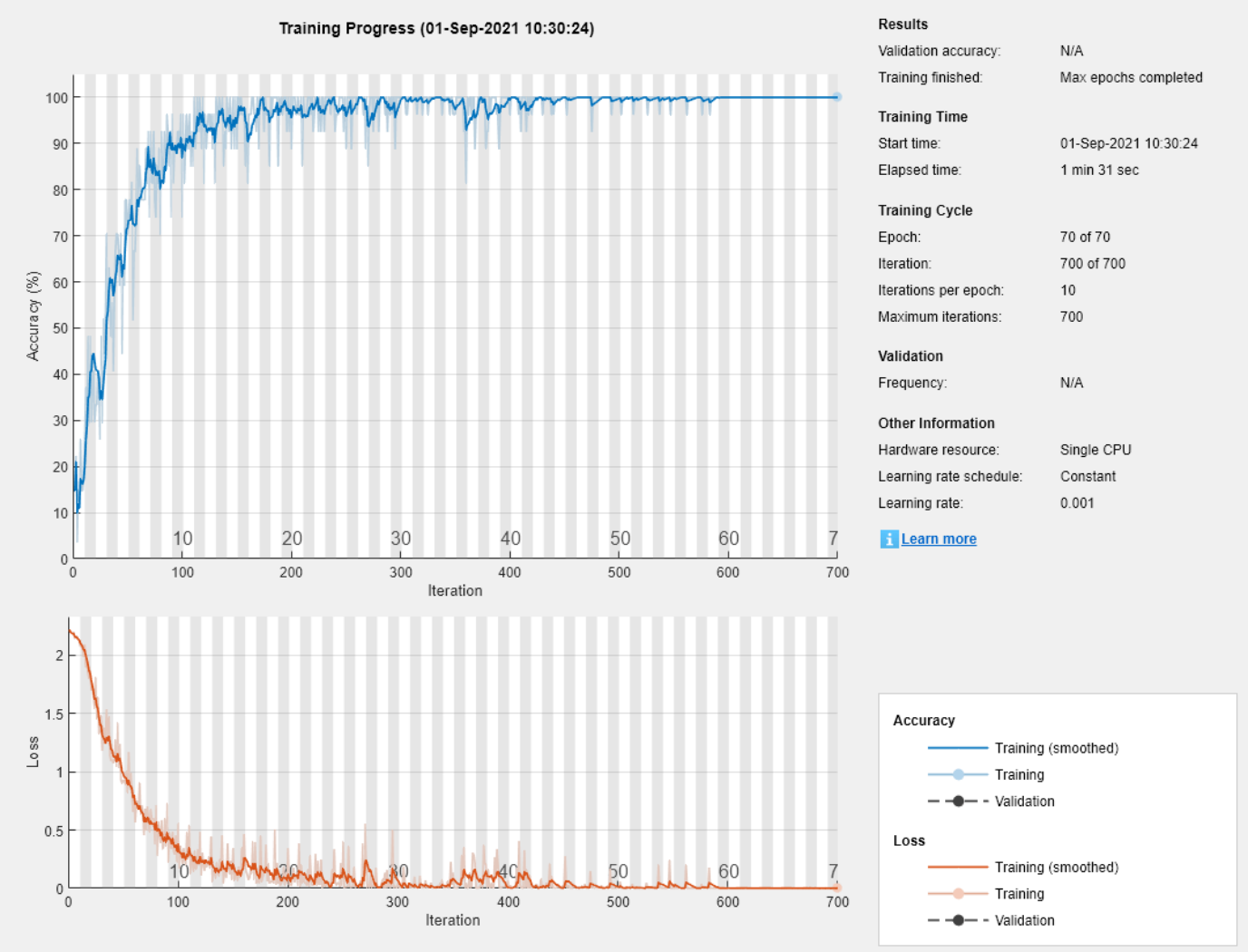
Specify the training options. Specify the solver as `'adam'` and `'GradientThreshold'` as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 70.

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).

```
maxEpochs = 70;
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',1, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the LSTM network with the specified training options.

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



Load the test set and classify the sequences into speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Classify the test data. Specify the same mini-batch size used for training.

```
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9541
```

Train Network with Numeric Features

If you have a data set of numeric features (for example a collection of numeric data without spatial or time dimensions), then you can train a deep learning network using a feature input layer.

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

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

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

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

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

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

Loop over the categorical input variables. For each variable:

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

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

Split the vectors into separate columns using the splitvars function.

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

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

```
head(tbl)
```

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

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

View the class names of the data set.

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

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

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

Determine the number of observations for each partition.

```
numObservations = size(tbl,1);
numObservationsTrain = floor(0.85*numObservations);
numObservationsTest = numObservations - numObservationsTrain;
```

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

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

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

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

Define a network with a feature input layer and specify the number of features. Also, configure the input layer to normalize the data using Z-score normalization.

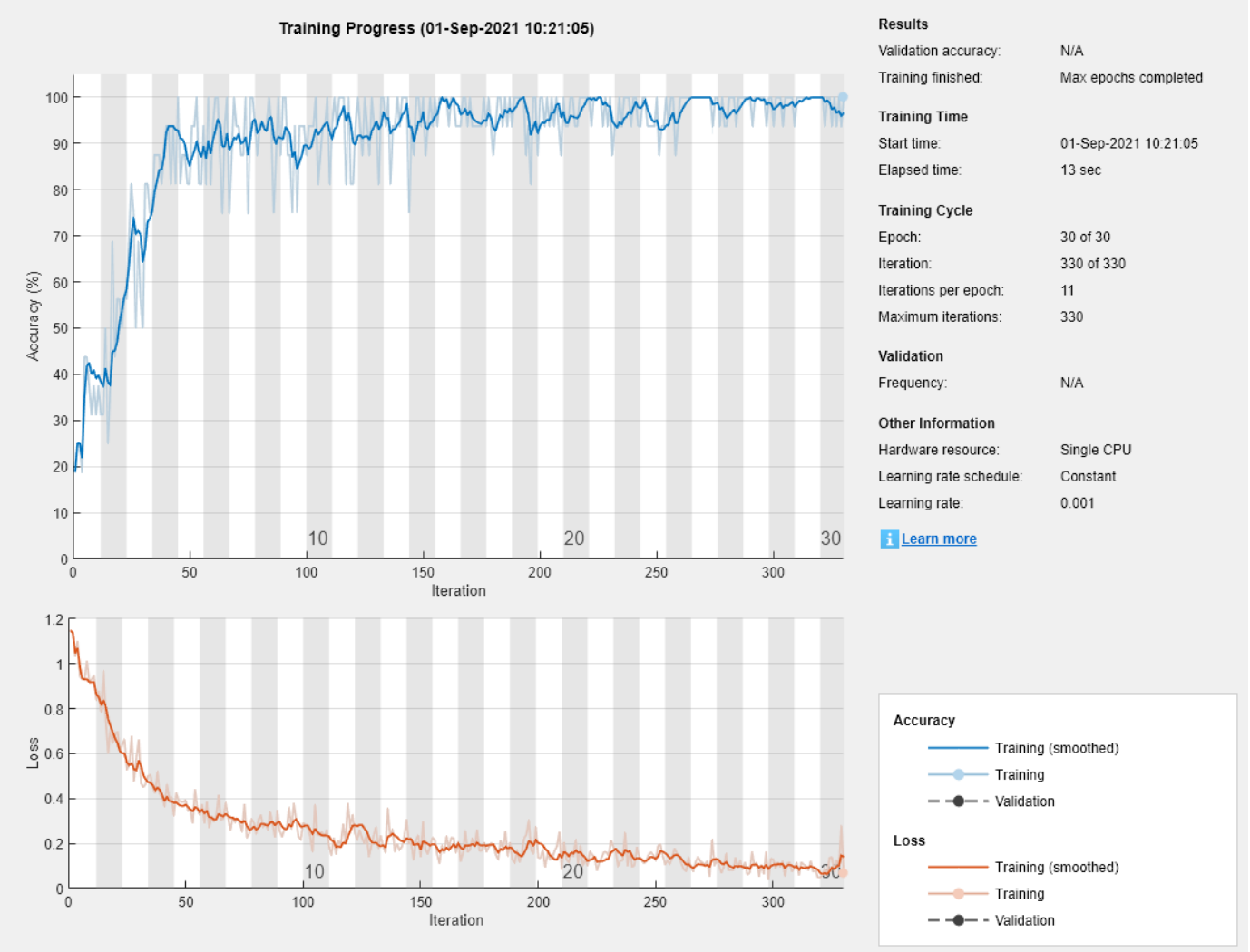
```
numFeatures = size(tbl,2) - 1;
numClasses = numel(classNames);

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

Specify the training options.

```
miniBatchSize = 16;

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

Train the network using the architecture defined by `layers`, the training data, and the training options.

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



Predict the labels of the test data using the trained network and calculate the accuracy. The accuracy is the proportion of the labels that the network predicts correctly.

```
YPred = classify(net,tblTest,'MiniBatchSize',miniBatchSize);
YTest = tblTest{:,labelName};

accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9688
```

Input Arguments

`images` — Image data

datastore | numeric array | table

Image data, specified as one of the following:

Data Type		Description	Example Usage
Datastore	`ImageDatastore`	Datastore that contains images and labels.	Train image classification neural network with images saved on disk, where the images are the same size. When the images are different sizes, use an `AugmentedImageDatastore` object. `ImageDatastore` objects support image classification tasks only. To use image datastores for regression networks, create a transformed or combined datastore that contains the images and responses using the `transform` and `combine` functions, respectively.
	`AugmentedImageDatastore`	Datastore that applies random affine geometric transformations, including resizing, rotation, reflection, shear, and translation.	• Train image classification neural network with images saved on disk, where the images are different sizes. • Train image classification neural network and generate new data using augmentations.

Data Type		Description	Example Usage
	`TransformedDatastore`	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function.	• Train image regression neural network. • Train networks with multiple inputs. • Transform outputs of datastores not supported by the `trainNetwork` function to the have the required format. • Apply custom transformations to datastore output.
	`CombinedDatastore`	Datastore that reads from two or more underlying datastores.	• Train image regression neural network. • Train networks with multiple inputs. • Combine predictors and responses from different data sources.
	`PixelLabelImageDatastore`	Datastore that applies identical affine geometric transformations to images and corresponding pixel labels.	Train neural network for semantic segmentation.
	`RandomPatchExtractionDatastore`	Datastore that extracts pairs of random patches from images or pixel label images and optionally applies identical random affine geometric transformations to the pairs.	Train neural network for object detection.
	`DenoisingImageDatastore`	Datastore that applies randomly generated Gaussian noise.	Train neural network for image denoising.

Data Type		Description	Example Usage
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data.	Train neural network using data in a format that other datastores do not support. For details, see "Develop Custom Mini-Batch Datastore".
Numeric array		Images specified as numeric array. If you specify images as a numeric array, then you must also specify the `responses` argument.	Train neural network using data that fits in memory and does not require additional processing like augmentation.
Table		Images specified as a table. If you specify images as a table, then you can also specify which columns contain the responses using the `responses` argument.	Train neural network using data stored in a table.

For networks with multiple inputs, the datastore must be a `TransformedDatastore` or `CombinedDatastore` object.

Tip For sequences of images, for example video data, use the `sequences` input argument.

Datastore

Datastores read mini-batches of images and responses. Datastores are best suited when you have data that does not fit in memory or when you want to apply augmentations or transformations to the data.

The list below lists the datastores that are directly compatible with `trainNetwork` for image data.

- `ImageDatastore`
- `AugmentedImageDatastore`
- `CombinedDatastore`
- `TransformedDatastore`
- `PixelLabelImageDatastore`
- `RandomPatchExtractionDatastore`
- `DenoisingImageDatastore`
- Custom mini-batch datastore

For example, you can create an image datastore using the `imageDatastore` function and use the names of the folders containing the images as labels by setting the `'LabelSource'` option to `'foldernames'`. Alternatively, you can specify the labels manually using the `Labels` property of the image datastore.

Note that `ImageDatastore` objects allow for batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` objects do not prefetch.

Tip Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.

You can use other built-in datastores for training deep learning networks by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the format required by `trainNetwork`.

For networks with multiple inputs, the datastore must be a `TransformedDatastore` or `CombinedDatastore` object.

The required format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output

Single input layer	Table or cell array with two columns.
	
	The first and second columns specify the predictors and responses, respectively.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom mini-batch datastores must output tables.
		Table for network with one input and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×2 table`
		
		`Predictors Response`
		`{224×224×3 double} 2`
		`{224×224×3 double} 7`
		`{224×224×3 double} 9`
		`{224×224×3 double} 9`

		Cell array for network with one input and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×2 cell array`
		
		`{224×224×3 double} {[2]}`
		`{224×224×3 double} {[7]}`
		`{224×224×3 double} {[9]}`
		`{224×224×3 double} {[9]}`

Network Architecture	Datastore Output	Example Output

Multiple input layers	Cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs.
	
	The first `numInputs` columns specify the predictors for each input and the last column specifies the responses.
	
	The order of inputs is given by the `InputNames` property of the layer graph `layers`.
		Cell array for network with two inputs and one output.
		
		`data = read(ds)`
		
		`data =`
		
		`4×3 cell array`
		
		`{224×224×3 double}  {128×128×3 do`
		`{224×224×3 double}  {128×128×3 do`
		`{224×224×3 double}  {128×128×3 do`
		`{224×224×3 double}  {128×128×3 do`

The format of the predictors depends on the type of data.

Data	Format
2-D images	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively.
3-D images	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively.

For predictors returned in tables, the elements must contain a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing the numeric array.

The format of the responses depends on the type of task.

Task	Response Format
Image classification	Categorical scalar
Image regression	• Numeric scalar • Numeric vector • 3-D numeric array representing a 2-D image • 4-D numeric array representing a 3-D image

For responses returned in tables, the elements must be a categorical scalar, a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

For more information, see “Datastores for Deep Learning”.

Numeric Array

For data that fits in memory and does not require additional processing like augmentation, you can specify a data set of images as a numeric array. If you specify images as a numeric array, then you must also specify the `responses` argument.

The size and shape of the numeric array depends on the type of image data.

Data	Format
2-D images	*h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.

Table

As an alternative to datastores or numeric arrays, you can also specify images and responses in a table. If you specify images as a table, then you can also specify which columns contain the responses using the `responses` argument.

When specifying images and responses in a table, each row in the table corresponds to an observation.

For image input, the predictors must be in the first column of the table, specified as one of the following:

- Absolute or relative file path to an image, specified as a character vector
- 1-by-1 cell array containing a *h*-by-*w*-by-*c* numeric array representing a 2-D image, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively.

The format of the responses depends on the type of task.

Task	Response Format
Image classification	Categorical scalar
Image regression	• Numeric scalar • Two or more columns of scalar values • 1-by-1 cell array containing a *h*-by-*w*-by-*c* numeric array representing a 2-D image • 1-by-1 cell array containing a *h*-by-*w*-by-*d*-by-*c* numeric array representing a 3-D image

For neural networks with image input, if you do not specify `responses`, then the function, by default, uses the first column of `tbl` for the predictors and the subsequent columns as responses.

Tip If the predictors or the responses contains `NaN`s, then they are propagated through the network during training. In these cases, the training usually fails to converge.

Tip For regression tasks, normalizing the responses often helps to stabilize and speed up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

sequences — Sequence or time series data
datastore | cell array of numeric arrays | numeric array

Sequence or time series data, specified as one of the following:

Data Type		Description	Example Usage
Datastore	`TransformedDatastore`	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function.	• Transform outputs of datastores not supported by `trainNetwork` to the have the required format. • Apply custom transformations to datastore output.
	`CombinedDatastore`	Datastore that reads from two or more underlying datastores.	Combine predictors and responses from different data sources.
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data.	Train neural network using data in a format that other datastores do not support. For details, see "Develop Custom Mini-Batch Datastore".
Numeric or cell array		A single sequence specified as a numeric array or a data set of sequences specified as cell array of numeric arrays. If you specify sequences as a numeric or cell array, then you must also specify the `responses` argument.	Train neural network using data that fits in memory and does not require additional processing like custom transformations.

Datastore

Datastores read mini-batches of sequences and responses. Datastores are best suited when you have data that does not fit in memory or when you want to apply transformations to the data.

The list below lists the datastores that are directly compatible with `trainNetwork` for sequence data.

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore

You can use other built-in datastores for training deep learning networks by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the table or cell array format required by `trainNetwork`. For example, you can transform and combine data read from in-memory arrays and CSV files using `ArrayDatastore` and `TabularTextDatastore` objects, respectively.

The datastore must return data in a table or cell array. Custom mini-batch datastores must output tables.

Datastore Output	Example Output

Table	`data = read(ds)`
	
	`data =`
	
	`  4×2 table`
	
	`  Predictors  Response`
	`  ____________  ________`
	
	`  {12×50 double}  2`
	`  {12×50 double}  7`
	`  {12×50 double}  9`
	`  {12×50 double}  9`

Cell array	`data = read(ds)`
	
	`data =`
	
	`  4×2 cell array`
	
	`  {12×50 double}  {[2]}`
	`  {12×50 double}  {[7]}`
	`  {12×50 double}  {[9]}`
	`  {12×50 double}  {[9]}`

The format of the predictors depend on the type of data.

Data	Format of Predictors
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.
1-D image sequence	*h*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.

Data	Format of Predictors
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.

For predictors returned in tables, the elements must contain a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

The format of the responses depends on the type of task.

Task	Format of Responses
Sequence-to-label classification	Categorical scalar
Sequence-to-one regression	Scalar
Sequence-to-vector regression	Numeric row vector
Sequence-to-sequence classification	• 1-by-*s* sequence of categorical labels, where *s* is the sequence length of the corresponding predictor sequence. • *h*-by-*w*-by-*s* sequence of categorical labels, where *h*, *w*, and *s* are the height, width, and sequence length of the corresponding predictor sequence, respectively. • *h*-by-*w*-by-*d*-by-*s* sequence of categorical labels, where *h*, *w*, *d*, and *s* are the height, width, depth, and sequence length of the corresponding predictor sequence, respectively.
Sequence-to-sequence regression	• *R*-by-*s* matrix, where *R* is the number of responses and *s* is the sequence length of the corresponding predictor sequence. • *h*-by-*w*-by-*R*-by-*s* sequence of numeric responses, where *R* is the number of responses , and *h*, *w*, and *s* are the height, width, and sequence length of the corresponding predictor sequence, respectively. • *h*-by-*w*-by-*d*-by-*R*-by-*s* sequence of numeric responses, where *R* is the number of responses , and *h*, *w*, *d*, and *s* are the height, width, depth, and sequence length of the corresponding predictor sequence, respectively.

For responses returned in tables, the elements must be a categorical scalar, a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

For more information, see “Datastores for Deep Learning”.

Numeric or Cell Array

For data that fits in memory and does not require additional processing like custom transformations, you can specify a single sequence as a numeric array or a data set of sequences as a cell array of numeric arrays. If you specify sequences as a cell or numeric array, then you must also specify the `responses` argument.

For cell array input, the cell array must be an *N*-by-1 cell array of numeric arrays, where *N* is the number of observations. The size and shape of the numeric array representing a sequence depends on the type of sequence data:ce

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
1-D image sequences	*h*-by-*c*-by-*s* arrays, where *h* and *c* correspond to the height and number of channels of the images, respectively, and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*s*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *s* is the sequence length.

Tip If the predictors or the responses contains `NaN`s, then they are propagated through the network during training. In these cases, the training usually fails to converge.

Tip For regression tasks, normalizing the responses often helps to stabilize and speed up training. For more information, see “Train Convolutional Neural Network for Regression”.

`features` — Feature data

datastore | numeric array | table

Feature data, specified as one of the following:

Data Type		Description	Example Usage
Datastore	`TransformedDatastore`	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function.	• Train networks with multiple inputs. • Transform outputs of datastores not supported by `trainNetwork` to the have the required format. • Apply custom transformations to datastore output.
	`CombinedDatastore`	Datastore that reads from two or more underlying datastores.	• Train networks with multiple inputs. • Combine predictors and responses from different data sources.
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data.	Train neural network using data in a format that other datastores do not support. For details, see “Develop Custom Mini-Batch Datastore”.
Table		Feature data specified as a table. If you specify features as a table, then you can also specify which columns contain the responses using the `responses` argument.	Train neural network using data stored in a table.
Numeric array		Feature data specified as numeric array. If you specify features as a numeric array, then you must also specify the `responses` argument.	Train neural network using data that fits in memory and does not require additional processing like custom transformations.

Datastore

Datastores read mini-batches of feature data and responses. Datastores are best suited when you have data that does not fit in memory or when you want to apply transformations to the data.

The list below lists the datastores that are directly compatible with `trainNetwork` for feature data.

- `CombinedDatastore`
- `TransformedDatastore`

- Custom mini-batch datastore

You can use other built-in datastores for training deep learning networks by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the table or cell array format required by `trainNetwork`. For more information, see “Datastores for Deep Learning”.

For networks with multiple inputs, the datastore must be a `TransformedDatastore` or `CombinedDatastore` object.

The datastore must return data in a table or a cell array. Custom mini-batch datastores must output tables. The format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output

Single input layer	Table or cell array with two columns.
	
	The first and second columns specify the predictors and responses, respectively.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom mini-batch datastores must output tables.
		Table for network with one input and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×2 table`
		
		`Predictors  Response`
		`{24×1 double}  2`
		`{24×1 double}  7`
		`{24×1 double}  9`
		`{24×1 double}  9`

		Cell array for network with one input and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×2 cell array`
		
		`{24×1 double}  {[2]}`
		`{24×1 double}  {[7]}`
		`{24×1 double}  {[9]}`
		`{24×1 double}  {[9]}`

Multiple input layers	Cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs.
	
	The first `numInputs` columns specify the predictors for each input and the last column specifies the responses.
	
	The order of inputs is given by the `InputNames` property of the layer graph `layers`.
		Cell array for network with two inputs and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×3 cell array`
		
		`{24×1 double}  {28×1 double}  {`
		`{24×1 double}  {28×1 double}  {`
		`{24×1 double}  {28×1 double}  {`
		`{24×1 double}  {28×1 double}  {`

The predictors must be *c*-by-1 column vectors, where *c* is the number of features.

The format of the responses depends on the type of task.

Task	Format of Responses
Classification	Categorical scalar
Regression	• Scalar • Numeric vector

For more information, see “Datastores for Deep Learning”.

Table

For feature data that fits in memory and does not require additional processing like custom transformations, you can specify feature data and responses as a table.

Each row in the table corresponds to an observation. The arrangement of predictors and responses in the table columns depends on the type of task.

Task	Predictors	Responses
Feature classification	Features specified in one or more columns as scalars. If you do not specify the `responses` argument, then the predictors must be in the first `numFeatures` columns of the table, where `numFeatures` is the number of features of the input data.	Categorical label
Feature regression		One or more columns of scalar values

For classification networks with feature input, if you do not specify the `responses` argument, then the function, by default, uses the first (`numColumns - 1`) columns of `tbl` for the predictors and the last column for the labels, where `numFeatures` is the number of features in the input data.

For regression networks with feature input, if you do not specify the `responseNames` argument, then the function, by default, uses the first `numFeatures` columns for the predictors and the subsequent columns for the responses, where `numFeatures` is the number of features in the input data.

Numeric Array

For feature data that fits in memory and does not require additional processing like custom transformations, you can specify feature data as a numeric array. If you specify feature data as a numeric array, then you must also specify the `responses` argument.

The numeric array must be an *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

Tip Normalizing the responses often helps to stabilize and speed up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

Tip Responses must not contain `NaN`s. If the predictor data contains `NaN`s, then they are propagated through the training. However, in most cases, the training fails to converge.

`responses` — Responses

categorical vector | numeric array | cell array of sequences | character vector | cell array of character vectors | string array

Responses.

When the input data is a numeric array of a cell array, specify the responses as one of the following.

- categorical vector of labels
- numeric array of numeric responses
- cell array of categorical or numeric sequences

When the input data is a table, you can optionally specify which columns of the table contains the responses as one of the following:

- character vector
- cell array of character vectors
- string array

When the input data is a numeric array or a cell array, then the format of the responses depends on the type of task.

Task		Format
Classification	Image classification	*N*-by-1 categorical vector of labels, where *N* is the number of observations.
	Feature classification	
	Sequence-to-label classification	
	Sequence-to-sequence classification	*N*-by-1 cell array of categorical sequences of labels, where *N* is the number of observations. Each sequence must have the same number of time steps as the corresponding predictor sequence. For sequence-to-sequence classification tasks with one observation, `sequences` can also be a vector. In this case, `Y` must be a categorical sequence of labels.

Task		Format
Regression	2-D image regression	• *N*-by-*R* matrix, where *N* is the number of images and *R* is the number of responses. • *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
	3-D image regression	• *N*-by-*R* matrix, where *N* is the number of images and *R* is the number of responses. • *h*-by-*w*-by-*d*-by-*c*-by-*N* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the images, respectively, and *N* is the number of images.
	Feature regression	*N*-by-*R* matrix, where *N* is the number of observations and *R* is the number of responses.
	Sequence-to-one regression	*N*-by-*R* matrix, where *N* is the number of sequences and *R* is the number of responses.

Task		Format
	Sequence-to-sequence regression	*N*-by-1 cell array of numeric sequences, where *N* is the number of sequences, with sequences given by one of the following: • *R*-by-*s* matrix, where *R* is the number of responses and *s* is the sequence length of the corresponding predictor sequence. • *h*-by-*w*-by-*R*-by-*s* array, where *h* and *w* are the height and width of the output, respectively, *R* is the number of responses, and *s* is the sequence length of the corresponding predictor sequence. • *h*-by-*w*-by-*d*-by-*R*-by-*s* array, where *h*, *w*, and *d* are the height, width, and depth of the output, respectively, *R* is the number of responses, and *s* is the sequence length of the corresponding predictor sequence. For sequence-to-sequence regression tasks with one observation, `sequences` can be a numeric array. In this case, `responses` must be a numeric array of responses.

Tip Normalizing the responses often helps to stabilize and speed up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

Tip Responses must not contain `NaN`s. If the predictor data contains `NaN`s, then they are propagated through the training. However, in most cases, the training fails to converge.

`layers` — Network layers

`Layer` array | `LayerGraph` object

Network layers, specified as a `Layer` array or a `LayerGraph` object.

To create a network with all layers connected sequentially, you can use a `Layer` array as the input argument. In this case, the returned network is a `SeriesNetwork` object.

A directed acyclic graph (DAG) network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG network, specify the network architecture as a `LayerGraph` object and then use that layer graph as the input argument to `trainNetwork`.

For a list of built-in layers, see “List of Deep Learning Layers”.

`options` — Training options
`TrainingOptionsSGDM` | `TrainingOptionsRMSProp` | `TrainingOptionsADAM`

Training options, specified as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object returned by the `trainingOptions` function.

Output Arguments

`net` — Trained network
`SeriesNetwork` object | `DAGNetwork` object

Trained network, returned as a `SeriesNetwork` object or a `DAGNetwork` object.

If you train the network using a `Layer` array, then `net` is a `SeriesNetwork` object. If you train the network using a `LayerGraph` object, then `net` is a `DAGNetwork` object.

`info` — Training information
structure

Training information, returned as a structure, where each field is a scalar or a numeric vector with one element per training iteration.

For classification tasks, `info` contains the following fields:

- `TrainingLoss` — Loss function values
- `TrainingAccuracy` — Training accuracies
- `ValidationLoss` — Loss function values
- `ValidationAccuracy` — Validation accuracies
- `BaseLearnRate` — Learning rates
- `FinalValidationLoss` — Validation loss of returned network
- `FinalValidationAccuracy` — Validation accuracy of returned network
- `OutputNetworkIteration` — Iteration number of returned network

For regression tasks, `info` contains the following fields:

- `TrainingLoss` — Loss function values
- `TrainingRMSE` — Training RMSE values
- `ValidationLoss` — Loss function values
- `ValidationRMSE` — Validation RMSE values
- `BaseLearnRate` — Learning rates
- `FinalValidationLoss` — Validation loss of returned network
- `FinalValidationRMSE` — Validation RMSE of returned network
- `OutputNetworkIteration` — Iteration number of returned network

The structure only contains the fields `ValidationLoss`, `ValidationAccuracy`, `ValidationRMSE` , `FinalValidationLoss` , `FinalValidationAccuracy`, and `FinalValidationRMSE` when `options` specifies validation data. The `ValidationFrequency` training option determines which iterations the software calculates validation metrics. The final validation metrics are scalar. The other fields of the structure are row vectors, where each element corresponds to a training iteration. For iterations when the software does not calculate validation metrics, the corresponding values in the structure are `NaN`.

For networks containing batch normalization layers, if the `BatchNormalizationStatistics` training option is `'population'` 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. For more information, see `batchNormalizationLayer`.

More About

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”.

Floating-Point Arithmetic

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

Compatibility Considerations

Support for specifying tables of MAT file paths will be removed
Warns starting in R2021a

When specifying sequence data for the `trainNetwork` function, support for specifying tables of MAT file paths will be removed in a future release.

To train networks with sequences that do not fit in memory, use a datastore. You can use any datastore to read your data and then use the `transform` function to transform the datastore output to the format the `trainNetwork` function requires. For example, you can read data using a `FileDatastore` or `TabularTextDatastore` object then transform the output using the `transform` function.

`trainNetwork` automatically stops training when loss is `NaN`
Behavior changed in R2021b

When you train a network using the `trainNetwork` function, training automatically stops when the loss is `NaN`. Usually, a loss value of `NaN` introduces `NaN` values to the network learnable parameters, which in turn can cause the network to fail to train or to make valid predictions. This change helps identify issues with the network before training completes.

In previous releases, the network continues to train when the loss is `NaN`.

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

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run computation in parallel, set the `'ExecutionEnvironment'` option to `'multi-gpu'` or `'parallel'`.

Use `trainingOptions` to set the `'ExecutionEnvironment'` and supply the `options` to `trainNetwork`. If you do not set `'ExecutionEnvironment'`, then `trainNetwork` runs on a GPU if available.

For details, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- To prevent out-of-memory errors, recommended practice is not to move large sets of training data onto the GPU. Instead, train your network on a GPU by using `trainingOptions` to set the `'ExecutionEnvironment'` to `"auto"` or `"gpu"` and supply the `options` to `trainNetwork`.
- When input data is a `gpuArray`, a cell array or table containing `gpuArray` data, or a datastore that returns `gpuArray` data, `"ExecutionEnvironment"` option must be `"auto"` or `"gpu"`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`trainingOptions` | `SeriesNetwork` | `DAGNetwork` | `LayerGraph` | `classify` | `predict` | `analyzeNetwork` | `assembleNetwork` | **Deep Network Designer**

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using Pretrained Network”
“Train Convolutional Neural Network for Regression”
“Sequence Classification Using Deep Learning”
“Deep Learning in MATLAB”
“Define Custom Deep Learning Layers”
“List of Deep Learning Layers”

Introduced in R2016a

transposedConv2dLayer

Transposed 2-D convolution layer

Syntax

```
layer = transposedConv2dLayer(filterSize,numFilters)
layer = transposedConv2dLayer(filterSize,numFilters,Name,Value)
```

Description

A transposed 2-D convolution layer upsamples feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

`layer = transposedConv2dLayer(filterSize,numFilters)` returns a transposed 2-D convolution layer and sets the `filterSize` and `numFilters` properties.

`layer = transposedConv2dLayer(filterSize,numFilters,Name,Value)` returns a transposed 2-D convolutional layer and specifies additional options using one or more name-value pair arguments.

Examples

Create Transposed Convolutional Layer

Create a transposed convolutional layer with 96 filters, each with a height and width of 11. Use a stride of 4 in the horizontal and vertical directions.

```
layer = transposedConv2dLayer(11,96,'Stride',4);
```

Input Arguments

`filterSize` — Height and width of filters

vector of two positive integers

Height and width of the filters, specified as a vector of two positive integers `[h w]`, where `h` is the height and `w` is the width. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

If you set `FilterSize` using an input argument, then you can specify `FilterSize` as scalar to use the same value for both dimensions.

Example: `[5 5]` specifies filters of height 5 and width 5.

`numFilters` — Number of filters

positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: `96`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Cropping',1`

Transposed Convolution

`Stride` — Up-sampling factor

`1` (default) | vector of two positive integers | positive integer

Up-sampling factor of the input, specified as one of the following:

- A vector of two positive integers `[a b]`, where `a` is the vertical stride and `b` is the horizontal stride.
- A positive integer the corresponds to both the vertical and horizontal stride.

Example: `'Stride',[2 1]`

`Cropping` — Output size reduction

`0` (default) | `'same'` | nonnegative integer | vector of two nonnegative integers

Output size reduction, specified as one of the following:

- `'same'` – Set the cropping so that the output size equals `inputSize .* Stride`, where `inputSize` is the height and width of the layer input. If you set the `'Cropping'` option to `'same'`, then the software automatically sets the `CroppingMode` property of the layer to `'same'`.

  The software trims an equal amount from the top and bottom, and the left and right, if possible. If the vertical crop amount has an odd value, then the software trims an extra row from the bottom. If the horizontal crop amount has an odd value, then the software trims an extra column from the right.
- A positive integer – Crop the specified amount of data from all the edges.
- A vector of nonnegative integers `[a b]` - Crop `a` from the top and bottom and crop `b` from the left and right.
- A vector `[t b l r]` - Crop `t`, `b`, `l`, `r` from the top, bottom, left, and right of the input, respectively.

If you set the `'Cropping'` option to a numeric value, then the software automatically sets the `CroppingMode` property of the layer to `'manual'`.

Example: `[1 2]`

`NumChannels` — Number of channels for each filter

`'auto'` (default) | positive integer

Number of channels for each filter, specified as '`NumChannels`' and `'auto'` or a positive integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights

`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = filterSize(1)*filterSize(2)*NumChannels`, `numOut = filterSize(1)*filterSize(2)*numFilters`, and `NumChannels` is the number of input channels.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = filterSize(1)*filterSize(2)*NumChannels` and `NumChannels` is the number of input channels.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias

`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights

`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `filterSize(1)`-by-`filterSize(2)`-by-`numFilters`-by-`NumChannels` array.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-`numFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

Output Arguments

`layer` — Transposed 2-D convolution layer
`TransposedConvolution2DLayer` object

Transposed 2-D convolution layer, returned as a `TransposedConvolution2DLayer` object.

Compatibility Considerations

Default weights initialization is Glorot
Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'WeightsInitializer'` option of the layer to `'narrow-normal'`.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support asymmetric cropping of the input. For example, specifying a vector `[t b l r]` for the `'Cropping'` parameter to crop the top, bottom, left, and right of the input is not supported.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`averagePooling2dLayer` | `maxPooling2dLayer` | `TransposedConvolution2DLayer` | `SoftmaxLayer`

Topics

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2017b

transposedConv3dLayer

Transposed 3-D convolution layer

Syntax

```
layer = transposedConv3dLayer(filterSize,numFilters)
layer = transposedConv3dLayer(filterSize,numFilters,Name,Value)
```

Description

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

`layer = transposedConv3dLayer(filterSize,numFilters)` returns a transposed 3-D convolution layer and sets the `FilterSize` and `NumFilters` properties.

`layer = transposedConv3dLayer(filterSize,numFilters,Name,Value)` returns a transposed 3-D convolutional layer and specifies additional options using one or more name-value pair arguments.

Examples

Create Transposed 3-D Convolutional Layer

Create a transposed 3-D convolutional layer with 32 filters, each with a height, width, and depth of 11. Use a stride of 4 in the horizontal and vertical directions and 2 along the depth.

```
layer = transposedConv3dLayer(11,32,'Stride',[4 4 2])
```

```
layer =
  TransposedConvolution3DLayer with properties:

           Name: ''

   Hyperparameters
      FilterSize: [11 11 11]
     NumChannels: 'auto'
      NumFilters: 32
          Stride: [4 4 2]
    CroppingMode: 'manual'
    CroppingSize: [2x3 double]

   Learnable Parameters
         Weights: []
            Bias: []

  Show all properties
```

Input Arguments

`filterSize` — Height, width, and depth of filters

vector of three positive integers

Height, width, and depth of the filters, specified as a vector `[h w d]` of three positive integers, where `h` is the height, `w` is the width, and `d` is the depth. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

If you set `FilterSize` using an input argument, then you can specify `FilterSize` as scalar to use the same value for all three dimensions.

Example: `[5 5 5]` specifies filters with a height, width, and depth of 5.

`numFilters` — Number of filters

positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: `96`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Cropping',1`

Transposed Convolution

`Stride` — Step size for traversing input

`[1 1 1]` (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector `[a b c]` of three positive integers, where `a` is the vertical step size, `b` is the horizontal step size, and `c` is the step size along the depth. When creating the layer, you can specify `Stride` as a scalar to use the same value for step sizes in all three directions.

Example: `[2 3 1]` specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

`Cropping` — Output size reduction

`0` (default) | `'same'` | vector of nonnegative integers | matrix of nonnegative integers

Output size reduction, specified as one of the following:

- `'same'` – Set the cropping so that the output size equals `inputSize .* Stride`, where `inputSize` is the height, width, and depth of the layer input. If you set the `'Cropping'` option to `'same'`, then the software automatically sets the `CroppingMode` property of the layer to `'same'`.

  The software trims an equal amount from the top and bottom, the left and right, and the front and back, if possible. If the vertical crop amount has an odd value, then the software trims an extra row from the bottom. If the horizontal crop amount has an odd value, then the software trims an

extra column from the right. If the depth crop amount has an odd value, then the software trims an extra plane from the back.

- A positive integer – Crop the specified amount of data from all the edges.
- A vector of nonnegative integers `[a b c]` – Crop `a` from the top and bottom, crop `b` from the left and right, and crop `c` from the front and back.
- a matrix of nonnegative integers `[t l f; b r bk]` of nonnegative integers — Crop `t`, `l`, `f`, `b`, `r`, `bk` from the top, left, front, bottom, right, and back of the input, respectively.

Example: `[1 2 2]`

`NumChannels` — Number of channels for each filter

`'auto'` (default) | positive integer

Number of channels for each filter, specified as '`NumChannels`' and `'auto'` or a positive integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights

`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = filterSize(1)*filterSize(2)*filterSize(3)*NumChannels`, `numOut = filterSize(1)*filterSize(2)*filterSize(3)*numFilters`, and `NumChannels` is the number of input channels.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = filterSize(1)*filterSize(2)*filterSize(3)*NumChannels` and `NumChannels` is the number of input channels.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias

`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | numeric array

Layer weights for the transposed convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`numFilters`-by-`NumChannels` array.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the transposed convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` 1-by-1-by-1-by-`numFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights

`1` (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases

`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

Output Arguments

`layer` — Transposed 3-D convolution layer

`TransposedConvolution3DLayer` object

Transposed 3-D convolution layer, returned as a `TransposedConvolution3dLayer` object.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`maxPooling3dLayer` | `averagePooling3dLayer` | `TransposedConvolution3dLayer` | `SoftmaxLayer` | `transposedConv2dLayer`

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a

TransposedConvolution2DLayer

Transposed 2-D convolution layer

Description

A transposed 2-D convolution layer upsamples feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

Creation

Create a transposed convolution 2-D output layer using `transposedConv2dLayer`.

Properties

Transposed Convolution

`FilterSize` — Height and width of filters
vector of two positive integers

Height and width of the filters, specified as a vector of two positive integers `[h w]`, where `h` is the height and `w` is the width. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

If you set `FilterSize` using an input argument, then you can specify `FilterSize` as scalar to use the same value for both dimensions.

Example: `[5 5]` specifies filters of height 5 and width 5.

`NumFilters` — Number of filters
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: `96`

`Stride` — Step size for traversing input
`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector `[a b]` of two positive integers, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both step sizes.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`CroppingMode` — Method to determine cropping size
`'manual'` (default) | `'same'`

Method to determine cropping size, specified as `'manual'` or same.

The software automatically sets the value of `CroppingMode` based on the `'Cropping'` value you specify when creating the layer.

- If you set the `'Cropping'` option to a numeric value, then the software automatically sets the `CroppingMode` property of the layer to `'manual'`.
- If you set the `'Cropping'` option to `'same'`, then the software automatically sets the `CroppingMode` property of the layer to `'same'` and set the cropping so that the output size equals `inputSize .* Stride`, where `inputSize` is the height and width of the layer input.

To specify the cropping size, use the `'Cropping'` option of `transposedConv2dLayer`.

`CroppingSize` — Output size reduction
`[0 0 0 0]` (default) | vector of four nonnegative integers

Output size reduction, specified as a vector of four nonnegative integers `[t b l r]`, where `t`, `b`, `l`, `r` are the amounts to crop from the top, bottom, left, and right, respectively.

To specify the cropping size manually, use the `'Cropping'` option of `transposedConv2dLayer`.

Example: `[0 1 0 1]`

`Cropping` — Output size reduction
`[0 0]` (default) | vector of two nonnegative integers

Note `Cropping` property will be removed in a future release. Use `CroppingSize` instead. To specify the cropping size manually, use the `'Cropping'` option of `transposedConv2dLayer`.

Output size reduction, specified as a vector of two nonnegative integers `[a b]`, where `a` corresponds to the cropping from the top and bottom and `b` corresponds to the cropping from the left and right.

To specify the cropping size manually, use the `'Cropping'` option of `transposedConv2dLayer`.

Example: `[0 1]`

`NumChannels` — Number of channels for each filter
`'auto'` (default) | integer

Number of channels for each filter, specified as '`NumChannels`' and `'auto'` or an integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and

variance `2/(numIn + numOut)`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannels` and `numOut = FilterSize(1)*FilterSize(2)*NumFilters`.

- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize(1)*FilterSize(2)*NumChannels`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a `FilterSize(1)`-by-`FilterSize(2)`-by-`NumFilters`-by-`NumChannels` array.

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

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-`NumFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights
`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases
`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights
1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases
`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Transposed Convolutional Layer

Create a transposed convolutional layer with 96 filters, each with a height and width of 11. Use a stride of 4 in the horizontal and vertical directions.

```
layer = transposedConv2dLayer(11,96,'Stride',4);
```

Compatibility Considerations

Default weights initialization is Glorot
Behavior changed in R2019a

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the `'WeightsInitializer'` option of the layer to `'narrow-normal'`.

`Cropping` property of `TransposedConvolution2DLayer` will be removed
Not recommended starting in R2019a

`Cropping` property of `TransposedConvolution2DLayer` will be removed, use `CroppingSize` instead. To update your code, replace all instances of the `Cropping` property with `CroppingSize`.

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`averagePooling2dLayer` | `transposedConv2dLayer` | `maxPooling2dLayer` | `convolution2dLayer`

Topics
“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”
“Compare Layer Weight Initializers”

Introduced in R2017b

TransposedConvolution3dLayer

Transposed 3-D convolution layer

Description

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

Creation

Create a transposed convolution 3-D output layer using `transposedConv3dLayer`.

Properties

Transposed Convolution

`FilterSize` — Height, width, and depth of filters
vector of three positive integers

Height, width, and depth of the filters, specified as a vector `[h w d]` of three positive integers, where `h` is the height, `w` is the width, and `d` is the depth. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify `FilterSize` as a scalar to use the same value for the height, width, and depth.

Example: `[5 5 5]` specifies filters with a height, width, and depth of 5.

`NumFilters` — Number of filters
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

`Stride` — Step size for traversing input
`[1 1 1]` (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector `[a b c]` of three positive integers, where `a` is the vertical step size, `b` is the horizontal step size, and `c` is the step size along the depth. When creating the layer, you can specify `Stride` as a scalar to use the same value for step sizes in all three directions.

Example: `[2 3 1]` specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

`CroppingMode` — Method to determine cropping size
`'manual'` (default) | `'same'`

Method to determine cropping size, specified as `'manual'` or `'same'`.

The software automatically sets the value of `CroppingMode` based on the `'Cropping'` value you specify when creating the layer.

- If you set the `'Cropping'` option to a numeric value, then the software automatically sets the `CroppingMode` property of the layer to `'manual'`.
- If you set the `'Cropping'` option to `'same'`, then the software automatically sets the `CroppingMode` property of the layer to `'same'` and set the cropping so that the output size equals `inputSize .* Stride`, where `inputSize` is the height, width, and depth of the layer input.

To specify the cropping size, use the `'Cropping'` option of `transposedConv3dLayer`.

`CroppingSize` — Output size reduction
`[0 0 0;0 0 0]` (default) | matrix of nonnegative integers

Output size reduction, specified as a matrix of nonnegative integers `[t l f; b r bk]`, `t`, `l`, `f`, `b`, `r`, `bk` are the amounts to crop from the top, left, front, bottom, right, and back of the input, respectively.

To specify the cropping size manually, use the `'Cropping'` option of `transposedConv2dLayer`.

Example: `[0 1 0 1 0 1]`

`NumChannels` — Number of channels for each filter
`'auto'` (default) | integer

Number of channels for each filter, specified `'auto'` or an integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

Parameters and Initialization

`WeightsInitializer` — Function to initialize weights
`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` – Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance `2/(numIn + numOut)`, where `numIn = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumChannels` and `numOut = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumFilters`.
- `'he'` – Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance `2/numIn`, where `numIn = FilterSize(1)*FilterSize(2)*FilterSize(3)*NumChannels`.
- `'narrow-normal'` – Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

- `'zeros'` – Initialize the weights with zeros.
- `'ones'` – Initialize the weights with ones.
- Function handle – Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

Data Types: `char` | `string` | `function_handle`

`BiasInitializer` — Function to initialize bias
`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the bias, specified as one of the following:

- `'zeros'` – Initialize the bias with zeros.
- `'ones'` – Initialize the bias with ones.
- `'narrow-normal'` – Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

`Weights` — Layer weights
`[]` (default) | numeric array

Layer weights for the transposed convolutional layer, specified as a numeric array.

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

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`NumFilters`-by-`NumChannels` array.

Data Types: `single` | `double`

`Bias` — Layer biases
`[]` (default) | numeric array

Layer biases for the transposed convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When you train a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-1-by-`NumFilters` array.

Data Types: `single` | `double`

Learning Rate and Regularization

`WeightLearnRateFactor` — Learning rate factor for weights

`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is `2`, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasLearnRateFactor` — Learning rate factor for biases

`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is `2`, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings you specify using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightL2Factor` — L_2 regularization factor for weights

1 (default) | nonnegative scalar

L_2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the weights in this layer. For example, if `WeightL2Factor` is `2`, then the L_2 regularization for the weights in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BiasL2Factor` — L_2 regularization factor for biases

`0` (default) | nonnegative scalar

L_2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L_2 regularization factor to determine the L_2 regularization for the biases in this layer. For example, if `BiasL2Factor` is `2`, then the L_2 regularization for the biases in this layer is twice the global L_2 regularization factor. You can specify the global L_2 regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

`Name` — Layer name

`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
1 (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
1 (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Create Transposed 3-D Convolutional Layer

Create a transposed 3-D convolutional layer with 32 filters, each with a height, width, and depth of 11. Use a stride of 4 in the horizontal and vertical directions and 2 along the depth.

```
layer = transposedConv3dLayer(11,32,'Stride',[4 4 2])
```

```
layer =
  TransposedConvolution3DLayer with properties:

            Name: ''

   Hyperparameters
```

```
      FilterSize: [11 11 11]
     NumChannels: 'auto'
      NumFilters: 32
          Stride: [4 4 2]
    CroppingMode: 'manual'
    CroppingSize: [2x3 double]

   Learnable Parameters
         Weights: []
            Bias: []

  Show all properties
```

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`transposedConv2dLayer` | `transposedConv3dLayer` | `maxPooling3dLayer` | `averagePooling3dLayer` | `convolution3dLayer`

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a

unfreezeParameters

Convert nonlearnable network parameters in `ONNXParameters` to learnable

Syntax

```
params = unfreezeParameters(params,names)
```

Description

`params = unfreezeParameters(params,names)` unfreezes the network parameters specified by `names` in the `ONNXParameters` object `params`. The function moves the specified parameters from `params.Nonlearnables` in the input argument `params` to `params.Learnables` in the output argument `params`.

Examples

Train Imported ONNX Function Using Custom Training Loop

Import the `squeezenet` convolution neural network as a function and fine-tune the pretrained network with transfer learning to perform classification on a new collection of images.

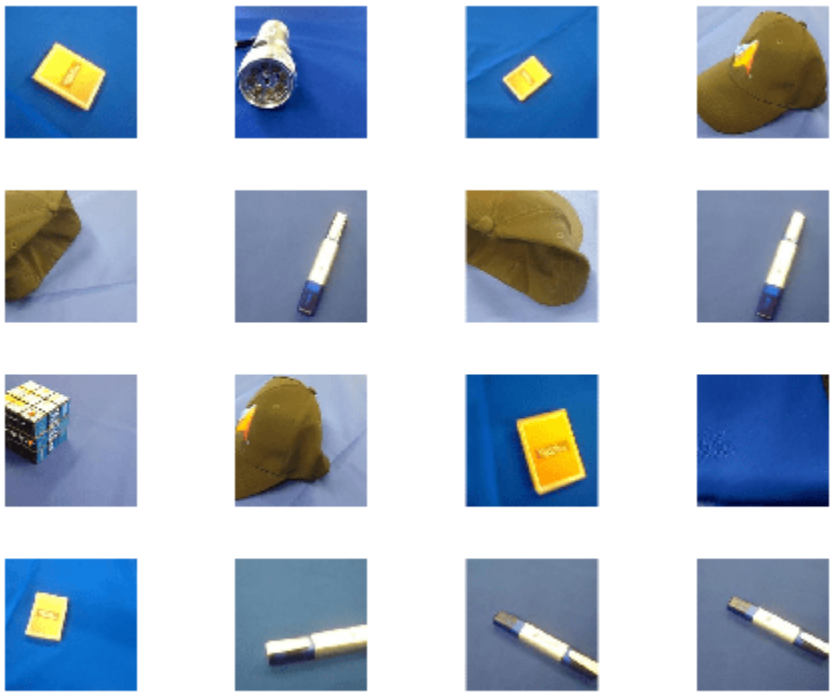
This example uses several helper functions. To view the code for these functions, see Helper Functions on page 1-0 .

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. Specify the mini-batch size.

```
unzip('MerchData.zip');
miniBatchSize = 8;
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames',...
    'ReadSize', miniBatchSize);
```

This data set is small, containing 75 training images. Display some sample images.

```
numImages = numel(imds.Labels);
idx = randperm(numImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imds,idx(i));
    imshow(I)
end
```



Extract the training set and one-hot encode the categorical classification labels.

```
XTrain = readall(imds);
XTrain = single(cat(4,XTrain{:}));
YTrain_categ = categorical(imds.Labels);
YTrain = onehotencode(YTrain_categ,2)';
```

Determine the number of classes in the data.

```
classes = categories(YTrain_categ);
numClasses = numel(classes)
```

```
numClasses = 5
```

`squeezenet` is a convolutional neural network that is trained on more than a million images from the ImageNet database. As a result, the network has learned rich feature representations for a wide range of images. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

Import the pretrained `squeezenet` network as a function.

```
squeezenetONNX()
params = importONNXFunction('squeezenet.onnx','squeezenetFcn')
```

```
A function containing the imported ONNX network has been saved to the file squeezenetFcn.m.
To learn how to use this function, type: help squeezenetFcn.

params =
  ONNXParameters with properties:
```

```
            Learnables: [1×1 struct]
       Nonlearnables: [1×1 struct]
               State: [1×1 struct]
       NumDimensions: [1×1 struct]
NetworkFunctionName: 'squeezenetFcn'
```

`params` is an `ONNXParameters` object that contains the network parameters. `squeezenetFcn` is a model function that contains the network architecture. `importONNXFunction` saves `squeezenetFcn` in the current folder.

Calculate the classification accuracy of the pretrained network on the new training set.

```
accuracyBeforeTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy before transfer learning\n',accuracyBeforeTraining);
```

```
0.01 accuracy before transfer learning
```

The accuracy is very low.

Display the learnable parameters of the network by typing `params.Learnables`. These parameters, such as the weights (`W`) and bias (`B`) of convolution and fully connected layers, are updated by the network during training. Nonlearnable parameters remain constant during training.

The last two learnable parameters of the pretrained network are configured for 1000 classes.

`conv10_W: [1×1×512×1000 dlarray]`

`conv10_B: [1000×1 dlarray]`

The parameters `conv10_W` and `conv10_B` must be fine-tuned for the new classification problem. Transfer the parameters to classify five classes by initializing the parameters.

```
params.Learnables.conv10_W = rand(1,1,512,5);
params.Learnables.conv10_B = rand(5,1);
```

Freeze all the parameters of the network to convert them to nonlearnable parameters. Because you do not need to compute the gradients of the frozen layers, freezing the weights of many initial layers can significantly speed up network training.

```
params = freezeParameters(params,'all');
```

Unfreeze the last two parameters of the network to convert them to learnable parameters.

```
params = unfreezeParameters(params,'conv10_W');
params = unfreezeParameters(params,'conv10_B');
```

Now the network is ready for training. Initialize the training progress plot.

```
plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end
```

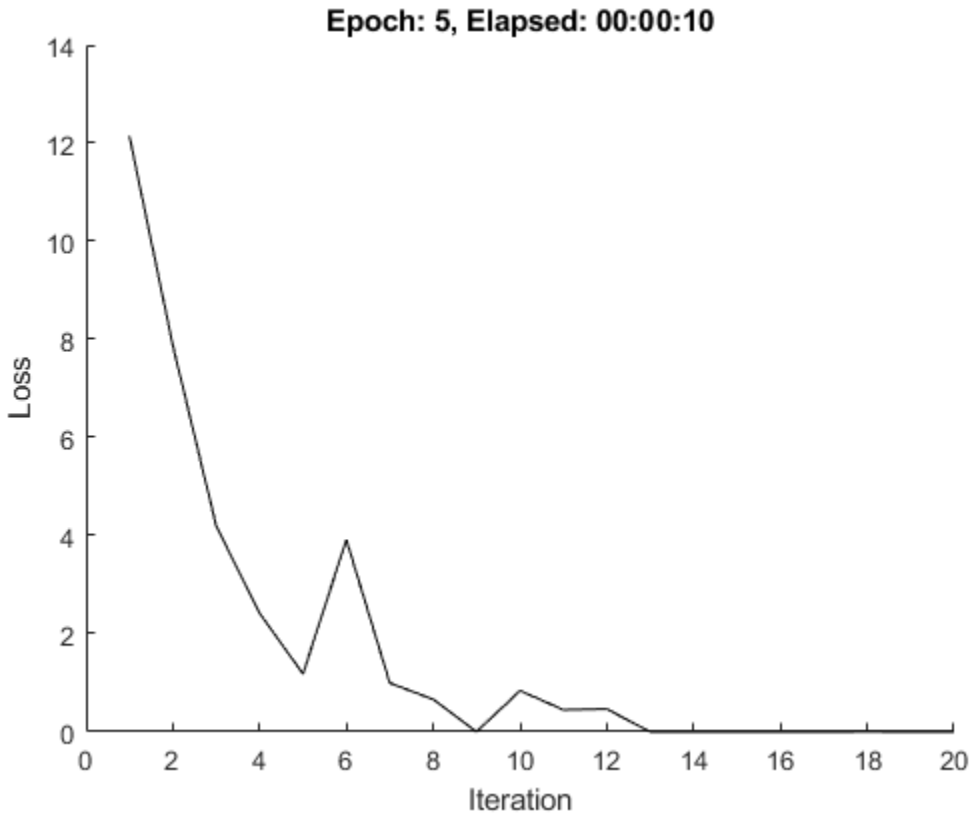
Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Train the network.

```
iteration = 0;
start = tic;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle 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.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);
        Y = YTrain(:,idx);

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

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
        params.State = state;

        % Determine the learning rate for the time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);

        % 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
```



Calculate the classification accuracy of the network after fine-tuning.

```
accuracyAfterTraining = getNetworkAccuracy(XTrain,YTrain,params);
fprintf('%.2f accuracy after transfer learning\n',accuracyAfterTraining);
```

```
1.00 accuracy after transfer learning
```

Helper Functions

This section provides the code of the helper functions used in this example.

The `getNetworkAccuracy` function evaluates the network performance by calculating the classification accuracy.

```
function accuracy = getNetworkAccuracy(X,Y,onnxParams)

N = size(X,4);
Ypred = squeezenetFcn(X,onnxParams,'Training',false);

[~,YIdx] = max(Y,[],1);
[~,YpredIdx] = max(Ypred,[],1);
numIncorrect = sum(abs(YIdx-YpredIdx) > 0);
accuracy = 1 - numIncorrect/N;

end
```

The `modelGradients` function calculates the loss and gradients.

```
function [grad, loss, state] = modelGradients(X,Y,onnxParams)

[y,state] = squeezenetFcn(X,onnxParams,'Training',true);
loss = crossentropy(y,Y,'DataFormat','CB');
grad = dlgradient(loss,onnxParams.Learnables);

end
```

The `squeezenetONNX` function generates an ONNX model of the `squeezenet` network.

```
function squeezenetONNX()

exportONNXNetwork(squeezenet,'squeezenet.onnx');

end
```

Input Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, specified as an `ONNXParameters` object. `params` contains the network parameters of the imported ONNX model.

`names` — Names of parameters to unfreeze

`'all'` | string array

Names of the parameters to unfreeze, specified as `'all'` or a string array. Unfreeze all nonlearnable parameters by setting `names` to `'all'`. Unfreeze `k` nonlearnable parameters by defining the parameter names in the 1-by-`k` string array `names`.

Example: `["gpu_0_sl_pred_b_0", "gpu_0_sl_pred_w_0"]`

Data Types: `char` | `string`

Output Arguments

`params` — Network parameters

`ONNXParameters` object

Network parameters, returned as an `ONNXParameters` object. `params` contains the network parameters updated by `unfreezeParameters`.

See Also

`importONNXFunction` | `ONNXParameters` | `freezeParameters`

Introduced in R2020b

updateInfo

Package: `experiments`

Update information columns in experiment results table

Syntax

```
updateInfo(monitor,infoName=value)
updateInfo(monitor,infoName1=value1,...,infoNameN=valueN)
updateInfo(monitor,structure)
```

Description

`updateInfo(monitor,infoName=value)` updates the specified information column for a trial in the **Experiment Manager** results table.

`updateInfo(monitor,infoName1=value1,...,infoNameN=valueN)` updates multiple information columns for a trial.

`updateInfo(monitor,structure)` updates the information columns using the values specified by the structure `structure`.

Examples

Track Progress, Display Information and Record Metric Values, and Produce Training Plots

Use an `experiments.Monitor` object to track the progress of the training, display information and metric values in the experiment results table, and produce training plots for custom training experiments.

Before starting the training, specify the names of the information and metric columns of the Experiment Manager results table.

```
monitor.Info = ["GradientDecayFactor","SquaredGradientDecayFactor"];
monitor.Metrics = ["TrainingLoss","ValidationLoss"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot.

```
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
```

Update the values of the gradient decay factor and the squared gradient decay factor for the trial in the results table.

```
updateInfo(monitor, ...
    GradientDecayFactor=gradientDecayFactor, ...
    SquaredGradientDecayFactor=squaredGradientDecayFactor);
```

After each iteration of the custom training loop, record the value of training and validation loss for the trial in the results table and the training plot.

```
recordMetrics(monitor,iteration, ...
    TrainingLoss=trainingLoss, ...
    ValidationLoss=validationLoss);
```

Update the training progress for the trial based on the fraction of iterations completed.

```
monitor.Progress = (iteration/numIterations) * 100;
```

Specify Information Values by Using Structure

Use a structure to update values of information columns in the results table.

```
structure.GradientDecayFactor = gradientDecayFactor;
structure.SquaredGradientDecayFactor = squaredGradientDecayFactor;
updateInfo(monitor,structure);
```

Input Arguments

`monitor` — Experiment monitor

`experiments.Monitor` object

Experiment monitor for the trial, specified as an `experiments.Monitor` object. When you run a custom training experiment, Experiment Manager passes this object as the second input argument of the training function.

`infoName` — Information column name

string | character vector

Information column name, specified as a string or character vector. This name must be an element of the `Info` property of the `experiments.Monitor` object `monitor`.

Data Types: `char` | `string`

`value` — Information column value

numeric scalar | string | character vector

Information column value, specified as a numeric scalar, string, or character vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`structure` — Information column names and values

structure

Information column names and values, specified as a structure. Names must be elements of the `Info` property of the `experiments.Monitor` object `monitor` and can appear in any order in the structure.

Example: `struct(GradientDecayFactor=gradientDecayFactor,SquaredGradientDecayFactor=squaredGradientDecayFactor)`

Data Types: `struct`

Tips

- Both information and metric columns display values in the results table for your experiment. Additionally, the training plot shows a record of the metric values. Use information columns for text and for numerical values that you want to display in the results table but not in the training plot.

See Also

Apps
Experiment Manager

Objects
`experiments.Monitor`

Functions
`groupSubPlot` | `recordMetrics` | `struct`

Introduced in R2021a

validate

Quantize and validate a deep neural network

Syntax

```
validationResults = validate(quantObj, valData)
validationResults = validate(quantObj, valData, quantOpts)
```

Description

`validationResults = validate(quantObj, valData)` quantizes the weights, biases, and activations in the convolution layers of the network, and validates the network specified by `dlquantizer` object, `quantObj` and using the data specified by `valData`.

`validationResults = validate(quantObj, valData, quantOpts)` quantizes the weights, biases, and activations in the convolution layers of the network, and validates the network specified by `dlquantizer` object, `quantObj`, using the data specified by `valData`, and the optional argument `quantOpts` that specifies a metric function to evaluate the performance of the quantized network.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

Examples

Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load net
net
```

```
net =
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

In this example, use the images in the `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

Create a `dlquantizer` object and specify the network to quantize.

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare with predicted label with actual ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx,:));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=95×5 table
                  Optimized Layer Name                       Network Layer Name        Learnables
    ___________________________________________________    ____________________________    __________

    {'conv1_relu_conv1_Weights'                       }    {'relu_conv1'           }          "We
    {'conv1_relu_conv1_Bias'                          }    {'relu_conv1'           }          "Bi
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}          "We
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}          "Bi
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }          "We
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }          "Bi
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }          "We
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }          "Bi
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}          "We
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}          "Bi
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }          "We
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }          "Bi
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }          "We
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }          "Bi
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}          "We
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}          "Bi
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

Quantize a Neural Network for FPGA Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

Create a file in your current working directory called `getLogoNetwork.m`. Enter these lines into the file:

```
function net = getLogoNetwork()
    data = getLogoData();
    net  = data.convnet;
end

function data = getLogoData()
    if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
end
```

Load the pretrained network.

```
snet = getLogoNetwork();
```

```
snet =

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The calibration data is used to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

This example uses the images in the `logos_dataset` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
```

Create a `dlquantizer` object and specify the network to quantize.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
 dlQuantObj.calibrate(calibrationData)
```

```
ans =
       Optimized Layer Name        Network Layer Name    Learnables / Activations    MinValue      MaxValue
    ___________________________    __________________    ________________________    __________    _________

    {'conv_1_Weights'          }      {'conv_1'    }          "Weights"                -0.048978      0.039352
    {'conv_1_Bias'             }      {'conv_1'    }          "Bias"                     0.99996        1.0028
    {'conv_2_Weights'          }      {'conv_2'    }          "Weights"                -0.055518      0.061901
    {'conv_2_Bias'             }      {'conv_2'    }          "Bias"                  -0.00061171       0.00227
    {'conv_3_Weights'          }      {'conv_3'    }          "Weights"                -0.045942      0.046927
    {'conv_3_Bias'             }      {'conv_3'    }          "Bias"                   -0.0013998     0.0015218
    {'conv_4_Weights'          }      {'conv_4'    }          "Weights"                -0.045967          0.051
    {'conv_4_Bias'             }      {'conv_4'    }          "Bias"                    -0.00164     0.0037892
    {'fc_1_Weights'            }      {'fc_1'      }          "Weights"                -0.051394      0.054344
    {'fc_1_Bias'               }      {'fc_1'      }          "Bias"                  -0.00052319    0.00084454
    {'fc_2_Weights'            }      {'fc_2'      }          "Weights"                 -0.05016      0.051557
    {'fc_2_Bias'               }      {'fc_2'      }          "Bias"                   -0.0017564     0.0018502
    {'fc_3_Weights'            }      {'fc_3'      }          "Weights"                -0.050706       0.04678
    {'fc_3_Bias'               }      {'fc_3'      }          "Bias"                    -0.02951      0.024855
    {'imageinput'              }      {'imageinput'}          "Activations"                    0           255
    {'imageinput_normalization'}      {'imageinput'}          "Activations"             -139.34        198.72
```

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To create the target object, enter:

```
hTarget = dlhdl.Target('Intel', 'Interface', 'JTAG');
```

Define a metric function to use to compare the behavior of the network before and after quantization. Save this function in a local file.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% hComputeModelAccuracy test helper function computes model level accuracy statistics

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with ground truth
    predictionError = {};
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx, :));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
```

```
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Specify the metric function in a `dlquantizationOptions` object.

```
options = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x, snet, validationData)},'Bitstream','arria10soc_int8',...
'Target',hTarget);
```

To compile and deploy the quantized network, run the `validate` function of the `dlquantizer` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function checks for the Intel Quartus tool and the supported tool version. It then starts programming the FPGA device by using the sof file, displays progress messages, and the time it takes to deploy the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
prediction = dlQuantObj.validate(validationData,options);
```

```
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "48.0 MB"
    "OutputResultOffset"        "0x03000000"     "4.0 MB"
    "SystemBufferOffset"        "0x03400000"     "60.0 MB"
    "InstructionDataOffset"     "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"        "0x08000000"     "12.0 MB"
    "EndOffset"                 "0x08c00000"     "Total: 140.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Jul-2020 12:45:10
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Jul-2020 12:45:26
### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results

                  LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                        -------------              -------------             ---------        ---------       ---------
Network                   13570959                    0.09047                   30          380609145            11.8
    conv_module           12667786                    0.08445
        conv_1             3938907                    0.02626
        maxpool_1          1544560                    0.01030
        conv_2             2910954                    0.01941
        maxpool_2           577524                    0.00385
        conv_3             2552707                    0.01702
        maxpool_3           676542                    0.00451
        conv_4              455434                    0.00304
        maxpool_4            11251                    0.00008
    fc_module               903173                    0.00602
        fc_1                536164                    0.00357
        fc_2                342643                    0.00228
        fc_3                 24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


             Deep Learning Processor Profiler Performance Results
```

```
                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                -------------              -------------       ---------        ---------     ---------
Network                          13570364                    0.09047              30          380612682          11.8
    conv_module                  12667103                    0.08445
        conv_1                    3939296                    0.02626
        maxpool_1                 1544371                    0.01030
        conv_2                    2910747                    0.01940
        maxpool_2                  577654                    0.00385
        conv_3                    2551829                    0.01701
        maxpool_3                  676548                    0.00451
        conv_4                     455396                    0.00304
        maxpool_4                   11355                    0.00008
    fc_module                      903261                    0.00602
        fc_1                       536206                    0.00357
        fc_2                       342688                    0.00228
        fc_3                        24365                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                -------------              -------------       ---------        ---------     ---------
Network                          13571561                    0.09048              30          380608338          11.8
    conv_module                  12668340                    0.08446
        conv_1                    3939070                    0.02626
        maxpool_1                 1545327                    0.01030
        conv_2                    2911061                    0.01941
        maxpool_2                  577557                    0.00385
        conv_3                    2552082                    0.01701
        maxpool_3                  676506                    0.00451
        conv_4                     455582                    0.00304
        maxpool_4                   11248                    0.00007
    fc_module                      903221                    0.00602
        fc_1                       536167                    0.00357
        fc_2                       342643                    0.00228
        fc_3                        24409                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                    LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                                -------------              -------------       ---------        ---------     ---------
Network                          13569862                    0.09047              30          380613327          11.8
    conv_module                  12666756                    0.08445
        conv_1                    3939212                    0.02626
        maxpool_1                 1543267                    0.01029
        conv_2                    2911184                    0.01941
        maxpool_2                  577275                    0.00385
        conv_3                    2552868                    0.01702
        maxpool_3                  676438                    0.00451
        conv_4                     455353                    0.00304
        maxpool_4                   11252                    0.00008
    fc_module                      903106                    0.00602
        fc_1                       536050                    0.00357
        fc_2                       342645                    0.00228
        fc_3                        24409                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.
```

```
              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13570823                    0.09047                      30          380619836            11.8
    conv_module            12667607                    0.08445
        conv_1              3939074                    0.02626
        maxpool_1           1544519                    0.01030
        conv_2              2910636                    0.01940
        maxpool_2            577769                    0.00385
        conv_3              2551800                    0.01701
        maxpool_3            676795                    0.00451
        conv_4               455859                    0.00304
        maxpool_4             11248                    0.00007
    fc_module                903216                    0.00602
        fc_1                 536165                    0.00357
        fc_2                 342643                    0.00228
        fc_3                  24406                    0.00016
 * The clock frequency of the DL processor is: 150MHz
```

offset_name	offset_address	allocated_space
"InputDataOffset"	"0x00000000"	"48.0 MB"
"OutputResultOffset"	"0x03000000"	"4.0 MB"
"SystemBufferOffset"	"0x03400000"	"60.0 MB"
"InstructionDataOffset"	"0x07000000"	"8.0 MB"
"ConvWeightDataOffset"	"0x07800000"	"8.0 MB"
"FCWeightDataOffset"	"0x08000000"	"12.0 MB"
"EndOffset"	"0x08c00000"	"Total: 140.0 MB"

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target FPGA.
### Deep learning network programming has been skipped as the same network is already loaded on the target FPGA.
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13572329                    0.09048                      10          127265075            11.8
    conv_module            12669135                    0.08446
        conv_1              3939559                    0.02626
        maxpool_1           1545378                    0.01030
        conv_2              2911243                    0.01941
        maxpool_2            577422                    0.00385
        conv_3              2552064                    0.01701
        maxpool_3            676678                    0.00451
        conv_4               455657                    0.00304
        maxpool_4             11227                    0.00007
    fc_module                903194                    0.00602
        fc_1                 536140                    0.00357
        fc_2                 342688                    0.00228
        fc_3                  24364                    0.00016
 * The clock frequency of the DL processor is: 150MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                    -------------              -------------                   ---------        ---------       ---------
Network                    13572527                    0.09048                      10          127266427            11.8
    conv_module            12669266                    0.08446
        conv_1              3939776                    0.02627
```

```
       maxpool_1          1545632                 0.01030
       conv_2             2911169                 0.01941
       maxpool_2           577592                 0.00385
       conv_3             2551613                 0.01701
       maxpool_3           676811                 0.00451
       conv_4              455418                 0.00304
       maxpool_4            11348                 0.00008
   fc_module               903261                 0.00602
       fc_1                536205                 0.00357
       fc_2                342689                 0.00228
       fc_3                 24365                 0.00016
 * The clock frequency of the DL processor is: 150MHz
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
validateOut = prediction.MetricResults.Result
```

```
ans =
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}         0.9875
     {'Quantized'     }         0.9875
```

Examine the `QuantizedNetworkFPS` field of the validation output to see the frames per second performance of the quantized network.

```
prediction.QuantizedNetworkFPS
```

```
ans = 11.8126
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

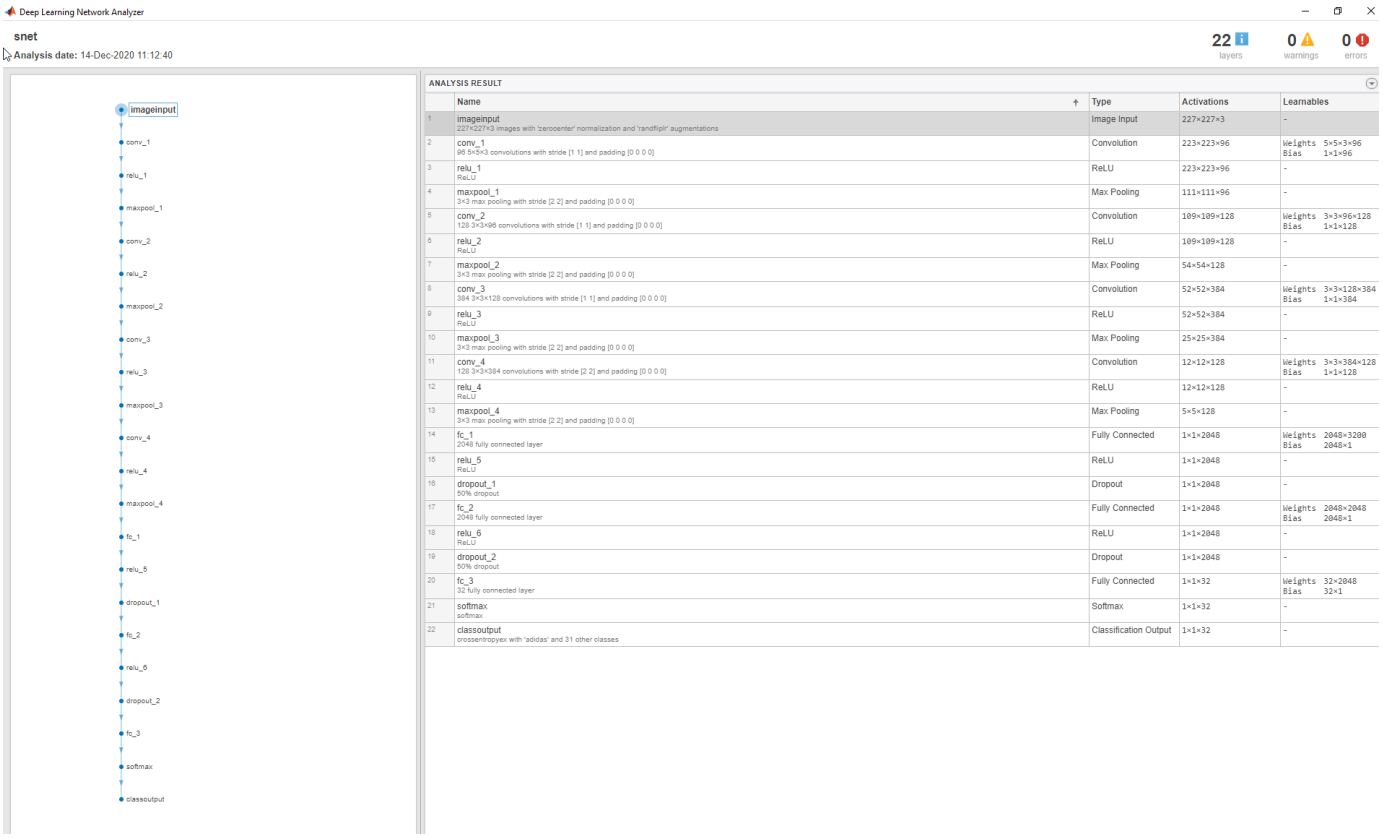
Validate Quantized Network by Using MATLAB Simulation

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and validate the quantized network. Rapidly prototype the quantized network by using MATLAB based simulation to validate the quantized network. For this type of simulation, you do not need hardware FPGA board from the prototyping process. In this example, you quantize the LogoNet neural network.

For this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

Load the pretrained network and analyze the network architecture.

```
snet = getLogoNetwork;
analyzeNetwork(snet);
```



Define calibration and validation data to use for quantization.

This example uses the `logos_dataset` data set. The data set consists of 320 images. Each image is 227-by-227 in size and has three color channels (RGB). Create an `augmentedImageDatastore` object to use for calibration and validation. Expedite the calibration and validation process by reducing the calibration data set to 20 images. The MATLAB simulation workflow has a maximum limit of five images when validating the quantized network. Reduce the validation data set sizes to five images.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir,'f');
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_reduced = calibrationData.subset(1:20);
validationData_reduced = validationData.subset(1:5);
```

Create a quantized network by using the `dlquantizer` object. To use the MATLAB simulation environment set Simulation to on.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA','Simulation','on')
```

Use the `calibrate` function to exercise the network with sample inputs and collect the range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The calibrate function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
dlQuantObj.calibrate(calibrationData_reduced)
```

```
ans =

  35×5 table

       Optimized Layer Name        Network Layer Name    Learnables / Activations     MinValue
    ___________________________    __________________    ________________________    __________

    {'conv_1_Weights'          }    {'conv_1'     }          "Weights"                -0.048978
    {'conv_1_Bias'             }    {'conv_1'     }          "Bias"                    0.99990
    {'conv_2_Weights'          }    {'conv_2'     }          "Weights"                -0.055518
    {'conv_2_Bias'             }    {'conv_2'     }          "Bias"                  -0.0006117
    {'conv_3_Weights'          }    {'conv_3'     }          "Weights"                -0.045942
    {'conv_3_Bias'             }    {'conv_3'     }          "Bias"                   -0.001398
    {'conv_4_Weights'          }    {'conv_4'     }          "Weights"                -0.045967
    {'conv_4_Bias'             }    {'conv_4'     }          "Bias"                    -0.0016
    {'fc_1_Weights'            }    {'fc_1'       }          "Weights"                -0.051394
    {'fc_1_Bias'               }    {'fc_1'       }          "Bias"                  -0.0005231
    {'fc_2_Weights'            }    {'fc_2'       }          "Weights"                 -0.0501
    {'fc_2_Bias'               }    {'fc_2'       }          "Bias"                   -0.001756
    {'fc_3_Weights'            }    {'fc_3'       }          "Weights"                -0.050706
    {'fc_3_Bias'               }    {'fc_3'       }          "Bias"                    -0.0295
    {'imageinput'              }    {'imageinput' }          "Activations"                   0
    {'imageinput_normalization'}    {'imageinput' }          "Activations"             -139.34
    {'conv_1'                  }    {'conv_1'     }          "Activations"             -431.01
    {'relu_1'                  }    {'relu_1'     }          "Activations"                   0
    {'maxpool_1'               }    {'maxpool_1'  }          "Activations"                   0
    {'conv_2'                  }    {'conv_2'     }          "Activations"             -166.41
    {'relu_2'                  }    {'relu_2'     }          "Activations"                   0
    {'maxpool_2'               }    {'maxpool_2'  }          "Activations"                   0
    {'conv_3'                  }    {'conv_3'     }          "Activations"              -219.6
    {'relu_3'                  }    {'relu_3'     }          "Activations"                   0
    {'maxpool_3'               }    {'maxpool_3'  }          "Activations"                   0
    {'conv_4'                  }    {'conv_4'     }          "Activations"             -245.37
    {'relu_4'                  }    {'relu_4'     }          "Activations"                   0
    {'maxpool_4'               }    {'maxpool_4'  }          "Activations"                   0
    {'fc_1'                    }    {'fc_1'       }          "Activations"             -123.79
    {'relu_5'                  }    {'relu_5'     }          "Activations"                   0
    {'fc_2'                    }    {'fc_2'       }          "Activations"             -16.557
    {'relu_6'                  }    {'relu_6'     }          "Activations"                   0
    {'fc_3'                    }    {'fc_3'       }          "Activations"             -13.049
    {'softmax'                 }    {'softmax'    }          "Activations"           1.4971e-22
    {'classoutput'             }    {'classoutput'}          "Activations"           1.4971e-22
```

Set your target metric function and create a `dlquantizationOptions` object with the target metric function and the validation data set. In this example the target metric function calculates the Top-5 accuracy.

```
options = dlquantizationOptions('MetricFcn', {@(x)hComputeAccuracy(x,snet,validationData_reduced)});
```

Note If no custom metric function is specified, the default metric function will be used for validation. The default metric function uses at most 5 files from the validation datastore when the MATLAB simulation environment is selected. Custom metric functions do not have this restriction.

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network. The `validate` function simulates the quantized network in MATLAB. The `validate`

function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the single data type network object to the results of the quantized network object.

```
prediction = dlQuantObj.validate(validationData_reduced,options)
```

```
### Notice: (Layer  1) The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: (Layer  2) The layer 'out_imageinput' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in software.
Compiling leg: conv_1>>maxpool_4 ...
### Notice: (Layer  1) The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: (Layer 14) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in software.
Compiling leg: conv_1>>maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
### Notice: (Layer  1) The layer 'maxpool_4' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: (Layer  7) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in software.
Compiling leg: fc_1>>fc_3 ... complete.
### Should not enter here. It means a component is unaccounted for in MATLAB Emulation.
### Notice: (Layer  1) The layer 'fc_3' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: (Layer  2) The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: (Layer  3) The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software.

prediction =

  struct with fields:

      NumSamples: 5
   MetricResults: [1×1 struct]
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
validateOut = prediction.MetricResults.Result
```

```
validateOut =

  2×2 table

    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

Input Arguments

`quantObj` — Network to quantize

`dlquantizer` object

`dlquantizer` object specifying the network to quantize.

`valData` — Data to use for validation of quantized network

`imageDataStore` object | `augmentedImageDataStore` object | `pixelLabelImageDataStore` object

Data to use for validation of quantized network, specified as an `imageDataStore` object, an `augmentedImageDataStore` object, or a `pixelLabelImageDataStore` object.

`quantOpts` — Options for quantizing network

`dlQuantizationOptions` object

Options for quantizing the network, specified as a `dlquantizationOptions` object.

Output Arguments

`validationResults` — Results of quantization of network
struct

Results of quantization of the network, returned as a struct. The struct contains these fields.

- `NumSamples` – The number of sample inputs used to validate the network.
- `MetricResults` – Struct containing results of the metric function defined in the `dlquantizationOptions` object. When more than one metric function is specified in the `dlquantizationOptions` object, `MetricResults` is an array of structs.

  `MetricResults` contains these fields.

Field	Description
`MetricFunction`	Function used to determine the performance of the quantized network. This function is specified in the `dlquantizationOptions` object.
`Result`	Table indicating the results of the metric function before and after quantization. The first row in the table contains the information for the original, floating-point implementation. The second row contains the information for the quantized implementation. The output of the metric function is displayed in the `MetricOutput` column.

See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `dlquantizer` | `dlquantizationOptions`

Topics
“Quantization of Deep Neural Networks”

Introduced in R2020a

vgg16

VGG-16 convolutional neural network

Syntax

```
net = vgg16
net = vgg16('Weights','imagenet')

layers = vgg16('Weights','none')
```

Description

VGG-16 is a convolutional neural network that is 16 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the VGG-16 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with VGG-16.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load VGG-16 instead of GoogLeNet.

`net = vgg16` returns a VGG-16 network trained on the ImageNet data set.

This function requires Deep Learning Toolbox Model *for VGG-16 Network* support package. If this support package is not installed, then the function provides a download link.

`net = vgg16('Weights','imagenet')` returns a VGG-16 network trained on the ImageNet data set. This syntax is equivalent to `net = vgg16`.

`layers = vgg16('Weights','none')` returns the untrained VGG-16 network architecture. The untrained model does not require the support package.

Examples

Download VGG-16 Support Package

Download and install Deep Learning Toolbox Model *for VGG-16 Network* support package.

Type `vgg16` at the command line.

```
vgg16
```

If Deep Learning Toolbox Model *for VGG-16 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `vgg16` at the command line.

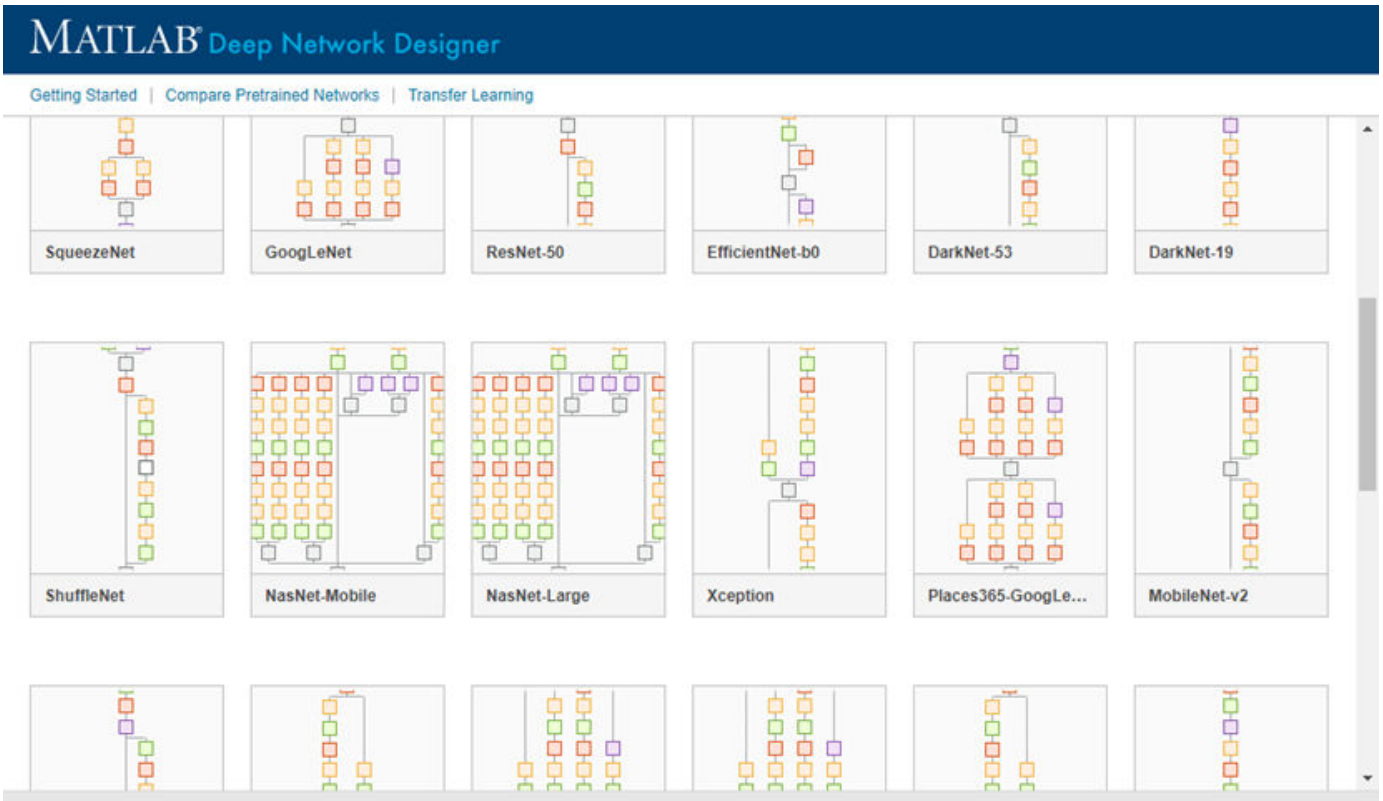
```
vgg16
```

```
ans =

  SeriesNetwork with properties:

    Layers: [41×1 nnet.cnn.layer.Layer]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(vgg16)
```

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



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

Load Pretrained VGG-16 Convolutional Neural Network

Load a pretrained VGG-16 convolutional neural network and examine the layers and classes.

Use `vgg16` to load the pretrained VGG-16 network. The output `net` is a `SeriesNetwork` object.

```
net = vgg16
```

```
net =
  SeriesNetwork with properties:
```

```
    Layers: [41×1 nnet.cnn.layer.Layer]
```

View the network architecture using the `Layers` property. The network has 41 layers. There are 16 layers with learnable weights: 13 convolutional layers, and 3 fully connected layers.

```
net.Layers
```

```
ans =
  41x1 Layer array with layers:

     1   'input'     Image Input             224x224x3 images with 'zerocenter' normalization
     2   'conv1_1'   Convolution             64 3x3x3 convolutions with stride [1  1] and padding
     3   'relu1_1'   ReLU                    ReLU
     4   'conv1_2'   Convolution             64 3x3x64 convolutions with stride [1  1] and paddi
     5   'relu1_2'   ReLU                    ReLU
     6   'pool1'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
     7   'conv2_1'   Convolution             128 3x3x64 convolutions with stride [1  1] and paddi
     8   'relu2_1'   ReLU                    ReLU
     9   'conv2_2'   Convolution             128 3x3x128 convolutions with stride [1  1] and pado
    10   'relu2_2'   ReLU                    ReLU
    11   'pool2'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    12   'conv3_1'   Convolution             256 3x3x128 convolutions with stride [1  1] and pado
    13   'relu3_1'   ReLU                    ReLU
    14   'conv3_2'   Convolution             256 3x3x256 convolutions with stride [1  1] and pado
    15   'relu3_2'   ReLU                    ReLU
    16   'conv3_3'   Convolution             256 3x3x256 convolutions with stride [1  1] and pado
    17   'relu3_3'   ReLU                    ReLU
    18   'pool3'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    19   'conv4_1'   Convolution             512 3x3x256 convolutions with stride [1  1] and pado
    20   'relu4_1'   ReLU                    ReLU
    21   'conv4_2'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    22   'relu4_2'   ReLU                    ReLU
    23   'conv4_3'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    24   'relu4_3'   ReLU                    ReLU
    25   'pool4'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    26   'conv5_1'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    27   'relu5_1'   ReLU                    ReLU
    28   'conv5_2'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    29   'relu5_2'   ReLU                    ReLU
    30   'conv5_3'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    31   'relu5_3'   ReLU                    ReLU
    32   'pool5'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    33   'fc6'       Fully Connected         4096 fully connected layer
    34   'relu6'     ReLU                    ReLU
    35   'drop6'     Dropout                 50% dropout
    36   'fc7'       Fully Connected         4096 fully connected layer
    37   'relu7'     ReLU                    ReLU
    38   'drop7'     Dropout                 50% dropout
    39   'fc8'       Fully Connected         1000 fully connected layer
    40   'prob'      Softmax                 softmax
    41   'output'    Classification Output   crossentropyex with 'tench' and 999 other classes
```

To view the names of the classes learned by the network, you can view the `Classes` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical array
     tench
     goldfish
     great white shark
     tiger shark
     hammerhead
     electric ray
     stingray
     cock
     hen
     ostrich
```

Output Arguments

net — Pretrained VGG-16 convolutional neural network
`SeriesNetwork` object

Pretrained VGG-16 convolutional neural network returned as a `SeriesNetwork` object.

layers — Untrained VGG-16 convolutional neural network architecture
`Layer` array

Untrained VGG-16 convolutional neural network architecture, returned as a `Layer` array.

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] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] *Very Deep Convolutional Networks for Large-Scale Visual Recognition* http://www.robots.ox.ac.uk/~vgg/research/very_deep/

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = vgg16` or by passing the `vgg16` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg16')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `vgg16('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = vgg16` or by passing the `vgg16` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg16')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `vgg16('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `alexnet` | `vgg19` | `googlenet` | `densenet201` | `resnet18` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Transfer Learning Using Pretrained Network”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a

vgg19

VGG-19 convolutional neural network

Syntax

```
net = vgg19
net = vgg19('Weights','imagenet')

layers = vgg19('Weights','none')
```

Description

VGG-19 is a convolutional neural network that is 19 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the VGG-19 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with VGG-19.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load VGG-19 instead of GoogLeNet.

`net = vgg19` returns a VGG-19 network trained on the ImageNet data set.

This function requires Deep Learning Toolbox Model *for VGG-19 Network* support package. If this support package is not installed, then the function provides a download link.

`net = vgg19('Weights','imagenet')` returns a VGG-19 network trained on the ImageNet data set. This syntax is equivalent to `net = vgg19`.

`layers = vgg19('Weights','none')` returns the untrained VGG-19 network architecture. The untrained model does not require the support package.

Examples

Download VGG-19 Support Package

This example shows how to download and install Deep Learning Toolbox Model *for VGG-19 Network* support package.

Type `vgg19` at the command line.

```
vgg19
```

If Deep Learning Toolbox Model *for VGG-19 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the

support package, click the link, and then click **Install**. Check that the installation is successful by typing `vgg19` at the command line.

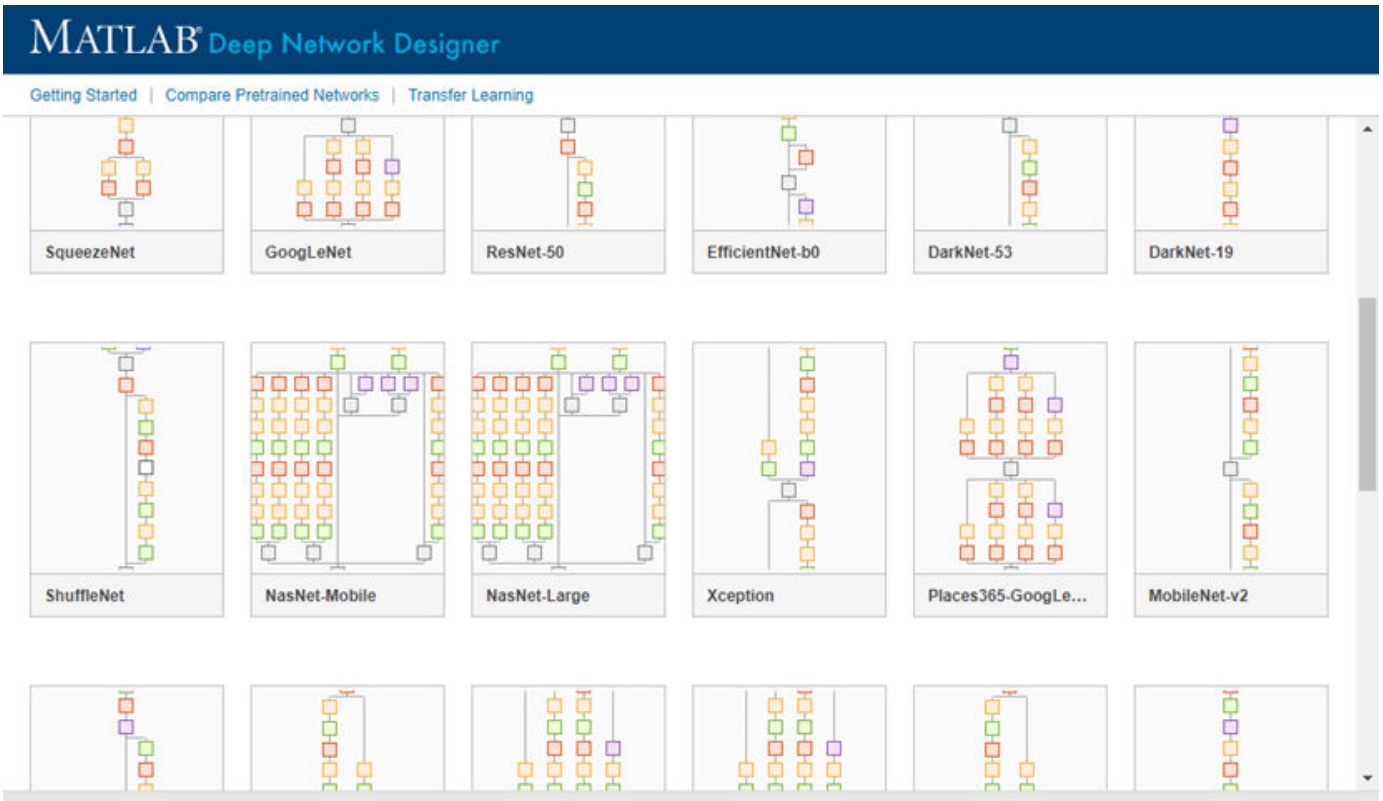
```
vgg19
```

```
ans =

  SeriesNetwork with properties:

    Layers: [47×1 nnet.cnn.layer.Layer]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(vgg19)
```

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



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

Load Pretrained VGG-19 Convolutional Neural Network

Load a pretrained VGG-19 convolutional neural network and examine the layers and classes.

Use `vgg19` to load a pretrained VGG-19 network. The output `net` is a `SeriesNetwork` object.

```
net = vgg19
```

```
net =
  SeriesNetwork with properties:

    Layers: [47×1 nnet.cnn.layer.Layer]
```

View the network architecture using the `Layers` property. The network has 47 layers. There are 19 layers with learnable weights: 16 convolutional layers, and 3 fully connected layers.

```
net.Layers
```

```
ans =
  47x1 Layer array with layers:

     1   'input'     Image Input             224x224x3 images with 'zerocenter' normalization
     2   'conv1_1'   Convolution             64 3x3x3 convolutions with stride [1  1] and padding
     3   'relu1_1'   ReLU                    ReLU
     4   'conv1_2'   Convolution             64 3x3x64 convolutions with stride [1  1] and paddin
     5   'relu1_2'   ReLU                    ReLU
     6   'pool1'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
     7   'conv2_1'   Convolution             128 3x3x64 convolutions with stride [1  1] and paddi
     8   'relu2_1'   ReLU                    ReLU
     9   'conv2_2'   Convolution             128 3x3x128 convolutions with stride [1  1] and pado
    10   'relu2_2'   ReLU                    ReLU
    11   'pool2'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    12   'conv3_1'   Convolution             256 3x3x128 convolutions with stride [1  1] and pado
    13   'relu3_1'   ReLU                    ReLU
    14   'conv3_2'   Convolution             256 3x3x256 convolutions with stride [1  1] and pado
    15   'relu3_2'   ReLU                    ReLU
    16   'conv3_3'   Convolution             256 3x3x256 convolutions with stride [1  1] and pado
    17   'relu3_3'   ReLU                    ReLU
    18   'conv3_4'   Convolution             256 3x3x256 convolutions with stride [1  1] and pado
    19   'relu3_4'   ReLU                    ReLU
    20   'pool3'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    21   'conv4_1'   Convolution             512 3x3x256 convolutions with stride [1  1] and pado
    22   'relu4_1'   ReLU                    ReLU
    23   'conv4_2'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    24   'relu4_2'   ReLU                    ReLU
    25   'conv4_3'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    26   'relu4_3'   ReLU                    ReLU
    27   'conv4_4'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    28   'relu4_4'   ReLU                    ReLU
    29   'pool4'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    30   'conv5_1'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    31   'relu5_1'   ReLU                    ReLU
    32   'conv5_2'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    33   'relu5_2'   ReLU                    ReLU
    34   'conv5_3'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    35   'relu5_3'   ReLU                    ReLU
    36   'conv5_4'   Convolution             512 3x3x512 convolutions with stride [1  1] and pado
    37   'relu5_4'   ReLU                    ReLU
    38   'pool5'     Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0
    39   'fc6'       Fully Connected         4096 fully connected layer
    40   'relu6'     ReLU                    ReLU
    41   'drop6'     Dropout                 50% dropout
    42   'fc7'       Fully Connected         4096 fully connected layer
    43   'relu7'     ReLU                    ReLU
    44   'drop7'     Dropout                 50% dropout
```

```
    45   'fc8'       Fully Connected         1000 fully connected layer
    46   'prob'      Softmax                 softmax
    47   'output'    Classification Output   crossentropyex with 'tench' and 999 other classes
```

To view the names of the classes learned by the network, you can view the `Classes` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical array
     tench
     goldfish
     great white shark
     tiger shark
     hammerhead
     electric ray
     stingray
     cock
     hen
     ostrich
```

Output Arguments

net — Pretrained VGG-19 convolutional neural network

`SeriesNetwork` object

Pretrained VGG-19 convolutional neural network returned as a `SeriesNetwork` object.

layers — Untrained VGG-19 convolutional neural network architecture

`Layer` array

Untrained VGG-19 convolutional neural network architecture, returned as a `Layer` array.

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] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] *Very Deep Convolutional Networks for Large-Scale Visual Recognition* http://www.robots.ox.ac.uk/~vgg/research/very_deep/

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = vgg19` or by passing the `vgg19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg19')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `vgg19('Weights','none')` is not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = vgg19` or by passing the `vgg19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg19')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `vgg19('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `alexnet` | `vgg16` | `googlenet` | `resnet18` | `resnet50` | `resnet101` | `deepDreamImage` | `inceptionresnetv2` | `squeezenet` | `densenet201`

Topics
“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Transfer Learning Using Pretrained Network”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a

vggish

VGGish neural network

Syntax

```
net = vggish
```

Description

`net = vggish` returns a pretrained VGGish model.

This function requires both Audio Toolbox™ and Deep Learning Toolbox.

Examples

Download VGGish Network

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggish` at the Command Window. If the Audio Toolbox model for VGGish is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the VGGish model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'VGGishDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/vggish.zip');
VGGishLocation = tempdir;
unzip(loc,VGGishLocation)
addpath(fullfile(VGGishLocation,'vggish'))
```

Check that the installation is successful by typing `vggish` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` object.

```
vggish
```

```
ans =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

Load Pretrained VGGish Network

Load a pretrained VGGish convolutional neural network and examine the layers and classes.

Use `vggish` to load the pretrained VGGish network. The output `net` is a `SeriesNetwork` object.

```
net = vggish
```

```
net =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

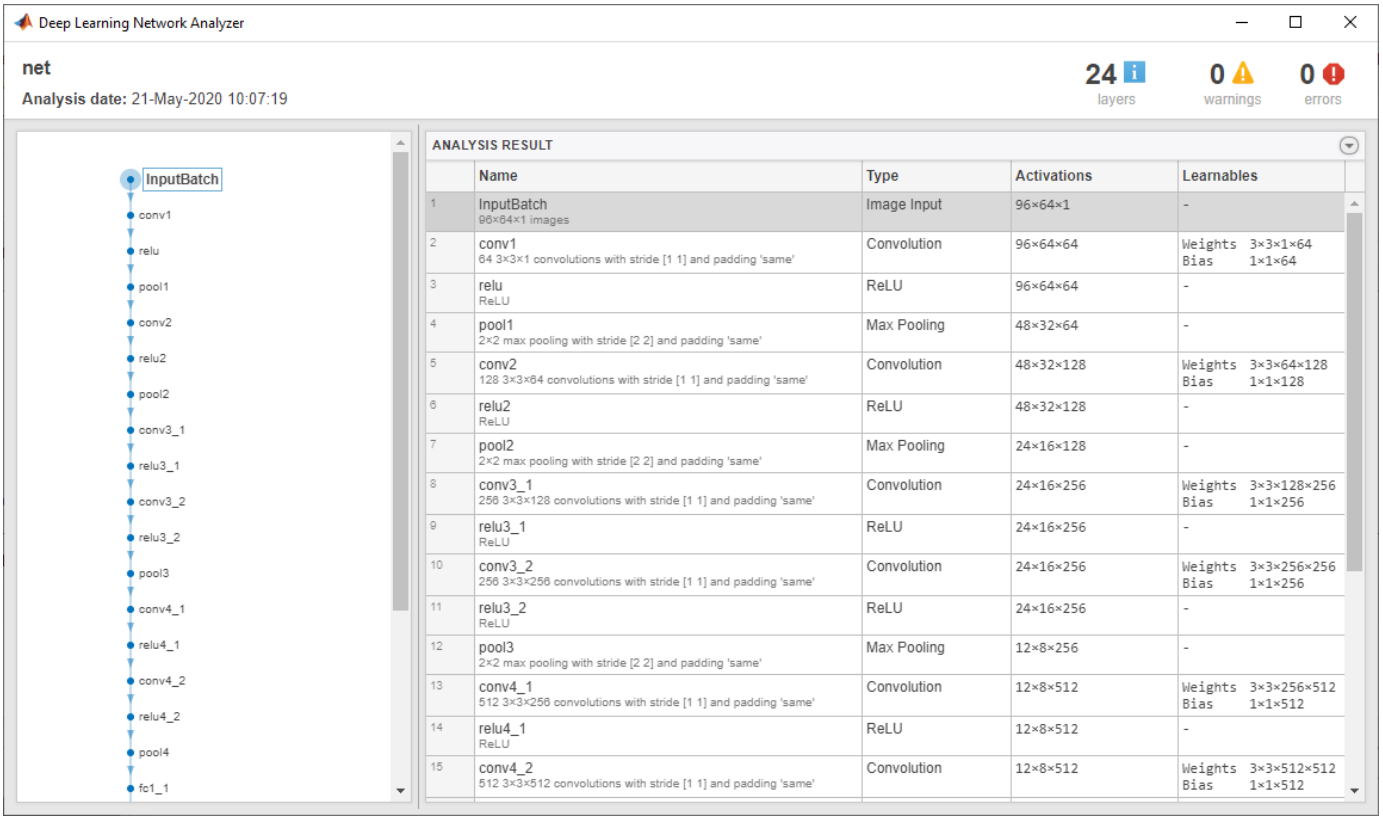
View the network architecture using the `Layers` property. The network has 24 layers. There are nine layers with learnable weights, of which six are convolutional layers and three are fully connected layers.

```
net.Layers
```

```
ans =
  24×1 Layer array with layers:

     1   'InputBatch'         Image Input         96×64×1 images
     2   'conv1'              Convolution         64 3×3×1 convolutions with stride [1  1] and pa
     3   'relu'               ReLU                ReLU
     4   'pool1'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv2'              Convolution         128 3×3×64 convolutions with stride [1  1] and
     6   'relu2'              ReLU                ReLU
     7   'pool2'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv3_1'            Convolution         256 3×3×128 convolutions with stride [1  1] and
     9   'relu3_1'            ReLU                ReLU
    10   'conv3_2'            Convolution         256 3×3×256 convolutions with stride [1  1] and
    11   'relu3_2'            ReLU                ReLU
    12   'pool3'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    13   'conv4_1'            Convolution         512 3×3×256 convolutions with stride [1  1] and
    14   'relu4_1'            ReLU                ReLU
    15   'conv4_2'            Convolution         512 3×3×512 convolutions with stride [1  1] and
    16   'relu4_2'            ReLU                ReLU
    17   'pool4'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    18   'fc1_1'              Fully Connected     4096 fully connected layer
    19   'relu5_1'            ReLU                ReLU
    20   'fc1_2'              Fully Connected     4096 fully connected layer
    21   'relu5_2'            ReLU                ReLU
    22   'fc2'                Fully Connected     128 fully connected layer
    23   'EmbeddingBatch'     ReLU                ReLU
    24   'regressionoutput'   Regression Output   mean-squared-error
```

Use `analyzeNetwork` to visually explore the network.

```
analyzeNetwork(net)
```



Extract Features Using VGGish

The VGGish network requires you to preprocess and extract features from audio signals by converting them to the sample rate the network was trained on, and then extracting log mel spectrograms. This example walks through the required preprocessing and feature extraction to match the preprocessing and feature extraction used to train VGGish. The `vggishFeatures` (Audio Toolbox) function performs these steps for you.

Read in an audio signal to classify. Resample the audio signal to 16 kHz and then convert it to single precision.

```
[audioIn,fs0] = audioread( Ambiance-16-44p1-... );

fs = 16e3;
audioIn = resample(audioIn,fs,fs0);

audioIn = single(audioIn);
```

Define mel spectrogram parameters and then extract features using the `melSpectrogram` (Audio Toolbox) function.

```
FFTLength = 512;
numBands = 64;
frequencyRange = [125 7500];
```

```
windowLength = 0.025*fs;
overlapLength = 0.015*fs;

melSpect = melSpectrogram(audioIn,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');
```

Convert the mel spectrogram to the log scale.

```
melSpect = log(melSpect + single(0.001));
```

Reorient the mel spectrogram so that time is along the first dimension as rows.

```
melSpect = melSpect.';
[numSTFTWindows,numBands] = size(melSpect)
```

```
numSTFTWindows = 1222
```

```
numBands = 64
```

Partition the spectrogram into frames of length 96 with an overlap of 48. Place the frames along the fourth dimension.

```
frameWindowLength = 96;
frameOverlapLength = 48;

hopLength = frameWindowLength - frameOverlapLength;
numHops = floor((numSTFTWindows - frameWindowLength)/hopLength) + 1;

frames = zeros(frameWindowLength,numBands,1,numHops,'like',melSpect);
for hop = 1:numHops
    range = 1 + hopLength*(hop-1):hopLength*(hop - 1) + frameWindowLength;
    frames(:,:,1,hop) = melSpect(range,:);
end
```

Create a VGGish network.

```
net = vggish;
```

Call `predict` to extract feature embeddings from the spectrogram images. The feature embeddings are returned as a `numFrames`-by-128 matrix, where `numFrames` is the number of individual spectrograms, and 128 is the number of elements in each feature vector.

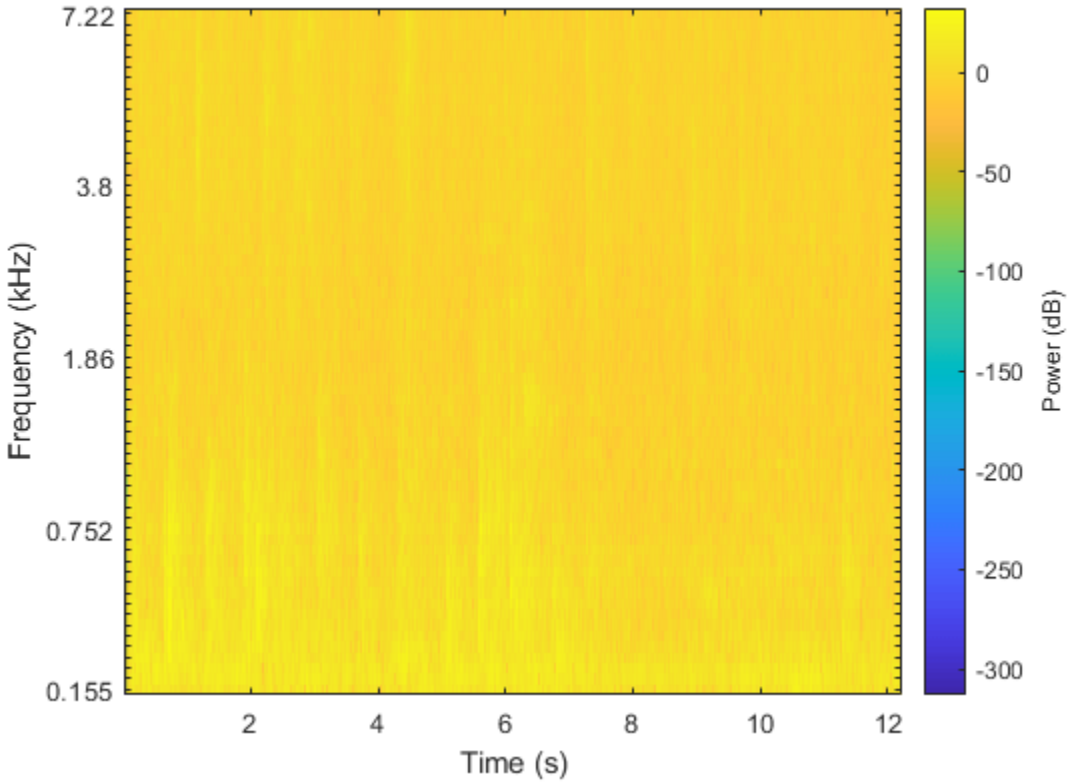
```
features = predict(net,frames);

[numFrames,numFeatures] = size(features)
```

```
numFrames = 24
```

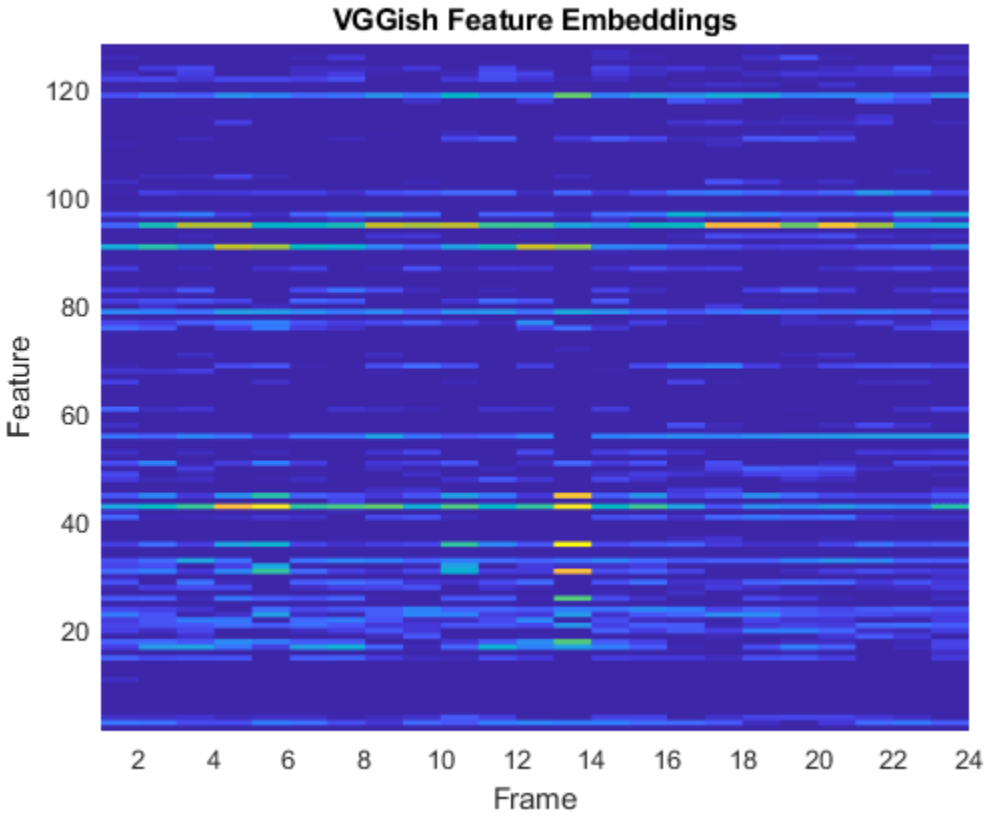
```
numFeatures = 128
```

Compare visualizations of the mel spectrogram and the VGGish feature embeddings.

```
melSpectrogram(audioIn,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');
```




```
surf(features,'EdgeColor','none')
view([90,-90])
axis([1 numFeatures 1 numFrames])
xlabel('Feature')
ylabel('Frame')
title('VGGish Feature Embeddings')
```



Transfer Learning Using VGGish

In this example, you transfer the learning in the VGGish regression model to an audio classification task.

Download and unzip the environmental sound classification data set. This data set consists of recordings labeled as one of 10 different audio sound classes (ESC-10).

```
url = 'http://ssd.mathworks.com/supportfiles/audio/ESC-10.zip';
downloadFolder = fullfile(tempdir,'ESC-10');
datasetLocation = tempdir;

if ~exist(fullfile(tempdir,'ESC-10'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'ESC-10'))
end
```

Create an `audioDatastore` (Audio Toolbox) object to manage the data and split it into train and validation sets. Call `countEachLabel` (Audio Toolbox) to display the distribution of sound classes and the number of unique labels.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');
labelTable = countEachLabel(ads)
```

```
labelTable=10×2 table
        Label         Count
    ______________    _____

    chainsaw           40
    clock_tick         40
    crackling_fire     40
    crying_baby        40
    dog                40
    helicopter         40
    rain               40
    rooster            38
    sea_waves          40
    sneezing           40
```

Determine the total number of classes.

```
numClasses = size(labelTable,1);
```

Call `splitEachLabel` (Audio Toolbox) to split the data set into training and validation sets. Inspect the distribution of labels in the training and validation sets.

```
[adsTrain, adsValidation] = splitEachLabel(ads,0.8);

countEachLabel(adsTrain)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw           32
    clock_tick         32
    crackling_fire     32
    crying_baby        32
    dog                32
    helicopter         32
    rain               32
    rooster            30
    sea_waves          32
    sneezing           32
```

```
countEachLabel(adsValidation)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw            8
    clock_tick          8
    crackling_fire      8
    crying_baby         8
    dog                 8
    helicopter          8
    rain                8
    rooster             8
    sea_waves           8
```

```
    sneezing            8
```

The VGGish network expects audio to be preprocessed into log mel spectrograms. The supporting function `vggishPreprocess` on page 1-0 takes an `audioDatastore` object and the overlap percentage between log mel spectrograms as input, and returns matrices of predictors and responses suitable as input to the VGGish network.

```
overlapPercentage = 75;

[trainFeatures,trainLabels] = vggishPreprocess(adsTrain,overlapPercentage);
[validationFeatures,validationLabels,segmentsPerFile] = vggishPreprocess(adsValidation,overlapPe
```

Load the VGGish model and convert it to a `layerGraph` object.

```
net = vggish;

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

Use `removeLayers` to remove the final regression output layer from the graph. After you remove the regression layer, the new final layer of the graph is a ReLU layer named `'EmbeddingBatch'`.

```
lgraph = removeLayers(lgraph,'regressionoutput');
lgraph.Layers(end)

ans =
  ReLULayer with properties:

    Name: 'EmbeddingBatch'
```

Use `addLayers` to add a `fullyConnectedLayer`, a `softmaxLayer`, and a `classificationLayer` to the graph.

```
lgraph = addLayers(lgraph,fullyConnectedLayer(numClasses,'Name','FCFinal'));
lgraph = addLayers(lgraph,softmaxLayer('Name','softmax'));
lgraph = addLayers(lgraph,classificationLayer('Name','classOut'));
```

Use `connectLayers` to append the fully connected, softmax, and classification layers to the layer graph.

```
lgraph = connectLayers(lgraph,'EmbeddingBatch','FCFinal');
lgraph = connectLayers(lgraph,'FCFinal','softmax');
lgraph = connectLayers(lgraph,'softmax','classOut');
```

To define training options, use `trainingOptions`.

```
miniBatchSize = 128;
options = trainingOptions('adam', ...
    'MaxEpochs',5, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{validationFeatures,validationLabels}, ...
    'ValidationFrequency',50, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.5, ...
    'LearnRateDropPeriod',2);
```

To train the network, use `trainNetwork`.

```
[trainedNet, netInfo] = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```

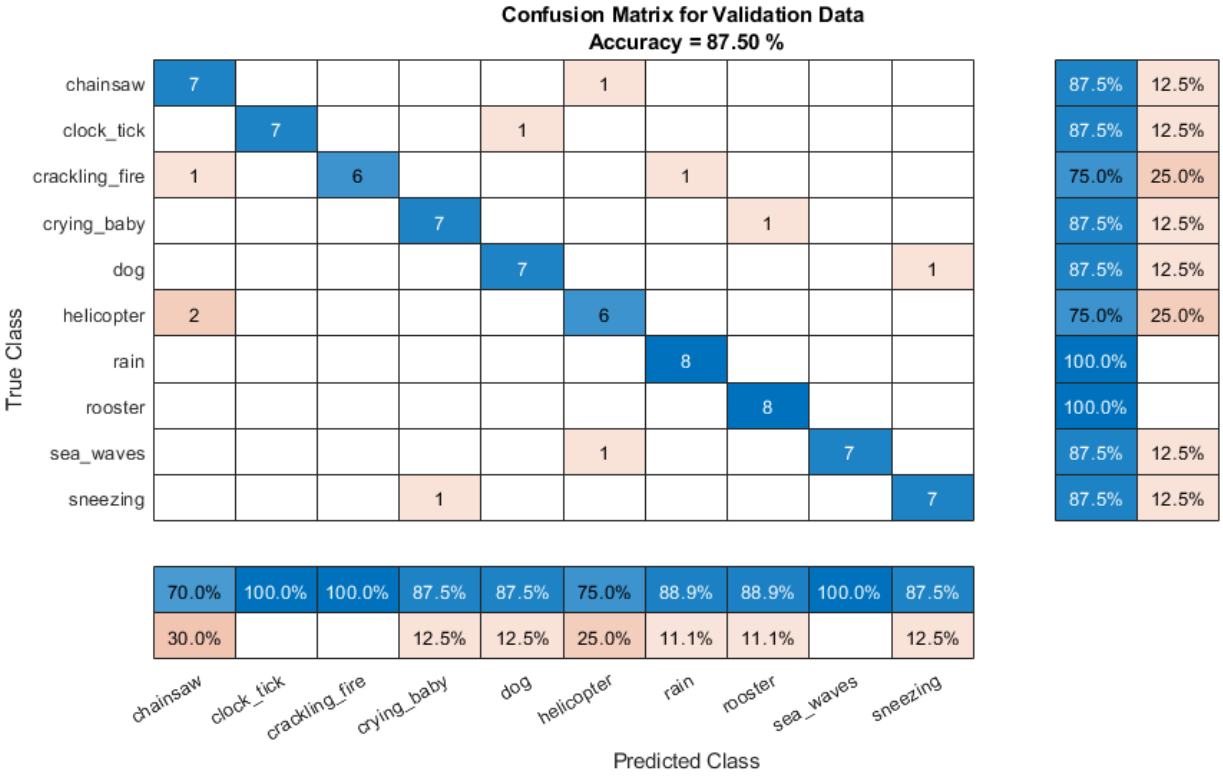
```
Training on single GPU.
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:00 |       10.94% |       26.03% |       2.2253 |       2.0
|       2 |          50 |       00:00:05 |       93.75% |       83.75% |       0.1884 |       0.1
|       3 |         100 |       00:00:10 |       96.88% |       80.07% |       0.1150 |       0.1
|       4 |         150 |       00:00:15 |       92.97% |       81.99% |       0.1656 |       0.1
|       5 |         200 |       00:00:20 |       92.19% |       79.04% |       0.1738 |       0.8
|       5 |         210 |       00:00:21 |       95.31% |       80.15% |       0.1389 |       0.8
|=========================================================================================================
```

Each audio file was split into several segments to feed into the VGGish network. Combine the predictions for each file in the validation set using a majority-rule decision.

```
validationPredictions = classify(trainedNet,validationFeatures);

idx = 1;
validationPredictionsPerFile = categorical;
for ii = 1:numel(adsValidation.Files)
    validationPredictionsPerFile(ii,1) = mode(validationPredictions(idx:idx+segmentsPerFile(ii)-1
    idx = idx + segmentsPerFile(ii);
end
```

Use `confusionchart` to evaluate the performance of the network on the validation set.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
cm = confusionchart(adsValidation.Labels,validationPredictionsPerFile);
cm.Title = sprintf('Confusion Matrix for Validation Data \nAccuracy = %0.2f %%',mean(validationP
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



Supporting Functions

```
function [predictor,response,segmentsPerFile] = vggishPreprocess(ads,overlap)
% This function is for example purposes only and may be changed or removed
% in a future release.

% Create filter bank
FFTLength = 512;
numBands = 64;
fs0 = 16e3;
filterBank = designAuditoryFilterBank(fs0, ...
    'FrequencyScale','mel', ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',[125 7500], ...
    'NumBands',numBands, ...
    'Normalization','none', ...
    'FilterBankDesignDomain','warped');

% Define STFT parameters
windowLength = 0.025 * fs0;
hopLength = 0.01 * fs0;
win = hann(windowLength,'periodic');

% Define spectrogram segmentation parameters
segmentDuration = 0.96; % seconds
segmentRate = 100; % hertz
segmentLength = segmentDuration*segmentRate; % Number of spectrums per auditory spectrograms
segmentHopDuration = (100-overlap) * segmentDuration / 100; % Duration (s) advanced between audi
segmentHopLength = round(segmentHopDuration * segmentRate); % Number of spectrums advanced betwe
```

```
% Preallocate cell arrays for the predictors and responses
numFiles = numel(ads.Files);
predictor = cell(numFiles,1);
response = predictor;
segmentsPerFile = zeros(numFiles,1);

% Extract predictors and responses for each file
for ii = 1:numFiles
    [audioIn,info] = read(ads);

    x = single(resample(audioIn,fs0,info.SampleRate));

    Y = stft(x, ...
        'Window',win, ...
        'OverlapLength',windowLength-hopLength, ...
        'FFTLength',FFTLength, ...
        'FrequencyRange','onesided');
    Y = abs(Y);

    logMelSpectrogram = log(filterBank*Y + single(0.01))';

    % Segment log-mel spectrogram
    numHops = floor((size(Y,2)-segmentLength)/segmentHopLength) + 1;
    segmentedLogMelSpectrogram = zeros(segmentLength,numBands,1,numHops);
    for hop = 1:numHops
        segmentedLogMelSpectrogram(:,:,1,hop) = logMelSpectrogram(1+segmentHopLength*(hop-1):segr
    end

    predictor{ii} = segmentedLogMelSpectrogram;
    response{ii} = repelem(info.Label,numHops);
    segmentsPerFile(ii) = numHops;
end

% Concatenate predictors and responses into arrays
predictor = cat(4,predictor{:});
response = cat(2,response{:});
end
```

Output Arguments

net — Pretrained VGGish neural network

`SeriesNetwork` object

Pretrained VGGish neural network, returned as a `SeriesNetwork` object.

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. 2017. “CNN Architectures for Large-Scale Audio Classification.”

In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

Apps

Audio Labeler

Blocks

Sound Classifier | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `melSpectrogram` | `vggishFeatures` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

Introduced in R2020b

xception

Xception convolutional neural network

Syntax

```
net = xception
net = xception('Weights','imagenet')

lgraph = xception('Weights','none')
```

Description

Xception is a convolutional neural network that is 71 layers deep. You can load a pretrained version of the network trained on more than a million images from the ImageNet database [1]. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the Xception model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Xception.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Xception instead of GoogLeNet.

`net = xception` returns an Xception network trained on the ImageNet data set.

This function requires the Deep Learning Toolbox Model *for Xception Network* support package. If this support package is not installed, then the function provides a download link.

`net = xception('Weights','imagenet')` returns an Xception network trained on the ImageNet data set. This syntax is equivalent to `net = xception`.

`lgraph = xception('Weights','none')` returns the untrained Xception network architecture. The untrained model does not require the support package.

Examples

Download Xception Support Package

Download and install the Deep Learning Toolbox Model *for Xception Network* support package.

Type `xception` at the command line.

```
xception
```

If the Deep Learning Toolbox Model *for Xception Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `xception` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

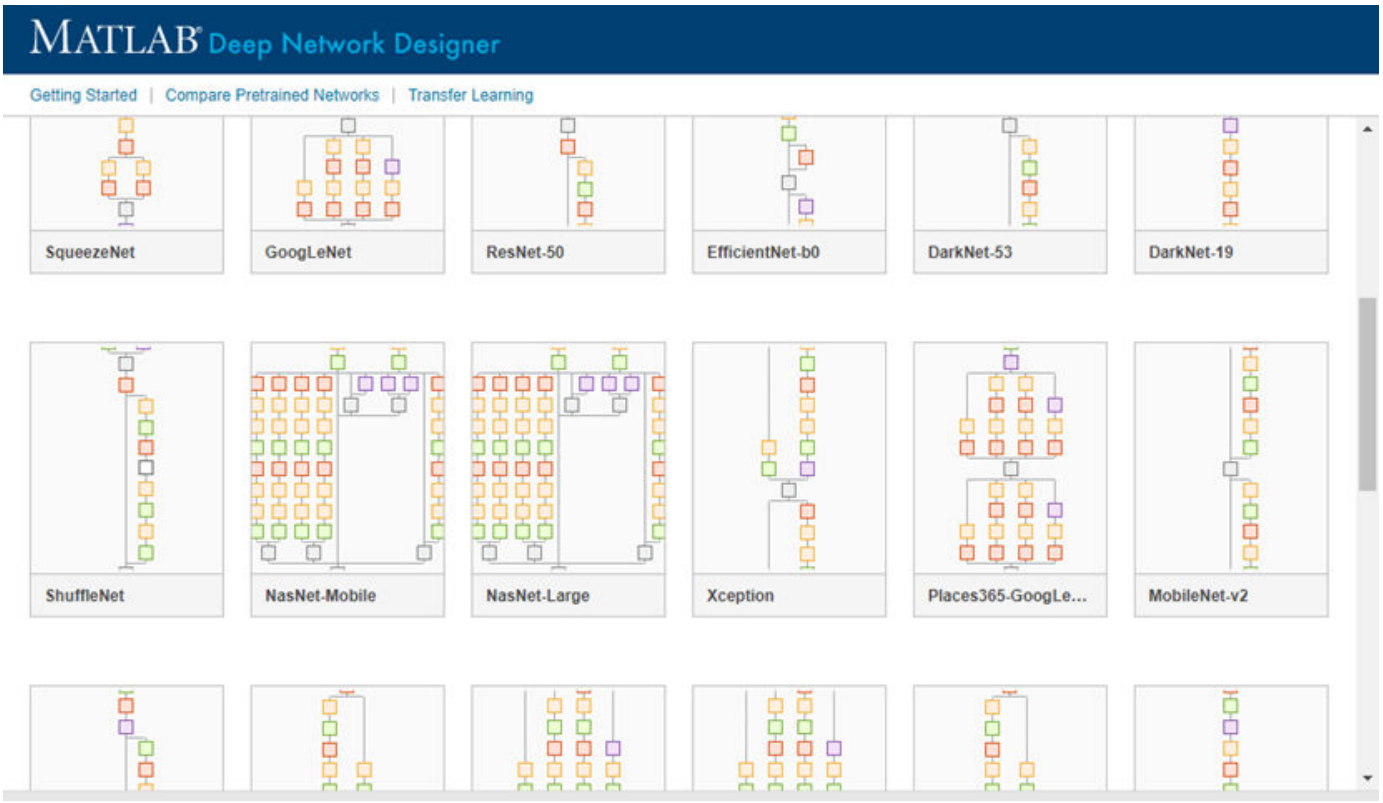
```
xception
```

```
ans =

  DAGNetwork with properties:

         Layers: [171×1 nnet.cnn.layer.Layer]
    Connections: [182×2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(xception)
```

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



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

Output Arguments

net — Pretrained Xception convolutional neural network

`DAGNetwork` object

Pretrained Xception convolutional neural network, returned as a `DAGNetwork` object.

`lgraph` — Untrained Xception convolutional neural network architecture
`LayerGraph` object

Untrained Xception convolutional neural network architecture, returned as a `LayerGraph` object.

References

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

[2] Chollet, F., 2017. "Xception: Deep Learning with Depthwise Separable Convolutions." *arXiv preprint*, pp.1610-02357.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = xception` or by passing the `xception` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('xception')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

The syntax `xception('Weights','none')` is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For code generation, you can load the network by using the syntax `net = xception` or by passing the `xception` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('xception')`

  For more information, see “Load Pretrained Networks for Code Generation” (GPU Coder).
- The syntax `xception('Weights','none')` is not supported for GPU code generation.

See Also

Deep Network Designer | `vgg16` | `vgg19` | `googlenet` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `resnet50` | `resnet101` | `inceptionresnetv2` | `squeezenet` | `densenet201`

Topics

“Transfer Learning with Deep Network Designer”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a

yamnet

YAMNet neural network

Syntax

```
net = yamnet
```

Description

`net = yamnet` returns a pretrained YAMNet model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download YAMNet

Download and unzip the Audio Toolbox™ model for YAMNet.

Type `yamnet` at the Command Window. If the Audio Toolbox model for YAMNet is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

Check that the installation is successful by typing `yamnet` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` object.

```
yamnet
```

```
ans =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

Load Pretrained YAMNet

Load a pretrained YAMNet convolutional neural network and examine the layers and classes.

Use `yamnet` to load the pretrained YAMNet network. The output net is a `SeriesNetwork` object.

```
net = yamnet
```

```
net =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

View the network architecture using the `Layers` property. The network has 86 layers. There are 28 layers with learnable weights: 27 convolutional layers, and 1 fully connected layer.

```
net.Layers
```

```
ans =
  86x1 Layer array with layers:

     1   'input_1'                   Image Input             96×64×1 images
     2   'conv2d'                    Convolution             32 3×3×1 convolutions with stride
     3   'b'                         Batch Normalization     Batch normalization with 32 channe
     4   'activation'                ReLU                    ReLU
     5   'depthwise_conv2d'          Grouped Convolution     32 groups of 1 3×3×1 convolutions
     6   'L11'                       Batch Normalization     Batch normalization with 32 channe
     7   'activation_1'              ReLU                    ReLU
     8   'conv2d_1'                  Convolution             64 1×1×32 convolutions with stride
     9   'L12'                       Batch Normalization     Batch normalization with 64 channe
    10   'activation_2'              ReLU                    ReLU
    11   'depthwise_conv2d_1'        Grouped Convolution     64 groups of 1 3×3×1 convolutions
    12   'L21'                       Batch Normalization     Batch normalization with 64 channe
    13   'activation_3'              ReLU                    ReLU
    14   'conv2d_2'                  Convolution             128 1×1×64 convolutions with strid
    15   'L22'                       Batch Normalization     Batch normalization with 128 chann
    16   'activation_4'              ReLU                    ReLU
    17   'depthwise_conv2d_2'        Grouped Convolution     128 groups of 1 3×3×1 convolutions
    18   'L31'                       Batch Normalization     Batch normalization with 128 chann
    19   'activation_5'              ReLU                    ReLU
    20   'conv2d_3'                  Convolution             128 1×1×128 convolutions with stri
    21   'L32'                       Batch Normalization     Batch normalization with 128 chann
    22   'activation_6'              ReLU                    ReLU
    23   'depthwise_conv2d_3'        Grouped Convolution     128 groups of 1 3×3×1 convolutions
    24   'L41'                       Batch Normalization     Batch normalization with 128 chann
    25   'activation_7'              ReLU                    ReLU
    26   'conv2d_4'                  Convolution             256 1×1×128 convolutions with stri
    27   'L42'                       Batch Normalization     Batch normalization with 256 chann
    28   'activation_8'              ReLU                    ReLU
    29   'depthwise_conv2d_4'        Grouped Convolution     256 groups of 1 3×3×1 convolutions
    30   'L51'                       Batch Normalization     Batch normalization with 256 chann
    31   'activation_9'              ReLU                    ReLU
    32   'conv2d_5'                  Convolution             256 1×1×256 convolutions with stri
    33   'L52'                       Batch Normalization     Batch normalization with 256 chann
    34   'activation_10'             ReLU                    ReLU
    35   'depthwise_conv2d_5'        Grouped Convolution     256 groups of 1 3×3×1 convolutions
    36   'L61'                       Batch Normalization     Batch normalization with 256 chann
    37   'activation_11'             ReLU                    ReLU
    38   'conv2d_6'                  Convolution             512 1×1×256 convolutions with stri
```

```
    39   'L62'                        Batch Normalization      Batch normalization with 512 chann
    40   'activation_12'              ReLU                     ReLU
    41   'depthwise_conv2d_6'         Grouped Convolution      512 groups of 1 3×3×1 convolutions
    42   'L71'                        Batch Normalization      Batch normalization with 512 chann
    43   'activation_13'              ReLU                     ReLU
    44   'conv2d_7'                   Convolution              512 1×1×512 convolutions with stri
    45   'L72'                        Batch Normalization      Batch normalization with 512 chann
    46   'activation_14'              ReLU                     ReLU
    47   'depthwise_conv2d_7'         Grouped Convolution      512 groups of 1 3×3×1 convolutions
    48   'L81'                        Batch Normalization      Batch normalization with 512 chann
    49   'activation_15'              ReLU                     ReLU
    50   'conv2d_8'                   Convolution              512 1×1×512 convolutions with stri
    51   'L82'                        Batch Normalization      Batch normalization with 512 chann
    52   'activation_16'              ReLU                     ReLU
    53   'depthwise_conv2d_8'         Grouped Convolution      512 groups of 1 3×3×1 convolutions
    54   'L91'                        Batch Normalization      Batch normalization with 512 chann
    55   'activation_17'              ReLU                     ReLU
    56   'conv2d_9'                   Convolution              512 1×1×512 convolutions with stri
    57   'L92'                        Batch Normalization      Batch normalization with 512 chann
    58   'activation_18'              ReLU                     ReLU
    59   'depthwise_conv2d_9'         Grouped Convolution      512 groups of 1 3×3×1 convolutions
    60   'L101'                       Batch Normalization      Batch normalization with 512 chann
    61   'activation_19'              ReLU                     ReLU
    62   'conv2d_10'                  Convolution              512 1×1×512 convolutions with stri
    63   'L102'                       Batch Normalization      Batch normalization with 512 chann
    64   'activation_20'              ReLU                     ReLU
    65   'depthwise_conv2d_10'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    66   'L111'                       Batch Normalization      Batch normalization with 512 chann
    67   'activation_21'              ReLU                     ReLU
    68   'conv2d_11'                  Convolution              512 1×1×512 convolutions with stri
    69   'L112'                       Batch Normalization      Batch normalization with 512 chann
    70   'activation_22'              ReLU                     ReLU
    71   'depthwise_conv2d_11'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    72   'L121'                       Batch Normalization      Batch normalization with 512 chann
    73   'activation_23'              ReLU                     ReLU
    74   'conv2d_12'                  Convolution              1024 1×1×512 convolutions with str
    75   'L122'                       Batch Normalization      Batch normalization with 1024 chan
    76   'activation_24'              ReLU                     ReLU
    77   'depthwise_conv2d_12'        Grouped Convolution      1024 groups of 1 3×3×1 convolution
    78   'L131'                       Batch Normalization      Batch normalization with 1024 chan
    79   'activation_25'              ReLU                     ReLU
    80   'conv2d_13'                  Convolution              1024 1×1×1024 convolutions with st
    81   'L132'                       Batch Normalization      Batch normalization with 1024 chan
    82   'activation_26'              ReLU                     ReLU
    83   'global_average_pooling2d'   Global Average Pooling   Global average pooling
    84   'dense'                      Fully Connected          521 fully connected layer
    85   'softmax'                    Softmax                  softmax
    86   'Sound'                      Classification Output    crossentropyex with 'Speech' and 5
```

To view the names of the classes learned by the network, you can view the `Classes` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

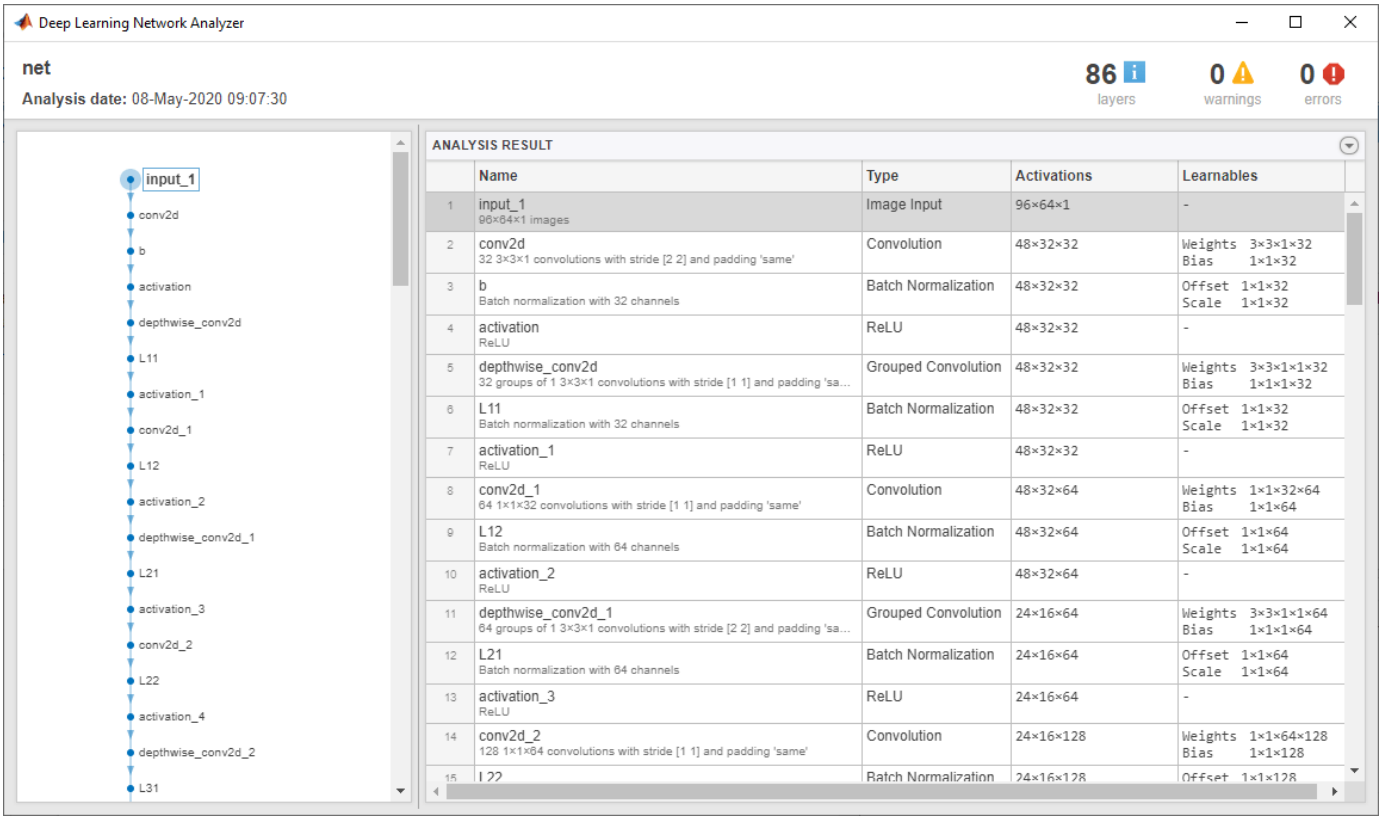
```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical
     Speech
     Child speech, kid speaking
```

```
Conversation
Narration, monologue
Babbling
Speech synthesizer
Shout
Bellow
Whoop
Yell
```

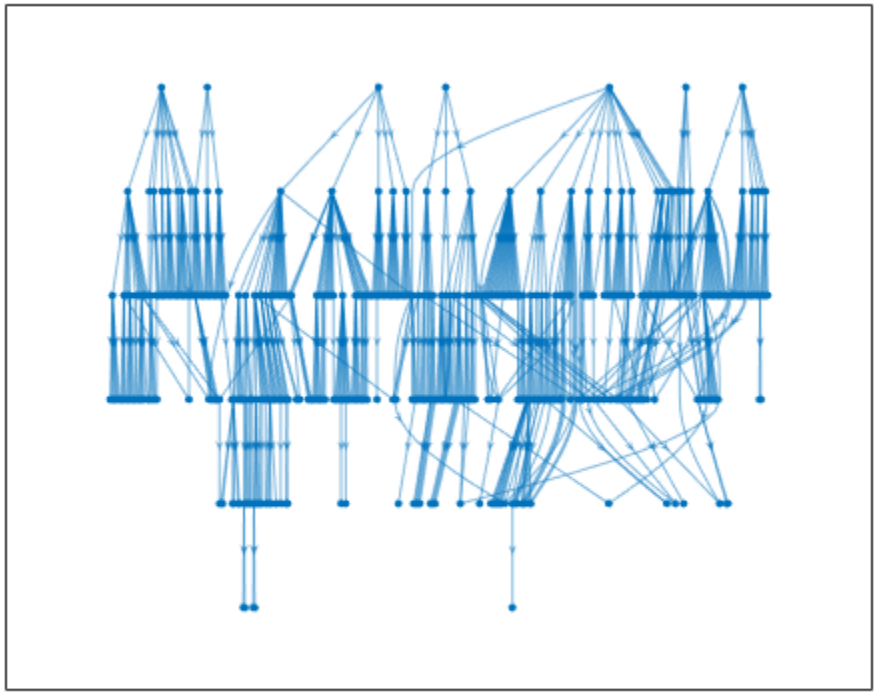
Use `analyzeNetwork` to visually explore the network.

```
analyzeNetwork(net)
```



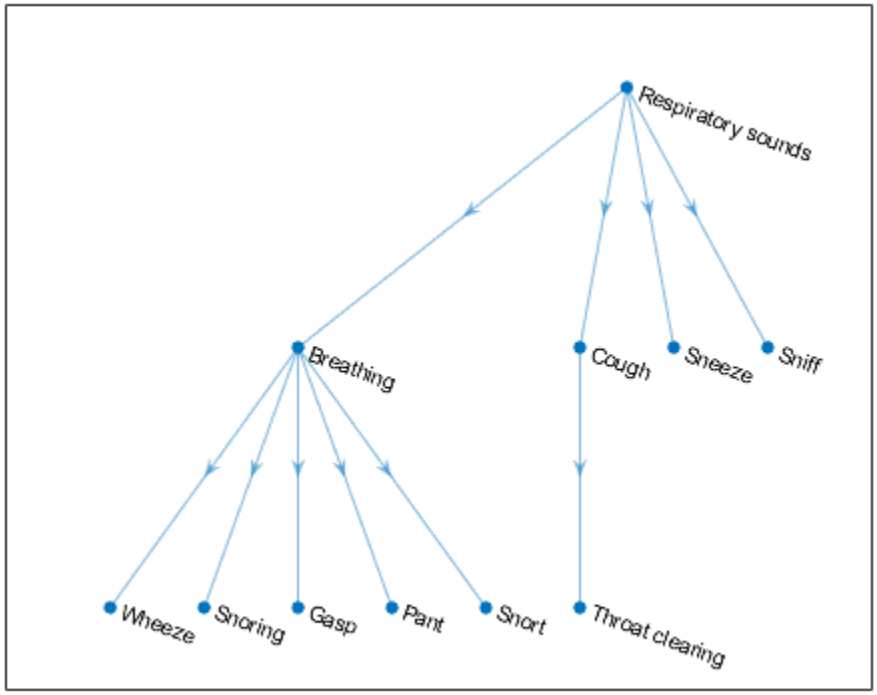
YAMNet was released with a corresponding sound class ontology, which you can explore using the `yamnetGraph` (Audio Toolbox) object.

```
ygraph = yamnetGraph;
p = plot(ygraph);
layout(p,'layered')
```



The ontology graph plots all 521 possible sound classes. Plot a subgraph of the sounds related to respiratory sounds.

```
allRespiratorySounds = dfsearch(ygraph,"Respiratory sounds");
ygraphSpeech = subgraph(ygraph,allRespiratorySounds);
plot(ygraphSpeech)
```



Classify Sounds Using YAMNet

The YAMNet network requires you to preprocess and extract features from audio signals by converting them to the sample rate the network was trained on, and then extracting overlapping log-mel spectrograms. This example walks through the required preprocessing and feature extraction necessary to match the preprocessing and feature extraction used to train YAMNet. The `classifySound` (Audio Toolbox) function performs these steps for you.

Read in an audio signal to classify it. Resample the audio signal to 16 kHz and then convert it to single precision.

```
[audioIn,fs0] = audioread('Counting-16-44p1-mono-15secs.wav');

fs = 16e3;
audioIn = resample(audioIn,fs,fs0);

audioIn = single(audioIn);
```

Define mel spectrogram parameters and then extract features using the `melSpectrogram` (Audio Toolbox) function.

```
FFTLength = 512;
numBands = 64;
frequencyRange = [125 7500];
```

```
windowLength = 0.025*fs;
overlapLength = 0.015*fs;

melSpect = melSpectrogram(audioIn,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');
```

Convert the mel spectrogram to the log scale.

```
melSpect = log(melSpect + single(0.001));
```

Reorient the mel spectrogram so that time is along the first dimension as rows.

```
melSpect = melSpect.';
[numSTFTWindows,numBands] = size(melSpect)
```

```
numSTFTWindows = 1551
```

```
numBands = 64
```

Partition the spectrogram into frames of length 96 with an overlap of 48. Place the frames along the fourth dimension.

```
frameWindowLength = 96;
frameOverlapLength = 48;

hopLength = frameWindowLength - frameOverlapLength;
numHops = floor((numSTFTWindows - frameWindowLength)/hopLength) + 1;

frames = zeros(frameWindowLength,numBands,1,numHops,'like',melSpect);
for hop = 1:numHops
    range = 1 + hopLength*(hop-1):hopLength*(hop - 1) + frameWindowLength;
    frames(:,:,1,hop) = melSpect(range,:);
end
```

Create a YAMNet network.

```
net = yamnet();
```

Classify the spectrogram images.

```
classes = classify(net,frames);
```

Classify the audio signal as the most frequently occurring sound.

```
  mySound = mode(classes)
```

```
mySound = categorical
     Speech
```

Transfer Learning Using YAMNet

Download and unzip the air compressor data set [1] on page 1-0 . This data set consists of recordings from air compressors in a healthy state or one of 7 faulty states.

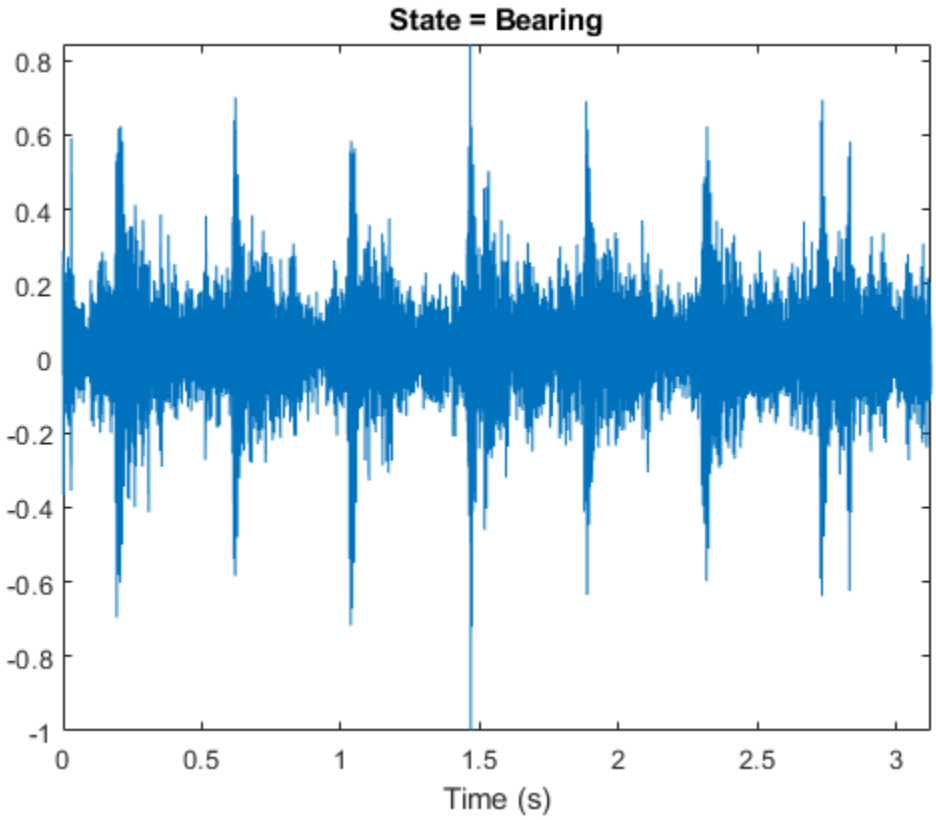
```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'aircompressordataset');
datasetLocation = tempdir;

if ~exist(fullfile(tempdir,'AirCompressorDataSet'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'AirCompressorDataSet'))
end
```

Create an `audioDatastore` (Audio Toolbox) object to manage the data and split it into train and validation sets.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');

[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

Read an audio file from the datastore and save the sample rate for later use. Reset the datastore to return the read pointer to the beginning of the data set. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;

reset(adsTrain)

sound(x,fs)

figure
t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel('Time (s)')
title('State = ' + string(fileInfo.Label))
axis tight
```



Extract Mel spectrograms from the train set using `yamnetPreprocess`. There are multiple spectrograms for each audio signal. Replicate the labels so that they are in one-to-one correspondence with the spectrograms.

```
emptyLabelVector = adsTrain.Labels;
emptyLabelVector(:) = [];

trainFeatures = [];
trainLabels = emptyLabelVector;
while hasdata(adsTrain)
    [audioIn,fileInfo] = read(adsTrain);
    features = yamnetPreprocess(audioIn,fileInfo.SampleRate);
    numSpectrums = size(features,4);
    trainFeatures = cat(4,trainFeatures,features);
    trainLabels = cat(2,trainLabels,repmat(fileInfo.Label,1,numSpectrums));
end
```

Extract features from the validation set and replicate the labels.

```
validationFeatures = [];
validationLabels = emptyLabelVector;
while hasdata(adsValidation)
    [audioIn,fileInfo] = read(adsValidation);
    features = yamnetPreprocess(audioIn,fileInfo.SampleRate);
    numSpectrums = size(features,4);
    validationFeatures = cat(4,validationFeatures,features);
    validationLabels = cat(2,validationLabels,repmat(fileInfo.Label,1,numSpectrums));
end
```

The air compressor data set has only eight classes.

Read in YAMNet and convert it to a `layerGraph`.

If YAMNet pretrained network is not installed on your machine, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

After you read in YAMNet and convert it to a `layerGraph`, replace the final `fullyConnectedLayer` and the final `classificationLayer` to reflect the new task.

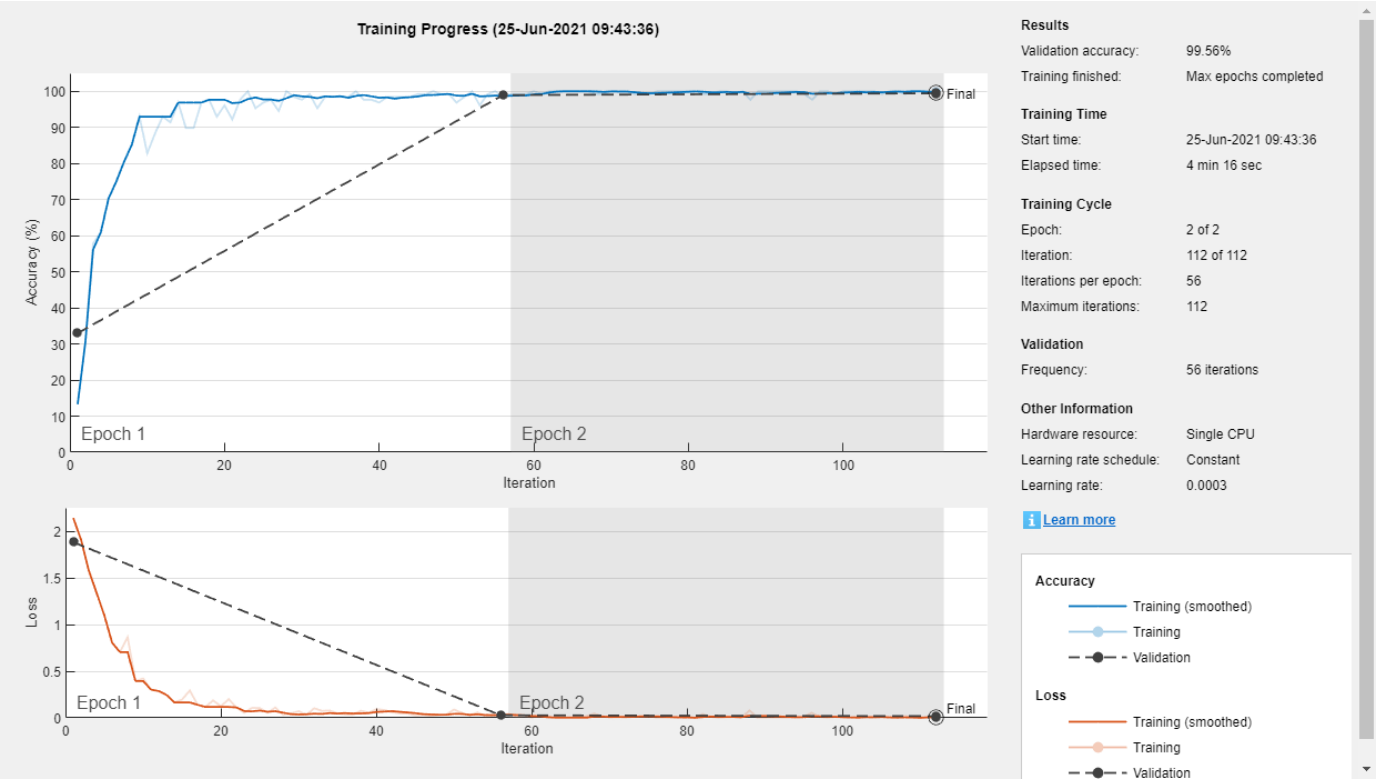
```
uniqueLabels = unique(adsTrain.Labels);
numLabels = numel(uniqueLabels);

net = yamnet;

lgraph = layerGraph(net.Layers);

newDenseLayer = fullyConnectedLayer(numLabels,"Name","dense");
lgraph = replaceLayer(lgraph,"dense",newDenseLayer);

newClassificationLayer = classificationLayer("Name","Sounds","Classes",uniqueLabels);
lgraph = replaceLayer(lgraph,"Sound",newClassificationLayer);
```

To define training options, use `trainingOptions`.

```
miniBatchSize = 128;
validationFrequency = floor(numel(trainLabels)/miniBatchSize);
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'MaxEpochs',2, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{single(validationFeatures),validationLabels}, ...
    'ValidationFrequency',validationFrequency);
```

To train the network, use `trainNetwork`.

```
airCompressorNet = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```



Save the trained network to `airCompressorNet.mat`. You can now use this pre-trained network by loading the `airCompressorNet.mat` file.

```
save airCompressorNet.mat airCompressorNet
```

References

[1] Verma, Nishchal K., et al. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. *DOI.org (Crossref)*, doi:10.1109/TR.2015.2459684.

Output Arguments

net — Pretrained YAMNet neural network

`SeriesNetwork` object

Pretrained YAMNet neural network, returned as a `SeriesNetwork` object.

References

[1] Gemmeke, Jort F., et al. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" (GPU Coder).

See Also

Apps

Audio Labeler

Blocks

Sound Classifier | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `designAuditoryFilterBank` | `melSpectrogram` | `vggish` | `vggishPreprocess` | `yamnetGraph` | `yamnetPreprocess`

Introduced in R2020b

Approximation, Clustering, and Control Functions

adapt

Adapt neural network to data as it is simulated

Syntax

```
[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)
```

To Get Help

Type `help network/adapt`.

Description

This function calculates network outputs and errors after each presentation of an input.

`[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)` takes

`net`	Network
`P`	Network inputs
`T`	Network targets (default = zeros)
`Pi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

`net`	Updated network
`Y`	Network outputs
`E`	Network errors
`Pf`	Final input delay conditions
`Af`	Final layer delay conditions
`tr`	Training record (`epoch` and `perf`)

Note that `T` is optional and is only needed for networks that require targets. `Pi` and `Pf` are also optional and only need to be used for networks that have input or layer delays.

`adapt`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented,

`P`	`Ni`-by-`TS` cell array	Each element `P{i,ts}` is an `Ri`-by-`Q` matrix.
`T`	`Nt`-by-`TS` cell array	Each element `T{i,ts}` is a `Vi`-by-`Q` matrix.

`Pi`	`Ni`-by-`ID` cell array	Each element `Pi{i,k}` is an `Ri`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.
`Y`	`No`-by-`TS` cell array	Each element `Y{i,ts}` is a `Ui`-by-`Q` matrix.
`E`	`No`-by-`TS` cell array	Each element `E{i,ts}` is a `Ui`-by-`Q` matrix.
`Pf`	`Ni`-by-`ID` cell array	Each element `Pf{i,k}` is an `Ri`-by-`Q` matrix.
`Af`	`Nl`-by-`LD` cell array	Each element `Af{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`No`	=	`net.numOutputs`
`ID`	=	`net.numInputDelays`
`LD`	=	`net.numLayerDelays`
`TS`	=	Number of time steps
`Q`	=	Batch size
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Ui`	=	`net.outputs{i}.size`

The columns of `Pi`, `Pf`, `Ai`, and `Af` are ordered from oldest delay condition to most recent:

`Pi{i,k}`	=	Input `i` at time `ts = k - ID`
`Pf{i,k}`	=	Input `i` at time `ts = TS + k - ID`
`Ai{i,k}`	=	Layer output `i` at time `ts = k - LD`
`Af{i,k}`	=	Layer output `i` at time `ts = TS + k - LD`

The matrix format can be used if only one time step is to be simulated (`TS = 1`). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

`P`	(`sum of Ri`)-by-`Q` matrix
`T`	(`sum of Vi`)-by-`Q` matrix
`Pi`	(`sum of Ri`)-by-(`ID*Q`) matrix
`Ai`	(`sum of Si`)-by-(`LD*Q`) matrix
`Y`	(`sum of Ui`)-by-`Q` matrix
`E`	(`sum of Ui`)-by-`Q` matrix

Pf	(sum of Ri)-by-(ID*Q) matrix
Af	(sum of Si)-by-(LD*Q) matrix

Examples

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.

```
p1 = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t1 = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
```

Here linearlayer is used to create a layer with an input range of [-1 1], one neuron, input delays of 0 and 1, and a learning rate of 0.1. The linear layer is then simulated.

```
net = linearlayer([0 1],0.1);
```

Here the network adapts for one pass through the sequence.

The network's mean squared error is displayed. (Because this is the first call to adapt, the default Pi is used.)

```
[net,y,e,pf] = adapt(net,p1,t1);
mse(e)
```

Note that the errors are quite large. Here the network adapts to another 12 time steps (using the previous Pf as the new initial delay conditions).

```
p2 = {1 -1 -1 1 1 -1  0 0 0 1 -1 -1};
t2 = {2  0 -2 0 2  0 -1 0 0 1  0 -1};
[net,y,e,pf] = adapt(net,p2,t2,pf);
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```
p3 = [p1 p2];
t3 = [t1 t2];
for i = 1:100
  [net,y,e] = adapt(net,p3,t3);
end
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

Algorithms

adapt calls the function indicated by net.adaptFcn, using the adaption parameter values indicated by net.adaptParam.

Given an input sequence with TS steps, the network is updated as follows: Each step in the sequence of inputs is presented to the network one at a time. The network's weight and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated TS times.

See Also

sim | init | train | revert

Introduced before R2006a

adaptwb

Adapt network with weight and bias learning rules

Syntax

```
[net,ar,Ac] = adapt(net,Pd,T,Ai)
```

Description

This function is normally not called directly, but instead called indirectly through the function `adapt` after setting a network's adaption function (`net.adaptFcn`) to this function.

`[net,ar,Ac] = adapt(net,Pd,T,Ai)` takes these arguments,

`net`	Neural network
`Pd`	Delayed processed input states and inputs
`T`	Targets
`Ai`	Initial layer delay states

and returns

`net`	Neural network after adaption
`ar`	Adaption record
`Ac`	Combined initial layer states and layer outputs

Examples

Linear layers use this adaption function. Here a linear layer with input delays of 0 and 1, and a learning rate of 0.5, is created and adapted to produce some target data `t` when given some input data `x`. The response is then plotted, showing the network's error going down over time.

```
x = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
net = linearlayer([0 1],0.5);
net.adaptFcn
[net,y,e,xf] = adapt(net,x,t);
plotresponse(t,y)
```

See Also

`adapt`

Introduced in R2010b

adddelay

Add delay to neural network response

Syntax

```
net = adddelay(net,n)
```

Description

`net = adddelay(net,n)` takes these arguments,

`net`	Neural network
`n`	Number of delays

and returns the network with input delay connections increased, and output feedback delays decreased, by the specified number of delays `n`. The result is a network that behaves identically, except that outputs are produced `n` timesteps later.

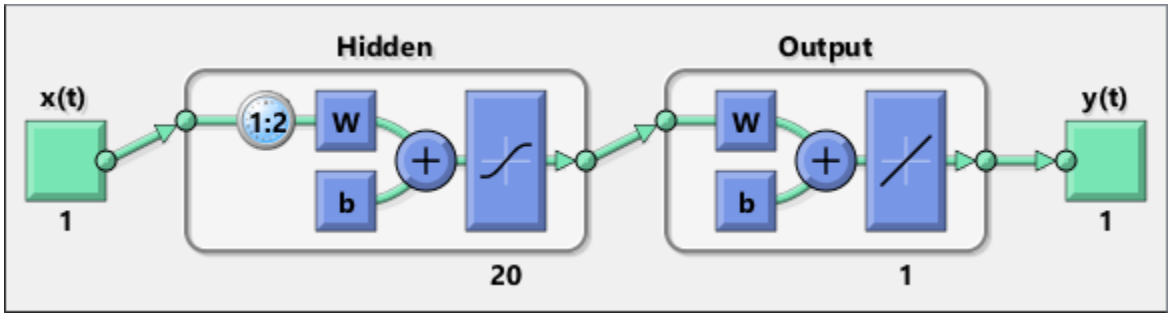
If the number of delays `n` is not specified, a default of one delay is used.

Examples

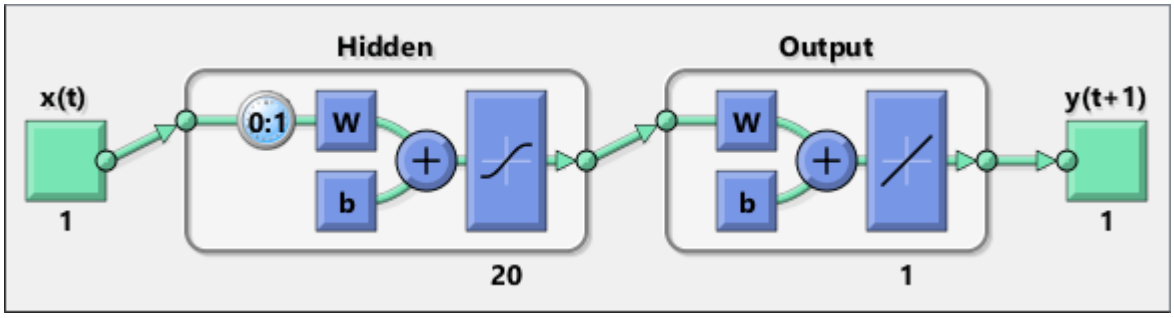
Time Delay Network

This example creates, trains, and simulates a time delay network in its original form, on an input time series `X` and target series `T`. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following step.

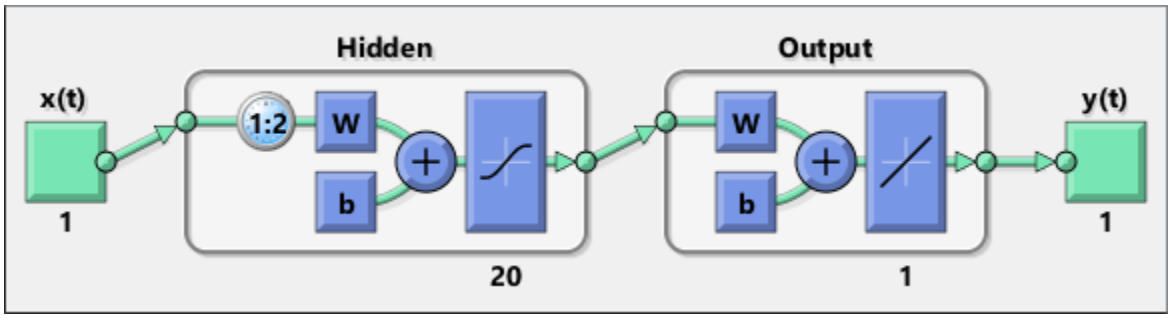
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```




```
net2 = removedelay(net1);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi);
view(net2)
```




```
net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
```



See Also

`closeloop` | `openloop` | `removedelay`

Introduced in R2010b

boxdist

Distance between two position vectors

Syntax

```
d = boxdist(pos)
```

Description

`boxdist` is a layer distance function that is used to find the distances between the layer's neurons, given their positions.

`d = boxdist(pos)` takes one argument,

`pos`	N-by-S matrix of neuron positions

and returns the S-by-S matrix of distances.

`boxdist` is most commonly used with layers whose topology function is `gridtop`.

Examples

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
d = boxdist(pos)
```

Network Use

To change a network so that a layer's topology uses `boxdist`, set `net.layers{i}.distanceFcn` to `'boxdist'`.

In either case, call `sim` to simulate the network with `boxdist`.

Algorithms

The box distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

```
Dij = max(abs(Pi-Pj))
```

See Also

`dist` | `linkdist` | `mandist` | `sim`

Introduced before R2006a

bttderiv

Backpropagation through time derivative function

Syntax

```
bttderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
bttderiv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the chain rule from a network's performance back through the network, and in the case of dynamic networks, back through time.

`bttderiv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

`bttderiv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = bttderiv('dperf_dwb',net,x,t)
jwb = bttderiv('de_dwb',net,x,t)
```

See Also

`defaultderiv` | `fpderiv` | `num2deriv` | `num5deriv` | `staticderiv`

Introduced in R2010b

cascadeforwardnet

Generate cascade-forward neural network

Syntax

```
net = cascadeforwardnet(hiddenSizes,trainFcn)
```

Description

`net = cascadeforwardnet(hiddenSizes,trainFcn)` returns a cascade-forward neural network with a hidden layer size of `hiddenSizes` and training function, specified by `trainFcn`.

Cascade-forward networks are similar to feed-forward networks, but include a connection from the input and every previous layer to following layers.

As with feed-forward networks, a two-or more layer cascade-network can learn any finite input-output relationship arbitrarily well given enough hidden neurons.

Examples

Construct and Train a Cascade-Forward Neural Network

This example shows how to use a cascade-forward neural network to solve a simple problem.

Load the training data.

```
[x,t] = simplefit_dataset;
```

The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a cascade-forward network with one hidden layer of size 10.

```
net = cascadeforwardnet(10);
```

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```

Estimate the targets using the trained network.

```
y = net(x);
```

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,y,t)
```

```
perf = 1.9372e-05
```

Input Arguments

`hiddenSizes` — Size of the hidden layers

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: `[10,8,5]`

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

`trainFcn` — Training function name

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

Output Arguments

`net` — Cascade-forward network

`network` object

Cascade-forward neural network, returned as a `network` object.

See Also

`feedforwardnet` | `network`

Topics

“Create, Configure, and Initialize Multilayer Shallow Neural Networks”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b

catelements

Concatenate neural network data elements

Syntax

```
catelements(x1,x2,...,xn)
[x1; x2; ... xn]
```

Description

`catelements(x1,x2,...,xn)` takes any number of neural network data values, and merges them along the element dimension (i.e., the matrix row dimension).

If all arguments are matrices, this operation is the same as `[x1; x2; ... xn]`.

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and then the matrices in the same positions in each argument are concatenated.

Examples

This code concatenates the elements of two matrix data values.

```
x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6; 2 9 1]
y = catelements(x1,x2)
```

This code concatenates the elements of two cell array data values.

```
x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3] [4 5 6]; [2 5 4] [9 7 5]}
y = catelements(x1,x2)
```

See Also

`nndata` | `numelements` | `getelements` | `setelements` | `catsignals` | `catsamples` | `cattimesteps`

Introduced in R2010b

catsamples

Concatenate neural network data samples

Syntax

```
catsamples(x1,x2,...,xn)
[x1 x2 ... xn]
catsamples(x1,x2,...,xn,'pad',v)
```

Description

`catsamples(x1,x2,...,xn)` takes any number of neural network data values, and merges them along the samples dimension (i.e., the matrix column dimension).

If all arguments are matrices, this operation is the same as `[x1 x2 ... xn]`.

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and then the matrices in the same positions in each argument are concatenated.

`catsamples(x1,x2,...,xn,'pad',v)` allows samples with varying numbers of timesteps (columns of cell arrays) to be concatenated by padding the shorter time series with the value `v`, until they are the same length as the longest series. If `v` is not specified, then the value `NaN` is used, which is often used to represent unknown or don't-care inputs or targets.

Examples

This code concatenates the samples of two matrix data values.

```
x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6]
y = catsamples(x1,x2)
```

This code concatenates the samples of two cell array data values.

```
x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3; 5 4 1] [4 5 6; 9 4 8]; [2 5 4] [9 7 5]}
y = catsamples(x1,x2)
```

Here the samples of two cell array data values, with unequal numbers of timesteps, are concatenated.

```
x1 = {1 2 3 4 5};
x2 = {10 11 12};
y = catsamples(x1,x2,'pad')
```

See Also

`nndata` | `numsamples` | `getsamples` | `setsamples` | `catelements` | `catsignals` | `cattimesteps`

Introduced in R2010b

catsignals

Concatenate neural network data signals

Syntax

```
catsignals(x1,x2,...,xn)
{x1; x2; ...; xn}
```

Description

`catsignals(x1,x2,...,xn)` takes any number of neural network data values, and merges them along the element dimension (i.e., the cell row dimension).

If all arguments are matrices, this operation is the same as `{x1; x2; ...; xn}`.

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and the cell arrays are concatenated as `[x1; x2; ...; xn]`.

Examples

This code concatenates the signals of two matrix data values.

```
x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6]
y = catsignals(x1,x2)
```

This code concatenates the signals of two cell array data values.

```
x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3; 5 4 1] [4 5 6; 9 4 8]; [2 5 4] [9 7 5]}
y = catsignals(x1,x2)
```

See Also

`nndata` | `numsignals` | `getsignals` | `setsignals` | `catelements` | `catsamples` | `cattimesteps`

Introduced in R2010b

cattimesteps

Concatenate neural network data timesteps

Syntax

```
cattimesteps(x1,x2,...,xn)
{x1 x2 ... xn}
```

Description

`cattimesteps(x1,x2,...,xn)` takes any number of neural network data values, and merges them along the element dimension (i.e., the cell column dimension).

If all arguments are matrices, this operation is the same as `{x1 x2 ... xn}`.

If any argument is a cell array, all non-cell array arguments are enclosed in cell arrays, and the cell arrays are concatenated as `[x1 x2 ... xn]`.

Examples

This code concatenates the elements of two matrix data values.

```
x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6]
y = cattimesteps(x1,x2)
```

This code concatenates the elements of two cell array data values.

```
x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3; 5 4 1] [4 5 6; 9 4 8]; [2 5 4] [9 7 5]}
y = cattimesteps(x1,x2)
```

See Also

`nndata` | `numtimesteps` | `gettimesteps` | `settimesteps` | `catelements` | `catsignals` | `catsamples`

Introduced in R2010b

cellmat

Create cell array of matrices

Syntax

```
cellmat(A,B,C,D,v)
```

Description

`cellmat(A,B,C,D,v)` takes four integer values and one scalar value `v`, and returns an `A`-by-`B` cell array of `C`-by-`D` matrices of value `v`. If the value `v` is not specified, zero is used.

Examples

Here two cell arrays of matrices are created.

```
cm1 = cellmat(2,3,5,4)
cm2 = cellmat(3,4,2,2,pi)
```

See Also

`nndata`

Introduced in R2010b

closeloop

Convert neural network open-loop feedback to closed loop

Syntax

```
net = closeloop(net)
[net,xi,ai] = closeloop(net,xi,ai)
```

Description

`net = closeloop(net)` takes a neural network and closes any open-loop feedback. For each feedback output `i` whose property `net.outputs{i}.feedbackMode` is `'open'`, it replaces its associated feedback input and their input weights with layer weight connections coming from the output. The `net.outputs{i}.feedbackMode` property is set to `'closed'`, and the `net.outputs{i}.feedbackInput` property is set to an empty matrix. Finally, the value of `net.outputs{i}.feedbackDelays` is added to the delays of the feedback layer weights (i.e., to the delays values of the replaced input weights).

`[net,xi,ai] = closeloop(net,xi,ai)` converts an open-loop network and its current input delay states `xi` and layer delay states `ai` to closed-loop form.

Examples

Convert NARX Network to Closed-Loop Form

This example shows how to design a NARX network in open-loop form, then convert it to closed-loop form.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Yopen = net(Xs,Xi,Ai)
net = closeloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yclosed = net(Xs,Xi,Ai);
```

Convert Delay States

For examples on using `closeloop` and `openloop` to implement multistep prediction, see `narxnet` and `narnet`.

See Also

`narnet` | `narxnet` | `noloop` | `openloop`

Introduced in R2010b

combvec

Create all combinations of vectors

Syntax

```
A = combvec(A1,A2,...)
```

Description

`A = combvec(A1,A2,...)` takes any number of inputs `A`, where each input `Ai` has `Ni` columns, and returns a matrix of `(N1*N2*...)` column vectors, where the columns consist of all combinations found by combining one column vector from each input `Ai`.

Examples

Generate All Combinations of Vectors Using the combvec Function

This example shows how to generate a matrix that contains all combinations of two matrices, `a1` and `a2`.

Create the two input matrices, `a1` and `a2`. Then call the `combvec` function to generate all possible combinations.

```
a1 = [1 2 3; 4 5 6];
a2 = [7 8; 9 10];
a3 = combvec(a1,a2)
```

```
a3 =

     1     2     3     1     2     3
     4     5     6     4     5     6
     7     7     7     8     8     8
     9     9     9    10    10    10
```

Input Arguments

A1 — Input matrix 1

matrix

Input matrix of which you want to calculate all combinations, specified as a matrix with `N1` column vectors.

A2 — Input matrix 2

matrix

Input matrix of which you want to calculate all combinations, specified as a matrix with `N2` column vectors.

Output Arguments

`A` — Output matrix

matrix

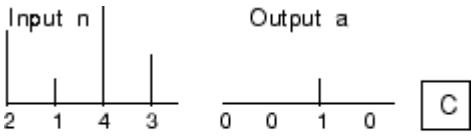
Output matrix, returned as a matrix of `(N1*N2*...)` column vectors, where the columns consist of all combinations found by combining one column vector from each input `Ai`.

Introduced before R2006a

compet

Competitive transfer function

Graph and Symbol



Compet Transfer Function

Syntax

```
A = compet(N,FP)
info = compet('code')
```

Description

`compet` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = compet(N,FP)` takes `N` and optional function parameters,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns the S-by-Q matrix `A` with a `1` in each column where the same column of `N` has its maximum value, and `0` elsewhere.

`info = compet('code')` returns information according to the code string specified:

`compet('name')` returns the name of this function.

`compet('output',FP)` returns the `[min max]` output range.

`compet('active',FP)` returns the `[min max]` active input range.

`compet('fullderiv')` returns `1` or `0`, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`compet('fpnames')` returns the names of the function parameters.

`compet('fpdefaults')` returns the default function parameters.

Examples

Here you define a net input vector `N`, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = compet(n);
```

```
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'compet';
```

See Also

`sim` | `softmax`

Introduced before R2006a

competlayer

Competitive layer

Syntax

```
competlayer(numClasses,kohonenLR,conscienceLR)
```

Description

Competitive layers learn to classify input vectors into a given number of classes, according to similarity between vectors, with a preference for equal numbers of vectors per class.

`competlayer(numClasses,kohonenLR,conscienceLR)` takes these arguments,

`numClasses`	Number of classes to classify inputs (default = 5)
`kohonenLR`	Learning rate for Kohonen weights (default = 0.01)
`conscienceLR`	Learning rate for conscience bias (default = 0.001)

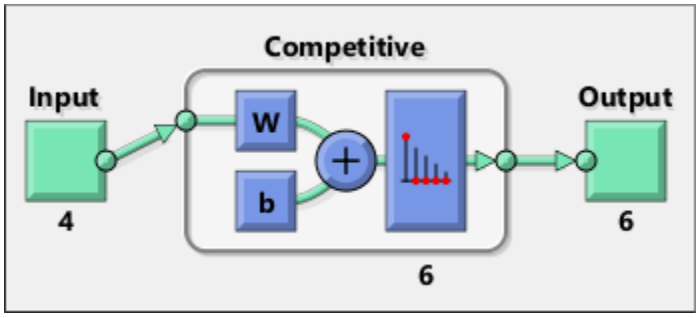
and returns a competitive layer with `numClasses` neurons.

Examples

Create and Train a Competitive Layer

Here a competitive layer is trained to classify 150 iris flowers into 6 classes.

```
inputs = iris_dataset;
net = competlayer(6);
net = train(net,inputs);
view(net)
outputs = net(inputs);
classes = vec2ind(outputs);
```



See Also

`selforgmap` | `patternnet` | `lvqnet`

Introduced in R2010b

con2seq

Convert concurrent vectors to sequential vectors

Syntax

```
S = con2seq(b)
S = con2seq(b,TS)
```

Description

Deep Learning Toolbox software arranges concurrent vectors with a matrix, and sequential vectors with a cell array (where the second index is the time step).

`con2seq` and `seq2con` allow concurrent vectors to be converted to sequential vectors, and back again.

`S = con2seq(b)` takes one input,

`b`	`R`-by-`TS` matrix

and returns one output,

`S`	1-by-`TS` cell array of `R`-by-1 vectors

`S = con2seq(b,TS)` can also convert multiple batches,

`b`	`N`-by-1 cell array of matrices with `M`*`TS` columns
`TS`	Time steps

and returns

`S`	`N`-by-`TS` cell array of matrices with `M` columns

Examples

Here a batch of three values is converted to a sequence.

```
p1 = [1 4 2]
p2 = con2seq(p1)
```

Here, two batches of vectors are converted to two sequences with two time steps.

```
p1 = {[1 3 4 5; 1 1 7 4]; [7 3 4 4; 6 9 4 1]}
p2 = con2seq(p1,2)
```

See Also

`seq2con` | `concur`

Introduced before R2006a

concur

Create concurrent bias vectors

Syntax

```
concur(B,Q)
```

Description

`concur(B,Q)`

`B`	`S`-by-`1` bias vector (or an `Nl`-by-`1` cell array of vectors)
`Q`	Concurrent size

and returns an `S`-by-`B` matrix of copies of `B` (or an `Nl`-by-`1` cell array of matrices).

Examples

Here `concur` creates three copies of a bias vector.

```
b = [1; 3; 2; -1];
concur(b,3)
```

Network Use

To calculate a layer's net input, the layer's weighted inputs must be combined with its biases. The following expression calculates the net input for a layer with the `netsum` net input function, two input weights, and a bias:

```
n = netsum(z1,z2,b)
```

The above expression works if `Z1`, `Z2`, and `B` are all `S`-by-`1` vectors. However, if the network is being simulated by `sim` (or `adapt` or `train`) in response to `Q` concurrent vectors, then `Z1` and `Z2` will be `S`-by-`Q` matrices. Before `B` can be combined with `Z1` and `Z2`, you must make `Q` copies of it.

```
n = netsum(z1,z2,concur(b,q))
```

See Also

`con2seq` | `netprod` | `netsum` | `seq2con` | `sim`

Introduced before R2006a

configure

Configure network inputs and outputs to best match input and target data

Syntax

```
net = configure(net,x,t)
net = configure(net,x)
net = configure(net,'inputs',x,i)
net = configure(net,'outputs',t,i)
```

Description

`net = configure(net,x,t)` takes input data `x` and target data `t`, and configures the network's inputs and outputs to match.

Configuration is the process of setting network input and output sizes and ranges, input preprocessing settings and output postprocessing settings, and weight initialization settings to match input and target data.

Configuration must happen before a network's weights and biases can be initialized. Unconfigured networks are automatically configured and initialized the first time `train` is called. Alternately, a network can be configured manually either by calling this function or by setting a network's input and output sizes, ranges, processing settings, and initialization settings properties manually.

`net = configure(net,x)` configures only inputs.

`net = configure(net,'inputs',x,i)` configures the inputs specified with the index vector `i`. If `i` is not specified all inputs are configured.

`net = configure(net,'outputs',t,i)` configures the outputs specified with the index vector `i`. If `i` is not specified all targets are configured.

Examples

Configure Network with `'configure'`

This example shows how to manually configure a network for a simple fitting problem instead of using the `train` function.

Create a feedforward network and manually configure it for a simple fitting problem.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20); view(net)
net = configure(net,x,t); view(net)
```

Input Arguments

net — Network to configure

`network` object

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

`x` — Input data
matrix

Network inputs, specified as a matrix.

`t` — Target data
matrix

Network targets, specified as a matrix.

`i` — Index vector
vector

Indexes of the inputs or outputs you want to configure, specified as a vector.

Output Arguments

`net` — Configured network
`network` object

Configured network, returned as a network object.

See Also

`isconfigured` | `unconfigure` | `init` | `train`

Introduced in R2010b

confusion

Classification confusion matrix

Syntax

```
[c,cm,ind,per] = confusion(targets,outputs)
```

Description

Tip To plot a confusion chart for a deep learning workflow, use the `confusionchart` function.

`[c,cm,ind,per] = confusion(targets,outputs)` takes target and output matrices, `targets` and `outputs`, and returns the confusion value, `c`, the confusion matrix, `cm`, a cell array, `ind`, that contains the sample indices of class `i` targets classified as class `j`, and a matrix of percentages, `per`, where each row summarizes four percentages associated with the `i`-th class.

Examples

Generate the Confusion Matrix of the `simpleclass` Dataset Using the `confusion` Function

This example shows how to generate the confusion matrix of the `simpleclass_dataset` dataset using the `confusion` function.

Load the `simpleclass_dataset` dataset. Define a network and then train it.

```
[x,t] = simpleclass_dataset;
net = patternnet(10);
net = train(net,x,t);
y = net(x);
[c,cm,ind,per] = confusion(t,y)
```

```
a3 =

     1     2     3     1     2     3
     4     5     6     4     5     6
     7     7     7     8     8     8
     9     9     9    10    10    10
```

Input Arguments

`targets` — Matrix of targets

matrix

Matrix of targets, specified as an `S`-by-`Q` matrix, where each column vector contains a single `1` value, with all other elements equal to `0`. The index of the value equal to `1` indicates which of the `S` categories that vector represents.

`outputs` — Matrix of outputs
matrix

Matrix of outputs, specified as an `S`-by-`Q` matrix, where each column contains values in the range `[0,1]`. The index of the largest element in the column indicates which of the `S` categories that vector represents.

Output Arguments

`c` — Confusion value
scalar

Fraction of misclassified samples, returned as a scalar.

`cm` — Confusion matrix
matrix

Confusion matrix, returned as an `S`-by-`S` confusion matrix, where `cm(i,j)` is the number of samples whose target is the `i`-th class that was classified as `j`.

`ind` — Array of indices
cell array

Array of indices, returned as an `S`-by-`S` cell array, where `ind{i,j}` contains the indices of samples with the `i`-th target class, but `j`-th output class.

`per` — Matrix of percentages
matrix

Matrix of percentages, returned as an `S`-by-`4` matrix, where each row summarizes four percentages associated with the `i`-th class:

```
per(i,1) false negative rate
          = (false negatives)/(all output negatives)
per(i,2) false positive rate
          = (false positives)/(all output positives)
per(i,3) true positive rate
          = (true positives)/(all output positives)
per(i,4) true negative rate
          = (true negatives)/(all output negatives)
```

See Also
`plotconfusion` | `roc`

Introduced in R2006a

convwf

Convolution weight function

Syntax

```
Z = convwf(W,P)
dim = convwf('size',S,R,FP)
dw = convwf('dw',W,P,Z,FP)
info = convwf('code')
```

Description

Weight functions apply weights to an input to get weighted inputs.

`Z = convwf(W,P)` returns the convolution of a weight matrix `W` and an input `P`.

`dim = convwf('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size.

`dw = convwf('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

`info = convwf('code')` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Reduced derivative = 2, full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,1);
P = rand(8,1);
Z = convwf(W,P)
```

Network Use

To change a network so an input weight uses `convwf`, set `net.inputWeights{i,j}.weightFcn` to `'convwf'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'convwf'`.

In either case, call `sim` to simulate the network with `convwf`.

Introduced in R2006a

crossentropy

Neural network performance

Syntax

```
perf = crossentropy(net,targets,outputs,perfWeights)
perf = crossentropy( ___ ,Name,Value)
```

Description

`perf = crossentropy(net,targets,outputs,perfWeights)` calculates a network performance given targets and outputs, with optional performance weights and other parameters. The function returns a result that heavily penalizes outputs that are extremely inaccurate (`y` near `1-t`), with very little penalty for fairly correct classifications (`y` near `t`). Minimizing cross-entropy leads to good classifiers.

The cross-entropy for each pair of output-target elements is calculated as: `ce = -t .* log(y)`.

The aggregate cross-entropy performance is the mean of the individual values: `perf = sum(ce(:))/numel(ce)`.

Special case (N = 1): If an output consists of only one element, then the outputs and targets are interpreted as binary encoding. That is, there are two classes with targets of 0 and 1, whereas in 1-of-N encoding, there are two or more classes. The binary cross-entropy expression is: `ce = -t .* log(y) - (1-t) .* log(1-y)` .

`perf = crossentropy( ___ ,Name,Value)` supports customization according to the specified name-value pair arguments.

Examples

Calculate Network Performance

This example shows how to design a classification network with cross-entropy and 0.1 regularization, then calculate performance on the whole dataset.

```
[x,t] = iris_dataset;
net = patternnet(10);
net.performParam.regularization = 0.1;
net = train(net,x,t);
y = net(x);
perf = crossentropy(net,t,y,{1},'regularization',0.1)
```

```
perf = 0.0267
```

Set crossentropy as Performance Function

This example shows how to set up the network to use the `crossentropy` during training.

```
net = feedforwardnet(10);
net.performFcn = 'crossentropy';
net.performParam.regularization = 0.1;
net.performParam.normalization = 'none';
```

Input Arguments

net — neural network

network object

Neural network, specified as a network object.

Example: `net = feedforwardnet(10);`

targets — neural network target values

matrix or cell array of numeric values

Neural network target values, specified as a matrix or cell array of numeric values. Network target values define the desired outputs, and can be specified as an `N`-by-`Q` matrix of `Q` `N`-element vectors, or an `M`-by-`TS` cell array where each element is an `Ni`-by-`Q` matrix. In each of these cases, `N` or `Ni` indicates a vector length, `Q` the number of samples, `M` the number of signals for neural networks with multiple outputs, and `TS` is the number of time steps for time series data. `targets` must have the same dimensions as `outputs`.

The target matrix columns consist of all zeros and a single 1 in the position of the class being represented by that column vector. When N = 1, the software uses cross entropy for binary encoding, otherwise it uses cross entropy for 1-of-N encoding. `NaN` values are allowed to indicate unknown or don't-care output values. The performance of `NaN` target values is ignored.

Data Types: `double` | `cell`

outputs — neural network output values

matrix or cell array of numeric values

Neural network output values, specified as a matrix or cell array of numeric values. Network output values can be specified as an `N`-by-`Q` matrix of `Q` `N`-element vectors, or an `M`-by-`TS` cell array where each element is an `Ni`-by-`Q` matrix. In each of these cases, `N` or `Ni` indicates a vector length, `Q` the number of samples, `M` the number of signals for neural networks with multiple outputs and `TS` is the number of time steps for time series data. `outputs` must have the same dimensions as `targets`.

Outputs can include `NaN` to indicate unknown output values, presumably produced as a result of `NaN` input values (also representing unknown or don't-care values). The performance of `NaN` output values is ignored.

General case (N>=2): The columns of the output matrix represent estimates of class membership, and should sum to 1. You can use the `softmax` transfer function to produce such output values. Use `patternnet` to create networks that are already set up to use cross-entropy performance with a softmax output layer.

Data Types: `double` | `cell`

perfWeights — performance weights

`{1}` (default) | vector or cell array of numeric values

Performance weights, specified as a vector or cell array of numeric values. Performance weights are an optional argument defining the importance of each performance value, associated with each target

value, using values between 0 and 1. Performance values of 0 indicate targets to ignore, values of 1 indicate targets to be treated with normal importance. Values between 0 and 1 allow targets to be treated with relative importance.

Performance weights have many uses. They are helpful for classification problems, to indicate which classifications (or misclassifications) have relatively greater benefits (or costs). They can be useful in time series problems where obtaining a correct output on some time steps, such as the last time step, is more important than others. Performance weights can also be used to encourage a neural network to best fit samples whose targets are known most accurately, while giving less importance to targets which are known to be less accurate.

`perfWeights` can have the same dimensions as `targets` and `outputs`. Alternately, each dimension of the performance weights can either match the dimension of `targets` and `outputs`, or be 1. For instance, if `targets` is an `N`-by-`Q` matrix defining `Q` samples of `N`-element vectors, the performance weights can be `N`-by-`Q` indicating a different importance for each target value, or `N`-by-`1` defining a different importance for each row of the targets, or `1`-by-`Q` indicating a different importance for each sample, or be the scalar 1 (i.e. 1-by-1) indicating the same importance for all target values.

Similarly, if `outputs` and `targets` are cell arrays of matrices, the `perfWeights` can be a cell array of the same size, a row cell array (indicating the relative importance of each time step), a column cell array (indicating the relative importance of each neural network output), or a cell array of a single matrix or just the matrix (both cases indicating that all matrices have the same importance values).

For any problem, a `perfWeights` value of `{1}` (the default) or the scalar 1 indicates all performances have equal importance.

Data Types: `double` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'normalization','standard'` specifies the inputs and targets to be normalized to the range (-1,+1).

`regularization` — proportion of performance attributed to weight/bias values
0 (default) | numeric value in the range (0,1)

Proportion of performance attributed to weight/bias values, specified as a double between 0 (the default) and 1. A larger value penalizes the network for large weights, and the more likely the network function will avoid overfitting.

Example: `'regularization',0`

Data Types: `single` | `double`

`normalization` — Normalization mode for outputs, targets, and errors
`'none'` (default) | `'standard'` | `'percent'`

Normalization mode for outputs, targets, and errors, specified as `'none'`, `'standard'`, or `'percent'`. `'none'` performs no normalization. `'standard'` results in outputs and targets being normalized to (-1, +1), and therefore errors in the range (-2, +2).`'percent'` normalizes outputs and targets to (-0.5, 0.5) and errors to (-1, 1).

Example: `'normalization','standard'`

Data Types: `char`

Output Arguments

`perf` — network performance

double

Network performance, returned as a double in the range (0,1).

See Also

`mae` | `mse` | `patternnet` | `sae` | `softmax` | `sse`

Introduced in R2013b

defaultderiv

Default derivative function

Syntax

```
defaultderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
defaultderiv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function chooses the recommended derivative algorithm for the type of network whose derivatives are being calculated. For static networks, `defaultderiv` calls `staticderiv`; for dynamic networks it calls `bttderiv` to calculate the gradient and `fpderiv` to calculate the Jacobian.

`defaultderiv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an `R`-by-`Q` matrix (or `N`-by-`TS` cell array of `Ri`-by-`Q` matrices)
`T`	Targets, an `S`-by-`Q` matrix (or `M`-by-`TS` cell array of `Si`-by-`Q` matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where `R` and `S` are the number of input and output elements and `Q` is the number of samples (or `N` and `M` are the number of input and output signals, `Ri` and `Si` are the number of each input and outputs elements, and `TS` is the number of timesteps).

`defaultderiv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = defaultderiv('dperf_dwb',net,x,t)
```

See Also

`bttderiv` | `fpderiv` | `num2deriv` | `num5deriv` | `staticderiv`

Introduced in R2010b

dist

Euclidean distance weight function

Syntax

```
Z = dist(W,P)
dim = dist('size',S,R,FP)
dw = dist('dw',W,P,Z,FP)
D = dist(pos)
info = dist(code)
```

Description

`Z = dist(W,P)` takes an `S`-by-`R` weight matrix, `W`, and an `R`-by-`Q` matrix of `Q` input (column) vectors, `P`, and returns the `S`-by-`Q` matrix of vector distances, `Z`.

Weight functions apply weights to an input to get weighted inputs.

`dim = dist('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, `FP`, and returns the weight size [`S`-by-`R`].

`dw = dist('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

`dist` is also a layer distance function which can be used to find the distances between neurons in a layer.

`D = dist(pos)` takes `N`-by-`S` matrix of neuron positions, `pos` and returns the `S`-by-`S` matrix of distances, `D`.

`info = dist(code)` returns information about this function. For more information, see the **code** argument description.

Examples

Calculate the Weighted Input By Using the `dist` Function

This example shows how to calculate the corresponding weighted input `Z`, given a random weight matrix `W` and input vector `P`.

```
W = rand(4,3);
P = rand(3,1);
Z = dist(W,P)
```

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = dist(pos)
```

Input Arguments

W — Weight matrix

matrix

Weight matrix, specified as an S-by-R matrix.

P — Input matrix

matrix

Input matrix, specified as an R-by-Q matrix of Q input (column) vectors.

S — Layer dimension

scalar

Layer dimension, specified as a scalar.

R — Input dimension

scalar

Input dimension, specified as a scalar.

pos — Neuron positions

matrix

Matrix of neuron positions, specified as an N-by-S matrix.

code — Information option

`'name'` | `'output'` | `'active'` | `'fullderiv'` | `'fpnames'` | `'fpdefaults'`

Information you want to retrieve from the function, specified as one of the following:

- `'name'` returns the name of this function.
- `'deriv'` returns the name of the derivative function
- `'fullderiv'` returns 1 for full derivative and 0 for linear derivative.
- `'pfullderiv'` returns 2 for reduced derivative, 1 for full derivative, and 0 for linear derivative.
- `'fpnames'` returns the names of the function parameters.
- `'fpdefaults'` returns the default function parameters.

Output Arguments

Z — Vector distances

matrix

Vector distances, returned as an S-by-Q matrix.

dim — Weight size

row vector

Weight size, returned as a row vector.

dw — Derivative of w
cell array

Derivative of Z with respect to W, returned as a cell array.

D — Distances
matrix

Distances, returned as an S-by-S matrix.

More About

Network Use

You can create a standard network that uses dist by calling newpnn or newgrnn.

To change a network so an input weight uses dist, set net.inputWeights{i,j}.weightFcn to 'dist'. For a layer weight, set net.layerWeights{i,j}.weightFcn to 'dist'.

To change a network so that a layer's topology uses dist, set net.layers{i}.distanceFcn to 'dist'.

In either case, call sim to simulate the network with dist.

See newpnn or newgrnn for simulation examples.

Algorithms

The Euclidean distance d between two vectors X and Y is

```
d = sum((x-y).^2).^0.5
```

See Also

sim | dotprod | negdist | normprod | mandist | linkdist

Introduced before R2006a

distdelaynet

Distributed delay network

Syntax

```
distdelaynet(delays,hiddenSizes,trainFcn)
```

Description

Distributed delay networks are similar to feedforward networks, except that each input and layer weights has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the time delay neural network (`timedelaynet`), which only has delays on the input weight.

`distdelaynet(delays,hiddenSizes,trainFcn)` takes these arguments,

`delays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a distributed delay neural network.

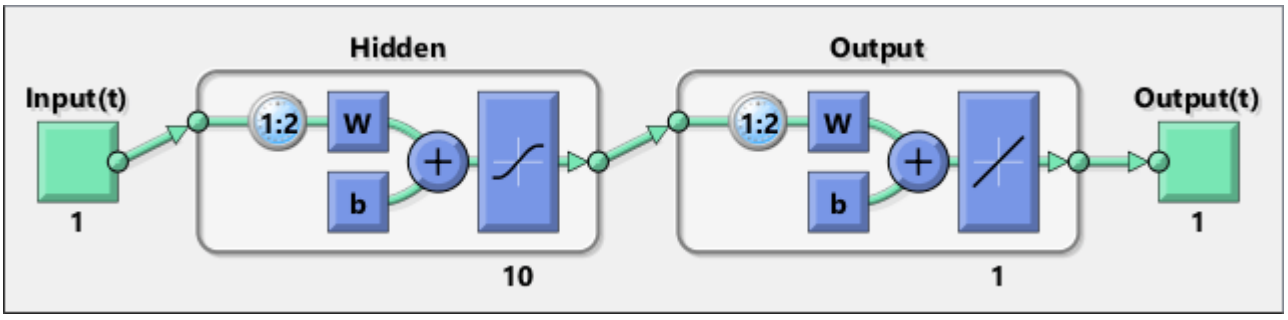
Examples

Distributed Delay Network

Here a distributed delay neural network is used to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;
net = distdelaynet({1:2,1:2},10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
```

```
perf =

    0.0323
```



See Also

`preparets` | `removedelay` | `timedelaynet` | `narnet` | `narxnet`

Introduced in R2010b

divideblock

Divide targets into three sets using blocks of indices

Syntax

```
[trainInd,valInd,testInd] = divideblock(Q,trainRatio,valRatio,testRatio)
```

Description

`[trainInd,valInd,testInd] = divideblock(Q,trainRatio,valRatio,testRatio)` separates targets into three sets: training, validation, and testing. It takes the following inputs:

`Q`	Number of targets to divide up.
`trainRatio`	Ratio of targets for training. Default = `0.7`.
`valRatio`	Ratio of targets for validation. Default = `0.15`.
`testRatio`	Ratio of targets for testing. Default = `0.15`.

and returns

`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
[trainInd,valInd,testInd] = divideblock(3000,0.6,0.2,0.2);
```

Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

```
net.divideFcn
net.divideParam
net.divideMode
```

See Also

`divideind` | `divideint` | `dividerand` | `dividetrain`

Introduced in R2008a

divideind

Divide targets into three sets using specified indices

Syntax

```
[trainInd,valInd,testInd] = divideind(Q,trainInd,valInd,testInd)
```

Description

`[trainInd,valInd,testInd] = divideind(Q,trainInd,valInd,testInd)` separates targets into three sets: training, validation, and testing, according to indices provided. It actually returns the same indices it receives as arguments; its purpose is to allow the indices to be used for training, validation, and testing for a network to be set manually.

The indices are returned after removing any indices greater than Q. Note that some indices in the range 1:Q may not be assigned to any of the three sets, and the same indices should not be used in more than one set.

Examples

Divide Samples into Three Sets Using Specified Indices

This example shows how to divide samples into three sets using specified indices for a network.

Divide 20 samples into training, validation and test indices, so that only 16 are actually used.

```
[trainInd,valInd,testInd] = divideind(20,1:8,9:12,14:16)
```

This code shows you how to ensure a network performs the same kind of data division when it is trained:

```
net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:8;
net.divideParam.valInd = 9:12;
net.divideParam.testInd= 14:16;
```

Input Arguments

`Q` — Number of targets
scalar

Number of targets to divide up, specified as a scalar.

`trainInd` — Training indices
vector

Training indices, specified as a 1-by-`Q` vector.

`valInd` — Validation indices
vector

Validation indices, specified as a 1-by-Q vector.

`testInd` — Testing indices
vector

Testing indices, specified as a 1-by-Q vector.

Output Arguments

`trainInd` — Training indices
vector

Training indices, returned as a vector.

`valInd` — Validation indices
vector

Validation indices, returned as a vector.

`testInd` — Testing indices
vector

Testing indices, returned as a vector.

More About

Network Use

These are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

```
net.divideFcn
net.divideParam
net.divideMode
```

See Also

`divideblock` | `divideint` | `dividerand` | `dividetrain`

Introduced in R2008a

divideint

Divide targets into three sets using interleaved indices

Syntax

```
[trainInd,valInd,testInd] = divideint(Q,trainRatio,valRatio,testRatio)
```

Description

`[trainInd,valInd,testInd] = divideint(Q,trainRatio,valRatio,testRatio)` separates targets into three sets: training, validation, and testing. It takes the following inputs,

`Q`	Number of targets to divide up.
`trainRatio`	Ratio of vectors for training. Default = `0.7`.
`valRatio`	Ratio of vectors for validation. Default = `0.15`.
`testRatio`	Ratio of vectors for testing. Default = `0.15`.

and returns

`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
[trainInd,valInd,testInd] = divideint(3000,0.6,0.2,0.2);
```

Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

```
net.divideFcn
net.divideParam
net.divideMode
```

See Also

`divideblock` | `divideind` | `dividerand` | `dividetrain`

Introduced in R2008a

dividerand

Divide targets into three sets using random indices

Syntax

```
[trainInd,valInd,testInd] = dividerand(Q,trainRatio,valRatio,testRatio)
```

Description

`[trainInd,valInd,testInd] = dividerand(Q,trainRatio,valRatio,testRatio)` takes the number of targets to divide up, the ratio of vectors for training, the ratio of vectors for validation, and the ratio of vectors for testing, and returns the training indices, the validation indices, and the test indices.

Examples

Obtain Training, Validation, and Test Indices Using `'dividerand'` Function

This example shows how to obtain the training, validation, and test indices using the `dividerand` function.

Divide `3000` samples into `60%` for training, `20%` for validation, and `20%` for testing.

```
[trainInd,valInd,testInd] = dividerand(3000,0.6,0.2,0.2)
```

Input Arguments

`Q` — Targets number
scalar

Number of targets to divide up, specified as a scalar.

`trainRatio` — Training ratio
`0.7` (default) | scalar

Ratio of vectors for training, specified as a scalar.

`valRatio` — Validation ratio
`0.15` (default) | scalar

Ratio of vectors for validation, specified as a scalar.

`testRatio` — Testing ratio
`0.15` (default) | scalar

Ratio of vectors for testing, specified as a scalar.

Output Arguments

`trainInd` — Training indices

vector

Training indices, returned as a row vector.

`valInd` — Validation indices

vector

Validation indices, returned as a row vector.

`testInd` — Testing indices

vector

Testing indices, returned as a row vector.

More About

Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

```
net.divideFcn
net.divideParam
net.divideMode
```

See Also

`divideblock` | `divideind` | `divideint` | `dividetrain`

Introduced in R2008a

dividetrain

Assign all targets to training set

Syntax

```
[trainInd,valInd,testInd] = dividetrain(Q)
```

Description

`[trainInd,valInd,testInd] = dividetrain(Q)` assigns all targets to the training set and no targets to the validation or test sets. It takes the following inputs:

`Q`	Number of targets to divide up.

and returns

`trainInd`	Training indices equal to `1:Q`
`valInd`	Empty validation indices, `[]`
`testInd`	Empty test indices, `[]`

Examples

```
[trainInd,valInd,testInd] = dividetrain(250);
```

Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

```
net.divideFcn
net.divideParam
net.divideMode
```

See Also

`divideblock` | `divideind` | `divideint` | `dividerand`

Introduced in R2010b

dotprod

Dot product weight function

Syntax

```
Z = dotprod(W,P,FP)
dim = dotprod('size',S,R,FP)
dw = dotprod('dw',W,P,Z,FP)
info = dotprod('code')
```

Description

Weight functions apply weights to an input to get weighted inputs.

`Z = dotprod(W,P,FP)` takes these inputs,

`W`	S-by-R weight matrix
`P`	R-by-Q matrix of Q input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

and returns the S-by-Q dot product of `W` and `P`.

`dim = dotprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [S-by-R].

`dw = dotprod('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

`info = dotprod('code')` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = dotprod(W,P)
```

Network Use

You can create a standard network that uses `dotprod` by calling `feedforwardnet`.

To change a network so an input weight uses `dotprod`, set `net.inputWeights{i,j}.weightFcn` to `'dotprod'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'dotprod'`.

In either case, call `sim` to simulate the network with `dotprod`.

See Also

`sim` | `dist` | `feedforwardnet` | `negdist` | `normprod`

Introduced before R2006a

elliotsig

Elliot symmetric sigmoid transfer function

Syntax

```
A = elliotsig(N)
```

Description

Transfer functions convert a neural network layer's net input into its net output.

`A = elliotsig(N)` takes an `S`-by-`Q` matrix of `S` `N`-element net input column vectors and returns an `S`-by-`Q` matrix `A` of output vectors, where each element of `N` is squashed from the interval `[-inf inf]` to the interval `[-1 1]` with an "S-shaped" function.

The advantage of this transfer function over other sigmoids is that it is fast to calculate on simple computing hardware as it does not require any exponential or trigonometric functions. Its disadvantage is that it only flattens out for large inputs, so its effect is not as local as other sigmoid functions. This might result in more training iterations, or require more neurons to achieve the same accuracy.

Examples

Calculate a layer output from a single net input vector:

```
n = [0; 1; -0.5; 0.5];
a = elliotsig(n);
```

Plot the transfer function:

```
n = -5:0.01:5;
plot(n, elliotsig(n))
set(gca,'dataaspectratio',[1 1 1],'xgrid','on','ygrid','on')
```

For a network you have already defined, change the transfer function for layer `i`:

```
 net.layers{i}.transferFcn = 'elliotsig';
```

See Also

`elliot2sig` | `logsig` | `tansig`

Introduced in R2012b

elliot2sig

Elliot 2 symmetric sigmoid transfer function

Syntax

```
A = elliot2sig(N)
```

Description

Transfer functions convert a neural network layer's net input into its net output. This function is a variation on the original Elliot sigmoid function. It has a steeper slope, closer to `tansig`, but is not as smooth at the center.

`A = elliot2sig(N)` takes an `S`-by-`Q` matrix of `S` `N`-element net input column vectors and returns an `S`-by-`Q` matrix `A` of output vectors, where each element of `N` is squashed from the interval `[-inf inf]` to the interval `[-1 1]` with an "S-shaped" function.

The advantage of this transfer function over other sigmoids is that it is fast to calculate on simple computing hardware as it does not require any exponential or trigonometric functions. Its disadvantage is that it departs from the classic sigmoid shape around zero.

Examples

Calculate a layer output from a single net input vector:

```
n = [0; 1; -0.5; 0.5];
a = elliot2sig(n);
```

Plot the transfer function:

```
n = -5:0.01:5;
plot(n, elliot2sig(n))
set(gca,'dataaspectratio',[1 1 1],'xgrid','on','ygrid','on')
```

For a network you have already defined, change the transfer function for layer `i`:

```
 net.layers{i}.transferFcn = 'elliot2sig';
```

See Also

`elliotsig` | `logsig` | `tansig`

Introduced in R2012b

elmannet

Elman neural network

Syntax

```
elmannet(layerdelays,hiddenSizes,trainFcn)
```

Description

Elman networks are feedforward networks (`feedforwardnet`) with the addition of layer recurrent connections with tap delays.

With the availability of full dynamic derivative calculations (`fpderiv` and `bttderiv`), the Elman network is no longer recommended except for historical and research purposes. For more accurate learning try time delay (`timedelaynet`), layer recurrent (`layrecnet`), NARX (`narxnet`), and NAR (`narnet`) neural networks.

Elman networks with one or more hidden layers can learn any dynamic input-output relationship arbitrarily well, given enough neurons in the hidden layers. However, Elman networks use simplified derivative calculations (using `staticderiv`, which ignores delayed connections) at the expense of less reliable learning.

`elmannet(layerdelays,hiddenSizes,trainFcn)` takes these arguments,

`layerdelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns an Elman neural network.

Examples

Here an Elman neural network is used to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;
net = elmannet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)
```

See Also

`preparets` | `removedelay` | `timedelaynet` | `layrecnet` | `narnet` | `narxnet`

Introduced in R2010b

errsurf

Error surface of single-input neuron

Syntax

```
errsurf(P,T,WV,BV,F)
```

Description

`errsurf(P,T,WV,BV,F)` takes these arguments,

`P`	1-by-`Q` matrix of input vectors
`T`	1-by-`Q` matrix of target vectors
`WV`	Row vector of values of `W`
`BV`	Row vector of values of `B`
`F`	Transfer function (string)

and returns a matrix of error values over `WV` and `BV`.

Examples

```
p = [-6.0 -6.1 -4.1 -4.0 +4.0 +4.1 +6.0 +6.1];
t = [+0.0 +0.0 +.97 +.99 +.01 +.03 +1.0 +1.0];
wv = -1:.1:1; bv = -2.5:.25:2.5;
es = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,es,[60 30])
```

See Also

`plotes`

Introduced before R2006a

extendts

Extend time series data to given number of timesteps

Syntax

```
extendts(x,ts,v)
```

Description

`extendts(x,ts,v)` takes these values,

`x`	Neural network time series data
`ts`	Number of timesteps
`v`	Value

and returns the time series data either extended or truncated to match the specified number of timesteps. If the value `v` is specified, then extended series are filled in with that value, otherwise they are extended with random values.

Examples

Here, a 20-timestep series is created and then extended to 25 timesteps with the value zero.

```
x = nndata(5,4,20);
y = extendts(x,25,0)
```

See Also

`nndata` | `catsamples` | `preparets`

Introduced in R2010b

feedforwardnet

Generate feedforward neural network

Syntax

```
net = feedforwardnet(hiddenSizes,trainFcn)
```

Description

`net = feedforwardnet(hiddenSizes,trainFcn)` returns a feedforward neural network with a hidden layer size of `hiddenSizes` and training function, specified by `trainFcn`.

Feedforward networks consist of a series of layers. The first layer has a connection from the network input. Each subsequent layer has a connection from the previous layer. The final layer produces the network's output.

You can use feedforward networks for any kind of input to output mapping. A feedforward network with one hidden layer and enough neurons in the hidden layers can fit any finite input-output mapping problem.

Specialized versions of the feedforward network include fitting and pattern recognition networks. For more information, see the `fitnet` and `patternnet` functions.

A variation on the feedforward network is the cascade forward network, which has additional connections from the input to every layer, and from each layer to all following layers. For more information on cascade forward networks, see the `cascadeforwardnet` function.

Examples

Construct and Train a Feedforward Neural Network

This example shows how to use a feedforward neural network to solve a simple problem.

Load the training data.

```
[x,t] = simplefit_dataset;
```

The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a feedforward network with one hidden layer of size 10.

```
net = feedforwardnet(10);
```

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```

Estimate the targets using the trained network.

```
y = net(x);
```

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,y,t)
```

```
perf = 1.4639e-04
```

Input Arguments

`hiddenSizes` — Size of the hidden layers

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: `[10,8,5]`

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

`trainFcn` — Training function name

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

Output Arguments

`net` — Feedforward network

`network` object

Feedforward neural network, returned as a `network` object.

See Also

`fitnet` | `network` | `patternnet` | `cascadeforwardnet`

Topics

“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b

fixunknowns

Process data by marking rows with unknown values

Syntax

```
[y,ps] = fixunknowns(X)
[y,ps] = fixunknowns(X,FP)
Y = fixunknowns('apply',X,PS)
X = fixunknowns('reverse',Y,PS)
name = fixunknowns('name')
fp = fixunknowns('pdefaults')
pd = fixunknowns('pdesc')
fixunknowns('pcheck',fp)
```

Description

`fixunknowns` processes matrices by replacing each row containing unknown values (represented by `NaN`) with two rows of information.

The first row contains the original row, with `NaN` values replaced by the row's mean. The second row contains 1 and 0 values, indicating which values in the first row were known or unknown, respectively.

`[y,ps] = fixunknowns(X)` takes these inputs,

`X`	`N`-by-`Q` matrix

and returns

`Y`	`M`-by-`Q` matrix with `M - N` rows added
`PS`	Process settings that allow consistent processing of values

`[y,ps] = fixunknowns(X,FP)` takes an empty struct `FP` of parameters.

`Y = fixunknowns('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = fixunknowns('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`name = fixunknowns('name')` returns the name of this process method.

`fp = fixunknowns('pdefaults')` returns the default process parameter structure.

`pd = fixunknowns('pdesc')` returns the process parameter descriptions.

`fixunknowns('pcheck',fp)` throws an error if any parameter is illegal.

Examples

Here is how to format a matrix with a mixture of known and unknown values in its second row:

```
x1 = [1 2 3 4; 4 NaN 6 5; NaN 2 3 NaN]
[y1,ps] = fixunknowns(x1)
```

Next, apply the same processing settings to new values:

```
x2 = [4 5 3 2; NaN 9 NaN 2; 4 9 5 2]
y2 = fixunknowns('apply',x2,ps)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = fixunknowns('reverse',y1,ps)
```

More About

Recode Data with NaNs Using `fixunknowns`

If you have input data with unknown values, you can represent them with `NaN` values. For example, here are five 2-element vectors with unknown values in the first element of two of the vectors:

```
p1 = [1 NaN 3 2 NaN; 3 1 -1 2 4];
```

The network will not be able to process the `NaN` values properly. Use the function `fixunknowns` to transform each row with `NaN` values (in this case only the first row) into two rows that encode that same information numerically.

```
[p2,ps] = fixunknowns(p1);
```

Here is how the first row of values was recoded as two rows.

```
p2 =
   1  2  3  2  2
   1  0  1  1  0
   3  1 -1  2  4
```

The first new row is the original first row, but with the mean value for that row (in this case `2`) replacing all `NaN` values. The elements of the second new row are now either `1`, indicating the original element was a known value, or `0` indicating that it was unknown. The original second row is now the new third row. In this way both known and unknown values are encoded numerically in a way that lets the network be trained and simulated.

Whenever supplying new data to the network, you should transform the inputs in the same way, using the settings `ps` returned by `fixunknowns` when it was used to transform the training input data.

```
p2new = fixunknowns('apply',p1new,ps);
```

The function `fixunkowns` is only recommended for input processing. Unknown targets represented by `NaN` values can be handled directly by the toolbox learning algorithms. For instance, performance functions used by backpropagation algorithms recognize `NaN` values as unknown or unimportant values.

See Also

`mapminmax` | `mapstd` | `processpca`

Introduced in R2006a

formwb

Form bias and weights into single vector

Syntax

```
formwb(net,b,IW,LW)
```

Description

`formwb(net,b,IW,LW)` takes a neural network and bias `b`, input weight `IW`, and layer weight `LW` values, and combines the values into a single vector.

Examples

Here a network is created, configured, and its weights and biases formed into a vector.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = configure(net,x,t);
wb = formwb(net,net.b,net.IW,net.LW)
```

See Also

`getwb` | `setwb` | `separatewb`

Introduced in R2010b

fpderiv

Forward propagation derivative function

Syntax

```
fpderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
fpderiv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the chain rule from inputs to outputs, and in the case of dynamic networks, forward through time.

`fpderiv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an `R`-by-`Q` matrix (or `N`-by-`TS` cell array of `Ri`-by-`Q` matrices)
`T`	Targets, an `S`-by-`Q` matrix (or `M`-by-`TS` cell array of `Si`-by-`Q` matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where `R` and `S` are the number of input and output elements and `Q` is the number of samples (or `N` and `M` are the number of input and output signals, `Ri` and `Si` are the number of each input and outputs elements, and `TS` is the number of timesteps).

`fpderiv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = fpderiv('dperf_dwb',net,x,t)
jwb = fpderiv('de_dwb',net,x,t)
```

See Also

`bttderiv` | `defaultderiv` | `num2deriv` | `num5deriv` | `staticderiv`

Introduced in R2010b

fromnndata

Convert data from standard neural network cell array form

Syntax

```
fromnndata(x,toMatrix,columnSample,cellTime)
```

Description

`fromnndata(x,toMatrix,columnSample,cellTime)` takes these arguments,

`net`	Neural network
`toMatrix`	True if result is to be in matrix form
`columnSample`	True if samples are to be represented as columns, false if rows
`cellTime`	True if time series are to be represented as a cell array, false if represented with a matrix

and returns the original data reformatted accordingly.

Examples

Here time-series data is converted from a matrix representation to standard cell array representation, and back. The original data consists of a 5-by-6 matrix representing one time-series sample consisting of a 5-element vector over 6 timesteps arranged in a matrix with the samples as columns.

```
x = rands(5,6)
columnSamples = true; % samples are by columns.
cellTime = false;     % time-steps in matrix, not cell array.
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```

Here data is defined in standard neural network data cell form. Converting this data does not change it. The data consists of three time series samples of 2-element signals over 3 timesteps.

```
x = {rands(2,3);rands(2,3);rands(2,3)}
columnSamples = true;
cellTime = true;
[y,wasMatrix] = tonndata(x)
x2 = fromnndata(y,wasMatrix,columnSamples)
```

See Also

`tonndata`

Introduced in R2010b

gadd

Generalized addition

Syntax

```
gadd(a,b)
```

Description

`gadd(a,b)` takes two matrices or cell arrays, and adds them in an element-wise manner.

Examples

Add Matrix and Cell Array Values

This example shows how to add matrix and cell array values.

```
gadd([1 2 3; 4 5 6],[10;20])
```

```
ans = 2×3

    11    12    13
    24    25    26
```

```
gadd({1 2; 3 4},{1 3; 5 2})
```

```
ans=2×2 cell array
    {[2]}    {[5]}
    {[8]}    {[6]}
```

```
gadd({1 2 3 4},{10;20;30})
```

```
ans=3×4 cell array
    {[11]}    {[12]}    {[13]}    {[14]}
    {[21]}    {[22]}    {[23]}    {[24]}
    {[31]}    {[32]}    {[33]}    {[34]}
```

See Also

`gsubtract` | `gmultiply` | `gdivide` | `gnegate` | `gsqrt`

Introduced in R2010b

gdivide

Generalized division

Syntax

```
gdivide(a,b)
```

Description

`gdivide(a,b)` takes two matrices or cell arrays, and divides them in an element-wise manner.

Examples

Divide Matrix and Cell Array Values

This example shows how to divide matrix and cell array values.

```
gdivide([1 2 3; 4 5 6],[10;20])
```

```
ans = 2×3

    0.1000    0.2000    0.3000
    0.2000    0.2500    0.3000
```

```
gdivide({1 2; 3 4},{1 3; 5 2})
```

```
ans=2×2 cell array
    {[     1]}    {[0.6667]}
    {[0.6000]}    {[     2]}
```

```
gdivide({1 2 3 4},{10;20;30})
```

```
ans=3×4 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}    {[0.4000]}
    {[0.0500]}    {[0.1000]}    {[0.1500]}    {[0.2000]}
    {[0.0333]}    {[0.0667]}    {[0.1000]}    {[0.1333]}
```

See Also

`gadd` | `gsubtract` | `gmultiply` | `gnegate` | `gsqrt`

Introduced in R2010b

gensim

Generate Simulink block for shallow neural network simulation

Syntax

```
gensim(net,st)
```

Description

`gensim(net,st)` creates a Simulink system containing a block that simulates neural network `net` with a sampling time of `st`.

If `net` has no input or layer delays (`net.numInputDelays` and `net.numLayerDelays` are both 0), you can use –1 for `st` to get a network that samples continuously.

`gensim` does not support deep learning networks such as convolutional or LSTM networks. For more information on code generation for deep learning, see “Deep Learning Code Generation”.

For more information on `gensim`, at the MATLAB command prompt, enter `help network/gensim`.

Examples

Generate a Simulink Block for a Feedforward Network

This example shows how to generate a Simulink block for a feedforward network.

Create a feed-forward network using the data from the simple fit data set and generate the Simulink block.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t)
gensim(net)
```

Generate a Simulink Block for a NARX Network

This example shows how to generate a Simulink block for a NARX network.

Create a NARX network.

```
[x,t] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
net = train(net,xs,ts,xi,ai);
y = net(xs,xi,ai);
```

Convert the network to closed loop.

```
net = closeloop(net);
view(net)
```

Prepare the data and simulate the network's closed loop response.

```
[xs,xi,ai,ts] = preparets(net,x,{},t);
y = net(xs,xi,ai);
```

Convert the network to a Simulink system with workspace input and output ports.

```
[sysName,netName] = gensim(net,'InputMode','Workspace',...
    'OutputMode','WorkSpace','SolverMode','Discrete');
```

Initialize the delay states. Note that this is an important step to obtain the same output as in MATLAB.

```
setsiminit(sysName,netName,net,xi,ai,1);
```

Define the model input `X1` in the workspace, simulate the system programmatically.

```
x1 = nndata2sim(xs,1,1);
out = sim(sysName,'ReturnWorkspaceOutputs','on','StopTime',num2str(x1.time(end)));
ysim = sim2nndata(out.y1);
```

Input Arguments

`net` — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

`st` — Sample time

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see "Specify Sample Time" (Simulink).

See Also

`preparets`

Introduced before R2006a

genFunction

Generate MATLAB function for simulating shallow neural network

Syntax

```
genFunction(net,pathname)
genFunction( ___ ,'MatrixOnly','yes')
genFunction( ___ ,'ShowLinks','no')
```

Description

This function generates a MATLAB function for simulating a shallow neural network. `genFunction` does not support deep learning networks such as convolutional or LSTM networks. For more information on code generation for deep learning, see “Deep Learning Code Generation”.

`genFunction(net,pathname)` generates a complete stand-alone MATLAB function for simulating a neural network including all settings, weight and bias values, module functions, and calculations in one file. The result is a standalone MATLAB function file. You can also use this function with MATLAB Compiler and MATLAB Coder tools.

`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'`.

Examples

Create Functions from Static Neural Network

This example shows how to create a MATLAB function and a MEX-function from a static neural network.

First, train a static network and calculate its outputs for the training data.

```
[x,t] = bodyfat_dataset;
bodyfatNet = feedforwardnet(10);
bodyfatNet = train(bodyfatNet,x,t);
y = bodyfatNet(x);
```

Next, generate and test a MATLAB function. Then the new function is compiled to a shared/dynamically linked library with `mcc`.

```
genFunction(bodyfatNet,'bodyfatFcn');
y2 = bodyfatFcn(x);
accuracy2 = max(abs(y-y2))
mcc -W lib:libBodyfat -T link:lib bodyfatFcn
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB Coder tool `codegen` to generate a MEX-function, 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 = bodyfatodeGen(x);
accuracy4 = max(abs(y-y4))
```

Create Functions from Dynamic Neural Network

This example shows how to create a MATLAB function and a MEX-function from a dynamic neural network.

First, train a dynamic network and calculate its outputs for the training data.

```
[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, generate and test a MATLAB function. Use the function 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
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB Coder tool `codegen` to generate a MEX-function, 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))
```

Input Arguments

net — neural network

network object

Neural network, specified as a network object.

Example: `net = feedforwardnet(10);`

`pathname` — location and name of generated function file
(default) | character string

Location and name of generated function file, specified as a character string. If you do not specify a file name extension of `.m`, it is automatically appended. If you do not specify a path to the file, the default location is the current working folder.

Example: `'myFcn.m'`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use `genFunction` in the Deep Learning Toolbox to generate a standalone MATLAB function for a trained neural network. You can generate C/C++ code from this standalone MATLAB function. To generate Simulink blocks, use the `genSim` function. See “Deploy Shallow Neural Network Functions”.

See Also

`gensim`

Topics

“Deploy Shallow Neural Network Functions”

Introduced in R2013b

getelements

Get neural network data elements

Syntax

```
getelements(x,ind)
```

Description

`getelements(x,ind)` returns the elements of neural network data `x` indicated by the indices `ind`. The neural network data may be in matrix or cell array form.

If `x` is a matrix, the result is the `ind` rows of `x`.

If `x` is a cell array, the result is a cell array with as many columns as `x`, whose elements `(1,i)` are matrices containing the `ind` rows of `[x{:,i}]`.

Examples

This code gets elements 1 and 3 from matrix data:

```
x = [1 2 3; 4 7 4]
y = getelements(x,[1 3])
```

This code gets elements 1 and 3 from cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
y = getelements(x,[1 3])
```

See Also

`nndata` | `numelements` | `setelements` | `catelements` | `getsamples` | `gettimesteps` | `getsignals`

Introduced in R2010b

getsamples

Get neural network data samples

Syntax

```
getsamples(x,ind)
```

Description

`getsamples(x,ind)` returns the samples of neural network data `x` indicated by the indices `ind`. The neural network data may be in matrix or cell array form.

If `x` is a matrix, the result is the `ind` columns of `x`.

If `x` is a cell array, the result is a cell array the same size as `x`, whose elements are the `ind` columns of the matrices in `x`.

Examples

This code gets samples 1 and 3 from matrix data:

```
x = [1 2 3; 4 7 4]
y = getsamples(x,[1 3])
```

This code gets elements 1 and 3 from cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
y = getsamples(x,[1 3])
```

See Also

`nndata` | `numsamples` | `setsamples` | `catsamples` | `getelements` | `gettimesteps` | `getsignals`

Introduced in R2010b

getsignals

Get neural network data signals

Syntax

```
getsignals(x,ind)
```

Description

`getsignals(x,ind)` returns the signals of neural network data `x` indicated by the indices `ind`. The neural network data may be in matrix or cell array form.

If `x` is a matrix, `ind` may only be 1, which will return `x`, or `[]` which will return an empty matrix.

If `x` is a cell array, the result is the `ind` rows of `x`.

Examples

This code gets signal 2 from cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
y = getsignals(x,2)
```

See Also

`nndata` | `numsignals` | `setsignals` | `catsignals` | `getelements` | `getsamples` | `gettimesteps`

Introduced in R2010b

getsiminit

Get Simulink neural network block initial input and layer delays states

Syntax

```
[xi,ai] = getsiminit(sysName,netName,net)
```

Description

`[xi,ai] = getsiminit(sysName,netName,net)` takes these arguments,

`sysName`	The name of the Simulink system containing the neural network block
`netName`	The name of the Simulink neural network block
`net`	The original neural network

and returns,

`xi`	Initial input delay states
`ai`	Initial layer delay states

Examples

Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[x,t] = simplenarx_dataset;
     net = narxnet(1:2,1:2,20);
     view(net)
     [xs,xi,ai,ts] = preparets(net,x,{},t);
     net = train(net,xs,ts,xi,ai);
     y = net(xs,xi,ai);
```

Now the network is converted to closed-loop, and the data is reformatted to simulate the network's closed-loop response.

```
net = closeloop(net);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
y = net(xs,xi,ai);
```

Here the network is converted to a Simulink system with workspace input and output ports. Its delay states are initialized, inputs `X1` defined in the workspace, and it is ready to be simulated in Simulink.

```
[sysName,netName] = gensim(net,'InputMode','Workspace',...
    'OutputMode','WorkSpace','SolverMode','Discrete');
setsiminit(sysName,netName,net,xi,ai,1);
x1 = nndata2sim(x,1,1);
```

Finally the initial input and layer delays are obtained from the Simulink model. (They will be identical to the values set with `setsiminit`.)

```
[xi,ai] = getsiminit(sysName,netName,net);
```

See Also

gensim | setsiminit | nndata2sim | sim2nndata

Introduced in R2010b

gettimesteps

Get neural network data timesteps

Syntax

```
gettimesteps(x,ind)
```

Description

`gettimesteps(x,ind)` returns the timesteps of neural network data `x` indicated by the indices `ind`. The neural network data may be in matrix or cell array form.

If `x` is a matrix, `ind` can only be 1, which will return `x`; or `[]`, which will return an empty matrix.

If `x` is a cell array the result is the `ind` columns of `x`.

Examples

This code gets timestep 2 from cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
y = gettimesteps(x,2)
```

See Also

`nndata` | `numtimesteps` | `settimesteps` | `cattimesteps` | `getelements` | `getsamples` | `getsignals`

Introduced in R2010b

getwb

Get network weight and bias values as single vector

Syntax

```
getwb(net)
```

Description

`getwb(net)` returns a neural network’s weight and bias values as a single vector.

Examples

Here a feedforward network is trained to fit some data, then its bias and weight values are formed into a vector.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
wb = getwb(net)
```

See Also

`setwb` | `formwb` | `separatewb`

Introduced in R2010b

gmultiply

Generalized multiplication

Syntax

```
gmultiply(a,b)
```

Description

`gmultiply(a,b)` takes two matrices or cell arrays, and multiplies them in an element-wise manner.

Examples

Multiply Matrix and Cell Array Values

This example shows how to multiply matrix and cell array values.

```
gmultiply([1 2 3; 4 5 6],[10;20])
```

```
ans = 2×3

    10    20    30
    80   100   120
```

```
gmultiply({1 2; 3 4},{1 3; 5 2})
```

```
ans=2×2 cell array
    {[ 1]}    {[6]}
    {[15]}    {[8]}
```

```
gmultiply({1 2 3 4},{10;20;30})
```

```
ans=3×4 cell array
    {[10]}    {[20]}    {[30]}    {[ 40]}
    {[20]}    {[40]}    {[60]}    {[ 80]}
    {[30]}    {[60]}    {[90]}    {[120]}
```

See Also

`gadd` | `gsubtract` | `gdivide` | `gnegate` | `gsqrt`

Introduced in R2010b

gnegate

Generalized negation

Syntax

```
gnegate(x)
```

Description

`gnegate(x)` takes a matrix or cell array of matrices, and negates their element values.

Examples

Negate a Cell Array

This example shows how to negate a cell array:

```
x = {[1 2; 3 4],[1 -3; -5 2]};
y = gnegate(x);
y{1}, y{2}
```

```
ans = 2×2

    -1    -2
    -3    -4


ans = 2×2

    -1     3
     5    -2
```

See Also

`gadd` | `gsubtract` | `gdivide` | `gmultiply` | `gsqrt`

Introduced in R2010b

gpu2nndata

Reformat neural data back from GPU

Syntax

```
X = gpu2nndata(Y,Q)
X = gpu2nndata(Y)
X = gpu2nndata(Y,Q,N,TS)
```

Description

Training and simulation of neural networks require that matrices be transposed. But they do not require (although they are more efficient with) padding of column length so that each column is memory aligned. This function copies data back from the current GPU and reverses this transform. It can be used on data formatted with `nndata2gpu` or on the results of network simulation.

`X = gpu2nndata(Y,Q)` copies the `QQ`-by-`N` gpuArray `Y` into RAM, takes the first `Q` rows and transposes the result to get an `N`-by-`Q` matrix representing `Q` `N`-element vectors.

`X = gpu2nndata(Y)` calculates `Q` as the index of the last row in `Y` that is not all `NaN` values (those rows were added to pad `Y` for efficient GPU computation by `nndata2gpu`). `Y` is then transformed as before.

`X = gpu2nndata(Y,Q,N,TS)` takes a `QQ`-by-(`N`*`TS`) gpuArray where `N` is a vector of signal sizes, `Q` is the number of samples (less than or equal to the number of rows after alignment padding `QQ`), and `TS` is the number of time steps.

The gpuArray `Y` is copied back into RAM, the first `Q` rows are taken, and then it is partitioned and transposed into an `M`-by-`TS` cell array, where `M` is the number of elements in `N`. Each `Y{i,ts}` is an `N(i)`-by-`Q` matrix.

Examples

Copy a matrix to the GPU and back:

```
x = rand(5,6)
[y,q] = nndata2gpu(x)
x2 = gpu2nndata(y,q)
```

Copy from the GPU a neural network cell array data representing four time series, each consisting of five time steps of 2-element and 3-element signals.

```
x = nndata([2;3],4,5)
[y,q,n,ts] = nndata2gpu(x)
x2 = gpu2nndata(y,q,n,ts)
```

See Also

`nndata2gpu`

Introduced in R2012b

gridtop

Grid layer topology function

Syntax

```
gridtop(dimensions)
```

Description

`pos = gridtop` calculates neuron positions for layers whose neurons are arranged in an `N`-dimensional grid.

`gridtop(dimensions)` takes one argument:

`dimensions`	Row vector of dimension sizes

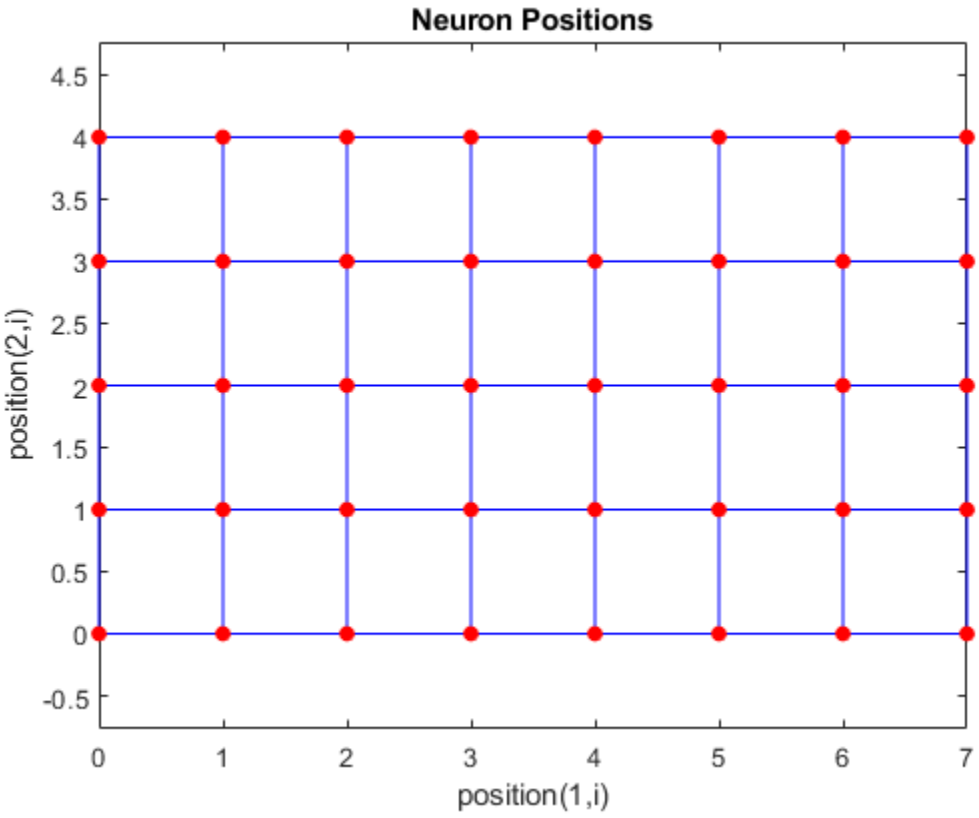
and returns an `N`-by-`S` matrix of `N` coordinate vectors where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Grid Pattern

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 grid pattern.

```
pos = gridtop([8 5]);
plotsom(pos)
```



See Also

`hextop` | `randtop` | `tritop`

Introduced before R2006a

gsqrt

Generalized square root

Syntax

```
gsqrt(x)
```

Description

`gsqrt(x)` takes a matrix or cell array of matrices, and generates the element-wise square root of the matrices.

Examples

Compute Element-Wise Square Root

This example shows how to get the element-wise square root of a cell array:

```
gsqrt({1 2; 3 4})
```

```
ans=2×2 cell array
    {[     1]}    {[1.4142]}
    {[1.7321]}    {[     2]}
```

See Also

`gadd` | `gsubtract` | `gdivide` | `gmultiply` | `gnegate`

Introduced in R2010b

gsubtract

Generalized subtraction

Syntax

```
gsubtract(a,b)
```

Description

`gsubtract(a,b)` takes two matrices or cell arrays, and subtracts them in an element-wise manner.

Examples

Subtract Matrix and Cell Array Values

This example shows how to subtract matrix and cell array values.

```
gsubtract([1 2 3; 4 5 6],[10;20])
```

```
ans = 2×3

    -9    -8    -7
   -16   -15   -14
```

```
gsubtract({1 2; 3 4},{1 3; 5 2})
```

```
ans=2×2 cell array
    {[ 0]}    {[-1]}
    {[-2]}    {[ 2]}
```

```
gsubtract({1 2 3 4},{10;20;30})
```

```
ans=3×4 cell array
    {[ -9]}    {[ -8]}    {[ -7]}    {[ -6]}
    {[-19]}    {[-18]}    {[-17]}    {[-16]}
    {[-29]}    {[-28]}    {[-27]}    {[-26]}
```

See Also

`gadd` | `gmultiply` | `gdivide` | `gnegate` | `gsqrt`

Introduced in R2010b

hardlim

Hard-limit transfer function

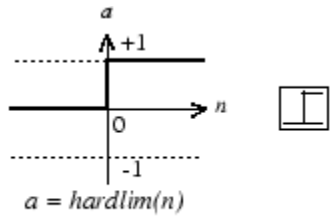
Syntax

```
A = hardlim(N)
info = hardlim('code')
```

Description

`A = hardlim(N)` takes an `S`-by-`Q` matrix of net input (column) vectors, `N`, and returns `A`, the `S`-by-`Q` Boolean matrix with elements equal to `1` where `N` is greater than or equal to `0`.

`hardlim` is a neural transfer function. Transfer functions calculate a layer's output from its net input.



Hard-Limit Transfer Function

`info = hardlim('code')` returns useful information for each `code` character vector:

- `hardlim('name')` returns the name of this function.
- `hardlim('output')` returns the `[min max]` output range.
- `hardlim('active')` returns the `[min max]` active input range.
- `hardlim('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S`-by-`S`-by-`Q` or `S`-by-`Q`.
- `hardlim('fpnames')` returns the names of the function parameters.
- `hardlim('fpdefaults')` returns the default function parameters.

Examples

Create a Plot of the `hardlim` Transfer Function

This example shows how to create a plot of the `hardlim` transfer function.

Create the input matrix, n. Then call the `hardlim` function and plot the results.

```
n = -5:0.1:5;
a = hardlim(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'hardlim';
```

Input Arguments

N — Input matrix

matrix

Net input column vectors, specified as an S-by-Q matrix.

Output Arguments

A — Output matrix

matrix

Output matrix, returned as an S-by-Q Boolean matrix with elements equal to 1 where N is greater than or equal to 0.

Algorithms

hardlim(n) = 1 if n ≥ 0

0 otherwise

See Also

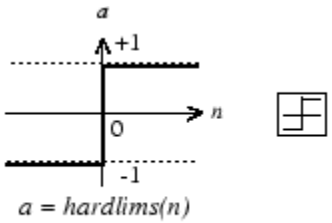
sim | hardlims

Introduced before R2006a

hardlims

Symmetric hard-limit transfer function

Graph and Symbol



Symmetric Hard-Limit Transfer Function

Syntax

```
A = hardlims(N,FP)
```

Description

`hardlims` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = hardlims(N,FP)` takes `N` and optional function parameters,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the S-by-Q +1/–1 matrix with +1s where $N \geq 0$.

`info = hardlims('code')` returns information according to the code string specified:

`hardlims('name')` returns the name of this function.

`hardlims('output',FP)` returns the `[min max]` output range.

`hardlims('active',FP)` returns the `[min max]` active input range.

`hardlims('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`hardlims('fpnames')` returns the names of the function parameters.

`hardlims('fpdefaults')` returns the default function parameters.

Examples

Here is how to create a plot of the `hardlims` transfer function.

```
n = -5:0.1:5;
a = hardlims(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'hardlims';
```

Algorithms

`hardlims(n)` = 1 if n ≥ 0, –1 otherwise.

See Also

`sim` | `hardlim`

Introduced before R2006a

hextop

Hexagonal layer topology function

Syntax

```
hextop(dimensions)
```

Description

`hextop` calculates the neuron positions for layers whose neurons are arranged in an `N`-dimensional hexagonal pattern.

`hextop(dimensions)` takes one argument:

`dimensions`	Row vector of dimension sizes

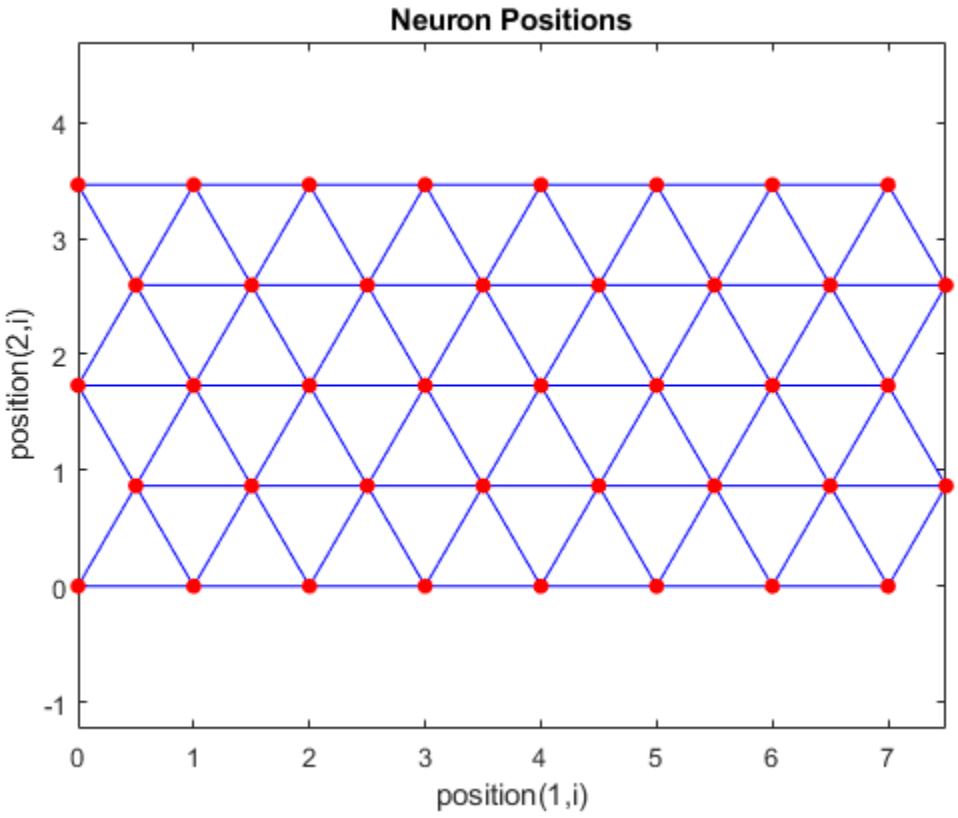
and returns an `N`-by-`S` matrix of `N` coordinate vectors where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Hexagonal Pattern

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 hexagonal pattern.

```
pos = hextop([8 5]);
plotsom(pos)
```



See Also

`gridtop` | `randtop` | `tritop`

Introduced before R2006a

ind2vec

Convert indices to vectors

Syntax

```
vec = ind2vec(ind)
vec = ind2vec(ind,N)
```

Description

`vec = ind2vec(ind)` takes a row vector of indices, `ind`, and returns a sparse matrix of vectors, `vec`, containing a `1` in the row of the index they represent, as indicated by `ind`.

`ind2vec` and `vec2ind` allow indices to be represented either by themselves, or as vectors containing a 1 in the row of the index they represent.

`vec = ind2vec(ind,N)` returns an `N`-by-`M` sparse matrix, where `N` can be equal to or greater than the maximum index.

Examples

Convert Indices into Vector Representation

This example shows how to convert indices to vector representation using the `ind2vec` function.

Define four indices and then convert them to vector representation.

```
ind = [1 3 2 3];
vec = ind2vec(ind)
```

```
vec =
   (1,1)        1
   (3,2)        1
   (2,3)        1
   (3,4)        1
```

Convert a Vector to Indices and Back

This example shows how to convert a vector to indices and back, using both the `ind2vec` and `vec2ind` functions.

Define a vector with all zeros in the last row and convert it to indices.

```
vec = [0 0 1 0; 1 0 0 0; 0 1 0 0]'
[ind,n] = vec2ind(vec)
```

```
vec =
     0     1     0
     0     0     1
```

```
     1     0     0
     0     0     0
ind =
     3     1     2

n =
     4
```

Convert the indices to vector, while preserving the number of rows.

```
vec2 = full(ind2vec(ind,n))
```

```
vec2 =
     0     1     0
     0     0     1
     1     0     0
     0     0     0
```

Input Arguments

`ind` — Indices

row vector

Indices, specified as a row vector.

`N` — Number of rows

scalar

Number of rows of the output matrix, specified as a scalar.

Output Arguments

`vec` — Converted vector of indices

matrix

Vector representation of the indices, returned as an `N`-by-`M` sparse matrix.

See Also

`vec2ind` | `sub2ind` | `ind2sub`

Introduced before R2006a

init

Initialize neural network

Syntax

```
init_net = init(net)
```

Description

`init_net = init(net)` returns a neural network `net` with weight and bias values updated according to the network initialization function, specified by `net.initFcn`, and the parameter values, specified by `net.initParam`.

For more information on this function, at the MATLAB command prompt, type `help network/init`.

Examples

Reinitialize a Perceptron with 'init' Function

This example shows how to reinitialize a perceptron network by using the `init` function.

Create a perceptron and configure it so that its input, output, weight, and bias dimensions match the input and target data.

```
x = [0 1 0 1; 0 0 1 1];
t = [0 0 0 1];
net = perceptron;
net = configure(net,x,t);
net.iw{1,1}
net.b{1}
```

Train the perceptron to alter its weight and bias values.

```
net = train(net,x,t);
net.iw{1,1}
net.b{1}
```

`init` reinitializes those weight and bias values.

```
net = init(net);
net.iw{1,1}
net.b{1}
```

The weights and biases are zeros again, which are the initial values used by perceptron networks.

Input Arguments

net — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`init_net` — Reinitialized network

network

Network after the `init` reinitialization, returned as a network object.

Algorithms

`init` calls `net.initFcn` to initialize the weight and bias values according to the parameter values `net.initParam`.

Typically, `net.initFcn` is set to `'initlay'`, which initializes each layer's weights and biases according to its `net.layers{i}.initFcn`.

Backpropagation networks have `net.layers{i}.initFcn` set to `'initnw'`, which calculates the weight and bias values for layer `i` using the Nguyen-Widrow initialization method.

Other networks have `net.layers{i}.initFcn` set to `'initwb'`, which initializes each weight and bias with its own initialization function. The most common weight and bias initialization function is `rands`, which generates random values between –1 and 1.

See Also

`sim` | `adapt` | `train` | `initlay` | `initnw` | `initwb` | `rands` | `revert`

Introduced before R2006a

initcon

Conscience bias initialization function

Syntax

```
initcon (S,PR)
```

Description

`initcon` is a bias initialization function that initializes biases for learning with the `learncon` learning function.

`initcon (S,PR)` takes two arguments,

S	Number of rows (neurons)
PR	R-by-2 matrix of `R = [Pmin Pmax]` (default = `[1 1]`)

and returns an S-by-1 bias vector.

Note that for biases, R is always 1. `initcon` could also be used to initialize weights, but it is not recommended for that purpose.

Examples

Here initial bias values are calculated for a five-neuron layer.

```
b = initcon(5)
```

Network Use

You can create a standard network that uses `initcon` to initialize weights by calling `competlayer`.

To prepare the bias of layer `i` of a custom network to initialize with `initcon`,

1. Set `net.initFcn` to `'initlay'`. (`net.initParam` automatically becomes `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set `net.biases{i}.initFcn` to `'initcon'`.

To initialize the network, call `init`.

Algorithms

`learncon` updates biases so that each bias value `b(i)` is a function of the average output `c(i)` of the neuron `i` associated with the bias.

`initcon` gets initial bias values by assuming that each neuron has responded to equal numbers of vectors in the past.

See Also

`competlayer` | `init` | `initlay` | `initwb` | `learncon`

Introduced before R2006a

initlay

Layer-by-layer network initialization function

Syntax

```
net = initlay(net)
info = initlay('code')
```

Description

`initlay` is a network initialization function that initializes each layer `i` according to its own initialization function `net.layers{i}.initFcn`.

`net = initlay(net)` takes

`net`	Neural network

and returns the network with each layer updated.

`info = initlay('code')` returns useful information for each supported `code` character vector:

`'pnames'`	Names of initialization parameters
`'pdefaults'`	Default initialization parameters

`initlay` does not have any initialization parameters.

Network Use

You can create a standard network that uses `initlay` by calling `feedforwardnet`, `cascadeforwardnet`, and many other network functions.

To prepare a custom network to be initialized with `initlay`,

1. Set `net.initFcn` to `'initlay'`. This sets `net.initParam` to the empty matrix `[]`, because `initlay` has no initialization parameters.
2. Set each `net.layers{i}.initFcn` to a layer initialization function. (Examples of such functions are `initwb` and `initnw`.)

To initialize the network, call `init`.

Algorithms

The weights and biases of each layer `i` are initialized according to `net.layers{i}.initFcn`.

See Also

`cascadeforwardnet` | `feedforwardnet` | `init` | `initnw` | `initwb`

Introduced before R2006a

initlvq

LVQ weight initialization function

Syntax

```
initlvq('configure',x)
initlvq('configure',net,'IW',i,j,settings)
initlvq('configure',net,'LW',i,j,settings)
initlvq('configure',net,'b',i,)
```

Description

`initlvq('configure',x)` takes input data `x` and returns initialization settings for an LVQ weights associated with that input.

`initlvq('configure',net,'IW',i,j,settings)` takes a network, and indices indicating an input weight to layer `i` from input `j`, and that weights settings, and returns new weight values.

`initlvq('configure',net,'LW',i,j,settings)` takes a network, and indices indicating a layer weight to layer `i` from layer `j`, and that weights settings, and returns new weight values.

`initlvq('configure',net,'b',i,)` takes a network, and an index indicating a bias for layer `i`, and returns new bias values.

See Also

`lvqnet` | `init`

Introduced in R2010b

initnw

Nguyen-Widrow layer initialization function

Syntax

```
net = initnw(net,i)
```

Description

`initnw` is a layer initialization function that initializes a layer's weights and biases according to the Nguyen-Widrow initialization algorithm. This algorithm chooses values in order to distribute the active region of each neuron in the layer approximately evenly across the layer's input space. The values contain a degree of randomness, so they are not the same each time this function is called.

`initnw` requires that the layer it initializes have a transfer function with a finite active input range. This includes transfer functions such as `tansig` and `satlin`, but not `purelin`, whose active input range is the infinite interval `[-inf, inf]`. Transfer functions, such as `tansig`, will return their active input range as follows:

```
activeInputRange = tansig('active')
activeInputRange =
    -2     2
```

`net = initnw(net,i)` takes two arguments,

`net`	Neural network
`i`	Index of a layer

and returns the network with layer `i`'s weights and biases updated.

There is a random element to Nguyen-Widrow initialization. Unless the default random generator is set to the same seed before each call to `initnw`, it will generate different weight and bias values each time.

Network Use

You can create a standard network that uses `initnw` by calling `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be initialized with `initnw`,

1. Set `net.initFcn` to `'initlay'`. This sets `net.initParam` to the empty matrix `[]`, because `initlay` has no initialization parameters.
2. Set `net.layers{i}.initFcn` to `'initnw'`.

To initialize the network, call `init`.

Algorithms

The Nguyen-Widrow method generates initial weight and bias values for a layer so that the active regions of the layer's neurons are distributed approximately evenly over the input space.

Advantages over purely random weights and biases are

- Few neurons are wasted (because all the neurons are in the input space).
- Training works faster (because each area of the input space has neurons). The Nguyen-Widrow method can only be applied to layers
  - With a bias
  - With weights whose `weightFcn` is `dotprod`
  - With `netInputFcn` set to `netsum`
  - With `transferFcn` whose active region is finite

If these conditions are not met, then `initnw` uses `rands` to initialize the layer's weights and biases.

See Also

`cascadeforwardnet` | `feedforwardnet` | `init` | `initlay` | `initwb`

Introduced before R2006a

initwb

By weight and bias layer initialization function

Syntax

```
initwb(net,i)
```

Description

`initwb` is a layer initialization function that initializes a layer's weights and biases according to their own initialization functions.

`initwb(net,i)` takes two arguments,

`net`	Neural network
`i`	Index of a layer

and returns the network with layer `i`'s weights and biases updated.

Network Use

You can create a standard network that uses `initwb` by calling `perceptron` or `linearlayer`.

To prepare a custom network to be initialized with `initwb`,

1. Set `net.initFcn` to `'initlay'`. This sets `net.initParam` to the empty matrix `[]`, because `initlay` has no initialization parameters.
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set each `net.inputWeights{i,j}.initFcn` to a weight initialization function. Set each `net.layerWeights{i,j}.initFcn` to a weight initialization function. Set each `net.biases{i}.initFcn` to a bias initialization function. Examples of initialization functions are `rands` (for weights and biases) and `midpoint` (for weights only).

To initialize the network, call `init`.

Algorithms

Each weight (bias) in layer `i` is set to new values calculated according to its weight (bias) initialization function.

See Also

`init` | `initlay` | `initnw` | `linearlayer` | `perceptron`

Introduced before R2006a

initzero

Zero weight and bias initialization function

Syntax

```
W = initzero(S,PR)
b = initzero(S,[1 1])
```

Description

`W = initzero(S,PR)` takes two arguments,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R weight matrix of zeros.

`b = initzero(S,[1 1])` returns an S-by-1 bias vector of zeros.

Examples

Here initial weights and biases are calculated for a layer with two inputs ranging over `[0 1]` and `[-2 2]` and four neurons.

```
W = initzero(5,[0 1; -2 2])
b = initzero(5,[1 1])
```

See Also

`initwb` | `initlay` | `init`

Introduced before R2006a

isconfigured

Indicate if network inputs and outputs are configured

Syntax

```
[flag,inputflags,outputflags] = isconfigured(net)
```

Description

`[flag,inputflags,outputflags] = isconfigured(net)` takes a neural network and returns three values,

`flag`	True if all network inputs and outputs are configured (have non-zero sizes)
`inputflags`	Vector of true/false values for each configured/unconfigured input
`outputflags`	Vector of true/false values for each configured/unconfigured output

Examples

Here are the flags returned for a new network before and after being configured:

```
net = feedforwardnet;
[flag,inputFlags,outputFlags] = isconfigured(net)
[x,t] = simplefit_dataset;
net = configure(net,x,t);
[flag,inputFlags,outputFlags] = isconfigured(net)
```

See Also

`configure` | `unconfigure`

Introduced in R2010b

layrecnet

Layer recurrent neural network

Syntax

```
layrecnet(layerDelays,hiddenSizes,trainFcn)
```

Description

`layrecnet(layerDelays,hiddenSizes,trainFcn)` takes these arguments:

- Row vector of increasing 0 or positive delays, `layerDelays`
- Row vector of one or more hidden layer sizes, `hiddenSizes`
- Backpropagation training function, `trainFcn`

and returns a layer recurrent neural network.

Layer recurrent neural networks are similar to feedforward networks, except that each layer has a recurrent connection with a tap delay associated with it. This allows the network to have an infinite dynamic response to time series input data. This network is similar to the time delay (`timedelaynet`) and distributed delay (`distdelaynet`) neural networks, which have finite input responses.

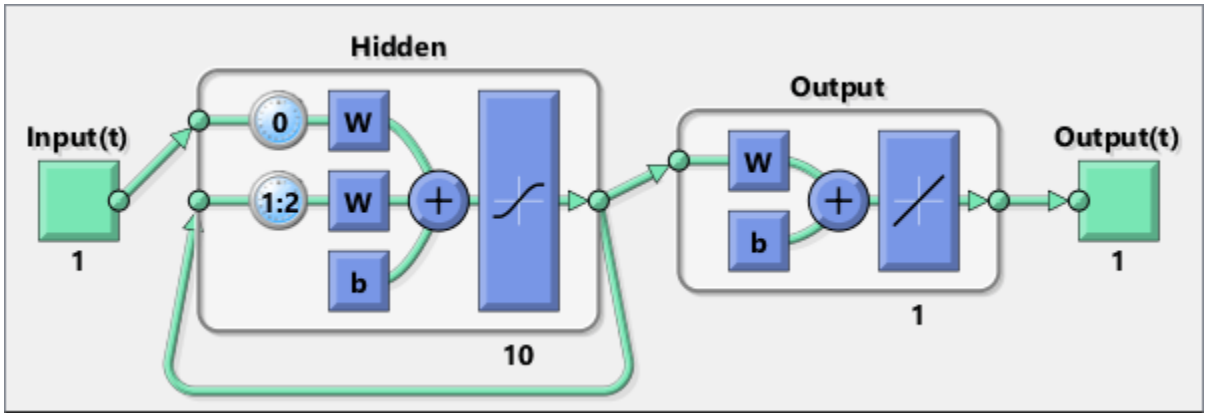
Examples

Recurrent Neural Network

This example shows how to use a layer recurrent neural network to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;
net = layrecnet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
```

```
perf =

   6.1239e-11
```



Input Arguments

`layerDelays` — Input delays

`[1:2]` (default) | row vector

Zero or positive input delays, specified as an increasing row vector.

`hiddenSizes` — Hidden sizes

`10` (default) | row vector

Sizes of the hidden layers, specified as a row vector of one or more elements.

`trainFcn` — Training function name

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

See Also

`preparets` | `removedelay` | `distdelaynet` | `timedelaynet` | `narnet` | `narxnet`

Introduced in R2010b

learncon

Conscience bias learning function

Syntax

```
[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learncon('code')
```

Description

`learncon` is the conscience bias learning function used to increase the net input to neurons that have the lowest average output until each neuron responds approximately an equal percentage of the time.

`[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`B`	`S`-by-`1` bias vector
`P`	`1`-by-`Q` ones vector
`Z`	`S`-by-`Q` weighted input vectors
`N`	`S`-by-`Q` net input vectors
`A`	`S`-by-`Q` output vectors
`T`	`S`-by-`Q` layer target vectors
`E`	`S`-by-`Q` layer error vectors
`gW`	`S`-by-`R` gradient with respect to performance
`gA`	`S`-by-`Q` output gradient with respect to performance
`D`	`S`-by-`S` neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dB`	`S`-by-`1` weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learncon`'s learning parameter, shown here with its default value.

`LP.lr - 0.001`	Learning rate

`info = learncon('code')` returns useful information for each supported *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Deep Learning Toolbox 2.0 compatibility: The `LP.lr` described above equals 1 minus the bias time constant used by `trainc` in the Deep Learning Toolbox 2.0 software.

Examples

Here you define a random output `A` and bias vector `W` for a layer with three neurons. You also define the learning rate `LR`.

```
a = rand(3,1);
b = rand(3,1);
lp.lr = 0.5;
```

Because `learncon` only needs these values to calculate a bias change (see "Algorithm" below), use them to do so.

```
dW = learncon(b,[],[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the bias of layer `i` of a custom network to learn with `learncon`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set `net.inputWeights{i}.learnFcn` to `'learncon'`
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learncon'`. .(Each weight learning parameter property is automatically set to `learncon`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties as desired.
2. Call `train` (or `adapt`).

Algorithms

`learncon` calculates the bias change `db` for a given neuron by first updating each neuron's *conscience*, i.e., the running average of its output:

```
c = (1-lr)*c + lr*a
```

The conscience is then used to compute a bias for the neuron that is greatest for smaller conscience values.

```
b = exp(1-log(c)) - b
```

(`learncon` recovers `C` from the bias values each time it is called.)

See Also

`learnk` | `learnos` | `adapt` | `train`

Introduced before R2006a

learngd

Gradient descent weight and bias learning function

Syntax

```
[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngd('code')
```

Description

`learngd` is the gradient descent weight and bias learning function.

`[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs:

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q output gradient with respect to performance x Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `[]`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learngd`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learngd('code')` returns useful information for each supported *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random gradient `gW` for a weight going to a layer with three neurons from an input with two elements. Also define a learning rate of 0.5.

```
gW = rand(3,2);
lp.lr = 0.5;
```

Because `learngd` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learngd([],[],[],[],[],[],[],gW,[],[],lp,[])
```

Algorithms

`learngd` calculates the weight change `dW` for a given neuron from the neuron's input `P` and error `E`, and the weight (or bias) learning rate `LR`, according to the gradient descent `dw = lr*gW`.

See Also

`adapt` | `learngdm` | `train`

Introduced before R2006a

learngdm

Gradient descent with momentum weight and bias learning function

Syntax

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngdm('code')
```

Description

`learngdm` is the gradient descent with momentum weight and bias learning function.

`[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learngdm`'s learning parameters, shown here with their default values.

`LP.lr - 0.01`	Learning rate
`LP.mc - 0.9`	Momentum constant

`info = learngdm('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random gradient `G` for a weight going to a layer with three neurons from an input with two elements. Also define a learning rate of 0.5 and momentum constant of 0.8:

```
gW = rand(3,2);
lp.lr = 0.5;
lp.mc = 0.8;
```

Because `learngdm` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so. Use the default initial learning state.

```
ls = [];
[dW,ls] = learngdm([],[],[],[],[],[],[],gW,[],[],lp,ls)
```

`learngdm` returns the weight change and a new learning state.

Algorithms

`learngdm` calculates the weight change `dW` for a given neuron from the neuron's input `P` and error `E`, the weight (or bias) `W`, learning rate `LR`, and momentum constant `MC`, according to gradient descent with momentum:

```
dW = mc*dWprev + (1-mc)*lr*gW
```

The previous weight change `dWprev` is stored and read from the learning state `LS`.

See Also

`adapt` | `learngd` | `train`

Introduced before R2006a

learnh

Hebb weight learning rule

Syntax

```
[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnh('code')
```

Description

`learnh` is the Hebb weight learning function.

`[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnh`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learnh('code')` returns useful information for each `code` character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input P and output A for a layer with a two-element input and three neurons. Also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
lp.lr = 0.5;
```

Because learnh only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnh([],p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnh,

1. Set net.trainFcn to 'trainr'. (net.trainParam automatically becomes trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam automatically becomes trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnh'.
4. Set each net.layerWeights{i,j}.learnFcn to 'learnh'. (Each weight learning parameter property is automatically set to learnh's default parameters.)

To train the network (or enable it to adapt),

1. Set net.trainParam (or net.adaptParam) properties to desired values.
2. Call train (adapt).

Algorithms

learnh calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the Hebb learning rule:

```
dw = lr*a*p'
```

References

Hebb, D.O., *The Organization of Behavior*, New York, Wiley, 1949

See Also

learnhd | adapt | train

Introduced before R2006a

learnhd

Hebb with decay weight learning rule

Syntax

```
[dW,LS] = learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnhd('code')
```

Description

`learnhd` is the Hebb weight learning function.

`[dW,LS] = learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnhd`'s learning parameters, shown here with default values.

`LP.dr - 0.01`	Decay rate
`LP.lr - 0.1`	Learning rate

`info = learnhd('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P`, output `A`, and weights `W` for a layer with a two-element input and three neurons. Also define the decay and learning rates.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.dr = 0.05;
lp.lr = 0.5;
```

Because `learnhd` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnhd(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnhd`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnhd'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnhd'`. (Each weight learning parameter property is automatically set to `learnhd`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2. Call `train` (`adapt`).

Algorithms

`learnhd` calculates the weight change `dW` for a given neuron from the neuron's input `P`, output `A`, decay rate `DR`, and learning rate `LR` according to the Hebb with decay learning rule:

```
dw = lr*a*p' - dr*w
```

See Also

`learnh` | `adapt` | `train`

Introduced before R2006a

learnis

Instar weight learning function

Syntax

```
[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnis('code')
```

Description

`learnis` is the instar weight learning function.

`[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnis`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learnis('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Because `learnis` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnis(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network so that it can learn with `learnis`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnis'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnis'`. (Each weight learning parameter property is automatically set to `learnis`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (`net.adaptParam`) properties to desired values.
2. Call `train` (`adapt`).

Algorithms

`learnis` calculates the weight change `dW` for a given neuron from the neuron's input `P`, output `A`, and learning rate `LR` according to the instar learning rule:

```
dw = lr*a*(p'-w)
```

References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland, Reidel Press, 1982

See Also

`learnk` | `learnos` | `adapt` | `train`

Introduced before R2006a

learnk

Kohonen weight learning function

Syntax

```
[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnk('code')
```

Description

`learnk` is the Kohonen weight learning function.

`[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnk`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learnk('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Because `learnk` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnk(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights of layer `i` of a custom network to learn with `learnk`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnk'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnk'`. (Each weight learning parameter property is automatically set to `learnk`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties as desired.
2. Call `train` (or `adapt`).

Algorithms

`learnk` calculates the weight change `dW` for a given neuron from the neuron's input `P`, output `A`, and learning rate `LR` according to the Kohonen learning rule:

`dw = lr*(p'-w)`, if `a ~= 0`; `= 0`, otherwise

References

Kohonen, T., *Self-Organizing and Associative Memory*, New York, Springer-Verlag, 1984

See Also

`learnis` | `learnos` | `adapt` | `train`

Introduced before R2006a

learnlv1

LVQ1 weight learning function

Syntax

```
[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv1('code')
```

Description

`learnlv1` is the LVQ1 weight learning function.

`[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnlv1`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learnlv1('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input P, output A, weight matrix W, and output gradient gA for a layer with a two-element input and three neurons. Also define the learning rate LR.

```
p = rand(2,1);
w = rand(3,2);
a = compet(negdist(w,p));
gA = [-1;1; 1];
lp.lr = 0.5;
```

Because learnlv1 only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnlv1(w,p,[],[],a,[],[],[],gA,[],lp,[])
```

Network Use

You can create a standard network that uses learnlv1 with lvqnet. To prepare the weights of layer i of a custom network to learn with learnlv1,

1. Set net.trainFcn to 'trainr'. (net.trainParam automatically becomes trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam automatically becomes trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnlv1'.
4. Set each net.layerWeights{i,j}.learnFcn to 'learnlv1'. (Each weight learning parameter property is automatically set to learnlv1's default parameters.)

To train the network (or enable it to adapt),

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

Algorithms

learnlv1 calculates the weight change dW for a given neuron from the neuron's input P, output A, output gradient gA, and learning rate LR, according to the LVQ1 rule, given i, the index of the neuron whose output a(i) is 1:

dw(i,:) = +lr*(p-w(i,:)) if gA(i) = 0;= -lr*(p-w(i,:)) if gA(i) = -1

See Also

learnlv2 | adapt | train

Introduced before R2006a

learnlv2

LVQ2.1 weight learning function

Syntax

```
[dW,LS] = learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv2('code')
```

Description

`learnlv2` is the LVQ2 weight learning function.

`[dW,LS] = learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnlv2`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate
`LP.window - 0.25`	Window size (0 to 1, typically 0.2 to 0.3)

`info = learnlv2('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a sample input `P`, output `A`, weight matrix `W`, and output gradient `gA` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```
p = rand(2,1);
w = rand(3,2);
n = negdist(w,p);
a = compet(n);
gA = [-1;1; 1];
lp.lr = 0.5;
```

Because `learnlv2` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnlv2(w,p,[],n,a,[],[],[],gA,[],lp,[])
```

Network Use

You can create a standard network that uses `learnlv2` with `lvqnet`.

To prepare the weights of layer `i` of a custom network to learn with `learnlv2`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnlv2'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnlv2'`. (Each weight learning parameter property is automatically set to `learnlv2`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties as desired.
2. Call `train` (or `adapt`).

Algorithms

`learnlv2` implements Learning Vector Quantization 2.1, which works as follows:

For each presentation, if the winning neuron `i` should not have won, and the runnerup `j` should have, and the distance `di` between the winning neuron and the input `p` is roughly equal to the distance `dj` from the runnerup neuron to the input `p` according to the given window,

```
min(di/dj, dj/di) > (1-window)/(1+window)
```

then move the winning neuron `i` weights away from the input vector, and move the runnerup neuron `j` weights toward the input according to

```
dw(i,:) = - lp.lr*(p'-w(i,:))
dw(j,:) = + lp.lr*(p'-w(j,:))
```

See Also

`learnlv1` | `adapt` | `train`

Introduced before R2006a

learnos

Outstar weight learning function

Syntax

```
[dW,LS] = learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnos('code')
```

Description

`learnos` is the outstar weight learning function.

`[dW,LS] = learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnos`'s learning parameter, shown here with its default value.

`LP.lr - 0.01`	Learning rate

`info = learnos('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Because `learnos` only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnos(w,p,[],[],a,[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnos`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnos'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnos'`. (Each weight learning parameter property is automatically set to `learnos`'s default parameters.)

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2. Call `train` (`adapt`).

Algorithms

`learnos` calculates the weight change `dW` for a given neuron from the neuron's input `P`, output `A`, and learning rate `LR` according to the outstar learning rule:

```
dw = lr*(a-w)*p'
```

References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland, Reidel Press, 1982

See Also

`learnis` | `learnk` | `adapt` | `train`

Introduced before R2006a

learnp

Perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnp('code')
```

Description

`learnp` is the perceptron weight/bias learning function.

`[dW,LS] = learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or `b`, and S-by-`1` bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

`info = learnp('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P` and error `E` for a layer with a two-element input and three neurons.

```
p = rand(2,1);
e = rand(3,1);
```

Because `learnp` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```
dW = learnp([],p,[],[],[],[],e,[],[],[],[],[])
```

Algorithms

`learnp` calculates the weight change `dW` for a given neuron from the neuron’s input `P` and error `E` according to the perceptron learning rule:

```
dw = 0, if e = 0
     = p', if e = 1
     = -p', if e = -1
```

This can be summarized as

```
dw = e*p'
```

References

Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C., Spartan Press, 1961

See Also

`adapt` | `learnpn` | `train`

Introduced before R2006a

learnpn

Normalized perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnpn('code')
```

Description

`learnpn` is a weight and bias learning function. It can result in faster learning than `learnp` when input vectors have widely varying magnitudes.

`[dW,LS] = learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

`info = learnpn('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input `P` and error `E` for a layer with a two-element input and three neurons.

```
p = rand(2,1);
e = rand(3,1);
```

Because `learnpn` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```
dW = learnpn([],p,[],[],[],[],e,[],[],[],[],[])
```

Limitations

Perceptrons do have one real limitation. The set of input vectors must be linearly separable if a solution is to be found. That is, if the input vectors with targets of 1 cannot be separated by a line or hyperplane from the input vectors associated with values of 0, the perceptron will never be able to classify them correctly.

Algorithms

`learnpn` calculates the weight change `dW` for a given neuron from the neuron’s input `P` and error `E` according to the normalized perceptron learning rule:

```
pn = p / sqrt(1 + p(1)^2 + p(2)^2) + ... + p(R)^2)
dw = 0,  if e = 0
     = pn', if e = 1
     = -pn', if e = -1
```

The expression for `dW` can be summarized as

```
dw = e*pn'
```

See Also

`adapt` | `learnp` | `train`

Introduced before R2006a

learnsom

Self-organizing map weight learning function

Syntax

```
[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnsom('code')
```

Description

`learnsom` is the self-organizing map weight learning function.

`[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnsom`'s learning parameters, shown here with their default values.

`LP.order_lr`	`0.9`	Ordering phase learning rate
`LP.order_steps`	`1000`	Ordering phase steps
`LP.tune_lr`	`0.02`	Tuning phase learning rate
`LP.tune_nd`	`1`	Tuning phase neighborhood distance

`info = learnsom('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters

'pdefaults'	Default learning parameters
'needg'	Returns 1 if this function uses gW or gA

Examples

Here you define a random input P, output A, and weight matrix W for a layer with a two-element input and six neurons. You also calculate positions and distances for the neurons, which are arranged in a 2-by-3 hexagonal pattern. Then you define the four learning parameters.

```
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp.order_lr = 0.9;
lp.order_steps = 1000;
lp.tune_lr = 0.02;
lp.tune_nd = 1;
```

Because learnsom only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
ls = [];
[dW,ls] = learnsom(w,p,[],[],a,[],[],[],[],d,lp,ls)
```

Algorithms

learnsom calculates the weight change dW for a given neuron from the neuron's input P, activation A2, and learning rate LR:

```
dw = lr*a2*(p'-w)
```

where the activation A2 is found from the layer output A, neuron distances D, and the current neighborhood size ND:

```
a2(i,q) = 1,  if a(i,q) = 1
         = 0.5, if a(j,q) = 1 and D(i,j) <= nd
         = 0, otherwise
```

The learning rate LR and neighborhood size NS are altered through two phases: an ordering phase and a tuning phase.

The ordering phases lasts as many steps as LP.order_steps. During this phase LR is adjusted from LP.order_lr down to LP.tune_lr, and ND is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase LR decreases slowly from LP.tune_lr, and ND is always set to LP.tune_nd. During this phase the weights are expected to spread out relatively evenly over the input space while retaining their topological order, determined during the ordering phase.

See Also

adapt | train

Introduced before R2006a

learnsomb

Batch self-organizing map weight learning function

Syntax

```
[dW,LS] = learnsomb(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnsomb('code')
```

Description

`learnsomb` is the batch self-organizing map weight learning function.

`[dW,LS] = learnsomb(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs:

`W`	S-by-R weight matrix (or S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns the following:

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to `learnsomb`'s learning parameter, shown here with its default value:

`LP.init_neighborhood`	3	Initial neighborhood size
`LP.steps`	100	Ordering phase steps

`info = learnsomb('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Returns names of learning parameters.
`'pdefaults'`	Returns default learning parameters.
`'needg'`	Returns 1 if this function uses `gW` or `gA`.

Examples

This example defines a random input `P`, output `A`, and weight matrix `W` for a layer with a 2-element input and 6 neurons. This example also calculates the positions and distances for the neurons, which appear in a 2-by-3 hexagonal pattern.

```
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp = learnsomb('pdefaults');
```

Because `learnsom` only needs these values to calculate a weight change (see Algorithm).

```
ls = [];
[dW,ls] = learnsomb(w,p,[],[],a,[],[],[],[],d,lp,ls)
```

Network Use

You can create a standard network that uses `learnsomb` with `selforgmap`. To prepare the weights of layer i of a custom network to learn with `learnsomb`:

1. Set `NET.trainFcn` to `'trainr'`. (`NET.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `NET.adaptFcn` to `'trains'`. (`NET.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `NET.inputWeights{i,j}.learnFcn` to `'learnsomb'`.
4. Set each `NET.layerWeights{i,j}.learnFcn` to `'learnsomb'`. (Each weight learning parameter property is automatically set to `learnsomb`'s default parameters.)

To train the network (or enable it to adapt):

1. Set `NET.trainParam` (or `NET.adaptParam`) properties as desired.
2. Call `train` (or `adapt`).

Algorithms

`learnsomb` calculates the weight changes so that each neuron's new weight vector is the weighted average of the input vectors that the neuron and neurons in its neighborhood responded to with an output of 1.

The ordering phase lasts as many steps as `LP.steps`.

During this phase, the neighborhood is gradually reduced from a maximum size of `LP.init_neighborhood` down to `1`, where it remains from then on.

See Also

`adapt` | `selforgmap` | `train`

Introduced in R2008a

learnwh

Widrow-Hoff weight/bias learning function

Syntax

```
[dW,LS] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnwh('code')
```

Description

`learnwh` is the Widrow-Hoff weight/bias learning function, and is also known as the delta or least mean squared (LMS) rule.

`[dW,LS] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W`	S-by-R weight matrix (or `b`, and S-by-1 bias vector)
`P`	R-by-Q input vectors (or `ones(1,Q)`)
`Z`	S-by-Q weighted input vectors
`N`	S-by-Q net input vectors
`A`	S-by-Q output vectors
`T`	S-by-Q layer target vectors
`E`	S-by-Q layer error vectors
`gW`	S-by-R weight gradient with respect to performance
`gA`	S-by-Q output gradient with respect to performance
`D`	S-by-S neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be `= []`

and returns

`dW`	S-by-R weight (or bias) change matrix
`LS`	New learning state

Learning occurs according to the `learnwh` learning parameter, shown here with its default value.

`LP.lr — 0.01`	Learning rate

`info = learnwh('code')` returns useful information for each *`code`* character vector:

`'pnames'`	Names of learning parameters
`'pdefaults'`	Default learning parameters
`'needg'`	Returns 1 if this function uses `gW` or `gA`

Examples

Here you define a random input P and error E for a layer with a two-element input and three neurons. You also define the learning rate LR learning parameter.

```
p = rand(2,1);
e = rand(3,1);
lp.lr = 0.5;
```

Because `learnwh` needs only these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnwh([],p,[],[],[],[],e,[],[],[],lp,[])
```

Network Use

You can create a standard network that uses `learnwh` with `linearlayer`.

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnwh`,

1. Set `net.trainFcn` to `'trainb'`. `net.trainParam` automatically becomes `trainb`'s default parameters.
2. Set `net.adaptFcn` to `'trains'`. `net.adaptParam` automatically becomes `trains`'s default parameters.
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnwh'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnwh'`.
5. Set `net.biases{i}.learnFcn` to `'learnwh'`. Each weight and bias learning parameter property is automatically set to the `learnwh` default parameters.

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2. Call `train` (or `adapt`).

Algorithms

`learnwh` calculates the weight change `dW` for a given neuron from the neuron's input `P` and error `E`, and the weight (or bias) learning rate `LR`, according to the Widrow-Hoff learning rule:

```
dw = lr*e*pn'
```

References

Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record*, New York IRE, pp. 96–104, 1960

Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985

See Also

`adapt` | `linearlayer` | `train`

Introduced before R2006a

linearlayer

Create linear layer

Syntax

```
layer = linearlayer(inputDelays,widrowHoffLR)
```

Description

`layer = linearlayer(inputDelays,widrowHoffLR)` takes a row vector of increasing 0 or positive delays and the Widrow-Hoff learning rate, and returns a linear layer.

Linear layers are single layers of linear neurons. They are static, with input delays of 0, or dynamic, with input delays greater than 0. You can train them on simple linear time series problems, but often are used adaptively to continue learning while deployed so they can adjust to changes in the relationship between inputs and outputs while being used.

If the learning rate is too small, learning happens very slowly. However, a greater danger is that it might be too large and learning becomes unstable resulting in large changes to weight vectors and errors increasing instead of decreasing. If a data set is available which characterizes the relationship the layer is to learn, you can calculate the maximum stable learning rate with the `maxlinlr` function.

If you need a network to solve a nonlinear time series relationship, see `timedelaynet`, `narxnet`, and `narnet`.

Examples

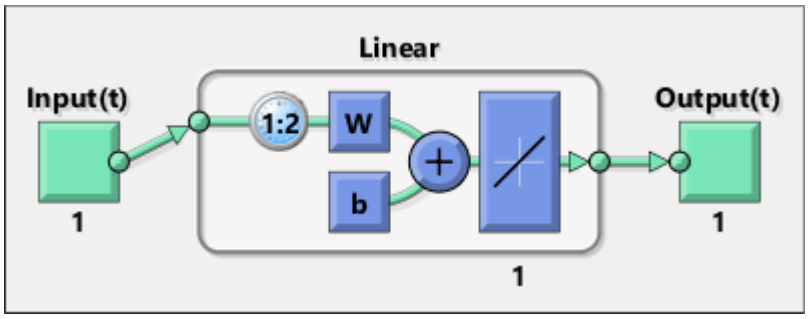
Create and Train a Linear Layer

This example shows how to create and train a linear layer.

Create a linear layer and train it on a simple time series problem.

```
x = {0 -1 1 1 0 -1 1 0 0 1};
t = {0 -1 0 2 1 -1 0 1 0 1};
net = linearlayer(1:2,0.01);
[Xs,Xi,Ai,Ts] = preparets(net,x,t);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi);
perf = perform(net,Ts,Y)
```

```
perf =

    0.2396
```



Input Arguments

`inputDelays` — Input delays

row vector

Increasing 0 or positive delays, specified as a row vector.

`widrowHoffLR` — Widrow-Hoff learning rate

`0.01` (default) | scalar

Widrow-Hoff learning rate, specified as a scalar.

Output Arguments

`layer` — Network layer

network

Linear layer of a network, returned as a network object.

See Also

`preparets` | `removedelay` | `timedelaynet` | `narnet` | `narxnet`

Introduced in R2010b

linkdist

Link distance function

Syntax

```
d = linkdist(pos)
```

Description

`linkdist` is a layer distance function used to find the distances between the layer's neurons given their positions.

`d = linkdist(pos)` takes one argument,

`pos`	`N`-by-`S` matrix of neuron positions

and returns the `S`-by-`S` matrix of distances.

Examples

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = linkdist(pos)
```

Network Use

You can create a standard network that uses `linkdist` as a distance function by calling `selforgmap`.

To change a network so that a layer's topology uses `linkdist`, set `net.layers{i}.distanceFcn` to `'linkdist'`.

In either case, call `sim` to simulate the network with `dist`.

Algorithms

The link distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

```
Dij = 0, if i == j
     = 1, if (sum((Pi-Pj).^2)).^0.5 is <= 1
     = 2, if k exists, Dik = Dkj = 1
     = 3, if k1, k2 exist, Dik1 = Dk1k2 = Dk2j = 1
     = N, if k1..kN exist, Dik1 = Dk1k2 = ...= DkNj = 1
     = S, if none of the above conditions apply
```

See Also

`dist` | `mandist` | `selforgmap` | `sim`

Introduced before R2006a

logsig

Log-sigmoid transfer function

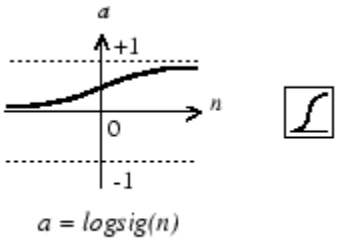
Syntax

```
A = logsig(N)
dA_dN = logsig('dn',N,A,FP)
info = logsig(code)
```

Description

Tip To use a logistic sigmoid activation for deep learning, use `sigmoidLayer` or the `dlarray` method sigmoid.

`A = logsig(N)` takes a matrix of net input vectors, `N` and returns the `S`-by-`Q` matrix, `A`, of the elements of `N` squashed into `[0, 1]`.



Log-Sigmoid Transfer Function

`logsig` is a transfer function. Transfer functions calculate a layer's output from its net input.

`dA_dN = logsig('dn',N,A,FP)` returns the `S`-by-`Q` derivative of `A` with respect to `N`. If `A` or `FP` are not supplied or are set to `[]`, `FP` reverts to the default parameters, and `A` is calculated from `N`.

`info = logsig(code)` returns information about this function. For more information, see the **code** argument description.

Examples

Create a Plot of the `logsig` Transfer Function

This example shows how to calculate and plot the log-sigmoid transfer function of an input matrix.

Create the input matrix, `n`. Then call the `logsig` function and plot the results.

```
n = -5:0.1:5;
a = logsig(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'logsig';
```

Input Arguments

N — Input matrix

matrix

Net input column vectors, specified as an S-by-Q matrix.

code — Information option

`'name'` | `'output'` | `'active'` | `'fullderiv'` | `'fpnames'` | `'fpdefaults'`

Information you want to retrieve from the function, specified as one of the following:

- `'name'` returns the name of this function.
- `'output'` returns the `[min max]` output range.
- `'active'` returns the `[min max]` active input range.
- `'fullderiv'` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.
- `'fpnames'` returns the names of the function parameters.
- `'fpdefaults'` returns the default function parameters.

Output Arguments

A — Output matrix

matrix

Output vectors, returned as an S-by-Q matrix, where each element of `N` is squashed from the interval `[-inf inf]` to the interval `[0 1]` with an "S-shaped" function.

Algorithms

```
logsig(n) = 1 / (1 + exp(-n))
```

See Also

`sim` | `tansig`

Introduced before R2006a

lvqnet

Learning vector quantization neural network

Syntax

```
lvqnet(hiddenSize,lvqLR,lvqLF)
```

Description

LVQ (learning vector quantization) neural networks consist of two layers. The first layer maps input vectors into clusters that are found by the network during training. The second layer merges groups of first layer clusters into the classes defined by the target data.

The total number of first layer clusters is determined by the number of hidden neurons. The larger the hidden layer the more clusters the first layer can learn, and the more complex mapping of input to target classes can be made. The relative number of first layer clusters assigned to each target class are determined according to the distribution of target classes at the time of network initialization. This occurs when the network is automatically configured the first time `train` is called, or manually configured with the function `configure`, or manually initialized with the function `init` is called.

`lvqnet(hiddenSize,lvqLR,lvqLF)` takes these arguments,

`hiddenSize`	Size of hidden layer (default = 10)
`lvqLR`	LVQ learning rate (default = 0.01)
`lvqLF`	LVQ learning function (default = `'learnlv1'`)

and returns an LVQ neural network.

The other option for the `lvq` learning function is `learnlv2`.

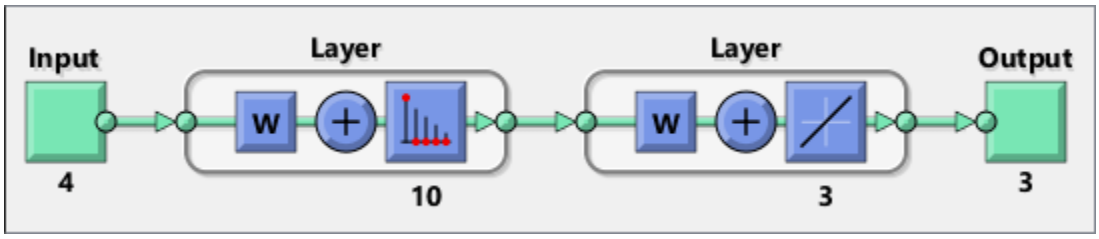
Examples

Train a Learning Vector Quantization Network

Here, an LVQ network is trained to classify iris flowers.

```
[x,t] = iris_dataset;
net = lvqnet(10);
net.trainParam.epochs = 50;
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,y,t)
classes = vec2ind(y);
```

```
perf =

    0.0489
```



See Also

`competlayer` | `patternnet` | `selforgmap`

Introduced in R2010b

lvqoutputs

LVQ outputs processing function

Syntax

```
[X,settings] = lvqoutputs(X)
X = lvqoutputs('apply',X,PS)
X = lvqoutputs('reverse',X,PS)
dx_dy = lvqoutputs('dx_dy',X,X,PS)
```

Description

`[X,settings] = lvqoutputs(X)` returns its argument unchanged, but stores the ratio of target classes in the settings for use by `initlvq` to initialize weights.

`X = lvqoutputs('apply',X,PS)` returns `X`.

`X = lvqoutputs('reverse',X,PS)` returns `X`.

`dx_dy = lvqoutputs('dx_dy',X,X,PS)` returns the identity derivative.

See Also

`lvqnet` | `initlvq`

Introduced in R2010b

mae

Mean absolute error performance function

Syntax

```
perf = mae(E,Y,X)
dPerf_dx = mae('dx',E,Y,X,perf)
info = mae('code')
```

Description

`perf = mae(E,Y,X)` takes a matrix or cell array of error vectors, `E`, and optionally a matrix or cell array of output vectors, `Y`, a vector of all weight and bias values, `X`, and returns network performance as the mean of absolute errors, `perf`.

`dPerf_dx = mae('dx',E,Y,X,perf)` returns the derivative of `perf` with respect to `X`.

`info = mae('code')` returns useful information for each `code` character vector:

- `mae('name')` returns the name of this function.
- `mae('pnames')` returns the names of the training parameters.
- `mae('pdefaults')` returns the default function parameters.

Examples

Calculate Network Performance with 'mae'

This example shows how to calculate the network performance as the mean of absolute errors.

Create and configure a perceptron to have one input and one neuron:

```
net = perceptron;
net = configure(net,0,0);
```

The network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the mean absolute error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = net(p)
e = t-y
perf = mae(e)
```

Note that `mae` can be called with only one argument because the other arguments are ignored. `mae` supports those arguments to conform to the standard performance function argument list.

Input Arguments

E — Errors

vector | matrix | cell array

Errors, specified as a vector, a matrix, or a cell array.

Y — Outputs

vector | matrix | cell array

Network outputs, specified as a vector, a matrix, or a cell array.

X — Weight and bias

vector

Weight and bias values, specified as a vector.

Output Arguments

perf — Network performance

scalar

Network performance as the mean of absolute errors, returned as a scalar.

dPerf_dx — Derivative of network performance

scalar

Derivative of `perf` with respect to `X`, returned as a scalar.

More About

Network Use

You can create a standard network that uses `mae` with `perceptron`.

To prepare a custom network to be trained with `mae`, set `net.performFcn` to `'mae'`. This automatically sets `net.performParam` to the empty matrix `[]`, because `mae` has no performance parameters.

In either case, calling `train` or `adapt`, results in `mae` being used to calculate performance.

See Also

`mse` | `perceptron`

Introduced before R2006a

mandist

Manhattan distance weight function

Syntax

```
Z = mandist(W,P)
D = mandist(pos)
```

Description

`Z = mandist(W,P)` takes an `S`-by-`R` weight matrix, `W`, and an `R`-by-`Q` matrix of `Q` input (column) vectors, `P`, and returns the `S`-by-`Q` matrix of vector distances, `Z`.

`mandist` is the Manhattan distance weight function. Weight functions apply weights to an input to get weighted inputs.

`mandist` is also a layer distance function, which can be used to find the distances between neurons in a layer.

`D = mandist(pos)` takes the `N`-by-`S` matrix of neuron positions, `pos`, and returns the `S`-by-`S` matrix of distances, `D`.

Examples

Calculate the Weighted Input Matrix

This example shows how to calculate the weighted input matrix.

Define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = mandist(W,P)
```

Find the Distances of 10 Neurons

This example shows how to calculate the distances of 10 neurons arranged in a three-dimensional space.

Define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
D = mandist(pos)
```

Input Arguments

W — Weight matrix

matrix

Weight matrix, specified as an S-by-R matrix.

P — Input matrix

matrix

Input matrix, specified as an R-by-Q matrix of Q input (column) vectors.

pos — Neuron positions

row matrix

Matrix of neuron positions, specified as an N-by-S matrix.

Output Arguments

Z — Vector distances

matrix

Matrix of vector distances, returned as an S-by-Q matrix.

D — Distances

matrix

Matrix of distances, returned as an S-by-S matrix.

More About

Network Use

To change a network so an input weight uses `mandist`, set `net.inputWeights{i,j}.weightFcn` to `'mandist'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'mandist'`.

To change a network so a layer's topology uses `mandist`, set `net.layers{i}.distanceFcn` to `'mandist'`.

In either case, call `sim` to simulate the network with `dist`. See `newpnn` or `newgrnn` for simulation examples.

Algorithms

The Manhattan distance D between two vectors X and Y is

```
D = sum(abs(x-y))
```

See Also

`dist` | `linkdist` | `sim`

Introduced before R2006a

mapminmax

Process matrices by mapping row minimum and maximum values to [-1 1]

Syntax

```
[Y,PS] = mapminmax(X,YMIN,YMAX)
[Y,PS] = mapminmax(X,FP)
Y = mapminmax('apply',X,PS)
X = mapminmax('reverse',Y,PS)
dx_dy = mapminmax('dx_dy',X,Y,PS)
```

Description

Tip To rescale data for deep learning workflows, use the Normalization name value pair for the input layer.

`[Y,PS] = mapminmax(X,YMIN,YMAX)` takes a `N`-by-`Q` matrix, `X` and optionally a minimum and a maximum value for each row of `Y`, `YMIN` and `YMAX`, and returns a `N`-by-`Q` matrix, `Y`, and a process settings that allow consistent processing of values, `PS`.

`mapminmax` processes matrices by normalizing the minimum and maximum values of each row to [`YMIN`, `YMAX`].

`[Y,PS] = mapminmax(X,FP)` takes parameters as a struct: `FP.ymin`, `FP.ymax`.

`Y = mapminmax('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = mapminmax('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`dx_dy = mapminmax('dx_dy',X,Y,PS)` returns the reverse derivative.

Examples

Format a Matrix With the `mapminmax` Function

This example shows how to format a matrix so that the minimum and maximum values of each row are mapped to default interval [-1,+1].

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,PS] = mapminmax(x1)
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = mapminmax('apply',x2,PS)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = mapminmax('reverse',y1,PS)
```

Input Arguments

X — Input matrix

matrix

Matrix you want to process, specified as an N-by-Q matrix.

YMIN — Minimum value

-1 (default) | scalar

Minimum value for each row of the output matrix Y, specified as a scalar.

YMAX — Maximum value

-1 (default) | scalar

Maximum value for each row of the output matrix Y, specified as a scalar.

Output Arguments

Y — Output matrix

matrix

Processed matrix, returned as an N-by-Q matrix.

PS — Process settings

structure

Process settings that allow consistent processing of values, returned as a structure.

More About

Normalize Inputs and Targets Using `mapminmax`

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function `mapminmax` scales inputs and targets so that they fall in the range [–1,1]. The following code illustrates how to use this function.

```
[pn,ps] = mapminmax(p);
[tn,ts] = mapminmax(t);
net = train(net,pn,tn);
```

The original network inputs and targets are given in the matrices `p` and `t`. The normalized inputs and targets `pn` and `tn` that are returned will all fall in the interval [–1,1]. The structures `ps` and `ts` contain the settings, in this case the minimum and maximum values of the original inputs and targets. After the network has been trained, the `ps` settings should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If `mapminmax` is used to scale the targets, then the output of the network will be trained to produce outputs in the range [–1,1]. To convert these outputs back into the same units that were used for the original targets, use the settings `ts`. The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.

```
an = sim(net,pn);
a = mapminmax('reverse',an,ts);
```

The network output `an` corresponds to the normalized targets `tn`. The unnormalized network output `a` is in the same units as the original targets `t`.

If `mapminmax` is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums that were computed for the training set stored in the settings `ps`. The following code applies a new set of inputs to the network already trained.

```
pnewn = mapminmax('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapminmax('reverse',anewn,ts);
```

For most networks, including `feedforwardnet`, these steps are done automatically, so that you only need to use the `sim` command.

Algorithms

It is assumed that `X` has only finite real values, and that the elements of each row are not all equal. (If `xmax=xmin` or if either `xmax` or `xmin` are non-finite, then `y=x` and no change occurs.)

```
y = (ymax-ymin)*(x-xmin)/(xmax-xmin) + ymin;
```

See Also

`fixunknowns` | `mapstd` | `processpca`

Introduced in R2006a

mapstd

Process matrices by mapping each row's means to 0 and deviations to 1

Syntax

```
[Y,PS] = mapstd(X,ymean,ystd)
[Y,PS] = mapstd(X,FP)
Y = mapstd('apply',X,PS)
X = mapstd('reverse',Y,PS)
dx_dy = mapstd('dx_dy',X,Y,PS)
```

Description

`mapstd` processes matrices by transforming the mean and standard deviation of each row to `ymean` and `ystd`.

`[Y,PS] = mapstd(X,ymean,ystd)` takes `X` and optional parameters,

`X`	`N`-by-`Q` matrix
`ymean`	Mean value for each row of `Y` (default is 0)
`ystd`	Standard deviation for each row of `Y` (default is 1)

and returns

`Y`	`N`-by-`Q` matrix
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = mapstd(X,FP)` takes parameters as a struct: `FP.ymean`, `FP.ystd`.

`Y = mapstd('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = mapstd('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`dx_dy = mapstd('dx_dy',X,Y,PS)` returns the reverse derivative.

Examples

Here you format a matrix so that the minimum and maximum values of each row are mapped to default mean and STD of 0 and 1.

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,PS] = mapstd(x1)
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = mapstd('apply',x2,PS)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = mapstd('reverse',y1,PS)
```

More About

Normalize Network Inputs and Targets Using `mapstd`

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. The function `mapstd` normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of `mapstd`.

```
[pn,ps] = mapstd(p);
[tn,ts] = mapstd(t);
```

The original network inputs and targets are given in the matrices `p` and `t`. The normalized inputs and targets `pn` and `tn` that are returned will have zero means and unity standard deviation. The settings structures `ps` and `ts` contain the means and standard deviations of the original inputs and original targets. After the network has been trained, you should use these settings to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If `mapstd` is used to scale the targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. To convert these outputs back into the same units that were used for the original targets, use `ts`. The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.

```
an = sim(net,pn);
a = mapstd('reverse',an,ts);
```

The network output `an` corresponds to the normalized targets `tn`. The unnormalized network output `a` is in the same units as the original targets `t`.

If `mapstd` is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the means and standard deviations that were computed for the training set using `ps`. The following commands apply a new set of inputs to the network already trained:

```
pnewn = mapstd('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapstd('reverse',anewn,ts);
```

For most networks, including `feedforwardnet`, these steps are done automatically, so that you only need to use the `sim` command.

Algorithms

It is assumed that `X` has only finite real values, and that the elements of each row are not all equal.

```
y = (x-xmean)*(ystd/xstd) + ymean;
```

See Also

`fixunknowns` | `mapminmax` | `processpca`

Introduced in R2006a

maxlinlr

Maximum learning rate for linear layer

Syntax

```
lr = maxlinlr(P)
lr = maxlinlr(P,'bias')
```

Description

`maxlinlr` is used to calculate learning rates for `linearlayer`.

`lr = maxlinlr(P)` takes one argument,

P	R-by-Q matrix of input vectors

and returns the maximum learning rate for a linear layer without a bias that is to be trained only on the vectors in P.

`lr = maxlinlr(P,'bias')` returns the maximum learning rate for a linear layer with a bias.

Examples

Here you define a batch of four two-element input vectors and find the maximum learning rate for a linear layer with a bias.

```
P = [1 2 -4 7; 0.1 3 10 6];
lr = maxlinlr(P,'bias')
```

See Also

`learnwh` | `linearlayer`

Introduced before R2006a

meanabs

Mean of absolute elements of matrix or matrices

Syntax

```
[m,n] = meanabs(x)
```

Description

`[m,n] = meanabs(x)` takes a matrix or cell array of matrices and returns,

`m`	Mean value of all absolute finite values
`n`	Number of finite values

If `x` contains no finite values, the mean returned is 0.

Examples

```
m = meanabs([1 2;3 4])
[m,n] = meanabs({[1 2; NaN 4], [4 5; 2 3]})
```

See Also

`meansqr` | `sumabs` | `sumsqr`

Introduced in R2010b

meansqr

Mean of squared elements of matrix or matrices

Syntax

```
[m,n] = meansqr(x)
```

Description

`[m,n] = meansqr(x)` takes a matrix or cell array of matrices and returns,

`m`	Mean value of all squared finite values
`n`	Number of finite values

If `x` contains no finite values, the mean returned is 0.

Examples

```
m = meansqr([1 2;3 4])
[m,n] = meansqr({[1 2; NaN 4], [4 5; 2 3]})
```

See Also

`meanabs` | `sumabs` | `sumsqr`

Introduced in R2010b

midpoint

Midpoint weight initialization function

Syntax

```
W = midpoint(S,PR)
```

Description

`midpoint` is a weight initialization function that sets weight (row) vectors to the center of the input ranges.

`W = midpoint(S,PR)` takes two arguments,

S	Number of rows (neurons)
PR	R-by-Q matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R matrix with rows set to `(Pmin+Pmax)'/2`.

Examples

Here initial weight values are calculated for a five-neuron layer with input elements ranging over `[0 1]` and `[-2 2]`.

```
W = midpoint(5,[0 1; -2 2])
```

See Also

`initwb` | `initlay` | `init`

Introduced before R2006a

minmax

Ranges of matrix rows

Syntax

```
pr = minmax(P)
```

Description

`pr = minmax(P)` takes one argument,

`P`	`R`-by-`Q` matrix

and returns the `R`-by-`2` matrix `pr` of minimum and maximum values for each row of `P`.

Alternatively, `P` can be an `M`-by-`N` cell array of matrices. Each matrix `P{i,j}` should have `Ri` rows and `Q` columns. In this case, `minmax` returns an `M`-by-`1` cell array where the `m`th element is an `Ri`-by-`2` matrix of the minimum and maximum values of elements for the matrix on the `i`th row of `P`.

Examples

```
x = rands(4,5)
mm = minmax(x)
x = nndata([1;2],3,4)
mm = minmax(x)
```

Introduced before R2006a

mse

Mean squared normalized error performance function

Syntax

```
perf = mse(net,t,y,ew)
```

Description

Tip To use mean squared error with deep learning, use `regressionLayer`, or use the `dlarray` method mse.

`perf = mse(net,t,y,ew)` takes a neural network, `net`, a matrix or cell array of targets, `t`, a matrix or cell array of outputs, `y`, and error weights, `ew`, and returns the mean squared error.

This function has two optional parameters, which are associated with networks whose `net.trainFcn` is set to this function:

- `'regularization'` can be set to any value between 0 and 1. The greater the regularization value, the more squared weights and biases are included in the performance calculation relative to errors. The default is 0, corresponding to no regularization.
- `'normalization'` can be set to `'none'` (the default); `'standard'`, which normalizes errors between -2 and 2, corresponding to normalizing outputs and targets between -1 and 1; and `'percent'`, which normalizes errors between -1 and 1. This feature is useful for networks with multi-element outputs. It ensures that the relative accuracy of output elements with differing target value ranges are treated as equally important, instead of prioritizing the relative accuracy of the output element with the largest target value range.

You can create a standard network that uses `mse` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `mse`, set `net.performFcn` to `'mse'`. This automatically sets `net.performParam` to a structure with the default optional parameter values.

`mse` is a network performance function. It measures the network's performance according to the mean of squared errors.

Examples

Train Neural Network Using `mse` Performance Function

This example shows shows how to train a neural network using the `mse` performance function.

Here a two-layer feedforward network is created and trained to estimate body fat percentage using the `mse` performance function and a regularization value of 0.01.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10);
net.performParam.regularization = 0.01;
```

MSE is the default performance function for `feedforwardnet`.

```
net.performFcn
```

```
ans =
'mse'
```

Train the network and evaluate performance.

```
net = train(net, x, t);
y = net(x);
perf = perform(net, t, y)
```

```
perf = 20.7769
```

Alternatively, you can call `mse` directly.

```
perf = mse(net, t, y, 'regularization', 0.01)
```

```
perf = 20.7769
```

Input Arguments

`net` — Input matrix

matrix

Network you want to calculate the performance of, specified as a `SeriesNetwork` or a `DAGNetwork` object.

`t` — Targets

matrix | cell array

Targets, specified as a matrix or a cell array.

`y` — Outputs

matrix | cell array

Outputs, specified as a matrix or a cell array.

`ew` — Error weights

1 (default) | scalar

Error weights, specified as a scalar.

Output Arguments

`perf` — Network performance

scalar

Performance of the network as the mean squared errors.

See Also

`mae`

Introduced before R2006a

narnet

Nonlinear autoregressive neural network

Syntax

```
narnet(feedbackDelays,hiddenSizes,feedbackMode,trainFcn)
```

Description

`narnet(feedbackDelays,hiddenSizes,feedbackMode,trainFcn)` takes these arguments:

- Row vector of increasing 0 or positive feedback delays, `feedbackDelays`
- Row vector of one or more hidden layer sizes, `hiddenSizes`
- Type of feedback, `feedbackMode`
- Training function, `trainFcn`

and returns a NAR neural network.

You can train NAR (nonlinear autoregressive) neural networks to predict a time series from the past values of that series.

Examples

Train NAR Network and Predict on New Data

Train a nonlinear autoregressive (NAR) neural network and predict on new time series data. Predicting a sequence of values in a time series is also known as *multistep prediction*. Closed-loop networks can perform multistep predictions. When external feedback is missing, closed-loop networks can continue to predict by using internal feedback. In NAR prediction, the future values of a time series are predicted only from past values of that series.

Load the simple time series prediction data.

```
T = simplenar_dataset;
```

Create a NAR network. Define the feedback delays and size of the hidden layers.

```
net = narnet(1:2,10);
```

Prepare the time series data using `preparets`. This function automatically shifts input and target time series by the number of steps needed to fill the initial input and layer delay states.

```
[Xs,Xi,Ai,Ts] = preparets(net,{},{},T);
```

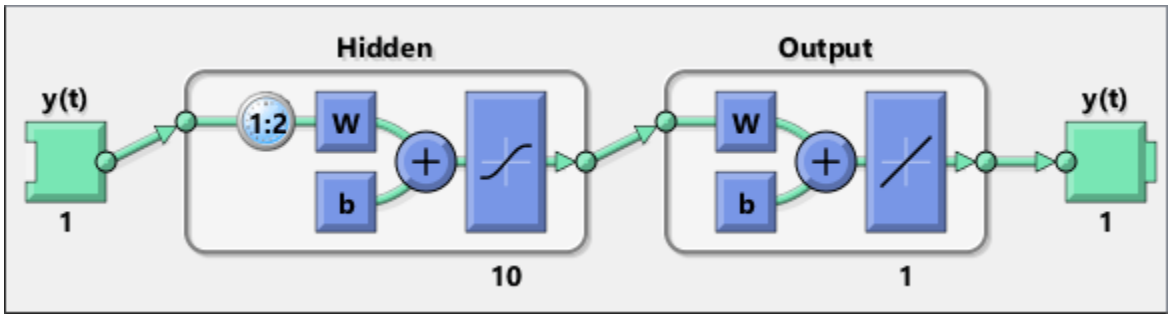
A recommended practice is to fully create the network in an open loop, and then transform the network to a closed loop for multistep-ahead prediction. Then, the closed-loop network can predict as many future values as you want. If you simulate the neural network in closed-loop mode only, the network can perform as many predictions as the number of time steps in the input series.

Train the NAR network. The `train` function trains the network in an open loop (series-parallel architecture), including the validation and testing steps.

```
net = train(net,Xs,Ts,Xi,Ai);
```

Display the trained network.

```
view(net)
```



Calculate the network output `Y`, final input states `Xf`, and final layer states `Af` of the open-loop network from the network input `Xs`, initial input states `Xi`, and initial layer states `Ai`.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
```

Calculate the network performance.

```
perf = perform(net,Ts,Y)
```

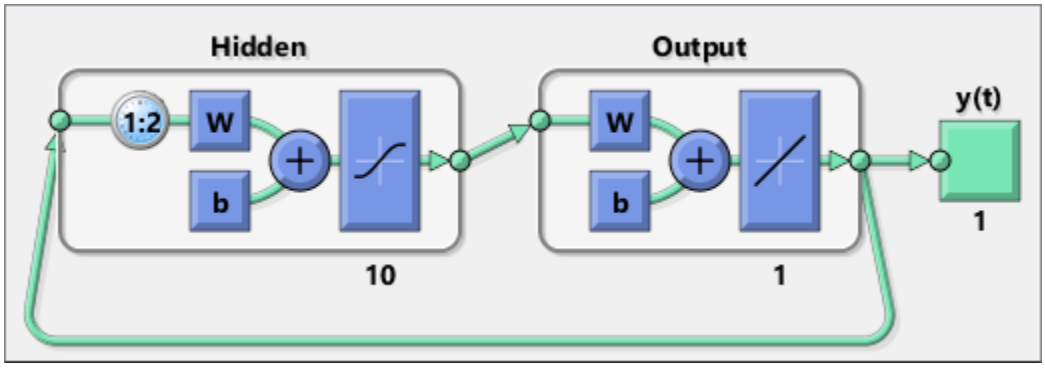
```
perf =

   1.0100e-09
```

To predict the output for the next 20 time steps, first simulate the network in closed-loop mode. The final input states `Xf` and layer states `Af` of the open-loop network `net` become the initial input states `Xic` and layer states `Aic` of the closed-loop network `netc`.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
```

Display the closed-loop network. The network has only one input. In closed-loop mode, this input connects to the output. A direct delayed output connection replaces the delayed target input.

```
view(netc)
```



To simulate the network 20 time steps ahead, input an empty cell array of length 20. The network requires only the initial conditions given in `Xic` and `Aic`.

```
Yc = netc(cell(0,20),Xic,Aic)
```

```
Yc =

  1x20 cell array

  Columns 1 through 5

    {[0.8346]}    {[0.3329]}    {[0.9084]}    {[1.0000]}    {[0.3190]}

  Columns 6 through 10

    {[0.7329]}    {[0.9801]}    {[0.6409]}    {[0.5146]}    {[0.9746]}

  Columns 11 through 15

    {[0.9077]}    {[0.2807]}    {[0.8651]}    {[0.9897]}    {[0.4093]}

  Columns 16 through 20

    {[0.6838]}    {[0.9976]}    {[0.7007]}    {[0.4311]}    {[0.9660]}
```

Input Arguments

`feedbackDelays` — Feedback delays

`[1:2]` (default) | row vector

Zero or positive feedback delays, specified as an increasing row vector.

`hiddenSizes` — Hidden sizes

`10` (default) | row vector

Sizes of the hidden layers, specified as a row vector of one or more elements.

`feedbackMode` — Feedback mode

`'open'` (default) | `'closed'` | `'none'`

Type of feedback, specified as either `'open'`, `'closed'`, or `'none'`.

`trainFcn` — Training function name

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training Function	Algorithm
`'trainlm'`	Levenberg-Marquardt
`'trainbr'`	Bayesian Regularization
`'trainbfg'`	BFGS Quasi-Newton
`'trainrp'`	Resilient Backpropagation

Training Function	Algorithm
`'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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

See Also

`preparets` | `removedelay` | `timedelaynet` | `narxnet` | `closeloop` | `network` | `train` | `openloop`

Topics

“Multistep Neural Network Prediction”

Introduced in R2010b

narxnet

Nonlinear autoregressive neural network with external input

Syntax

```
narxnet(inputDelays,feedbackDelays,hiddenSizes,feedbackMode,trainFcn)
```

Description

`narxnet(inputDelays,feedbackDelays,hiddenSizes,feedbackMode,trainFcn)` takes these arguments:

- Row vector of increasing 0 or positive input delays, `inputDelays`
- Row vector of increasing 0 or positive feedback delays, `feedbackDelays`
- Row vector of one or more hidden layer sizes, `hiddenSizes`
- Type of feedback, `feedbackMode`
- Backpropagation training function, `trainFcn`

and returns a NARX neural network.

NARX (Nonlinear autoregressive with external input) networks can learn to predict one time series given past values of the same time series, the feedback input, and another time series called the external (or exogenous) time series.

Examples

Train NARX Network and Predict on New Data

Train a nonlinear autoregressive with external input (NARX) neural network and predict on new time series data. Predicting a sequence of values in a time series is also known as *multistep prediction*. Closed-loop networks can perform multistep predictions. When external feedback is missing, closed-loop networks can continue to predict by using internal feedback. In NARX prediction, the future values of a time series are predicted from past values of that series, the feedback input, and an external time series.

Load the simple time series prediction data.

```
[X,T] = simpleseries_dataset;
```

Partition the data into training data `XTrain` and `TTrain`, and data for prediction `XPredict`. Use `XPredict` to perform prediction after you create the closed-loop network.

```
XTrain = X(1:80);
TTrain = T(1:80);
XPredict = X(81:100);
```

Create a NARX network. Define the input delays, feedback delays, and size of the hidden layers.

```
net = narxnet(1:2,1:2,10);
```

Prepare the time series data using `preparets`. This function automatically shifts input and target time series by the number of steps needed to fill the initial input and layer delay states.

```
[Xs,Xi,Ai,Ts] = preparets(net,XTrain,{},TTrain);
```

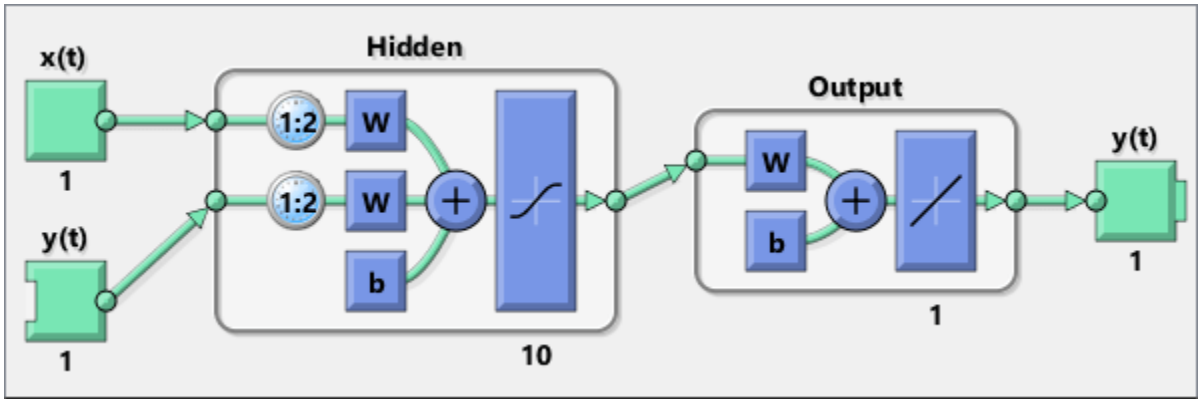
A recommended practice is to fully create the network in an open loop, and then transform the network to a closed loop for multistep-ahead prediction. Then, the closed-loop network can predict as many future values as you want. If you simulate the neural network in closed-loop mode only, the network can perform as many predictions as the number of time steps in the input series.

Train the NARX network. The `train` function trains the network in an open loop (series-parallel architecture), including the validation and testing steps.

```
net = train(net,Xs,Ts,Xi,Ai);
```

Display the trained network.

```
view(net)
```



Calculate the network output `Y`, final input states `Xf`, and final layer states `Af` of the open-loop network from the network input `Xs`, initial input states `Xi`, and initial layer states `Ai`.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
```

Calculate the network performance.

```
perf = perform(net,Ts,Y)
```

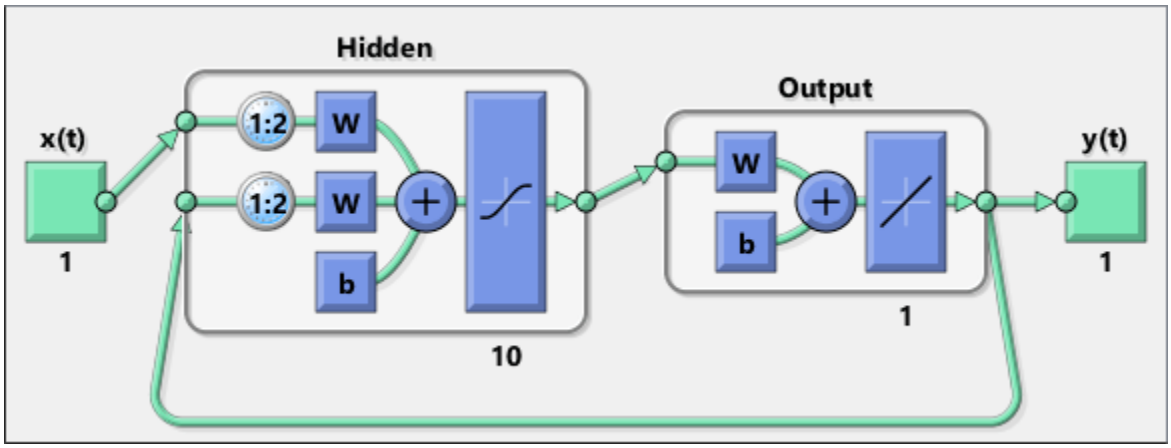
```
perf =

    0.0153
```

To predict the output for the next 20 time steps, first simulate the network in closed-loop mode. The final input states `Xf` and layer states `Af` of the open-loop network net become the initial input states `Xic` and layer states `Aic` of the closed-loop network `netc`.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
```

Display the closed-loop network.

```
view(netc)
```



Run the prediction for 20 time steps ahead in closed-loop mode.

```
Yc = netc(XPredict,Xic,Aic)
```

```
Yc =

  1x20 cell array

  Columns 1 through 5

    {[-0.0156]}    {[0.1133]}    {[-0.1472]}    {[-0.0706]}    {[0.0355]}

  Columns 6 through 10

    {[-0.2829]}    {[0.2047]}    {[-0.3809]}    {[-0.2836]}    {[0.1886]}

  Columns 11 through 15

    {[-0.1813]}    {[0.1373]}    {[0.2189]}    {[0.3122]}    {[0.2346]}

  Columns 16 through 20

    {[-0.0156]}    {[0.0724]}    {[0.3395]}    {[0.1940]}    {[0.0757]}
```

Input Arguments

`inputDelays` — Input delays
`[1:2]` (default) | row vector

Zero or positive input delays, specified as an increasing row vector.

`feedbackDelays` — Feedback delays
`[1:2]` (default) | row vector

Zero or positive feedback delays, specified as an increasing row vector.

`hiddenSizes` — Hidden sizes
`10` (default) | row vector

Sizes of the hidden layers, specified as a row vector of one or more elements.

`feedbackMode` — Feedback mode
`'open'` (default) | `'closed'` | `'none'`

Type of feedback, specified as either `'open'`, `'closed'`, or `'none'`.

`trainFcn` — Training function name
`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

See Also

`closeloop` | `narnet` | `openloop` | `preparets` | `removedelay` | `timedelaynet` | `network` | `train`

Topics
“Multistep Neural Network Prediction”
“Design Time Series NARX Feedback Neural Networks”

Introduced in R2010b

nctool

Neural network classification or clustering tool

Syntax

```
nctool
```

Description

`nctool` opens the **Neural Net Clustering** app.

For more information and an example of its usage, see “Cluster Data with a Self-Organizing Map”.

Algorithms

`nctool` leads you through solving a clustering problem using a self-organizing map. The map forms a compressed representation of the inputs space, reflecting both the relative density of input vectors in that space, and a two-dimensional compressed representation of the input-space topology.

See Also

Neural Net Clustering | `nftool` | `nprtool` | `ntstool`

Introduced in R2008a

negdist

Negative distance weight function

Syntax

```
Z = negdist(W,P)
dim = negdist('size',S,R,FP)
dw = negdist('dz_dw',W,P,Z,FP)
```

Description

`negdist` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = negdist(W,P)` takes these inputs,

`W`	`S`-by-`R` weight matrix
`P`	`R`-by-`Q` matrix of `Q` input (column) vectors
`FP`	Row cell array of function parameters (optional, ignored)

and returns the `S`-by-`Q` matrix of negative vector distances.

`dim = negdist('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S`-by-`R`].

`dw = negdist('dz_dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = negdist(W,P)
```

Network Use

You can create a standard network that uses `negdist` by calling `competlayer` or `selforgmap`.

To change a network so an input weight uses `negdist`, set `net.inputWeights{i,j}.weightFcn` to `'negdist'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'negdist'`.

In either case, call `sim` to simulate the network with `negdist`.

Algorithms

`negdist` returns the negative Euclidean distance:

```
z = -sqrt(sum(w-p)^2)
```

See Also

`competlayer` | `dist` | `dotprod` | `selforgmap` | `sim`

Introduced before R2006a

netinv

Inverse transfer function

Syntax

```
A = netinv(N,FP)
```

Description

`netinv` is a transfer function. Transfer functions calculate a layer's output from its net input.

`A = netinv(N,FP)` takes inputs

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `1/N`.

`info = netinv('code')` returns information about this function. The following codes are supported:

`netinv('name')` returns the name of this function.

`netinv('output',FP)` returns the [`min max`] output range.

`netinv('active',FP)` returns the [`min max`] active input range.

`netinv('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`netinv('fpnames')` returns the names of the function parameters.

`netinv('fpdefaults')` returns the default function parameters.

Examples

Here you define 10 five-element net input vectors `N` and calculate `A`.

```
n = rand(5,10);
a = netinv(n);
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'netinv';
```

See Also

`tansig` | `logsig`

Introduced in R2006a

netprod

Product net input function

Syntax

```
N = netprod({Z1,Z2,...,Zn})
info = netprod('code')
```

Description

`netprod` is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

`N = netprod({Z1,Z2,...,Zn})` takes

`Zi`	`S`-by-`Q` matrices in a row cell array

and returns an element-wise product of `Z1` to `Zn`.

`info = netprod('code')` returns information about this function. The following codes are supported:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full `N`-by-`S`-by-`Q` derivative = 1, element-wise `S`-by-`Q` derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

Examples

Here `netprod` combines two sets of weighted input vectors (user-defined).

```
Z1 = [1 2 4;3 4 1];
Z2 = [-1 2 2; -5 -6 1];
Z = {Z1,Z2};
N = netprod({Z})
```

Here `netprod` combines the same weighted inputs with a bias vector. Because `Z1` and `Z2` each contain three concurrent vectors, three concurrent copies of `B` must be created with `concur` so that all sizes match.

```
B = [0; -1];
Z = {Z1, Z2, concur(B,3)};
N = netprod(Z)
```

Network Use

You can create a standard network that uses `netprod` by calling `newpnn` or `newgrnn`.

To change a network so that a layer uses `netprod`, set `net.layers{i}.netInputFcn` to `'netprod'`.

In either case, call `sim` to simulate the network with `netprod`. See `newpnn` or `newgrnn` for simulation examples.

See Also

`sim` | `netsum` | `concur`

Introduced before R2006a

netsum

Sum net input function

Syntax

```
N = netsum({Z1,Z2,...,Zn},FP)
info = netsum('code')
```

Description

`netsum` is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

`N = netsum({Z1,Z2,...,Zn},FP)` takes `Z1` to `Zn` and optional function parameters,

`Zi`	`S`-by-`Q` matrices in a row cell array
`FP`	Row cell array of function parameters (ignored)

and returns the elementwise sum of `Z1` to `Zn`.

`info = netsum('code')` returns information about this function. The following codes are supported:

`netsum('name')` returns the name of this function.

`netsum('type')` returns the type of this function.

`netsum('fpnames')` returns the names of the function parameters.

`netsum('fpdefaults')` returns default function parameter values.

`netsum('fpcheck', FP)` throws an error for illegal function parameters.

`netsum('fullderiv')` returns 0 or 1, depending on whether the derivative is `S`-by-`Q` or `N`-by-`S`-by-`Q`.

Examples

Here `netsum` combines two sets of weighted input vectors and a bias. You must use `concur` to make `b` the same dimensions as `z1` and `z2`.

```
z1 = [1, 2, 4; 3, 4, 1]
z2 = [-1, 2, 2; -5, -6, 1]
b = [0; -1]
n = netsum({z1, z2, concur(b, 3)})
```

Assign this net input function to the first layer of a network.

```
net = feedforwardnet();
net.layers{1}.netInputFcn = 'netsum';
```

See Also

`cascadeforwardnet` | `feedforwardnet` | `netprod` | `netinv`

Introduced before R2006a

network

Create custom shallow neural network

Syntax

```
net = network
net =
network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,outputConne
ct)
```

To Get Help

Type `help network/network`.

Tip To learn how to create a deep learning network, see “Specify Layers of Convolutional Neural Network”.

Description

`network` creates new custom networks. It is used to create networks that are then customized by functions such as `feedforwardnet` and `narxnet`.

`net = network` without arguments returns a new neural network with no inputs, layers or outputs.

`net =
network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,outputConne
ct)` takes these optional arguments (shown with default values):

`numInputs`	Number of inputs, 0
`numLayers`	Number of layers, 0
`biasConnect`	`numLayers`-by-1 Boolean vector, zeros
`inputConnect`	`numLayers`-by-`numInputs` Boolean matrix, zeros
`layerConnect`	`numLayers`-by-`numLayers` Boolean matrix, zeros
`outputConnect`	1-by-`numLayers` Boolean vector, zeros

and returns

`net`	New network with the given property values

Properties

Architecture Properties

`net.numInputs`	0 or a positive integer	Number of inputs.

`net.numLayers`	0 or a positive integer	Number of layers.
`net.biasConnect`	`numLayer`-by-1 Boolean vector	If `net.biasConnect(i)` is 1, then layer `i` has a bias, and `net.biases{i}` is a structure describing that bias.
`net.inputConnect`	`numLayer`-by-`numInputs` Boolean vector	If `net.inputConnect(i,j)` is 1, then layer `i` has a weight coming from input `j`, and `net.inputWeights{i,j}` is a structure describing that weight.
`net.layerConnect`	`numLayer`-by-`numLayers` Boolean vector	If `net.layerConnect(i,j)` is 1, then layer `i` has a weight coming from layer `j`, and `net.layerWeights{i,j}` is a structure describing that weight.
`net.outputConnect`	1-by-`numLayers` Boolean vector	If `net.outputConnect(i)` is 1, then the network has an output from layer `i`, and `net.outputs{i}` is a structure describing that output.
`net.numOutputs`	0 or a positive integer (read only)	Number of network outputs according to `net.outputConnect`.
`net.numInputDelays`	0 or a positive integer (read only)	Maximum input delay according to all `net.inputWeights{i,j}.delays`.
`net.numLayerDelays`	0 or a positive number (read only)	Maximum layer delay according to all `net.layerWeights{i,j}.delays`.

Subobject Structure Properties

`net.inputs`	`numInputs`-by-1 cell array	`net.inputs{i}` is a structure defining input `i`.
`net.layers`	`numLayers`-by-1 cell array	`net.layers{i}` is a structure defining layer `i`.
`net.biases`	`numLayers`-by-1 cell array	If `net.biasConnect(i)` is 1, then `net.biases{i}` is a structure defining the bias for layer `i`.
`net.inputWeights`	`numLayers`-by-`numInputs` cell array	If `net.inputConnect(i,j)` is 1, then `net.inputWeights{i,j}` is a structure defining the weight to layer `i` from input `j`.
`net.layerWeights`	`numLayers`-by-`numLayers` cell array	If `net.layerConnect(i,j)` is 1, then `net.layerWeights{i,j}` is a structure defining the weight to layer `i` from layer `j`.
`net.outputs`	1-by-`numLayers` cell array	If `net.outputConnect(i)` is 1, then `net.outputs{i}` is a structure defining the network output from layer `i`.

Function Properties

`net.adaptFcn`	Name of a network adaption function or `''`
`net.initFcn`	Name of a network initialization function or `''`

net.performFcn	Name of a network performance function or ''
net.trainFcn	Name of a network training function or ''

Parameter Properties

net.adaptParam	Network adaption parameters
net.initParam	Network initialization parameters
net.performParam	Network performance parameters
net.trainParam	Network training parameters

Weight and Bias Value Properties

net.IW	numLayers-by-numInputs cell array of input weight values
net.LW	numLayers-by-numLayers cell array of layer weight values
net.b	numLayers-by-1 cell array of bias values

Other Properties

net.userdata	Structure you can use to store useful values

Examples

Create Network with One Input and Two Layers

This example shows how to create a network without any inputs and layers, and then set its numbers of inputs and layers to 1 and 2 respectively.

```
net = network
net.numInputs = 1
net.numLayers = 2
```

Alternatively, you can create the same network with one line of code.

```
net = network(1,2)
```

Create Feedforward Network and View Properties

This example shows how to create a one-input, two-layer, feedforward network. Only the first layer has a bias. An input weight connects to layer 1 from input 1. A layer weight connects to layer 2 from layer 1. Layer 2 is a network output and has a target.

```
net = network(1,2,[1;0],[1; 0],[0 0; 1 0],[0 1])
```

You can view the network subobjects with the following code.

```
net.inputs{1}
net.layers{1}, net.layers{2}
net.biases{1}
net.inputWeights{1,1}, net.layerWeights{2,1}
net.outputs{2}
```

You can alter the properties of any of the network subobjects. This code changes the transfer functions of both layers:

```
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'logsig';
```

You can view the weights for the connection from the first input to the first layer as follows. The weights for a connection from an input to a layer are stored in `net.IW`. If the values are not yet set, these result is empty.

```
net.IW{1,1}
```

You can view the weights for the connection from the first layer to the second layer as follows. Weights for a connection from a layer to a layer are stored in `net.LW`. Again, if the values are not yet set, the result is empty.

```
net.LW{2,1}
```

You can view the bias values for the first layer as follows.

```
net.b{1}
```

To change the number of elements in input 1 to 2, set each element's range:

```
net.inputs{1}.range = [0 1; -1 1];
```

To simulate the network for a two-element input vector, the code might look like this:

```
p = [0.5; -0.1];
y = sim(net,p)
```

See Also

`sim`

Topics

"Neural Network Object Properties"
"Neural Network Subobject Properties"

Introduced before R2006a

newgrnn

Design generalized regression neural network

Syntax

```
net = newgrnn(P,T,spread)
```

Description

Generalized regression neural networks (grnns) are a kind of radial basis network that is often used for function approximation. grnns can be designed very quickly.

`net = newgrnn(P,T,spread)` takes three inputs,

P	R-by-Q matrix of Q input vectors
T	S-by-Q matrix of Q target class vectors
spread	Spread of radial basis functions (default = 1.0)

and returns a new generalized regression neural network.

The larger the `spread`, the smoother the function approximation. To fit data very closely, use a `spread` smaller than the typical distance between input vectors. To fit the data more smoothly, use a larger `spread`.

Properties

`newgrnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates weighted inputs with `dist` and net input with `netprod`. The second layer has `purelin` neurons, calculates weighted input with `normprod`, and net inputs with `netsum`. Only the first layer has biases.

`newgrnn` sets the first layer weights to `P'`, and the first layer biases are all set to `0.8326/spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/– `spread`. The second layer weights `W2` are set to `T`.

Examples

Here you design a radial basis network, given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newgrnn(P,T);
```

The network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York, Van Nostrand Reinhold, 1993, pp. 155–61

See Also

`sim` | `newrb` | `newrbe` | `newpnn`

Introduced before R2006a

newlind

Design linear layer

Syntax

```
net = newlind(P,T,Pi)
```

Description

`net = newlind(P,T,Pi)` takes these input arguments,

`P`	`R`-by-`Q` matrix of `Q` input vectors
`T`	`S`-by-`Q` matrix of `Q` target class vectors
`Pi`	`1`-by-`ID` cell array of initial input delay states

where each element `Pi{i,k}` is an `Ri`-by-`Q` matrix, and the default = `[]`; and returns a linear layer designed to output `T` (with minimum sum square error) given input `P`.

`newlind(P,T,Pi)` can also solve for linear networks with input delays and multiple inputs and layers by supplying input and target data in cell array form:

`P`	`Ni`-by-`TS` cell array	Each element `P{i,ts}` is an `Ri`-by-`Q` input matrix
`T`	`Nt`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix
`Pi`	`Ni`-by-`ID` cell array	Each element `Pi{i,k}` is an `Ri`-by-`Q` matrix, default = `[]`

and returns a linear network with `ID` input delays, `Ni` network inputs, and `Nl` layers, designed to output `T` (with minimum sum square error) given input `P`.

Examples

You want a linear layer that outputs `T` given `P` for the following definitions:

```
P = [1 2 3];
T = [2.0 4.1 5.9];
```

Use `newlind` to design such a network and check its response.

```
net = newlind(P,T);
Y = sim(net,P)
```

You want another 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 = {5.0 6.1 4.0 6.0 6.9 8.0};
net = newlind(P,T,Pi);
Y = sim(net,P,Pi)
```

You want a linear network with two outputs `Y1` and `Y2` that generate sequences `T1` and `T2`, given the sequences `P1` and `P2`, with three initial input delay states `Pi1` for input 1 and three initial delays states `Pi2` for input 2.

```
P1 = {1 2 1 3 3 2}; Pi1 = {1 3 0};
P2 = {1 2 1 1 2 1}; Pi2 = {2 1 2};
T1 = {5.0 6.1 4.0 6.0 6.9 8.0};
T2 = {11.0 12.1 10.1 10.9 13.0 13.0};
net = newlind([P1; P2],[T1; T2],[Pi1; Pi2]);
Y = sim(net,[P1; P2],[Pi1; Pi2]);
Y1 = Y(1,:)
Y2 = Y(2,:)
```

Algorithms

`newlind` calculates weight `W` and bias `B` values for a linear layer from inputs `P` and targets `T` by solving this linear equation in the least squares sense:

```
[W b] * [P; ones] = T
```

See Also

`sim`

Introduced before R2006a

newpnn

Design probabilistic neural network

Syntax

```
net = newpnn(P,T,spread)
```

Description

Probabilistic neural networks (PNN) are a kind of radial basis network suitable for classification problems.

`net = newpnn(P,T,spread)` takes two or three arguments,

`P`	R-by-`Q` matrix of `Q` input vectors
`T`	S-by-`Q` matrix of `Q` target class vectors
`spread`	Spread of radial basis functions (default = 0.1)

and returns a new probabilistic neural network.

If `spread` is near zero, the network acts as a nearest neighbor classifier. As `spread` becomes larger, the designed network takes into account several nearby design vectors.

Examples

Here a classification problem is defined with a set of inputs `P` and class indices `Tc`.

```
P = [1 2 3 4 5 6 7];
Tc = [1 2 3 2 2 3 1];
```

The class indices are converted to target vectors, and a PNN is designed and tested.

```
T = ind2vec(Tc)
net = newpnn(P,T);
Y = sim(net,P)
Yc = vec2ind(Y)
```

Algorithms

`newpnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist` and its net input with `netprod`. The second layer has `compet` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Only the first layer has biases.

`newpnn` sets the first-layer weights to `P'`, and the first-layer biases are all set to `0.8326/spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/– `spread`. The second-layer weights `W2` are set to `T`.

References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York, Van Nostrand Reinhold, 1993, pp. 35–55

See Also

`sim` | `ind2vec` | `vec2ind` | `newrb` | `newrbe` | `newgrnn`

Introduced before R2006a

newrb

Design radial basis network

Syntax

```
net = newrb(P,T,goal,spread,MN,DF)
```

Description

`net = newrb(P,T,goal,spread,MN,DF)` takes two of these arguments:

- `P` — R-by-Q matrix of Q input vectors
- `T` — S-by-Q matrix of Q target class vectors
- `goal` — Mean squared error goal
- `spread` — Spread of radial basis functions
- `MN` — Maximum number of neurons
- `DF` — Number of neurons to add between displays

Radial basis networks can be used to approximate functions. `newrb` adds neurons to the hidden layer of a radial basis network until it meets the specified mean squared error goal.

The larger `spread` is, the smoother the function approximation. Too large a spread means a lot of neurons are required to fit a fast-changing function. Too small a spread means many neurons are required to fit a smooth function, and the network might not generalize well. Call `newrb` with different spreads to find the best value for a given problem.

Examples

Design a Radial Basis Network

This example shows how to design a radial basis network.

Design a radial basis network with inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);
```

Simulate the network for a new input.

```
P = 1.5;
Y = sim(net,P)
```

Input Arguments

P — Input matrix

matrix

Input vectors, specified as an R-by-Q matrix.

T — Target class matrix
matrix

Target class vectors, specified as an S-by-Q matrix.

goal — Error goal
0.0 (default) | scalar

Mean squared error goal, specified as a scalar.

spread — Spread of basis functions
1 (default) | scalar

Spread of radial basis functions, specified as a scalar.

MN — Neurons maximum number
Q (default) | scalar

Maximum number of neurons, specified as a scalar.

DF — Neurons between displays
25 (default) | scalar

Number of neurons to add between displays, specified as a scalar.

Output Arguments

net — Radial basis network
network

New radial basis network, returned as a network object

Algorithms

newrb creates a two-layer network. The first layer has radbas neurons, and calculates its weighted inputs with dist and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

Initially the radbas layer has no neurons. The following steps are repeated until the network's mean squared error falls below goal.

1. The network is simulated.
2. The input vector with the greatest error is found.
3. A radbas neuron is added with weights equal to that vector.
4. The purelin layer weights are redesigned to minimize error.

See Also
sim | newrbe | newgrnn | newpnn

Introduced before R2006a

newrbe

Design exact radial basis network

Syntax

```
net = newrbe(P,T,spread)
```

Description

Radial basis networks can be used to approximate functions. `newrbe` very quickly designs a radial basis network with zero error on the design vectors.

`net = newrbe(P,T,spread)` takes two or three arguments,

`P`	RxQ matrix of Q R-element input vectors
`T`	SxQ matrix of Q S-element target class vectors
`spread`	Spread of radial basis functions (default = 1.0)

and returns a new exact radial basis network.

The larger the `spread` is, the smoother the function approximation will be. Too large a spread can cause numerical problems.

Examples

Here you design a radial basis network given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrbe(P,T);
```

The network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

Algorithms

`newrbe` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist` and its net input with `netprod`. The second layer has `purelin` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Both layers have biases.

`newrbe` sets the first-layer weights to `P'`, and the first-layer biases are all set to `0.8326/spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/– `spread`.

The second-layer weights `IW{2,1}` and biases `b{2}` are found by simulating the first-layer outputs `A{1}` and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

See Also

`sim` | `newrb` | `newgrnn` | `newpnn`

Introduced before R2006a

nftool

Neural net fitting tool

Syntax

```
nftool
```

Description

`nftool` opens the **Neural Net Fitting** app.

For more information and an example of its usage, see “Fit Data with a Shallow Neural Network”.

Algorithms

`nftool` leads you through solving a data fitting problem, solving it with a two-layer feed-forward network trained with Levenberg-Marquardt.

See Also

Neural Net Fitting | `nctool` | `nprtool` | `ntstool`

Introduced in R2006a

nncell2mat

Combine neural network cell data into matrix

Syntax

```
[y,i,j] nncell2mat(x)
```

Description

`[y,i,j] nncell2mat(x)` takes a cell array of matrices and returns,

`y`	Cell array formed by concatenating matrices
`i`	Array of row sizes
`ji`	Array of column sizes

The row and column sizes returned by `nncell2mat` can be used to convert the returned matrix back into a cell of matrices with `mat2cell`.

Examples

Here neural network data is converted to a matrix and back.

```
c = {rands(2,3) rands(2,3); rands(5,3) rands(5,3)};
[m,i,j] = nncell2mat(c)
c3 = mat2cell(m,i,j)
```

See Also

`nndata` | `nnsize`

Introduced in R2010b

nncorr

Crross correlation between neural network time series

Syntax

```
nncorr(a,b,maxlag,'flag')
```

Description

`nncorr(a,b,maxlag,'flag')` takes these arguments,

`a`	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of `N`.
`b`	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of `M`.
`maxlag`	Maximum number of time lags
`flag`	Type of normalization (default = `'none'`)

and returns an `N`-by-`M` cell array where each `{i,j}` element is a `2*maxlag+1` length row vector formed from the correlations of `a` elements (i.e., matrix row) `i` and `b` elements (i.e., matrix column) `j`.

If `a` and `b` are specified with row vectors, the result is returned in matrix form.

The options for the normalization *`flag`* are:

- `'biased'` — scales the raw cross-correlation by 1/N.
- `'unbiased'` — scales the raw correlation by `1/(N-abs(k))`, where `k` is the index into the result.
- `'coeff'` — normalizes the sequence so that the correlations at zero lag are 1.0.
- `'none'` — no scaling. This is the default.

Examples

Here the autocorrelation of a random 1-element, 1-sample, 20-timestep signal is calculated with a maximum lag of 10.

```
a = nndata(1,1,20)
aa = nncorr(a,a,10)
```

Here the cross-correlation of the first signal with another random 2-element signal are found, with a maximum lag of 8.

```
b = nndata(2,1,20)
ab = nncorr(a,b,8)
```

See Also

`confusion` | `regression`

Introduced in R2010b

nndata

Create neural network data

Syntax

```
nndata(N,Q,TS,v)
```

Description

`nndata(N,Q,TS,v)` takes these arguments,

`N`	Vector of `M` element sizes
`Q`	Number of samples
`TS`	Number of timesteps
`v`	Scalar value

and returns an `M`-by-`TS` cell array where each row `i` has `N(i)`-by-`Q` sized matrices of value `v`. If `v` is not specified, random values are returned.

You can access subsets of neural network data with `getelements`, `getsamples`, `gettimesteps`, and `getsignals`.

You can set subsets of neural network data with `setelements`, `setsamples`, `settimesteps`, and `setsignals`.

You can concatenate subsets of neural network data with `catelements`, `catsamples`, `cattimesteps`, and `catsignals`.

Examples

Here four samples of five timesteps, for a 2-element signal consisting of zero values is created:

```
x = nndata(2,4,5,0)
```

To create random data with the same dimensions:

```
x = nndata(2,4,5)
```

Here static (1 timestep) data of 12 samples of 4 elements is created.

```
x = nndata(4,12)
```

See Also

`nnsize` | `tonndata` | `fromnndata` | `nndata2sim` | `sim2nndata`

Introduced in R2010b

nndata2gpu

Format neural data for efficient GPU training or simulation

Syntax

```
nndata2gpu(x)
[Y,Q,N,TS] = nndata2gpu(X)
nndata2gpu(X,PRECISION)
```

Description

`nndata2gpu` requires Parallel Computing Toolbox.

`nndata2gpu(x)` takes an `N`-by-`Q` matrix `X` of `Q` `N`-element column vectors, and returns it in a form for neural network training and simulation on the current GPU device.

The `N`-by-`Q` matrix becomes a `QQ`-by-`N` gpuArray where `QQ` is `Q` rounded up to the next multiple of 32. The extra rows `(Q+1):QQ` are filled with `NaN` values. The gpuArray has the same precision (`'single'` or `'double'`) as `X`.

`[Y,Q,N,TS] = nndata2gpu(X)` can also take an `M`-by-`TS` cell array of `M` signals over `TS` time steps. Each element of `X{i,ts}` should be an `Ni`-by-`Q` matrix of `Q` `Ni`-element vectors, representing the `i`th signal vector at time step `ts`, across all `Q` time series. In this case, the gpuArray `Y` returned is `QQ`-by-`(sum(Ni)*TS)`. Dimensions `Ni`, `Q`, and `TS` are also returned so they can be used with `gpu2nndata` to perform the reverse formatting.

`nndata2gpu(X,PRECISION)` specifies the default precision of the gpuArray, which can be `'double'` or `'single'`.

Examples

Copy a matrix to the GPU and back:

```
x = rand(5,6)
[y,q] = nndata2gpu(x)
x2 = gpu2nndata(y,q)
```

Copy neural network cell array data, representing four time series, each consisting of five time steps of 2-element and 3-element signals:

```
x = nndata([2;3],4,5)
[y,q,n,ts] = nndata2gpu(x)
x2 = gpu2nndata(y,q,n,ts)
```

See Also

`gpu2nndata`

Introduced in R2012b

nndata2sim

Convert neural network data to Simulink time series

Syntax

```
nndata2sim(x,i,q)
```

Description

`nndata2sim(x,i,q)` takes these arguments,

`x`	Neural network data
`i`	Index of signal (default = 1)
`q`	Index of sample (default = 1)

and returns time series `q` of signal `i` as a Simulink time series structure.

Examples

Here random neural network data is created with two signals having 4 and 3 elements respectively, over 10 timesteps. Three such series are created.

```
x = nndata([4;3],3,10);
```

Now the second signal of the first series is converted to Simulink form.

```
y_2_1 = nndata2sim(x,2,1)
```

See Also

`nndata` | `sim2nndata` | `nnsize`

Introduced in R2010b

nnsize

Number of neural data elements, samples, timesteps, and signals

Syntax

```
[N,Q,TS,M] = nnsize(X)
```

Description

`[N,Q,TS,M] = nnsize(X)` takes neural network data `x` and returns,

`N`	Vector containing the number of element sizes for each of `M` signals
`Q`	Number of samples
`TS`	Number of timesteps
`M`	Number of signals

If `X` is a matrix, `N` is the number of rows of `X`, `Q` is the number of columns, and both `TS` and `M` are 1.

If `X` is a cell array, `N` is an `Sx1` vector, where `M` is the number of rows in `X`, and `N(i)` is the number of rows in `X{i,1}`. `Q` is the number of columns in the matrices in `X`.

Examples

This code gets the dimensions of matrix data:

```
x = [1 2 3; 4 7 4]
[n,q,ts,s] = nnsize(x)
```

This code gets the dimensions of cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
[n,q,ts,s] = nnsize(x)
```

See Also

`nndata` | `numelements` | `numsamples` | `numsignals` | `numtimesteps`

Introduced in R2010b

nnstart

Neural network getting started GUI

Syntax

```
nnstart
```

Description

`nnstart` opens a window with launch buttons for **Neural Net Fitting** app, **Neural Net Pattern Recognition** app, **Neural Net Clustering** app, and **Neural Net Time Series** app. It also provides links to lists of data sets, examples, and other useful information for getting started.

See Also

`nctool` | `nftool` | `nprtool` | `ntstool`

Introduced in R2010b

nntool

(To be removed) Open Network/Data Manager

Note `nntool` will be removed in a future release. Use `nnstart` instead. For more information, see "Compatibility Considerations".

Syntax

```
nntool
```

Description

`nntool` opens the Network/Data Manager window, which allows you to import, create, use, and export neural networks and data.

Compatibility Considerations

`nntool` will be removed

Warns starting in R2020b

`nntool` will be removed in a future release. Use `nnstart` instead. `nnstart` provides graphical interfaces that allow you to design and deploy fitting, pattern recognition, clustering, and time-series neural networks.

See Also

`nnstart`

Introduced before R2006a

nntraintool

(To be removed) Neural network training tool

Note `nntraintool` will be removed in a future release. To train a network and open the training window, use `train` instead.

Syntax

```
nntraintool
nntraintool close
nntraintool('close')
```

Description

`nntraintool` opens the neural network training GUI.

This function can be called to make the training GUI visible before training has occurred, after training if the window has been closed, or just to bring the training GUI to the front.

To access additional useful plots, related to the current or last network trained, during or after training, click their respective buttons in the training window.

`nntraintool close` or `nntraintool('close')` closes the training window.

Compatibility Considerations

`nntraintool` will be removed

Warns starting in R2021b

`nntraintool` will be removed in a future release. To train a network and open the training window, use `train` instead.

See Also

`train`

Introduced in R2008a

noloop

Remove neural network open- and closed-loop feedback

Syntax

```
net = noloop(net)
```

Description

`net = noloop(net)` takes a neural network and returns the network with open- and closed-loop feedback removed.

For outputs `i`, where `net.outputs{i}.feedbackMode` is `'open'`, the feedback mode is set to `'none'`, `outputs{i}.feedbackInput` is set to the empty matrix, and the associated network input is deleted.

For outputs `i`, where `net.outputs{i}.feedbackMode` is `'closed'`, the feedback mode is set to `'none'`.

Examples

Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai)
```

Now the network is converted to no loop form. The output and second input are no longer associated.

```
net = noloop(net);
view(net)
[Xs,Xi,Ai] = preparets(net,X,T);
Y = net(Xs,Xi,Ai)
```

See Also

`closeloop` | `openloop`

Introduced in R2010b

normc

Normalize columns of matrix

Syntax

```
normc(M)
```

Description

`normc(M)` normalizes the columns of `M` to a length of 1.

Examples

Normalize the Columns of a Matrix by Using the `normc` Function

This example shows how to use the `normc` function to normalize the columns of a matrix to a length of 1.

Create the matrix, `m`, of which you want to normalize the columns. Then call the `normc` function on this matrix.

```
m = [1 2; 3 4];
normc(m)
```

```
ans =
     0.3162     0.4472
     0.9487     0.8944
```

Input Arguments

`M` — Input matrix

matrix | cell array of matrices

Matrix of which you want to normalize the columns to a length of 1, specified as a matrix or a cell array of matrices.

See Also

`normr`

Introduced before R2006a

normprod

Normalized dot product weight function

Syntax

```
Z = normprod(W,P,FP)
dim = normprod('size',S,R,FP)
dw = normprod('dz_dw',W,P,Z,FP)
```

Description

`normprod` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = normprod(W,P,FP)` takes these inputs,

`W`	`S`-by-`R` weight matrix
`P`	`R`-by-`Q` matrix of `Q` input (column) vectors
`FP`	Row cell array of function parameters (optional, ignored)

and returns the `S`-by-`Q` matrix of normalized dot products.

`dim = normprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S`-by-`R`].

`dw = normprod('dz_dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = normprod(W,P)
```

Network Use

You can create a standard network that uses `normprod` by calling `newgrnn`.

To change a network so an input weight uses `normprod`, set `net.inputWeights{i,j}.weightFcn` to `'normprod'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'normprod'`.

In either case, call `sim` to simulate the network with `normprod`. See `newgrnn` for simulation examples.

Algorithms

`normprod` returns the dot product normalized by the sum of the input vector elements.

```
z = w*p/sum(p)
```

See Also

dotprod

Introduced before R2006a

normr

Normalize rows of matrix

Syntax

```
normalized_M = normr(M)
```

Description

`normalized_M = normr(M)` takes a single matrix or cell array of matrices, `M`, and returns the matrices with rows normalized to a length of one.

Examples

Normalize rows of a matrix

This example shows how to use the `normr` function to normalize the rows of a matrix.

Create a `2x2` matrix and call the `normr` function to normalize its rows to a length of `1`.

```
m = [1 2; 3 4];
normr(m)
ans =
      0.4472     0.8944
      0.6000     0.8000

ans =
      0.4472     0.8944
      0.6000     0.8000
```

Input Arguments

`M` — Matrix to normalize

matrix | cell array of matrices

Matrix to normalize, specified as a matrix or a cell array of matrices.

Output Arguments

`normalized_M` — Normalized matrix

matrix | cell array of matrices

Normalized matrix, returned as a matrix or a cell array of matrices.

See Also

`normc`

Introduced before R2006a

nprtool

Neural Net Pattern Recognition tool

Syntax

```
nprtool
```

Description

`nprtool` opens the **Neural Net Pattern Recognition** app.

For more information and an example of its usage, see “Classify Patterns with a Shallow Neural Network”.

Algorithms

`nprtool` leads you through solving a pattern-recognition classification problem using a two-layer feed-forward `patternnet` network with sigmoid output neurons.

See Also

Neural Net Pattern Recognition | `nctool` | `nftool` | `ntstool`

Introduced in R2008a

ntstool

Neural network time series tool

Syntax

```
ntstool
ntstool('close')
```

Description

`ntstool` opens the **Neural Net Time Series** app and leads you through solving a fitting problem using a two-layer feed-forward network.

For more information and an example of its usage, see “Shallow Neural Network Time-Series Prediction and Modeling”.

`ntstool('close')` closes the app.

See Also

Neural Net Time Series | `nctool` | `nftool` | `nprtool`

Introduced in R2010b

num2deriv

Numeric two-point network derivative function

Syntax

```
num2deriv('dperf_dwb',net,X,T,Xi,Ai,EW)
num2deriv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the two-point numeric derivative rule.

$$\frac{dy}{dx} = \frac{y(x+dx) - y(x)}{dx}$$

This function is much slower than the analytical (non-numerical) derivative functions, but is provided as a means of checking the analytical derivative functions. The other numerical function, `num5deriv`, is slower but more accurate.

`num2deriv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

`num2deriv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = num2deriv('dperf_dwb',net,x,t)
```

See Also

bttderiv | defaultderiv | fpderiv | num5deriv | staticderiv

Introduced in R2010b

num5deriv

Numeric five-point stencil neural network derivative function

Syntax

```
num5deriv('dperf_dwb',net,X,T,Xi,Ai,EW)
num5deriv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the five-point numeric derivative rule.

$$y_1 = y(x + 2dx)$$
$$y_2 = y(x + dx)$$
$$y_3 = y(x - dx)$$
$$y_4 = y(x - 2dx)$$
$$\frac{dy}{dx} = \frac{-y1 + 8y_2 - 8y_3 + y_4}{12dx}$$

This function is much slower than the analytical (non-numerical) derivative functions, but is provided as a means of checking the analytical derivative functions. The other numerical function, `num2deriv`, is faster but less accurate.

`num5deriv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

`num5deriv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
```

```
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = num5deriv('dperf_dwb',net,x,t)
```

See Also

bttderiv | defaultderiv | fpderiv | num2deriv | staticderiv

Introduced in R2010b

numelements

Number of elements in neural network data

Syntax

```
numelements(x)
```

Description

`numelements(x)` takes neural network data `x` in matrix or cell array form, and returns the number of elements in each signal.

If `x` is a matrix the result is the number of rows of `x`.

If `x` is a cell array the result is an `S`-by-`1` vector, where `S` is the number of signals (i.e., rows of `X`), and each element `S(i)` is the number of elements in each signal `i` (i.e., rows of `x{i,1}`).

Examples

This code calculates the number of elements represented by matrix data:

```
x = [1 2 3; 4 7 4]
n = numelements(x)
```

This code calculates the number of elements represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numelements(x)
```

See Also

`nndata` | `nnsize` | `getelements` | `setelements` | `catelements` | `numsamples` | `numsignals` | `numtimesteps`

Introduced in R2010b

numfinite

Number of finite values in neural network data

Syntax

```
numfinite(x)
```

Description

`numfinite(x)` takes a matrix or cell array of matrices and returns the number of finite elements in it.

Examples

```
x = [1 2; 3 NaN]
n = numfinite(x)

x = {[1 2; 3 NaN] [5 NaN; NaN 8]}
n = numfinite(x)
```

See Also

`numnan` | `nndata` | `nnsize`

Introduced in R2010b

numnan

Number of NaN values in neural network data

Syntax

```
numnan(x)
```

Description

`numnan(x)` takes a matrix or cell array of matrices and returns the number of NaN elements in it.

Examples

```
x = [1 2; 3 NaN]
n = numnan(x)

x = {[1 2; 3 NaN] [5 NaN; NaN 8]}
n = numnan(x)
```

See Also

`numnan` | `nndata` | `nnsize`

Introduced in R2010b

numsamples

Number of samples in neural network data

Syntax

```
numsamples(x)
```

Description

`numsamples(x)` takes neural network data `x` in matrix or cell array form, and returns the number of samples.

If `x` is a matrix, the result is the number of columns of `x`.

If `x` is a cell array, the result is the number of columns of the matrices in `x`.

Examples

This code calculates the number of samples represented by matrix data:

```
x = [1 2 3; 4 7 4]
n = numsamples(x)
```

This code calculates the number of samples represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numsamples(x)
```

See Also

`nndata` | `nnsize` | `getsamples` | `setsamples` | `catsamples` | `numelements` | `numsignals` | `numtimesteps`

Introduced in R2010b

numsignals

Number of signals in neural network data

Syntax

```
numsignals(x)
```

Description

`numsignals(x)` takes neural network data `x` in matrix or cell array form, and returns the number of signals.

If `x` is a matrix, the result is 1.

If `x` is a cell array, the result is the number of rows in `x`.

Examples

This code calculates the number of signals represented by matrix data:

```
x = [1 2 3; 4 7 4]
n = numsignals(x)
```

This code calculates the number of signals represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numsignals(x)
```

See Also

`nndata` | `nnsize` | `getsignals` | `setsignals` | `catsignals` | `numelements` | `numsamples` | `numtimesteps`

Introduced in R2010b

numtimesteps

Number of time steps in neural network data

Syntax

```
numtimesteps(x)
```

Description

`numtimesteps(x)` takes neural network data `x` in matrix or cell array form, and returns the number of signals.

If `x` is a matrix, the result is 1.

If `x` is a cell array, the result is the number of columns in `x`.

Examples

This code calculates the number of time steps represented by matrix data:

```
x = [1 2 3; 4 7 4]
n = numtimesteps(x)
```

This code calculates the number of time steps represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numtimesteps(x)
```

See Also

`nndata` | `nnsize` | `gettimesteps` | `settimesteps` | `cattimesteps` | `numelements` | `numsamples` | `numsignals`

Introduced in R2010b

openloop

Convert neural network closed-loop feedback to open loop

Syntax

```
net = openloop(net)
[net,xi,ai] = openloop(net,xi,ai)
```

Description

`net = openloop(net)` takes a neural network and opens any closed-loop feedback. For each feedback output `i` whose property `net.outputs{i}.feedbackMode` is `'closed'`, it replaces its associated feedback layer weights with a new input and input weight connections. The `net.outputs{i}.feedbackMode` property is set to `'open'`, and the `net.outputs{i}.feedbackInput` property is set to the index of the new input. Finally, the value of `net.outputs{i}.feedbackDelays` is subtracted from the delays of the feedback input weights (i.e., to the delays values of the replaced layer weights).

`[net,xi,ai] = openloop(net,xi,ai)` converts a closed-loop network and its current input delay states `xi` and layer delay states `ai` to open-loop form.

Examples

Convert NARX Network to Open-Loop Form

Here a NARX network is designed in open-loop form and then converted to closed-loop form, then converted back.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Yopen = net(Xs,Xi,Ai)
net = closeloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yclosed = net(Xs,Xi,Ai);
net = openloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yopen = net(Xs,Xi,Ai)
```

Convert Delay States

For examples on using `closeloop` and `openloop` to implement multistep prediction, see `narxnet` and `narnet`.

See Also

`closeloop` | `narnet` | `narxnet` | `noloop`

Introduced in R2010b

patternnet

Generate pattern recognition network

Syntax

```
net = patternnet(hiddenSizes,trainFcn,performFcn)
```

Description

`net = patternnet(hiddenSizes,trainFcn,performFcn)` returns a pattern recognition neural network with a hidden layer size of `hiddenSizes`, a training function, specified by `trainFcn`, and a performance function, specified by `performFcn`.

Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes. The target data for pattern recognition networks should consist of vectors of all zero values except for a 1 in element `i`, where `i` is the class they are to represent.

Examples

Construct and Train a Pattern Recognition Neural Network

This example shows how to design a pattern recognition network to classify iris flowers.

Load the training data.

```
[x,t] = iris_dataset;
```

Construct a pattern network with one hidden layer of size 10.

```
net = patternnet(10);
```

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```

Estimate the targets using the trained network.

```
y = net(x);
```

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,t,y)
```

```
perf = 0.0302
```

```
classes = vec2ind(y);
```

Input Arguments

`hiddenSizes` — Size of the hidden layers

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: `[10,8,5]`

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

`trainFcn` — Training function name

`'trainscg'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainlm'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

`performFcn` — Performance function

character vector

Performance function. The default value is `'crossentropy'`.

This argument defines the function used to measure the network’s performance. The performance function is used to calculate network performance during training.

For a list of functions, in the MATLAB command window, type `help nnperformance`.

Output Arguments

net — Pattern recognition network
`network` object

Pattern recognition neural network, returned as a `network` object.

See Also

`competlayer` | `lvqnet` | `network` | `nprtool` | `selforgmap`

Topics
“Classify Patterns with a Shallow Neural Network”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b

perceptron

Simple single-layer binary classifier

Syntax

```
perceptron(hardlimitTF,perceptronLF)
```

Description

Note Deep Learning Toolbox supports perceptrons for historical interest. For better results, you should instead use `patternnet`, which can solve nonlinearly separable problems. Sometimes the term “perceptrons” refers to feed-forward pattern recognition networks; but the original perceptron, described here, can solve only simple problems.

`perceptron(hardlimitTF,perceptronLF)` takes a hard limit transfer function, `hardlimitTF`, and a perceptron learning rule, `perceptronLF`, and returns a perceptron.

In addition to the default hard limit transfer function, perceptrons can be created with the `hardlims` transfer function. The other option for the perceptron learning rule is `learnpn`.

Perceptrons are simple single-layer binary classifiers, which divide the input space with a linear decision boundary.

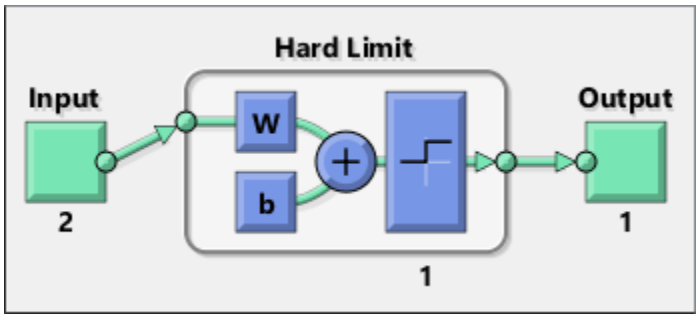
Perceptrons can learn to solve a narrow range of classification problems. They were one of the first neural networks to reliably solve a given class of problem, and their advantage is a simple learning rule.

Examples

Solve Simple Classification Problem Using Perceptron

This example shows how to use a perceptron to solve a simple classification logical-OR problem.

```
x = [0 0 1 1; 0 1 0 1];
t = [0 1 1 1];
net = perceptron;
net = train(net,x,t);
view(net)
y = net(x);
```



Input Arguments

`hardlimitTF` — Hard limit transfer function
`'hardlim'` (default)

Hard limit transfer function.

`perceptronLF` — Perceptron learning rule
`'learnp'` (default)

Perceptron learning rule.

See Also

`preparets` | `removedelay` | `patternnet` | `timedelaynet` | `narnet` | `narxnet`

Introduced in R2010b

perform

Calculate network performance

Syntax

```
perf = perform(net,t,y,ew)
```

Description

`perf = perform(net,t,y,ew)` takes a network `net`, targets `T`, outputs `Y`, and optionally error weights `EW`, and returns network performance calculated according to the `net.performFcn` and `net.performParam` property values.

The target and output data must have the same dimensions. The error weights may be the same dimensions as the targets, in the most general case, but may also have any of its dimensions be 1. This gives the flexibility of defining error weights across any dimension desired.

Examples

Calculate Network Performance with 'perform' Function

This example shows how to calculate the performance of a feed-forward network with the `perform` function.

Create a feed-forward network using the data from the simple fit data set and calculate its performance.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y)
```

```
perf =

   2.3654e-06
```

Input Arguments

net — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

t — Network targets

matrix | cell array

Network targets, specified as a matrix or cell array.

y — Network outputs
matrix | cell array

Network outputs, specified as a matrix or cell array.

ew — Error weights
vector | matrix | cell array

Error weights, specified as a vector, matrix, or cell array.

Error weights can be defined by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2]; % Across 4 samples
ew = [0.1; 0.5; 1.0]; % Across 3 elements
ew = {0.1 0.2 0.3 0.5 1.0}; % Across 5 timesteps
ew = {1.0; 0.5}; % Across 2 outputs
```

The error weights can also be defined across any combination, such as across two time-series (i.e., two samples) over four timesteps.

```
ew = {[0.5 0.4],[0.3 0.5],[1.0 1.0],[0.7 0.5]};
```

In the general case, error weights may have exactly the same dimensions as targets, in which case each target value will have an associated error weight.

The default error weight treats all errors the same.

```
ew = {1}
```

Output Arguments

perf — Network performance
scalar

Network performance, returned as a scalar.

See Also

`train` | `configure` | `init`

Introduced in R2010b

plotconfusion

Plot classification confusion matrix

Syntax

```
plotconfusion(targets,outputs)
plotconfusion(targets,outputs,name)
plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn,namen)
```

Description

`plotconfusion(targets,outputs)` plots a confusion matrix for the true labels `targets` and predicted labels `outputs`. Specify the labels as categorical vectors, or in one-of-N (one-hot) form.

Tip `plotconfusion` is not recommended for categorical labels. Use `confusionchart` instead.

On the confusion matrix plot, the rows correspond to the predicted class (Output Class) and the columns correspond to the true class (Target Class). The diagonal cells correspond to observations that are correctly classified. The off-diagonal cells correspond to incorrectly classified observations. Both the number of observations and the percentage of the total number of observations are shown in each cell.

The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the precision (or positive predictive value) and false discovery rate, respectively. The row at the bottom of the plot shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the recall (or true positive rate) and false negative rate, respectively. The cell in the bottom right of the plot shows the overall accuracy.

`plotconfusion(targets,outputs,name)` plots a confusion matrix and adds `name` to the beginning of the plot title.

`plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn,namen)` plots multiple confusion matrices in one figure and adds the `name` arguments to the beginnings of the titles of the corresponding plots.

Examples

Plot Confusion Matrix Using Categorical Labels

Load the data consisting of synthetic images of handwritten digits. `XTrain` is a 28-by-28-by-1-by-5000 array of images and `YTrain` is a categorical vector containing the image labels.

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

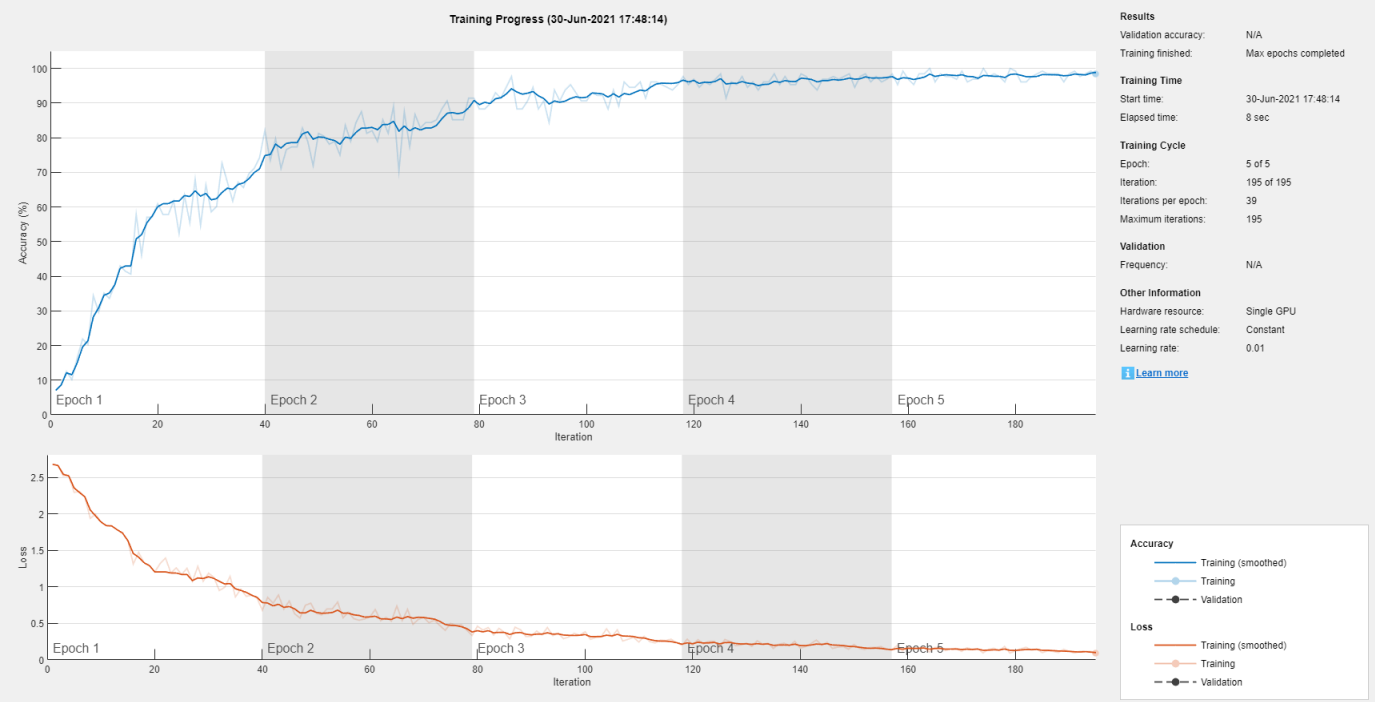
```
  Name           Size              Bytes  Class          Attributes

  YTrain      5000x1               6062  categorical
```

Define the architecture of a convolutional neural network.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,16,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify training options and train the network.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',5, ...
    'Verbose',false, ...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,layers,options);
```

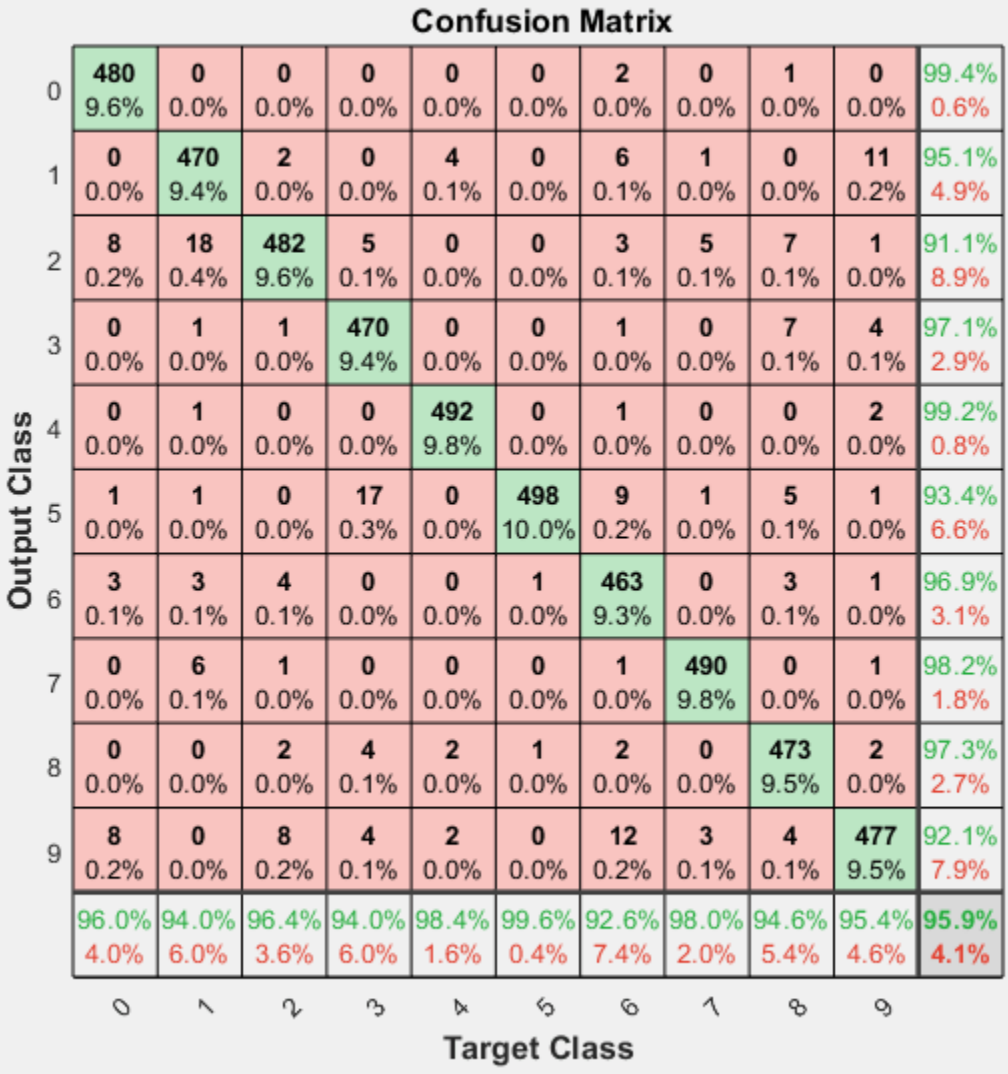


Load and classify test data using the trained network.

```
[XTest,YTest] = digitTest4DArrayData;
YPredicted = classify(net,XTest);
```

Plot the confusion matrix of the true test labels `YTest` and the predicted labels `YPredicted`.

```
plotconfusion(YTest,YPredicted)
```



The rows correspond to the predicted class (Output Class) and the columns correspond to the true class (Target Class). The diagonal cells correspond to observations that are correctly classified. The off-diagonal cells correspond to incorrectly classified observations. Both the number of observations and the percentage of the total number of observations are shown in each cell.

The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the precision (or positive predictive value) and false discovery rate, respectively. The row at the bottom of the plot

shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the recall (or true positive rate) and false negative rate, respectively. The cell in the bottom right of the plot shows the overall accuracy.

Close all figures.

```
close(findall(groot,'Type','figure'))
```

Plot Confusion Matrix Using One-of-N Labels

Load sample data using the `cancer_dataset` function. `XTrain` is a 9-by-699 matrix defining nine attributes of 699 biopsies. `YTrain` is a 2-by-699 matrix where each column indicates the correct category of the corresponding observation. Each column of `YTrain` has one element that equals one in either the first or second row, corresponding to the cancer being benign or malignant, respectively. For more information on this dataset, type `help cancer_dataset` at the command line.

```
rng default
[XTrain,YTrain] = cancer_dataset;
YTrain(:,1:10)
```

```
ans = 2×10

     1     1     1     0     1     1     0     0     0     1
     0     0     0     1     0     0     1     1     1     0
```

Create a pattern recognition network and train it using the sample data.

```
net = patternnet(10);
net = train(net,XTrain,YTrain);
```

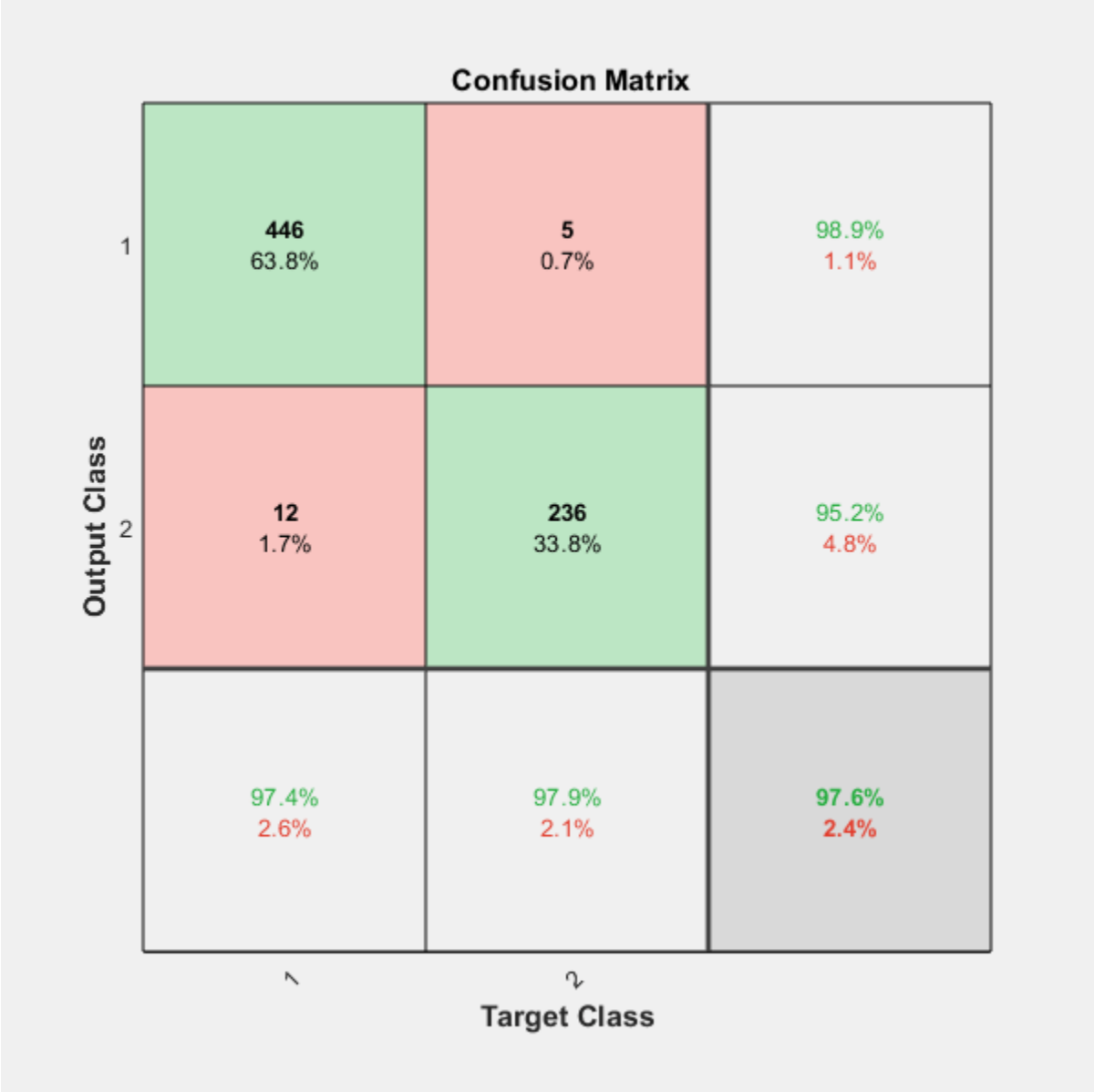
Estimate the cancer status using the trained network. Each column of the matrix `YPredicted` contains the predicted probabilities of each observation belonging to class 1 and class 2, respectively.

```
YPredicted = net(XTrain);
YPredicted(:,1:10)
```

```
ans = 2×10

    0.9980    0.9979    0.9894    0.0578    0.9614    0.9960    0.0026    0.0023    0.0084    0.9
    0.0020    0.0021    0.0106    0.9422    0.0386    0.0040    0.9974    0.9977    0.9916    0.0
```

Plot the confusion matrix. To create the plot, `plotconfusion` labels each observation according to the highest class probability.

```
plotconfusion(YTrain,YPredicted)
```



In this figure, the first two diagonal cells show the number and percentage of correct classifications by the trained network. For example, 446 biopsies are correctly classified as benign. This corresponds to 63.8% of all 699 biopsies. Similarly, 236 cases are correctly classified as malignant. This corresponds to 33.8% of all biopsies.

5 of the malignant biopsies are incorrectly classified as benign and this corresponds to 0.7% of all 699 biopsies in the data. Similarly, 12 of the benign biopsies are incorrectly classified as malignant and this corresponds to 1.7% of all data.

Out of 451 benign predictions, 98.9% are correct and 1.1% are wrong. Out of 248 malignant predictions, 95.2% are correct and 4.8% are wrong. Out of 458 benign cases, 97.4% are correctly predicted as benign and 2.6% are predicted as malignant. Out of 241 malignant cases, 97.9% are correctly classified as malignant and 2.1% are classified as benign.

Overall, 97.6% of the predictions are correct and 2.4% are wrong.

Input Arguments

`targets` — True class labels

categorical vector | matrix

True class labels, specified one of the following:

- A categorical vector, where each element is the class label of one observation. The `outputs` and `targets` arguments must have the same number of elements. If the categorical vectors define underlying classes, then `plotconfusion` displays all the underlying classes, even if there are no observations of some of the underlying classes. If the arguments are ordinal categorical vectors, then they must both define the same underlying categories, in the same order.
- An *N*-by-*M* matrix, where *N* is the number of classes and *M* is the number of observations. Each column of the matrix must be in one-of-N (one-hot) form, where a single element equal to 1 indicates the true label and all other elements equal 0.

`outputs` — Predicted class labels

categorical vector | matrix

Predicted class labels, specified one of the following:

- A categorical vector, where each element is the class label of one observation. The `outputs` and `targets` arguments must have the same number of elements. If the categorical vectors define underlying classes, then `plotconfusion` displays all the underlying classes, even if there are no observations of some of the underlying classes. If the arguments are ordinal categorical vectors, then they must both define the same underlying categories, in the same order.
- An *N*-by-*M* matrix, where *N* is the number of classes and *M* is the number of observations. Each column of the matrix can be in one-of-N (one-hot) form, where a single element equal to 1 indicates the predicted label, or in the form of probabilities that sum to one.

`name` — Name of the confusion matrix

character array

Name of the confusion matrix, specified as a character array. `plotconfusion` adds the specified `name` to the beginning of the plot title.

Data Types: `char`

See Also

`trainNetwork` | `trainingOptions`

Introduced in R2008a

plotep

Plot weight-bias position on error surface

Syntax

```
H = plotep(W,B,E)
H = plotep(W,B,E,H)
```

Description

`plotep` is used to show network learning on a plot created by `plotes`.

`H = plotep(W,B,E)` takes these arguments,

`W`	Current weight value
`B`	Current bias value
`E`	Current error

and returns a cell array `H`, containing information for continuing the plot.

`H = plotep(W,B,E,H)` continues plotting using the cell array `H` returned by the last call to `plotep`.

`H` contains handles to dots plotted on the error surface, so they can be deleted next time; as well as points on the error contour, so they can be connected.

See Also

`errsurf` | `plotes`

Introduced before R2006a

ploterrcorr

Plot autocorrelation of error time series

Syntax

```
ploterrcorr(error)
ploterrcorr(errors,'outputIndex',outIdx)
```

Description

`ploterrcorr(error)` takes an error time series and plots the autocorrelation of errors across varying lags.

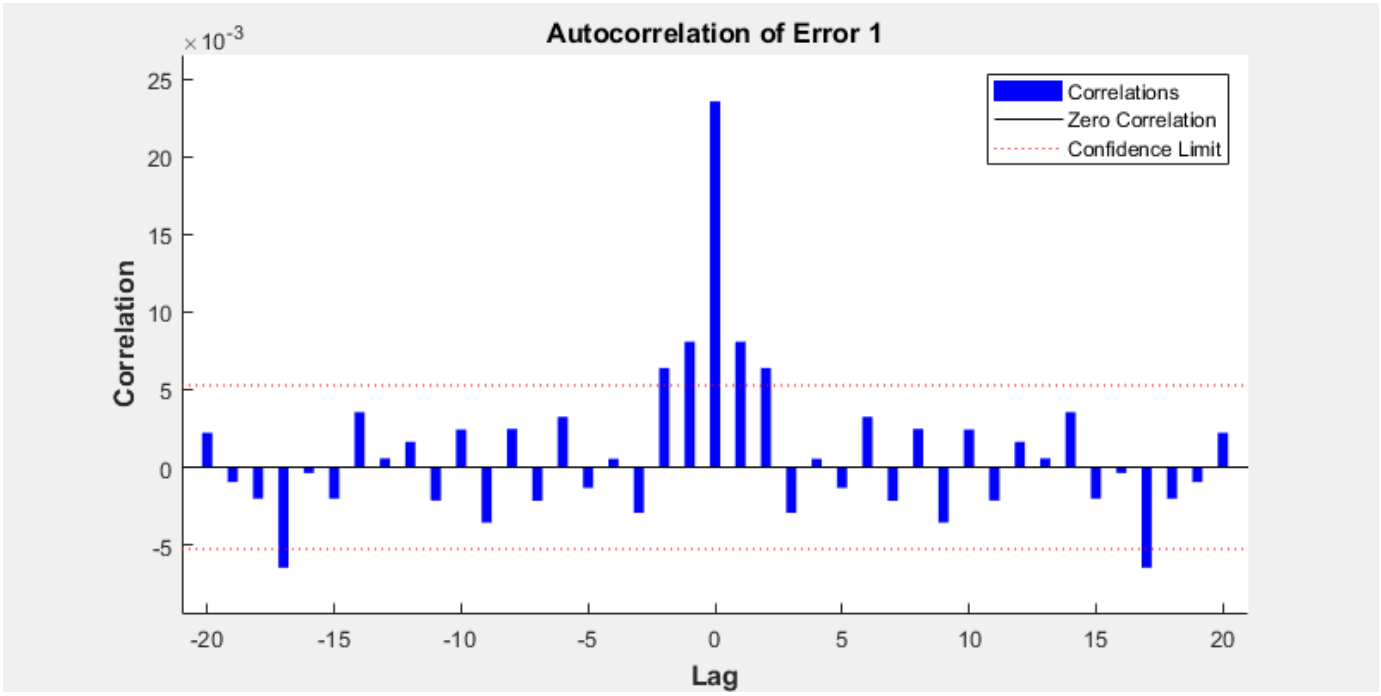
`ploterrcorr(errors,'outputIndex',outIdx)` uses the optional property name/value pair to define which output error autocorrelation is plotted. The default is 1.

Examples

Plot Autocorrelation of Errors

Here a NARX network is used to solve a time series problem.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
ploterrcorr(E)
```



See Also

`plotinerrcorr` | `plotresponse`

Introduced in R2010b

ploterrhist

Plot error histogram

Syntax

```
ploterrhist(e)
ploterrhist(e1,'name1',e2,'name2',...)
ploterrhist(...,'bins',bins)
```

Description

`ploterrhist(e)` plots a histogram of error values `e`.

`ploterrhist(e1,'name1',e2,'name2',...)` takes any number of errors and names and plots each pair.

`ploterrhist(...,'bins',bins)` takes an optional property name-value pair which defines the number of bins to use in the histogram plot. The default is 20.

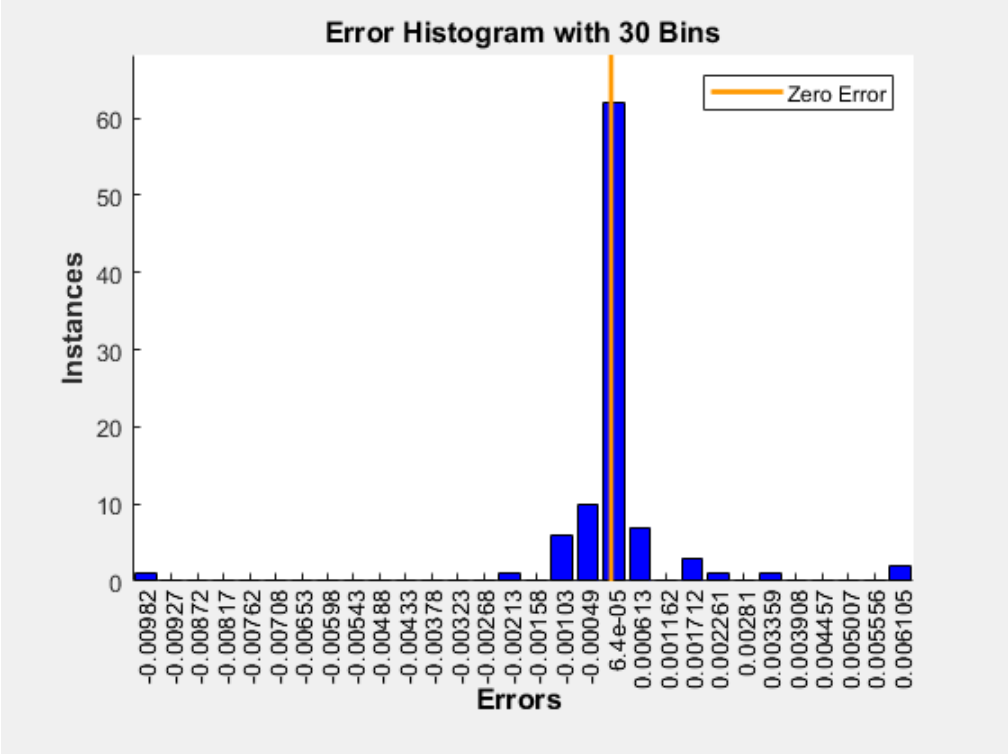
Examples

Plot Histogram of Error Values

This example shows how to plot the histogram of error values of a trained feed-forward network.

Create a feed-forward network and train it using the data from the simple fit data set. Then plot the histogram of error values.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
e = t - y;
ploterrhist(e,'bins',30)
```



Input Arguments

e — Errors

vector | matrix

Errors to plot, specified as a vector or a matrix.

'bins',bins — Number of bins

20 (default) | integer

Number of bins to use in the histogram plot, specified as the comma-separated pair consisting of `'bins'` and an integer.

See Also

`plotconfusion` | `ploterrcorr` | `plotinerrcorr`

Introduced in R2010b

plotes

Plot error surface of single-input neuron

Syntax

```
plotes(WV,BV,ES,V)
```

Description

`plotes(WV,BV,ES,V)` takes these arguments,

`WV`	1-by-`N` row vector of values of `W`
`BV`	1-by-`M` row vector of values of `B`
`ES`	`M`-by-`N` matrix of error vectors
`V`	View (default = `[-37.5, 30]`)

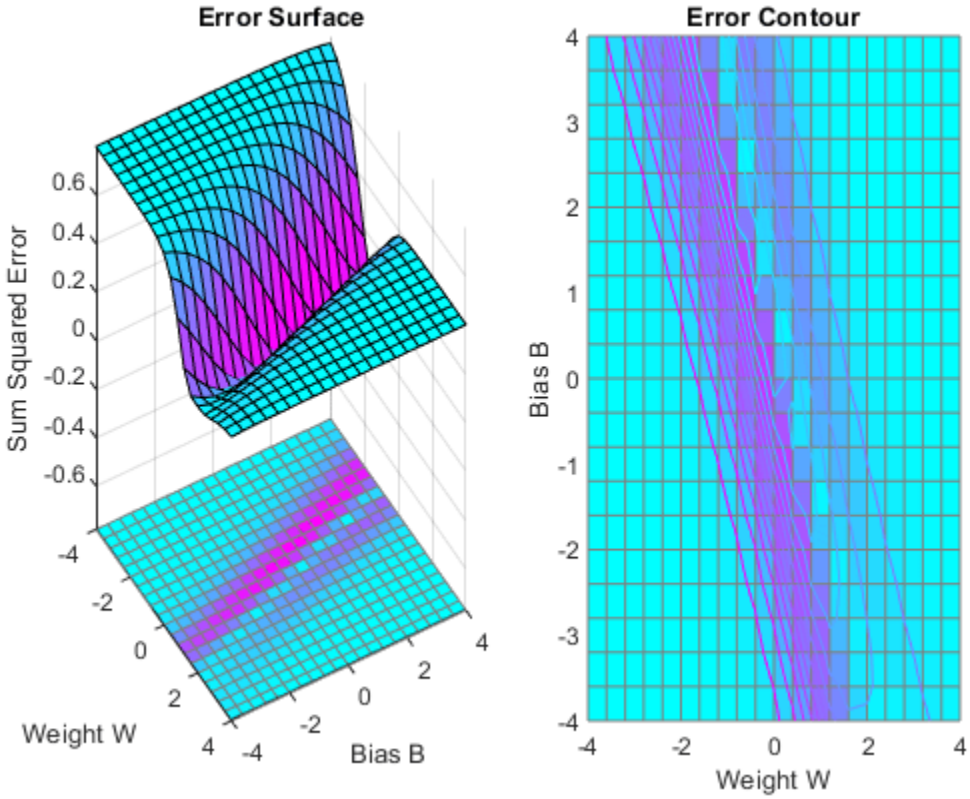
and plots the error surface with a contour underneath.

Calculate the error surface `ES` with `errsurf`.

Examples

Plot Error Surface of Single-Input Neuron

```
p = [3 2];
t = [0.4 0.8];
wv = -4:0.4:4;
bv = wv;
ES = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```



See Also

`errsurf`

Introduced before R2006a

plotfit

Plot function fit

Syntax

```
plotfit(net,inputs,targets)
plotfit(net,inputs1,targets1,name1,inputs2,targets2,name2,...)
plotfit(...,'outputIndex',outputIndex)
```

Description

`plotfit(net,inputs,targets)` plots the output function of a network across the range of the inputs `inputs` and also plots target `targets` and output data points associated with values in `inputs`. Error bars show the difference between outputs and `targets`.

The plot appears only for networks with one input.

Only the first output/targets appear if the network has more than one output.

`plotfit(net,inputs1,targets1,name1,inputs2,targets2,name2,...)` plots multiple sets of data.

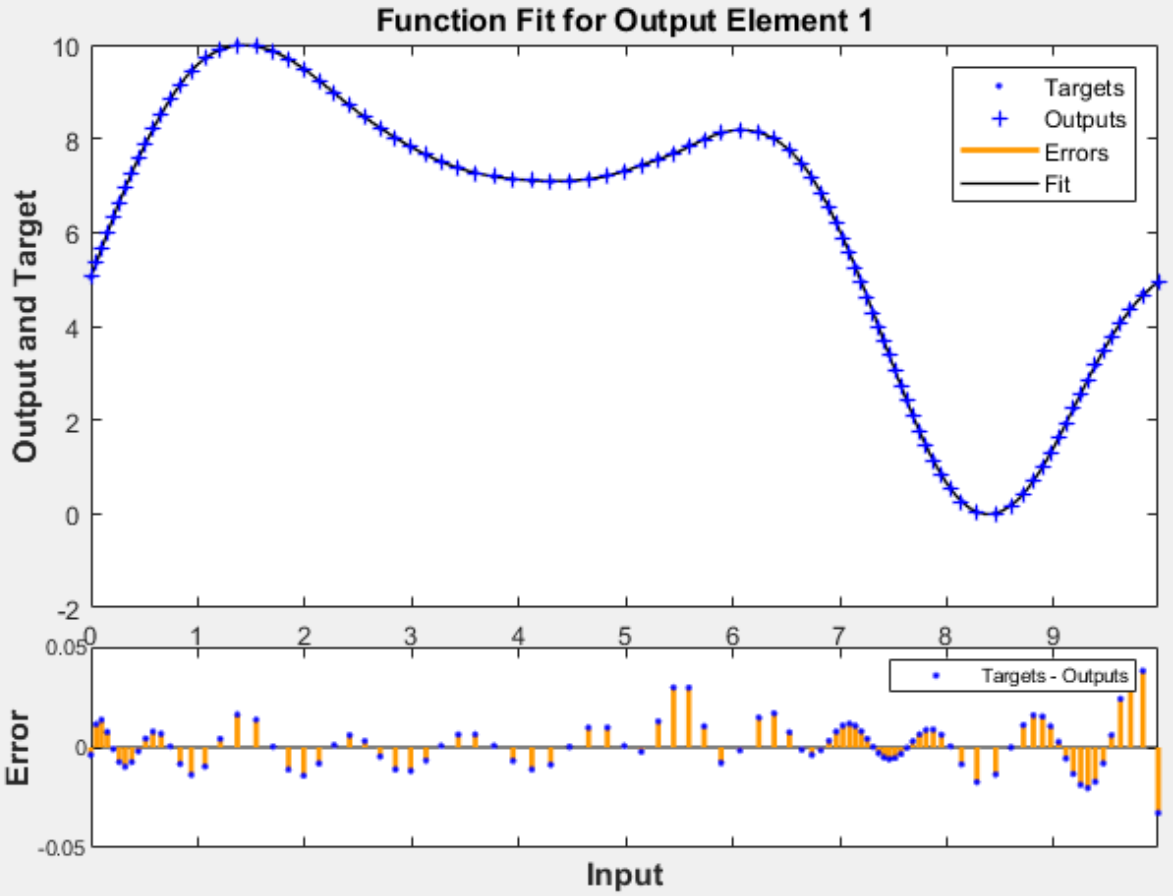
`plotfit(...,'outputIndex',outputIndex)` plots using an optional parameter that overrides the default index of the output element.

Examples

Plot Output and Target Values

This example shows how to use a feed-forward network to solve a simple fitting problem.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
plotfit(net,x,t)
```



Input Arguments

`net` — Input network
network object

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

`inputs` — Network inputs
matrix | cell array

Network inputs, specified as a matrix or cell array.

`targets` — Network targets
matrix | cell array

Network targets, specified as a matrix or cell array.

See Also

`plottrainstate`

Introduced in R2008a

plotinerrcorr

Plot input to error time-series cross-correlation

Syntax

```
plotinerrcorr(x,e)
plotinerrcorr(...,'inputIndex',inputIndex)
plotinerrcorr(...,'outputIndex',outputIndex)
```

Description

`plotinerrcorr(x,e)` takes an input time series `x` and an error time series `e`, and plots the cross-correlation of inputs to errors across varying lags.

`plotinerrcorr(...,'inputIndex',inputIndex)` optionally defines which input element is being correlated and plotted. The default is 1.

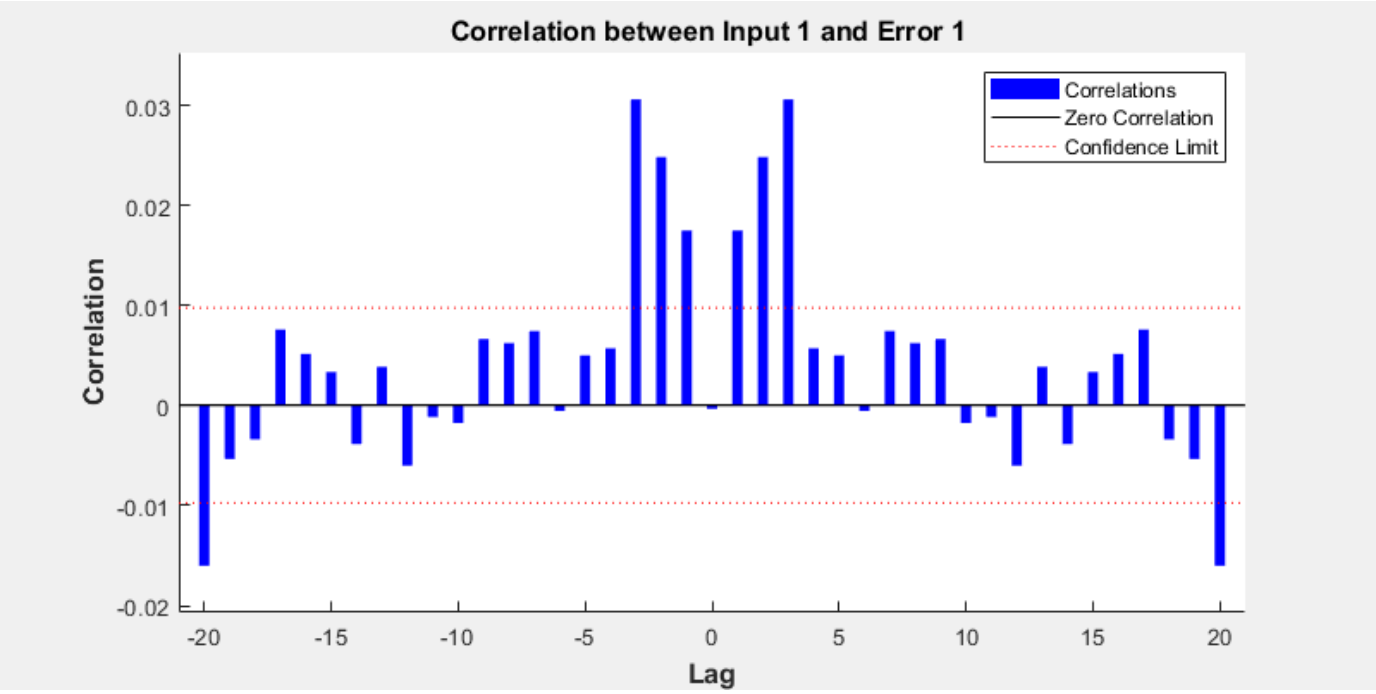
`plotinerrcorr(...,'outputIndex',outputIndex)` optionally defines which error element is being correlated and plotted. The default is 1.

Examples

Plot Cross-Correlation of Inputs to Errors

Here a NARX network is used to solve a time series problem.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
plotinerrcorr(Xs,E)
```



See Also

`ploterrcorr` | `plotresponse` | `ploterrhist`

Introduced in R2010b

plotpc

Plot classification line on perceptron vector plot

Syntax

```
plotpc(W,B)
plotpc(W,B,H)
```

Description

`plotpc(W,B)` takes these inputs,

`W`	S-by-R weight matrix (R must be 3 or less)
`B`	S-by-1 bias vector

and returns a handle to a plotted classification line.

`plotpc(W,B,H)` takes an additional input,

`H`	Handle to last plotted line

and deletes the last line before plotting the new one.

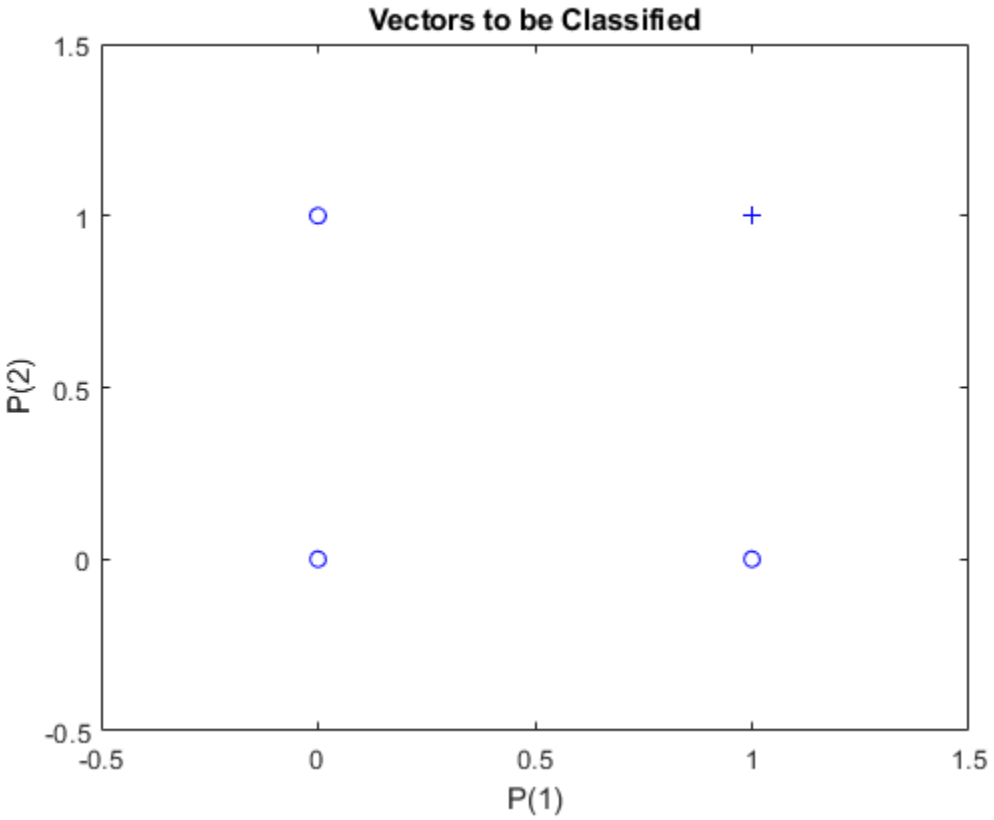
This function does not change the current axis and is intended to be called after `plotpv`.

Examples

Plot Classification Line

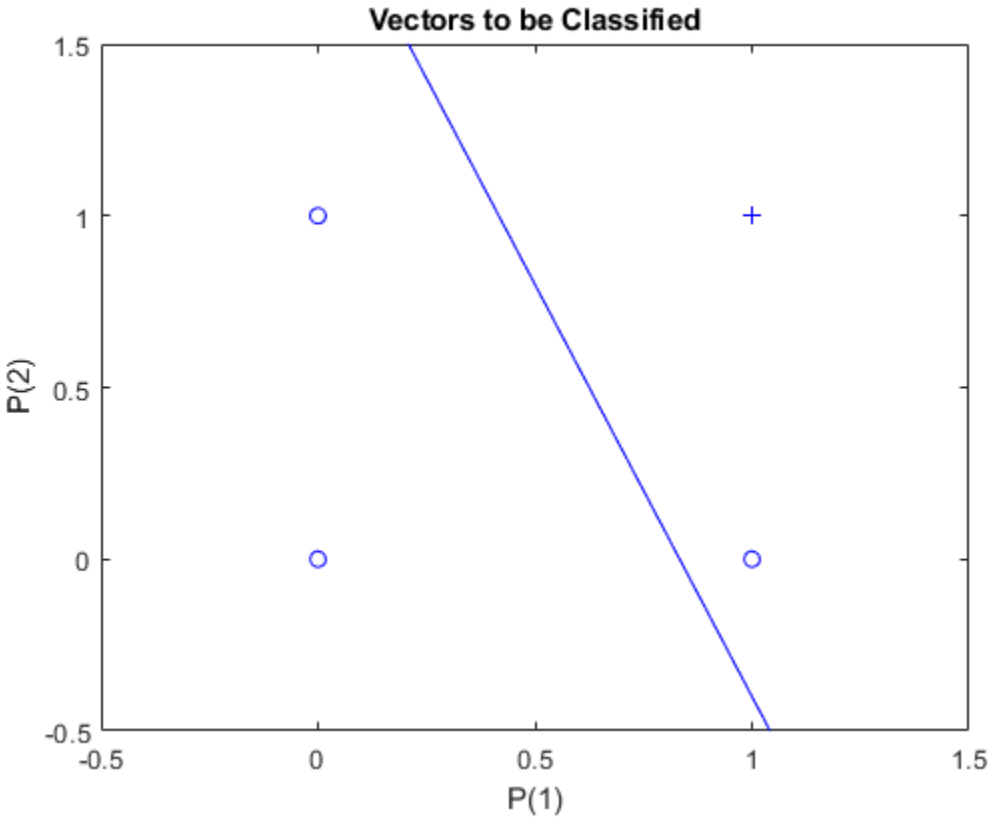
The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```



The following code creates a perceptron, assigns values to its weights and biases, and plots the resulting classification line.

```
net = perceptron;
net = configure(net,p,t);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```



See Also

`plotpv`

Introduced before R2006a

plotperform

Plot network performance

Syntax

```
plotperform(TR)
```

Description

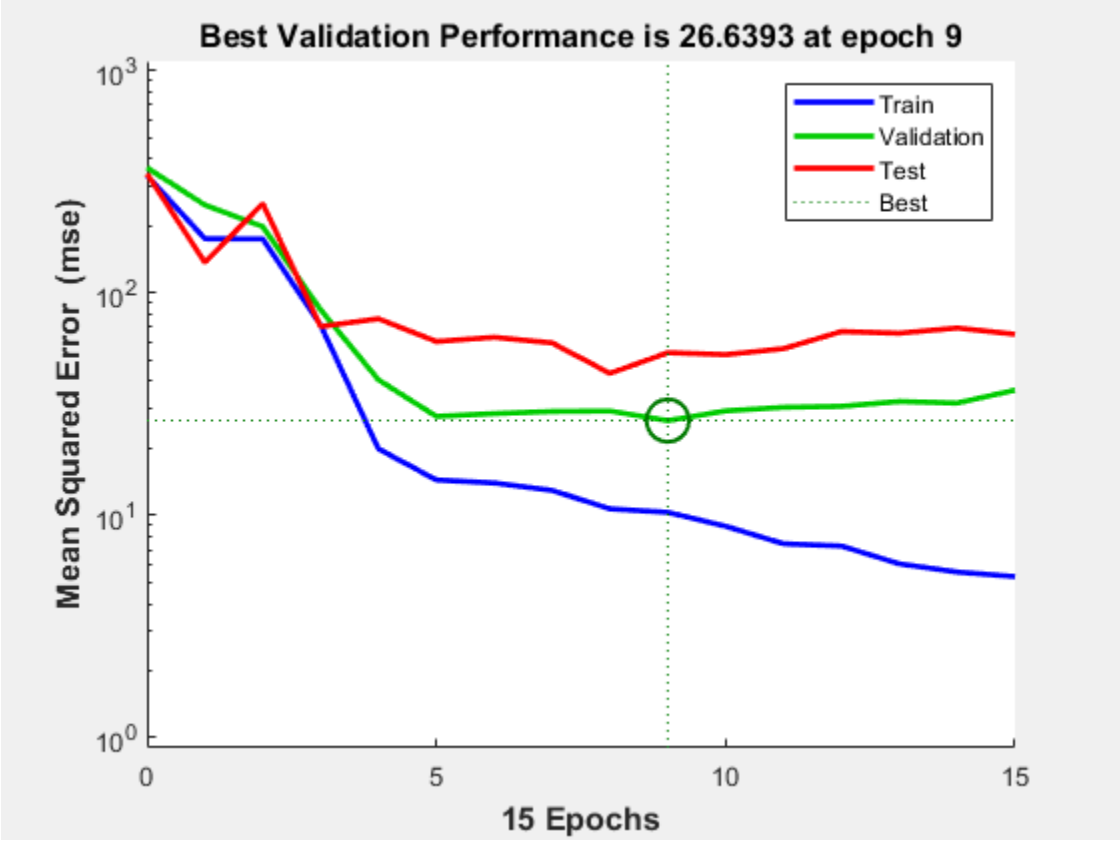
`plotperform(TR)` plots error vs. epoch for the training, validation, and test performances of the training record TR returned by the function `train`.

Examples

Plot Validation Performance of Network

This example shows how to use `plotperform` to obtain a plot of training record error values against the number of training epochs.

```
[x,t] = bodyfat_dataset;
net = feedforwardnet(10);
[net,tr] = train(net,x,t);
plotperform(tr)
```



Generally, the error reduces after more epochs of training, but might start to increase on the validation data set as the network starts overfitting the training data. In the default setup, the training stops after six consecutive increases in validation error, and the best performance is taken from the epoch with the lowest validation error.

Input Arguments

TR — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`)
- A list of training state names (`states`)
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

See Also

`plottrainstate`

Introduced in R2008a

plotpv

Plot perceptron input/target vectors

Syntax

```
plotpv(P,T)
plotpv(P,T,V)
```

Description

`plotpv(P,T)` takes these inputs,

`P`	R-by-Q matrix of input vectors (R must be 3 or less)
`T`	S-by-Q matrix of binary target vectors (S must be 3 or less)

and plots column vectors in `P` with markers based on `T`.

`plotpv(P,T,V)` takes an additional input,

`V`	Graph limits = `[x_min x_max y_min y_max]`

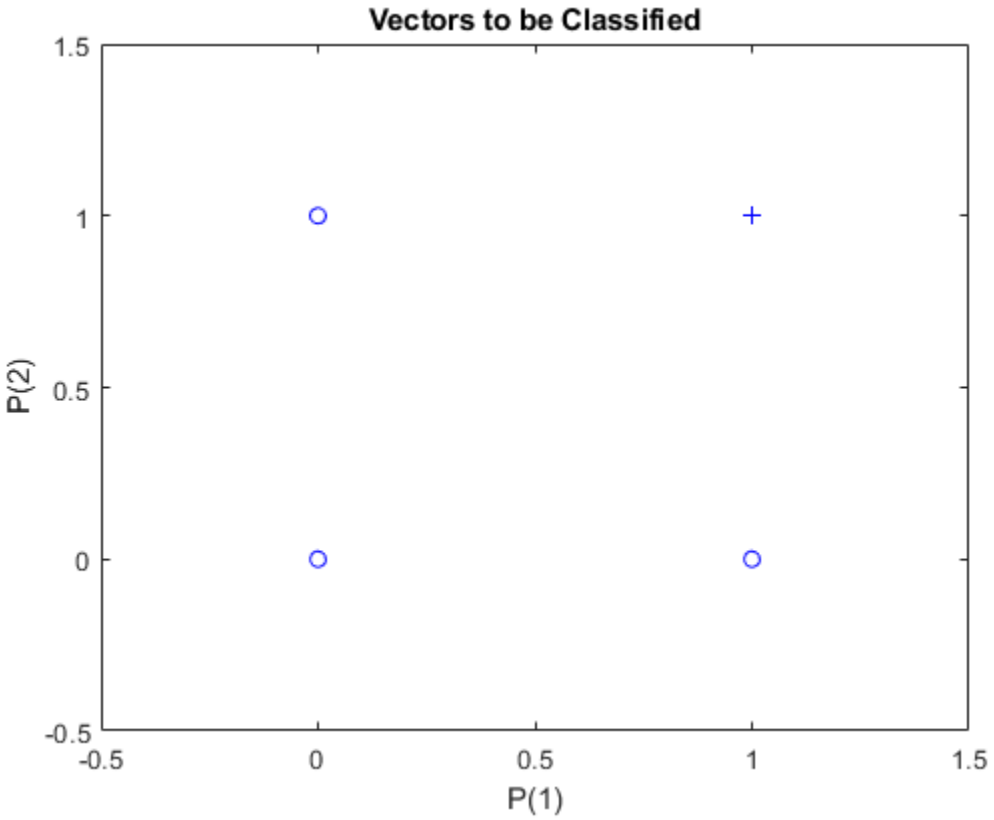
and plots the column vectors with limits set by `V`.

Examples

Plot Inputs and Targets for Perceptron

This example shows how to define and plot the inputs and targets for a perceptron.

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```



See Also

`plotpc`

Introduced before R2006a

plotregression

Plot linear regression

Syntax

```
plotregression(targets,outputs)
plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)
```

Description

`plotregression(targets,outputs)` plots the linear regression of `targets` relative to `outputs`.

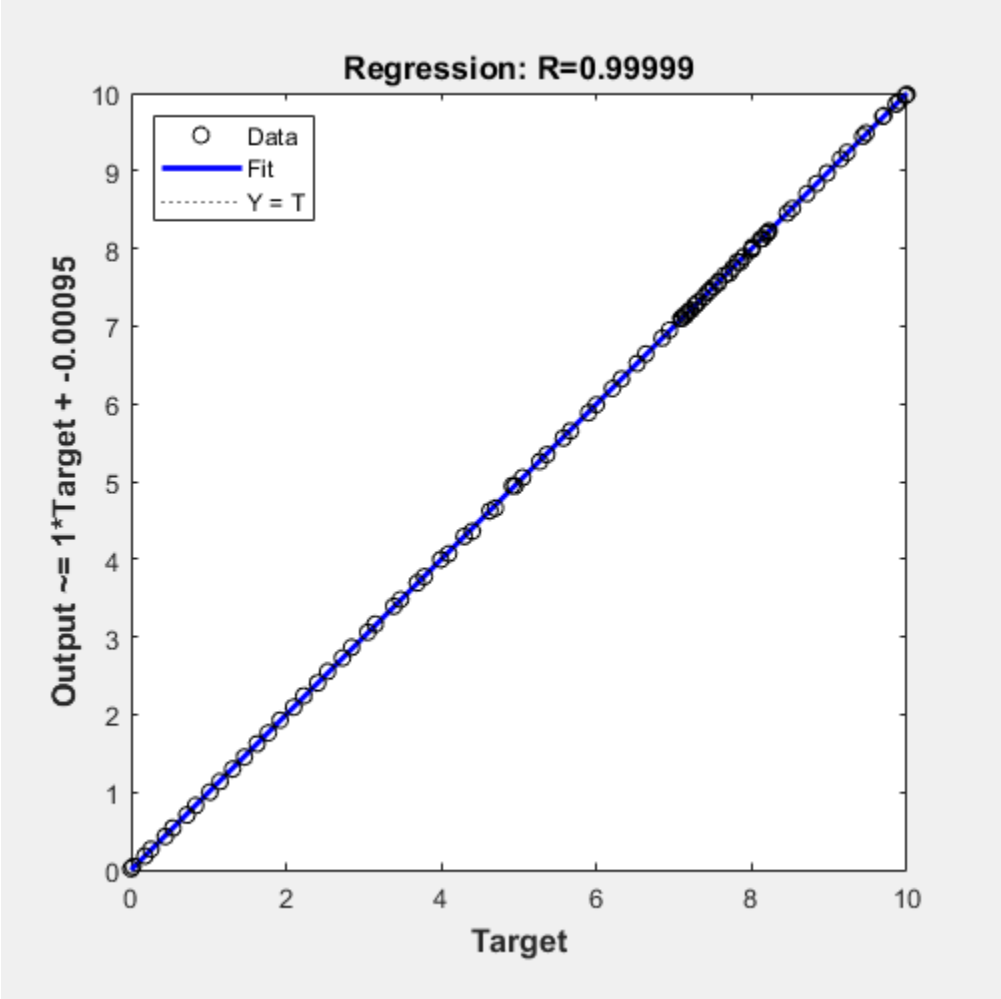
`plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)` generates multiple plots.

Examples

Plot Regression

This example shows how to plot the linear regression of a feedforward net.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
plotregression(t,y,'Regression')
```



Input Arguments

`targets` — Network targets
matrix | cell array

Network targets, specified as a matrix or cell array.

`outputs` — Network outputs
matrix | cell array

Network outputs, specified as a matrix or cell array.

See Also

`plottrainstate`

Introduced in R2008a

plotresponse

Plot dynamic network time series response

Syntax

```
plotresponse(t,y)
plotresponse(t1,'name',t2,'name2',...,y)
plotresponse(...,'outputIndex',outputIndex)
```

Description

`plotresponse(t,y)` takes a target time series `t` and an output time series `y`, and plots them on the same axis showing the errors between them.

`plotresponse(t1,'name',t2,'name2',...,y)` takes multiple target/name pairs, typically defining training, validation and testing targets, and the output. It plots the responses with colors indicating the different target sets.

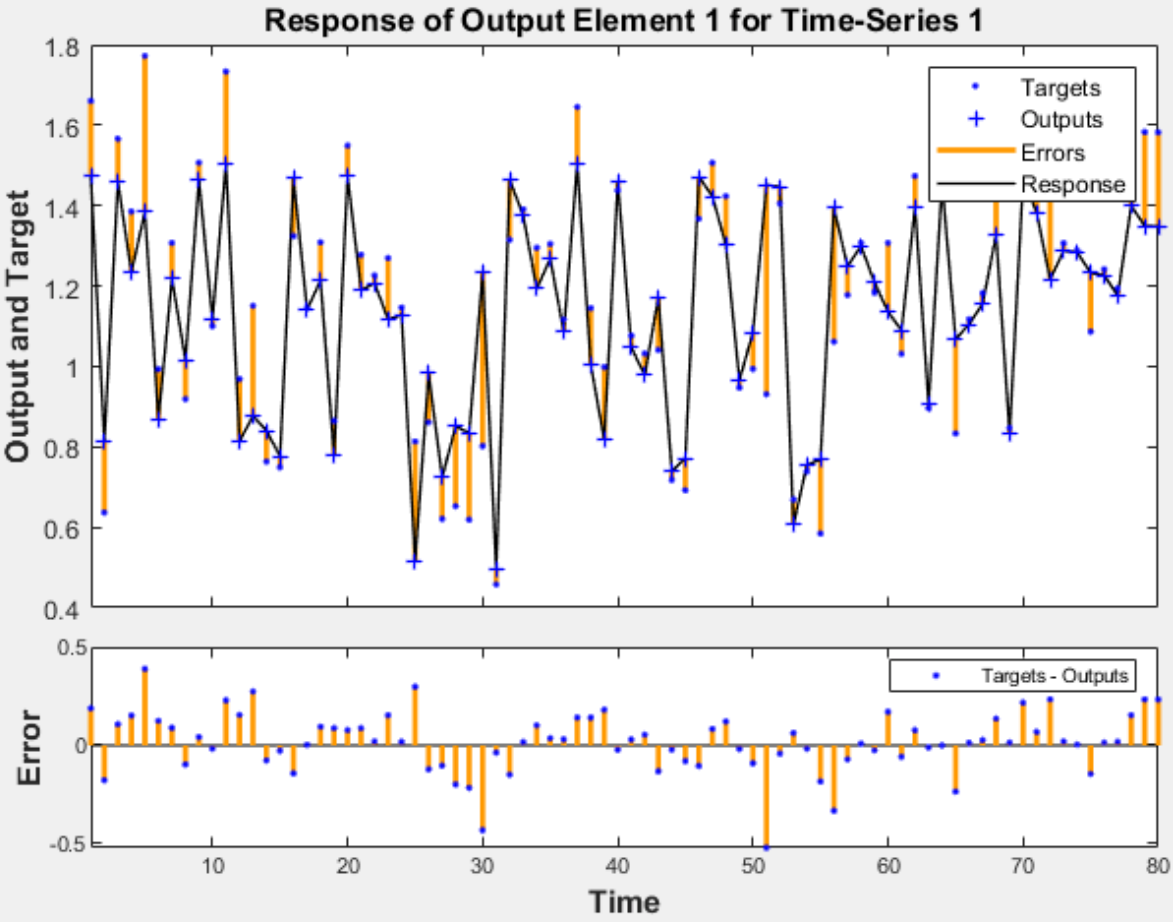
`plotresponse(...,'outputIndex',outputIndex)` optionally defines which error element is being correlated and plotted. The default is 1.

Examples

Plot Target and Output Time Series Data

This example shows how to use a NARX network to solve a time series problem.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
plotresponse(Ts,Y)
```



See Also

`ploterrcorr` | `plotinerrcorr` | `ploterrhist`

Introduced in R2010b

plotroc

Plot receiver operating characteristic

Syntax

```
plotroc(targets,outputs)
plotroc(targets1,outputs2,'name1',...)
```

Description

`plotroc(targets,outputs)` plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

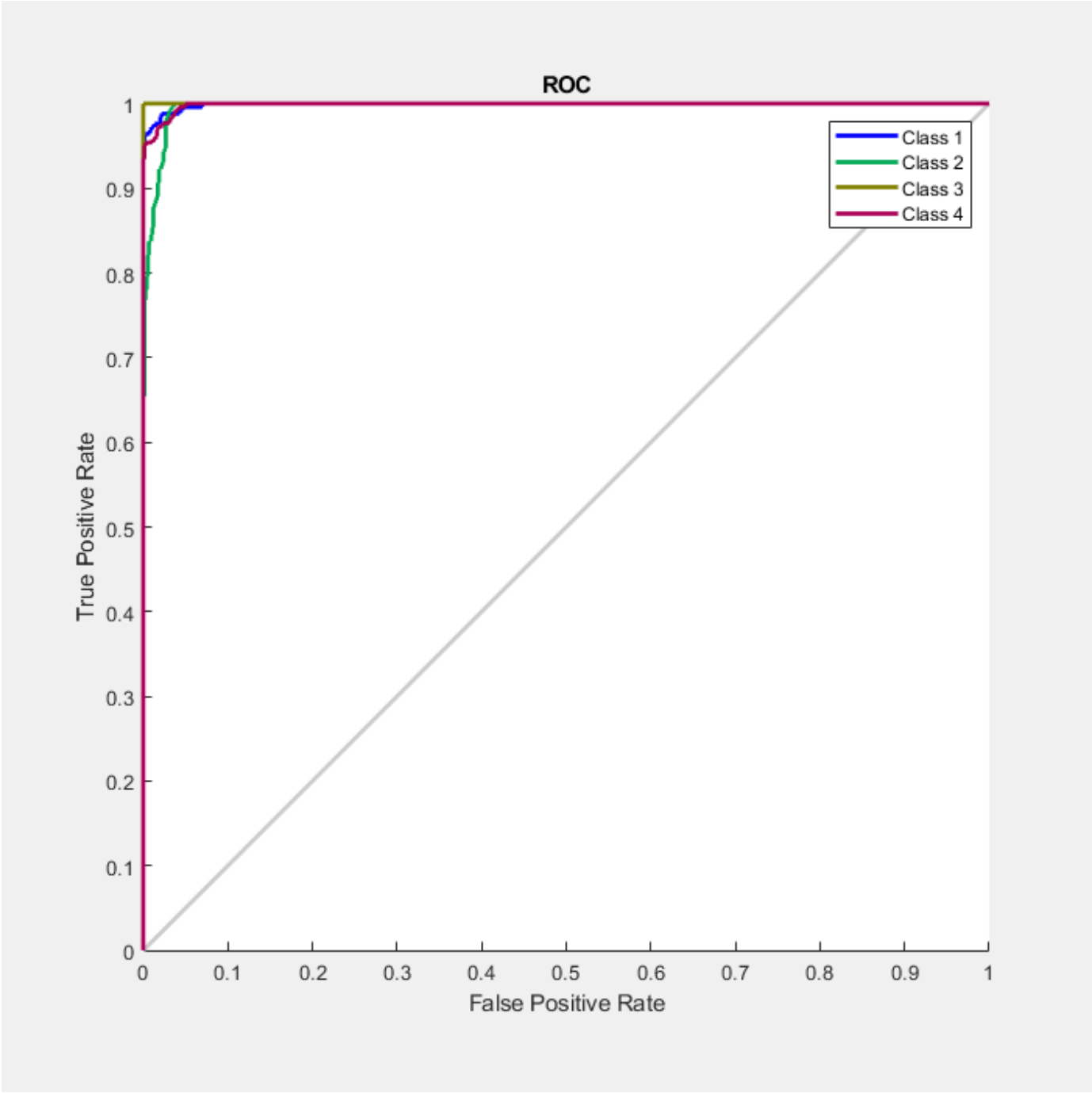
`plotroc(targets1,outputs2,'name1',...)` generates multiple plots.

Examples

Plot Receiver Operating Characteristic

This example shows how to plot the receiver operating characteristic for each output class in a pattern network.

```
load simplecluster_dataset
net = patternnet(20);
net = train(net,simpleclusterInputs,simpleclusterTargets);
simpleclusterOutputs = sim(net,simpleclusterInputs);
plotroc(simpleclusterTargets,simpleclusterOutputs)
```



Input Arguments

`targets` — Network targets

matrix | cell array

Network targets, specified as a matrix or cell array.

`outputs` — Network outputs
matrix | cell array

Network outputs, specified as a matrix or cell array.

See Also

`roc`

Introduced in R2008a

plotsom

Plot self-organizing map

Syntax

```
plotsom(pos)
plotsom(W,D,ND)
```

Description

`plotsom(pos)` takes one argument,

POS	N-by-S matrix of S N-dimension neural positions

and plots the neuron positions with red dots, linking the neurons within a Euclidean distance of 1.

`plotsom(W,D,ND)` takes three arguments,

W	S-by-R weight matrix
D	S-by-S distance matrix
ND	Neighborhood distance (default = 1)

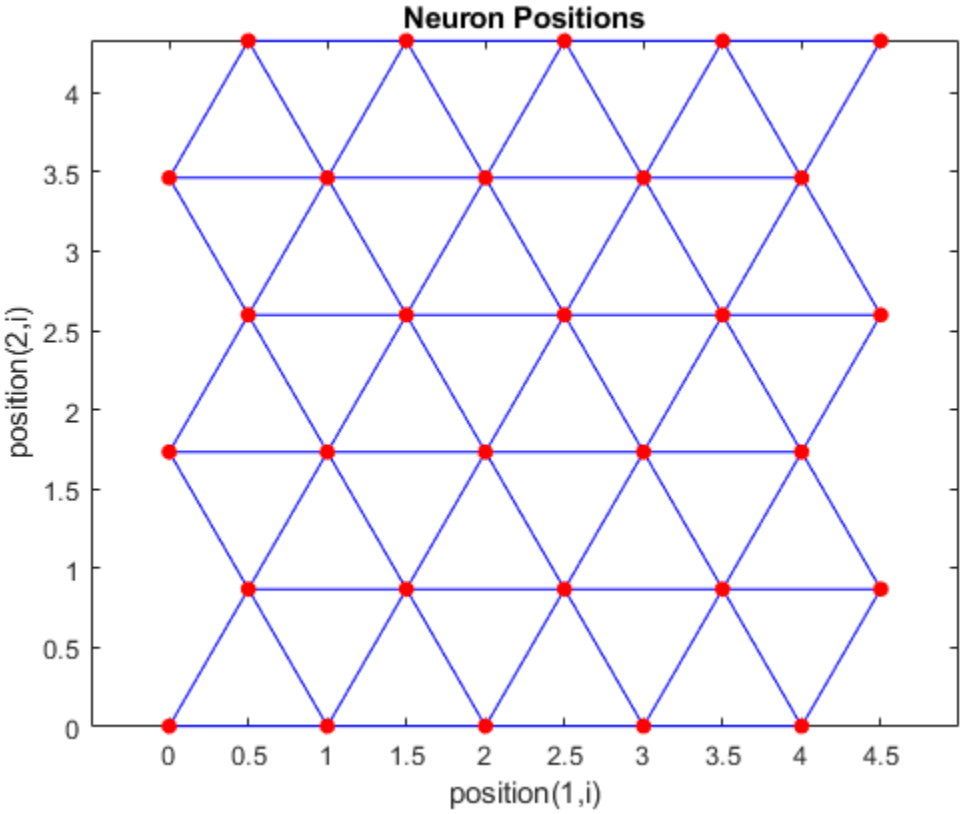
and plots the neuron's weight vectors with connections between weight vectors whose neurons are within a distance of 1.

Examples

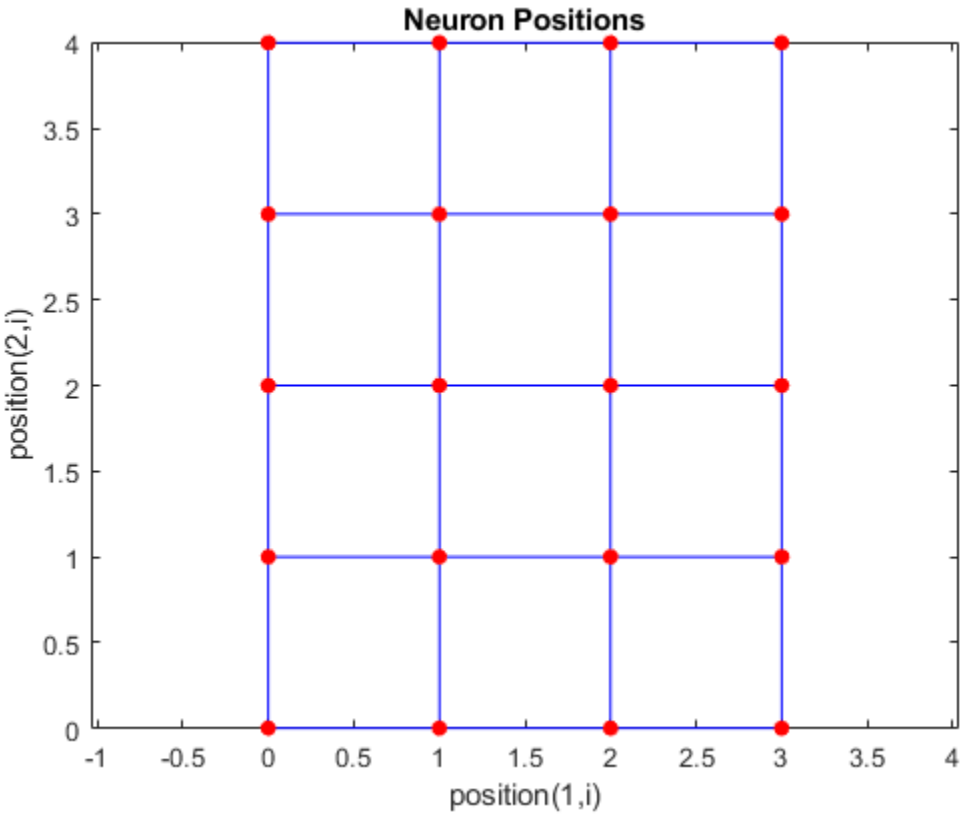
Plot Self-Organizing Maps

These examples generate plots of various layer topologies.

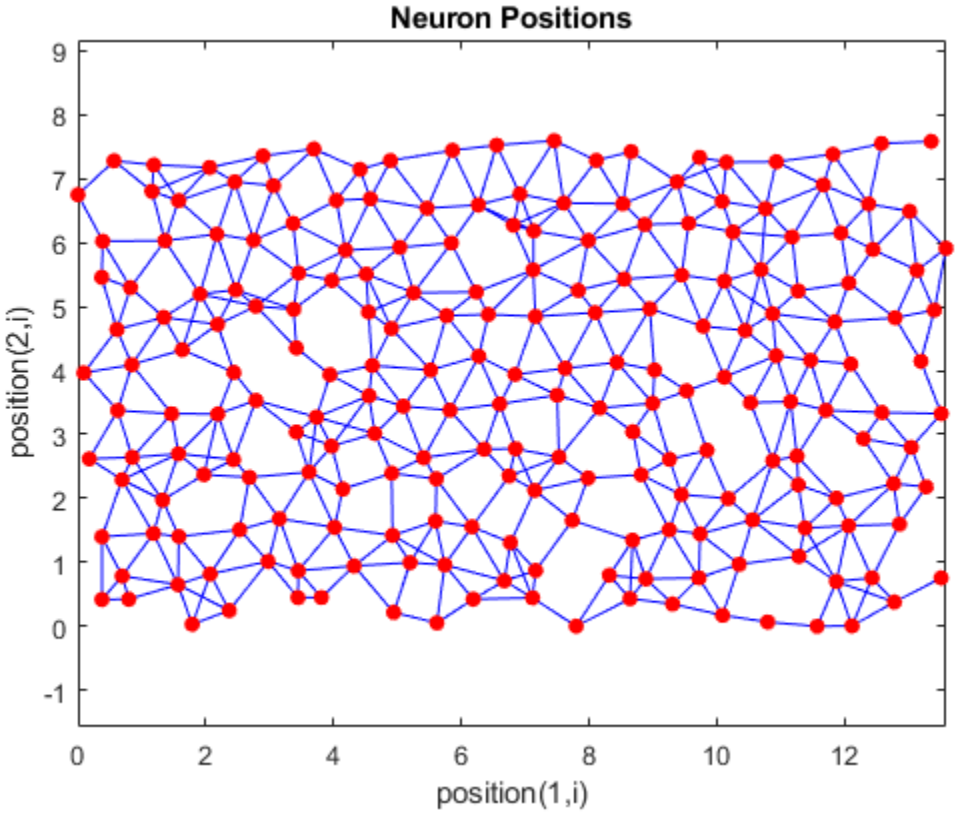
```
pos = hextop([5 6]);
plotsom(pos)
```



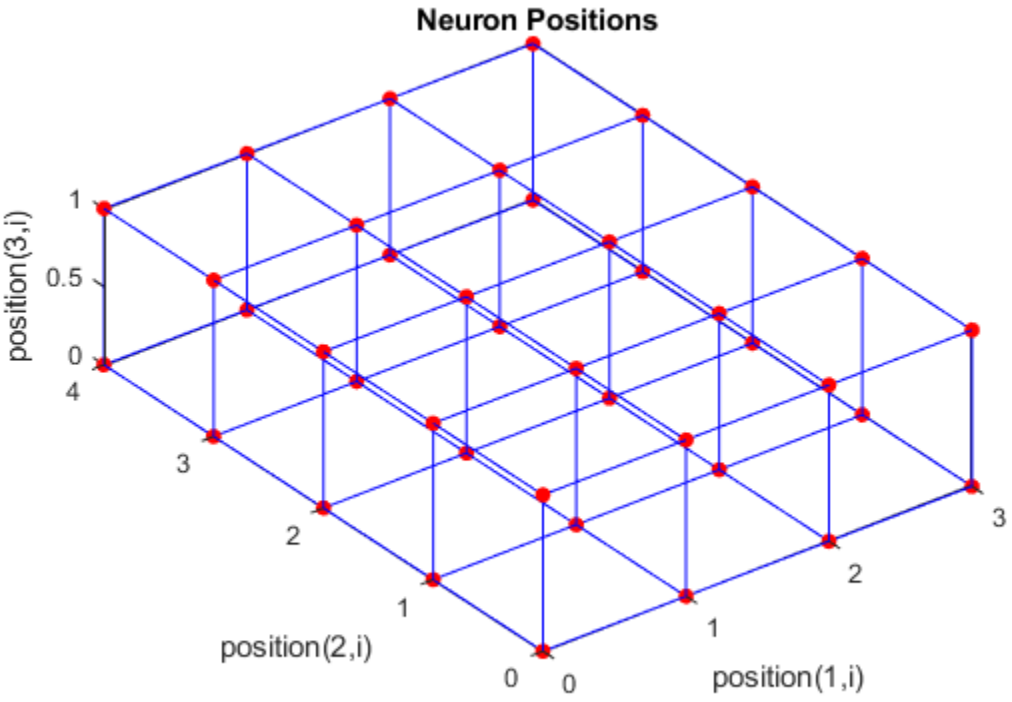

```
pos = gridtop([4 5]);
plotsom(pos)
```



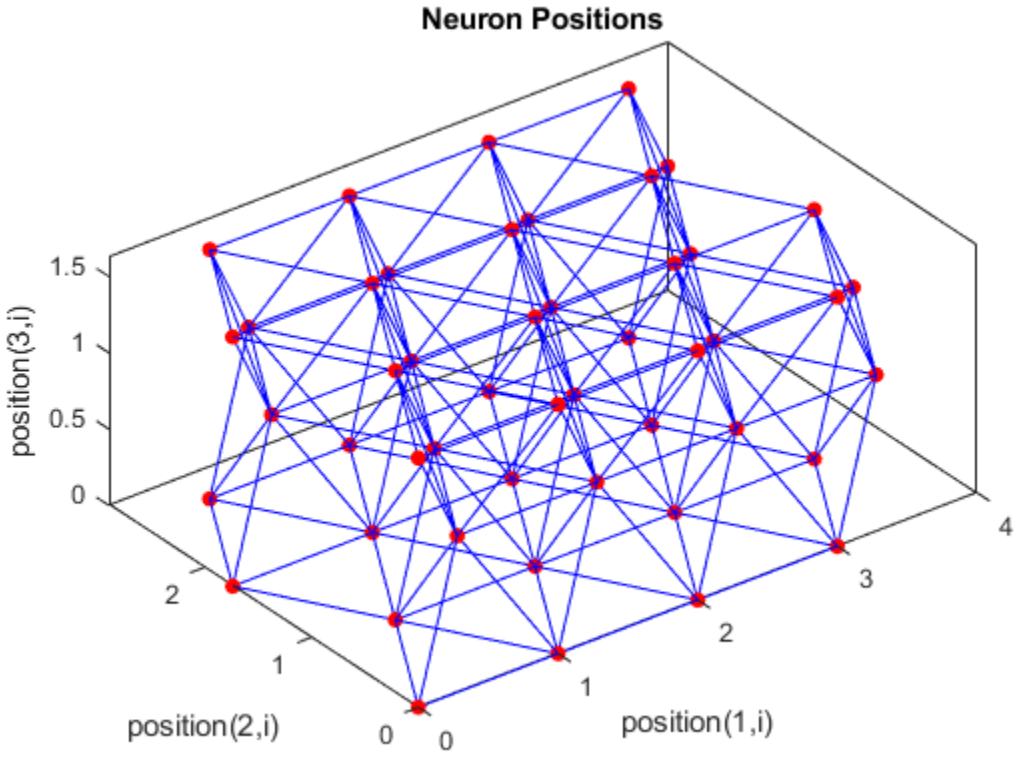

```
pos = randtop([18 12]);
plotsom(pos)
```




```
pos = gridtop([4 5 2]);
plotsom(pos)
```




```
pos = hextop([4 4 3]);
plotsom(pos)
```



See `plotsompos` for an example of plotting a layer's weight vectors with the input vectors they map.

See Also

`learnsom`

Introduced before R2006a

plotsomhits

Plot self-organizing map sample hits

Syntax

```
plotsomhits(net,inputs)
```

Description

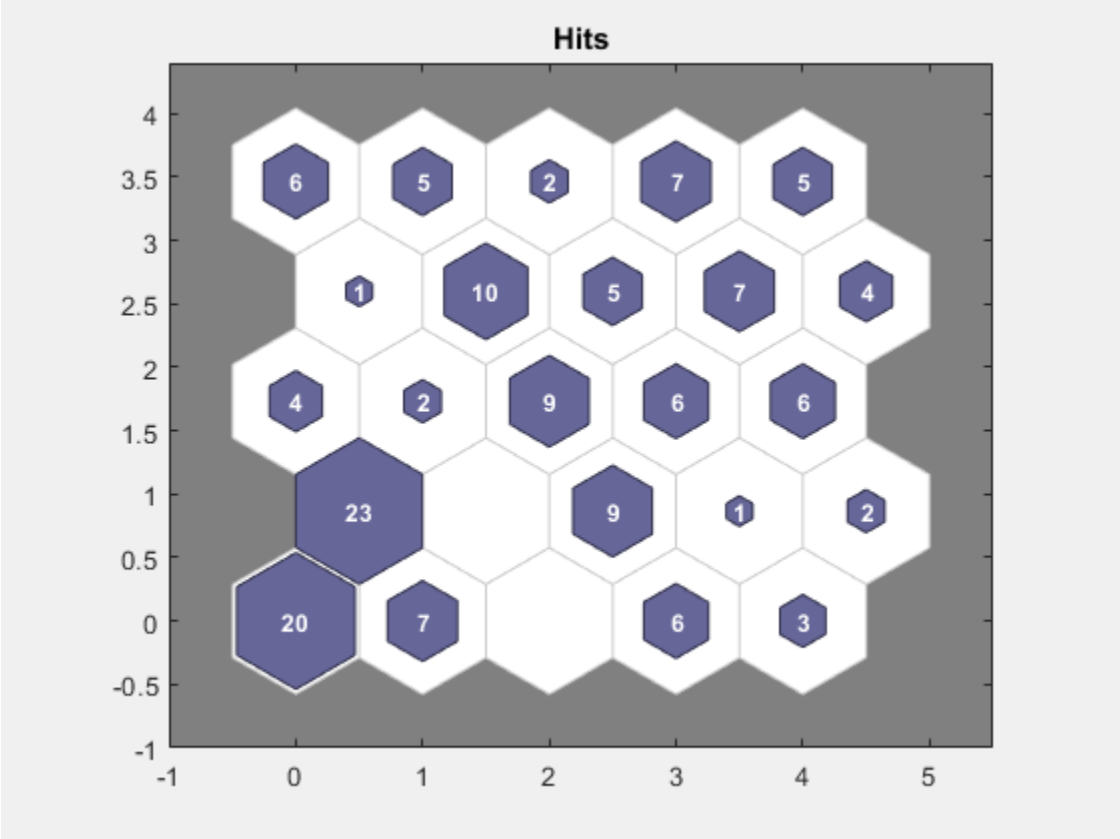
`plotsomhits(net,inputs)` plots a `SOM` layer, with each neuron showing the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples

Plot SOM Sample Hits

```
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomhits(net,x)
```



See Also

`plotsomplanes`

Introduced in R2008a

plotsomnc

Plot self-organizing map neighbor connections

Syntax

```
plotsomnc(net)
```

Description

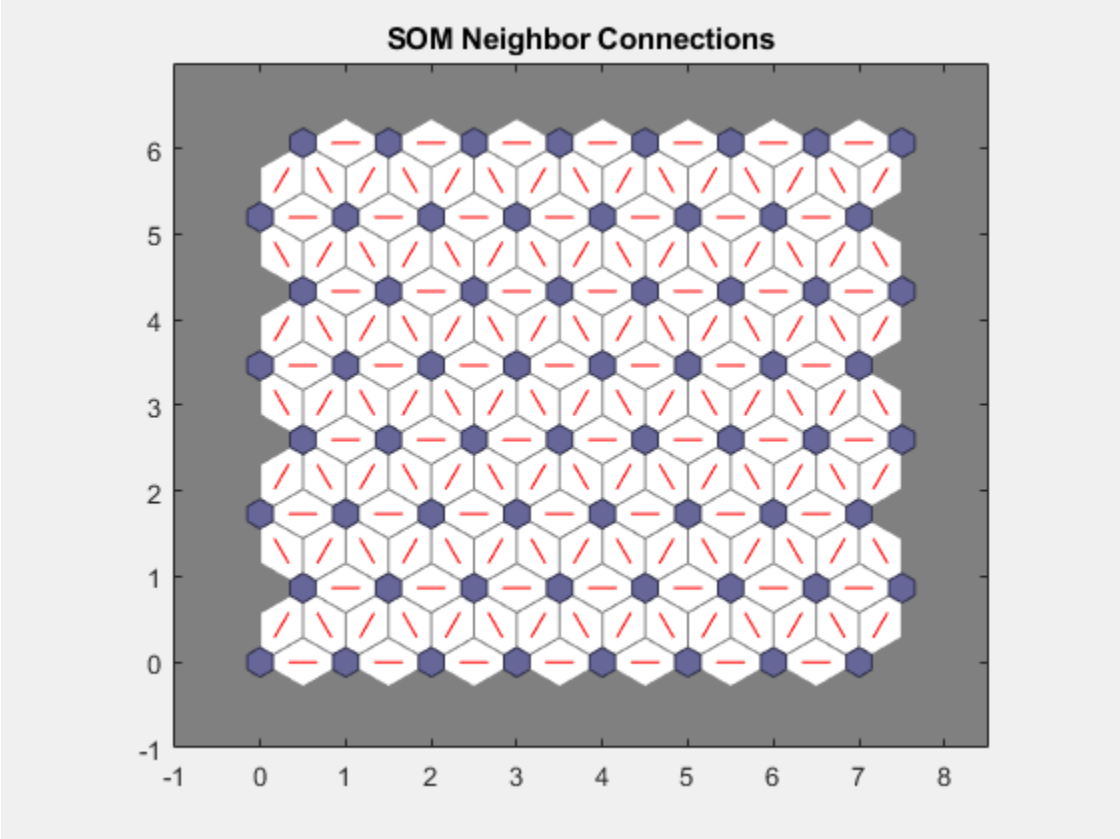
`plotsomnc(net)` plots a `SOM` layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples

Plot SOM Neighbor Connections

```
x = iris_dataset;
net = selforgmap([8 8]);
net = train(net,x);
plotsomnc(net)
```



See Also

`plotsomnd` | `plotsomplanes` | `plotsomhits`

Introduced in R2008a

plotsomnd

Plot self-organizing map neighbor distances

Syntax

```
plotsomnd(net)
```

Description

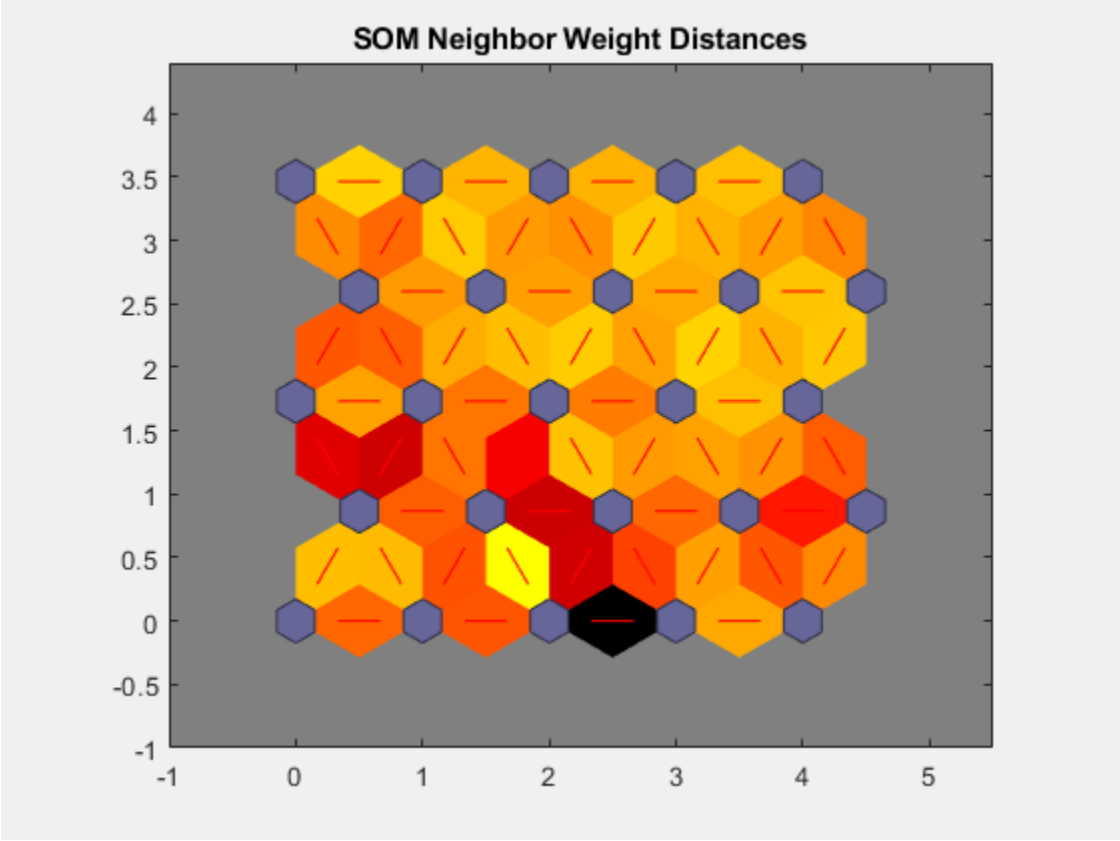
`plotsomnd(net)` plots a `SOM` layer showing neurons as gray-blue patches and their direct neighbor relations with red lines. The neighbor patches are colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples

Plot SOM Neighbor Distances

```
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomnd(net)
```



See Also

plotsomhits | plotsomnc | plotsomplanes

Introduced in R2008a

plotsomplanes

Plot self-organizing map weight planes

Syntax

```
plotsomplanes(net)
```

Description

`plotsomplanes(net)` generates a set of subplots. Each `i`th subplot shows the weights from the `i`th input to the layer's neurons, with the most negative connections shown as black, zero connections as red, and the strongest positive connections as yellow.

The plot is only shown for layers organized in one or two dimensions.

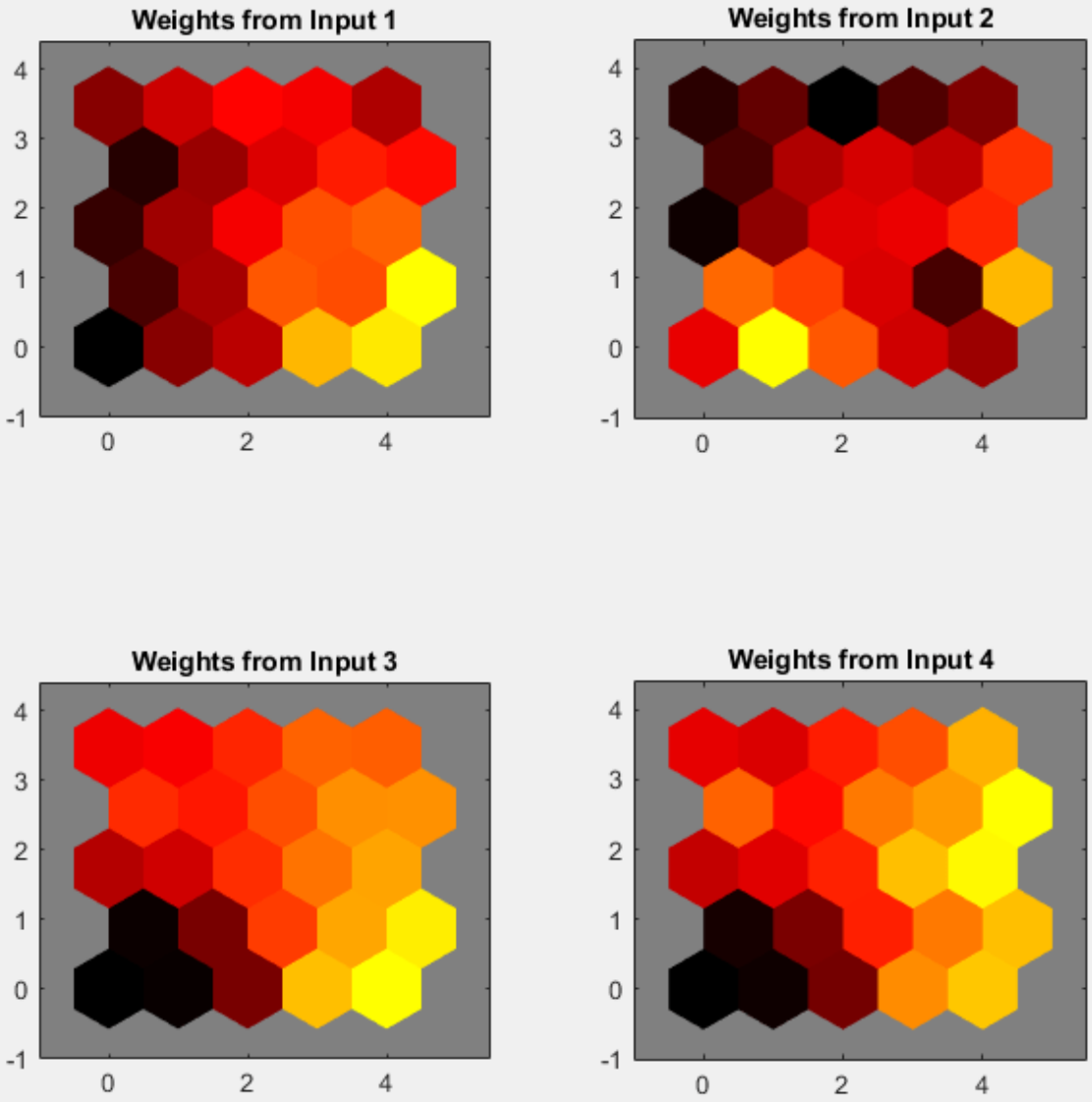
This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

This function can also be called with standardized plotting function arguments used by the function `train`.

Examples

Plot SOM Weight Planes

```
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomplanes(net)
```



See Also

`plotsomhits` | `plotsomnc` | `plotsomnd`

Introduced in R2008a

plotsompos

Plot self-organizing map weight positions

Syntax

```
plotsompos(net)
plotsompos(net,inputs)
```

Description

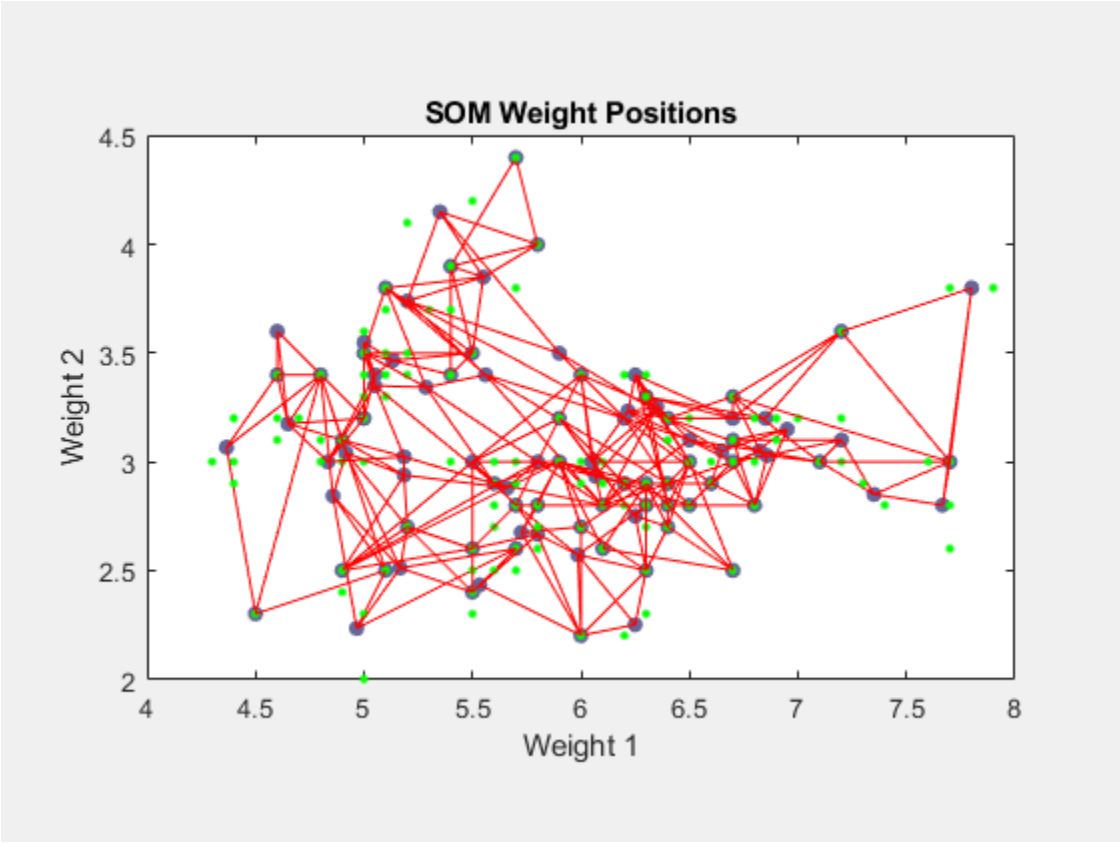
`plotsompos(net)` plots the input vectors as green dots and shows how the `SOM` classifies the input space by showing blue-gray dots for each neuron’s weight vector and connecting neighboring neurons with red lines.

`plotsompos(net,inputs)` plots the input data alongside the weights.

Examples

Plot SOM Weight Positions

```
x = iris_dataset;
net = selforgmap([10 10]);
net = train(net,x);
plotsompos(net,x)
```



See Also

`plotsomnd` | `plotsomplanes` | `plotsomhits`

Introduced in R2008a

plotsomtop

Plot self-organizing map topology

Syntax

```
plotsomtop(net)
```

Description

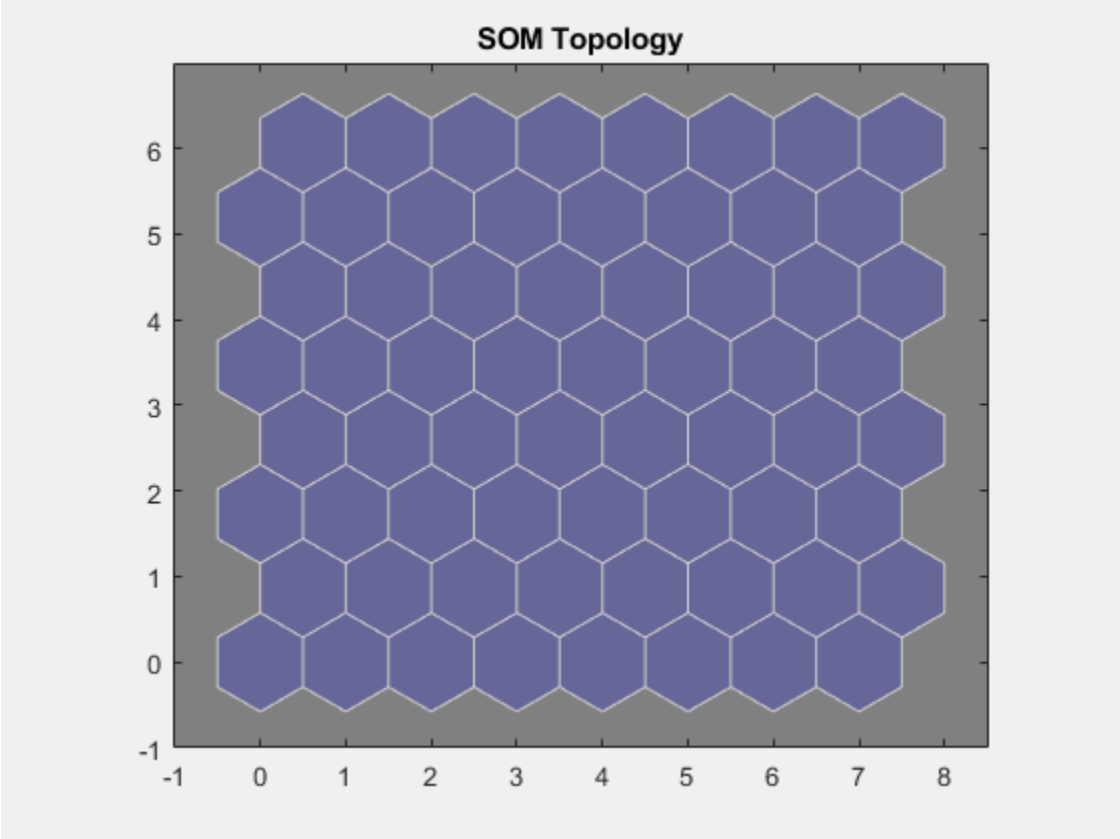
`plotsomtop(net)` plots the topology of a `SOM` layer.

This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples

Plot SOM Topology

```
x = iris_dataset;
net = selforgmap([8 8]);
plotsomtop(net)
```



See Also

`plotsomnd` | `plotsomplanes` | `plotsomhits`

Introduced in R2008a

plottrainstate

Plot training state values

Syntax

```
plottrainstate(tr)
```

Description

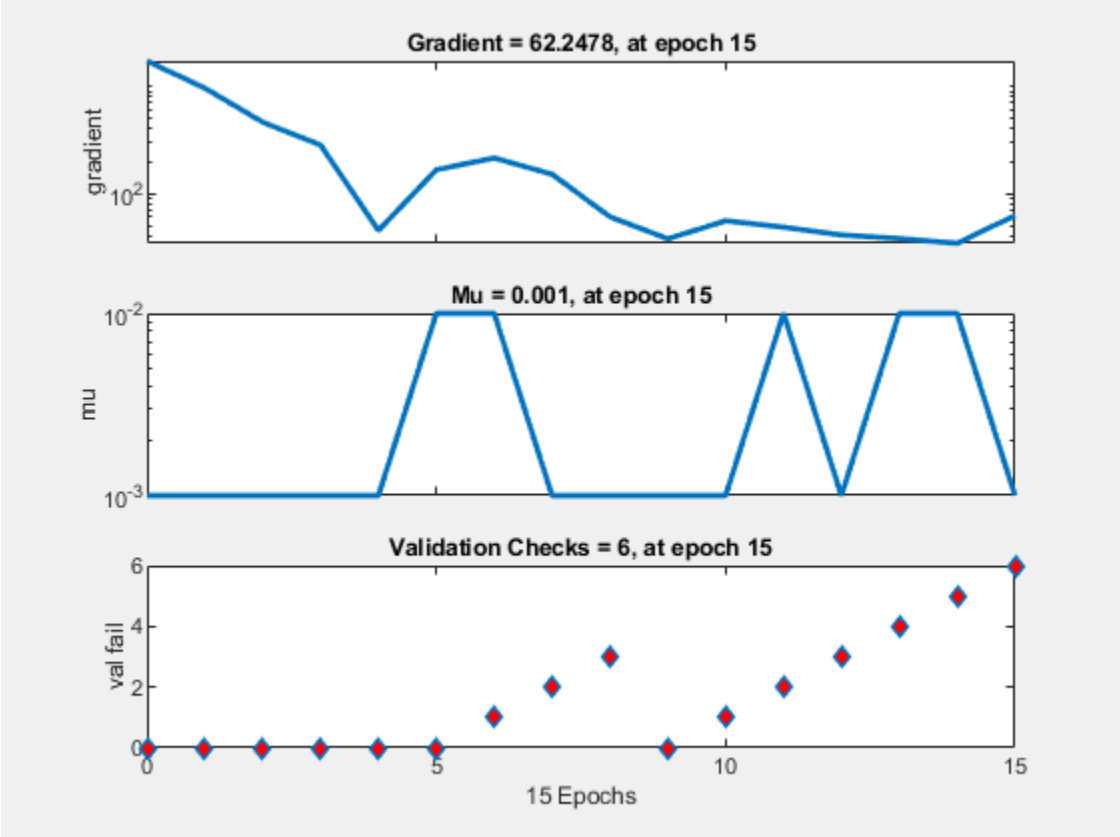
`plottrainstate(tr)` plots the training state from a training record `tr` returned by `train`.

Examples

Plot Training State Values

This example shows how to plot training state values using `plottrainstate`.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10);
[net, tr] = train(net, x, t);
plottrainstate(tr)
```



See Also

plotfit | plotperform | plotregression

Introduced in R2008a

plotv

Plot vectors as lines from origin

Syntax

```
plotv(M,T)
```

Description

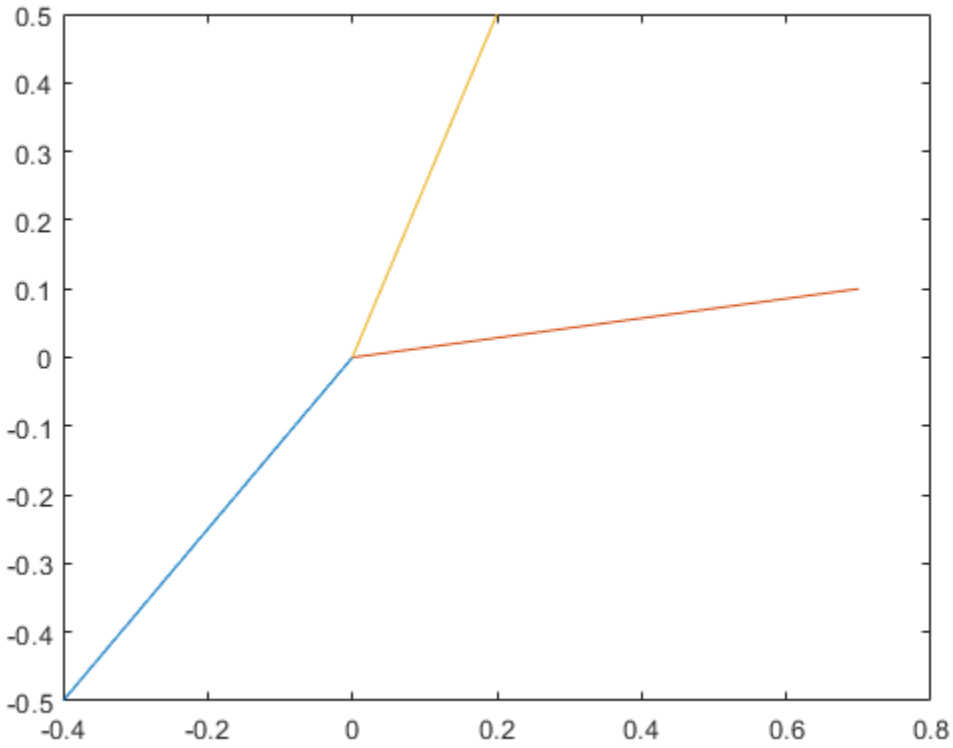
`plotv(M,T)` takes a matrix of column vectors, `M`, and the line plotting type, `T`, and plots the column vectors of `M`.

Examples

Plot Vectors Using the `plotv` Function

This example shows how to plot three 2-element vectors.

```
M = [-0.4 0.7 0.2 ;
     -0.5 0.1 0.5];
plotv(M,'-')
```



Input Arguments

`M` — Matrix to plot

matrix

Matrix of column vectors to plot, specified as a `R`-by-`Q` matrix of `Q` column vectors with `R` elements.

`R` must be 2 or greater. If `R` is greater than 2, this function only uses the first two rows of `M` for the plot.

`T` — Line plotting type

character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the nodes and no edges between them.

Example: `'--or'` is a red dashed line with circle markers.

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
`y`	Yellow
`m`	Magenta

Color	Description
c	Cyan
r	Red
g	Green
b	Blue
w	White
k	Black

See Also

`plotvec` | `plotfit`

Introduced before R2006a

plotvec

Plot vectors with different colors

Syntax

```
plotvec(X,C,M)
```

Description

`plotvec(X,C,M)` takes these inputs,

`X`	Matrix of (column) vectors
`C`	Row vector of color coordinates
`M`	Marker (default = `'+'`)

and plots each `i`th vector in `X` with a marker `M`, using the `i`th value in `C` as the color coordinate.

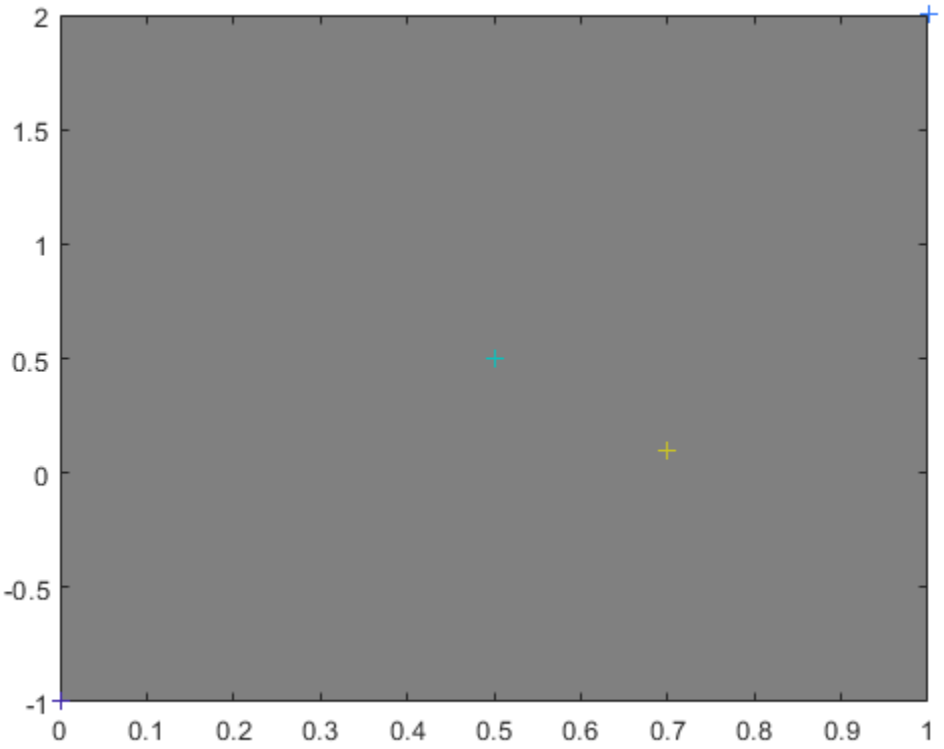
`plotvec(X)` only takes a matrix `X` and plots each `i`th vector in `X` with marker `'+'` using the index `i` as the color coordinate.

Examples

Plot Vectors with Different Colors

This example shows how to plot four 2-element vectors.

```
x = [ 0 1 0.5 0.7 ; ...
     -1 2 0.5 0.1];
c = [1 2 3 4];
plotvec(x,c)
```



Introduced before R2006a

plotwb

Plot Hinton diagram of weight and bias values

Syntax

```
plotwb(net)
plotwb(IW,LW,B)
plotwb(...,'toLayers',toLayers)
plotwb(...,'fromInputs',fromInputs)
plotwb(...,'fromLayers',fromLayers)
plotwb(...,'root',root)
```

Description

`plotwb(net)` takes a neural network and plots all its weights and biases.

`plotwb(IW,LW,B)` takes a neural networks input weights, layer weights and biases and plots them.

`plotwb(...,'toLayers',toLayers)` optionally defines which destination layers whose input weights, layer weights and biases will be plotted.

`plotwb(...,'fromInputs',fromInputs)` optionally defines which inputs will have their weights plotted.

`plotwb(...,'fromLayers',fromLayers)` optionally defines which layers will have weights coming from them plotted.

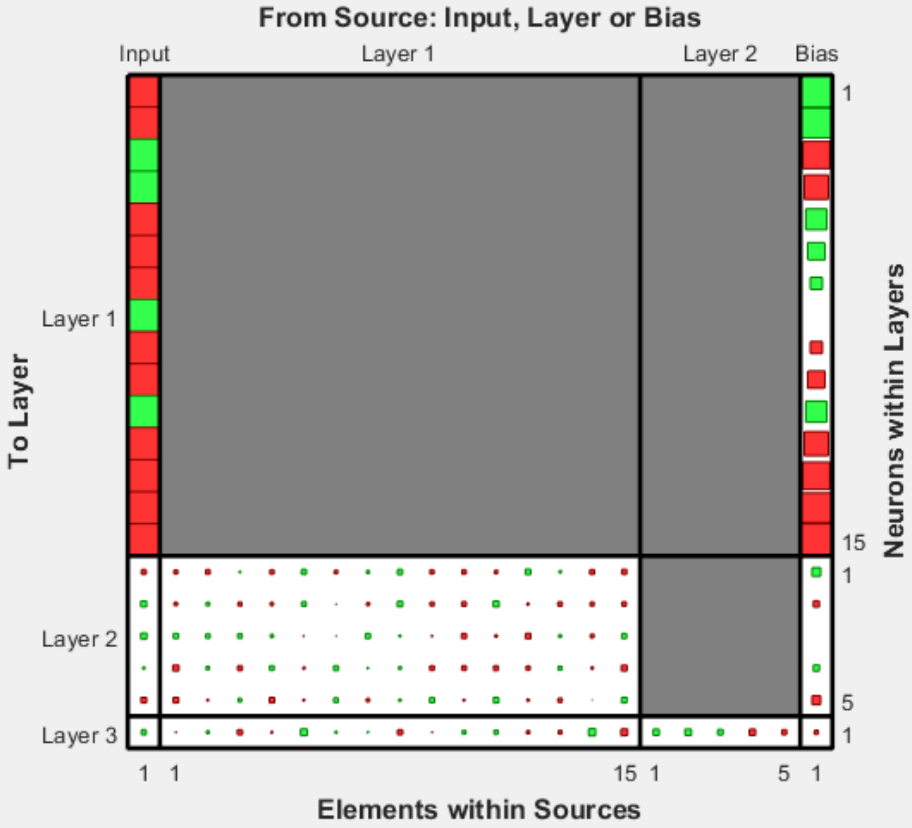
`plotwb(...,'root',root)` optionally defines the root used to scale the weight/bias patch sizes. The default is 2, which makes the 2-dimensional patch sizes scale directly with absolute weight and bias sizes. Larger values of root magnify the relative patch sizes of smaller weights and biases, making differences in smaller values easier to see.

Examples

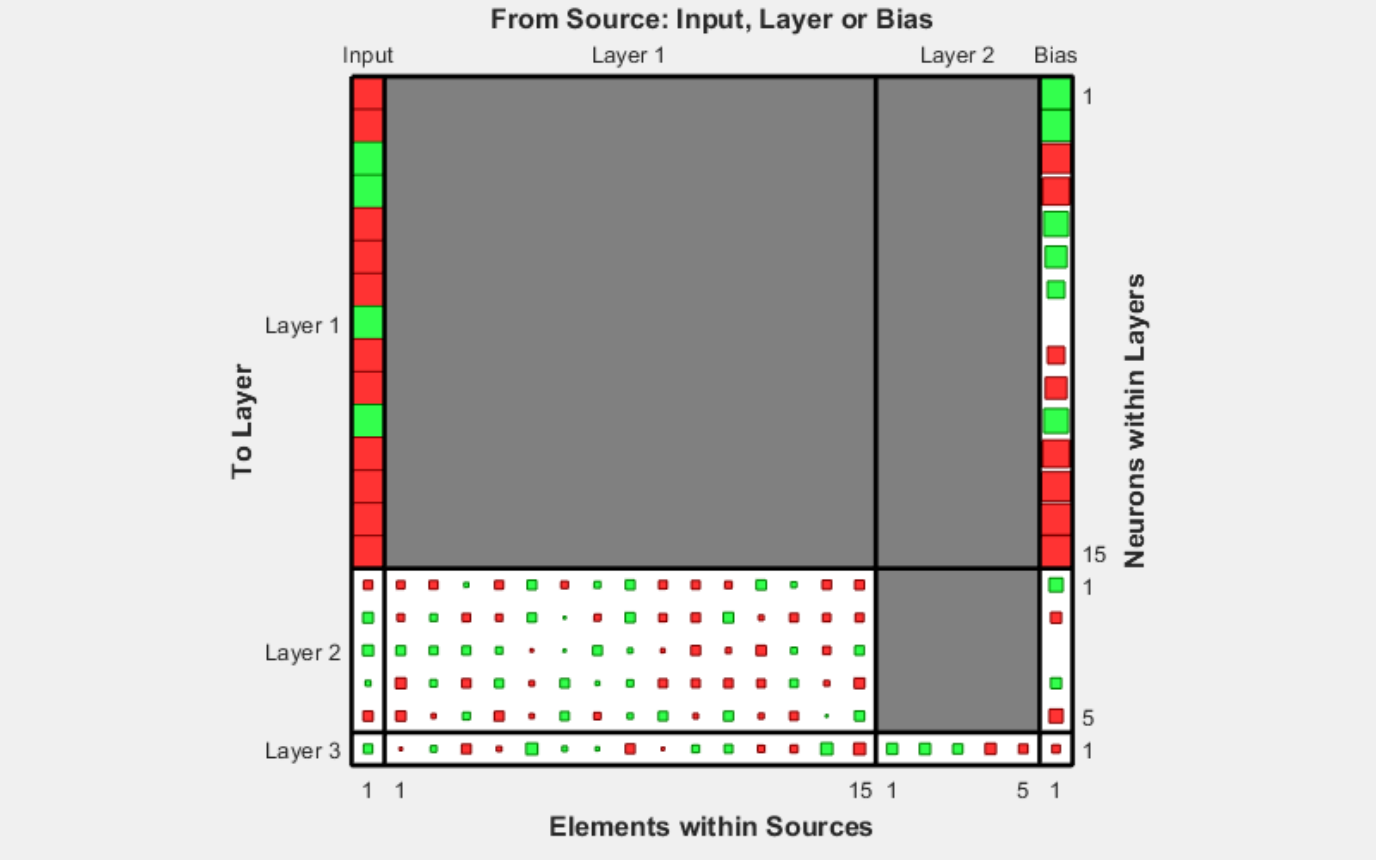
Plot Weights and Biases

Here a cascade-forward network is configured for particular data and its weights and biases are plotted in several ways.

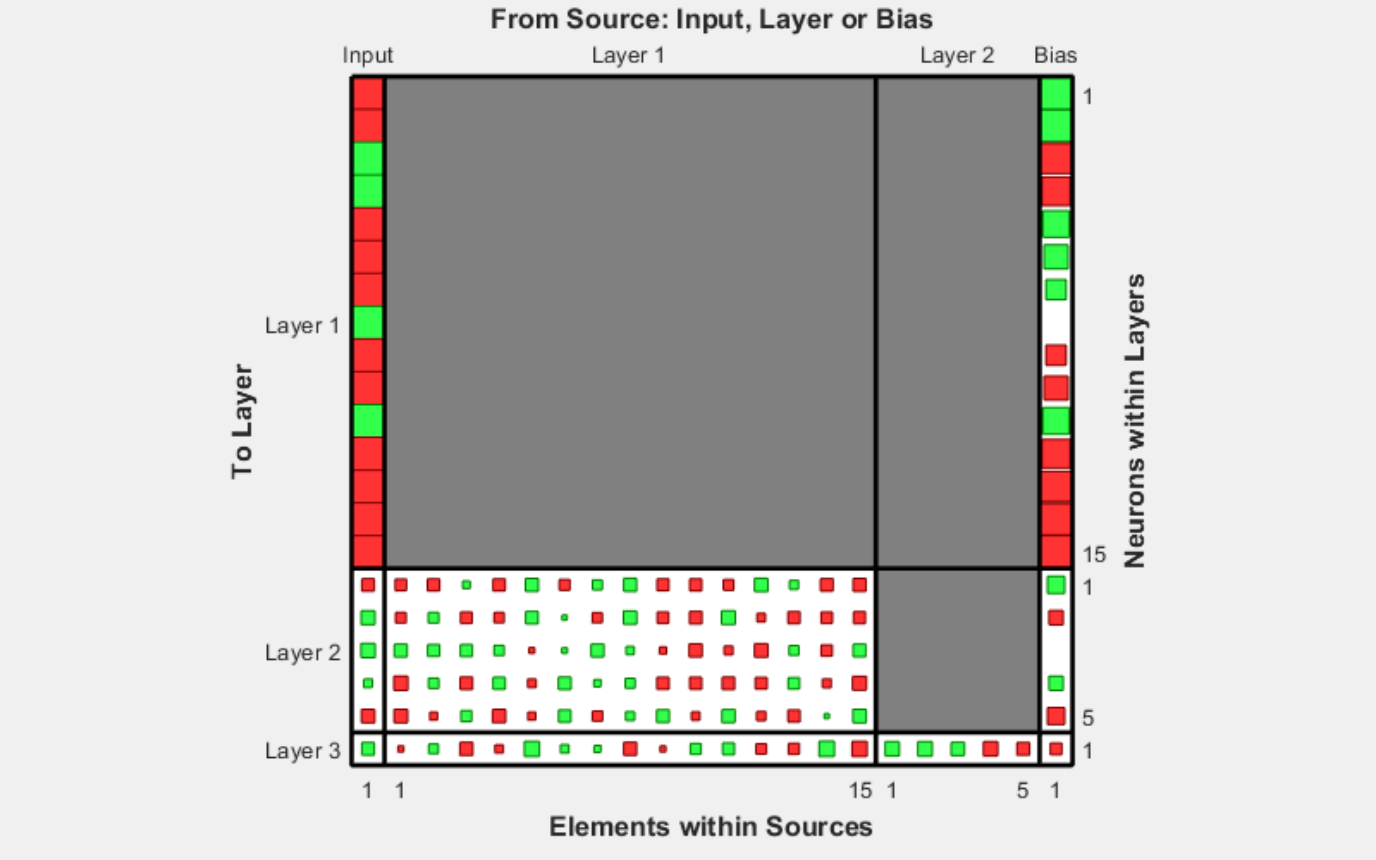
```
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
```



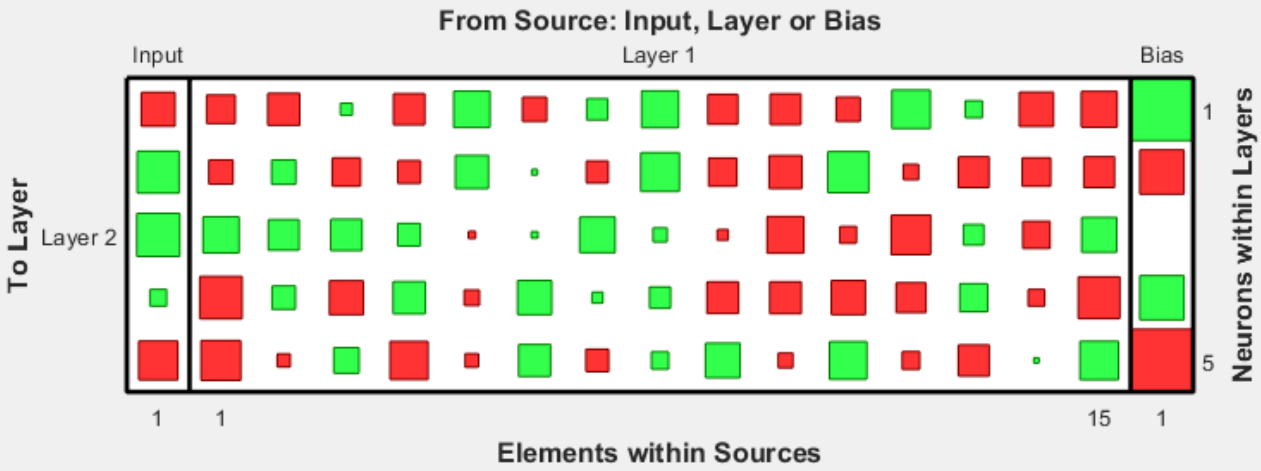

```
plotwb(net,'root',3)
```



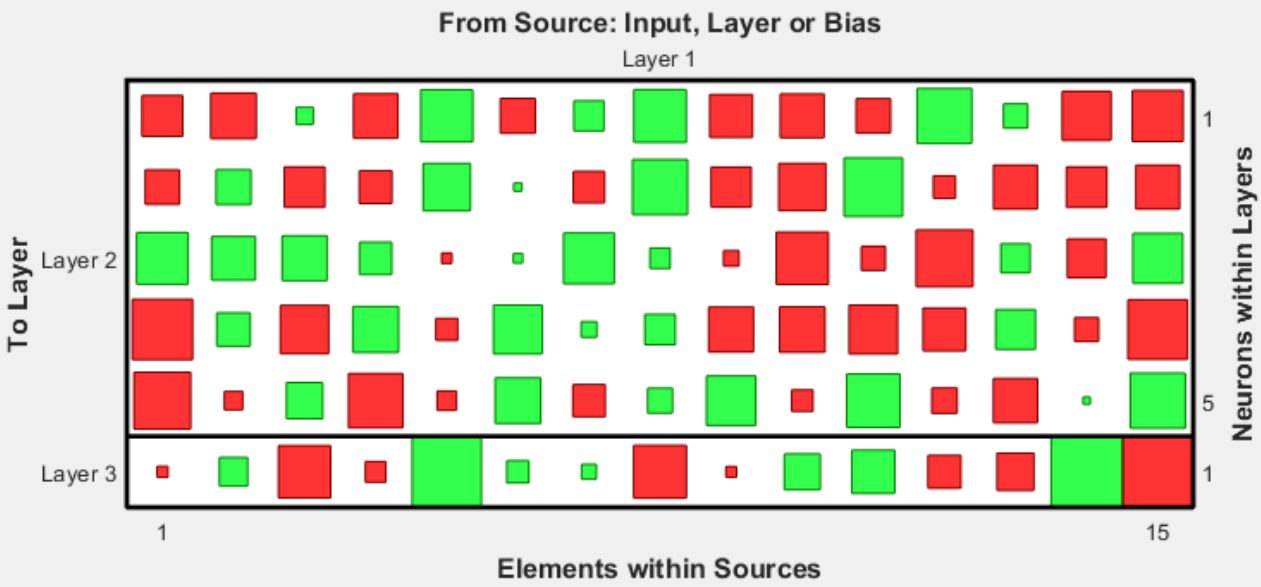

```
plotwb(net,'root',4)
```



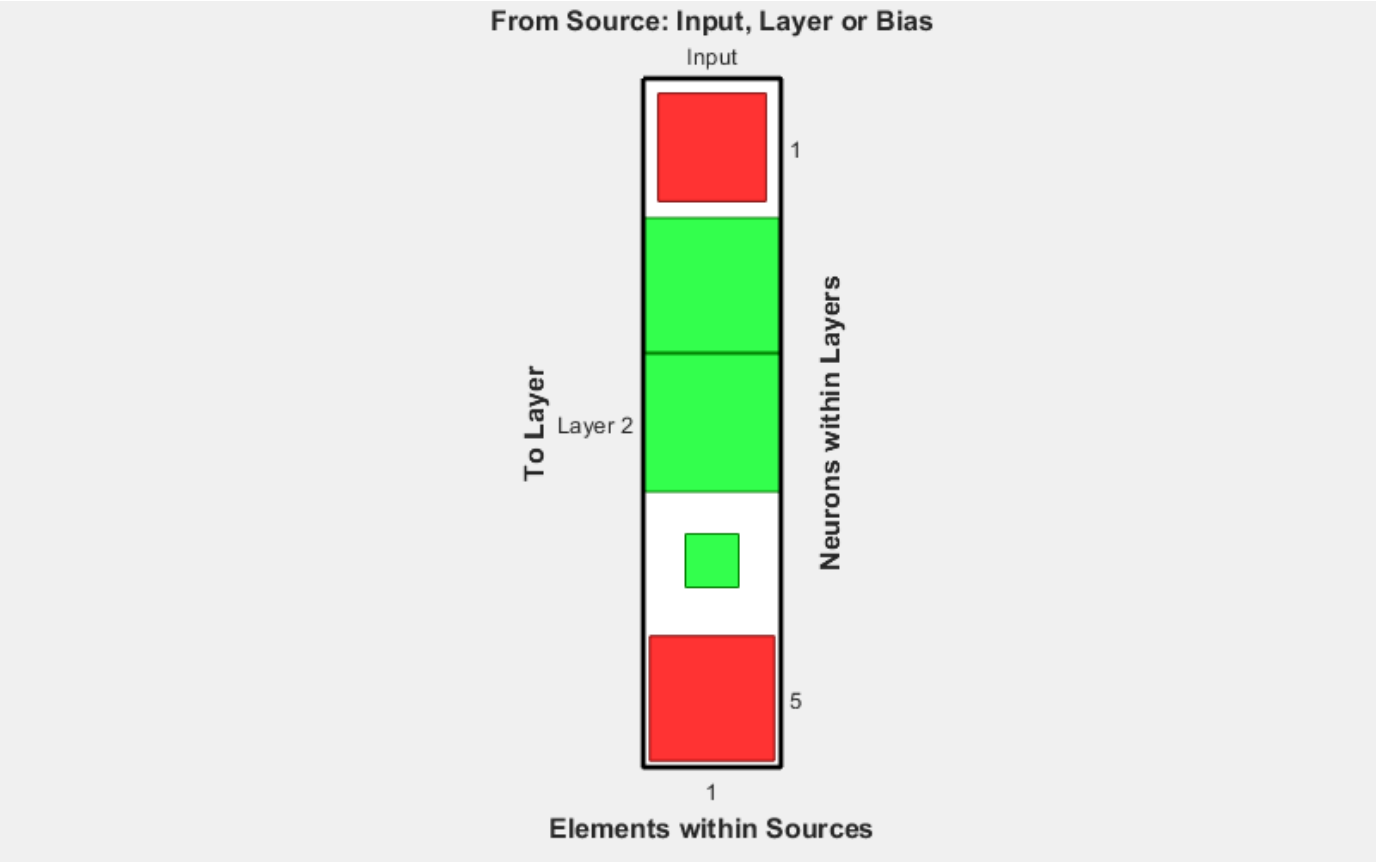

```
plotwb(net,'toLayers',2)
```




```
plotwb(net,'fromLayers',1)
```




```
plotwb(net,'toLayers',2,'fromInputs',1)
```



See Also

`plotsomplanes`

Introduced in R2010b

pnormc

Pseudonormalize columns of matrix

Syntax

```
pnormc(X,R)
```

Description

`pnormc(X,R)` takes these arguments,

X	M-by-N matrix
R	(Optional) radius to normalize columns to (default = 1)

and returns X with an additional row of elements, which results in new column vector lengths of R.

Caution For this function to work properly, the columns of X must originally have vector lengths less than R.

Examples

```
x = [0.1 0.6; 0.3 0.1];
y = pnormc(x)
```

See Also

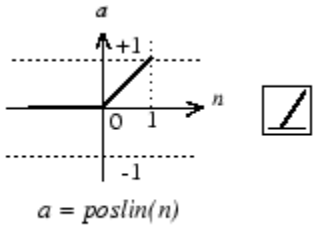
`normc` | `normr`

Introduced before R2006a

poslin

Positive linear transfer function

Graph and Symbol



Positive Linear Transfer Function

Syntax

```
A = poslin(N,FP)
info = poslin('code')
```

Description

`poslin` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = poslin(N,FP)` takes `N` and optional function parameters,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the S-by-Q matrix of `N`'s elements clipped to `[0, inf]`.

`info = poslin('code')` returns information about this function. The following codes are supported:

`poslin('name')` returns the name of this function.

`poslin('output',FP)` returns the `[min max]` output range.

`poslin('active',FP)` returns the `[min max]` active range.

`poslin('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`poslin('fpnames')` returns the names of the function parameters.

`poslin('fpdefaults')` returns the default function parameters.

Examples

Here is the code to create a plot of the `poslin` transfer function.

```
n = -5:0.1:5;
a = poslin(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'poslin';
```

Network Use

To change a network so that a layer uses `poslin`, set `net.layers{i}.transferFcn` to `'poslin'`.

Call `sim` to simulate the network with `poslin`.

Algorithms

The transfer function `poslin` returns the output `n` if `n` is greater than or equal to zero and 0 if `n` is less than or equal to zero.

```
poslin(n) = n, if n >= 0
          = 0, if n <= 0
```

See Also

`sim` | `purelin` | `satlin` | `satlins`

Introduced before R2006a

preparets

Prepare input and target time series data for network simulation or training

Syntax

```
[Xs,Xi,Ai,Ts,EWs,shift] = preparets(net,Xnf,Tnf,Tf,EW)
```

Description

`[Xs,Xi,Ai,Ts,EWs,shift] = preparets(net,Xnf,Tnf,Tf,EW)` takes these arguments:

- `net` — Neural network
- `Xnf` — Non-feedback inputs
- `Tnf` — Non-feedback targets
- `Tf` — Feedback targets
- `EW` — Error weights (optional)

and returns these arguments:

- `Xs` — Shifted inputs
- `Xi` — Initial input delay states
- `Ai` — Initial layer delay states
- `Ts` — Shifted targets
- `EWs` — Shifted error weights
- `shift` — The number of timesteps truncated from the front of `X` and `T` in order to properly fill `Xi` and `Ai`.

This function simplifies the normally complex and error prone task of reformatting input and target time series. It automatically shifts input and target time series as many steps as are needed to fill the initial input and layer delay states. If the network has open-loop feedback, then it copies feedback targets into the inputs as needed to define the open-loop inputs.

Each time a new network is designed, with different numbers of delays or feedback settings, `preparets` can reformat input and target data accordingly. Also, each time a network is transformed with `openloop`, `closeloop`, `removedelay` or `adddelay`, this function can reformat the data accordingly.

Examples

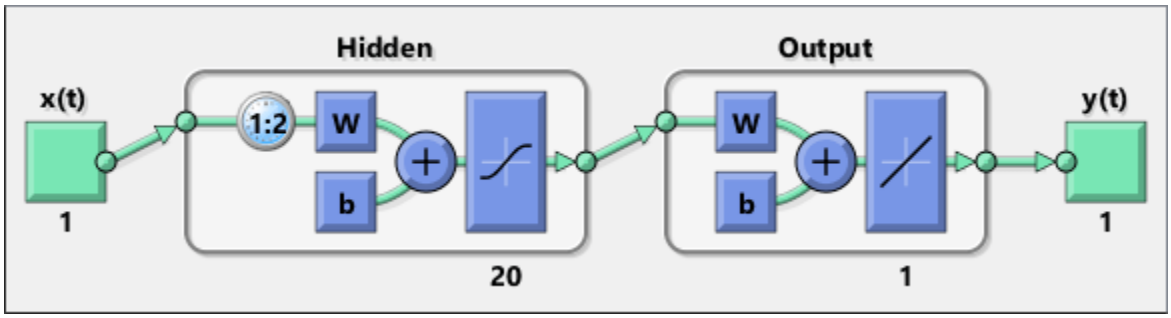
Prepare Data for Open- and Closed-Loop Networks

This example shows how to prepare data for open-loop and closed-loop networks.

Create a time-delay network with 20 hidden neurons, then train and simulate it.

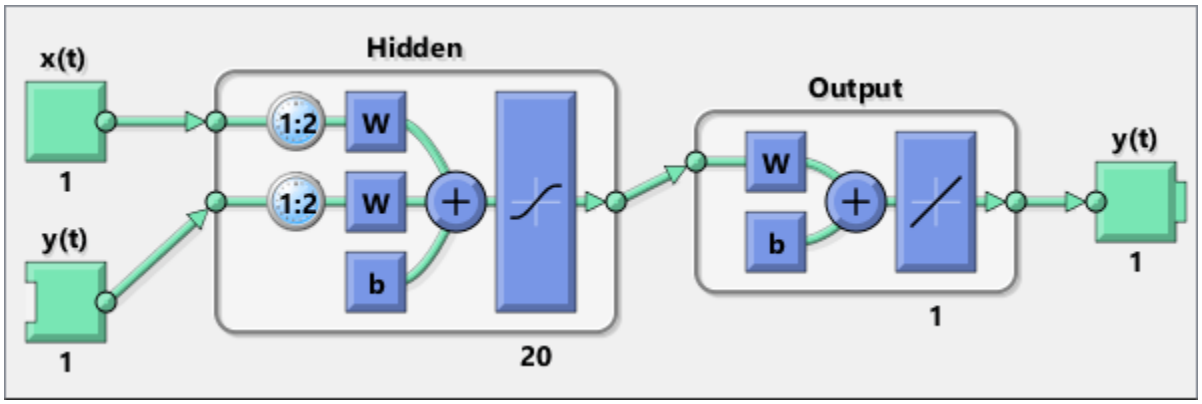
```
[X,T] = simpleseries_dataset;
net = timedelaynet(1:2,20);
```

```
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts);
view(net)
Y = net(Xs,Xi,Ai);
```



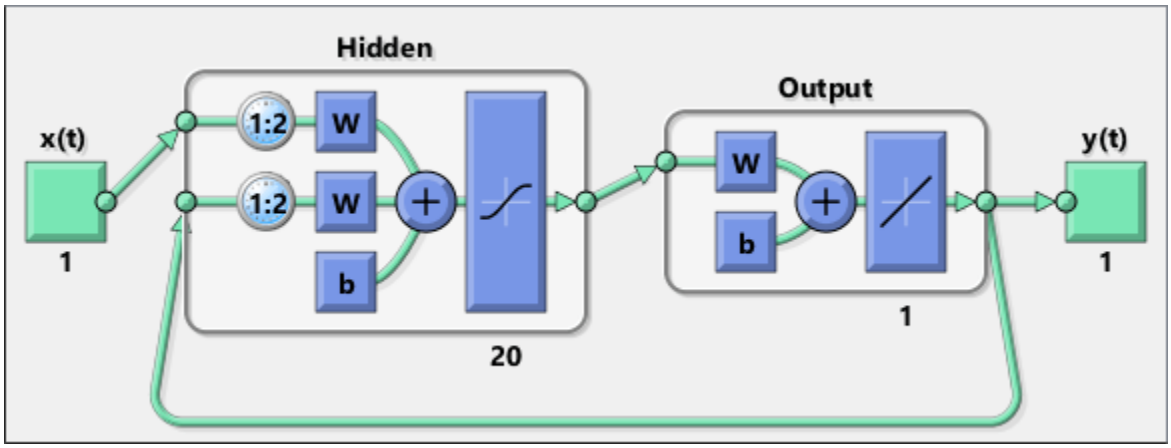
Design a NARX network. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
y = net(Xs,Xi,Ai);
```



Now convert the network to closed loop, and reformat the data to simulate the network's closed-loop response.

```
net = closeloop(net);
view(net)
[Xs,Xi,Ai] = preparets(net,X,{},T);
y = net(Xs,Xi,Ai);
```



Input Arguments

`net` — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

`Xnf` — Non-feedback inputs

cell array

Non-feedback input data (inputs not associated with an open loop feedback output), specified as cell array.

`Tnf` — Non-feedback targets

cell array

Target data for non-feedback outputs, specified as a cell array.

`Tf` — Feedback targets

cell array

Target data for outputs with feedback, specified as a cell array.

`EW` — Error weights

cell array

Error weights, specified as a cell array.

Output Arguments

`Xs` — Shifted inputs

cell array

Shifted inputs, returned as a cell array.

`Xi` — Initial input delay states

cell array

Initial input delay states, returned as cell array.

`Ai` — Initial layer delay states
cell array

Initial layer delay states, returned as a cell array.

`Ts` — Shifted targets
cell array

Shifted targets, returned as a cell array.

`EWs` — Shifted error weights
cell array

Shifted error weights, returned as a cell array.

`shift` — Timesteps
scalar

Number of timesteps truncated from the front of `X` and `T` in order to properly fill `Xi` and `Ai`, returned as a scalar.

See Also

`adddelay` | `closeloop` | `narnet` | `narxnet` | `openloop` | `removedelay` | `timedelaynet`

Introduced in R2010b

processpca

Process columns of matrix with principal component analysis

Syntax

```
[Y,PS] = processpca(X,maxfrac)
[Y,PS] = processpca(X,FP)
Y = processpca('apply',X,PS)
X = processpca('reverse',Y,PS)
name = processpca('name')
fp = processpca('pdefaults')
names = processpca('pdesc')
processpca('pcheck',fp);
```

Description

`processpca` processes matrices using principal component analysis so that each row is uncorrelated, the rows are in the order of the amount they contribute to total variation, and rows whose contribution to total variation are less than `maxfrac` are removed.

`[Y,PS] = processpca(X,maxfrac)` takes `X` and an optional parameter,

`X`	`N`-by-`Q` matrix
`maxfrac`	Maximum fraction of variance for removed rows (default is 0)

and returns

`Y`	`M`-by-`Q` matrix with `N - M` rows deleted
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = processpca(X,FP)` takes parameters as a struct: `FP.maxfrac`.

`Y = processpca('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = processpca('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`name = processpca('name')` returns the name of this process method.

`fp = processpca('pdefaults')` returns default process parameter structure.

`names = processpca('pdesc')` returns the process parameter descriptions.

`processpca('pcheck',fp);` throws an error if any parameter is illegal.

Examples

Here is how to format a matrix with an independent row, a correlated row, and a completely redundant row so that its rows are uncorrelated and the redundant row is dropped.

```
x1_independent = rand(1,5)
x1_correlated = rand(1,5) + x1_independent;
x1_redundant = x1_independent + x1_correlated
x1 = [x1_independent; x1_correlated; x1_redundant]
[y1,ps] = processpca(x1)
```

Next, apply the same processing settings to new values.

```
x2_independent = rand(1,5)
x2_correlated = rand(1,5) + x1_independent;
x2_redundant = x1_independent + x1_correlated
x2 = [x2_independent; x2_correlated; x2_redundant];
y2 = processpca('apply',x2,ps)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = processpca('reverse',y1,ps)
```

More About

Reduce Input Dimensionality Using `processpca`

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the components of the input vectors (so that they are uncorrelated with each other), it orders the resulting orthogonal components (principal components) so that those with the largest variation come first, and it eliminates those components that contribute the least to the variation in the data set. The following code illustrates the use of `processpca`, which performs a principal-component analysis using the processing setting `maxfrac of 0.02`.

```
[pn,ps1] = mapstd(p);
[ptrans,ps2] = processpca(pn,0.02);
```

The input vectors are first normalized, using `mapstd`, so that they have zero mean and unity variance. This is a standard procedure when using principal components. In this example, the second argument passed to `processpca` is 0.02. This means that `processpca` eliminates those principal components that contribute less than 2% to the total variation in the data set. The matrix `ptrans` contains the transformed input vectors. The settings structure `ps2` contains the principal component transformation matrix. After the network has been trained, these settings should be used to transform any future inputs that are applied to the network. It effectively becomes a part of the network, just like the network weights and biases. If you multiply the normalized input vectors `pn` by the transformation matrix `transMat`, you obtain the transformed input vectors `ptrans`.

If `processpca` is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the transformation matrix that was computed for the training set, using `ps2`. The following code applies a new set of inputs to a network already trained.

```
pnewn = mapstd('apply',pnew,ps1);
pnewtrans = processpca('apply',pnewn,ps2);
a = sim(net,pnewtrans);
```

Principal component analysis is not reliably reversible. Therefore it is only recommended for input processing. Outputs require reversible processing functions.

Principal component analysis is not part of the default processing for `feedforwardnet`. You can add this with the following command:

```
net.inputs{1}.processFcns{end+1} = 'processpca';
```

Algorithms

Values in rows whose elements are not all the same value are set to

```
y = 2*(x-minx)/(maxx-minx) - 1;
```

Values in rows with all the same value are set to 0.

See Also

`fixunknowns` | `mapminmax` | `mapstd`

Introduced in R2006a

prune

Delete neural inputs, layers, and outputs with sizes of zero

Syntax

```
[net,pi,pl,po] = prune(net)
```

Description

This function removes zero-sized inputs, layers, and outputs from a network. This leaves a network which may have fewer inputs and outputs, but which implements the same operations, as zero-sized inputs and outputs do not convey any information.

One use for this simplification is to prepare a network with zero sized subobjects for Simulink, where zero sized signals are not supported.

The companion function `prunedata` can prune data to remain consistent with the transformed network.

`[net,pi,pl,po] = prune(net)` takes a neural network and returns

`net`	The same network with zero-sized subobjects removed
`pi`	Indices of pruned inputs
`pl`	Indices of pruned layers
`po`	Indices of pruned outputs

Examples

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.

```
net = narxnet(20);
view(net)
```

The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

```
X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);
```

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

```
[net2,pi,pl,po] = prune(net);
view(net2)
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts);
[sysName,netName] = gensim(net2);
setsiminit(sysName,netName,net2,Xi2,Ai2);
```

See Also

`prunedata` | `gensim`

Introduced in R2010b

prunedata

Prune data for consistency with pruned network

Syntax

```
[Xp,Xip,Aip,Tp] = prunedata(net,pi,pl,po,X,Xi,Ai,T)
```

Description

This function prunes data to be consistent with a network whose zero-sized inputs, layers, and outputs have been removed with `prune`.

One use for this simplification is to prepare a network with zero-sized subobjects for Simulink, where zero-sized signals are not supported.

`[Xp,Xip,Aip,Tp] = prunedata(net,pi,pl,po,X,Xi,Ai,T)` takes these arguments,

`net`	Pruned neural network
`pi`	Indices of pruned inputs
`pl`	Indices of pruned layers
`po`	Indices of pruned outputs
`X`	Input data
`Xi`	Initial input delay states
`Ai`	Initial layer delay states
`T`	Target data

and returns the pruned inputs, input and layer delay states, and targets.

Examples

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.

```
net = narxnet(20);
view(net)
```

The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

```
X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);
```

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

```
[net2,pi,pl,po] = prune(net);
view(net2)
```

```
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts);
[sysName,netName] = gensim(net2);
setsiminit(sysName,netName,net2,Xi2,Ai2);
```

See Also

prune | gensim

Introduced in R2010b

purelin

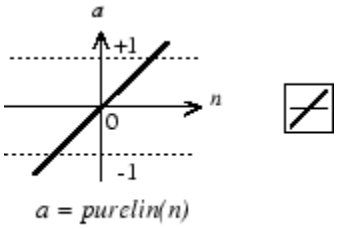
Linear transfer function

Syntax

```
A = purelin(N)
info = purelin('code')
```

Description

`A = purelin(N)` takes an `S`-by-`Q` matrix of net input (column) vectors, `N`, and returns an `S`-by-`Q` matrix equal to `N`, `A`.



Linear Transfer Function

`info = purelin('code')` returns useful information for each `code` character vector:

- `purelin('name')` returns the name of this function.
- `purelin('output')` returns the `[min max]` output range.
- `purelin('active')` returns the `[min max]` active input range.
- `purelin('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S`-by-`S`-by-`Q` or `S`-by-`Q`.
- `purelin('fpnames')` returns the names of the function parameters.
- `purelin('fpdefaults')` returns the default function parameters.

Examples

Create a purelin Transfer Function and Assign It to a Layer in a Network

This example shows how to create and plot a `purelin` transfer function and assign it to layer `i` in a network.

Create a plot of the `purelin` transfer function:

```
n = -5:0.1:5;
a = purelin(n);
plot(n,a)
```

Assign this transfer function to layer `i` in a network.

```
net.layers{i}.transferFcn = 'purelin';
```

Input Arguments

N — Net inputs

matrix

Net column vector inputs, specified as an`S`-by-`Q` matrix.

Output Arguments

A — Linear transfer function

matrix

Linear transfer function, returned as an `S`-by-`Q` matrix.

Algorithms

```
a = purelin(n) = n
```

See Also

`sim` | `satlin` | `satlins`

Introduced before R2006a

quant

Discretize values as multiples of quantity

Syntax

```
quant(X,Q)
```

Description

`quant(X,Q)` takes two inputs,

`X`	Matrix, vector, or scalar
`Q`	Minimum value

and returns values from `X` rounded to nearest multiple of `Q`.

Examples

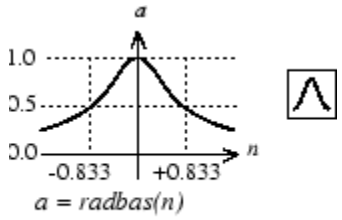
```
x = [1.333 4.756 -3.897];
y = quant(x,0.1)
```

Introduced before R2006a

radbas

Radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

```
A = radbas(N,FP)
```

Description

`radbas` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = radbas(N,FP)` takes one or two inputs,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, an `S`-by-`Q` matrix of the radial basis function applied to each element of `N`.

Examples

Here you create a plot of the `radbas` transfer function.

```
n = -5:0.1:5;
a = radbas(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'radbas';
```

Algorithms

```
a = radbas(n) = exp(-n^2)
```

See Also

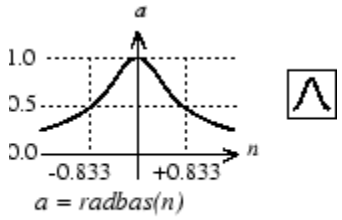
`sim` | `radbasn` | `tribas`

Introduced before R2006a

radbasn

Normalized radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

```
A = radbasn(N,FP)
```

Description

`radbasn` is a neural transfer function. Transfer functions calculate a layer's output from its net input. This function is equivalent to radbas, except that output vectors are normalized by dividing by the sum of the pre-normalized values.

`A = radbasn(N,FP)` takes one or two inputs,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, an `S`-by-`Q` matrix of the radial basis function applied to each element of `N`.

Examples

Here six random 3-element vectors are passed through the radial basis transform and normalized.

```
n = rand(3,6)
a = radbasn(n)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'radbasn';
```

Algorithms

```
a = radbasn(n) = exp(-n^2) / sum(exp(-n^2))
```

See Also

`sim` | `radbas` | `tribas`

Introduced in R2010b

randnc

Normalized column weight initialization function

Syntax

```
W = randnc(S,PR)
```

Description

`randnc` is a weight initialization function.

`W = randnc(S,PR)` takes two inputs,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R random matrix with normalized columns.

You can also call this in the form `randnc(S,R)`.

Examples

A random matrix of four normalized three-element columns is generated:

```
M = randnc(3,4)
M =
    -0.6007   -0.4715   -0.2724    0.5596
    -0.7628   -0.6967   -0.9172    0.7819
    -0.2395    0.5406   -0.2907    0.2747
```

See Also

`randnr`

Introduced before R2006a

randnr

Normalized row weight initialization function

Syntax

```
W = randnr(S,PR)
```

Description

`randnr` is a weight initialization function.

`W = randnr(S,PR)` takes two inputs,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R random matrix with normalized rows.

You can also call this in the form `randnr(S,R)`.

Examples

A matrix of three normalized four-element rows is generated:

```
M = randnr(3,4)
M =
    0.9713    0.0800   -0.1838   -0.1282
    0.8228    0.0338    0.1797    0.5381
   -0.3042   -0.5725    0.5436    0.5331
```

See Also

`randnc`

Introduced before R2006a

rands

Symmetric random weight/bias initialization function

Syntax

```
W = rands(S,PR)
M = rands(S,R)
v = rands(S)
```

Description

`rands` is a weight/bias initialization function.

`W = rands(S,PR)` takes

S	Number of neurons
PR	R-by-2 matrix of R input ranges

and returns an S-by-R weight matrix of random values between –1 and 1.

`M = rands(S,R)` returns an S-by-R matrix of random values. `v = rands(S)` returns an S-by-1 vector of random values.

Examples

Here, three sets of random values are generated with `rands`.

```
rands(4,[0 1; -2 2])
rands(4)
rands(2,3)
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `rands`,

1. Set `net.initFcn` to `'initlay'`. (`net.initParam` automatically becomes `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set each `net.inputWeights{i,j}.initFcn` to `'rands'`.
4. Set each `net.layerWeights{i,j}.initFcn` to `'rands'`.
5. Set each `net.biases{i}.initFcn` to `'rands'`.

To initialize the network, call `init`.

See Also

`randsmall` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

Introduced before R2006a

randsmall

Small random weight/bias initialization function

Syntax

```
W = randsmall(S,PR)
M = rands(S,R)
v = rands(S)
```

Description

`randsmall` is a weight/bias initialization function.

`W = randsmall(S,PR)` takes

S	Number of neurons
PR	R-by-2 matrix of R input ranges

and returns an S-by-R weight matrix of small random values between –0.1 and 0.1.

`M = rands(S,R)` returns an S-by-R matrix of random values. `v = rands(S)` returns an S-by-1 vector of random values.

Examples

Here three sets of random values are generated with `rands`.

```
randsmall(4,[0 1; -2 2])
randsmall(4)
randsmall(2,3)
```

Network Use

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `rands`,

1. Set `net.initFcn` to `'initlay'`. (`net.initParam` automatically becomes `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set each `net.inputWeights{i,j}.initFcn` to `'randsmall'`.
4. Set each `net.layerWeights{i,j}.initFcn` to `'randsmall'`.
5. Set each `net.biases{i}.initFcn` to `'randsmall'`.

To initialize the network, call `init`.

See Also

`rands` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

Introduced in R2010b

randtop

Random layer topology function

Syntax

```
pos = randtop(dimensions)
```

Description

`randtop` calculates the neuron positions for layers whose neurons are arranged in an `N`-dimensional random pattern.

`pos = randtop(dimensions)` takes one argument:

`dimensions`	Row vector of dimension sizes

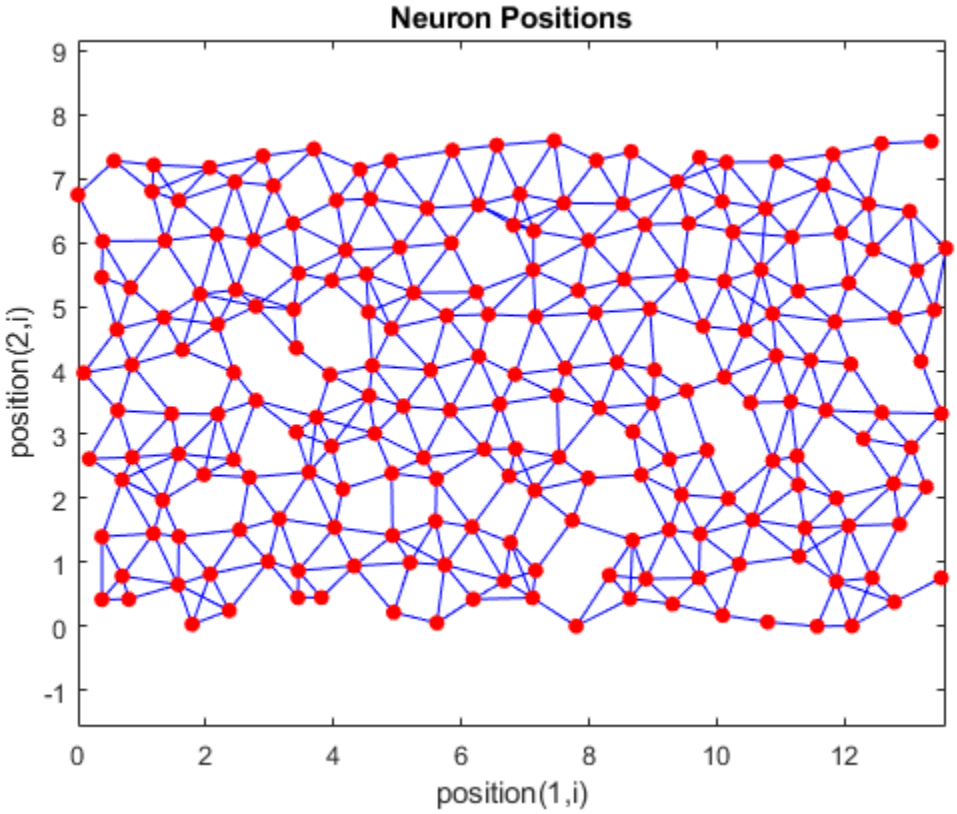
and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Random Pattern

This shows how to display a two-dimensional layer with neurons arranged in a random pattern.

```
pos = randtop([18 12]);
plotsom(pos)
```



See Also

`gridtop` | `hextop` | `tritop`

Introduced before R2006a

regression

(Not recommended) Perform linear regression of shallow network outputs on targets

Note `regression` is not recommended. Use `fitlm` instead. For more information, see "Compatibility Considerations".

Syntax

```
[r,m,b] = regression(t,y)
[r,m,b] = regression(t,y,'one')
```

Description

`[r,m,b] = regression(t,y)` calculates the linear regression between each element of the network response and the corresponding target.

This function takes cell array or matrix target t and output y, each with total matrix rows of N, and returns the regression values, r, the slopes of regression fit, m, and the y-intercepts, b, for each of the N matrix rows.

`[r,m,b] = regression(t,y,'one')` combines all matrix rows before regressing, and returns single scalar regression, slope, and offset values.

Examples

Fit Regression Model and Plot Fitted Values versus Targets

This example shows how to train a feedforward network and calculate and plot the regression between its targets and outputs.

Load the training data.

```
[x,t] = simplefit_dataset;
```

The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a feedforward neural network with one hidden layer of size 20.

```
net = feedforwardnet(20);
```

Train the network `net` using the training data.

```
net = train(net,x,t);
```

Estimate the targets using the trained network.

```
y = net(x);
```

Calculate and plot the regression between its targets and outputs.

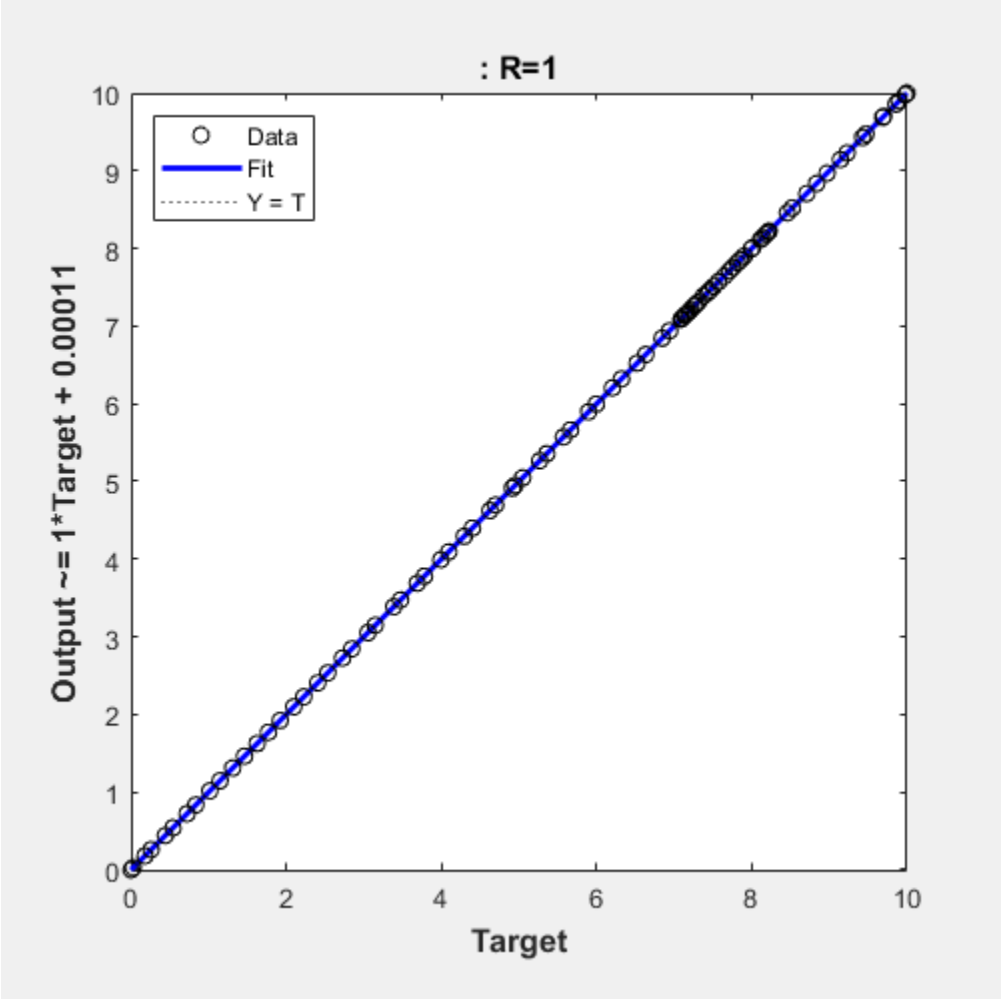
```
[r,m,b] = regression(t,y)
```

```
r = 1.0000
```

```
m = 1.0000
```

```
b = 1.0878e-04
```

```
plotregression(t,y)
```



Input Arguments

t — Target
matrix | cell array

Network targets, specified as a matrix or cell array.

y — Output
scalar

Network outputs, specified as a matrix or cell array.

Output Arguments

`r` — Regression value
scalar

Regression value, returned as a scalar.

`m` — Slope
scalar

Slope of regression fit, returned as a scalar.

`b` — Offset
scalar

Offset of regression fit, returned as a scalar.

Compatibility Considerations

`regression` is not recommended
Not recommended starting in R2020b

`regression` is not recommended. To fit a linear regression model, use `fitlm` instead.

See Also

`plotregression` | `confusion` | `fitlm`

Introduced in R2010b

removeconstantrows

Process matrices by removing rows with constant values

Syntax

```
[Y,PS] = removeconstantrows(X,max_range)
[Y,PS] = removeconstantrows(X,FP)
Y = removeconstantrows('apply',X,PS)
X = removeconstantrows('reverse',Y,PS)
```

Description

`removeconstantrows` processes matrices by removing rows with constant values.

`[Y,PS] = removeconstantrows(X,max_range)` takes `X` and an optional parameter,

`X`	`N`-by-`Q` matrix
`max_range`	Maximum range of values for row to be removed (default is 0)

and returns

`Y`	`M`-by-`Q` matrix with `N - M` rows deleted
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = removeconstantrows(X,FP)` takes parameters as a struct: `FP.max_range`.

`Y = removeconstantrows('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = removeconstantrows('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

Any `NaN` values in the input matrix are treated as missing data, and are not considered as unique values. So, for example, `removeconstantrows` removes the first row from the matrix `[1 1 1 NaN; 1 1 1 2]`.

Examples

Format a matrix so that the rows with constant values are removed.

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0];
[y1,PS] = removeconstantrows(x1);
```

```
y1 =
     1     2     4
     3     2     2

PS =
    max_range: 0
         keep: [1 3]
       remove: [2 4]
        value: [2x1 double]
```

```
        xrows: 4
        yrows: 2
    constants: [2x1 double]
    no_change: 0
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0];
y2 = removeconstantrows('apply',x2,PS)
```

```
5     2     3
6     7     3
```

Reverse the processing of `y1` to get the original `x1` matrix.

```
x1_again = removeconstantrows('reverse',y1,PS)
```

```
1     2     4
1     1     1
3     2     2
0     0     0
```

See Also

`fixunknowns` | `mapminmax` | `mapstd` | `processpca`

Introduced in R2006a

removedelay

Remove delay to neural network's response

Syntax

```
net = removedelay(net,n)
```

Description

`net = removedelay(net,n)` takes these arguments,

`net`	Neural network
`n`	Number of delays

and returns the network with input delay connections decreased, and output feedback delays increased, by the specified number of delays `n`. The result is a network which behaves identically, except that outputs are produced `n` timesteps earlier.

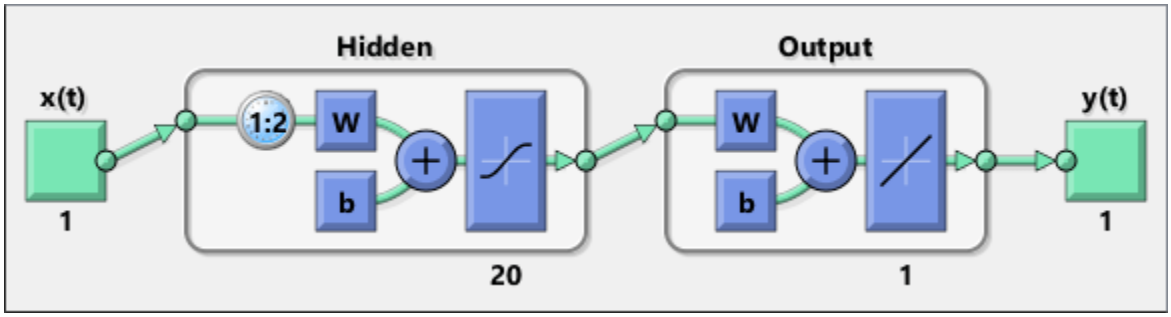
If the number of delays `n` is not specified, a default of one delay is used.

Examples

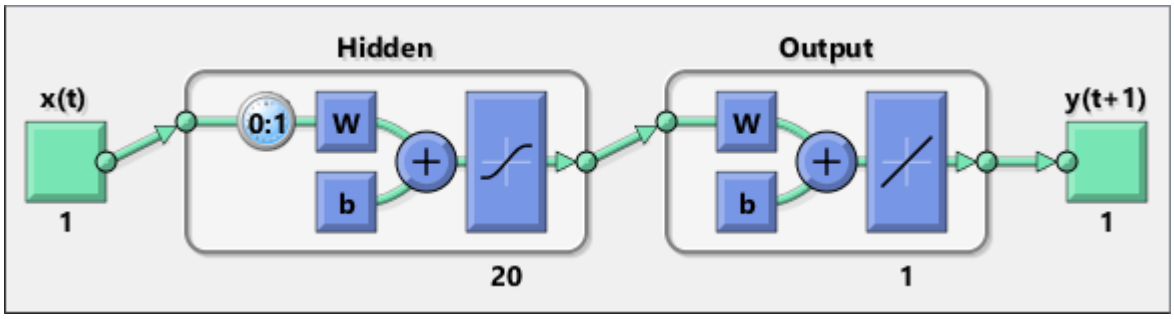
Remove and Add Delay to Network

This example creates, trains, and simulates a time delay network in its original form, on an input time series `X` and target series `T`. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following step.

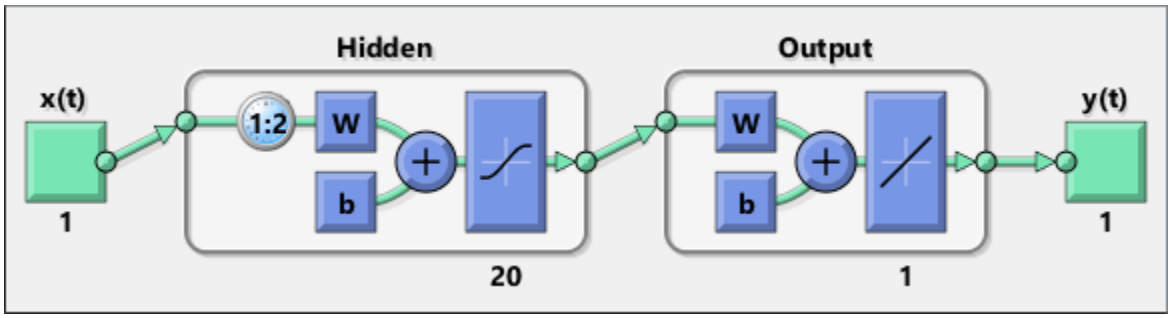
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```




```
net2 = removedelay(net1);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi);
view(net2)
```




```
net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
```



See Also

`adddelay` | `closeloop` | `openloop`

Introduced in R2010b

removerows

Process matrices by removing rows with specified indices

Syntax

```
[Y,PS] = removerows(X,'ind',ind)
[Y,PS] = removerows(X,FP)
Y = removerows('apply',X,PS)
X = removerows('reverse',Y,PS)
dx_dy = removerows('dx',X,Y,PS)
dx_dy = removerows('dx',X,[],PS)
name = removerows('name')
fp = removerows('pdefaults')
names = removerows('pdesc')
removerows('pcheck',FP)
```

Description

`removerows` processes matrices by removing rows with the specified indices.

`[Y,PS] = removerows(X,'ind',ind)` takes `X` and an optional parameter,

`X`	`N`-by-`Q` matrix
`ind`	Vector of row indices to remove (default is `[]`)

and returns

`Y`	`M`-by-`Q` matrix, where `M == N-length(ind)`
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = removerows(X,FP)` takes parameters as a struct: `FP.ind`.

`Y = removerows('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = removerows('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`dx_dy = removerows('dx',X,Y,PS)` returns the `M`-by-`N`-by-`Q` derivative of `Y` with respect to `X`.

`dx_dy = removerows('dx',X,[],PS)` returns the derivative, less efficiently.

`name = removerows('name')` returns the name of this process method.

`fp = removerows('pdefaults')` returns the default process parameter structure.

`names = removerows('pdesc')` returns the process parameter descriptions.

`removerows('pcheck',FP)` throws an error if any parameter is illegal.

Examples

Here is how to format a matrix so that rows 2 and 4 are removed:

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,ps] = removerows(x1,'ind',[2 4])
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = removerows('apply',x2,ps)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = removerows('reverse',y1,ps)
```

Algorithms

In the reverse calculation, the unknown values of replaced rows are represented with `NaN` values.

See Also

`fixunknowns` | `mapminmax` | `mapstd` | `processpca`

Introduced in R2006a

revert

Change network weights and biases to previous initialization values

Syntax

```
net = revert (net)
```

Description

`net = revert (net)` returns neural network `net` with weight and bias values restored to the values generated the last time the network was initialized.

If the network is altered so that it has different weight and bias connections or different input or layer sizes, then `revert` cannot set the weights and biases to their previous values and they are set to zeros instead.

Examples

Here a perceptron is created with input size set to 2 and number of neurons to 1.

```
net = perceptron;
net.inputs{1}.size = 2;
net.layers{1}.size = 1;
```

The initial network has weights and biases with zero values.

```
net.iw{1,1}, net.b{1}
```

Change these values as follows:

```
net.iw{1,1} = [1 2];
net.b{1} = 5;
net.iw{1,1}, net.b{1}
```

You can recover the network's initial values as follows:

```
net = revert(net);
net.iw{1,1}, net.b{1}
```

See Also

`init` | `sim` | `adapt` | `train`

Introduced before R2006a

roc

Receiver operating characteristic

Syntax

```
[tpr,fpr,thresholds] = roc(targets,outputs)
```

Description

`[tpr,fpr,thresholds] = roc(targets,outputs)` takes a matrix of targets and a matrix of outputs, and returns the true-positive/positive ratios, the false-positive/negative ratios and the thresholds over interval `[0,1]`.

For a single class problem, the function takes a matrix of boolean values indicating class membership and a matrix of outputs values in the range `[0,1]`.

The *receiver operating characteristic* is a metric used to check the quality of classifiers. For each class of a classifier, `roc` applies threshold values across the interval `[0,1]` to outputs. For each threshold, two values are calculated, the True Positive Ratio (TPR) and the False Positive Ratio (FPR). For a particular class *i*, TPR is the number of outputs whose actual and predicted class is class *i*, divided by the number of outputs whose predicted class is class *i*. FPR is the number of outputs whose actual class is not class *i*, but predicted class is class *i*, divided by the number of outputs whose predicted class is not class *i*.

You can visualize the results of this function with `plotroc`.

Examples

Calculate and Plot the ROC of a Network Trained to Recognize Iris Flowers

This example shows how to calculate and plot the ROC of a network trained to recognize Iris flowers.

```
load iris_dataset
net = patternnet(20);
net = train(net,irisInputs,irisTargets);
irisOutputs = sim(net,irisInputs);
[tpr,fpr,thresholds] = roc(irisTargets,irisOutputs)
```

Input Arguments

`targets` — Targets

matrix

Targets, specified as an `S`-by-`Q` matrix, where each column vector contains a single `1` value, with all other elements `0`. The index of the `1` indicates which of `S` categories that vector represents.

For a single class problem, this argument is specified as a `1`-by-`Q` matrix of boolean values indicating class membership.

`outputs` — Outputs
matrix

Outputs, specified as an `S`-by-`Q` matrix, where each column contains values in the range `[0,1]`. The index of the largest element in the column indicates which of `S` categories that vector presents. Alternately, `1`-by-`Q` vector, where values greater or equal to `0.5` indicate class membership, and values below `0.5`, nonmembership.

Output Arguments

`tpr` — True-positive ratios
cell array

Proportion of the targets that are greater than or equal to the threshold that actually have a target value of 1, returned as a `1`-by-`S` cell array of `1`-by-`N` vectors.

For a single class problem, this output argument is returned as a `1`-by-`N` vector.

`fpr` — False-positive ratios
cell array

Proportion of the targets that are greater than or equal to the threshold that actually have a target value of zero, returned as a `1`-by-`S` cell array of `1`-by-`N` vectors.

For a single class problem, this output argument is returned as a `1`-by-`N` vector.

`thresholds` — Thresholds
cell array

Thresholds, returned as a `1`-by-`S` cell array of `1`-by-`N` vectors over interval `[0,1]`.

For a single class problem, this output argument is returned as a `1`-by-`N` vector.

See Also
`plotroc` | `confusion`

Introduced before R2006a

sae

Sum absolute error performance function

Syntax

```
perf = sae(net,t,y,ew)
[...] = sae(...,'regularization',regularization)
[...] = sae(...,'normalization',normalization)
[...] = sae(...,FP)
```

Description

`sae` is a network performance function. It measures performance according to the sum of squared errors.

`perf = sae(net,t,y,ew)` takes these input arguments and optional function parameters,

`net`	Neural network
`t`	Matrix or cell array of target vectors
`y`	Matrix or cell array of output vectors
`ew`	Error weights (default = `{1}`)

and returns the sum squared error.

This function has two optional function parameters that can be defined with parameter name/pair arguments, or as a structure `FP` argument with fields having the parameter name and assigned the parameter values:

```
[...] = sae(...,'regularization',regularization)
```

```
[...] = sae(...,'normalization',normalization)
```

```
[...] = sae(...,FP)
```

- `regularization` — can be set to any value between the default of 0 and 1. The greater the regularization value, the more squared weights and biases are taken into account in the performance calculation.
- `normalization`
  - `'none'` — performs no normalization, the default.
  - `'standard'` — normalizes outputs and targets to `[-1, +1]`, and therefore normalizes errors to `[-2, +2]`.
  - `'percent'` — normalizes outputs and targets to `[-0.5, +0.5]`, and therefore normalizes errors to `[-1, +1]`.

Examples

Here a network is trained to fit a simple data set and its performance calculated

```
[x,t] = simplefit_dataset;
net = fitnet(10,'trainscg');
net.performFcn = 'sae';
net = train(net,x,t)
y = net(x)
e = t-y
perf = sae(net,t,y)
```

Network Use

To prepare a custom network to be trained with `sae`, set `net.performFcn` to `'sae'`. This automatically sets `net.performParam` to the default function parameters.

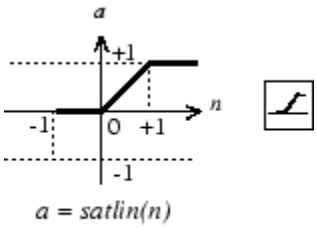
Then calling `train`, `adapt` or `perform` will result in `sae` being used to calculate performance.

Introduced in R2010b

satlin

Saturating linear transfer function

Graph and Symbol



Satlin Transfer Function

Syntax

```
A = satlin(N,FP)
```

Description

`satlin` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = satlin(N,FP)` takes two inputs,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the S-by-Q matrix of `N`'s elements clipped to `[0, 1]`.

`info = satlin('code')` returns useful information for each supported *`code`* character vector:

`satlin('name')` returns the name of this function.

`satlin('output',FP)` returns the `[min max]` output range.

`satlin('active',FP)` returns the `[min max]` active input range.

`satlin('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`satlin('fpnames')` returns the names of the function parameters.

`satlin('fpdefaults')` returns the default function parameters.

Examples

Here is the code to create a plot of the `satlin` transfer function.

```
n = -5:0.1:5;
a = satlin(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'satlin';
```

Algorithms

```
a = satlin(n) = 0, if n <= 0
n, if 0 <= n <= 1
1, if 1 <= n
```

See Also

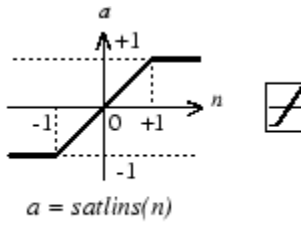
`sim` | `poslin` | `satlins` | `purelin`

Introduced before R2006a

satlins

Symmetric saturating linear transfer function

Graph and Symbol



Satlins Transfer Function

Syntax

```
A = satlins(N,FP)
```

Description

`satlins` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = satlins(N,FP)` takes `N` and an optional argument,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

and returns `A`, the S-by-Q matrix of `N`'s elements clipped to `[-1, 1]`.

`info = satlins('code')` returns useful information for each supported *`code`* character vector:

`satlins('name')` returns the name of this function.

`satlins('output',FP)` returns the `[min max]` output range.

`satlins('active',FP)` returns the `[min max]` active input range.

`satlins('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`satlins('fpnames')` returns the names of the function parameters.

`satlins('fpdefaults')` returns the default function parameters.

Examples

Here is the code to create a plot of the `satlins` transfer function.

```
n = -5:0.1:5;
a = satlins(n);
plot(n,a)
```

Algorithms

```
satlins(n) = -1, if n <= -1
n, if -1 <= n <= 1
1, if 1 <= n
```

See Also

sim | satlin | poslin | purelin

Introduced before R2006a

scalprod

Scalar product weight function

Syntax

```
Z = scalprod(W,P)
dim = scalprod('size',S,R,FP)
dw = scalprod('dw',W,P,Z,FP)
```

Description

`scalprod` is the scalar product weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = scalprod(W,P)` takes these inputs,

`W`	1-by-1 weight matrix
`P`	R-by-Q matrix of Q input (column) vectors

and returns the R-by-Q scalar product of `W` and `P` defined by `Z = w*P`.

`dim = scalprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size `[1-by-1]`.

`dw = scalprod('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(1,1);
P = rand(3,1);
Z = scalprod(W,P)
```

Network Use

To change a network so an input weight uses `scalprod`, set `net.inputWeights{i,j}.weightFcn` to `'scalprod'`.

For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'scalprod'`.

In either case, call `sim` to simulate the network with `scalprod`.

See Also

`dotprod` | `sim` | `dist` | `negdist` | `normprod`

Introduced in R2006a

selforgmap

Self-organizing map

Syntax

```
selfOrgMap = selforgmap(dimensions)
selfOrgMap = selforgmap(dimensions,coverSteps,initNeighbor,topologyFcn,
distanceFcn)
```

Description

Self-organizing maps learn to cluster data based on similarity, topology, with a preference (but no guarantee) of assigning the same number of instances to each class.

You can use self-organizing maps to cluster data and to reduce the dimensionality of data. They are inspired by the sensory and motor mappings in the mammal brain, which also appear to automatically organizing information topologically.

`selfOrgMap = selforgmap(dimensions)` takes a row vector of dimension sizes and returns a self-organizing map.

`selfOrgMap = selforgmap(dimensions,coverSteps,initNeighbor,topologyFcn, distanceFcn)` takes a row vector of dimension sizes and also a number of training steps for initial covering, an initial neighborhood size, a layer topology function, and a neuron distance function, and returns a self-organizing map.

Examples

Use Self-Organizing Map to Cluster Data

This example shows how to use a self-organizing map to cluster a simple set of data.

```
x = simplecluster_dataset;
net = selforgmap([8 8]);
net = train(net,x);
view(net)
y = net(x);
classes = vec2ind(y);
```

Input Arguments

`dimensions` — Dimension sizes
`[8 8]` (default) | row vector

Dimension sizes, specified as a row vector.

`coverSteps` — Initial covering steps
`100` (default) | scalar

Number of training steps for initial covering of the input space, specified as a scalar.

`initNeighbor` — Initial neighborhood size
3 (default) | scalar

Initial neighborhood size, specified as a scalar.

`topologyFcn` — Topology function
`'hextop'` (default) | `'randtop'` | `'gridtop'` | `'tritop'`

Layer topology function, specified as a topology function.

`distanceFcn` — Distance function
`'linkdist'` (default) | `'dist'` | `'mandist'`

Neuron distance function, specified as a distance function.

Output Arguments

`selfOrgMap` — Self-organizing map
`network` object

Self-organizing map, returned as a network object.

See Also
`lvqnet` | `competlayer` | `nctool`

Introduced in R2010b

separatewb

Separate biases and weight values from weight/bias vector

Syntax

```
[b,IW,LW] = separatewb(net,wb)
```

Description

`[b,IW,LW] = separatewb(net,wb)` takes two arguments,

`net`	Neural network
`wb`	Weight/bias vector

and returns

`b`	Cell array of bias vectors
`IW`	Cell array of input weight matrices
`LW`	Cell array of layer weight matrices

Examples

Here a feedforward network is trained to fit some data, then its bias and weight values formed into a vector. The single vector is then redivided into the original biases and weights.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
wb = formwb(net,net.b,net.iw,net.lw)
[b,iw,lw] = separatewb(net,wb)
```

See Also

`getwb` | `formwb` | `setwb`

Introduced in R2010b

seq2con

Convert sequential vectors to concurrent vectors

Syntax

```
b = seq2con(s)
```

Description

Deep Learning Toolbox software represents batches of vectors with a matrix, and sequences of vectors with multiple columns of a cell array.

`seq2con` and `con2seq` allow concurrent vectors to be converted to sequential vectors, and back again.

`b = seq2con(s)` takes one input,

`s`	`N`-by-`TS` cell array of matrices with `M` columns

and returns

`b`	`N`-by-`1` cell array of matrices with `M*TS` columns

Examples

Here three sequential values are converted to concurrent values.

```
p1 = {1 4 2}
p2 = seq2con(p1)
```

Here two sequences of vectors over three time steps are converted to concurrent vectors.

```
p1 = {[1; 1] [5; 4] [1; 2]; [3; 9] [4; 1] [9; 8]}
p2 = seq2con(p1)
```

See Also

`con2seq` | `concur`

Introduced before R2006a

setelements

Set neural network data elements

Syntax

```
setelements(x,i,v)
```

Description

`setelements(x,i,v)` takes these arguments,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

and returns the original data `x` with the data `v` stored in the elements indicated by the indices `i`.

Examples

This code sets elements 1 and 3 of matrix data:

```
x = [1 2; 3 4; 7 4]
v = [10 11; 12 13];
y = setelements(x,[1 3],v)
```

This code sets elements 1 and 3 of cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
v = {[20 21 22; 23 24 25] [26 27 28; 29 30 31]}
y = setelements(x,[1 3],v)
```

See Also

`nndata` | `numelements` | `getelements` | `catelements` | `setsamples` | `setsignals` | `settimesteps`

Introduced in R2010b

setsamples

Set neural network data samples

Syntax

```
setsamples(x,i,v)
```

Description

`setsamples(x,i,v)` takes these arguments,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

and returns the original data `x` with the data `v` stored in the samples indicated by the indices `i`.

Examples

This code sets samples 1 and 3 of matrix data:

```
x = [1 2 3; 4 7 4]
v = [10 11; 12 13];
y = setsamples(x,[1 3],v)
```

This code sets samples 1 and 3 of cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
v = {[20 21; 22 23] [24 25; 26 27]; [28 29] [30 31]}
y = setsamples(x,[1 3],v)
```

See Also

`nndata` | `numsamples` | `getsamples` | `catsamples` | `setelements` | `setsignals` | `settimesteps`

Introduced in R2010b

setsignals

Set neural network data signals

Syntax

```
setsignals(x,i,v)
```

Description

`setsignals(x,i,v)` takes these arguments,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

and returns the original data `x` with the data `v` stored in the signals indicated by the indices `i`.

Examples

This code sets signal 2 of cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
v = {[20:22] [23:25]}
y = setsignals(x,2,v)
```

See Also

`nndata` | `numsignals` | `getsignals` | `catsignals` | `setelements` | `setsamples` | `settimesteps`

Introduced in R2010b

setsiminit

Set neural network Simulink block initial conditions

Syntax

```
setsiminit(sysName,netName,net,xi,ai,Q)
```

Description

`setsiminit(sysName,netName,net,xi,ai,Q)` takes these arguments,

`sysName`	The name of the Simulink system containing the neural network block
`netName`	The name of the Simulink neural network block
`net`	The original neural network
`xi`	Initial input delay states
`ai`	Initial layer delay states
`Q`	Sample number (default is 1)

and sets the Simulink neural network blocks initial conditions as specified.

Examples

Here a NARX network is designed. The NARX network has a standard input and an open loop feedback output to an associated feedback input.

```
[x,t] = simplenarx_dataset;
     net = narxnet(1:2,1:2,20);
     view(net)
     [xs,xi,ai,ts] = preparets(net,x,{},t);
     net = train(net,xs,ts,xi,ai);
     y = net(xs,xi,ai);
```

Now the network is converted to closed loop, and the data is reformatted to simulate the network's closed loop response.

```
net = closeloop(net);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
y = net(xs,xi,ai);
```

Here the network is converted to a Simulink system with workspace input and output ports. Its delay states are initialized, inputs `X1` defined in the workspace, and it is ready to be simulated in Simulink.

```
[sysName,netName] = gensim(net,'InputMode','Workspace',...
    'OutputMode','WorkSpace','SolverMode','Discrete');
setsiminit(sysName,netName,net,xi,ai,1);
x1 = nndata2sim(x,1,1);
```

Finally the initial input and layer delays are obtained from the Simulink model. (They will be identical to the values set with `setsiminit`.)

```
[xi,ai] = getsiminit(sysName,netName,net);
```

See Also

gensim | getsiminit | nndata2sim | sim2nndata

Introduced in R2010b

settimesteps

Set neural network data timesteps

Syntax

```
settimesteps(x,i,v)
```

Description

`settimesteps(x,i,v)` takes these arguments,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

and returns the original data `x` with the data `v` stored in the timesteps indicated by the indices `i`.

Examples

This code sets timestep 2 of cell array data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
v = {[20:22; 23:25]; [25:27]}
y = settimesteps(x,2,v)
```

See Also

`nndata` | `numtimesteps` | `gettimesteps` | `cattimesteps` | `setelements` | `setsamples` | `setsignals`

Introduced in R2010b

setwb

Set all network weight and bias values with single vector

Syntax

```
net = setwb(net,wb)
```

Description

This function sets a network’s weight and biases to a vector of values.

`net = setwb(net,wb)` takes the following inputs:

`net`	Neural network
`wb`	Vector of weight and bias values

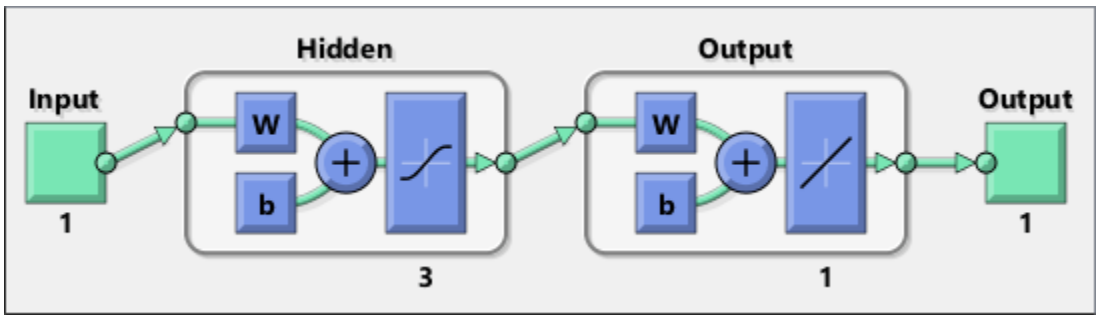
Examples

Set Network's Weights and Biases

This example shows how to set and view a network’s weight and bias values.

Create and configure a network.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(3);
net = configure(net,x,t);
view(net)
```



This network has three weights and three biases in the first layer, and three weights and one bias in the second layer. So, the total number of weight and bias values in the network is 10. Set the weights and biases to random values.

```
net = setwb(net,rand(10,1));
```

View the weight and bias values

```
net.IW{1,1}
net.b{1}
```

```
ans =

    0.1576
    0.9706
    0.9572


ans =

    0.5469
    0.9575
    0.9649
```

See Also

getwb | formwb | separatewb

Introduced in R2010b

sim

Simulate neural network

Syntax

```
[Y,Xf,Af] = sim(net,X,Xi,Ai,T)
[Y,Xf,Af] = sim(net,{Q TS},Xi,Ai)
[Y,...] = sim(net,...,'useParallel',...)
[Y,...] = sim(net,...,'useGPU',...)
[Y,...] = sim(net,...,'showResources',...)
[Ycomposite,...] = sim(net,Xcomposite,...)
[Ygpu,...] = sim(net,Xgpu,...)
```

To Get Help

Type `help network/sim`.

Description

`sim` simulates neural networks.

`[Y,Xf,Af] = sim(net,X,Xi,Ai,T)` takes

`net`	Network
`X`	Network inputs
`Xi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)
`T`	Network targets (default = zeros)

and returns

`Y`	Network outputs
`Xf`	Final input delay conditions
`Af`	Final layer delay conditions

`sim` is usually called implicitly by calling the neural network as a function. For instance, these two expressions return the same result:

```
y = sim(net,x,xi,ai)
y = net(x,xi,ai)
```

Note that arguments `Xi`, `Ai`, `Xf`, and `Af` are optional and need only be used for networks that have input or layer delays.

The signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

X	Ni-by-TS cell array	Each element X{i,ts} is an Ri-by-Q matrix.
Xi	Ni-by-ID cell array	Each element Xi{i,k} is an Ri-by-Q matrix.
Ai	Nl-by-LD cell array	Each element Ai{i,k} is an Si-by-Q matrix.
T	No-by-TS cell array	Each element X{i,ts} is a Ui-by-Q matrix.
Y	No-by-TS cell array	Each element Y{i,ts} is a Ui-by-Q matrix.
Xf	Ni-by-ID cell array	Each element Xf{i,k} is an Ri-by-Q matrix.
Af	Nl-by-LD cell array	Each element Af{i,k} is an Si-by-Q matrix.

where

Ni	=	net.numInputs
Nl	=	net.numLayers
No	=	net.numOutputs
ID	=	net.numInputDelays
LD	=	net.numLayerDelays
TS	=	Number of time steps
Q	=	Batch size
Ri	=	net.inputs{i}.size
Si	=	net.layers{i}.size
Ui	=	net.outputs{i}.size

The columns of Xi, Ai, Xf, and Af are ordered from oldest delay condition to most recent:

Xi{i,k}	=	Input i at time ts = k - ID
Xf{i,k}	=	Input i at time ts = TS + k - ID
Ai{i,k}	=	Layer output i at time ts = k - LD
Af{i,k}	=	Layer output i at time ts = TS + k - LD

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can also be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

X	(sum of Ri)-by-Q matrix
Xi	(sum of Ri)-by-(ID*Q) matrix
Ai	(sum of Si)-by-(LD*Q) matrix
T	(sum of Ui)-by-Q matrix
Y	(sum of Ui)-by-Q matrix
Xf	(sum of Ri)-by-(ID*Q) matrix
Af	(sum of Si)-by-(LD*Q) matrix

[Y,Xf,Af] = sim(net,{Q TS},Xi,Ai) is used for networks that do not have an input when cell array notation is used.

`[Y,...] = sim(net,...,'useParallel',...)`, `[Y,...] = sim(net,...,'useGPU',...)`, or `[Y,...] = sim(net,...,'showResources',...)` (or the network called as a function) accepts optional name/value pair arguments to control how calculations are performed. Two of these options allow training to happen faster or on larger datasets using parallel workers or GPU devices if Parallel Computing Toolbox is available. These are the optional name/value pairs:

`'useParallel','no'`	Calculations occur on normal MATLAB thread. This is the default `'useParallel'` setting.
`'useParallel','yes'`	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
`'useGPU','no'`	Calculations occur on the CPU. This is the default 'useGPU' setting.
`'useGPU','yes'`	Calculations occur on the current gpuDevice if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current gpuDevice is not supported, calculations remain on the CPU. If `'useParallel'` is also `'yes'` and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.
`'useGPU','only'`	If no parallel pool is open, then this setting is the same as `'yes'`. If a parallel pool is open, then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.
`'showResources','no'`	Do not display computing resources used at the command line. This is the default setting.
`'showResources','yes'`	Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.

`[Ycomposite,...] = sim(net,Xcomposite,...)` takes Composite data and returns Composite results. If Composite data is used, then `'useParallel'` is automatically set to `'yes'`.

`[Ygpu,...] = sim(net,Xgpu,...)` takes gpuArray data and returns gpuArray results. If gpuArray data is used, then `'useGPU'` is automatically set to `'yes'`.

Examples

In the following examples, the `sim` function is called implicitly by calling the neural network object (`net`) as a function.

Simulate Feedforward Networks

This example loads a dataset that maps anatomical measurements `x` to body fat percentages `t`. A feedforward network with 10 neurons is created and trained on that data, then simulated.

```
[x,t] =  bodyfat_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
```

Simulate NARX Time Series Networks

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current `x` and the magnet's vertical position response `t`, then simulates the network. The function `preparets` prepares the data before training and simulation. It creates the open-loop network's combined inputs `xo`, which contains both the external input `x` and previous values of position `t`. It also prepares the delay states `xi`.

```
[x,t] = maglev_dataset;
net = narxnet(10);
[xo,xi,~,to] = preparets(net,x,{},t);
net = train(net,xo,to,xi);
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```
netc = closeloop(net);
view(netc)
[xc,xi,ai,tc] = preparets(netc,x,{},t);
yc = netc(xc,xi,ai);
```

Simulate in Parallel on a Parallel Pool

With Parallel Computing Toolbox you can simulate and train networks faster and on larger datasets than can fit on one PC. Here training and simulation happens across parallel MATLAB workers.

```
parpool
[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useParallel','yes','showResources','yes');
Y = net(X,'useParallel','yes');
```

Simulate on GPUs

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed, then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

```
Xc = Composite;
for i=1:numel(Xc)
    Xc{i} = X+rand(size(X))*0.1;  % Use real data instead of random
end
Yc = net(Xc,'showResources','yes');
```

Networks can be simulated using the current GPU device, if it is supported by Parallel Computing Toolbox.

```
gpuDevice % Check if there is a supported GPU
Y = net(X,'useGPU','yes','showResources','yes');
```

To put the data on a GPU manually, and get the results on the GPU:

```
Xgpu = gpuArray(X);
Ygpu = net(Xgpu,'showResources','yes');
Y = gather(Ygpu);
```

To run in parallel, with workers associated with unique GPUs taking advantage of that hardware, while the rest of the workers use CPUs:

```
Y = net(X,'useParallel','yes','useGPU','yes','showResources','yes');
```

Using only workers with unique GPUs might result in higher speeds, as CPU workers might not keep up.

```
Y = net(X,'useParallel','yes','useGPU','only','showResources','yes');
```

Algorithms

`sim` uses these properties to simulate a network `net`.

```
net.numInputs, net.numLayers
net.outputConnect, net.biasConnect
net.inputConnect, net.layerConnect
```

These properties determine the network's weight and bias values and the number of delays associated with each weight:

```
net.IW{i,j}
net.LW{i,j}
net.b{i}
net.inputWeights{i,j}.delays
net.layerWeights{i,j}.delays
```

These function properties indicate how `sim` applies weight and bias values to inputs to get each layer's output:

```
net.inputWeights{i,j}.weightFcn
net.layerWeights{i,j}.weightFcn
net.layers{i}.netInputFcn
net.layers{i}.transferFcn
```

See Also

`init` | `adapt` | `train` | `revert`

Introduced before R2006a

sim2nndata

Convert Simulink time series to neural network data

Syntax

```
sim2nndata(x)
```

Description

`sim2nndata(x)` takes either a column vector of values or a Simulink time series structure and converts it to a neural network data time series.

Examples

Here a random Simulink 20-step time series is created and converted.

```
simts = rands(20,1);
nnts = sim2nndata(simts)
```

Here a similar time series is defined with a Simulink structure and converted.

```
simts.time = 0:19
simts.signals.values = rands(20,1);
simts.dimensions = 1;
nnts = sim2nndata(simts)
```

See Also

`nndata` | `nndata2sim`

Introduced in R2010b

softmax

Soft max transfer function

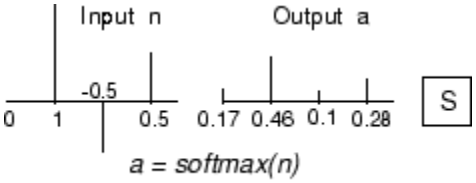
Syntax

```
A = softmax(N)
info = softmax(code)
```

Description

Tip To use a softmax activation for deep learning, use `softmaxLayer` or the `dlarray` method softmax.

`A = softmax(N)` takes a `S`-by-`Q` matrix of net input (column) vectors, `N`, and returns the `S`-by-`Q` matrix, `A`, of the softmax competitive function applied to each column of `N`.



Softmax Transfer Function

`softmax` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`info = softmax(code)` returns information about this function. For more information, see the **code** argument description.

Examples

Create and Plot the `softmax` Transfer Function

This example shows how to calculate and plot the softmax transfer function of an input matrix.

Create the input matrix, `n`. Then call the `softmax` function and plot the results.

```
n = [0; 1; -0.5; 0.5];
a = softmax(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'softmax';
```

Input Arguments

N — Input matrix

matrix

Net input column vectors, specified as an S-by-Q matrix.

code — Information option

`'name'` | `'output'` | `'active'` | `'fullderiv'` | `'fpnames'` | `'fpdefaults'`

Information you want to retrieve from the function, specified as one of the following:

- `'name'` returns the name of this function.
- `'output'` returns the `[min max]` output range.
- `'active'` returns the `[min max]` active input range.
- `'fullderiv'` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.
- `'fpnames'` returns the names of the function parameters.
- `'fpdefaults'` returns the default function parameters.

Output Arguments

A — Output matrix

matrix

Output matrix, returned as an S-by-Q matrix of the softmax competitive function applied to each column of N.

info — Information output

string | vector | scalar

Specific information about the function, according to the option specified in the `code` argument, returned as either a string, a vector, or a scalar.

Algorithms

```
a = softmax(n) = exp(n)/sum(exp(n))
```

See Also

`sim` | `compet`

Introduced before R2006a

srchbac

1-D minimization using backtracking

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)
```

Description

`srchbac` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called backtracking.

`[a,gX,perf,retcode,delta,tol] = srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

	`0` Normal
	`1` Minimum step taken
	`2` Maximum step taken
	`3` Beta condition not met
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the backstepping algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`low_lim`	Lower limit on change in step size
`up_lim`	Upper limit on change in step size
`maxstep`	Maximum step length
`minstep`	Minimum step length
`scale_tol`	Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`V`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

More About

Backtracking Search

The backtracking search routine `srchbac` is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and a step multiplier of 1. It also uses the value of the derivative of performance at the current point

to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in Dennis and Schnabel. It is used as the default line search for the quasi-Newton algorithms, although it might not be the best technique for all problems.

Algorithms

`srchbac` locates the minimum of the performance function in the search direction `dX`, using the backtracking algorithm described on page 126 and 328 of Dennis and Schnabel's book, noted below.

References

Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ, Prentice-Hall, 1983

See Also

`srchcha` | `srchgol` | `srchhyb`

Introduced before R2006a

srchbre

1-D interval location using Brent's method

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchbre` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called Brent's technique.

`[a,gX,perf,retcode,delta,tol] = srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

	`0` Normal
	`1` Minimum step taken
	`2` Maximum step taken
	`3` Beta condition not met
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the Brent algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`bmax`	Largest step size
`scale_tol`	Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

More About

Brent’s Search

Brent’s search is a linear search that is a hybrid of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate that is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods can take many iterations to become apparent. Brent’s search attempts to combine the best features of both approaches.

For Brent's search, you begin with the same interval of uncertainty used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.

See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm can require more performance evaluations than algorithms that use derivative information.

Algorithms

`srchbre` brackets the minimum of the performance function in the search direction `dX`, using Brent's algorithm, described on page 46 of Scales (see reference below). It is a hybrid algorithm based on the golden section search and the quadratic approximation.

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`srchbac` | `srchcha` | `srchgol` | `srchhyb`

Introduced before R2006a

srchcha

1-D minimization using Charalambous' method

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchcha` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique based on Charalambous' method.

`[a,gX,perf,retcode,delta,tol] = srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

	`0` Normal
	`1` Minimum step taken
	`2` Maximum step taken
	`3` Beta condition not met
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the Charalambous algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`gama`	Parameter to avoid small reductions in performance, usually set to 0.1
`scale_tol`	Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

More About

Charalambous’ Search

The method of Charalambous, `srchcha`, was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like `srchbre` and `srchhyb`, it is a hybrid search. It uses a cubic interpolation together with a type of sectioning.

See [Char92] for a description of Charalambous' search. This routine is used as the default search for most of the conjugate gradient algorithms because it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with

fewer steps. This is not true for all problems, and you might want to experiment with other line searches.

Algorithms

`srchcha` locates the minimum of the performance function in the search direction `dX`, using an algorithm based on the method described in Charalambous (see reference below).

References

Charalambous, C., “Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, Vol. 139, No. 3, June, 1992, pp. 301–310.

See Also

`srchbac` | `srchbre` | `srchgol` | `srchhyb`

Introduced before R2006a

srchgol

1-D minimization using golden section search

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchgol` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called the golden section search.

`[a,gX,perf,retcode,delta,tol] = srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

	`0` Normal
	`1` Minimum step taken
	`2` Maximum step taken
	`3` Beta condition not met
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the golden section algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`bmax`	Largest step size
`scale_tol`	Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

More About

Golden Section Search

The golden section search `srchgol` is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance function occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determine a section of the interval that can be discarded, and a new interior point is placed within the new interval. This procedure is continued until the interval of uncertainty is reduced to a width of `tol`, which is equal to `delta`/`scale_tol`.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the *Neural Network Design* demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.

Algorithms

`srchgol` locates the minimum of the performance function in the search direction `dX`, using the golden section search. It is based on the algorithm as described on page 33 of Scales (see reference below).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`srchbac` | `srchbre` | `srchcha` | `srchhyb`

Introduced before R2006a

srchhyb

1-D minimization using a hybrid bisection-cubic search

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`srchhyb` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique that is a combination of a bisection and a cubic interpolation.

`[a,gX,perf,retcode,delta,tol] = srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point

`retcode`	Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.
	`0` Normal
	`1` Minimum step taken
	`2` Maximum step taken
	`3` Beta condition not met
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the hybrid bisection-cubic algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`bmax`	Largest step size
`scale_tol`	Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

More About

Hybrid Bisection Cubic Search

Like Brent's search, `srchhyb` is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty, and the

performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is obtained by using the value of the performance and its derivative at the two endpoints. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent's algorithm.

Algorithms

`srchhyb` locates the minimum of the performance function in the search direction `dX`, using the hybrid bisection-cubic interpolation algorithm described on page 50 of Scales (see reference below).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York Springer-Verlag, 1985

See Also

`srchbac` | `srchbre` | `srchcha` | `srchgol`

Introduced before R2006a

sse

Sum squared error performance function

Syntax

```
perf = sse(net,t,y,ew)
perf = sse(net,t,y,ew,Name,Value)
```

Description

`perf = sse(net,t,y,ew)` takes a network `net`, targets `T`, outputs `Y`, and optionally error weights `EW`, and returns network performance calculated as the sum squared error.

`sse` is a network performance function. It measures performance according to the sum of squared errors.

`perf = sse(net,t,y,ew,Name,Value)` has two optional function parameters that set the regularization of the errors and the normalizations of the outputs and targets.

`sse` is a network performance function. It measures performance according to the sum of squared errors.

Examples

Calculate Network Performance with `'sse'` Function

This example shows how to calculate the performance of a feed-forward network with the `sse` function.

Create a network using the data from the simple fit data set, train it, and calculate its performance.

```
[x,t] = simplefit_dataset;
net = fitnet(10);
net.performFcn = 'sse';
net = train(net,x,t)
y = net(x)
e = t-y
perf = sse(net,t,y)
```

Input Arguments

net — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

t — Network targets

matrix | cell array

Network targets, specified as a matrix or cell array.

y — Network outputs
matrix | cell array

Network outputs, specified as a matrix or cell array.

ew — Network outputs
{1} (default) | vector | matrix | cell array

Error weights, specified as a vector, matrix, or cell array.

Error weights can be defined by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2]; % Across 4 samples
ew = [0.1; 0.5; 1.0]; % Across 3 elements
ew = {0.1 0.2 0.3 0.5 1.0}; % Across 5 timesteps
ew = {1.0; 0.5}; % Across 2 outputs
```

The error weights can also be defined across any combination, such as across two time-series (i.e., two samples) over four timesteps.

```
ew = {[0.5 0.4],[0.3 0.5],[1.0 1.0],[0.7 0.5]};
```

In the general case, error weights may have exactly the same dimensions as targets, in which case each target value will have an associated error weight.

The default error weight treats all errors the same.

```
ew = {1}
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'regularization',0.1`

regularization — Proportion of performance
0 (default) | integer between 0 and 1

Proportion of performance attributed to weight and bias values, specified as the comma-separated pair consisting of `'regularization'` and an integer between `0` and `1`. The larger this value is, the more the network is penalized for larger weights, and the more likely the network function avoids overfitting.

normalization — Output and target normalization
`'none'` (default) | `'standard'` | `'percent'`

Output and target normalization, specified as the comma-separated pair consisting of `'normalization'` and either:

- `'none'` — performs no normalization.
- `'standard'` — normalizes outputs and targets to `[-1, +1]`, and therefore normalizes errors to `[-2, +2]`.

- `'percent'` — normalizes outputs and targets to `[-0.5, +0.5]`, and therefore normalizes errors to `[-1, +1]`.

Output Arguments

`perf` — Network performance

scalar

Network performance calculated as the sum squared error, returned as a scalar.

More About

Network Use

To prepare a custom network to be trained with `sse`, set `net.performFcn` to `'sse'`. This automatically sets `net.performParam` to the default function parameters.

Then calling `train`, `adapt` or `perform` will result in `sse` being used to calculate performance.

See Also

`mse` | `mae`

Introduced before R2006a

staticderiv

Static derivative function

Syntax

```
staticderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
staticderiv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the chain rule from the networks performance or outputs back to its inputs. For time series data and dynamic networks this function ignores the delay connections resulting in a approximation (which may be good or not) of the actual derivative. This function is used by Elman networks (elmannet) which is a dynamic network trained by the static derivative approximation when full derivative calculations are not available. As full derivatives are calculated by all the other derivative functions, this function is not recommended for dynamic networks except for research into training algorithms.

`staticderiv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

and returns the gradient of performance with respect to the network's weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

`staticderiv('de_dwb',net,X,T,Xi,Ai,EW)` returns the Jacobian of errors with respect to the network's weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = staticderiv('dperf_dwb',net,x,t)
jwb = staticderiv('de_dwb',net,x,t)
```

See Also

`bttderiv` | `defaultderiv` | `fpderiv` | `num2deriv`

Introduced in R2010b

sumabs

Sum of absolute elements of matrix or matrices

Syntax

```
[s,n] = sumabs(x)
```

Description

`[s,n] = sumabs(x)` takes a matrix or cell array of matrices and returns,

`s`	Sum of all absolute finite values
`n`	Number of finite values

If `x` contains no finite values, the sum returned is 0.

Examples

```
m = sumabs([1 2;3 4])
[m,n] = sumabs({[1 2; NaN 4], [4 5; 2 3]})
```

See Also

`meanabs` | `meansqr` | `sumsqr`

Introduced in R2010b

sumsqr

Sum of squared elements of matrix or matrices

Syntax

```
[s,n] = sumsqr(x)
```

Description

`[s,n] = sumsqr(x)` takes a matrix or cell array of matrices, `x`, and returns the sum, `s`, of all squared finite values in `x`, and the number of finite values, `n`.

If `x` does not contain finite values, the sum returned is 0.

Examples

Calculate the Sum of Squared Elements Using the `sumsqr` Function

This example shows how to calculate the sum of squared elements of a matrix and a cell array using the `sumsqr` function.

```
m = sumsqr([1 2;3 4])
```

```
m = 30
```

```
[m,n] = sumsqr({[1 2; NaN 4], [4 5; 2 3]})
```

```
m = 75
```

```
n = 7
```

Input Arguments

x — Input matrix

matrix | cell array of matrices

Input elements, specified as a matrix or cell array of matrices.

Output Arguments

s — Sum of squared elements

scalar

Sum of all squared elements in `x`, returned as a scalar.

n — Number of finite values

scalar

Number of finite values in `x`, returned as a scalar.

See Also

meanabs | meansqr | sumabs

Introduced before R2006a

tansig

Hyperbolic tangent sigmoid transfer function

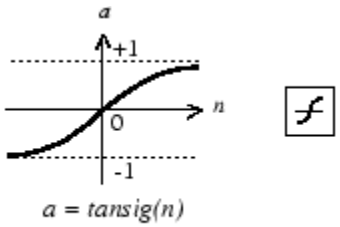
Syntax

```
A = tansig(N)
```

Description

Tip To use a hyperbolic tangent activation for deep learning, use the `tanhLayer` function or the `dlarray` method tanh.

`A = tansig(N)` takes a matrix of net input vectors, `N` and returns the `S`-by-`Q` matrix, `A`, of the elements of `N` squashed into `[-1 1]`.



Tan-Sigmoid Transfer Function

`tansig` is a neural transfer function. Transfer functions calculate the output of a layer from its net input.

Examples

Create a Plot of the `tansig` Transfer Function

This example shows how to calculate and plot the hyperbolic tangent sigmoid transfer function of an input matrix.

Create the input matrix, `n`. Then call the `tansig` function and plot the results.

```
n = -5:0.1:5;
a = tansig(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'tansig';
```

Input Arguments

N — Input matrix

matrix

Net input column vectors, specified as an S-by-Q matrix.

Output Arguments

A — Output matrix

matrix

Output vectors, returned as an S-by-Q matrix, where each element of N is squashed from the interval `[-inf inf]` to the interval `[-1 1]` with an "S-shaped" function.

Algorithms

```
a = tansig(N) = 2/(1+exp(-2*N))-1
```

This is mathematically equivalent to `tanh(N)`.

References

[1] Vogl, T. P., et al. *'Accelerating the Convergence of the Back-Propagation Method'*. *Biological Cybernetics*, vol. 59, no. 4–5, Sept. 1988, pp. 257–63. *DOI.org (Crossref)*, doi:10.1007/BF00332914.

See Also

`sim` | `logsig`

Introduced before R2006a

tapdelay

Shift neural network time series data for tap delay

Syntax

```
tapdelay(x,i,ts,delays)
```

Description

`tapdelay(x,i,ts,delays)` takes these arguments,

`x`	Neural network time series data
`i`	Signal index
`ts`	Timestep index
`delays`	Row vector of increasing zero or positive delays

and returns the tap delay values of signal `i` at timestep `ts` given the specified tap delays.

Examples

Here a random signal `x` consisting of eight timesteps is defined, and a tap delay with delays of `[0 1 4]` is simulated at timestep 6.

```
x = num2cell(rand(1,8));
y = tapdelay(x,1,6,[0 1 4])
```

See Also

`nndata` | `extendts` | `preparets`

Introduced in R2010b

timedelaynet

Time delay neural network

Syntax

```
timedelaynet(inputDelays,hiddenSizes,trainFcn)
```

Description

`timedelaynet(inputDelays,hiddenSizes,trainFcn)` takes these arguments:

- Row vector of increasing 0 or positive input delays, `inputDelays`
- Row vector of one or more hidden layer sizes, `hiddenSizes`
- Training function, `trainFcn`

and returns a time delay neural network.

Time delay networks are similar to feedforward networks, except that the input weight has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the distributed delay neural network (`distdelaynet`), which has delays on the layer weights in addition to the input weight.

Examples

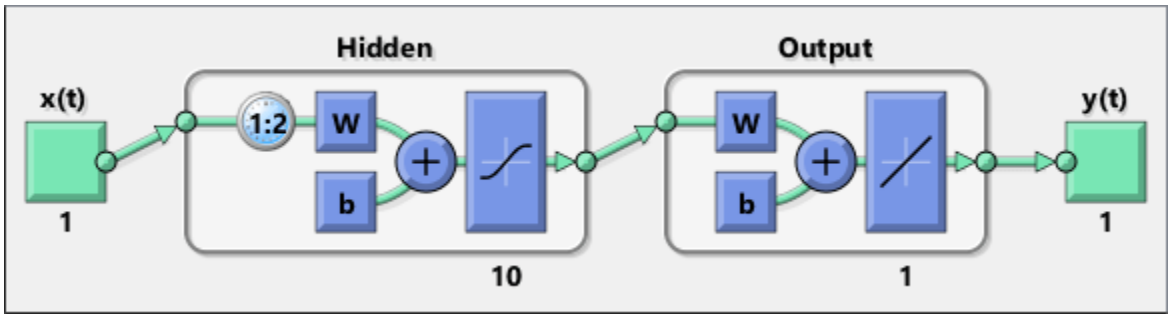
Train Time Delay Network and Predict on New Data

Partition the training set. Use Xnew to do prediction in closed loop mode later.

```
[X,T] = simpleseries_dataset;
Xnew = X(81:100);
X = X(1:80);
T = T(1:80);
```

Train a time delay network, and simulate it on the first 80 observations.

```
net = timedelaynet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```

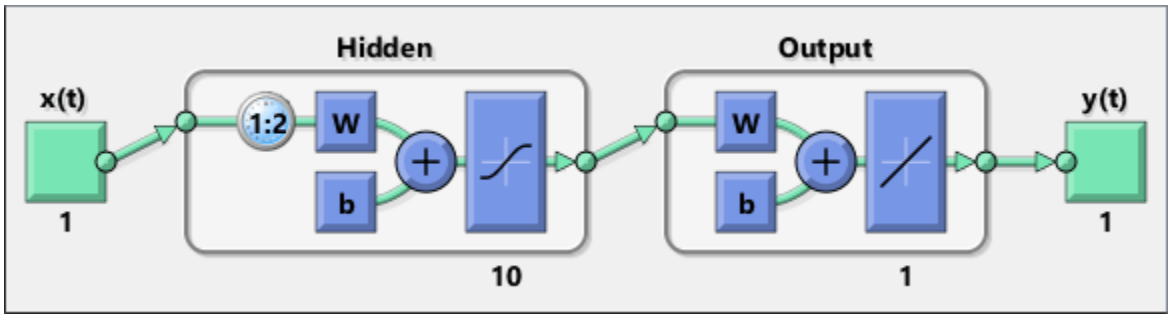


Calculate the network performance.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y);
```

Run the prediction for 20 timesteps ahead in closed loop mode.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
view(netc)
```




```
y2 = netc(Xnew,Xic,Aic);
```

Input Arguments

`inputDelays` — Input delays

`[1:2]` (default) | row vector

Zero or positive input delays, specified as an increasing row vector.

`hiddenSizes` — Hidden sizes

`10` (default) | row vector

Sizes of the hidden layers, specified as a row vector of one or more elements.

`trainFcn` — Training function name

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Training Function	Algorithm
`'traingdx'`	Variable Learning Rate Gradient Descent
`'traingdm'`	Gradient Descent with Momentum
`'traingd'`	Gradient Descent

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

See Also

`preparets` | `removedelay` | `distdelaynet` | `narnet` | `narxnet`

Introduced in R2010b

tonndata

Convert data to standard neural network cell array form

Syntax

```
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
```

Description

`[y,wasMatrix] = tonndata(x,columnSamples,cellTime)` takes these arguments,

`x`	Matrix or cell array of matrices
`columnSamples`	True if original samples are oriented as columns, false if rows
`cellTime`	True if original samples are columns of a cell array, false if they are stored in a matrix

and returns

`y`	Original data transformed into standard neural network cell array form
`wasMatrix`	True if original data was a matrix (as opposed to cell array)

If `columnSamples` is false, then matrix `x` or matrices in cell array `x` will be transposed, so row samples will now be stored as column vectors.

If `cellTime` is false, then matrix samples will be separated into columns of a cell array so time originally represented as vectors in a matrix will now be represented as columns of a cell array.

The returned value `wasMatrix` can be used by `fromnndata` to reverse the transformation.

Examples

Here data consisting of six timesteps of 5-element vectors, originally represented as a matrix with six columns, is converted to standard neural network representation and back.

```
x = rands(5,6)
columnSamples = true; % samples are by columns.
cellTime = false;     % time-steps in matrix, not cell array.
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```

See Also

`nndata` | `fromnndata` | `nndata2sim` | `sim2nndata`

Introduced in R2010b

train

Train shallow neural network

Syntax

```
trainedNet = train(net,X,T,Xi,Ai,EW)
[trainedNet,tr] = train(net,X,T,Xi,Ai,EW)
[trainedNet,tr] = train(net,X,T,Xi,Ai,EW,Name,Value)
```

Description

This function trains a shallow neural network. For deep learning with convolutional or LSTM neural networks, see `trainNetwork` instead.

`trainedNet = train(net,X,T,Xi,Ai,EW)` trains a network `net` according to `net.trainFcn` and `net.trainParam`.

`[trainedNet,tr] = train(net,X,T,Xi,Ai,EW)` also returns a training record.

`[trainedNet,tr] = train(net,X,T,Xi,Ai,EW,Name,Value)` trains a network with additional options specified by one or more name-value pair arguments.

Examples

Train and Plot Networks

Here input `x` and targets `t` define a simple function that you can plot:

```
x = [0 1 2 3 4 5 6 7 8];
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28 0.66 0.99];
plot(x,t,'o')
```

Here `feedforwardnet` creates a two-layer feed-forward network. The network has one hidden layer with ten neurons.

```
net = feedforwardnet(10);
net = configure(net,x,t);
y1 = net(x)
plot(x,t,'o',x,y1,'x')
```

The network is trained and then resimulated.

```
net = train(net,x,t);
y2 = net(x)
plot(x,t,'o',x,y1,'x',x,y2,'*')
```

Train NARX Time Series Network

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current `x` and the magnet's vertical position response `t`,

then simulates the network. The function `preparets` prepares the data before training and simulation. It creates the open-loop network's combined inputs `xo`, which contains both the external input `x` and previous values of position `t`. It also prepares the delay states `xi`.

```
[x,t] = maglev_dataset;
net = narxnet(10);
[xo,xi,~,to] = preparets(net,x,{},t);
net = train(net,xo,to,xi);
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```
netc = closeloop(net);
view(netc)
[xc,xi,ai,tc] = preparets(netc,x,{},t);
yc = netc(xc,xi,ai);
```

Train a Network in Parallel on a Parallel Pool

Parallel Computing Toolbox allows Deep Learning Toolbox to simulate and train networks faster and on larger datasets than can fit on one PC. Parallel training is currently supported for backpropagation training only, not for self-organizing maps.

Here training and simulation happens across parallel MATLAB workers.

```
parpool
[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useParallel','yes','showResources','yes');
Y = net(X);
```

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

```
[X,T] = vinyl_dataset;
Q = size(X,2);
Xc = Composite;
Tc = Composite;
numWorkers = numel(Xc);
ind = [0 ceil((1:numWorkers)*(Q/numWorkers))];
for i=1:numWorkers
    indi = (ind(i)+1):ind(i+1);
    Xc{i} = X(:,indi);
    Tc{i} = T(:,indi);
end
net = feedforwardnet;
net = configure(net,X,T);
net = train(net,Xc,Tc);
Yc = net(Xc);
```

Note in the example above the function configure was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when train is called, but when providing composite data this step must be done manually with non-Composite data.

Train a Network on GPUs

Networks can be trained using the current GPU device, if it is supported by Parallel Computing Toolbox. GPU training is currently supported for backpropagation training only, not for self-organizing maps.

```
[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useGPU','yes');
y = net(X);
```

To put the data on a GPU manually:

```
[X,T] = vinyl_dataset;
Xgpu = gpuArray(X);
Tgpu = gpuArray(T);
net = configure(net,X,T);
net = train(net,Xgpu,Tgpu);
Ygpu = net(Xgpu);
Y = gather(Ygpu);
```

Note in the example above the function configure was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when train is called, but when providing gpuArray data this step must be done manually with non-gpuArray data.

To run in parallel, with workers each assigned to a different unique GPU, with extra workers running on CPU:

```
net = train(net,X,T,'useParallel','yes','useGPU','yes');
y = net(X);
```

Using only workers with unique GPUs might result in higher speed, as CPU workers might not keep up.

```
net = train(net,X,T,'useParallel','yes','useGPU','only');
Y = net(X);
```

Train Network Using Checkpoint Saves

Here a network is trained with checkpoints saved at a rate no greater than once every two minutes.

```
[x,t] = vinyl_dataset;
net = fitnet([60 30]);
net = train(net,x,t,'CheckpointFile','MyCheckpoint','CheckpointDelay',120);
```

After a computer failure, the latest network can be recovered and used to continue training from the point of failure. The checkpoint file includes a structure variable `checkpoint`, which includes the network, training record, filename, time, and number.

```
[x,t] = vinyl_dataset;
load MyCheckpoint
net = checkpoint.net;
net = train(net,x,t,'CheckpointFile','MyCheckpoint');
```

Another use for the checkpoint feature is when you stop a parallel training session (started with the `'UseParallel'` parameter) even though the Neural Network Training Tool is not available during

parallel training. In this case, set a `'CheckpointFile'`, use Ctrl+C to stop training any time, then load your checkpoint file to get the network and training record.

Input Arguments

`net` — Input network

`network` object

Input network, specified as a `network` object. To create a `network` object, use for example, `feedforwardnet` or `narxnet`.

`X` — Network inputs

matrix | cell array | composite data | gpuArray

Network inputs, specified as an `R`-by-`Q` matrix or an `Ni`-by-`TS` cell array, where

- `R` is the input size
- `Q` is the batch size
- `Ni = net.numInputs`
- `TS` is the number of time steps

`train` arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

- The matrix format can be used if only one time step is to be simulated (`TS = 1`). It is convenient for networks with only one input and output, but can be used with networks that have more. When the network has multiple inputs, the matrix size is (sum of `Ri`)-by-`Q`.
- The cell array format is more general, and more convenient for networks with multiple inputs and outputs, allowing sequences of inputs to be presented. Each element `X{i,ts}` is an `Ri`-by-`Q` matrix, where `Ri = net.inputs{i}.size`.

If Composite data is used, then `'useParallel'` is automatically set to `'yes'`. The function takes Composite data and returns Composite results.

If gpuArray data is used, then `'useGPU'` is automatically set to `'yes'`. The function takes gpuArray data and returns gpuArray results

---

Note If a column of X contains at least one `NaN`, `train` does not use that column for training, testing, or validation.

`T` — Network targets

zeros (default) | matrix | cell array | composite data | gpuArray

Network targets, specified as a `U`-by-`Q` matrix or an `No`-by-`TS` cell array, where

- `U` is the output size
- `Q` is the batch size
- `No = net.numOutputs`

- `TS` is the number of time steps

`train` arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

- The matrix format can be used if only one time step is to be simulated (`TS = 1`). It is convenient for networks with only one input and output, but can be used with networks that have more. When the network has multiple inputs, the matrix size is (sum of `Ui`)-by-`Q`.
- The cell array format is more general, and more convenient for networks with multiple inputs and outputs, allowing sequences of inputs to be presented. Each element `T{i,ts}` is a `Ui`-by-`Q` matrix, where `Ui = net.outputs{i}.size`.

If Composite data is used, then `'useParallel'` is automatically set to `'yes'`. The function takes Composite data and returns Composite results.

If gpuArray data is used, then `'useGPU'` is automatically set to `'yes'`. The function takes gpuArray data and returns gpuArray results

Note that `T` is optional and need only be used for networks that require targets.

Note Any `NaN` values in the targets `T` are treated as missing data. If an element of `T` is `NaN`, that element is not used for training, testing, or validation.

`Xi` — Initial input delay conditions

zeros (default) | cell array | matrix

Initial input delay conditions, specified as an `Ni`-by-`ID` cell array or an `R`-by-`(ID*Q)` matrix, where

- `ID = net.numInputDelays`
- `Ni = net.numInputs`
- `R` is the input size
- `Q` is the batch size

For cell array input, the columns of `Xi` are ordered from the oldest delay condition to the most recent: `Xi{i,k}` is the input `i` at time `ts = k - ID`.

`Xi` is also optional and need only be used for networks that have input or layer delays.

`Ai` — Initial layer delay conditions

zeros (default) | cell array | matrix

Initial layer delay conditions, specified as a `Nl`-by-`LD` cell array or a (sum of `Si`)-by-(`LD*Q`) matrix, where

- `Nl = net.numLayers`
- `LD = net.numLayerDelays`
- `Si = net.layers{i}.size`
- `Q` is the batch size

For cell array input, the columns of `Ai` are ordered from the oldest delay condition to the most recent: `Ai{i,k}` is the layer output `i` at time `ts = k - LD`.

`EW` — Error weights
cell array

Error weights, specified as a `No`-by-`TS` cell array or a (sum of `Ui`)-by-`Q` matrix, where

- `No = net.numOutputs`
- `TS` is the number of time steps
- `Ui = net.outputs{i}.size`
- `Q` is the batch size

For cell array input. each element `EW{i,ts}` is a `Ui`-by-`Q` matrix, where

- `Ui = net.outputs{i}.size`
- `Q` is the batch size

The error weights `EW` can also have a size of 1 in place of all or any of `No`, `TS`, `Ui` or `Q`. In that case, `EW` is automatically dimension extended to match the targets `T`. This allows for conveniently weighting the importance in any dimension (such as per sample) while having equal importance across another (such as time, with `TS=1`). If all dimensions are 1, for instance if `EW = {1}`, then all target values are treated with the same importance. That is the default value of `EW`.

As noted above, the error weights `EW` can be of the same dimensions as the targets `T`, or have some dimensions set to 1. For instance if `EW` is 1-by-`Q`, then target samples will have different importances, but each element in a sample will have the same importance. If `EW` is (sum of `Ui`)-by-1, then each output element has a different importance, with all samples treated with the same importance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'useParallel','yes'`

`useParallel` — Option to specify parallel calculations
`'no'` (default) | `'yes'`

Option to specify parallel calculations, specified as `'yes'` or `'no'`.

- `'no'` – Calculations occur on normal MATLAB thread. This is the default `'useParallel'` setting.
- `'yes'` – Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.

`useGPU` — Option to specify GPU calculations
`'no'` (default) | `'yes'` | `'only'`

Option to specify GPU calculations, specified as `'yes'`, `'no'`, or `'only'`.

- `'no'` – Calculations occur on the CPU. This is the default `'useGPU'` setting.
- `'yes'` – Calculations occur on the current `gpuDevice` if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current `gpuDevice` is not supported,

calculations remain on the CPU. If `'useParallel'` is also `'yes'` and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.

- `'only'` – If no parallel pool is open, then this setting is the same as `'yes'`. If a parallel pool is open then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.

`showResources` — Option to show resources
`'no'` (default) | `'yes'`

Option to show resources, specified as `'yes'` or `'no'`.

- `'no'` – Do not display computing resources used at the command line. This is the default setting.
- `'yes'` – Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.

`reduction` — Memory reduction
1 (default) | positive integer

Memory reduction, specified as a positive integer.

For most neural networks, the default CPU training computation mode is a compiled MEX algorithm. However, for large networks the calculations might occur with a MATLAB calculation mode. This can be confirmed using `'showResources'`. If MATLAB is being used and memory is an issue, setting the reduction option to a value N greater than 1, reduces much of the temporary storage required to train by a factor of N, in exchange for longer training times.

`CheckpointFile` — Checkpoint file
`''` (default) | character vector

Checkpoint file, specified as a character vector.

The value for `'CheckpointFile'` can be set to a filename to save in the current working folder, to a file path in another folder, or to an empty string to disable checkpoint saves (the default value).

`CheckpointDelay` — Checkpoint delay
60 (default) | nonnegative integer

Checkpoint delay, specified as a nonnegative integer.

The optional parameter `'CheckpointDelay'` limits how often saves happen. Limiting the frequency of checkpoints can improve efficiency by keeping the amount of time saving checkpoints low compared to the time spent in calculations. It has a default value of 60, which means that checkpoint saves do not happen more than once per minute. Set the value of `'CheckpointDelay'` to 0 if you want checkpoint saves to occur only once every epoch.

Output Arguments

`trainedNet` — Trained network
`network` object

Trained network, returned as a `network` object.

`tr` — Training record
structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`)
- A list of training state names (`states`)
- Fields for each state name recording its value throughout training
- Best performances of the network, evaluated at each epoch: best performance on the training set (`best_perf`), best performance on the validation set (`best_vperf`), and best performance on the test set (`best_tperf`)

Algorithms

`train` calls the function indicated by `net.trainFcn`, using the training parameter values indicated by `net.trainParam`.

Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function `net.trainFcn` occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly for each epoch from concurrent input vectors (or sequences). `competlayer` returns networks that use `trainru`, a training function that does this.

See Also

`init` | `revert` | `sim` | `adapt`

Introduced before R2006a

trainb

Batch training with weight and bias learning rules

Syntax

```
net.trainFcn = 'trainb'
[net,tr] = train(net,...)
```

Description

`trainb` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainb'`, thus:

`net.trainFcn = 'trainb'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainb`.

`trainb` trains a network with weight and bias learning rules with batch updates. The weights and biases are updated at the end of an entire pass through the input data.

Training occurs according to `trainb`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `trainb` by calling `linearlayer`.

To prepare a custom network to be trained with `trainb`,

1. Set `net.trainFcn` to `'trainb'`. This sets `net.trainParam` to `trainb`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function. Set each `net.layerWeights{i,j}.learnFcn` to a learning function. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.

2 Set weight and bias learning parameters to desired values.

3 Call `train`.

Algorithms

Each weight and bias is updated according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions is met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance is minimized to the `goal`.
- The maximum amount of `time` is exceeded.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`linearlayer` | `train`

Introduced before R2006a

trainbfg

BFGS quasi-Newton backpropagation

Syntax

```
net.trainFcn = 'trainbfg'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'trainbfg'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `trainbfg`.

`trainbfg` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

Training occurs according to `trainbfg` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-6`.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.searchFcn` — Name of line search routine to use. The default value is `'srchbac'`.

Parameters related to line search methods (not all used for all methods):

- `net.trainParam.scal_tol` — Divide into delta to determine tolerance for linear search. The default value is 20.
- `net.trainParam.alpha` — Scale factor that determines sufficient reduction in perf. The default value is `0.001`.
- `net.trainParam.beta` — Scale factor that determines sufficiently large step size. The default value is `0.1`.
- `net.trainParam.delta` — Initial step size in interval location step. The default value is `0.01`.
- `net.trainParam.gamma` — Parameter to avoid small reductions in performance, usually set to 0.1 (see `srch_cha`). The default value is `0.1`.
- `net.trainParam.low_lim` — Lower limit on change in step size. The default value is `0.1`.
- `net.trainParam.up_lim` — Upper limit on change in step size. The default value is `0.5`.

- `net.trainParam.maxstep` — Maximum step length. The default value is `100`.
- `net.trainParam.minstep` — Minimum step length. The default value is `1.0e-6`.
- `net.trainParam.bmax` — Maximum step size. The default value is `26`.
- `net.trainParam.batch_frag` — In case of multiple batches, they are considered independent. Any nonzero value implies a fragmented batch, so the final layer's conditions of a previous trained epoch are used as initial conditions for the next epoch. The default value is `0`.

Examples

Train Neural Network Using `trainbfg` Train Function

This example shows how to train a neural network using the `trainbfg` train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainbfg');
net = train(net, x, t);
y = net(x);
```

Input Arguments

`net` — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object..

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `trainbfg` with `feedfowardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainbfg`:

1. Set `NET.trainFcn` to `'trainbfg'`. This sets `NET.trainParam` to `trainbfg`'s default parameters.
2. Set `NET.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainbfg`.

BFGS Quasi-Newton Backpropagation

Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where $\mathbf{A}_k^{-1}$ is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton's method, but which does not require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm is implemented in the `trainbfg` routine.

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is $n \times n$, where n is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

Algorithms

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is a approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

[1] Gill, Murray, & Wright, *Practical Optimization*, 1981

See Also

`cascadeforwardnet` | `feedforwardnet` | `traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainoss`

Introduced before R2006a

trainbfgc

BFGS quasi-Newton backpropagation for use with NN model reference adaptive controller

Syntax

```
[net,TR,Y,E,Pf,Af,flag_stop] = trainbfgc(net,P,T,Pi,Ai,epochs,TS,Q)
info = trainbfgc(code)
```

Description

`trainbfgc` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method. This function is called from `nnmodref`, a GUI for the model reference adaptive control Simulink block.

`[net,TR,Y,E,Pf,Af,flag_stop] = trainbfgc(net,P,T,Pi,Ai,epochs,TS,Q)` takes these inputs,

`net`	Neural network
`P`	Delayed input vectors
`T`	Layer target vectors
`Pi`	Initial input delay conditions
`Ai`	Initial layer delay conditions
`epochs`	Number of iterations for training
`TS`	Time steps
`Q`	Batch size

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch:
	`TR.epoch` Epoch number
	`TR.perf` Training performance
	`TR.vperf` Validation performance
	`TR.tperf` Test performance
`Y`	Network output for last epoch
`E`	Layer errors for last epoch
`Pf`	Final input delay conditions
`Af`	Collective layer outputs for last epoch
`flag_stop`	Indicates if the user stopped the training

Training occurs according to `trainbfgc`'s training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to train in seconds
net.trainParam.min_grad	1e-6	Minimum performance gradient
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.searchFcn	'srchbacx'	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine tolerance for linear search.
net.trainParam.alpha	0.001	Scale factor that determines sufficient reduction in perf
net.trainParam.beta	0.1	Scale factor that determines sufficiently large step size
net.trainParam.delta	0.01	Initial step size in interval location step
net.trainParam.gama	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

info = trainbfgc(code) returns useful information for each code character vector:

'pnames'	Names of training parameters
'pdefaults'	Default training parameters

Algorithms

trainbfgc can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where gX is the gradient and H is an approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Precision problems have occurred in the matrix inversion.

References

Gill, Murray, and Wright, *Practical Optimization*, 1981

Introduced in R2006a

trainbr

Bayesian regularization backpropagation

Syntax

```
net.trainFcn = 'trainbr'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'trainbr'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `trainbr`.

`trainbr` is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well. The process is called Bayesian regularization.

Training occurs according to `trainbr` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.mu` — Marquardt adjustment parameter. The default value is 0.005.
- `net.trainParam.mu_dec` — Decrease factor for `mu`. The default value is 0.1.
- `net.trainParam.mu_inc` — Increase factor for `mu`. The default value is 10.
- `net.trainParam.mu_max` — Maximum value for mu. The default value is `1e10`.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `inf`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-7`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.

Validation stops are disabled by default (`max_fail = inf`) so that training can continue until an optimal combination of errors and weights is found. However, some weight/bias minimization can still be achieved with shorter training times if validation is enabled by setting `max_fail` to 6 or some other strictly positive value.

Examples

Train Network with `'trainbr'`

This example shows how to solve a problem consisting of inputs `p` and targets `t` by using a network. It involves fitting a noisy sine wave.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

A feed-forward network is created with a hidden layer of 2 neurons.

```
net = feedforwardnet(2,'trainbr');
```

Here the network is trained and tested.

```
net = train(net,p,t);
a = net(p)
```

Input Arguments

`net` — Input network

matrix

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object.

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

Limitations

This function uses the Jacobian for calculations, which assumes that performance is a mean or sum of squared errors. Therefore networks trained with this function must use either the `mse` or `sse` performance function.

More About

Network Use

You can create a standard network that uses `trainbr` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainbr`,

1. Set `NET.trainFcn` to `'trainbr'`. This sets `NET.trainParam` to `trainbr`'s default parameters.
2. Set `NET.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainbr`. See `feedforwardnet` and `cascadeforwardnet` for examples.

Algorithms

`trainbr` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities. See MacKay (*Neural Computation*, Vol. 4, No. 3, 1992, pp. 415 to 447) and Foresee and Hagan (*Proceedings of the International Joint Conference on Neural Networks*, June, 1997) for more detailed discussions of Bayesian regularization.

This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian `jX` of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value `mu` is increased by `mu_inc` until the change shown above results in a reduced performance value. The change is then made to the network, and `mu` is decreased by `mu_dec`.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- `mu` exceeds `mu_max`.

References

[1] MacKay, David J. C. "Bayesian interpolation." *Neural computation.* Vol. 4, No. 3, 1992, pp. 415–447.

[2] Foresee, F. Dan, and Martin T. Hagan. "Gauss-Newton approximation to Bayesian learning." *Proceedings of the International Joint Conference on Neural Networks*, June, 1997.

See Also

`cascadeforwardnet` | `feedforwardnet` | `traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainbfg`

Introduced before R2006a

trainbu

Batch unsupervised weight/bias training

Syntax

```
net.trainFcn = 'trainbu'
[net,tr] = train(net,...)
```

Description

`trainbu` trains a network with weight and bias learning rules with batch updates. Weights and biases updates occur at the end of an entire pass through the input data.

`trainbu` is not called directly. Instead the `train` function calls it for networks whose `NET.trainFcn` property is set to `'trainbu'`, thus:

`net.trainFcn = 'trainbu'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainbu`.

Training occurs according to `trainbu` training parameters, shown here with the following default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Validation and test vectors have no impact on training for this function, but act as independent measures of network generalization.

Network Use

You can create a standard network that uses `trainbu` by calling `selforgmap`. To prepare a custom network to be trained with `trainbu`:

1. Set `NET.trainFcn` to `'trainbu'`. (This option sets `NET.trainParam` to `trainbu` default parameters.)
2. Set each `NET.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `NET.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `NET.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network:

1. Set `NET.trainParam` properties to desired values.

2 Set weight and bias learning parameters to desired values.

3 Call `train`.

See `selforgmap` for training examples.

Algorithms

Each weight and bias updates according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions is met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance is minimized to the `goal`.
- The maximum amount of `time` is exceeded.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`train` | `trainb`

Introduced in R2010b

trainc

Cyclical order weight/bias training

Syntax

```
net.trainFcn = 'trainc'
[net,tr] = train(net,...)
```

Description

`trainc` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainc'`, thus:

`net.trainFcn = 'trainc'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainc`.

`trainc` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in cyclic order.

Training occurs according to `trainc` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `trainc` by calling `competlayer`. To prepare a custom network to be trained with `trainc`,

1. Set `net.trainFcn` to `'trainc'`. This sets `net.trainParam` to `trainc`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function. Set each `net.layerWeights{i,j}.learnFcn` to a learning function. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `perceptron` for training examples.

Algorithms

For each epoch, each vector (or sequence) is presented in order to the network, with the weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance is minimized to the `goal`.
- The maximum amount of `time` is exceeded.

See Also

`competlayer` | `train`

Introduced before R2006a

traincgb

Conjugate gradient backpropagation with Powell-Beale restarts

Syntax

```
net.trainFcn = 'traincgb'
[net,tr] = train(net,...)
```

Description

`traincgb` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

`net.trainFcn = 'traincgb'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincgb`.

Training occurs according to `traincgb` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length

net.trainParam.bmax	26	Maximum step size

Network Use

You can create a standard network that uses `traincgb` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `traincgb`,

1. Set `net.trainFcn` to `'traincgb'`. This sets `net.trainParam` to `traincgb`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgb`.

Examples

Train Neural Network Using `traincgb` Train Function

This example shows how to train a neural network using the `traincgb` train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgb');
net = train(net, x, t);
y = net(x);
```

More About

Powell-Beale Algorithm

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. This technique restarts if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

$$\left|\mathbf{g}_{k-1}^{T}\mathbf{g}_{k}\right| \geq 0.2\left\|\mathbf{g}_{k}\right\|^{2}$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The `traincgb` routine has somewhat better performance than `traincgp` for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

Algorithms

`traincgb` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

```
dX = -gX + dX_old*Z;
```

where `gX` is the gradient. The parameter `Z` can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming,* Vol. 12, 1977, pp. 241 to 254, for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Powell, M.J.D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, Vol. 12, 1977, pp. 241–254

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgp` | `traincgf` | `trainscg` | `trainoss` | `trainbfg`

Introduced before R2006a

traincgf

Conjugate gradient backpropagation with Fletcher-Reeves updates

Syntax

```
net.trainFcn = 'traincgf'
[net,tr] = train(net,...)
```

Description

`traincgf` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Fletcher-Reeves updates.

`net.trainFcn = 'traincgf'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincgf`.

Training occurs according to `traincgf` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length

net.trainParam.bmax	26	Maximum step size

Network Use

You can create a standard network that uses traincgf with feedforwardnet or cascadeforwardnet.

To prepare a custom network to be trained with traincgf,

1. Set net.trainFcn to 'traincgf'. This sets net.trainParam to traincgf's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with traincgf.

Examples

Train Neural Network Using traincgf Train Function

This example shows how to train a neural network using the traincgf train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgf');
net = train(net, x, t);
y = net(x);
```

More About

Conjugate Gradient Algorithms

All the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k \alpha_k \mathbf{p}_k$$

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of the conjugate gradient algorithm are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than `trainrp`, although the results vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms. Therefore, these algorithms are good for networks with a large number of weights.

Try the *Neural Network Design* demonstration `nnd12cg` [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

Algorithms

`traincgf` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction, according to the formula

```
dX = -gX + dX_old*Z;
```

where `gX` is the gradient. The parameter `Z` can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

```
Z = normnew_sqr/norm_sqr;
```

where `norm_sqr` is the norm square of the previous gradient and `normnew_sqr` is the norm square of the current gradient. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgb` | `trainscg` | `traincgp` | `trainoss` | `trainbfg`

Introduced before R2006a

traincgp

Conjugate gradient backpropagation with Polak-Ribiére updates

Syntax

```
net.trainFcn = 'traincgp'
[net,tr] = train(net,...)
```

Description

`traincgp` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Polak-Ribiére updates.

`net.trainFcn = 'traincgp'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincgp`.

Training occurs according to `traincgp` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length

net.trainParam.bmax	26	Maximum step size

Network Use

You can create a standard network that uses traincgp with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traincgp,

1. Set net.trainFcn to 'traincgp'. This sets net.trainParam to traincgp's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with traincgp.

Examples

Train Neural Network Using traincgp Train Function

This example shows how to train a neural network using the traincgp train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgp');
net = train(net, x, t);
y = net(x);
```

More About

Conjugate Gradient Backpropagation with Polak-Ribiére Updates

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, traincgf, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

The traincgp routine has performance similar to traincgf. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

Algorithms

`traincgp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

```
dX = -gX + dX_old*Z;
```

where `gX` is the gradient. The parameter `Z` can be computed in several different ways. For the Polak-Ribiére variation of conjugate gradient, it is computed according to

```
Z = ((gX - gX_old)'*gX)/norm_sqr;
```

where `norm_sqr` is the norm square of the previous gradient, and `gX_old` is the gradient on the previous iteration. See page 78 of Scales (*Introduction to Non-Linear Optimization*, 1985) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `trainoss` | `trainbfg`

Introduced before R2006a

traingd

Gradient descent backpropagation

Syntax

```
net.trainFcn = 'traingd'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'traingd'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `traingd`.

`traingd` is a network training function that updates weight and bias values according to gradient descent.

Training occurs according to `traingd` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.lr` — Learning rate. The default value is 0.01.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-5`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.

Input Arguments

net — Input network

network object

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

trainedNet — Trained network

network object

Trained network, returned as a `network` object.

`tr` — Training record
structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `traingd` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingd`,

1. Set `net.trainFcn` to `'traingd'`. This sets `net.trainParam` to `traingd`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingd`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

Gradient Descent Backpropagation

The batch steepest descent training function is `traingd`. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network `trainFcn` to `traingd`, and then call the function `train`. There is only one training function associated with a given network.

There are seven training parameters associated with `traingd`:

- `epochs`
- `show`
- `goal`
- `time`
- `min_grad`
- `max_fail`
- `lr`

The learning rate `lr` is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. See page 12-8 of [HDB96] for a discussion of the choice of learning rate.

The training status is displayed for every `show` iterations of the algorithm. (If `show` is set to `NaN`, then the training status is never displayed.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds `epochs`, if the performance function drops below `goal`, if the magnitude of the gradient is less than `mingrad`, or if the training time is longer than `time` seconds. `max_fail`, which is associated with the early stopping technique, is discussed in Improving Generalization.

The following code creates a training set of inputs `p` and targets `t`. For batch training, all the input vectors are placed in one matrix.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
```

Create the feedforward network.

```
net = feedforwardnet(3,'traingd');
```

In this simple example, turn off a feature that is introduced later.

```
net.divideFcn = '';
```

At this point, you might want to modify some of the default training parameters.

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

If you want to use the default training parameters, the preceding commands are not necessary.

Now you are ready to train the network.

```
[net,tr] = train(net,p,t);
```

The training record `tr` contains information about the progress of training.

Now you can simulate the trained network to obtain its response to the inputs in the training set.

```
a = net(p)
a =
   -1.0026   -0.9962   1.0010   0.9960
```

Try the *Neural Network Design* demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the batch gradient descent algorithm.

Algorithms

`traingd` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent:

```
dX = lr * dperf/dX
```

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm`

Introduced before R2006a

traingda

Gradient descent with adaptive learning rate backpropagation

Syntax

```
net.trainFcn = 'traingda'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'traingda'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `traingda`.

`traingda` is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate.

Training occurs according to `traingda` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.lr` — Learning rate. The default value is 0.01.
- `net.trainParam.lr_inc` — Ratio to increase learning rate. The default value is 1.05.
- `net.trainParam.lr_dec` — Ratio to decrease learning rate. The default value is 0.7.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.max_perf_inc` — Maximum performance increase. The default value is `1.04`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-5`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.

Input Arguments

net — Input network

network

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

trainedNet — Trained network

network

Trained network, returned as a `network` object..

`tr` — Training record
structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `traingda` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingda`,

1. Set `net.trainFcn` to `'traingda'`. This sets `net.trainParam` to `traingda`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingda`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

Gradient Descent with Adaptive Learning Rate Backpropagation

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

You can improve the performance of the steepest descent algorithm if you allow the learning rate to change during the training process. An adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by `traingd`. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, `max_perf_inc` (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by `lr_dec` = 0.7). Otherwise, the new weights, etc., are

kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by `lr_inc` = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it is decreased until stable learning resumes.

Try the *Neural Network Design* demonstration `nnd12vl` [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function `traingda`, which is called just like `traingd`, except for the additional training parameters `max_perf_inc`, `lr_dec`, and `lr_inc`. Here is how it is called to train the previous two-layer network:

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingda');
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net = train(net,p,t);
y = net(p)
```

Algorithms

`traingda` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `dperf` with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

```
dX = lr*dperf/dX
```

At each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor `lr_inc`. If performance increases by more than the factor `max_perf_inc`, the learning rate is adjusted by the factor `lr_dec` and the change that increased the performance is not made.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`traingd` | `traingdm` | `traingdx` | `trainlm`

Introduced before R2006a

traingdm

Gradient descent with momentum backpropagation

Syntax

```
net.trainFcn = 'traingdm'
[net,tr] = train(net,...)
```

Description

`traingdm` is a network training function that updates weight and bias values according to gradient descent with momentum.

`net.trainFcn = 'traingdm'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traingdm`.

Training occurs according to `traingdm` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.mc`	`0.9`	Momentum constant
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `traingdm` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingdm`,

1. Set `net.trainFcn` to `'traingdm'`. This sets `net.trainParam` to `traingdm`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingdm`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

More About

Gradient Descent with Momentum

In addition to `traingd`, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by `traingdm`, allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12–9 of [HDB96] for a discussion of momentum.

Gradient descent with momentum depends on two training parameters. The parameter `lr` indicates the learning rate, similar to the simple gradient descent. The parameter `mc` is the momentum constant that defines the amount of momentum. `mc` is set between 0 (no momentum) and values close to 1 (lots of momentum). A momentum constant of 1 results in a network that is completely insensitive to the local gradient and, therefore, does not learn properly.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingdm');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net = train(net,p,t);
y = net(p)
```

Try the *Neural Network Design* demonstration `nnd12mo` [HDB96] for an illustration of the performance of the batch momentum algorithm.

Algorithms

`traingdm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`traingd` | `traingda` | `traingdx` | `trainlm`

Introduced before R2006a

traingdx

Gradient descent with momentum and adaptive learning rate backpropagation

Syntax

```
net.trainFcn = 'traingdx'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'traingdx'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `traingdx`.

`traingdx` is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

Training occurs according to `traingdx` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.lr` — Learning rate. The default value is 0.01.
- `net.trainParam.lr_inc` — Ratio to increase learning rate. The default value is 1.05.
- `net.trainParam.lr_dec` — Ratio to decrease learning rate. The default value is 0.7.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.max_perf_inc` — Maximum performance increase. The default value is `1.04`.
- `net.trainParam.mc` — Momentum constant. The default value is `0.9`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-5`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.

Input Arguments

net — Input network

matrix

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object.

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `traingdx` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingdx`,

1. Set `net.trainFcn` to `'traingdx'`. This sets `net.trainParam` to `traingdx`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingdx`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

Algorithms

The function `traingdx` combines adaptive learning rate with momentum training. It is invoked in the same way as `traingda`, except that it has the momentum coefficient `mc` as an additional training parameter.

`traingdx` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*mc*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor `lr_inc`. If performance increases by more than the factor `max_perf_inc`, the learning rate is adjusted by the factor `lr_dec` and the change that increased the performance is not made.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

`traingd` | `traingda` | `traingdm` | `trainlm`

Introduced before R2006a

trainlm

Levenberg-Marquardt backpropagation

Syntax

```
net.trainFcn = 'trainlm'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'trainlm'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `trainlm`.

`trainlm` is a network training function that updates weight and bias values according to Levenberg-Marquardt optimization.

`trainlm` is often the fastest backpropagation algorithm in the toolbox, and is highly recommended as a first-choice supervised algorithm, although it does require more memory than other algorithms.

Training occurs according to `trainlm` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-7`.
- `net.trainParam.mu` — Initial `mu`. The default value is 0.001.
- `net.trainParam.mu_dec` — Decrease factor for `mu`. The default value is 0.1.
- `net.trainParam.mu_inc` — Increase factor for `mu`. The default value is 10.
- `net.trainParam.mu_max` — Maximum value for `mu`. The default value is `1e10`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

Examples

Train Neural Network Using `trainlm` Train Function

This example shows how to train a neural network using the `trainlm` train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainlm');
net = train(net, x, t);
y = net(x);
```

Input Arguments

`net` — Input network

matrix

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object.

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

Limitations

This function uses the Jacobian for calculations, which assumes that performance is a mean or sum of squared errors. Therefore, networks trained with this function must use either the `mse` or `sse` performance function.

More About

Levenberg-Marquardt Algorithm

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} \tag{2-1}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e} \tag{2-2}$$

where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $\mathbf{e}$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an efficient implementation in MATLAB® software, because the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB environment.

Try the *Neural Network Design* demonstration `nnd12m` [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

Network Use

You can create a standard network that uses `trainlm` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainlm`,

1. Set `NET.trainFcn` to `trainlm`. This sets `NET.trainParam` to `trainlm`'s default parameters.
2. Set `NET.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainlm`. See `feedforwardnet` and `cascadeforwardnet` for examples.

Algorithms

`trainlm` supports training with validation and test vectors if the network's `NET.divideFcn` property is set to a data division function. Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`trainlm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate the Jacobian `jX` of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value `mu` is increased by `mu_inc` until the change above results in a reduced performance value. The change is then made to the network and `mu` is decreased by `mu_dec`.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- `mu` exceeds `mu_max`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

See Also

Introduced before R2006a

trainoss

One-step secant backpropagation

Syntax

```
net.trainFcn = 'trainoss'
[net,tr] = train(net,...)
```

Description

`trainoss` is a network training function that updates weight and bias values according to the one-step secant method.

`net.trainFcn = 'trainoss'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainoss`.

Training occurs according to `trainoss` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.searchFcn`	`'srchbac'`	Name of line search routine to use
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length

net.trainParam.bmax	26	Maximum step size

Network Use

You can create a standard network that uses `trainoss` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainoss`:

1. Set `net.trainFcn` to `'trainoss'`. This sets `net.trainParam` to `trainoss`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainoss`.

Examples

Train Neural Network Using `trainoss` Train Function

This example shows how to train a neural network using the `trainoss` train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainoss');
net = train(net, x, t);
y = net(x);
```

More About

One Step Secant Method

Because the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

Algorithms

`trainoss` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients, according to the following formula:

```
dX = -gX + Ac*X_step + Bc*dgX;
```

where `gX` is the gradient, `X_step` is the change in the weights on the previous iteration, and `dgX` is the change in the gradient from the last iteration. See Battiti (*Neural Computation,* Vol. 4, 1992, pp. 141–166) for a more detailed discussion of the one-step secant algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

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

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainbfg`

Introduced before R2006a

trainr

Random order incremental training with learning functions

Syntax

```
net.trainFcn = 'trainr'
[net,tr] = train(net,...)
```

Description

`trainr` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainr'`, thus:

`net.trainFcn = 'trainr'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainr`.

`trainr` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainr` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `trainr` by calling `competlayer` or `selforgmap`. To prepare a custom network to be trained with `trainr`,

1. Set `net.trainFcn` to `'trainr'`. This sets `net.trainParam` to `trainr`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.

3 Call `train`.

See `help competlayer` and `help selforgmap` for training examples.

Algorithms

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 accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance is minimized to the `goal`.
- The maximum amount of `time` is exceeded.

See Also

`train`

Introduced before R2006a

trainrp

Resilient backpropagation

Syntax

```
net.trainFcn = 'trainrp'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'trainrp'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `trainrp`.

`trainrp` is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (Rprop).

Training occurs according to `trainrp` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is `1000`.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is `25`.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.goal` — Performance goal. The default value is `0`.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-5`.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.lr` — Learning rate. The default value is `0.01`.
- `net.trainParam.delt_inc` — Increment to weight change. The default value is `1.2`.
- `net.trainParam.delt_dec` — Decrement to weight change. The default value is `0.5`.
- `net.trainParam.delta0` — Initial weight change. The default value is `0.07`.
- `net.trainParam.deltamax` — Maximum weight change. The default value is `50.0`.

Examples

Solve Problems with Network Trained with `'trainrp'`

This example shows how to train a feed-forward network with a `trainrp` training function to solve a problem with inputs `p` and targets `t`.

Create the inputs `p` and the targets `t` that you want to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Create a two-layer feed-forward network with two hidden neurons and this training function.

```
net = feedforwardnet(2,'trainrp');
```

Train and test the network.

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = net(p)
```

For more examples, see `help feedforwardnet` and `help cascadeforwardnet`.

Input Arguments

`net` — Input network

matrix

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object.

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `trainrp` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `trainrp`,

1. Set `net.trainFcn` to `'trainrp'`. This sets `net.trainParam` to `trainrp`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainrp`.

Resilient Backpropagation

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called "squashing" functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative can determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor `delt_inc` whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor `delt_dec` whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value remains the same. Whenever the weights are oscillating, the weight change is reduced. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases. A complete description of the Rprop algorithm is given in [RiBr93].

The following code recreates the previous network and trains it using the Rprop algorithm. The training parameters for `trainrp` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `delt_inc`, `delt_dec`, `delta0`, and `deltamax`. The first eight parameters have been previously discussed. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, the training parameters are left at the default values:

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'trainrp');
net = train(net,p,t);
y = net(p)
```

`rprop` is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. You do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

Algorithms

`trainrp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
dX = deltaX.*sign(gX);
```

where the elements of `deltaX` are all initialized to `delta0`, and `gX` is the gradient. At each iteration the elements of `deltaX` are modified. If an element of `gX` changes sign from one iteration to the next, then the corresponding element of `deltaX` is decreased by `delta_dec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltaX` is increased by `delta_inc`. See 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, pp. 586–591.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

[1] 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, pp. 586–591.

See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgp` | `traincgf` | `traincgb` | `trainscg` | `trainoss` | `trainbfg`

Introduced before R2006a

trainru

Unsupervised random order weight/bias training

Syntax

```
net.trainFcn = 'trainru'
[net,tr] = train(net,...)
```

Description

`trainru` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainru'`, thus:

`net.trainFcn = 'trainru'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainru`.

`trainru` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainru` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`Inf`	Maximum time to train in seconds

Network Use

To prepare a custom network to be trained with `trainru`,

1. Set `net.trainFcn` to `'trainru'`. This sets `net.trainParam` to `trainru`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

Algorithms

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 accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.

See Also

`train` | `trainr`

Introduced in R2010b

trains

Sequential order incremental training with learning functions

Syntax

```
net.trainFcn = 'trains'
[net,tr] = train(net,...)
```

Description

`trains` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trains'`, thus:

`net.trainFcn = 'trains'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trains`.

`trains` trains a network with weight and bias learning rules with sequential updates. The sequence of inputs is presented to the network with updates occurring after each time step.

This incremental training algorithm is commonly used for adaptive applications.

Training occurs according to `trains` training parameters, shown here with their default values:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`Inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `trains` for adapting by calling `perceptron` or `linearlayer`.

To prepare a custom network to adapt with `trains`,

1. Set `net.adaptFcn` to `'trains'`. This sets `net.adaptParam` to `trains`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function. Set each `net.layerWeights{i,j}.learnFcn` to a learning function. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To allow the network to adapt,

1. Set weight and bias learning parameters to desired values.

2 Call `adapt`.

See `help perceptron` and `help linearlayer` for adaption examples.

Algorithms

Each weight and bias is updated according to its learning function after each time step in the input sequence.

See Also

`train` | `trainb` | `trainc` | `trainr`

Introduced before R2006a

trainscg

Scaled conjugate gradient backpropagation

Syntax

```
net.trainFcn = 'trainscg'
[trainedNet,tr] = train(net,...)
```

Description

`net.trainFcn = 'trainscg'` sets the network `trainFcn` property.

`[trainedNet,tr] = train(net,...)` trains the network with `trainscg`.

`trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

Training occurs according to `trainscg` training parameters, shown here with their default values:

- `net.trainParam.epochs` — Maximum number of epochs to train. The default value is 1000.
- `net.trainParam.show` — Epochs between displays (`NaN` for no displays). The default value is 25.
- `net.trainParam.showCommandLine` — Generate command-line output. The default value is `false`.
- `net.trainParam.showWindow` — Show training GUI. The default value is `true`.
- `net.trainParam.goal` — Performance goal. The default value is 0.
- `net.trainParam.time` — Maximum time to train in seconds. The default value is `inf`.
- `net.trainParam.min_grad` — Minimum performance gradient. The default value is `1e-6`.
- `net.trainParam.max_fail` — Maximum validation failures. The default value is `6`.
- `net.trainParam.mu` — Marquardt adjustment parameter. The default value is 0.005.
- `net.trainParam.sigma` — Determine change in weight for second derivative approximation. The default value is `5.0e-5`.
- `net.trainParam.lambda` — Parameter for regulating the indefiniteness of the Hessian. The default value is `5.0e-7`.

Examples

Train Network with `'trainscg'`

This example shows how to solve a problem consisting of inputs `p` and targets `t` by using a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

A two-layer feed-forward network with two hidden neurons and this training function is created.

```
net = feedforwardnet(2,'trainscg');
```

Here the network is trained and tested.

```
net = train(net,p,t);
a = net(p)
```

See `help feedforwardnet` and `help cascadeforwardnet` for other examples.

Input Arguments

`net` — Input network

matrix

Input network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

Output Arguments

`trainedNet` — Trained network

network

Trained network, returned as a `network` object.

`tr` — Training record

structure

Training record (`epoch` and `perf`), returned as a structure whose fields depend on the network training function (`net.NET.trainFcn`). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network (`best_perf`, `best_vperf`, `best_tperf`)

More About

Network Use

You can create a standard network that uses `trainscg` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainscg`,

1. Set `net.trainFcn` to `'trainscg'`. This sets `net.trainParam` to `trainscg`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainscg`.

Algorithms

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`.

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincgf`, and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*, Vol. 6, 1993, pp. 525–533) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

[1] Moller. *Neural Networks*, Vol. 6, 1993, pp. 525–533

See Also

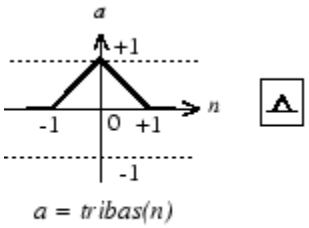
`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainbfg` | `traincgp` | `trainoss`

Introduced before R2006a

tribas

Triangular basis transfer function

Graph and Symbol



Triangular Basis Function

Syntax

```
A = tribas(N,FP)
```

Description

`tribas` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`A = tribas(N,FP)` takes `N` and optional function parameters,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, an S-by-Q matrix of the triangular basis function applied to each element of `N`.

`info = tribas('code')` can take the following forms to return specific information:

`tribas('name')` returns the name of this function.

`tribas('output',FP)` returns the `[min max]` output range.

`tribas('active',FP)` returns the `[min max]` active input range.

`tribas('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is S-by-S-by-Q or S-by-Q.

`tribas('fpnames')` returns the names of the function parameters.

`tribas('fpdefaults')` returns the default function parameters.

Examples

Here you create a plot of the `tribas` transfer function.

```
n = -5:0.1:5;
a = tribas(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'tribas';
```

Algorithms

```
a = tribas(n) = 1 - abs(n), if -1 <= n <= 1
              = 0, otherwise
```

See Also

`sim` | `radbas`

Introduced before R2006a

tritop

Triangle layer topology function

Syntax

```
pos = tritop(dimensions)
```

Description

`tritop` calculates neuron positions for layers whose neurons are arranged in an `N`-dimensional triangular grid.

`pos = tritop(dimensions)` takes one argument:

`dimensions`	Row vector of dimension sizes

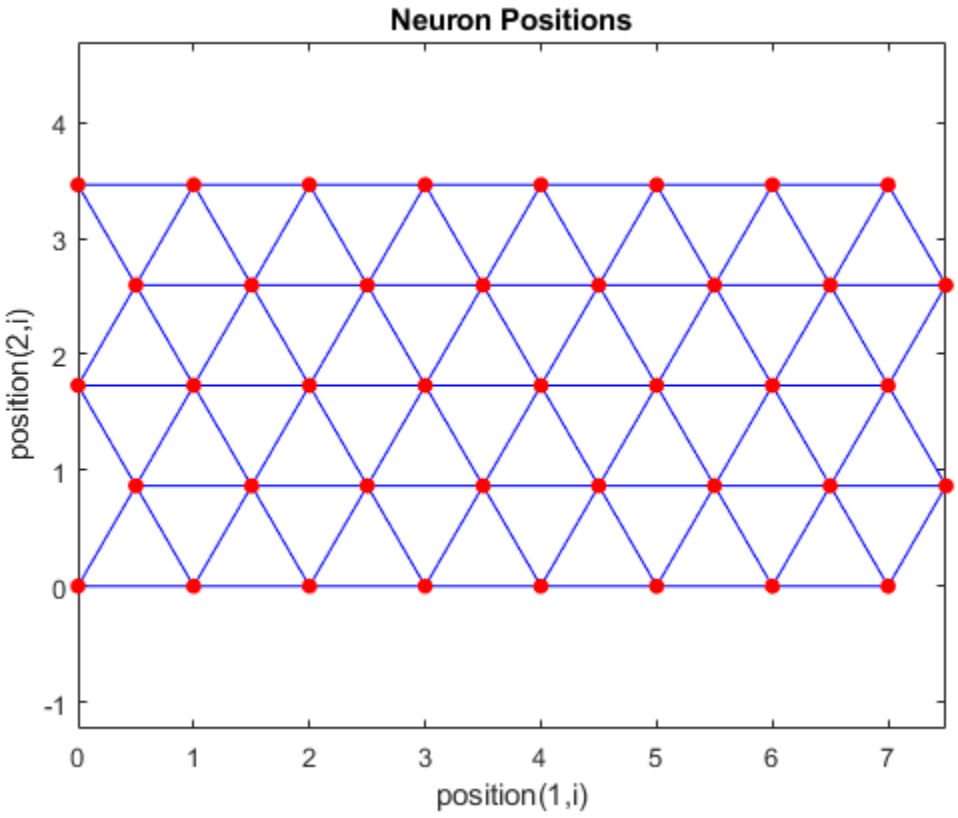
and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Triangular Pattern

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 triangular grid.

```
pos = tritop([8 5]);
plotsom(pos)
```



See Also

`gridtop` | `hextop` | `randtop`

Introduced in R2010b

unconfigure

Unconfigure network inputs and outputs

Syntax

```
unconfigure(net)
unconfigure(net, 'inputs', i)
unconfigure(net, 'outputs', i)
```

Description

`unconfigure(net)` returns a network with its input and output sizes set to 0, its input and output processing settings and related weight initialization settings set to values consistent with zero-sized signals. The new network will be ready to be reconfigured for data of the same or different dimensions than it was previously configured for.

`unconfigure(net, 'inputs', i)` unconfigures the inputs indicated by the indices `i`. If no indices are specified, all inputs are unconfigured.

`unconfigure(net, 'outputs', i)` unconfigures the outputs indicated by the indices `i`. If no indices are specified, all outputs are unconfigured.

Examples

Here a network is configured for a simple fitting problem, and then unconfigured.

```
[x,t] = simplefit_dataset;
net = fitnet(10);
view(net)
net = configure(net,x,t);
view(net)
net = unconfigure(net)
view(net)
```

See Also

`configure` | `isconfigured`

Introduced in R2010b

vec2ind

Convert indices to vectors

Syntax

```
[ind,N] = vec2ind(vec)
```

Description

`[ind,N] = vec2ind(vec)` takes a matrix of vectors, each containing a single `1` and returns the indices of the ones, `ind`, and the number of rows in `vec`, `N`.

`ind2vec` and `vec2ind` allow indices to be represented either by themselves or as vectors containing a 1 in the row of the index they represent.

Examples

Convert Three Vectors to Indices and Back

This example shows how to convert three vectors to indices and back, using both the `ind2vec` and `vec2ind` functions.

Define three vector with all zeros in the last row and convert it to indices.

```
vec = [0 0 1 0; 1 0 0 0; 0 1 0 0]'
[ind,n] = vec2ind(vec)
```

```
vec =
     0     1     0
     0     0     1
     1     0     0
     0     0     0
ind =
     3     1     2

n =
     4
```

Convert the indices to vector, while preserving the number of rows.

```
vec2 = full(ind2vec(ind,n))
```

```
vec2 =
     0     1     0
     0     0     1
```

```
     1     0     0
     0     0     0
```

Input Arguments

`vec` — Matrix of vectors

matrix

Vector representation of the indices, specified as a matrix of vectors, each containing a single `1`.

Output Arguments

`ind` — Indices

row vector

Indices, returned as a row vector.

`N` — Number of rows

scalar

Number of rows of the input matrix, returned as a scalar.

See Also

`ind2vec` | `sub2ind` | `ind2sub`

Introduced before R2006a

view

View shallow neural network

Syntax

```
view(net)
```

Description

`view(net)` opens a window that shows your shallow neural network (specified in `net`) as a graphical diagram.

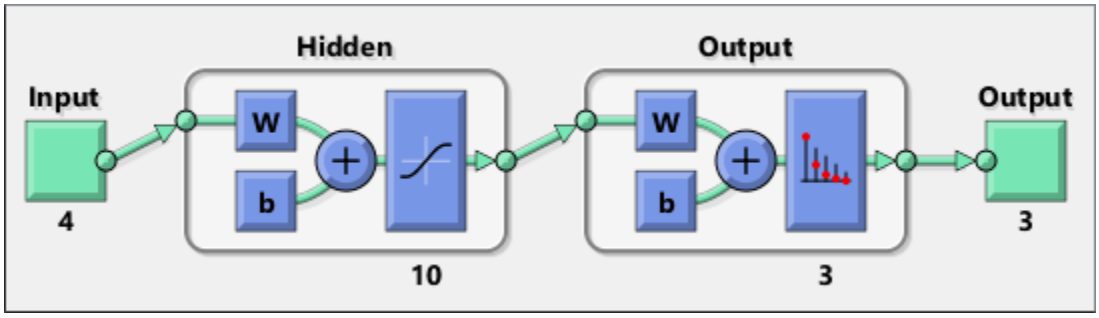
Tip To visualize deep learning networks, see Deep Network Designer.

Example

View Neural Network

This example shows how to view the diagram of a pattern recognition network.

```
[x,t] = iris_dataset;
net = patternnet;
net = configure(net,x,t);
view(net)
```



Introduced in R2008a

Neural Net Fitting

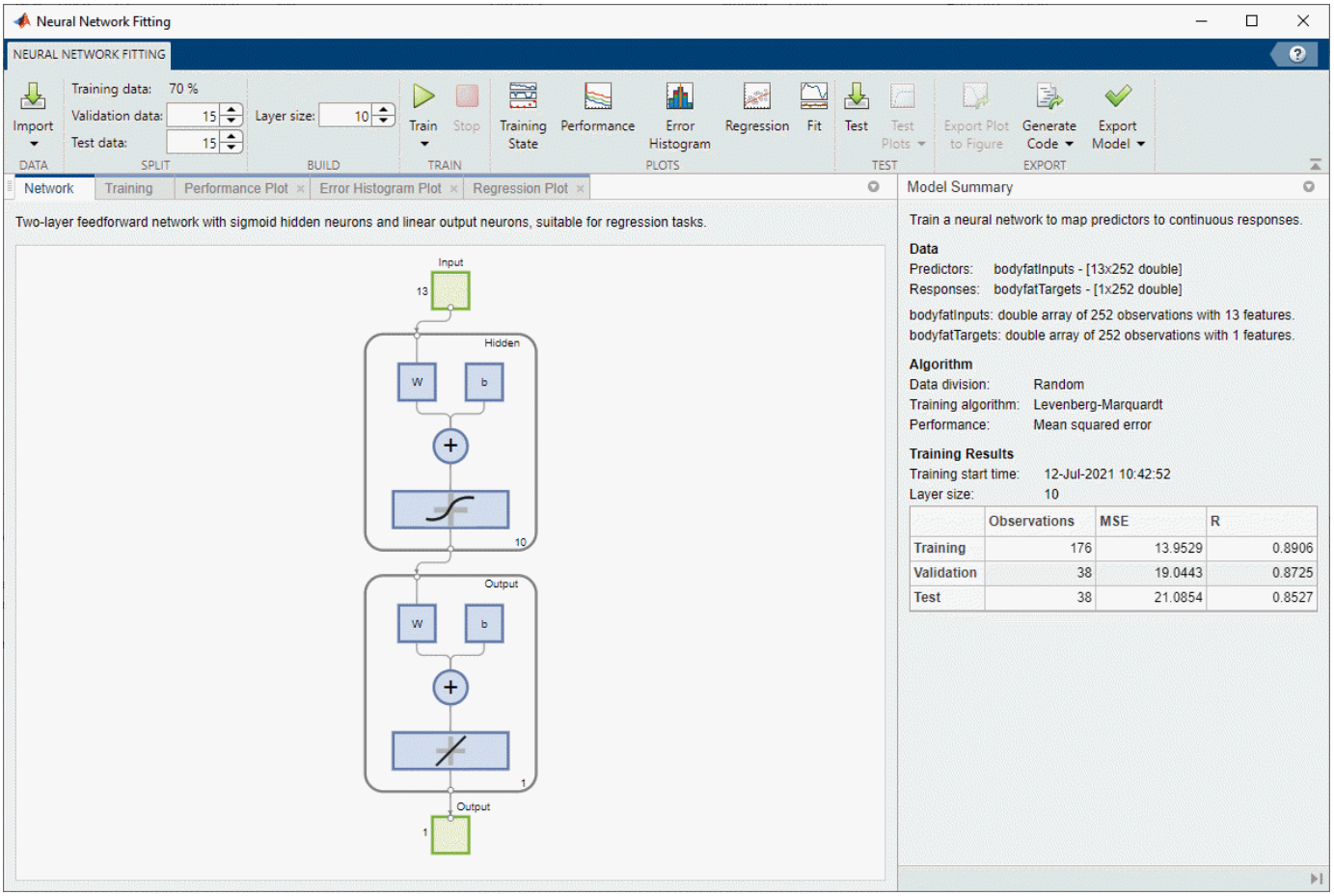
Solve fitting problem using two-layer feed-forward networks

Description

The **Neural Net Fitting** app lets you create, visualize, and train a two-layer feed-forward network to solve data fitting problems.

Using this app, you can:

- Import data from file, the MATLAB workspace, or use one of the example data sets.
- Split data into training, validation, and test sets.
- Define and train a neural network.
- Evaluate network performance using mean squared error and regression analysis.
- Analyze results using visualization plots, such as regression fit or histogram of errors.
- Generate MATLAB scripts to reproduce results and customize the training process.
- Generate functions suitable for deployment with MATLAB Compiler and MATLAB Coder tools, and export to Simulink for use with Simulink Coder.



Open the Neural Net Fitting App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `nftool`.

Examples

- “Fit Data with a Shallow Neural Network”

Algorithms

The **Neural Net Fitting** app provides built-in training algorithms that you can use to train your neural network.

Training Algorithm	Description
Levenberg-Marquardt	Update weight and bias values according to Levenberg-Marquardt optimization. Levenberg-Marquardt training is often the fastest training algorithm, although it does require more memory than other techniques. To implement this algorithm, the **Neural Net Fitting** app uses the `trainlm` function.
Bayesian regularization	Bayesian regularization updates the weight and bias values according to Levenberg-Marquardt optimization. It then minimizes a combination of squared errors and weights, and determines the correct combination so as to produce a network that generalizes well. This algorithm typically takes longer but is good at generalizing to noisy or small data sets. To implement this algorithm, the **Neural Net Fitting** app uses the `trainbr` function.
Scaled conjugate gradient backpropagation	Scaled conjugate gradient backpropagation updates weight and bias values according to the scaled conjugate gradient method. For large problems, scaled conjugate gradient is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations used by Levenberg-Marquardt or Bayesian regularization. To implement this algorithm, the **Neural Net Fitting** app uses the `trainscg` function.

See Also

Apps
Neural Net Time Series | **Neural Net Clustering** | **Neural Net Pattern Recognition**

Functions
`fitnet` | `feedforwardnet` | `trainlm` | `trainscg` | `trainbr`

Topics
“Fit Data with a Shallow Neural Network”

Neural Net Clustering

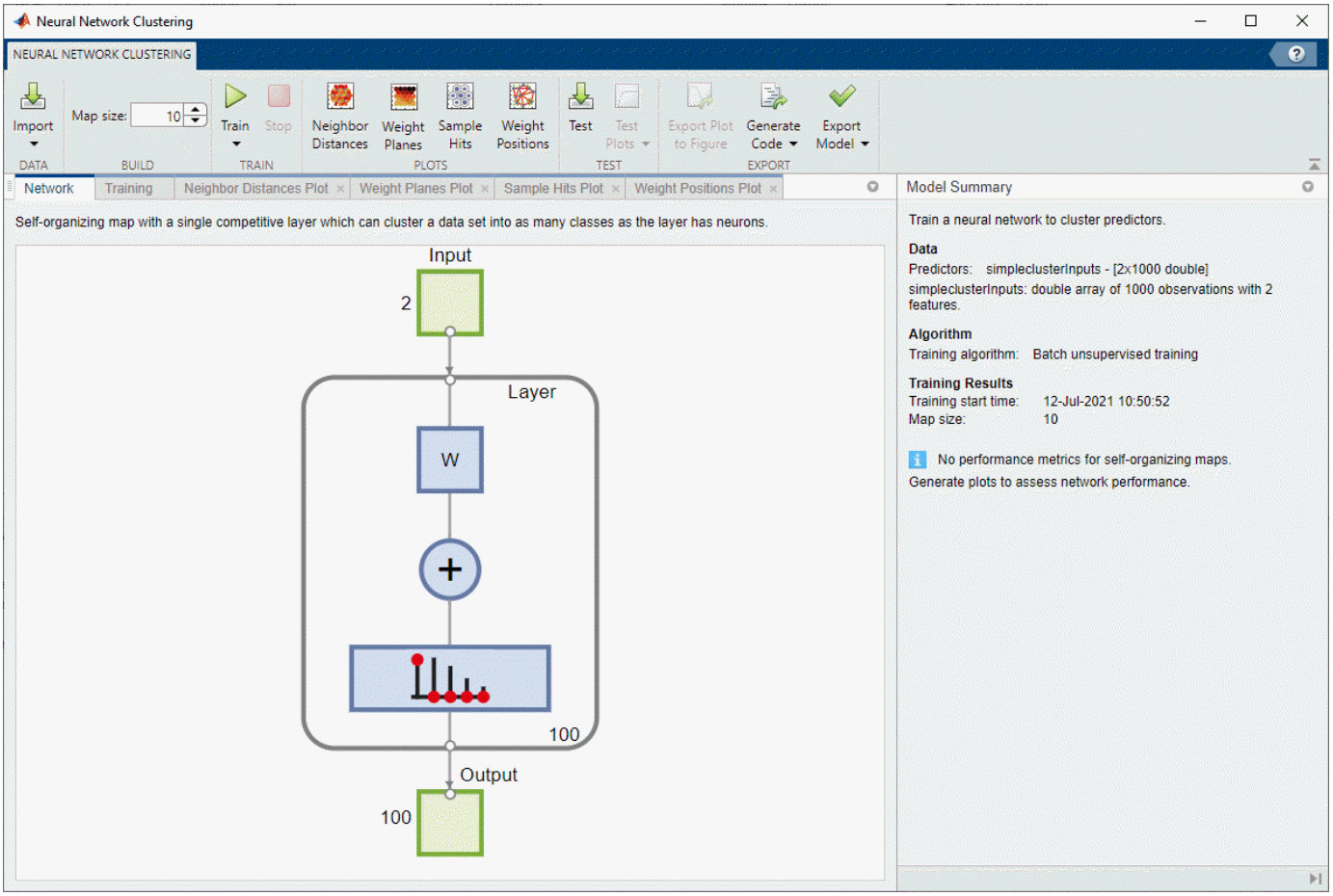
Solve clustering problem using self-organizing map (SOM) networks

Description

The **Neural Net Clustering** app lets you create, visualize, and train self-organizing map networks to solve clustering problems.

Using this app, you can:

- Import data from file, the MATLAB workspace, or use one of the example data sets.
- Define and train a neural network.
- Analyze results using visualization plots, such as neighbor distance, weight planes, sample hits, and weight position.
- Generate MATLAB scripts to reproduce results and customize the training process.
- Generate functions suitable for deployment with MATLAB Compiler and MATLAB Coder tools, and export to Simulink for use with Simulink Coder.



Open the Neural Net Clustering App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `nctool`.

Examples

- “Cluster Data with a Self-Organizing Map”

Algorithms

The **Neural Net Clustering** app provides a built-in training algorithm that you can use to train your neural network.

Training Algorithm	Description
Batch unsupervised weight and bias training	Train a network with unsupervised weight and bias learning rules with batch updates. The weights and biases are updated at the end of an entire pass through the input data. To implement this algorithm, the **Neural Net Clustering** app uses the `trainbu` function.

See Also

Apps
Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition**

Functions
`selforgmap` | `trainbu` | `learnsomb`

Topics
“Cluster Data with a Self-Organizing Map”

Neural Net Pattern Recognition

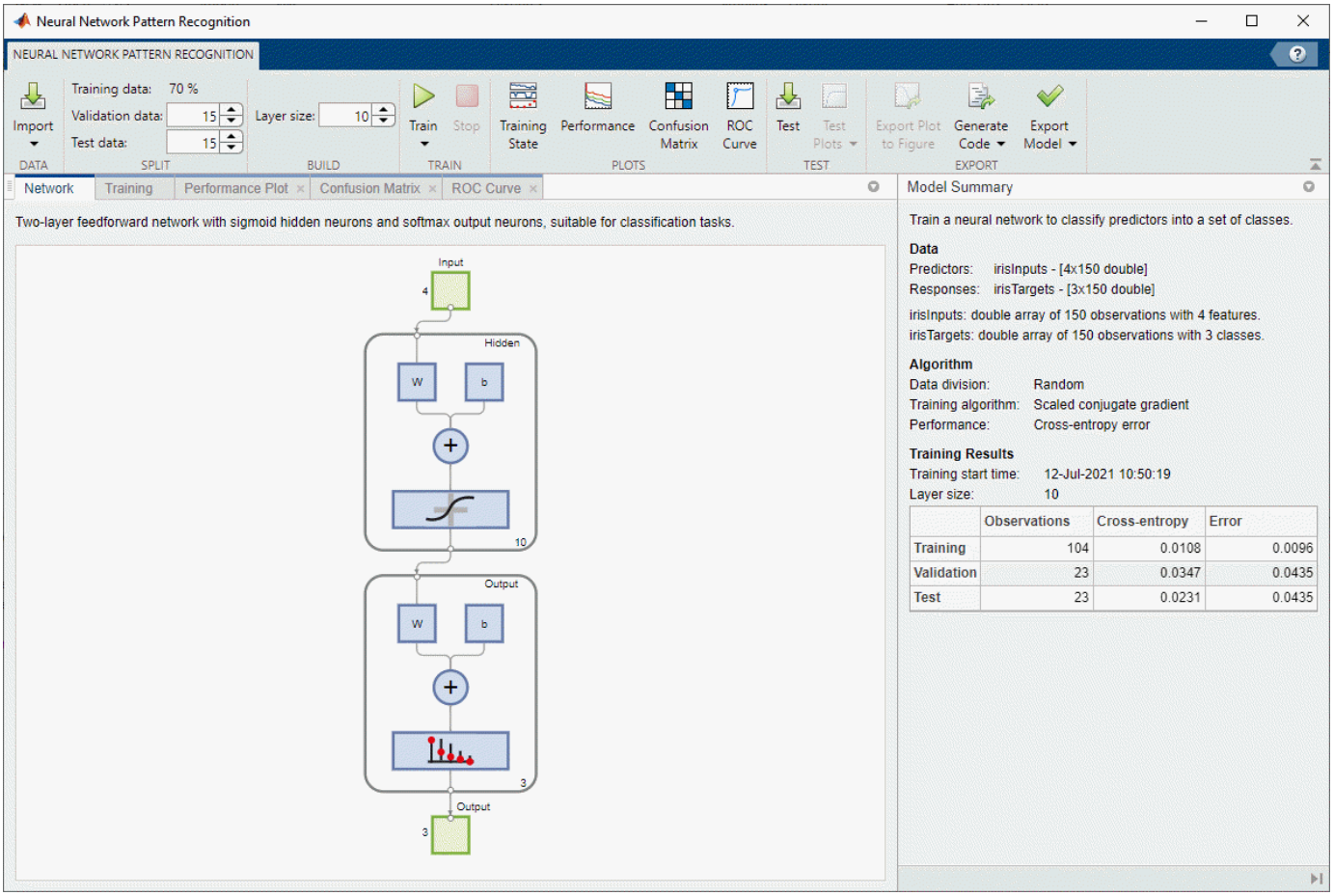
Solve pattern recognition problem using two-layer feed-forward networks

Description

The **Neural Net Pattern Recognition** app lets you create, visualize, and train two-layer feed-forward networks to solve data classification problems.

Using this app, you can:

- Import data from file, the MATLAB workspace, or use one of the example data sets.
- Split data into training, validation, and test sets.
- Define and train a neural network.
- Evaluate network performance using cross-entropy error and misclassification error.
- Analyze results using visualization plots, such as confusion matrices and receiver operating characteristic curves.
- Generate MATLAB scripts to reproduce results and customize the training process.
- Generate functions suitable for deployment with MATLAB Compiler and MATLAB Coder tools, and export to Simulink for use with Simulink Coder.



Open the Neural Net Pattern Recognition App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `nprtool`.

Examples

- “Classify Patterns with a Shallow Neural Network”

Algorithms

The **Neural Net Pattern Recognition** app provides a built-in training algorithm that you can use to train your neural network.

Training Algorithm	Description
Scaled conjugate gradient backpropagation	Scaled conjugate gradient backpropagation updates weight and bias values according to the scaled conjugate gradient method. To implement this algorithm, the **Neural Net Pattern Recognition** app uses the `trainscg` function.

See Also

Apps
Neural Net Fitting | **Neural Net Clustering** | **Neural Net Time Series**

Functions
`patternnet` | `trainscg`

Topics
“Classify Patterns with a Shallow Neural Network”

Neural Net Time Series

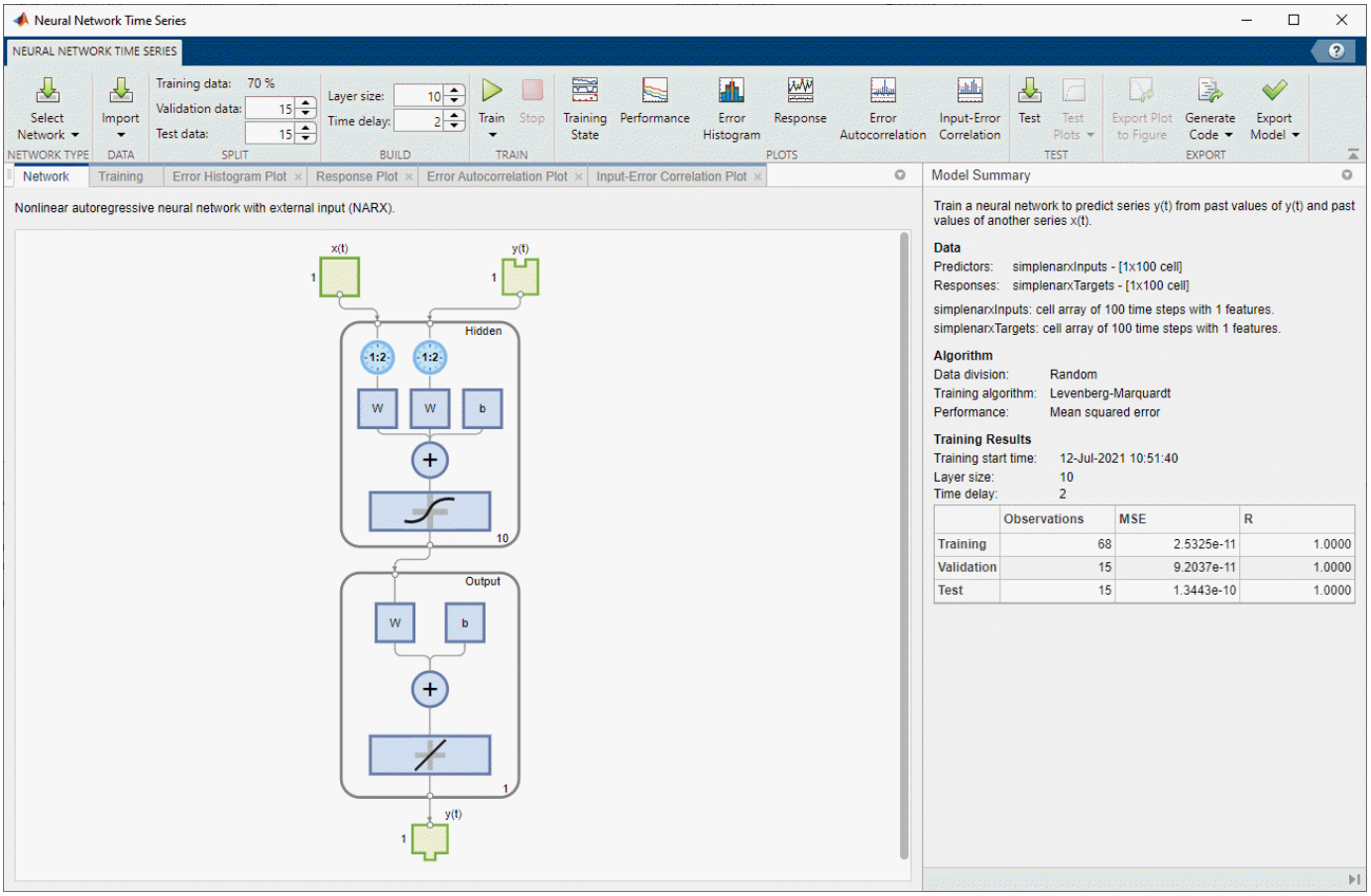
Solve nonlinear time series problem using dynamic neural networks

Description

The **Neural Net Time Series** app lets you create, visualize, and train dynamic neural networks to solve three different kinds of nonlinear time series problems.

Using this app, you can:

- Create three types of neural networks: NARX networks, NAR networks, and nonlinear input-output networks.
- Import data from file, the MATLAB workspace, or use one of the example data sets.
- Split data into training, validation, and test sets.
- Define and train a neural network.
- Evaluate network performance using mean squared error and regression analysis.
- Analyze results using visualization plots, such as autocorrelation plots or a histogram of errors.
- Generate MATLAB scripts to reproduce results and customize the training process.
- Generate functions suitable for deployment with MATLAB Compiler and MATLAB Coder tools, and export to Simulink for use with Simulink Coder.



Open the Neural Net Time Series App

- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `ntstool`.

Examples

- “Shallow Neural Network Time-Series Prediction and Modeling”

Algorithms

The **Neural Net Time Series** app provides built-in training algorithms that you can use to train your neural network.

Training Algorithm	Description
Levenberg-Marquardt	Update weight and bias values according to Levenberg-Marquardt optimization. Levenberg-Marquardt training is often the fastest training algorithm, although it does require more memory than other techniques. To implement this algorithm, the **Neural Net Time Series** app uses the `trainlm` function.
Bayesian regularization	Bayesian regularization updates the weight and bias values according to Levenberg-Marquardt optimization. It then minimizes a combination of squared errors and weights, and determines the correct combination so as to produce a network that generalizes well. This algorithm typically takes longer but is good at generalizing to noisy or small data sets. To implement this algorithm, the **Neural Net Time Series** app uses the `trainbr` function.
Scaled conjugate gradient backpropagation	Scaled conjugate gradient backpropagation updates weight and bias values according to the scaled conjugate gradient method. For large problems, scaled conjugate gradient is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations used by Levenberg-Marquardt or Bayesian regularization. To implement this algorithm, the **Neural Net Time Series** app uses the `trainscg` function.

See Also

Apps

Neural Net Fitting | **Neural Net Clustering** | **Neural Net Pattern Recognition**

Functions

`narxnet` | `narnet` | `closeloop` | `train` | `preparets` | `perform` | `removedelay`

Topics

“Shallow Neural Network Time-Series Prediction and Modeling”

matlab.io.datastore.MiniBatchable class

Package: `matlab.io.datastore`

Add mini-batch support to datastore

Description

`matlab.io.datastore.MiniBatchable` is an abstract mixin class that adds support for mini-batches to your custom datastore for use with Deep Learning Toolbox. A mini-batch datastore contains training and test data sets for use in Deep Learning Toolbox training, prediction, and classification.

To use this mixin class, you must inherit from the `matlab.io.datastore.MiniBatchable` class in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.MiniBatchable
    ...
end
```

To add support for mini-batches to your datastore:

- Inherit from an additional class `matlab.io.datastore.MiniBatchable`
- Define two additional properties: `MiniBatchSize` and `NumObservations`.

For more details and steps to create your custom mini-batch datastore to optimize performance during training, prediction, and classification, see “Develop Custom Mini-Batch Datastore”.

Properties

`MiniBatchSize` — Number of observations in each batch

positive integer

Number of observations that are returned in each batch, or call of the `read` function. For training, prediction, and classification, the `MiniBatchSize` property is set to the mini-batch size defined in `trainingOptions`.

Attributes:

`Abstract`	`true`
`Access`	`Public`

`NumObservations` — Total number of observations in the datastore

positive integer

Total number of observations contained within the datastore. This number of observations is the length of one training epoch.

Attributes:

Abstract	true
SetAccess	Protected
ReadAccess	Public

Attributes

Abstract	true
Sealed	false

For information on class attributes, see “Class Attributes”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

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];
```

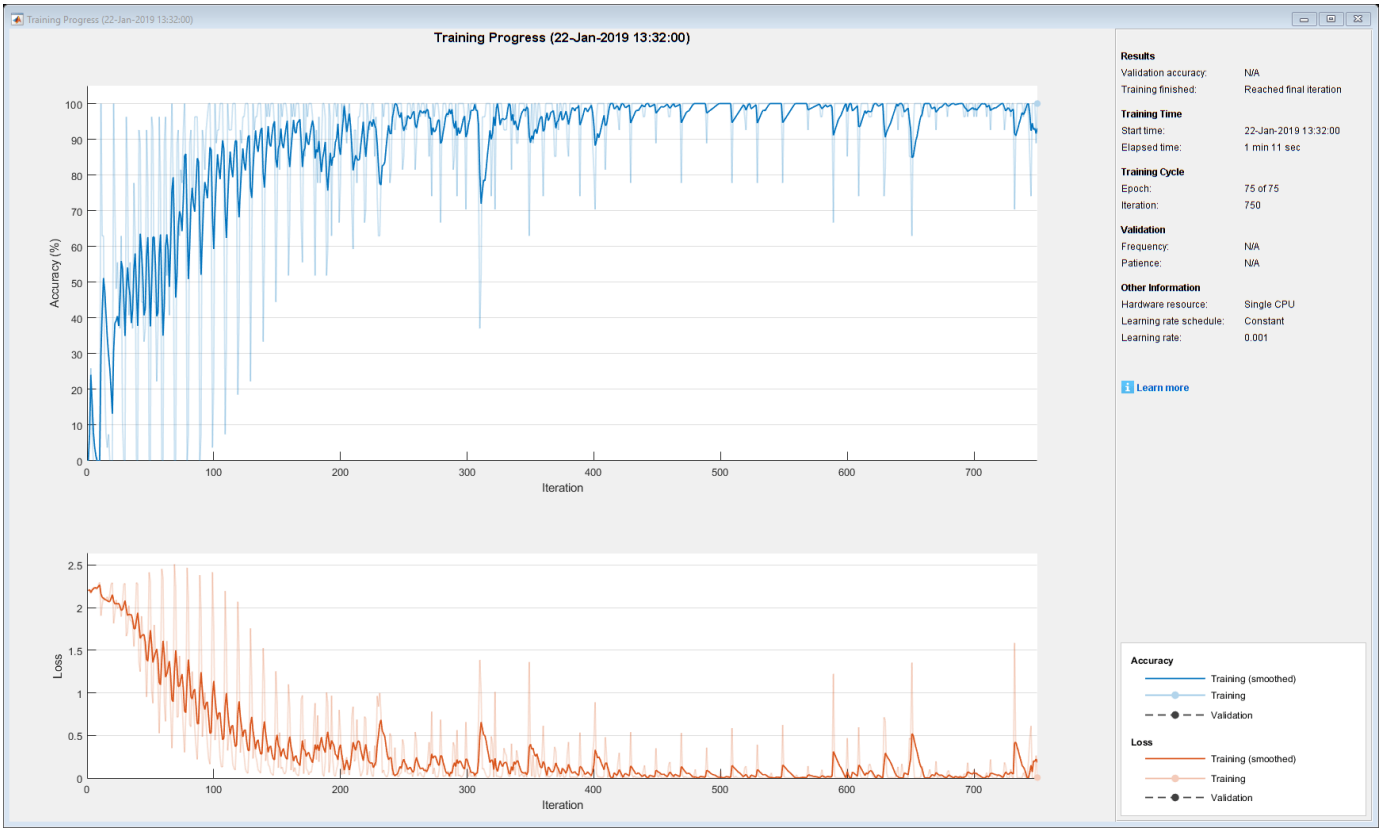
Specify the training options. Set the solver to `'adam'` and `'GradientThreshold'` to 2. Set the mini-batch size to 27 and set the maximum number of epochs to 75. The datastores do not support shuffling, so set `'Shuffle'` to `'never'`.

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
```

Compatibility Considerations

`matlab.io.datastore.MiniBatchable` is not recommended for custom image preprocessing

Not recommended starting in R2019a

Starting in R2019a, `matlab.io.datastore.MiniBatchable` is not recommended for custom image processing. Use the `transform` and `combine` functions with built-in datastores instead. For more information, see “Preprocess Images for Deep Learning”.

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

`matlab.io.Datastore` | `matlab.io.datastore.Partitionable` | `matlab.io.datastore.Shuffleable` | `read`

Topics
“Deep Learning in MATLAB”
“Develop Custom Mini-Batch Datastore”

Introduced in R2018a

read

Class: `matlab.io.datastore.MiniBatchable`
Package: `matlab.io.datastore`

Read data from mini-batch datastore

Note The `read` method of `matlab.io.datastore.MiniBatchable` is not recommended. For more information, see Compatibility Considerations.

Syntax

```
data = read(ds)
[data,info] = read(ds)
```

Description

`data = read(ds)` returns data from a mini-batch datastore. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(ds)` also returns information about the extracted data in `info`, including metadata.

Input Arguments

`mbds` — Mini-batch datastore

datastore | custom `MiniBatchable` datastore | ...

Mini-batch datastore, specified as a built-in datastore or custom mini-batch datastore. For more information, see “Datastores for Deep Learning”.

Output Arguments

`data` — Output data

table

Output data, returned as a table with `MiniBatchSize` number of rows. For the last mini-batch of data in the datastore, if `NumObservations` is not evenly divisible by `MiniBatchSize`, then `data` should contain the remaining observations in the datastore (a partial batch smaller than `MiniBatchSize`).

The table should have two columns, with predictors in the first column and responses in the second column.

`info` — Information about read data

structure array

Information about read data, returned as a structure array.

Attributes

`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

Compatibility Considerations

`read` is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, built-in datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

Note The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove the `read` method of `matlab.io.datastore.MiniBatchable` at this time.

See Also

`read (Datastore)` | `matlab.io.datastore.MiniBatchable` | `matlab.io.Datastore` | `transform` | `combine`

Topics

“Datastores for Deep Learning”
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

matlab.io.datastore.BackgroundDispatchable class

Package: `matlab.io.datastore`

(Not recommended) Add prefetch reading support to datastore

Note `matlab.io.datastore.BackgroundDispatchable` is not recommended. For more information, see Compatibility Considerations.

Description

`matlab.io.datastore.BackgroundDispatchable` is an abstract mixin class that adds support for prefetch reading to your custom datastore for use with Deep Learning Toolbox.

To use this mixin class, you must inherit from the `matlab.io.datastore.BackgroundDispatchable` class in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.BackgroundDispatchable
    ...
end
```

To add support for parallel processing to your custom datastore, you must:

- Inherit from an additional class `matlab.io.datastore.BackgroundDispatchable`
- Define the additional method: `readByIndex`

For more details and steps to create your custom datastore to optimize performance during training, prediction, and classification, see “Develop Custom Mini-Batch Datastore”.

Properties

`DispatchInBackground` — Dispatch observations in background

`true` (default) | `false`

Dispatch observations in the background during training, prediction, or classification, specified as `true` or `false`. To use background dispatching, you must have Parallel Computing Toolbox.

Attributes:

`Public`	`true`

Methods

readByIndex (Not recommended) Return observations from a datastore specified by index

Attributes

`Abstract`	`true`
`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Compatibility Considerations

`matlab.io.datastore.BackgroundDispatchable` is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

Note The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `matlab.io.datastore.BackgroundDispatchable` at this time.

See Also

`transform` | `combine` | `matlab.io.Datastore` | `matlab.io.datastore.Partitionable` | `matlab.io.datastore.Shuffleable`

Topics

“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

readByIndex

Class: `matlab.io.datastore.BackgroundDispatchable`
Package: `matlab.io.datastore`

(Not recommended) Return observations from a datastore specified by index

Note `readByIndex` is not recommended. For more information, see Compatibility Considerations.

Syntax

```
[data,info] = readByIndex(ds,ind)
```

Description

`[data,info] = readByIndex(ds,ind)` returns a subset of observations in a datastore, `ds`. The desired observations are specified by indices, `ind`.

Input Arguments

`ds` — Input datastore

`Datastore` object

Input datastore, specified as a `Datastore` object.

`ind` — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

`data` — Observations from datastore

table

Observations from the datastore, returned as a table or an array according to the `read` method of the datastore. For example, when `ds` is a custom mini-batch datastore, then `data` is a table with the same format as returned by the `read (MiniBatchable)` method.

`info` — Information about read data

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
`Filename`	`Filename` is a fully resolved path containing the path string, name of the file, and file extension.

Field Name	Description
`FileSize`	Total file size, in bytes. For MAT-files, `FileSize` is the total number of key-value pairs in the file.

Attributes

`Abstract`	`true`
`Access`	`Public`

To learn about attributes of methods, see Method Attributes.

Tips

- You must implement the `readByIndex` method by deriving a subclass from the `matlab.io.datastore.BackgroundDispatchable` class.

Compatibility Considerations

`readByIndex` is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.

- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

Note The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `matlab.io.datastore.BackgroundDispatchable` class or the `readByIndex` method at this time.

See Also

`transform` | `combine` | `matlab.io.Datastore` | `read` | `readall`

Topics

“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

matlab.io.datastore.PartitionableByIndex class

Package: `matlab.io.datastore`

(Not recommended) Add parallelization support to datastore

Note `matlab.io.datastore.PartitionableByIndex` is not recommended. For more information, see Compatibility Considerations.

Description

`matlab.io.datastore.PartitionableByIndex` is an abstract mixin class that adds parallelization support to your custom datastore for use with Deep Learning Toolbox. This class requires Parallel Computing Toolbox.

To use this mixin class, you must inherit from the `matlab.io.datastore.PartitionableByIndex` class in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.PartitionableByIndex
    ...
end
```

To add support for parallel processing to your custom datastore, you must:

- Inherit from an additional class `matlab.io.datastore.PartitionableByIndex`
- Define the additional method: `partitionByIndex`

For more details and steps to create your custom datastore with parallel processing support, see “Develop Custom Mini-Batch Datastore”.

Methods

partitionByIndex (Not recommended) Partition a datastore according to indices

Attributes

`Abstract`	`true`
`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Compatibility Considerations

`matlab.io.datastore.PartitionableByIndex` is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

Note The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `matlab.io.datastore.PartitionableByIndex` at this time.

See Also

`transform` | `combine` | `matlab.io.Datastore` | `matlab.io.datastore.Shuffleable` | `matlab.io.datastore.HadoopFileBased` | `matlab.io.datastore.Partitionable`

Topics
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

partitionByIndex

(Not recommended) Partition a datastore according to indices

Note `partitionByIndex` is not recommended. For more information, see Compatibility Considerations.

Syntax

```
ds2 = partitionByIndex(ds,ind)
```

Description

`ds2 = partitionByIndex(ds,ind)` partitions a subset of observations in a datastore, `ds`, into a new datastore, `ds2`. The desired observations are specified by indices, `ind`.

Input Arguments

`ds` — Input datastore

`Datastore` object

Input datastore, specified as a `Datastore` object.

`ind` — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

`ds2` — Partitioned datastore

`Datastore` object

Partitioned datastore, returned as a `Datastore` object.

Attributes

`Abstract`	`true`
`Access`	`Public`

To learn about attributes of methods, see Method Attributes.

Tips

- You must implement the `partitionByIndex` method by deriving a subclass from the `matlab.io.datastore.Partitionable` class.

Compatibility Considerations

`partitionByIndex` is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

Note The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `partitionByIndex` at this time.

See Also

`transform` | `combine` | `matlab.io.Datastore`

Topics
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

trainAutoencoder

Train an autoencoder

Syntax

```
autoenc = trainAutoencoder(X)
autoenc = trainAutoencoder(X,hiddenSize)
autoenc = trainAutoencoder( ___ ,Name,Value)
```

Description

`autoenc = trainAutoencoder(X)` returns an autoencoder, `autoenc`, trained using the training data in `X`.

`autoenc = trainAutoencoder(X,hiddenSize)` returns an autoencoder `autoenc`, with the hidden representation size of `hiddenSize`.

`autoenc = trainAutoencoder( ___ ,Name,Value)` returns an autoencoder `autoenc`, for any of the above input arguments with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the sparsity proportion or the maximum number of training iterations.

Examples

Train Sparse Autoencoder

Load the sample data.

```
X = abalone_dataset;
```

`X` is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with default settings.

```
autoenc = trainAutoencoder(X);
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```
mseError = mse(X-XReconstructed)
```

```
mseError = 0.0167
```

Train Autoencoder with Specified Options

Load the sample data.

```
X = abalone_dataset;
```

X is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with hidden size 4, 400 maximum epochs, and linear transfer function for the decoder.

```
autoenc = trainAutoencoder(X,4,'MaxEpochs',400,...
'DecoderTransferFunction','purelin');
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```
mseError = mse(X-XReconstructed)
```

```
mseError = 0.0050
```

Reconstruct Observations Using Sparse Autoencoder

Generate the training data.

```
rng(0,'twister'); % For reproducibility
n = 1000;
r = linspace(-10,10,n)';
x = 1 + r*5e-2 + sin(r)./r + 0.2*randn(n,1);
```

Train autoencoder using the training data.

```
hiddenSize = 25;
autoenc = trainAutoencoder(x',hiddenSize,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.10);
```

Generate the test data.

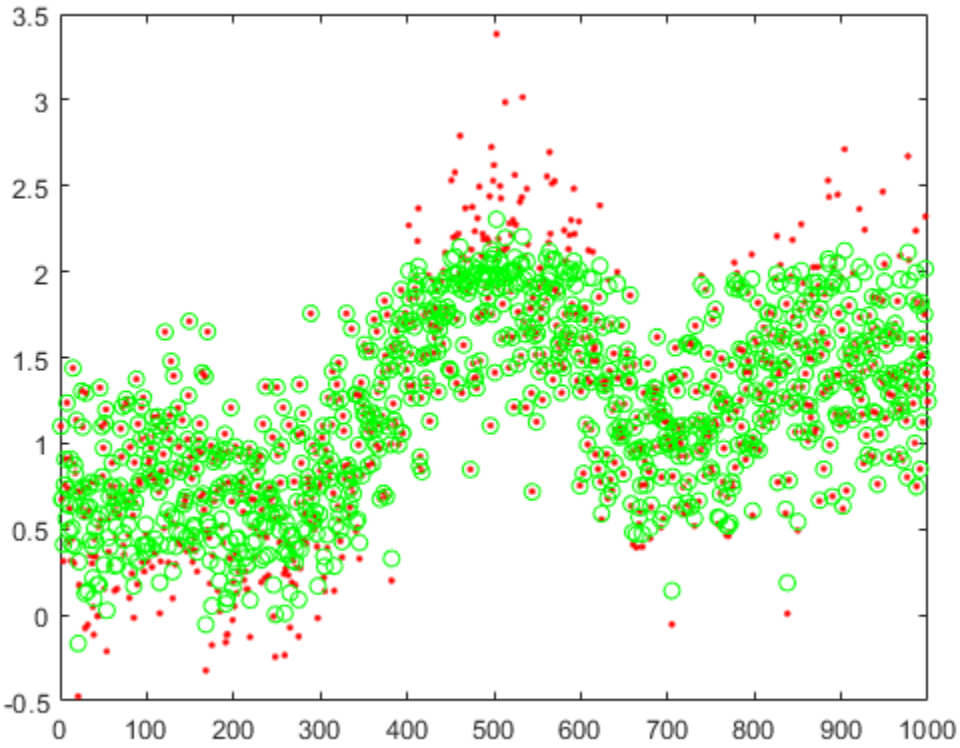
```
n = 1000;
r = sort(-10 + 20*rand(n,1));
xtest = 1 + r*5e-2 + sin(r)./r + 0.4*randn(n,1);
```

Predict the test data using the trained autoencoder, `autoenc` .

```
xReconstructed = predict(autoenc,xtest');
```

Plot the actual test data and the predictions.

```
figure;
plot(xtest,'r.');
hold on
plot(xReconstructed,'go');
```



Reconstruct Handwritten Digit Images Using Sparse Autoencoder

Load the training data.

```
XTrain = digitTrainCellArrayData;
```

The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.

```
hiddenSize = 25;
autoenc = trainAutoencoder(XTrain,hiddenSize,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.15);
```

Load the test data.

```
XTest = digitTestCellArrayData;
```

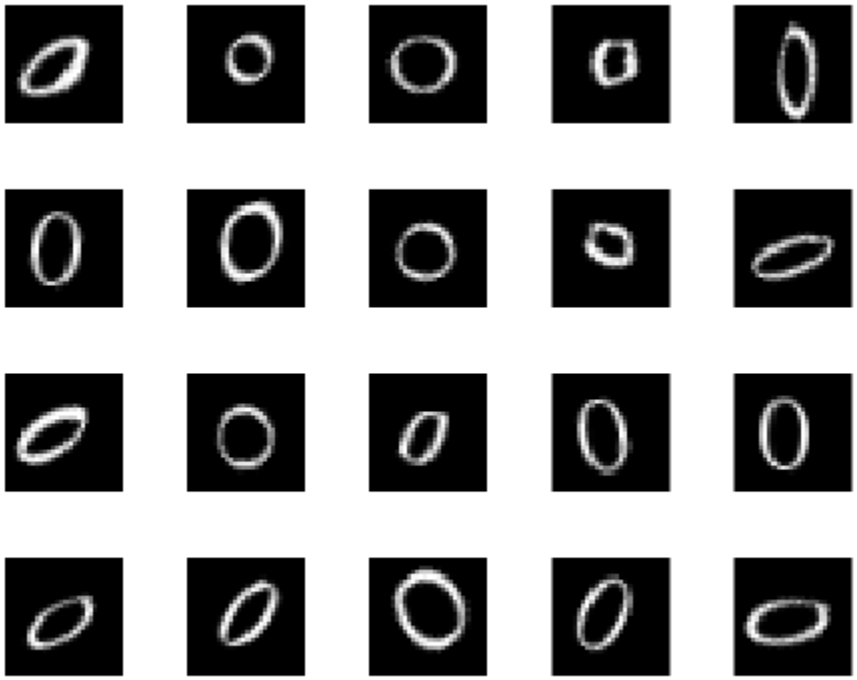
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,XTest);
```

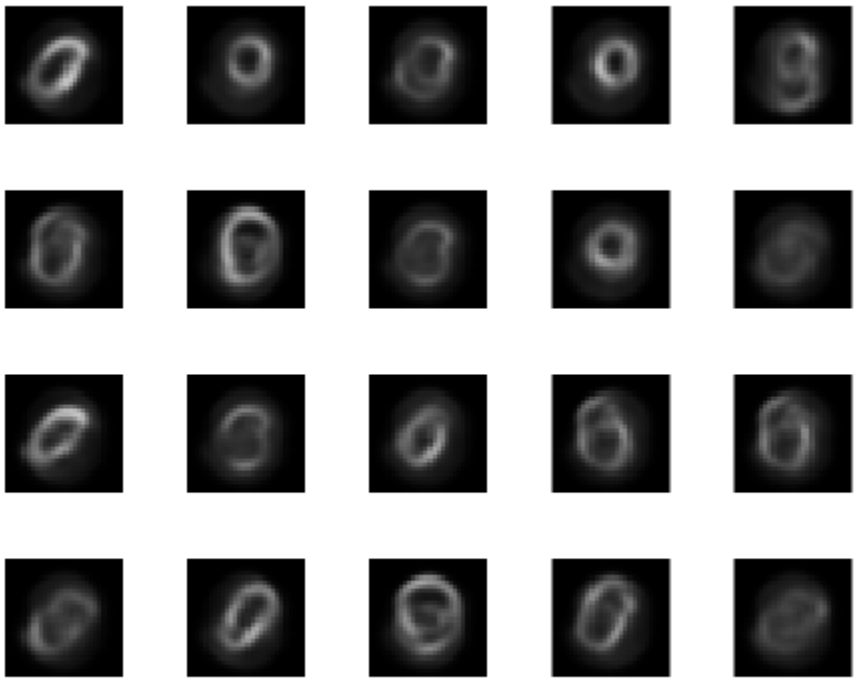
View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(XTest{i});
end
```



View the reconstructed test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(xReconstructed{i});
end
```



Input Arguments

X — Training data

matrix | cell array of image data

Training data, specified as a matrix of training samples or a cell array of image data. If X is a matrix, then each column contains a single sample. If X is a cell array of image data, then the data in each cell must have the same number of dimensions. The image data can be pixel intensity data for gray images, in which case, each cell contains an *m*-by-*n* matrix. Alternatively, the image data can be RGB data, in which case, each cell contains an *m*-by-*n*-3 matrix.

Data Types: `single` | `double` | `cell`

`hiddenSize` — Size of hidden representation of the autoencoder

10 (default) | positive integer value

Size of hidden representation of the autoencoder, specified as a positive integer value. This number is the number of neurons in the hidden layer.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'EncoderTransferFunction','satlin','L2WeightRegularization',0.05` specifies the transfer function for the encoder as the positive saturating linear transfer function and the L2 weight regularization as 0.05.

`EncoderTransferFunction` — Transfer function for the encoder
`'logsig'` (default) | `'satlin'`

Transfer function for the encoder, specified as the comma-separated pair consisting of `'EncoderTransferFunction'` and one of the following.

Transfer Function Option	Definition
`'logsig'`	Logistic sigmoid function $$f(z) = \frac{1}{1+e^{-z}}$$
`'satlin'`	Positive saturating linear transfer function $$f(z) = \begin{cases} 0, & \text{if } z \le 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \ge 1 \end{cases}$$

Example: `'EncoderTransferFunction','satlin'`

`DecoderTransferFunction` — Transfer function for the decoder
`'logsig'` (default) | `'satlin'` | `'purelin'`

Transfer function for the decoder, specified as the comma-separated pair consisting of `'DecoderTransferFunction'` and one of the following.

Transfer Function Option	Definition
`'logsig'`	Logistic sigmoid function $$f(z) = \frac{1}{1+e^{-z}}$$
`'satlin'`	Positive saturating linear transfer function $$f(z) = \begin{cases} 0, & \text{if } z \le 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \ge 1 \end{cases}$$
`'purelin'`	Linear transfer function $$f(z) = z$$

Example: `'DecoderTransferFunction','purelin'`

`MaxEpochs` — Maximum number of training epochs
1000 (default) | positive integer value

Maximum number of training epochs or iterations, specified as the comma-separated pair consisting of `'MaxEpochs'` and a positive integer value.

Example: `'MaxEpochs',1200`

L2WeightRegularization — The coefficient for the L_2 weight regularizer
0.001 (default) | a positive scalar value

The coefficient for the L_2 weight regularizer on page 2-522 in the cost function (`LossFunction`), specified as the comma-separated pair consisting of `'L2WeightRegularization'` and a positive scalar value.

Example: `'L2WeightRegularization',0.05`

LossFunction — Loss function to use for training
`'msesparse'` (default)

Loss function to use for training, specified as the comma-separated pair consisting of `'LossFunction'` and `'msesparse'`. It corresponds to the mean squared error function adjusted for training a sparse autoencoder on page 2-522 as follows:

$$E = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn}-\hat{x}_{kn}\right)^2 + \lambda * \underbrace{\Omega_{weights}}_{L_2\text{ regularization}} + \beta * \underbrace{\Omega_{sparsity}}_{\text{sparsity regularization}},$$

where λ is the coefficient for the L_2 regularization term on page 2-522 and β is the coefficient for the sparsity regularization term on page 2-522. You can specify the values of λ and β by using the `L2WeightRegularization` and `SparsityRegularization` name-value pair arguments, respectively, while training an autoencoder.

ShowProgressWindow — Indicator to show the training window
`true` (default) | `false`

Indicator to show the training window, specified as the comma-separated pair consisting of `'ShowProgressWindow'` and either `true` or `false`.

Example: `'ShowProgressWindow',false`

SparsityProportion — Desired proportion of training examples a neuron reacts to
0.05 (default) | positive scalar value in the range from 0 to 1

Desired proportion of training examples a neuron reacts to, specified as the comma-separated pair consisting of `'SparsityProportion'` and a positive scalar value. Sparsity proportion 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. Hence, a low sparsity proportion encourages higher degree of sparsity. See Sparse Autoencoders on page 2-522.

Example: `'SparsityProportion',0.01` is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples.

SparsityRegularization — Coefficient that controls the impact of the sparsity regularizer
1 (default) | a positive scalar value

Coefficient that controls the impact of the sparsity regularizer on page 2-522 in the cost function, specified as the comma-separated pair consisting of `'SparsityRegularization'` and a positive scalar value.

Example: `'SparsityRegularization',1.6`

`TrainingAlgorithm` — The algorithm to use for training the autoencoder
`'trainscg'` (default)

The algorithm to use for training the autoencoder, specified as the comma-separated pair consisting of `'TrainingAlgorithm'` and `'trainscg'`. It stands for scaled conjugate gradient descent [1].

`ScaleData` — Indicator to rescale the input data
`true` (default) | `false`

Indicator to rescale the input data, specified as the comma-separated pair consisting of `'ScaleData'` and either `true` or `false`.

Autoencoders attempt to replicate their input at their output. For it to be possible, the range of the input data must match the range of the transfer function for the decoder. `trainAutoencoder` automatically scales the training data to this range when training an autoencoder. If the data was scaled while training an autoencoder, the `predict`, `encode`, and `decode` methods also scale the data.

Example: `'ScaleData',false`

`UseGPU` — Indicator to use GPU for training
`false` (default) | `true`

Indicator to use GPU for training, specified as the comma-separated pair consisting of `'UseGPU'` and either `true` or `false`.

Example: `'UseGPU',true`

Output Arguments

`autoenc` — Trained autoencoder
`Autoencoder` object

Trained autoencoder, returned as an `Autoencoder` object. For information on the properties and methods of this object, see `Autoencoder` class page.

More About

Autoencoders

An autoencoder is a neural network which is trained to replicate its input at its output. Autoencoders can be used as tools to learn deep neural networks. Training an autoencoder is unsupervised in the sense that no labeled data is needed. The training process is still based on the optimization of a cost function. The cost function measures the error between the input x and its reconstruction at the output $\widehat{x}$.

An autoencoder is composed of an encoder and a decoder. The encoder and decoder can have multiple layers, but for simplicity consider that each of them has only one layer.

If the input to an autoencoder is a vector $x \in \mathbb{R}^{D_x}$, then the encoder maps the vector x to another vector $z \in \mathbb{R}^{D^{(1)}}$ as follows:

$$z = h^{(1)}\left(W^{(1)}x + b^{(1)}\right),$$

where the superscript (1) indicates the first layer. $h^{(1)}:\mathbb{R}^{D^{(1)}}\rightarrow\mathbb{R}^{D^{(1)}}$ is a transfer function for the encoder, $W^{(1)}\in\mathbb{R}^{D^{(1)}\times D_X}$ is a weight matrix, and $b^{(1)}\in\mathbb{R}^{D^{(1)}}$ is a bias vector. Then, the decoder maps the encoded representation z back into an estimate of the original input vector, x, as follows:

$$\widehat{x}=h^{(2)}\left(W^{(2)}z+b^{(2)}\right),$$

where the superscript (2) represents the second layer. $h^{(2)}:\mathbb{R}^{D_X}\rightarrow\mathbb{R}^{D_X}$ is the transfer function for the decoder,$W^{(1)}\in\mathbb{R}^{D_X\times D^{(1)}}$ is a weight matrix, and $b^{(2)}\in\mathbb{R}^{D_X}$ is a bias vector.

Sparse Autoencoders

Encouraging sparsity of an autoencoder is possible by adding a regularizer to the cost function [2]. This regularizer is a function of the average output activation value of a neuron. The average output activation measure of a neuron i is defined as:

$$\widehat{\rho}_i=\frac{1}{n}\sum_{j=1}^{n}z_i^{(1)}(x_j)=\frac{1}{n}\sum_{j=1}^{n}h\left(w_i^{(1)T}x_j+b_i^{(1)}\right),$$

where n is the total number of training examples. x_j is the jth training example, $w_i^{(1)T}$ is the ith row of the weight matrix $W^{(1)}$, and $b_i^{(1)}$ is the ith entry of the bias vector, $b^{(1)}$. A neuron is considered to be 'firing', if its output activation value is high. A low output activation value means that the neuron in the hidden layer fires in response to a small number of the training examples. Adding a term to the cost function that constrains the values of $\widehat{\rho}_i$ to be low encourages the autoencoder to learn a representation, where each neuron in the hidden layer fires to a small number of training examples. That is, each neuron specializes by responding to some feature that is only present in a small subset of the training examples.

Sparsity Regularization

Sparsity regularizer attempts to enforce a constraint on the sparsity of the output from the hidden layer. Sparsity can be encouraged by adding a regularization term that takes a large value when the average activation value, $\widehat{\rho}_i$, of a neuron i and its desired value, ρ, are not close in value [2]. One such sparsity regularization term can be the Kullback-Leibler divergence.

$$\Omega_{sparsity}=\sum_{i=1}^{D^{(1)}}KL(\rho\parallel\widehat{\rho}_i)=\sum_{i=1}^{D^{(1)}}\rho\log\left(\frac{\rho}{\widehat{\rho}_i}\right)+(1-\rho)\log\left(\frac{1-\rho}{1-\widehat{\rho}_i}\right)$$

Kullback-Leibler divergence is a function for measuring how different two distributions are. In this case, it takes the value zero when ρ and $\widehat{\rho}_i$ are equal to each other, and becomes larger as they diverge from each other. Minimizing the cost function forces this term to be small, hence ρ and $\widehat{\rho}_i$ to be close to each other. You can define the desired value of the average activation value using the `SparsityProportion` name-value pair argument while training an autoencoder.

L_2 Regularization

When training a sparse autoencoder, it is possible to make the sparsity regulariser small by increasing the values of the weights $w^{(l)}$ and decreasing the values of $z^{(1)}$ [2]. Adding a regularization term on the weights to the cost function prevents it from happening. This term is called the L_2 regularization term and is defined by:

$$\Omega_{weights} = \frac{1}{2}\sum_{l}^{L}\sum_{j}^{n}\sum_{i}^{k}\left(w_{ji}^{(l)}\right)^2,$$

where L is the number of hidden layers, n is the number of observations (examples), and k is the number of variables in the training data.

Cost Function

The cost function for training a sparse autoencoder on page 2-522 is an adjusted mean squared error function as follows:

$$E = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn} - \widehat{x}_{kn}\right)^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{L_2 \text{ regularization}} + \beta * \underbrace{\Omega_{sparsity}}_{\text{sparsity regularization}},$$

where λ is the coefficient for the L_2 regularization term on page 2-522 and β is the coefficient for the sparsity regularization term on page 2-522. You can specify the values of λ and β by using the `L2WeightRegularization` and `SparsityRegularization` name-value pair arguments, respectively, while training an autoencoder.

References

[1] Moller, M. F. “A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning”, *Neural Networks*, Vol. 6, 1993, pp. 525–533.

[2] Olshausen, B. A. and D. J. Field. “Sparse Coding with an Overcomplete Basis Set: A Strategy Employed by V1.” *Vision Research*, Vol.37, 1997, pp.3311–3325.

See Also

`trainSoftmaxLayer` | `Autoencoder` | `encode` | `stack`

Topics

“Train Stacked Autoencoders for Image Classification”

Introduced in R2015b

trainSoftmaxLayer

Train a softmax layer for classification

Syntax

```
net = trainSoftmaxLayer(X,T)
net = trainSoftmaxLayer(X,T,Name,Value)
```

Description

`net = trainSoftmaxLayer(X,T)` trains a softmax layer, `net`, on the input data `X` and the targets `T`.

`net = trainSoftmaxLayer(X,T,Name,Value)` trains a softmax layer, `net`, with additional options specified by one or more of the `Name,Value` pair arguments.

For example, you can specify the loss function.

Examples

Classify Using Softmax Layer

Load the sample data.

```
[X,T] = iris_dataset;
```

`X` is a 4x150 matrix of four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

`T` is a 3x150 matrix of associated class vectors defining which of the three classes each input is assigned to. Each row corresponds to a dummy variable representing one of the iris species (classes). In each column, a 1 in one of the three rows represents the class that particular sample (observation or example) belongs to. There is a zero in the rows for the other classes that the observation does not belong to.

Train a softmax layer using the sample data.

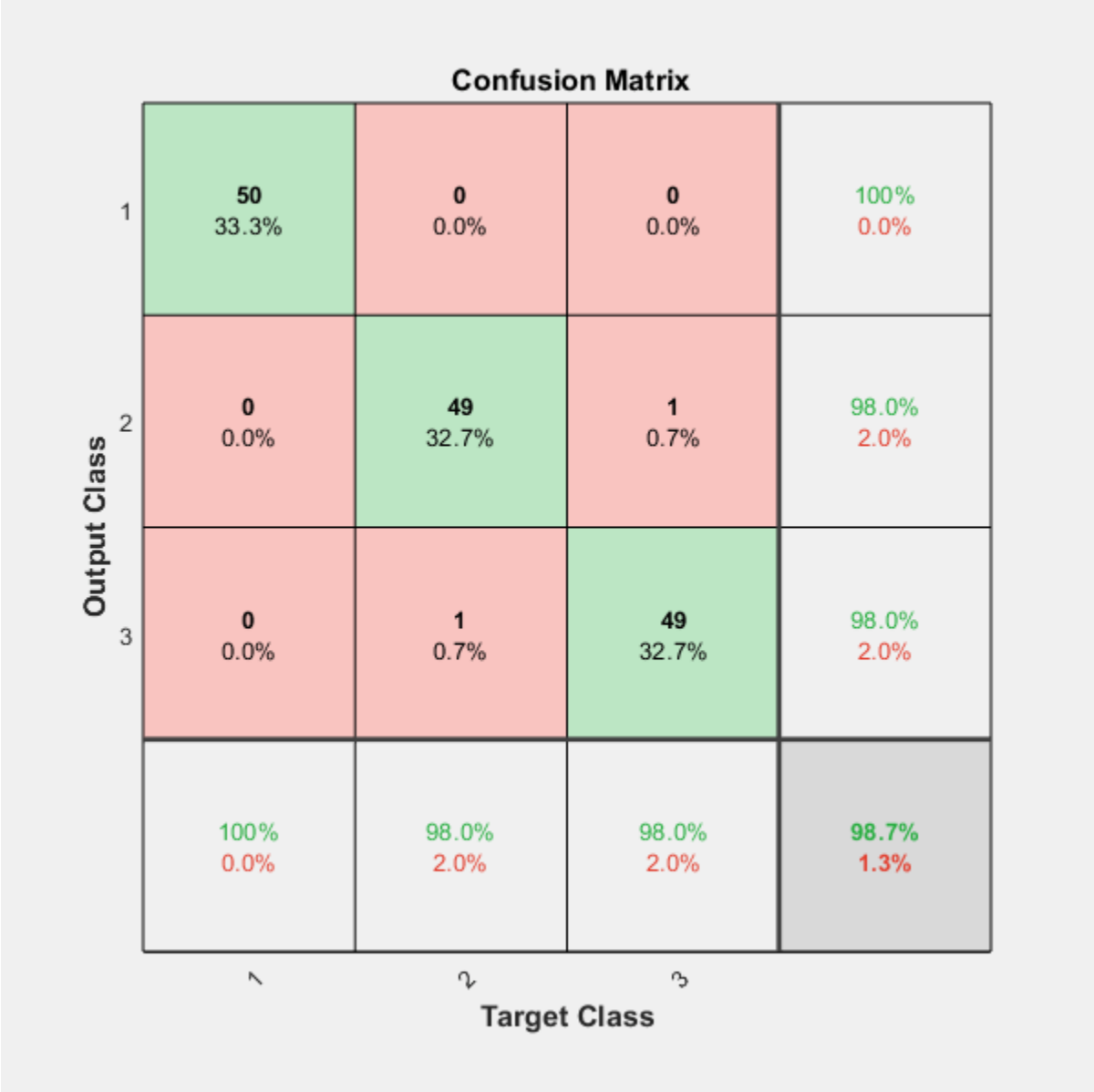
```
net = trainSoftmaxLayer(X,T);
```

Classify the observations into one of the three classes using the trained softmax layer.

```
Y = net(X);
```

Plot the confusion matrix using the targets and the classifications obtained from the softmax layer.

```
plotconfusion(T,Y);
```



Input Arguments

X — Training data

m-by-*n* matrix

Training data, specified as an *m*-by-*n* matrix, where *m* is the number of variables in training data, and *n* is the number of observations (examples). Hence, each column of X represents a sample.

Data Types: `single` | `double`

T — Target data

k-by-*n* matrix

Target data, specified as a *k*-by-*n* matrix, where *k* is the number of classes, and *n* is the number of observations. Each row is a dummy variable representing a particular class. In other words, each column represents a sample, and all entries of a column are zero except for a single one in a row. This single entry indicates the class for that sample.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaxEpochs',400,'ShowProgressWindow',false` specifies the maximum number of iterations as 400 and hides the training window.

`MaxEpochs` — Maximum number of training iterations

1000 (default) | positive integer value

Maximum number of training iterations, specified as the comma-separated pair consisting of `'MaxEpochs'` and a positive integer value.

Example: `'MaxEpochs',500`

Data Types: `single` | `double`

`LossFunction` — Loss function for the softmax layer

`'crossentropy'` (default) | `'mse'`

Loss function for the softmax layer, specified as the comma-separated pair consisting of `'LossFunction'` and either `'crossentropy'` or `'mse'`.

`mse` stands for mean squared error function, which is given by:

$$E = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k}\left(t_{ij} - y_{ij}\right)^2,$$

where *n* is the number of training examples, and *k* is the number of classes. t_{ij} is the *ij*th entry of the target matrix, `T`, and y_{ij} is the *i*th output from the autoencoder when the input vector is $\boldsymbol{x_j}$.

The cross entropy function is given by:

$$E = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k} t_{ij}\ln y_{ij} + \left(1 - t_{ij}\right)\ln\left(1 - y_{ij}\right).$$

Example: `'LossFunction','mse'`

`ShowProgressWindow` — Indicator to display the training window

`true` (default) | `false`

Indicator to display the training window during training, specified as the comma-separated pair consisting of `'ShowProgressWindow'` and either `true` or `false`.

Example: `'ShowProgressWindow',false`

Data Types: `logical`

`TrainingAlgorithm` — Training algorithm
`'trainscg'` (default)

Training algorithm used to train the softmax layer, specified as the comma-separated pair consisting of `'TrainingAlgorithm'` and `'trainscg'`, which stands for scaled conjugate gradient.

Example: `'TrainingAlgorithm','trainscg'`

Output Arguments

`net` — Softmax layer for classification
`network` object

Softmax layer for classification, returned as a `network` object. The softmax layer, `net`, is the same size as the target `T`.

See Also

`trainAutoencoder` | `stack`

Introduced in R2015b

Autoencoder class

Autoencoder class

Description

An `Autoencoder` object contains an autoencoder network, which consists of an encoder and a decoder. The encoder maps the input to a hidden representation. The decoder attempts to map this representation back to the original input.

Construction

`autoenc = trainAutoencoder(X)` returns an autoencoder trained using the training data in `X`.

`autoenc = trainAutoencoder(X,hiddenSize)` returns an autoencoder with the hidden representation size of `hiddenSize`.

`autoenc = trainAutoencoder(___,Name,Value)` returns an autoencoder for any of the above input arguments with additional options specified by one or more name-value pair arguments.

Input Arguments

`X` — Training data
matrix | cell array of image data

Training data, specified as a matrix of training samples or a cell array of image data. If `X` is a matrix, then each column contains a single sample. If `X` is a cell array of image data, then the data in each cell must have the same number of dimensions. The image data can be pixel intensity data for gray images, in which case, each cell contains an *m*-by-*n* matrix. Alternatively, the image data can be RGB data, in which case, each cell contains an *m*-by-*n*-3 matrix.

Data Types: `single` | `double` | `cell`

`hiddenSize` — Size of hidden representation of the autoencoder
10 (default) | positive integer value

Size of hidden representation of the autoencoder, specified as a positive integer value. This number is the number of neurons in the hidden layer.

Data Types: `single` | `double`

Properties

`HiddenSize` — Size of the hidden representation
a positive integer value

Size of the hidden representation in the hidden layer of the autoencoder, stored as a positive integer value.

Data Types: `double`

`EncoderTransferFunction` — Name of the transfer function for the encoder
string

Name of the transfer function for the encoder, stored as a string.

Data Types: `char`

`EncoderWeights` — Weights for the encoder
matrix

Weights for the encoder, stored as a matrix.

Data Types: `double`

`EncoderBiases` — Bias values for the encoder
vector

Bias values for the encoder, stored as a vector.

Data Types: `double`

`DecoderTransferFunction` — Name of the transfer function for the decoder
string

Name of the transfer function for the decoder, stored as a string.

Data Types: `char`

`DecoderWeights` — Weights for the decoder
matrix

Weights for the decoder, stored as a matrix.

Data Types: `double`

`DecoderBiases` — Bias values for the decoder
vector

Bias values for the decoder, stored as a vector.

Data Types: `double`

`TrainingParameters` — Parameters that `trainAutoencoder` uses for training the autoencoder
structure

Parameters that `trainAutoencoder` uses for training the autoencoder, stored as a structure.

Data Types: `struct`

`ScaleData` — Indicator for data that is rescaled
`true` or 1 (default) | `false` or 0

Indicator for data that is rescaled while passing to the autoencoder, stored as either `true` or `false`.

Autoencoders attempt to replicate their input at their output. For it to be possible, the range of the input data must match the range of the transfer function for the decoder. `trainAutoencoder` automatically scales the training data to this range when training an autoencoder. If the data was

scaled while training an autoencoder, the `predict`, `encode`, and `decode` methods also scale the data.

Data Types: `logical`

Methods

decode	Decode encoded data
encode	Encode input data
generateFunction	Generate a MATLAB function to run the autoencoder
generateSimulink	Generate a Simulink model for the autoencoder
network	Convert `Autoencoder` object into `network` object
plotWeights	Plot a visualization of the weights for the encoder of an autoencoder
predict	Reconstruct the inputs using trained autoencoder
stack	Stack encoders from several autoencoders together
view	View autoencoder

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`trainAutoencoder`

Topics

Class Attributes
Property Attributes

Introduced in R2015b

decode

Class: `Autoencoder`

Decode encoded data

Syntax

```
Y = decode(autoenc,Z)
```

Description

`Y = decode(autoenc,Z)` returns the decoded data on page 2-532 `Y`, using the autoencoder `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned by the `trainAutoencoder` function as an object of the `Autoencoder` class.

`Z` — Data encoded by `autoenc`

matrix

Data encoded by `autoenc`, specified as a matrix. Each column of `Z` represents an encoded sample (observation).

Data Types: `single` | `double`

Output Arguments

`Y` — Decoded data

matrix | cell array of image data

Decoded data, returned as a matrix or a cell array of image data.

If the autoencoder `autoenc` was trained on a cell array of image data, then `Y` is also a cell array of images.

If the autoencoder `autoenc` was trained on a matrix, then `Y` is also a matrix, where each column of `Y` corresponds to one sample or observation.

Examples

Decode Encoded Data For New Images

Load the training data.

```
X = digitTrainCellArrayData;
```

X is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder using the training data with a hidden size of 15.

```
hiddenSize = 15;
autoenc = trainAutoencoder(X,hiddenSize);
```

Extract the encoded data for new images using the autoencoder.

```
Xnew = digitTestCellArrayData;
features = encode(autoenc,Xnew);
```

Decode the encoded data from the autoencoder.

```
Y = decode(autoenc,features);
```

Y is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Algorithms

If the input to an autoencoder is a vector $x \in \mathbb{R}^{D_X}$, then the encoder maps the vector x to another vector $z \in \mathbb{R}^{D^{(1)}}$ as follows:

$$z = h^{(1)}\left(W^{(1)}x + b^{(1)}\right),$$

where the superscript (1) indicates the first layer. $h^{(1)}: \mathbb{R}^{D^{(1)}} \to \mathbb{R}^{D^{(1)}}$ is a transfer function for the encoder, $W^{(1)} \in \mathbb{R}^{D^{(1)} \times D_X}$ is a weight matrix, and $b^{(1)} \in \mathbb{R}^{D^{(1)}}$ is a bias vector. Then, the decoder maps the encoded representation z back into an estimate of the original input vector, x, as follows:

$$\widehat{x} = h^{(2)}\left(W^{(2)}z + b^{(2)}\right),$$

where the superscript (2) represents the second layer. $h^{(2)}: \mathbb{R}^{D_X} \to \mathbb{R}^{D_X}$ is the transfer function for the decoder,$W^{(1)} \in \mathbb{R}^{D_X \times D^{(1)}}$ is a weight matrix, and $b^{(2)} \in \mathbb{R}^{D_X}$ is a bias vector.

See Also

`encode` | `trainAutoencoder`

Introduced in R2015b

encode

Class: `Autoencoder`

Encode input data

Syntax

```
Z = encode(autoenc,Xnew)
```

Description

`Z = encode(autoenc,Xnew)` returns the encoded data on page 2-534, `Z`, for the input data `Xnew`, using the autoencoder, `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

`Xnew` — Input data

matrix | cell array of image data | array of single image data

Input data, specified as a matrix of samples, a cell array of image data, or an array of single image data.

If the autoencoder `autoenc` was trained on a matrix, where each column represents a single sample, then `Xnew` must be a matrix, where each column represents a single sample.

If the autoencoder `autoenc` was trained on a cell array of images, then `Xnew` must either be a cell array of image data or an array of single image data.

Data Types: `single` | `double` | `cell`

Output Arguments

`Z` — Data encoded by `autoenc`

matrix

Data encoded by `autoenc`, specified as a matrix. Each column of `Z` represents an encoded sample (observation).

Data Types: `single` | `double`

Examples

Encode Decoded Data for New Images

Load the sample data.

```
X = digitTrainCellArrayData;
```

`X` is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden size of 50 using the training data.

```
autoenc = trainAutoencoder(X,50);
```

Encode decoded data for new image data.

```
Xnew = digitTestCellArrayData;
Z = encode(autoenc,Xnew);
```

`Xnew` is a 1-by-5000 cell array. `Z` is a 50-by-5000 matrix, where each column represents the image data of one handwritten digit in the new data `Xnew`.

Algorithms

If the input to an autoencoder is a vector $x \in \mathbb{R}^{D_X}$, then the encoder maps the vector x to another vector $z \in \mathbb{R}^{D^{(1)}}$ as follows:

$$z = h^{(1)}\left(W^{(1)}x + b^{(1)}\right),$$

where the superscript (1) indicates the first layer. $h^{(1)}: \mathbb{R}^{D^{(1)}} \to \mathbb{R}^{D^{(1)}}$ is a transfer function for the encoder, $W^{(1)} \in \mathbb{R}^{D^{(1)} \times D_X}$ is a weight matrix, and $b^{(1)} \in \mathbb{R}^{D^{(1)}}$ is a bias vector.

See Also

`trainAutoencoder` | `decode` | `stack`

Introduced in R2015b

generateFunction

Class: `Autoencoder`

Generate a MATLAB function to run the autoencoder

Syntax

```
generateFunction(autoenc)
generateFunction(autoenc,pathname)
generateFunction(autoenc,pathname,Name,Value)
```

Description

`generateFunction(autoenc)` generates a complete stand-alone function in the current directory, to run the autoencoder `autoenc` on input data.

`generateFunction(autoenc,pathname)` generates a complete stand-alone function to run the autoencoder `autoenc` on input data in the location specified by `pathname`.

`generateFunction(autoenc,pathname,Name,Value)` generates a complete stand-alone function with additional options specified by the `Name,Value` pair argument.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

`pathname` — Location for generated function

string

Location for generated function, specified as a string.

Example: `'C:\MyDocuments\Autoencoders'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`ShowLinks` — Indicator to display the links to the generated code

`false` (default) | `true`

Indicator to display the links to the generated code in the command window, specified as the comma-separated pair consisting of `'ShowLinks'` and either `true` or `false`.

Example: `'ShowLinks',true`

Data Types: `logical`

Examples

Generate MATLAB Function for Running Autoencoder

Load the sample data.

```
X = iris_dataset;
```

Train an autoencoder with 4 neurons in the hidden layer.

```
autoenc = trainAutoencoder(X,4);
```

Generate the code for running the autoencoder. Show the links to the MATLAB function.

```
generateFunction(autoenc)
```

```
MATLAB function generated: neural_function.m
To view generated function code: edit neural_function
For examples of using function: help neural_function
```

Generate the code for the autoencoder in a specific path.

```
generateFunction(autoenc,'H:\Documents\Autoencoder')
```

```
MATLAB function generated: H:\Documents\Autoencoder.m
To view generated function code: edit Autoencoder
For examples of using function: help Autoencoder
```

Tips

- If you do not specify the path and the file name, `generateFunction`, by default, creates the code in an m-file with the name `neural_function.m`. You can change the file name after `generateFunction` generates it. Or you can specify the path and file name using the `pathname` input argument in the call to `generateFunction`.

See Also

`genFunction` | `generateSimulink`

Introduced in R2015b

generateSimulink

Class: `Autoencoder`

Generate a Simulink model for the autoencoder

Syntax

```
generateSimulink(autoenc)
```

Description

`generateSimulink(autoenc)` creates a Simulink model for the autoencoder, `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

Examples

Generate Simulink Model for Autoencoder

Load the training data.

```
X = digitsmall_dataset;
```

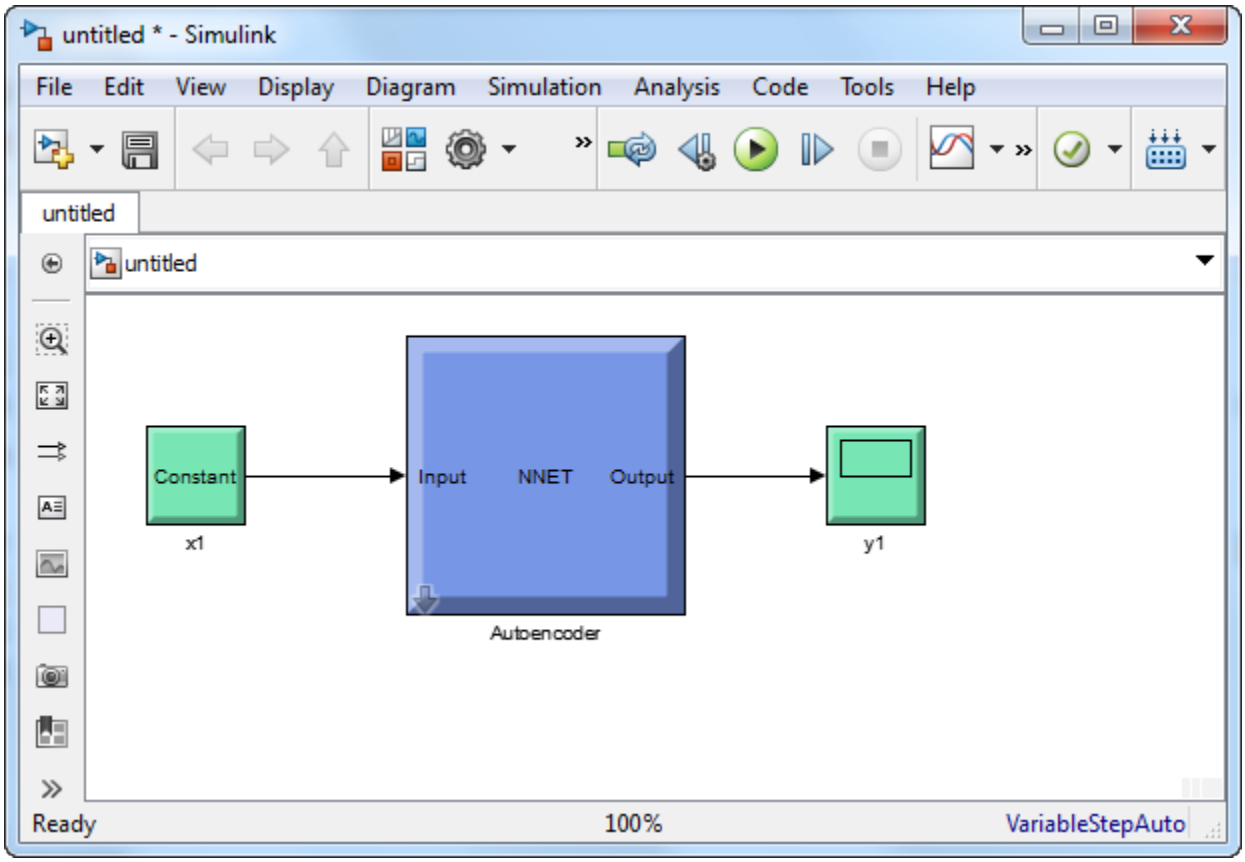
The training data is a 1-by-500 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.

```
hiddenSize = 25;
autoenc = trainAutoencoder(X,hiddenSize,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.15);
```

Create a Simulink model for the autoencoder, `autoenc`.

```
generateSimulink(autoenc)
```



See Also

`trainAutoencoder`

Introduced in R2015b

network

Class: `Autoencoder`

Convert `Autoencoder` object into `network` object

Syntax

```
net = network(autoenc)
```

Description

`net = network(autoenc)` returns a network object which is equivalent to the autoencoder, `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

Output Arguments

`net` — Neural network

`network` object

Neural network, that is equivalent to the autoencoder `autoenc`, returned as an object of the `network` class.

Examples

Create Network from Autoencoder

Load the sample data.

```
X = bodyfat_dataset;
```

X is a 13-by-252 matrix defining thirteen attributes of 252 different observations. For more information on the data, type `help bodyfat_dataset` in the command line.

Train an autoencoder on the attribute data.

```
autoenc = trainAutoencoder(X);
```

Create a network object from the autoencoder, `autoenc` .

```
net = network(autoenc);
```

Predict the attributes using the network, `net` .

```
Xpred = net(X);
```

Fit a linear regression model between the actual and estimated attributes data. Compute the estimated Pearson correlation coefficient, the slope and the intercept (bias) of the regression model, using all attribute data as one data set.

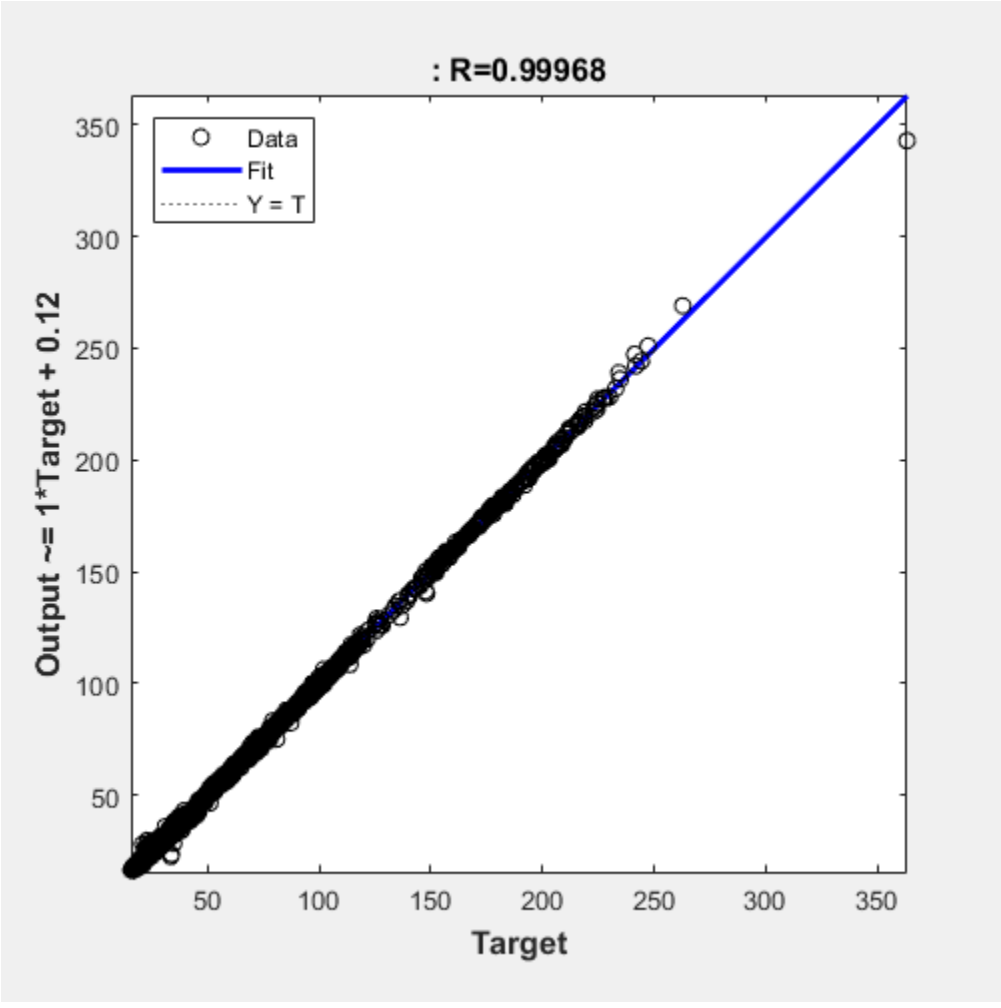
```
[C, S, B] = regression(X, Xpred, 'one')
```

```
C = 0.9997

S = 0.9983

B = 0.1160
```

The correlation coefficient is almost 1, which indicates that the attributes data and the estimations from the neural network are highly close to each other.

Plot the actual data and the fitted line.

```
plotregression(X, Xpred);
```



The data appears to be on the fitted line, which visually supports the conclusion that the predictions are very close to the actual data.

See Also

`trainAutoencoder` | `Autoencoder`

Introduced in R2015b

plotWeights

Class: `Autoencoder`

Plot a visualization of the weights for the encoder of an autoencoder

Syntax

```
plotWeights(autoenc)
h = plotWeights(autoenc)
```

Description

`plotWeights(autoenc)` visualizes the weights for the autoencoder, `autoenc`.

`h = plotWeights(autoenc)` returns a function handle `h`, for the visualization of the encoder weights for the autoencoder, `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

Output Arguments

`h` — Image object

handle

Image object, returned as a handle.

Examples

Visualize Learned Features

Load the training data.

```
X = digitTrainCellArrayData;
```

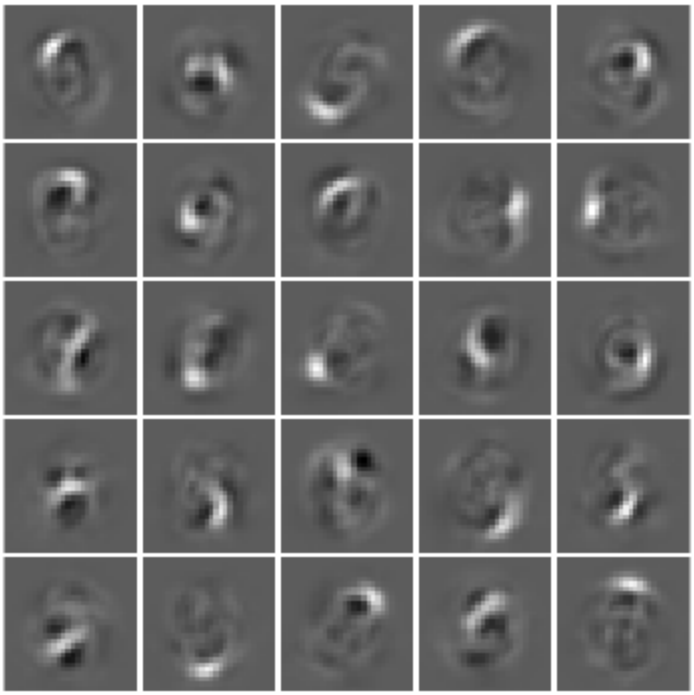
The training data is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer of 25 neurons.

```
hiddenSize = 25;
autoenc = trainAutoencoder(X,hiddenSize, ...
  'L2WeightRegularization',0.004, ...
  'SparsityRegularization',4, ...
  'SparsityProportion',0.2);
```

Visualize the learned features.

```
plotWeights(autoenc);
```



Tips

- `plotWeights` allows the visualization of the features that the autoencoder learns. Use it when the autoencoder is trained on image data. The visualization of the weights has the same dimensions as the images used for training.

See Also

`trainAutoencoder`

Introduced in R2015b

predict

Class: `Autoencoder`

Reconstruct the inputs using trained autoencoder

Syntax

```
Y = predict(autoenc,X)
```

Description

`Y = predict(autoenc,X)` returns the predictions `Y` for the input data `X`, using the autoencoder `autoenc`. The result `Y` is a reconstruction of `X`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

`Xnew` — Input data

matrix | cell array of image data | array of single image data

Input data, specified as a matrix of samples, a cell array of image data, or an array of single image data.

If the autoencoder `autoenc` was trained on a matrix, where each column represents a single sample, then `Xnew` must be a matrix, where each column represents a single sample.

If the autoencoder `autoenc` was trained on a cell array of images, then `Xnew` must either be a cell array of image data or an array of single image data.

Data Types: `single` | `double` | `cell`

Output Arguments

`Y` — Predictions for the input data `Xnew`

matrix | cell array of image data | array of single image data

Predictions for the input data `Xnew`, returned as a matrix or a cell array of image data.

- If `Xnew` is a matrix, then `Y` is also a matrix, where each column corresponds to a single sample (observation or example).
- If `Xnew` is a cell array of image data, then `Y` is also a cell array of image data, where each cell contains the data for a single image.
- If `Xnew` is an array of a single image data, then `Y` is also an array of a single image data.

Examples

Predict Continuous Measurements Using Trained Autoencoder

Load the training data.

```
X = iris_dataset;
```

The training data contains measurements on four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

Train an autoencoder on the training data using the positive saturating linear transfer function in the encoder and linear transfer function in the decoder.

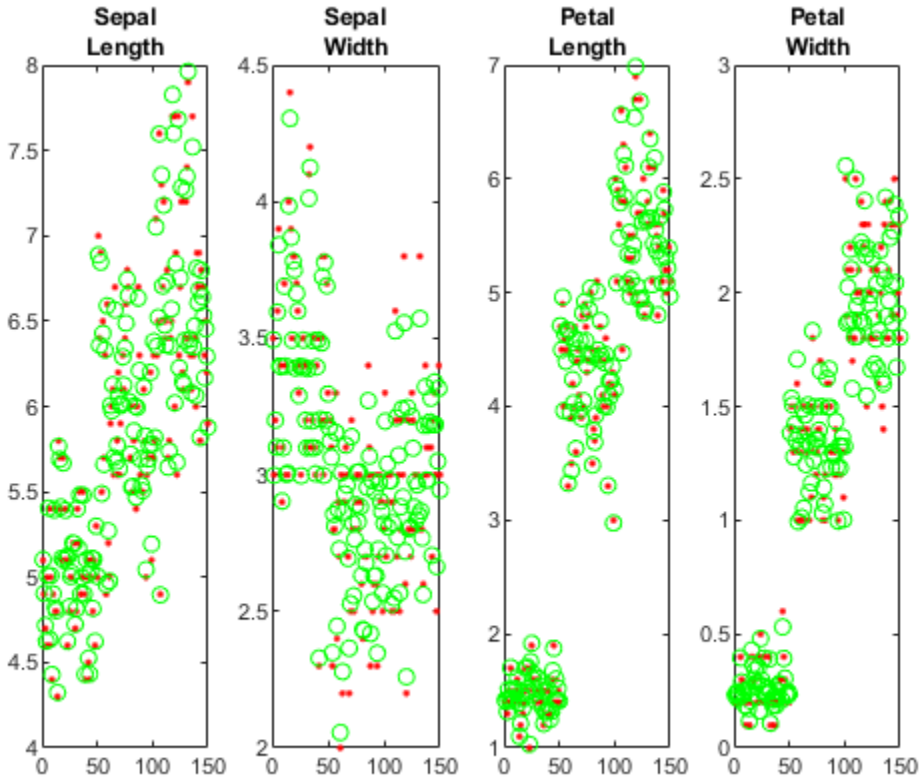
```
autoenc = trainAutoencoder(X,'EncoderTransferFunction',...
'satlin','DecoderTransferFunction','purelin');
```

Reconstruct the measurements using the trained network, `autoenc`.

```
xReconstructed = predict(autoenc,X);
```

Plot the predicted measurement values along with the actual values in the training dataset.

```
for i = 1:4
h(i) = subplot(1,4,i);
plot(X(i,:),'r.');
hold on
plot(xReconstructed(i,:),'go');
hold off;
end
title(h(1),{'Sepal';'Length'});
title(h(2),{'Sepal';'Width'});
title(h(3),{'Petal';'Length'});
title(h(4),{'Petal';'Width'});
```



The red dots represent the training data and the green circles represent the reconstructed data.

Reconstruct Handwritten Digit Images Using Sparse Autoencoder

Load the training data.

```
XTrain = digitTrainCellArrayData;
```

The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.

```
hiddenSize = 25;
autoenc = trainAutoencoder(XTrain,hiddenSize,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.15);
```

Load the test data.

```
XTest = digitTestCellArrayData;
```

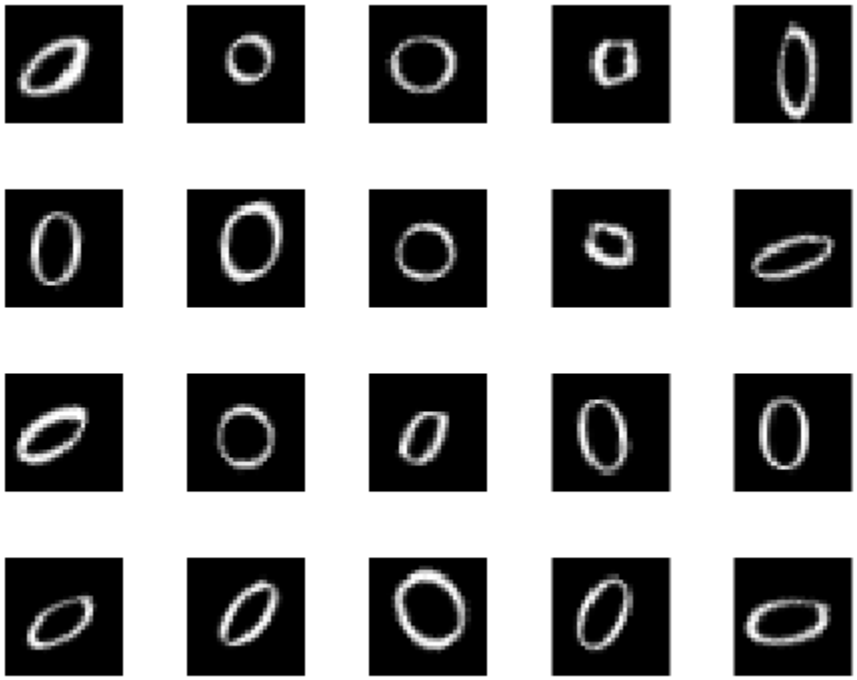
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,XTest);
```

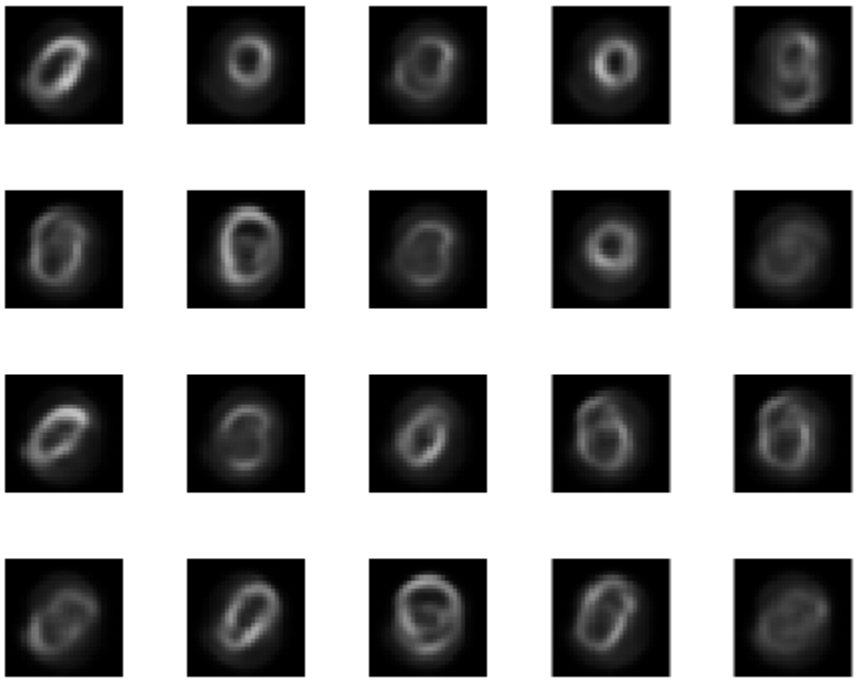
View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(XTest{i});
end
```



View the reconstructed test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(xReconstructed{i});
end
```



See Also

`trainAutoencoder`

Introduced in R2015b

stack

Class: `Autoencoder`

Stack encoders from several autoencoders together

Syntax

```
stackednet = stack(autoenc1,autoenc2,...)
stackednet = stack(autoenc1,autoenc2,...,net1)
```

Description

`stackednet = stack(autoenc1,autoenc2,...)` returns a `network` object created by stacking the encoders of the autoencoders, `autoenc1`, `autoenc2`, and so on.

`stackednet = stack(autoenc1,autoenc2,...,net1)` returns a network object created by stacking the encoders of the autoencoders and the network object `net1`.

The autoencoders and the network object can be stacked only if their dimensions match.

Input Arguments

`autoenc1` — Trained autoencoder
`Autoencoder` object

Trained autoencoder, specified as an `Autoencoder` object.

`autoenc2` — Trained autoencoder
`Autoencoder` object

Trained autoencoder, specified as an `Autoencoder` object.

`net1` — Trained neural network
`network` object

Trained neural network, specified as a `network` object. `net1` can be a softmax layer, trained using the `trainSoftmaxLayer` function.

Output Arguments

`stackednet` — Stacked neural network
`network` object

Stacked neural network (deep network), returned as a `network` object

Examples

Create a Stacked Network

Load the training data.

```
[X,T] = iris_dataset;
```

Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
hiddenSize = 5;
autoenc = trainAutoencoder(X, hiddenSize, ...
    'L2WeightRegularization', 0.001, ...
    'SparsityRegularization', 4, ...
    'SparsityProportion', 0.05, ...
    'DecoderTransferFunction','purelin');
```

Extract the features in the hidden layer.

```
features = encode(autoenc,X);
```

Train a softmax layer for classification using the `features` .

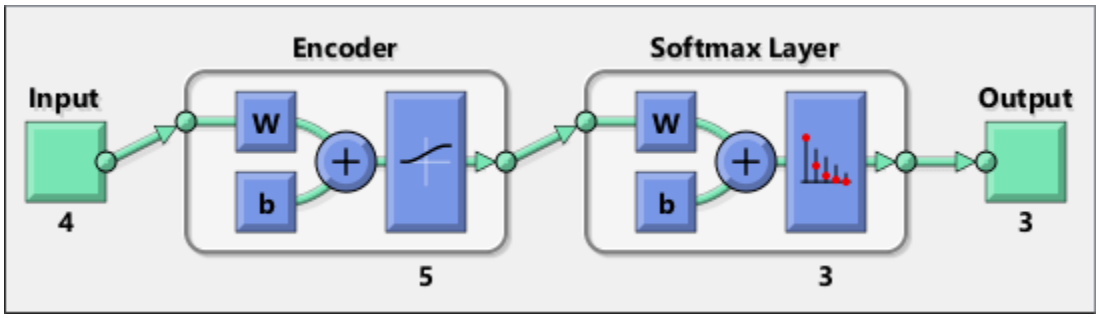
```
softnet = trainSoftmaxLayer(features,T);
```

Stack the encoder and the softmax layer to form a deep network.

```
stackednet = stack(autoenc,softnet);
```

View the stacked network.

```
view(stackednet);
```



Tips

- The size of the hidden representation of one autoencoder must match the input size of the next autoencoder or network in the stack.

  The first input argument of the stacked network is the input argument of the first autoencoder. The output argument from the encoder of the first autoencoder is the input of the second autoencoder in the stacked network. The output argument from the encoder of the second autoencoder is the input argument to the third autoencoder in the stacked network, and so on.
- The stacked network object `stacknet` inherits its training parameters from the final input argument `net1`.

See Also

`trainAutoencoder` | `Autoencoder`

Topics

“Train Stacked Autoencoders for Image Classification”

Introduced in R2015b

view

Class: `Autoencoder`

View autoencoder

Syntax

```
view(autoenc)
```

Description

`view(autoenc)` returns a diagram of the autoencoder, `autoenc`.

Input Arguments

`autoenc` — Trained autoencoder

`Autoencoder` object

Trained autoencoder, returned as an object of the `Autoencoder` class.

Examples

View Autoencoder

Load the training data.

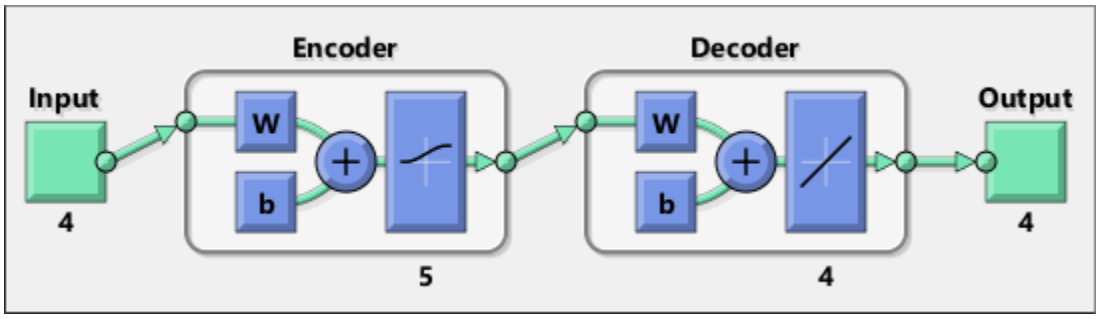
```
X = iris_dataset;
```

Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
hiddenSize = 5;
autoenc = trainAutoencoder(X, hiddenSize, ...
    'L2WeightRegularization',0.001, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.05, ...
    'DecoderTransferFunction','purelin');
```

View the autoencoder.

```
view(autoenc)
```



See Also

`trainAutoencoder`

Introduced in R2015b

fitnet

Function fitting neural network

Syntax

```
net = fitnet(hiddenSizes)
net = fitnet(hiddenSizes,trainFcn)
```

Description

`net = fitnet(hiddenSizes)` returns a function fitting neural network with a hidden layer size of `hiddenSizes`.

`net = fitnet(hiddenSizes,trainFcn)` returns a function fitting neural network with a hidden layer size of `hiddenSizes` and training function, specified by `trainFcn`.

Examples

Construct and Train a Function Fitting Network

Load the training data.

```
[x,t] = simplefit_dataset;
```

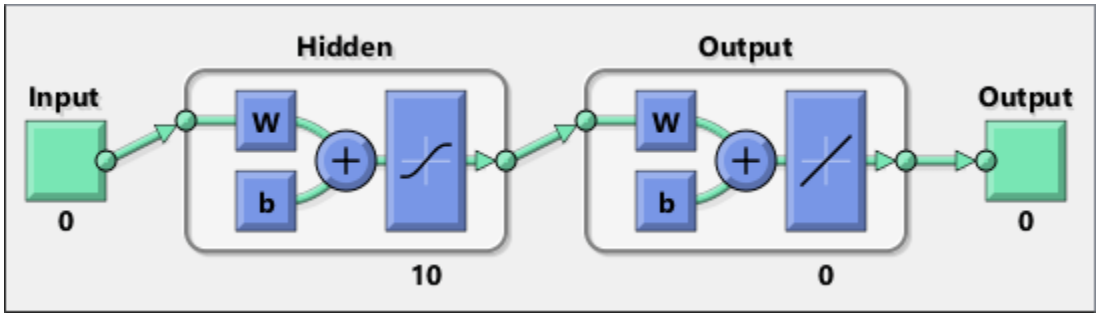
The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a function fitting neural network with one hidden layer of size 10.

```
net = fitnet(10);
```

View the network.

```
view(net)
```



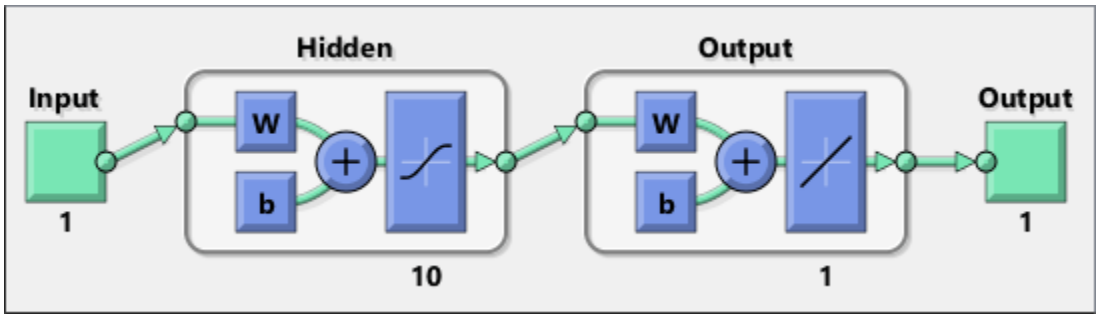
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```



You can see that the sizes of the input and output are 1.

Estimate the targets using the trained network.

```
y = net(x);
```

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,y,t)
```

```
perf =

   1.4639e-04
```

The default training algorithm for a function fitting network is Levenberg-Marquardt (`'trainlm'`). Use the Bayesian regularization training algorithm and compare the performance results.

```
net = fitnet(10,'trainbr');
net = train(net,x,t);
y = net(x);
perf = perform(net,y,t)
```

```
perf =

   3.3529e-10
```

The Bayesian regularization training algorithm improves the performance of the network in terms of estimating the target values.

Input Arguments

`hiddenSizes` — Size of the hidden layers

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: `[10,8,5]`

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

`trainFcn` — Training function name
`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...

Training function name, specified as one of the following.

Training 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

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `'traingdx'`

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

Output Arguments

`net` — Function fitting network
`network` object

Function fitting network, returned as a `network` object.

Tips

- Function fitting is the process of training a neural network on a set of inputs in order to produce an associated set of target outputs. After you construct the network with the desired hidden layers

and the training algorithm, you must train it using a set of training data. Once the neural network has fit the data, it forms a generalization of the input-output relationship. You can then use the trained network to generate outputs for inputs it was not trained on.

See Also

`train` | `feedforwardnet` | `network` | `nftool` | `trainlm` | `perform`

Topics

“Fit Data with a Shallow Neural Network”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

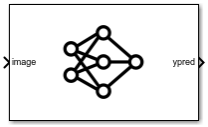
Introduced in R2010b

3

Deep Learning Blocks

Image Classifier

Classify data using a trained deep learning neural network
Library: Deep Learning Toolbox / Deep Neural Networks



Description

The Image Classifier block predicts class labels for the data at the input by using the trained network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

Limitations

- The Image Classifier block does not support sequence networks and multiple input and multiple output networks (MIMO).
- The Image Classifier block does not support MAT-file logging.

Ports

Input

`image` — Image or feature data

numeric array

A *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.

A *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

If the array contains `NaN`s, then they are propagated through the network.

Output

`ypred` — Predicted class labels

enumerated

Predicted class labels with the highest score, returned as a *N*-by-1 enumerated vector of labels, where *N* is the number of observations.

`scores` — Predicted class scores

matrix

Predicted scores, returned as a *N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.

`labels` — Class labels for predicted scores

matrix

Labels associated with the predicted scores, returned as a *N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.

Parameters

`Network` — Source for trained network

`Network from MAT-file` (default) | `Network from MATLAB function`

Specify the source for the trained network. Select one of the following:

- `Network from MAT-file`— Import a trained network from a MAT-file containing a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.
- `Network from MATLAB function`— Import a pretrained network from a MATLAB function. For example, by using the `googlenet` function.

Programmatic Use

Block Parameter: `Network`
Type: character vector, string
Values: `'Network from MAT-file'` | `'Network from MATLAB function'`
Default: `'Network from MAT-file'`

`File path` — MAT-file containing trained network

`untitled.mat` (default) | MAT-file path or name

This parameter specifies the name of the MAT-file that contains the trained deep learning network to load. If the file is not on the MATLAB path, use the **Browse** button to locate the file.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MAT-file`.

Programmatic Use

Block Parameter: `NetworkFilePath`
Type: character vector, string
Values: MAT-file path or name
Default: `'untitled.mat'`

`MATLAB function` — MATLAB function name

`squeezenet` (default) | MATLAB function name

This parameter specifies the name of the MATLAB function for the pretrained deep learning network. For example, use `googlenet` function to import the pretrained GoogLeNet model.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MATLAB function`.

Programmatic Use

Block Parameter: `NetworkFunction`
Type: character vector, string
Values: MATLAB function name
Default: `'squeezenet'`

`Mini-batch size` — Size of mini-batches
128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

Programmatic Use
Block Parameter: `MiniBatchSize`
Type: character vector, string
Values: positive integer
Default: `'128'`

`Resize input` — Resize input dimensions
`on` (default) | `off`

Resize the data at the input port to the input size of the network.

Programmatic Use
Block Parameter: `ResizeInput`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Classification` — Output predicted label with highest score
`on` (default) | `off`

Enable output port `ypred` that outputs the label with the highest score.

Programmatic Use
Block Parameter: `Classification`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Predictions` — Output all scores and associated labels
`off` (default) | `on`

Enable output ports `scores` and `labels` that output all predicted scores and associated class labels.

Programmatic Use
Block Parameter: `Predictions`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters** > **Code Generation** general category must be set to `C++`.

- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

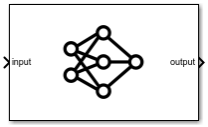
- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- For a list of networks and layers supported for CUDA code generation, see “Supported Networks, Layers, and Classes” (GPU Coder).
- To learn more about generating code for Simulink models containing the Image Classifier block, see “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” (GPU Coder).

See Also
Predict

Introduced in R2020b

Predict

Predict responses using a trained deep learning neural network
Library: Deep Learning Toolbox / Deep Neural Networks



Description

The Predict block predicts responses for the data at the input by using the trained network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

Note Use the Predict block to make predictions in Simulink. To make predictions programmatically using MATLAB code, use the `classify` and `predict` functions.

Ports

Input

`input` — Image, feature, sequence, or time series data

numeric array

The input ports of the Predict block takes the names of the input layers of the network loaded. For example, if you specify `googlenet` for `MATLAB function`, then the input port of the Predict block is labeled **data**. Based on the network loaded, the input to the predict block can be image, sequence, or time series data.

The format of the input depend on the type of data.

Data	Format of Predictors
2-D images	A *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.
Vector sequence	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.

Data	Format of Predictors
Features	A *N*-by-`numFeatures` numeric array, where *N* is the number of observations and `numFeatures` is the number of features of the input data.

If the array contains `NaN`s, then they are propagated through the network.

Output

`output` — Predicted scores, responses, or activations

numeric array

The outputs port of the Predict block takes the names of the output layers of the network loaded. For example, if you specify `googlenet` for `MATLAB function`, then the output port of the Predict block is labeled **output**. Based on the network loaded, the output of the Predict block can represent predicted scores or responses.

Predicted scores or responses, returned as a *N*-by-*K* array, where *N* is the number of observations, and *K* is the number of classes.

If you enable `Activations` for a network layer, the Predict block creates a new output port with the name of the selected network layer. This port outputs the activations from the selected network layer.

The activations from the network layer is returned as a numeric array. The format of output depends on the type of input data and the type of layer output.

For 2-D image output, activations is an *h*-by-*w*-by-*c*-by-*n* array, where *h*, *w*, and *c* are the height, width, and number of channels for the output of the chosen layer, respectively, and *n* is the number of images.

For a single time-step containing vector data, activations is a *c*-by-*n* matrix, where *n* is the number of sequences and *c* is the number of features in the sequence.

For a single time-step containing 2-D image data, activations is a *h*-by-*w*-by-*c*-by-*n* array, where *n* is the number of sequences, *h*, *w*, and *c* are the height, width, and the number of channels of the images, respectively.

Parameters

`Network` — Source for trained network

`Network from MAT-file` (default) | `Network from MATLAB function`

Specify the source for the trained network. Select one of the following:

- `Network from MAT-file`— Import a trained network from a MAT-file containing a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.
- `Network from MATLAB function`— Import a pretrained network from a MATLAB function. For example, by using the `googlenet` function.

Programmatic Use

Block Parameter: `Network`
Type: character vector, string
Values: `'Network from MAT-file'` | `'Network from MATLAB function'`

Default: `'Network from MAT-file'`

`File path` — MAT-file containing trained network

`untitled.mat` (default) | MAT-file path or name

This parameter specifies the name of the MAT-file that contains the trained deep learning network to load. If the file is not on the MATLAB path, use the **Browse** button to locate the file.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MAT-file`.

Programmatic Use

Block Parameter: `NetworkFilePath`
Type: character vector, string
Values: MAT-file path or name
Default: `'untitled.mat'`

`MATLAB function` — MATLAB function name

`squeezenet` (default) | MATLAB function name

This parameter specifies the name of the MATLAB function for the pretrained deep learning network. For example, use `googlenet` function to import the pretrained GoogLeNet model.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MATLAB function`.

Programmatic Use

Block Parameter: `NetworkFunction`
Type: character vector, string
Values: MATLAB function name
Default: `'squeezenet'`

`Mini-batch size` — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

Programmatic Use

Block Parameter: `MiniBatchSize`
Type: character vector, string
Values: positive integer
Default: `'128'`

`Predictions` — Output predicted scores or responses

`on` (default) | `off`

Enable output ports that return predicted scores or responses.

Programmatic Use

Block Parameter: `Predictions`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Activations` — Output network activations for a specific layer

layers of the network

Use the **Activations** list to select the layer to extract features from. The selected layers appear as an output port of the Predict block.

Programmatic Use

Block Parameter: `Activations`
Type: character vector, string
Values: character vector in the form of `'{'layerName1',layerName2',...}'`
Default: `''`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- For a list of networks and layers supported for CUDA code generation, see “Supported Networks, Layers, and Classes” (GPU Coder).
- To learn more about generating code for Simulink models containing the Predict block, see “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” (GPU Coder).

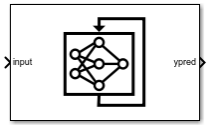
See Also

Image Classifier | Stateful Classify | Stateful Predict

Introduced in R2020b

Stateful Classify

Classify data using a trained deep learning recurrent neural network
Library: Deep Learning Toolbox / Deep Neural Networks



Description

The Stateful Classify block predicts class labels for the data at the input by using the trained recurrent neural network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function. This block updates the state of the network with every prediction.

Limitations

The Stateful Classify block does not support MAT-file logging.

Ports

Input

`input` — Sequence or time series data
numeric array

The format of the input depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.

Output

`ypred` — Predicted class labels
enumerated

Predicted class labels with the highest score, returned as a *N*-by-1 enumerated vector of labels, where *N* is the number of observations.

`scores` — Predicted class scores
matrix

Predicted scores, returned as a *N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.

`labels` — Class labels for predicted scores

matrix

Labels associated with the predicted scores, returned as a *N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.

Parameters

`Network` — Source for trained recurrent neural network

`Network from MAT-file` (default) | `Network from MATLAB function`

Specify the source for the trained recurrent neural network. The trained network must have at least one recurrent layer (for example, an LSTM network). Select one of the following:

- `Network from MAT-file`— Import a trained recurrent neural network from a MAT-file containing a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.
- `Network from MATLAB function`— Import a pretrained recurrent neural network from a MATLAB function.

Programmatic Use

Block Parameter: `Network`
Type: character vector, string
Values: `'Network from MAT-file'` | `'Network from MATLAB function'`
Default: `'Network from MAT-file'`

`File path` — MAT-file containing trained recurrent neural network

`untitled.mat` (default) | MAT-file name

This parameter specifies the name of the MAT-file that contains the trained recurrent neural network to load. If the file is not on the MATLAB path, use the **Browse** button to locate the file.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MAT-file`.

Programmatic Use

Block Parameter: `NetworkFilePath`
Type: character vector, string
Values: MAT-file path or name
Default: `'untitled.mat'`

`MATLAB function` — MATLAB function name

`untitled` (default) | MATLAB function name

This parameter specifies the name of the MATLAB function for the pretrained recurrent neural network.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MATLAB function`.

Programmatic Use
Block Parameter: `NetworkFunction`
Type: character vector, string
Values: MATLAB function name
Default: `'untitled'`

`Sample time` — Output sample period and optional time offset

`-1` (default) | scalar | vector

The **Sample time** parameter specifies when the block computes a new output value during simulation. For details, see "Specify Sample Time" (Simulink).

Specify the **Sample time** parameter as a scalar when you do not want the output to have a time offset. To add a time offset to the output, specify the **Sample time** parameter as a 1-by-2 vector where the first element is the sampling period and the second element is the offset.

By default, the **Sample time** parameter value is `-1` to inherit the value.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

`Classification` — Output predicted label with highest score

`on` (default) | `off`

Enable output port `ypred` that outputs the label with the highest score.

Programmatic Use
Block Parameter: `Classification`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'on'`

`Predictions` — Output all scores and associated labels

`off` (default) | `on`

Enable output ports `scores` and `labels` that output all predicted scores and associated class labels.

Programmatic Use
Block Parameter: `Predictions`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters** > **Code Generation** general category must be set to `C++`.

- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

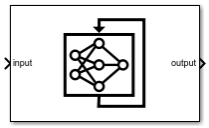
- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- GPU code generation supports this block only when targeting the cuDNN library.

See Also
Predict | Image Classifier | Stateful Predict

Introduced in R2021a

Stateful Predict

Predict responses using a trained recurrent neural network
Library: Deep Learning Toolbox / Deep Neural Networks



Description

The Stateful Predict block predicts responses for the data at the input by using the trained recurrent neural network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function. This block updates the state of the network with every prediction.

Ports

Input

`input` — Sequence or time series data
numeric array

The input ports of the Stateful Predict block takes the names of the input layers of the network loaded. Based on the network loaded, the input to the predict block can be sequence or time series data.

The dimensions of the numeric arrays containing the sequences depend on the type of data.

Input	Description
Vector sequences	*c*-by-*s* matrices, where *c* is the number of features of the sequences and *s* is the sequence length.
2-D image sequences	*h*-by-*w*-by-*c*-by-*s* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *s* is the sequence length.

Output

`output` — Predicted scores or responses
numeric array

The outputs port of the Stateful Predict block takes the names of the output layers of the network loaded. Based on the network loaded, the output of the Stateful Predict block can represent predicted scores or responses.

For sequence-to-label classification, the output is a *N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes.

For sequence-to-sequence classification problems, the output is a *K*-by-*S* matrix of scores, where *K* is the number of classes, and *S* is the total number of time steps in the corresponding input sequence.

Parameters

`Network` — Source for trained recurrent neural network

`Network from MAT-file` (default) | `Network from MATLAB function`

Specify the source for the trained recurrent neural network. The trained network must have at least one recurrent layer (for example, an LSTM network). Select one of the following:

- `Network from MAT-file`— Import a trained recurrent neural network from a MAT-file containing a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.
- `Network from MATLAB function`— Import a pretrained recurrent neural network from a MATLAB function.

Programmatic Use

Block Parameter: `Network`
Type: character vector, string
Values: `'Network from MAT-file'` | `'Network from MATLAB function'`
Default: `'Network from MAT-file'`

`File path` — MAT-file containing trained recurrent neural network

`untitled.mat` (default) | MAT-file name

This parameter specifies the name of the MAT-file that contains the trained recurrent neural network to load. If the file is not on the MATLAB path, use the **Browse** button to locate the file.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MAT-file`.

Programmatic Use

Block Parameter: `NetworkFilePath`
Type: character vector, string
Values: MAT-file path or name
Default: `'untitled.mat'`

`MATLAB function` — MATLAB function name

`untitled` (default) | MATLAB function name

This parameter specifies the name of the MATLAB function for the pretrained recurrent neural network.

Dependencies

To enable this parameter, set the **Network** parameter to `Network from MATLAB function`.

Programmatic Use

Block Parameter: `NetworkFunction`
Type: character vector, string
Values: MATLAB function name
Default: `'untitled'`

`Sample time` — Output sample period and optional time offset

`-1` (default) | scalar | vector

The **Sample time** parameter specifies when the block computes a new output value during simulation. For details, see “Specify Sample Time” (Simulink).

Specify the **Sample time** parameter as a scalar when you do not want the output to have a time offset. To add a time offset to the output, specify the **Sample time** parameter as a 1-by-2 vector where the first element is the sampling period and the second element is the offset.

By default, the **Sample time** parameter value is `-1` to inherit the value.

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to `C++`.
- GPU code generation supports this block only when targeting the cuDNN library.

See Also

Predict | Image Classifier | Stateful Classify

Introduced in R2021a