

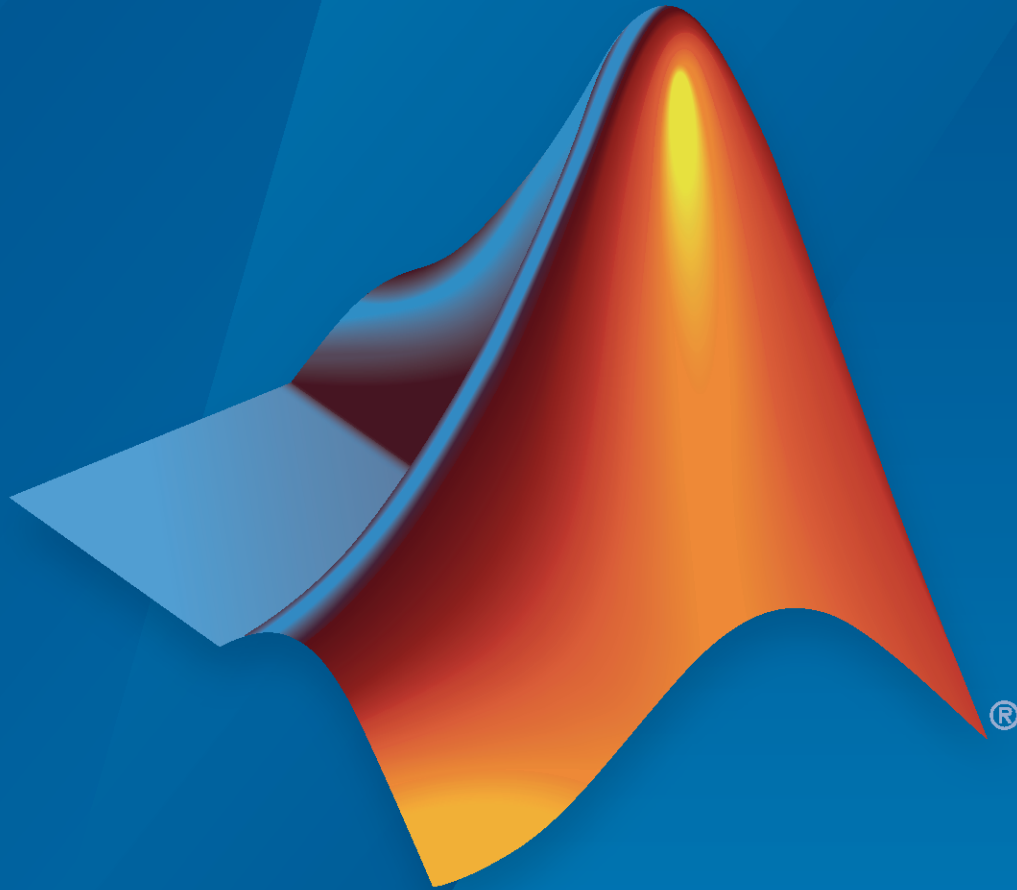
# Deep Learning Toolbox™

## Reference

*Mark Hudson Beale*

*Martin T. Hagan*

*Howard B. Demuth*



# MATLAB®

R2023a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://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*

© COPYRIGHT 1992–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## 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)
March 2022	Online only	Revised for Version 14.4 (Release 2022a)
September 2022	Online only	Revised for Version 14.5 (Release 2022b)
March 2023	Online only	Revised for Version 14.6 (Release 2023a)





<b>1</b>	<b>Deep Learning Functions</b>
<b>2</b>	<b>Approximation, Clustering, and Control Functions</b>
<b>3</b>	<b>Deep Learning Blocks</b>



# Deep Learning Functions

---

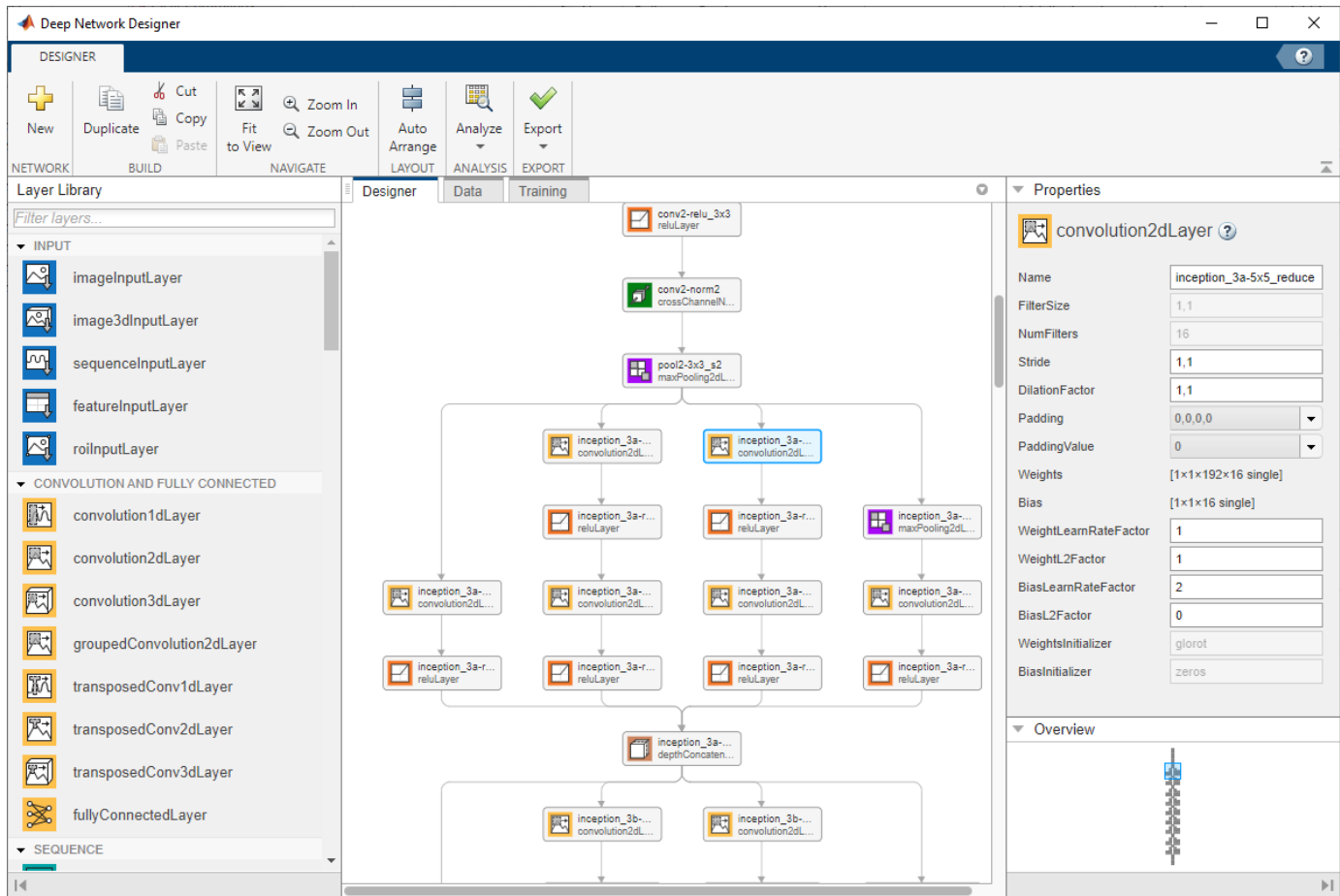
# Deep Network Designer

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 and create experiments for hyperparameter tuning using Experiment Manager.



## 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

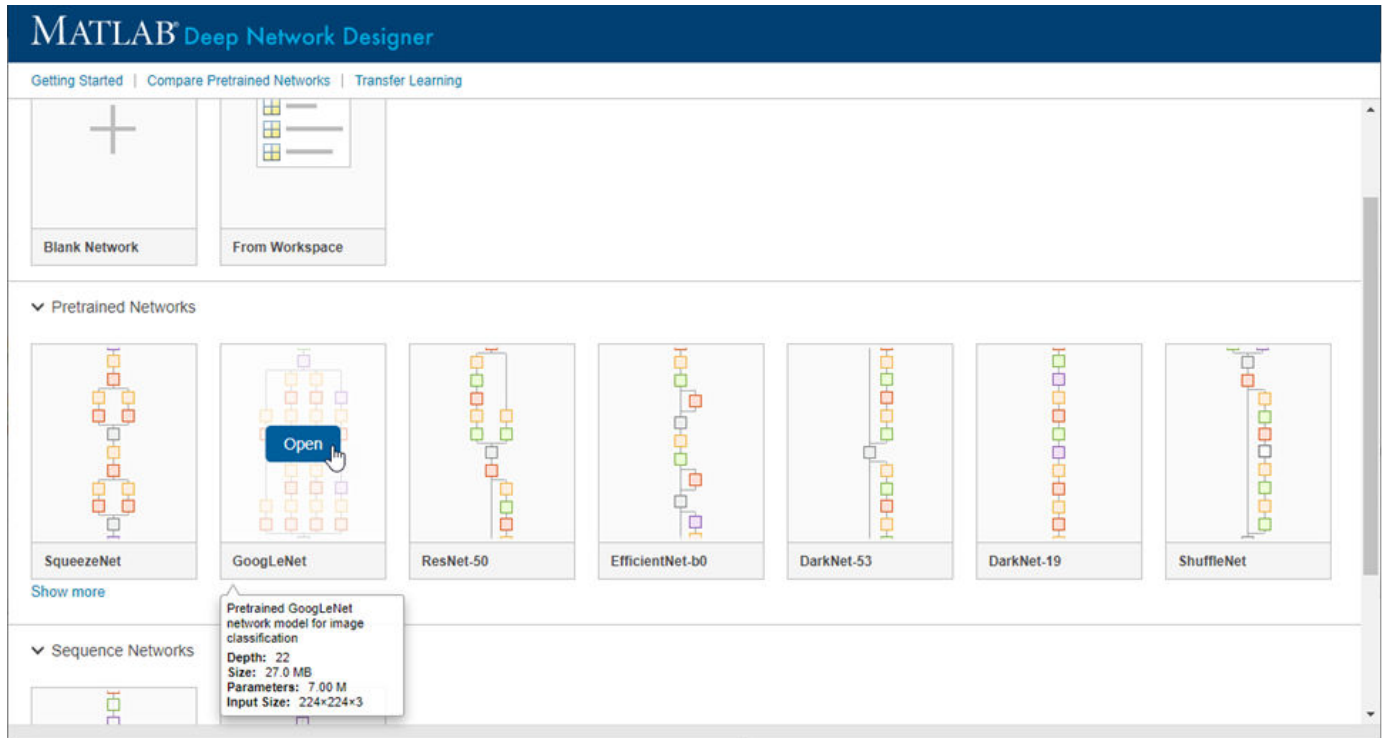
### Select Pretrained Network

Examine a pretrained network in Deep Network Designer.

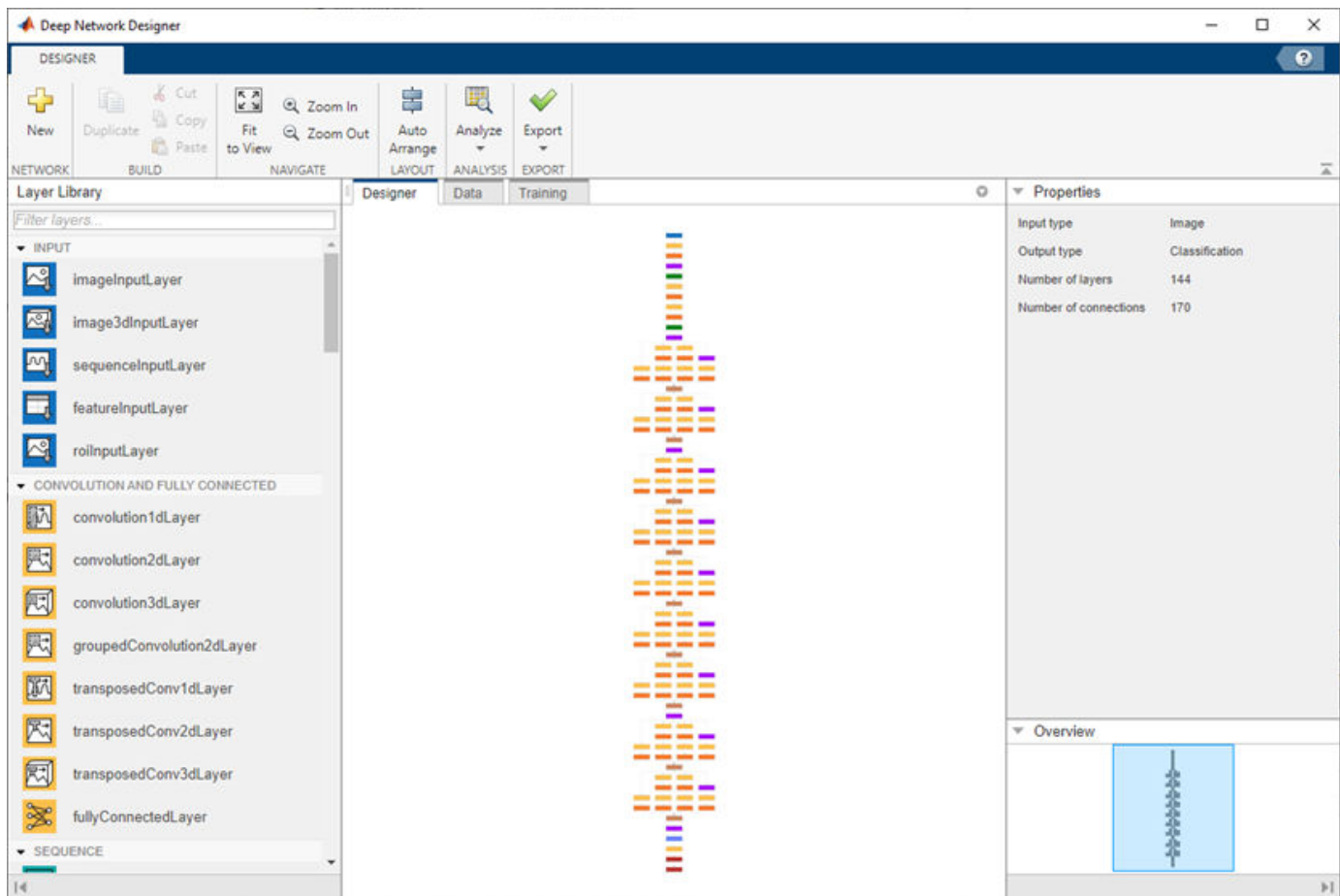
Open the app and select a pretrained network. You can also load a pretrained network by selecting the **Designer** tab and clicking **New**. If you need to download the network, then click **Install** to open the Add-On Explorer. Deep Network Designer has pretrained networks suitable for image and audio tasks. Loading pretrained audio networks requires Audio Toolbox™.

**Tip** To get started, try choosing one of the faster image classification 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 image classification 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.

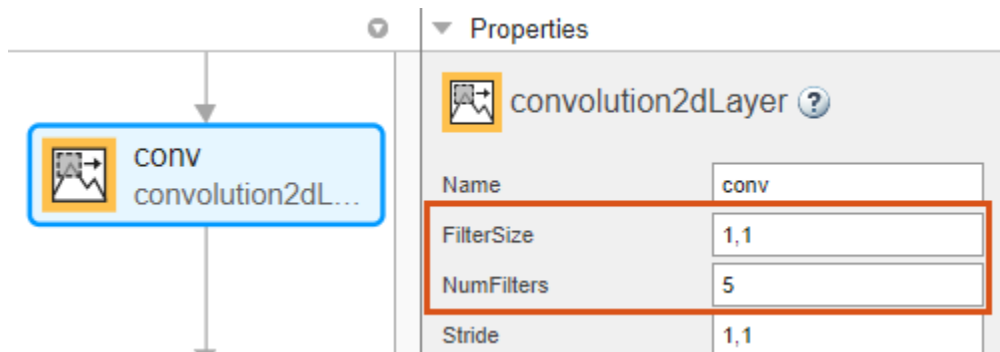
Transfer learning is the process of taking a pretrained deep learning network and fine-tuning it to learn a new task. You can quickly transfer learned features to a new task using a smaller number of training images. Transfer learning is therefore often faster and easier than training a network from 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)
```

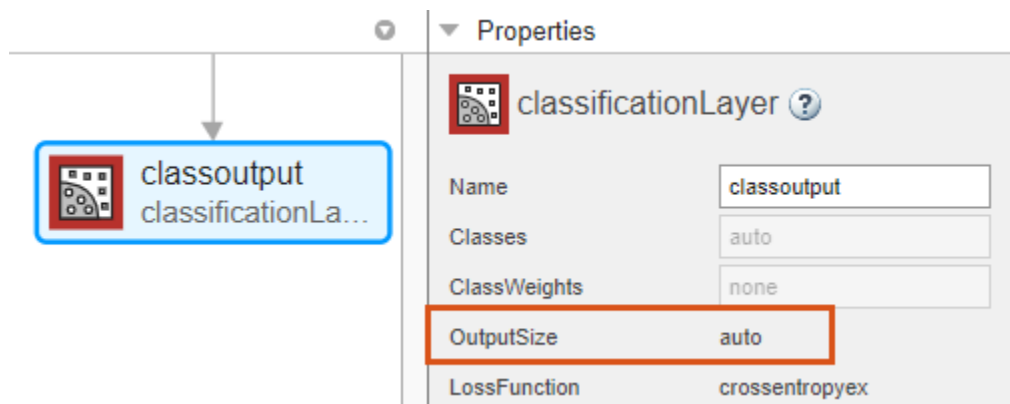
To prepare the network for transfer learning, replace the last learnable layer and the final classification layer. For SqueezeNet, the last learnable layer is a 2-D convolutional layer named 'conv10'.

- 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.



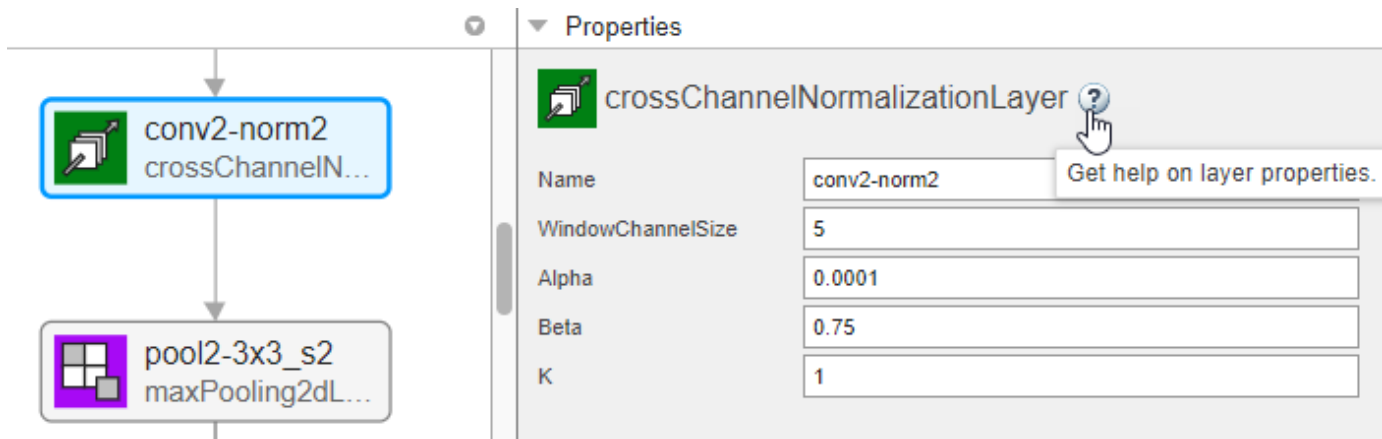
Check your network by clicking **Analyze** in the **Designer** tab. The network is ready for training if **Deep Learning Network Analyzer** reports zero errors. For an example showing how to train a network to classify new images, see “Transfer Learning with Deep Network Designer”.

### 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”.

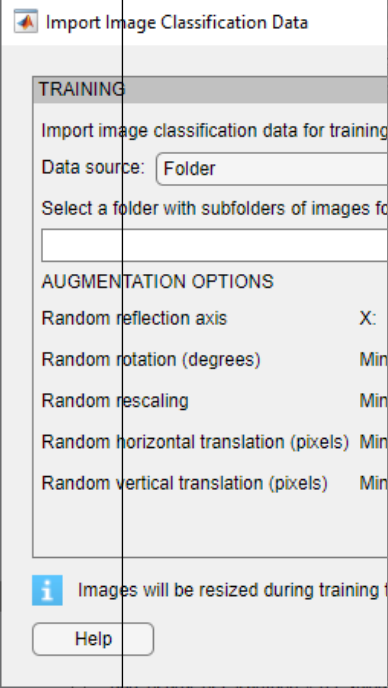
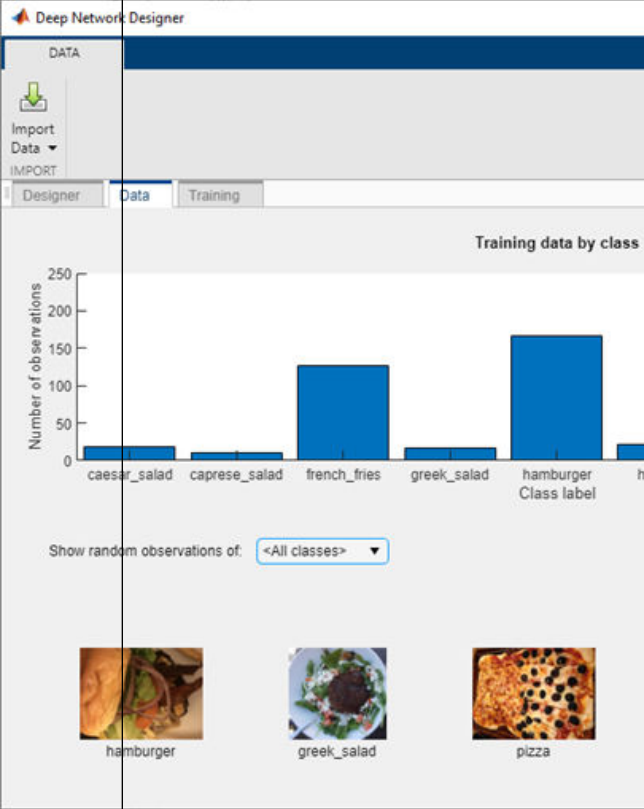
You can also combine networks in Deep Network Designer. For example, you can create a semantic segmentation network by combining a pretrained network with a decoder subnetwork.

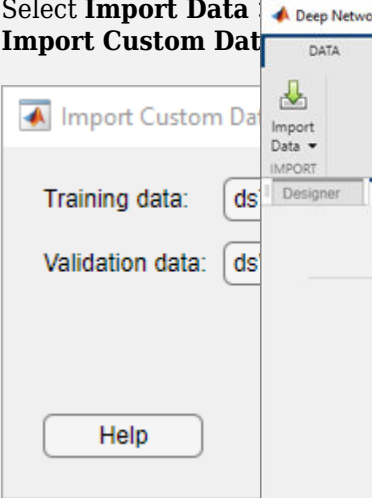
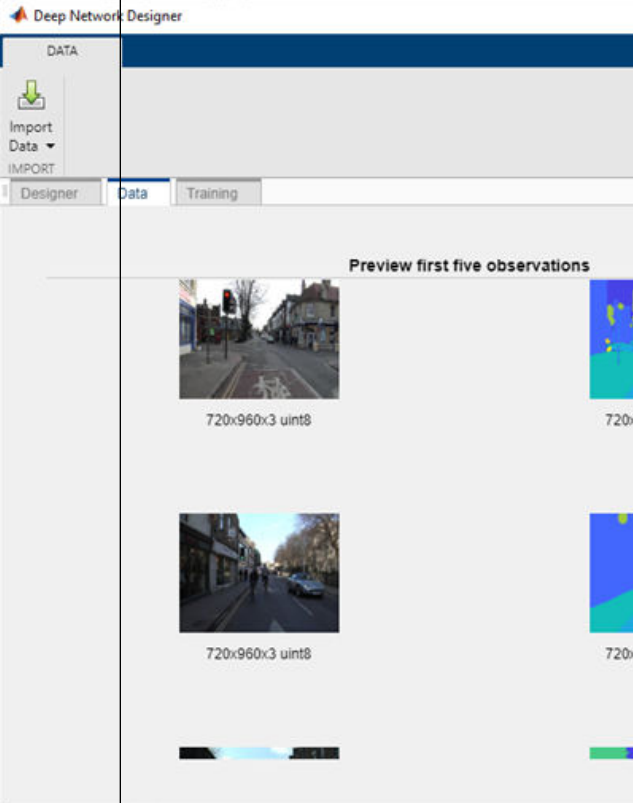
To view or edit the custom layer class definition, select the layer, and then click **Edit Layer Code**. For more information, see “View Autogenerated Custom Layers Using Deep Network Designer”.

### 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 subfolders containing images of a class. The labels are sourced from subfolders	<p>Select <b>Import Data</b> &gt; <b>Import Image Classification Data</b>.</p> 	
		<p>You can select augmentation options and specify the validation data in the Import Image Data dialog box. For more information, see “Import Data into Deep Network Designer”.</p>	

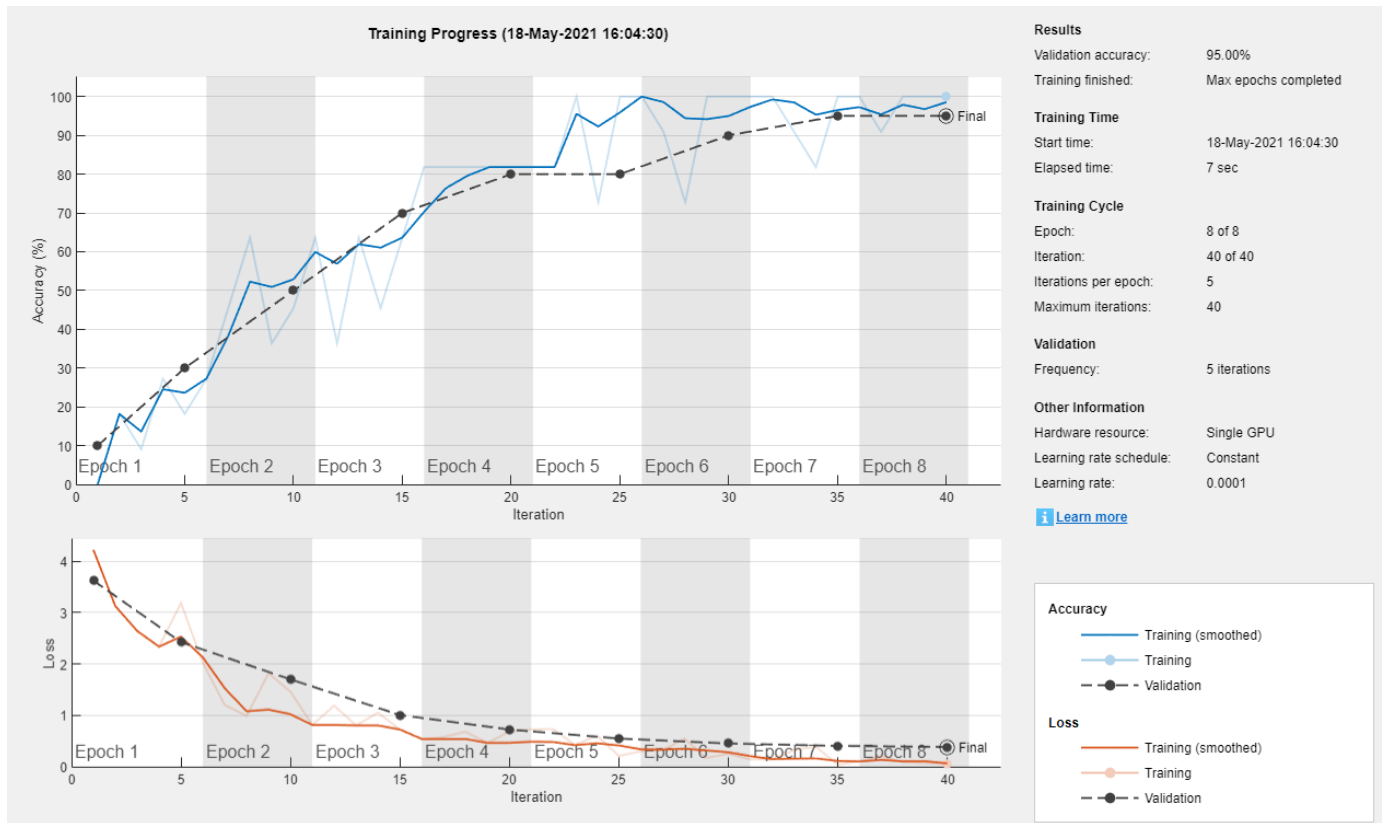
Task	Data Type	Data Import Method	Example Visualization
<p>Other extended workflows (such as numeric feature input, out-of-memory data, image processing, and audio and speech processing)</p>	<p>Datastore. For other extended workflows, use a suitable datastore object. For example, <code>AugmentedImageDatastore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

### Train Network

Train deep neural networks using Deep Network Designer.

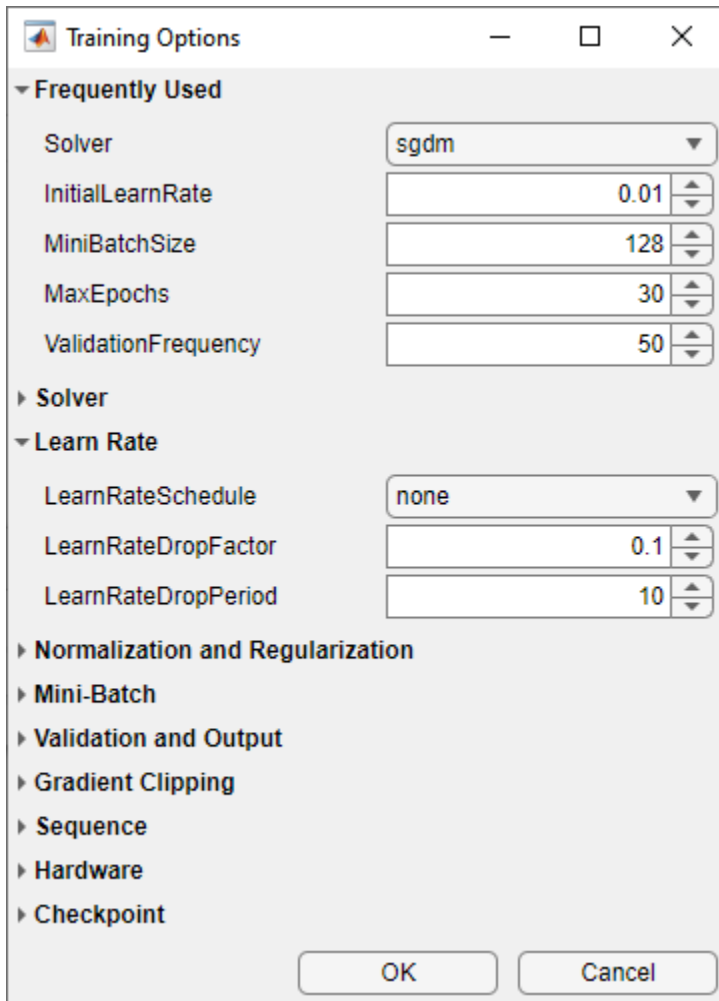
Using Deep Network Designer, you can train a network using image data or any datastore object that works with the `trainNetwork` function. For example, you can train a semantic segmentation network or a multi-input network using a `CombinedDatastore` object. For more information about importing data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

For an example showing how to export a network from Deep Network Designer to Simulink, see “Export Image Classification Network from Deep Network Designer to Simulink”.

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 Deep Network Designer 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 “Generate Experiment Using Deep Network Designer”.

- “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”
- “Transfer Learning with Pretrained Audio Networks in Deep Network Designer”

- “Import Custom Layer into Deep Network Designer”
- “Generate MATLAB Code from Deep Network Designer”
- “Export Image Classification Network from Deep Network Designer to Simulink”
- “Generate Experiment Using Deep Network Designer”
- “View Autogenerated Custom Layers Using Deep Network Designer”
- “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**. You can use Deep Network Designer to create experiments suitable for Experiment Manager.

## Version History

**Introduced in R2018b**

## 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”  
 “Transfer Learning with Pretrained Audio Networks in Deep Network Designer”  
 “Import Custom Layer into Deep Network Designer”  
 “Generate MATLAB Code from Deep Network Designer”  
 “Export Image Classification Network from Deep Network Designer to Simulink”

“Generate Experiment Using Deep Network Designer”  
“View Autogenerated Custom Layers Using Deep Network Designer”  
“List of Deep Learning Layers”



# Deep Network Quantizer

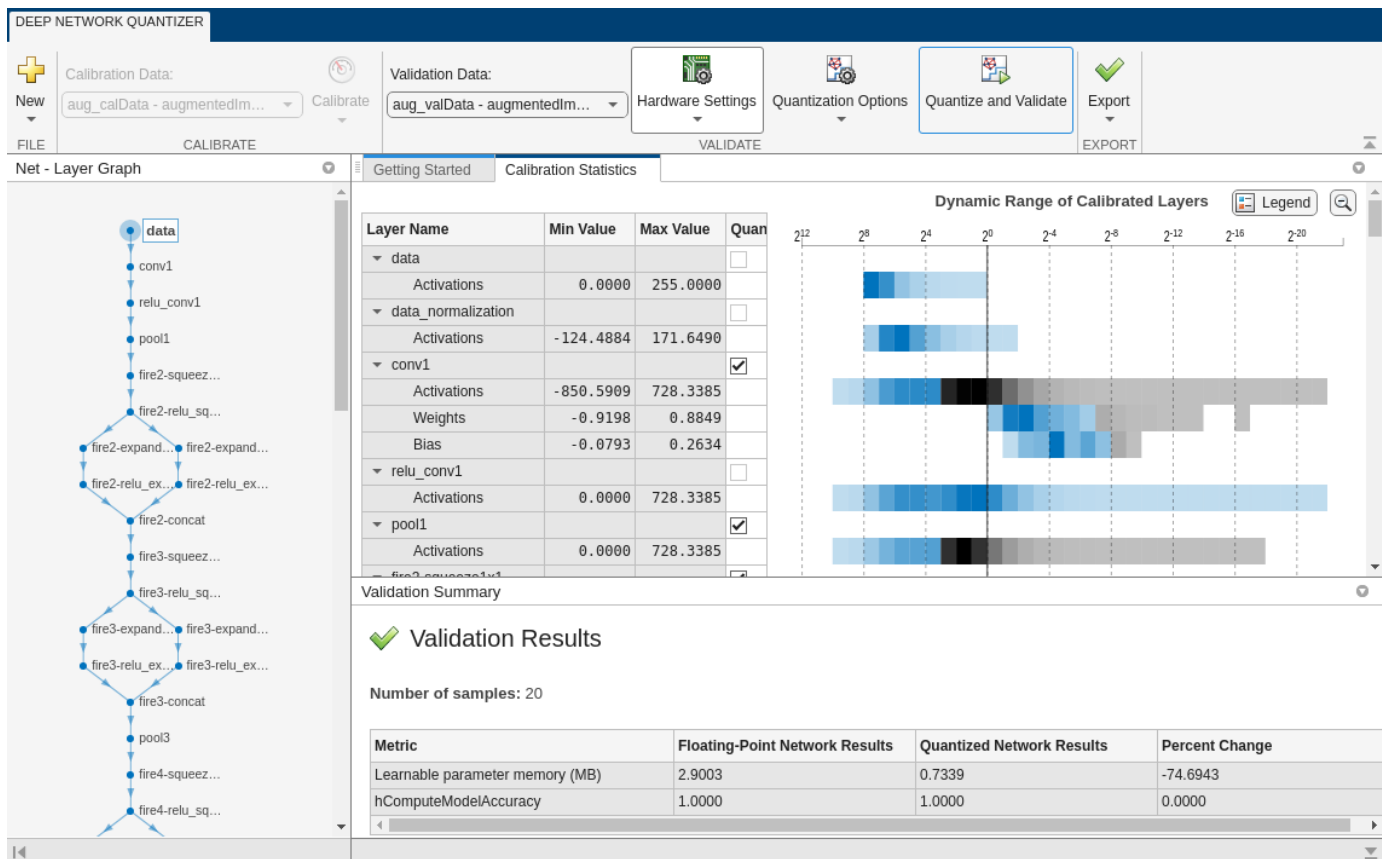
Quantize a deep neural network to 8-bit scaled integer data types

## 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™.
- Generate a simulatable quantized network that you can explore in MATLAB without generating code or deploying to hardware.

This app requires Deep Learning Toolbox Model Quantization Library. To learn about the products required to quantize a deep neural network, 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 squeezenetmerch
net
```

```
net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
  Connections: [75x2 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);
```

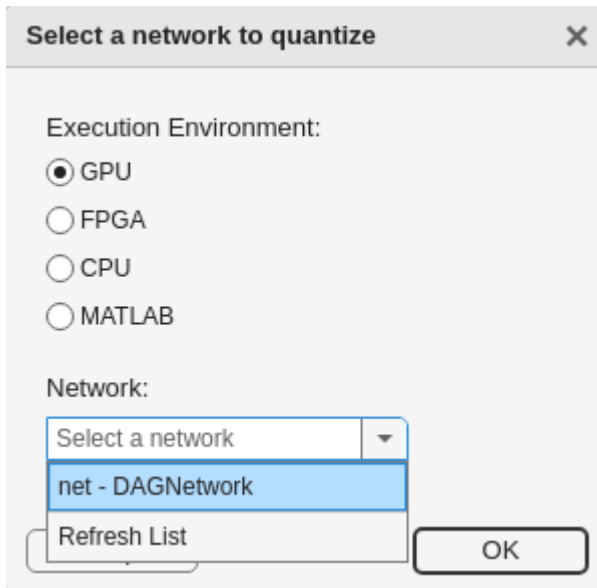
At the MATLAB command prompt, open the app.

```
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.

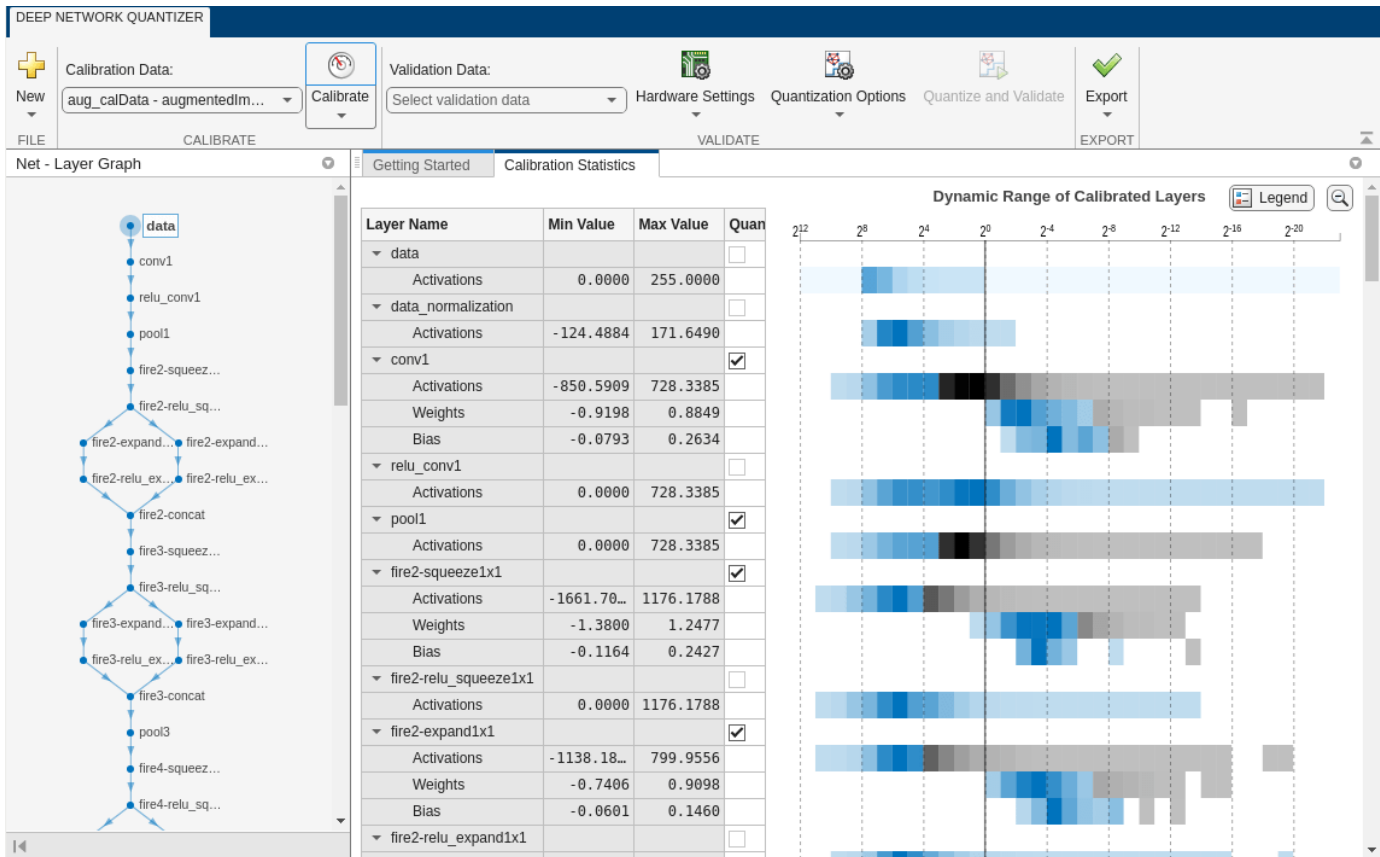


The app displays the layer graph of the selected network.

In the **Calibrate** section of the toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data, `aug_calData`. Select **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 quantized remain in single-precision after quantization.

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

In the **Validate** section of the toolbar, under **Quantization Options**, select the **Default** metric function and **MinMax** exponent scheme. Select **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)

The screenshot displays the Deep Network Quantizer interface. On the left is a 'Net - Layer Graph' showing a sequence of layers: data, conv1, relu\_conv1, pool1, fire2-squeeze, fire2-relu\_sq, fire2-expand, fire2-relu\_ex, fire2-concat, fire3-squeeze, fire3-relu\_sq, fire3-expand, fire3-relu\_ex, fire3-concat, pool3, fire4-squeeze, and fire4-relu\_sq. The center panel shows 'Calibration Statistics' for various layers:

Layer Name	Min Value	Max Value	Quan
data			<input type="checkbox"/>
Activations	0.0000	255.0000	
data_normalization			<input type="checkbox"/>
Activations	-124.4884	171.6490	
conv1			<input checked="" type="checkbox"/>
Activations	-850.5909	728.3385	
Weights	-0.9198	0.8849	
Bias	-0.0793	0.2634	
relu_conv1			<input type="checkbox"/>
Activations	0.0000	728.3385	
pool1			<input checked="" type="checkbox"/>
Activations	0.0000	728.3385	

Below the table is a 'Validation Summary' section with a green checkmark and the text 'Validation Results'. It indicates 'Number of samples: 20' and includes a table of metrics:

Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
Learnable parameter memory (MB)	2.9003	0.7339	-74.6943
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 = hComputeModelAccuracy(predictionScores, net, datastore)
%% Computes model-level accuracy statistics

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

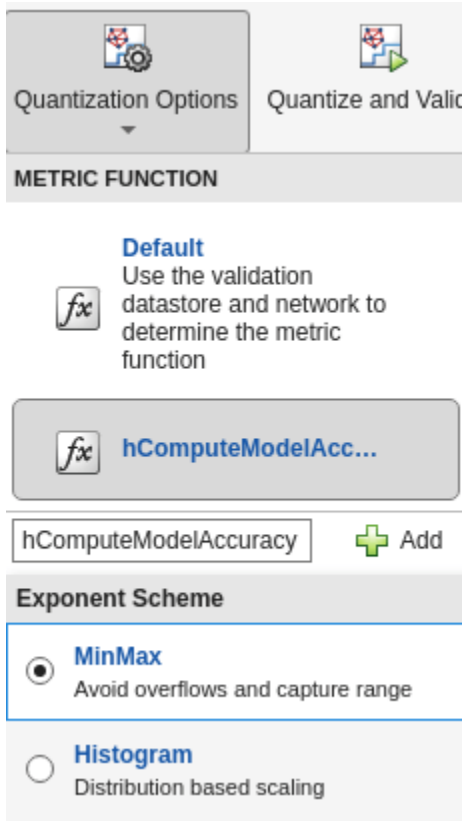
% Compare with predicted label with actual ground truth
predictionError = {};
for idx=1:numel(groundTruth)
    [~, idy] = max(predictionScores(idx,:));
    yActual = net.Layers(end).Classes(idy);
    predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
end

% 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

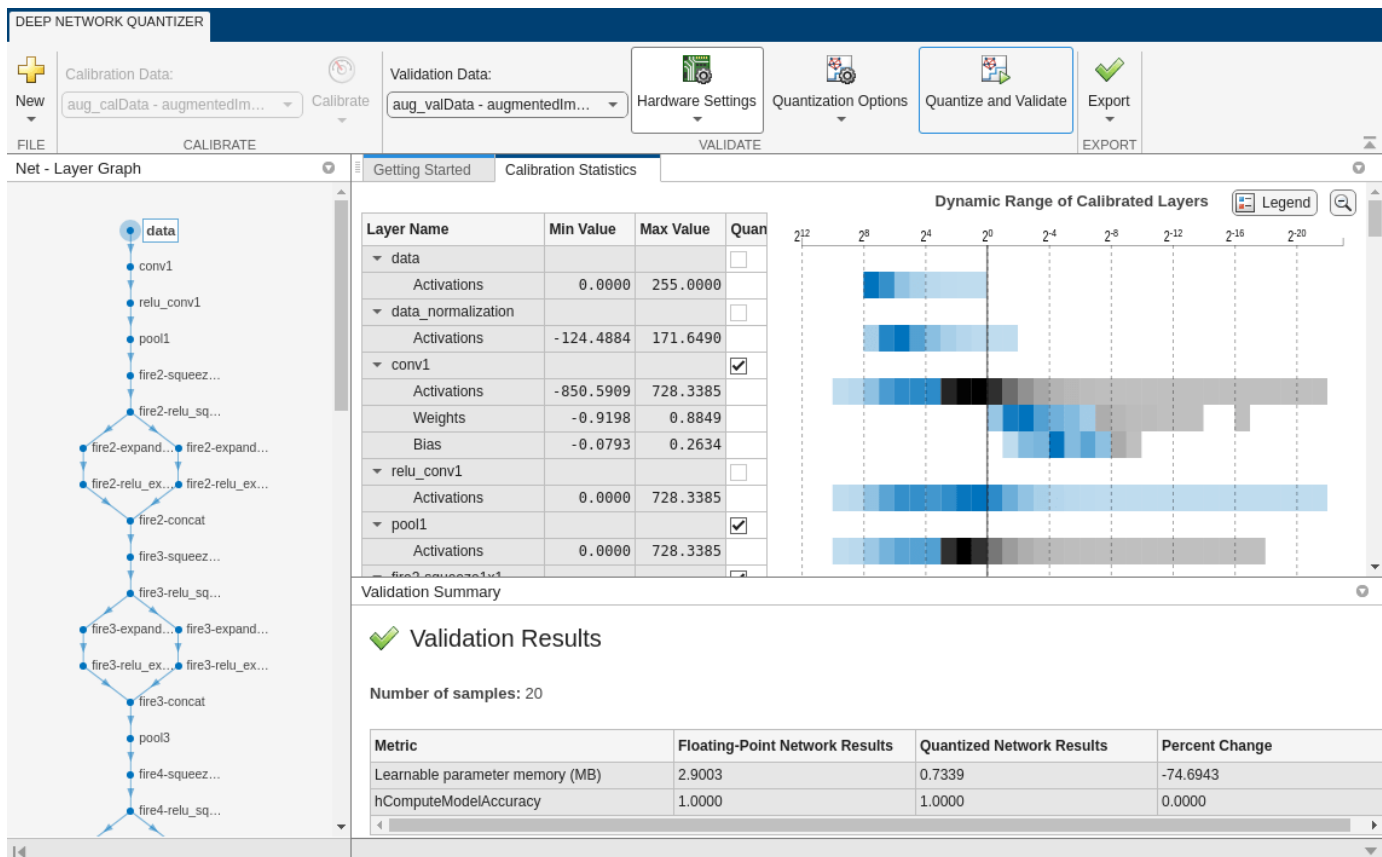
hComputeModelAccuracy to the list of metric functions available in the app. Select hComputeModelAccuracy as the metric function to use.

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



Select **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.

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. You can also explore the effects of choosing a different exponent selection scheme for quantization in the **Quantization Options** menu. To see the effects of these changes, select **Quantize and Validate** again.

After calibrating the network, you can choose to export the quantized network or the `dlquantizer` object. Select the **Export** button. In the drop down, select from the following options:

- **Export Quantized Network** - Add the quantized network to the base workspace. This option exports a simulatable quantized network that you can explore in MATLAB without deploying to hardware.
- **Export Quantizer** - Add the `dlquantizer` object to the base workspace. You can save the `dlquantizer` object and use it for further exploration in the **Deep Network Quantizer** app or at the command line, or use it to generate code for your target hardware.
- **Generate Code** - Open the **GPU Coder** app and generate GPU code from the quantized neural network. Generating GPU code requires a GPU Coder™ license.



## Quantize a Network for CPU 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 CPU execution environment.

Load the network to quantize into the base workspace.

```
load squeezenetmerch
net
net =
  DAGNetwork with properties:
    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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);
```

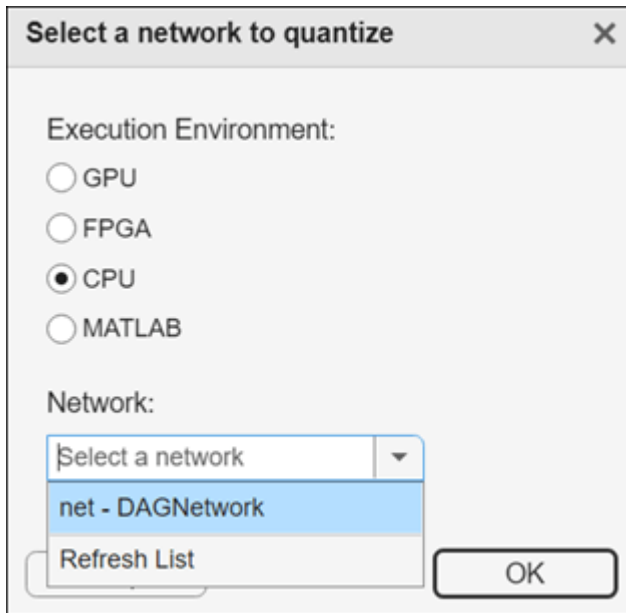
At the MATLAB command prompt, open the app.

```
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 CPU 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, `aug_calData`. Select **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".

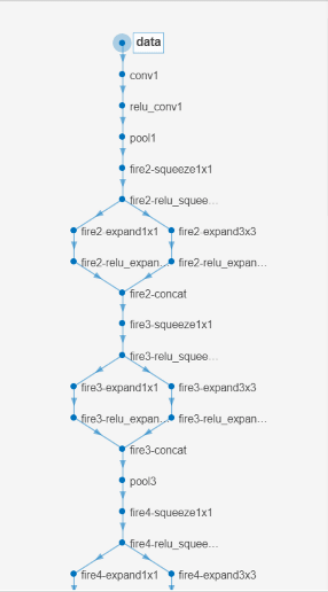
DEEP NETWORK QUANTIZER

Calibration Data:

Validation Data:

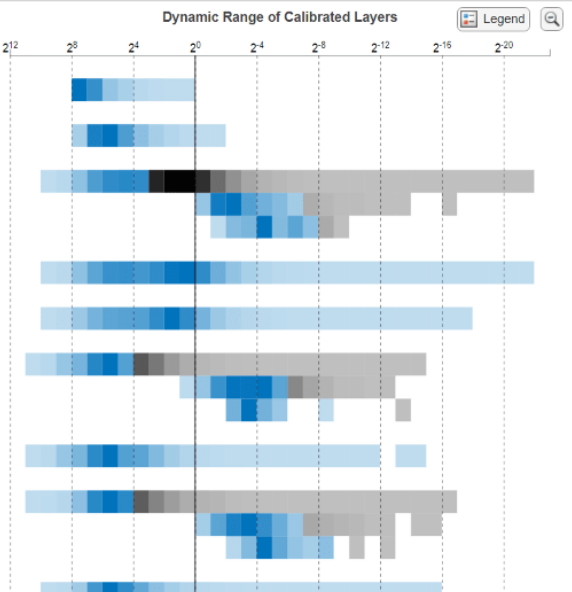
FILE CALIBRATE VALIDATE EXPORT

net - Layer Graph



Layer Name	Min Value	Max Value	Quantize Layer
data			
Activations	0.0000	255.0000	<input type="checkbox"/>
data_normalization			
Activations	-124.4884	171.6490	<input type="checkbox"/>
conv1			<input checked="" type="checkbox"/>
Activations	-850.5910	769.5881	
Weights	-0.9198	0.8849	
Bias	-0.0793	0.2634	
relu_conv1			<input type="checkbox"/>
Activations	0.0000	769.5881	
pool1			<input type="checkbox"/>
Activations	0.0000	769.5881	
fire2-squeeze1x1			<input checked="" type="checkbox"/>
Activations	-1662.7080	1176.1788	
Weights	-1.3800	1.2477	
Bias	-0.1164	0.2427	
fire2-relu_squeeze1x1			<input type="checkbox"/>
Activations	0.0000	1176.1788	
fire2-expand1x1			<input checked="" type="checkbox"/>
Activations	-1138.1814	799.9556	
Weights	-0.7406	0.9098	
Bias	-0.0601	0.1460	
fire2-relu_expand1x1			<input type="checkbox"/>
Activations	0.0000	799.9556	

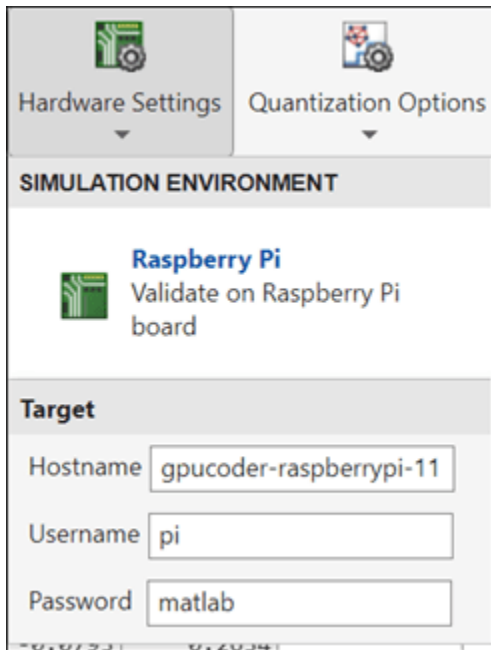
Dynamic Range of Calibrated Layers



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

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

In the **Validate** section of the toolbar, under **Hardware Settings**, select **Raspberry Pi** as the **Simulation Environment**. The app auto-populates the Target credentials from an existing connection or from the last successful connection. You can also use this option to create a new Raspberry Pi connection.



In the **Validate** section of the toolstrip, under **Quantization Options**, select the **Default** metric function and **MinMax** exponent scheme. Select **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)

The screenshot shows the Deep Network Quantizer application. The top toolbar includes buttons for 'Calibrate', 'Validate', 'Quantize and Validate', and 'Export'. The 'net - Layer Graph' on the left displays a neural network structure with layers like 'data', 'conv1', 'relu\_conv1', 'pool1', and 'conv1'. The 'Calibration Statistics' table is as follows:

Layer Name	Min Value	Max Value	Quantize La
data			
Activations	0.0000	255.0000	
data_normalization			
Activations	-124.4884	171.6490	
conv1			<input checked="" type="checkbox"/>
Activations	-850.5910	769.5881	
Weights	-0.9198	0.8849	
Bias	-0.0793	0.2634	
relu_conv1			
Activations	0.0000	769.5881	
pool1			
Activations	0.0000	769.5881	

The 'Validation Summary' section shows 'Validation Results' with a 'Number of samples: 20'. The results table is:

Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
Top-1 Accuracy	1.0000	1.0000	0.0000

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. You can also explore the effects of choosing a different exponent selection scheme for quantization in the **Quantization Options** menu. To see the effects of these changes, select **Quantize and Validate** again.

After calibrating the network, you can choose to export the quantized network or the `dlquantizer` object. Select the **Export** button. In the drop down, select from the following options:

- **Export Quantized Network** - Add the quantized network to the base workspace. This option exports a simulatable quantized network that you can explore in MATLAB without deploying to hardware.
- **Export Quantizer** - Add the `dlquantizer` object to the base workspace. You can save the `dlquantizer` object and use it for further exploration in the **Deep Network Quantizer** app or at the command line, or use it to generate code for your target hardware.
- **Generate Code** - Open the **MATLAB Coder** app and generate C++ code from the quantized neural network. Generating C++ code requires a MATLAB Coder™ license.

## 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 an FPGA target.

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

Load the pretrained network to quantize into the base workspace. 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/gpuocoder/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: [22x1 nnet.cnn.layer.Layer]
  InputNames: {'imageinput'}
  OutputNames: {'classoutput'}
```

Define calibration and validation data to use for quantization.

The **Deep Network Quantizer** 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 `imageDatastore` object, then split the data into calibration and validation data sets.

Expedite the calibration and validation process for this example by using a subset of the calibration and validation data. Store the new reduced calibration data set in `calData_concise` and the new reduced validation data set in `valData_concise`.

```
currentDir = pwd;
openExample('deeplearning_shared/QuantizeNetworkForFPGADeploymentExample')
unzip('logos_dataset.zip');

imds = imageDatastore(fullfile(currentDir,'logos_dataset'),...
    'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');

[calData,valData] = splitEachLabel(imds,0.7,'randomized');

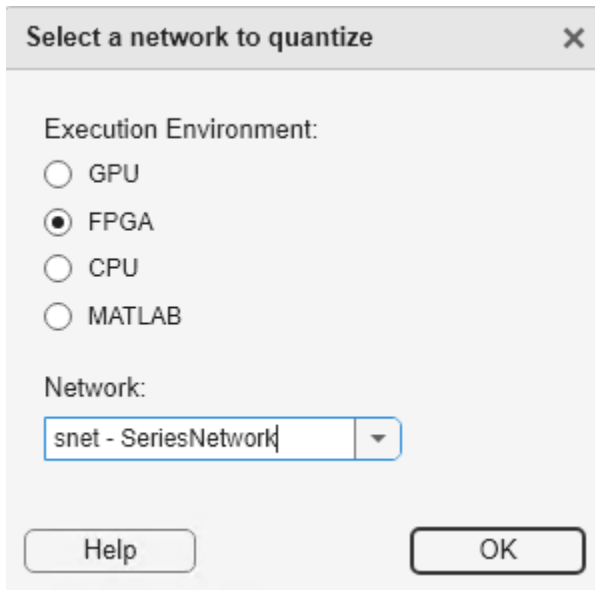
calData_concise = calData.subset(1:20);
valData_concise = valData.subset(1:6);
```

Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

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

Set the execution environment to **FPGA** and select `snet - SeriesNetwork` as the network to quantize.



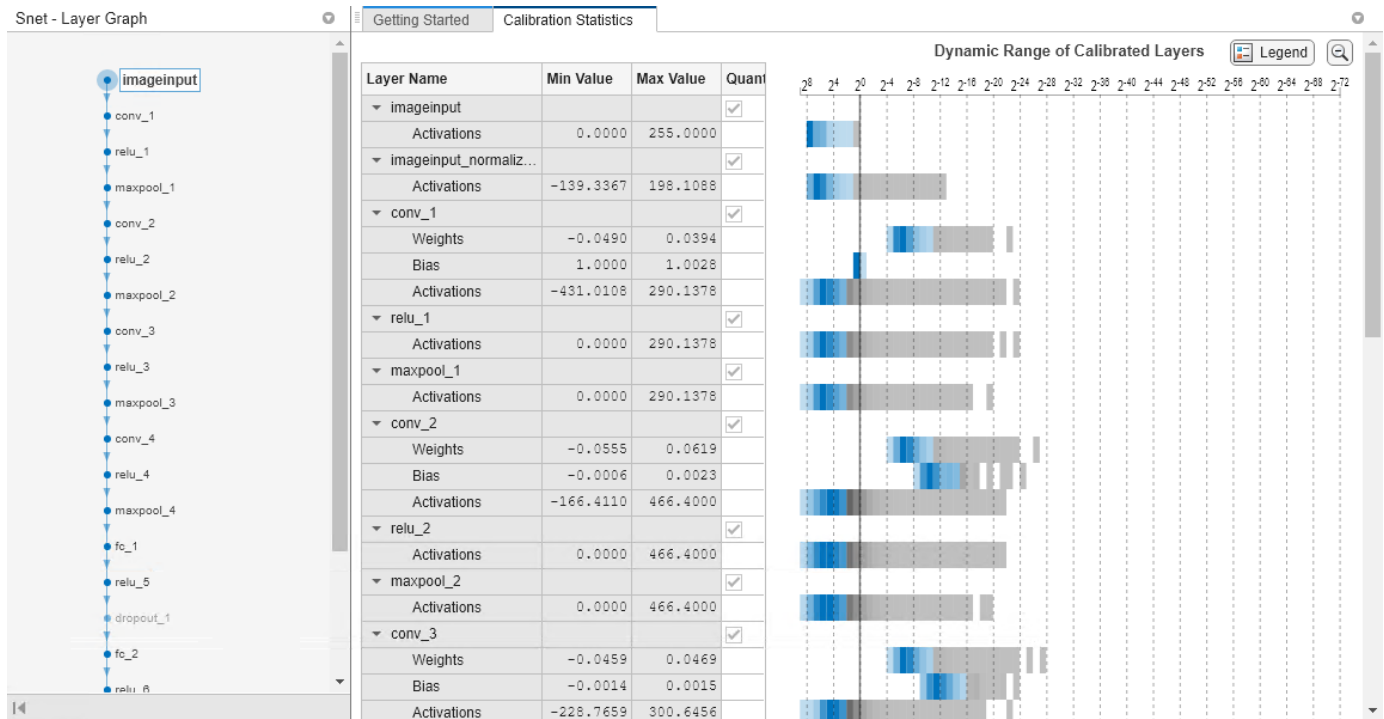
The app displays the layer graph of the selected network.

Under **Calibration Data**, select the `calData_concise - ImageDatastore` object from the base workspace containing the calibration data.

Click **Calibrate**. By default, the app uses the host GPU to collect calibration data, if one is available. Otherwise, the host CPU is used. You can use the **Calibrate** drop down menu to select the calibration environment.

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 recorded 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 Layer** column of the table, indicate whether to quantize the learnable parameters in the layer. Layers that are not quantized remain in single-precision.

Under **Validation Data**, select the valData\_concise - ImageDatastore object from the base workspace containing the validation data.

In the **Hardware Settings** section of the toolstrip, select the environment to use for validation of the quantized network. For more information on these options, see "Hardware Settings" on page 1-0 .

For this example, select **Xilinx ZC706 (zc706\_int8)** and **JTAG**.





Under **Quantization Options**, select the **Default** metric function and **MinMax** exponent scheme. For more information on these options, see “Quantization Options” on page 1-0 .

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 default metric function to use for the validation based on the type of network that is being quantized. For more information, see “Quantization Options” on page 1-0 .

When the validation is complete, the app displays the validation results.

## Validation Summary

 Validation Results

Number of samples: 6

Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
FramesPerSecond	0.8367	2.7262	225.8316
Number of Threads (Convolution)	4.0000	16.0000	300.0000
Number of Threads (Fully Connected)	4.0000	8.0000	100.0000
LUT Utilization (%)	63.1281	48.7790	-22.7300
BlockRAM Utilization (%)	99.9083	80.0000	-19.9265
DSP Utilization (%)	17.0000	31.2222	83.6601
Top-1 Accuracy	1.0000	1.0000	0.0000

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. For an example, see “Classify Images on FPGA Using Quantized Neural Network” (Deep Learning HDL Toolbox).

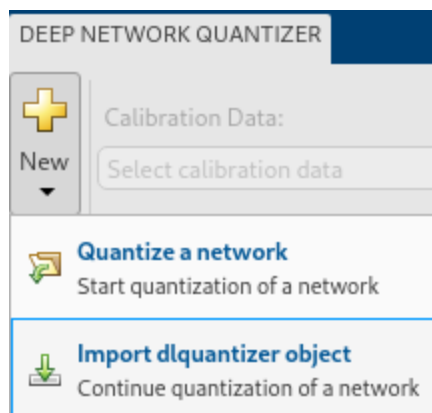
### Import a `dlquantizer` Object into the Deep Network Quantizer App

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.

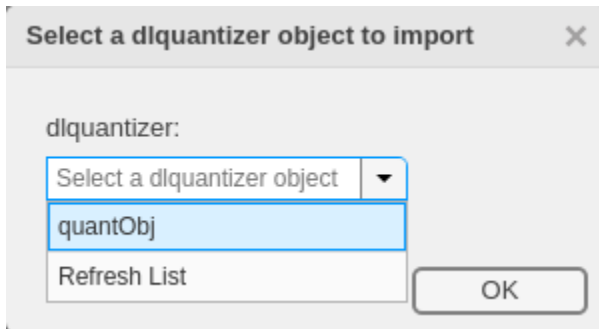
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

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

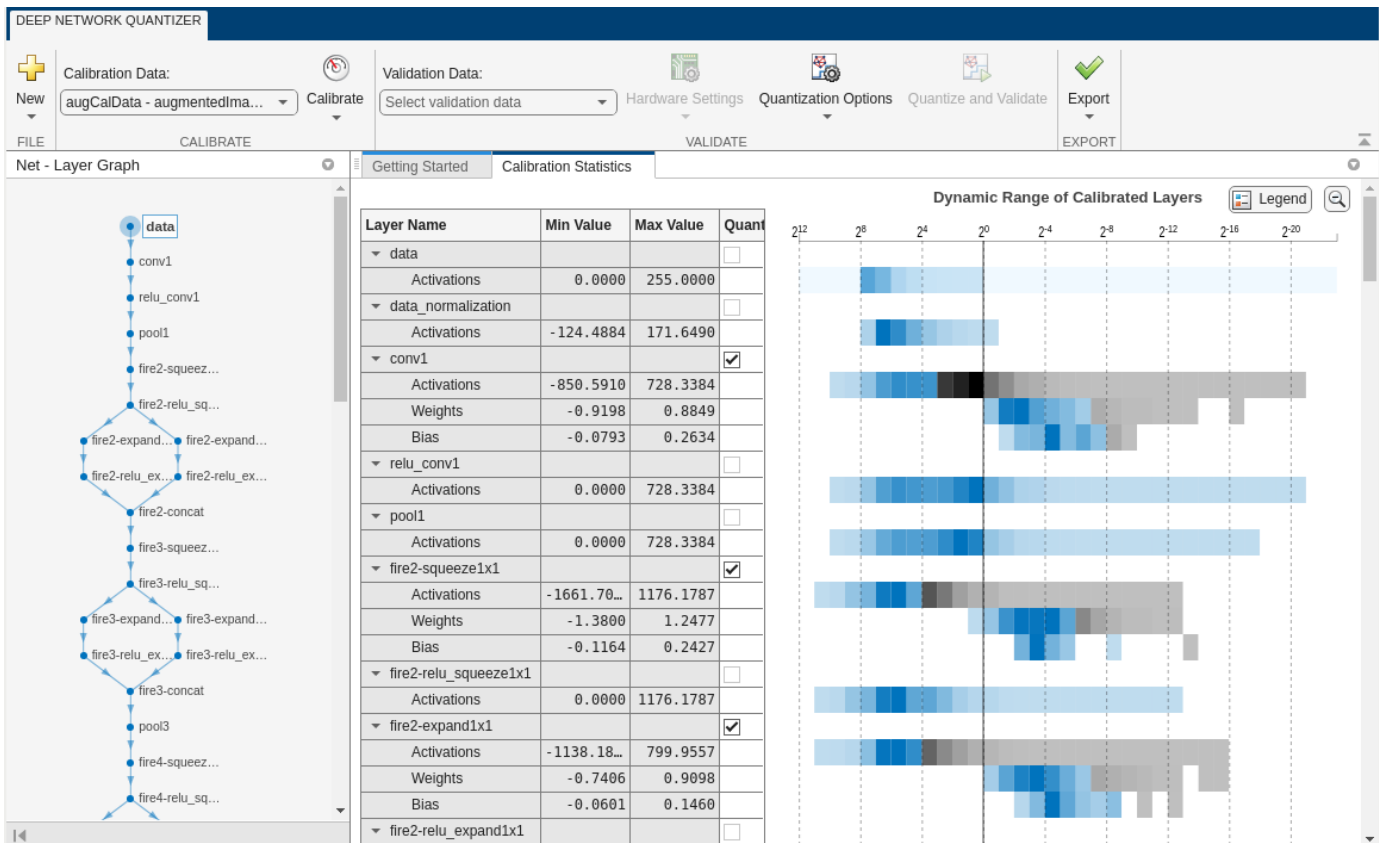


In the dialog, select the `dlquantizer` object to import from the base workspace. For this example, use `quantObj` that you create in the above example `Quantize a Neural Network for GPU Target`.



The app imports any data contained in the `dquantizer` 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 `dquantizer` 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”.



- “Quantization of Deep Neural Networks”
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code”

## Parameters

### Execution Environment — Execution Environment

GPU (default) | FPGA | CPU | MATLAB

When you select **New > Quantize a Network**, the app allows you to choose the execution environment for the quantized network. How the network is quantized depends on the choice of execution environment.

When you select the MATLAB execution environment, the app performs target-agnostic quantization of the neural network. This option does not require you to have target hardware in order to explore the quantized network in MATLAB.

### Hardware Settings — Hardware settings

simulation environment | target

Specify hardware settings based on your execution environment.

- **GPU Execution Environment**

Select from the following simulation environments:

Simulation Environment	Action
<b>GPU</b> Simulate on host GPU	Deploys the quantized network to the host GPU. Validates the quantized network by comparing performance to single-precision version of the network.
<b>MATLAB</b> Simulate in MATLAB	Simulates the quantized network in MATLAB. Validates the quantized network by comparing performance to single-precision version of the network.

- **FPGA Execution Environment**

Select from the following simulation environments:

Simulation Environment	Action
<b>Co-Simulation</b> Simulate with dlhdl.Simulator	Simulates the quantized network in MATLAB using <code>dlhdl.Simulator</code> . Validates the quantized network by comparing performance to single-precision version of the network.
<b>Intel Arria 10 SoC</b> arria10soc_int8	Deploys the quantized network to an Intel® Arria® 10 SoC board by using the <code>arria10soc_int8</code> bitstream. Validates the quantized network by comparing performance to single-precision version of the network.

Simulation Environment	Action
<b>Xilinx ZCU102</b> 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.
<b>Xilinx ZC706</b> 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**, **Xilinx ZCU102**, or **Xilinx ZC706** option, additionally select the interface to use to deploy and validate the quantized network.

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

- **CPU Execution Environment**

Select from the following simulation environments:

Simulation Environment	Action
<b>Raspberry Pi</b>	Deploys the quantized network to the Raspberry Pi board. Validates the quantized network by comparing performance to single-precision version of the network.

When you select the **Raspberry Pi** option, additionally specify the following details for the raspi connection.

Target Option	Description
Hostname	Hostname of the board, specified as a string.
Username	Linux® username, specified as a string.
Password	Linux user password, specified as a string.

**Quantization Options** — Options for quantization and validation  
metric function | exponent scheme

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	<b>Top-1 Accuracy</b> - Accuracy of the network
Object Detection	<b>Average Precision</b> - Average precision over all detection results. See <code>evaluateDetectionPrecision</code> .
Regression	<b>MSE</b> - Mean squared error of the network
Semantic Segmentation	<b>WeightedIOU</b> - Average IoU of each class, weighted by the number of pixels in that class. See <code>evaluateSemanticSegmentation</code> .

You can also specify a custom metric function to use for validation.

You can select the exponent selection scheme to use for quantization of the network:

- **MinMax** — (default) Evaluate the exponent based on the range information in the calibration statistics and avoid overflows.
- **Histogram** — Distribution-based scaling which evaluates the exponent to best fit the calibration data.

**Export** — Options for exporting quantized network

`Export Quantized Network` | `Export Quantizer` | `Generate Code`

- **Export Quantized Network** — After calibrating the network, quantize and add the quantized network to the base workspace. This option exports a simulatable quantized network, `quantizedNet`, that you can explore in MATLAB without deploying to hardware. This option is equivalent to using `quantize` at the command line.

Code generation is not supported for the exported quantized network, `quantizedNet`.

- **Export Quantizer** — Add the `dlquantizer` object to the base workspace. You can save the `dlquantizer` object and use it for further exploration in the **Deep Network Quantizer** app or at the command line, or use it to generate code for your target hardware.
- **Generate Code**

Execution Environment	Code Generation
<b>GPU</b>	Open the GPU Coder app and generate GPU code from the quantized and validated neural network. Generating GPU code requires a GPU Coder license.
<b>CPU</b>	Open the MATLAB Coder app and generate C++ code from the quantized and validated neural network. Generating C++ code requires a MATLAB Coder license.

## Version History

Introduced in R2020a

**R2022b: Calibrate on host GPU or host CPU**

You can now choose whether to calibrate your network using the host GPU or host CPU. By default, the `calibrate` function and the **Deep Network Quantizer** app will calibrate on the host GPU if one is available.

In previous versions, it was required that the execution environment be the same as the instrumentation environment used for the calibration step of quantization.

### **R2022b: dlnetwork support**

The **Deep Network Quantizer** app now supports calibration and validation for `dlnetwork` objects.

### **R2022a: Validate the performance of quantized network for CPU target**

The **Deep Network Quantizer** app now supports the quantization and validation workflow for CPU targets.

### **R2022a: Quantize neural networks without a specific target**

Specify `MATLAB` as the `Execution Environment` to quantize your neural networks without generating code or committing to a specific target for code deployment. This can be useful if you:

- Do not have access to your target hardware.
- Want to inspect your quantized network without generating code.

Your quantized network implements `int8` data instead of `single` data. It keeps the same layers and connections as the original network, and it has the same inference behavior as it would when running on hardware.

Once you have quantized your network, you can use the `quantizationDetails` function to inspect your quantized network. Additionally, you also have the option to deploy the code to a GPU target.

### **R2023a: Specify Raspberry Pi connection in Hardware Settings**

*Behavior changed in R2023a*

You can now validate the quantized network on a Raspberry Pi® board by selecting **Raspberry Pi** in `Hardware Settings` and specifying target credentials for a `raspi` object in the validation step.

## **See Also**

### **Functions**

`calibrate` | `quantize` | `validate` | `dlquantizer` | `dlquantizationOptions` | `quantizationDetails` | `estimateNetworkMetrics`

### **Topics**

“Quantization of Deep Neural Networks”

“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”

# 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. The `trainNetwork` function requires Deep Learning Toolbox.
- 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 and regression, sequence classification, audio classification, semantic segmentation, and custom training loops.

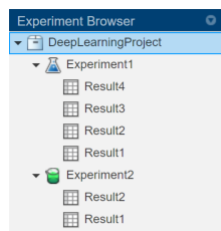
Experiment Manager provides visualization tools such as training plots and confusion matrices, filters to refine your experiment results, and annotations to record your observations. To improve reproducibility, every time that you run an experiment, Experiment Manager stores a copy of the experiment 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 projects.



- 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.

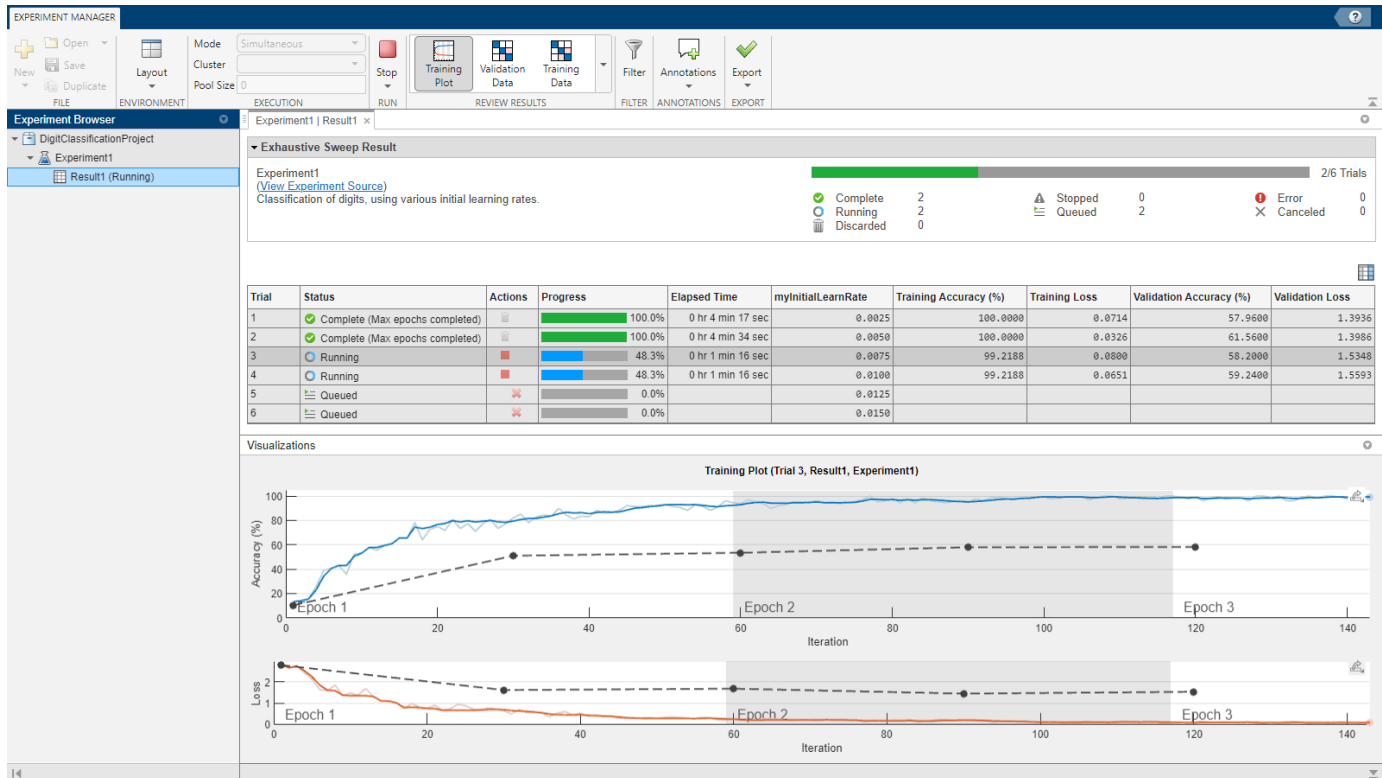
By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can configure your experiment to run multiple trials at the same time or to run a single trial at a time on multiple GPUs, on a cluster, or in the cloud. If you have MATLAB Parallel Server™, you can also offload experiments as batch jobs in a remote cluster so that you can continue working or close your MATLAB session during training. For more information, see “Use Experiment Manager to Train Networks in Parallel” and “Offload Experiments as Batch Jobs to Cluster”.

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

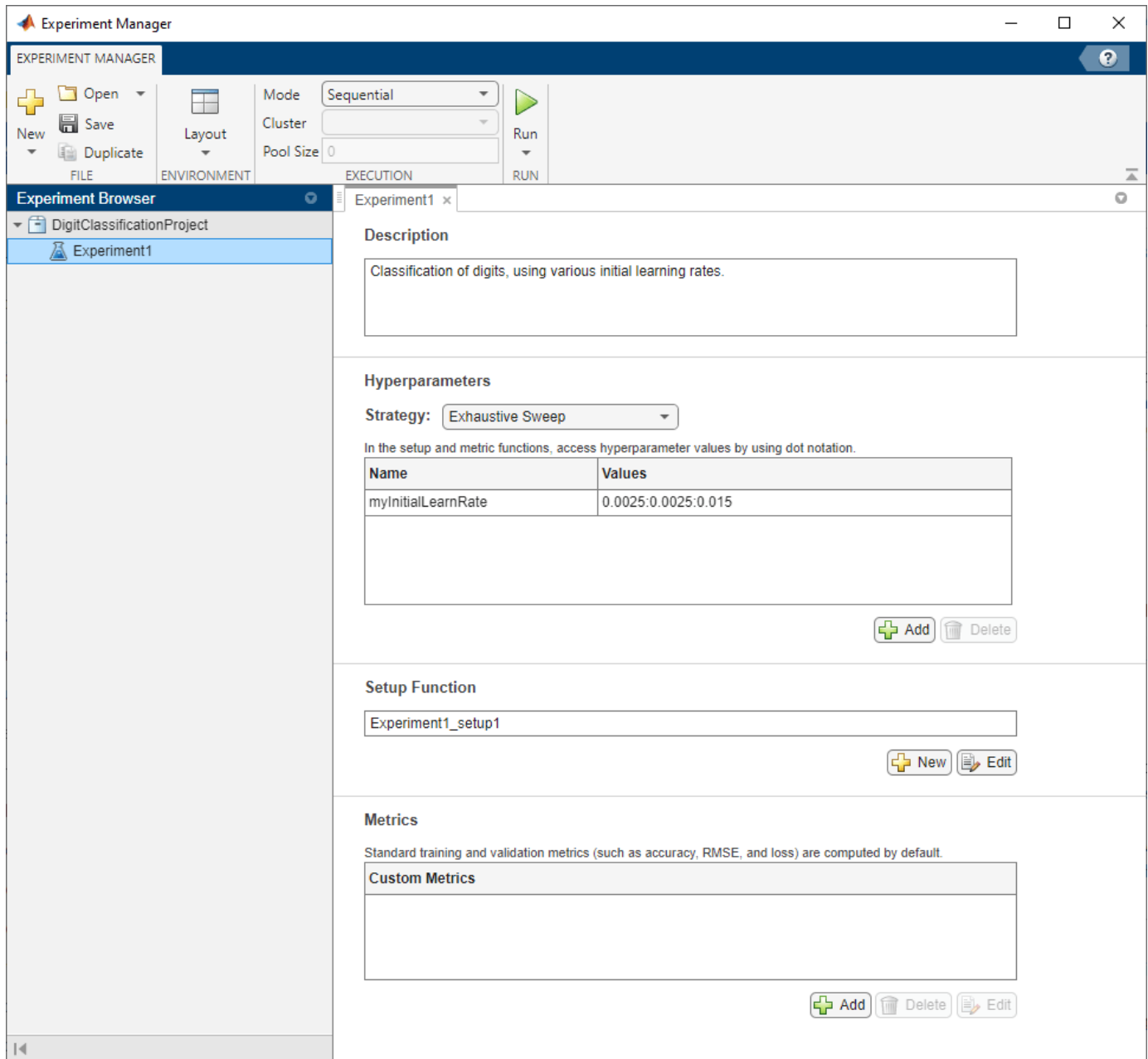
- MATLAB Toolstrip: On the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon.
- MATLAB command prompt: Enter `experimentManager`.

## Examples

### Image Classification by Sweeping Hyperparameters

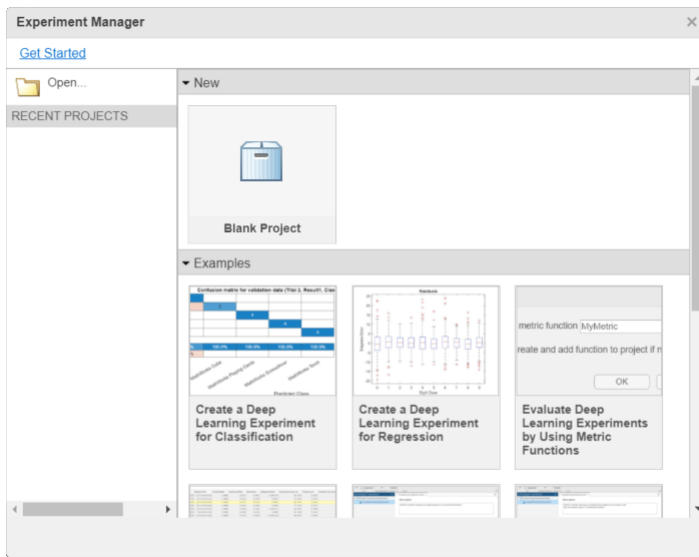
This example shows how to use the experiment template for image classification by sweeping hyperparameters. With this template, you can quickly set up a built-in training experiment that uses the `trainNetwork` function. The `trainNetwork` function requires Deep Learning Toolbox.

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 `Experiment1`.

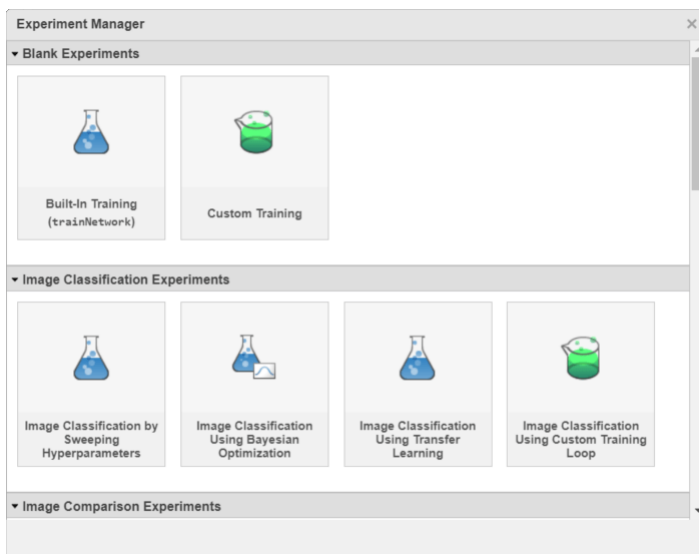


Alternatively, you can configure the experiment yourself by following these steps.

1. Open Experiment Manager. If a project is already open in MATLAB, Experiment Manager prompts you to load the current project. Otherwise, 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 and regression, sequence classification, audio 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**.


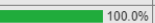
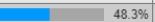
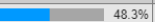
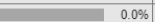
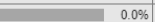
7. Under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. The setup function specifies the training data, network architecture, and training options for the experiment. In this experiment, the setup function has these sections:

- **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.

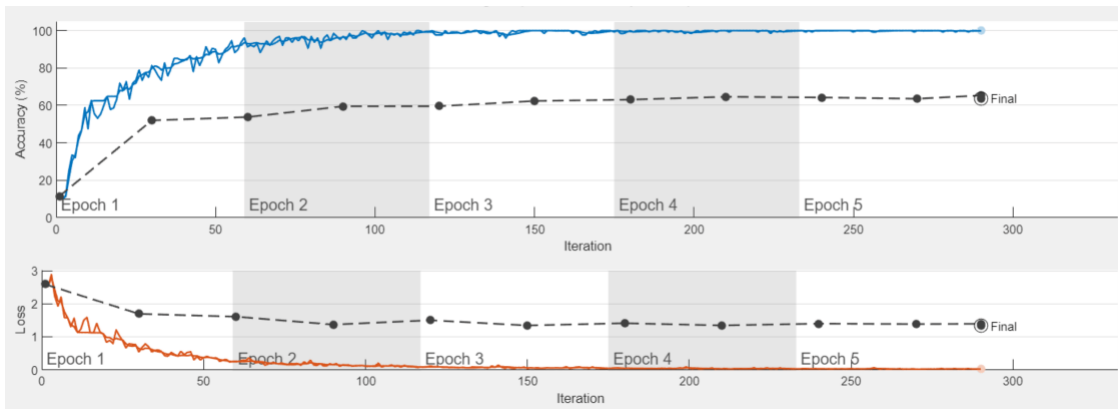
When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses one of the learning rates specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time or offload your experiment as a batch job in a cluster:

- To run one trial of the experiment at a time, on the **Experiment Manager** toolstrip, under **Mode**, select **Sequential** and click **Run**.
- To run multiple trials at the same time, under **Mode**, select **Simultaneous** and click **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then runs as many simultaneous trials as there are workers in your parallel pool. 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 Computing Requirements” (Parallel Computing Toolbox).
- To offload the experiment as a batch job, under **Mode**, select **Batch Sequential** or **Batch Simultaneous**, specify your **Cluster** and **Pool Size**, and click **Run**. For more information, see “Offload Experiments as Batch Jobs to Cluster”.

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

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete (Max epochs completed)		 100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.9600	1.3936
2	Complete (Max epochs completed)		 100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.5600	1.3986
3	Running		 48.3%	0 hr 1 min 16 sec	0.0075	99.2188	0.0000	58.2000	1.5348
4	Running		 48.3%	0 hr 1 min 16 sec	0.0100	99.2188	0.0651	59.2400	1.5593
5	Queued		 0.0%		0.0125				
6	Queued		 0.0%		0.0150				

To display the training plot and track the progress of each trial while the experiment is running, under **Review Results**, click **Training Plot**. You can also monitor the training progress in the MATLAB Command Window.



To display the confusion matrix for the validation data in each completed trial, under **Review Results**, click **Validation Data**.

0	169	10	11	3	13	8	13	4	2	17	67.6%	32.4%
1		224	3	7	4	3		9			89.6%	10.4%
2	8	16	133	36	8		7	15	27		53.2%	46.8%
3				170	20	10		4	46		68.0%	32.0%
4		3		1	208		4		34		83.2%	16.8%
5		22	5	45		127	19		22	10	50.8%	49.2%
6	3	73	10	9	21	1	119		14		47.6%	52.4%
7		13	5			1	2	217	11	1	86.8%	13.2%
8	3	27	33	42	10		26		89	20	35.6%	64.4%
9		29		2	39	1			40	139	55.6%	44.4%
	92.3%	53.7%	66.5%	54.0%	64.4%	84.1%	62.6%	87.1%	31.2%	74.3%		
	7.7%	46.3%	33.5%	46.0%	35.6%	15.9%	37.4%	12.9%	68.8%	25.7%		
	0	1	2	3	4	5	6	7	8	9		

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 1-57.

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 save the contents of the results table as a table array in the MATLAB workspace, select **Export > Results Table**.

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. The `trainNetwork` function requires Deep Learning Toolbox.

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 **Experiment1**.

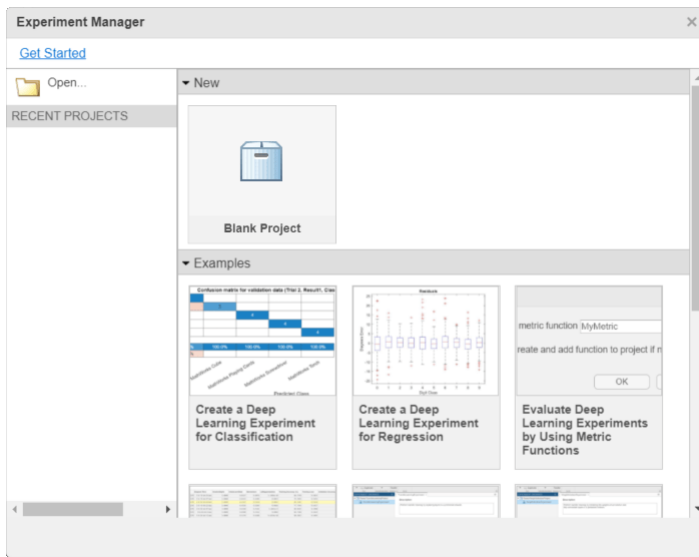
The screenshot shows the Experiment Manager window with the following configuration for Experiment1:

- Mode:** Sequential
- Cluster:** (empty)
- Pool Size:** 0
- Description:** Regression to predict angles of rotation of digits, using various initial learning rates.
- Hyperparameters:**
  - Strategy:** Exhaustive Sweep
  - Table:**

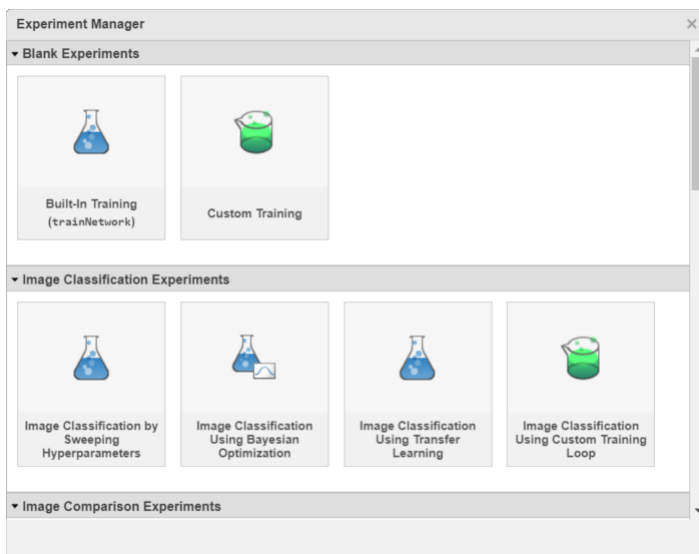
Name	Values
myInitialLearnRate	0.001:0.001:0.006
- Setup Function:** Experiment1\_setup1
- Metrics:** Custom Metrics (empty table)

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

1. Open Experiment Manager. If a project is already open in MATLAB, Experiment Manager prompts you to load the current project. Otherwise, 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 and regression, sequence classification, audio 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 these 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 or offload your experiment as a batch job in a cluster:

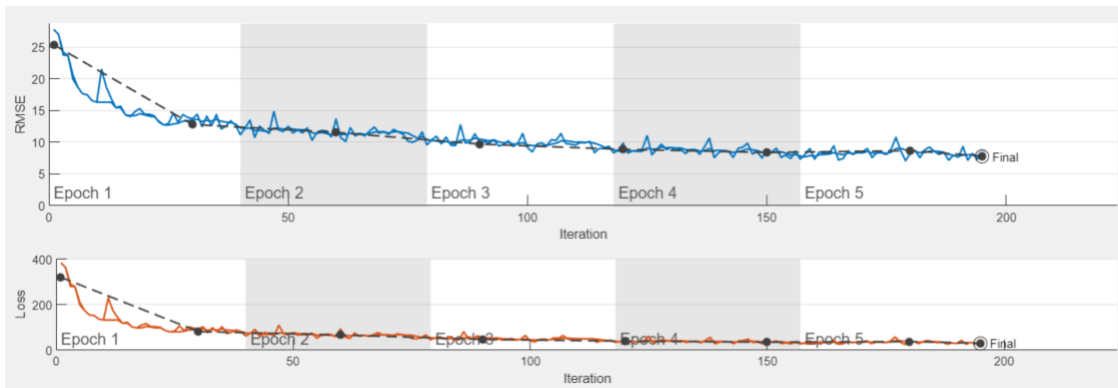
- To run one trial of the experiment at a time, on the **Experiment Manager** toolstrip, under **Mode**, select **Sequential** and click **Run**.
- To run multiple trials at the same time, under **Mode**, select **Simultaneous** and click **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then runs as many simultaneous trials as there are workers in your parallel pool. 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 Computing Requirements” (Parallel Computing Toolbox).
- To offload the experiment as a batch job, under **Mode**, select **Batch Sequential** or **Batch Simultaneous**, specify your **Cluster** and **Pool Size**, and click **Run**. For more information, see “Offload Experiments as Batch Jobs to Cluster”.

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

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training RMSE	Training Loss	Validation RM...	Validation Loss
1	Complete (Max epochs completed)		100.0%	0 hr 1 min 3 sec	0.0010	8.0616	32.4943	7.7715	30.1982
2	Complete (Max epochs completed)		100.0%	0 hr 1 min 5 sec	0.0020	7.9933	31.9462	7.9372	31.4995
3	Running		55.4%	0 hr 0 min 54 sec	0.0030	10.6922	57.1619	10.3233	53.2852
4	Running		55.9%	0 hr 0 min 54 sec	0.0040	9.0240	40.7159	10.0887	50.8906
5	Queued		0.0%		0.0050				
6	Queued		0.0%		0.0060				

To display the training plot and track the progress of each trial while the experiment is running, under **Review Results**, click **Training Plot**. 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 1-57.

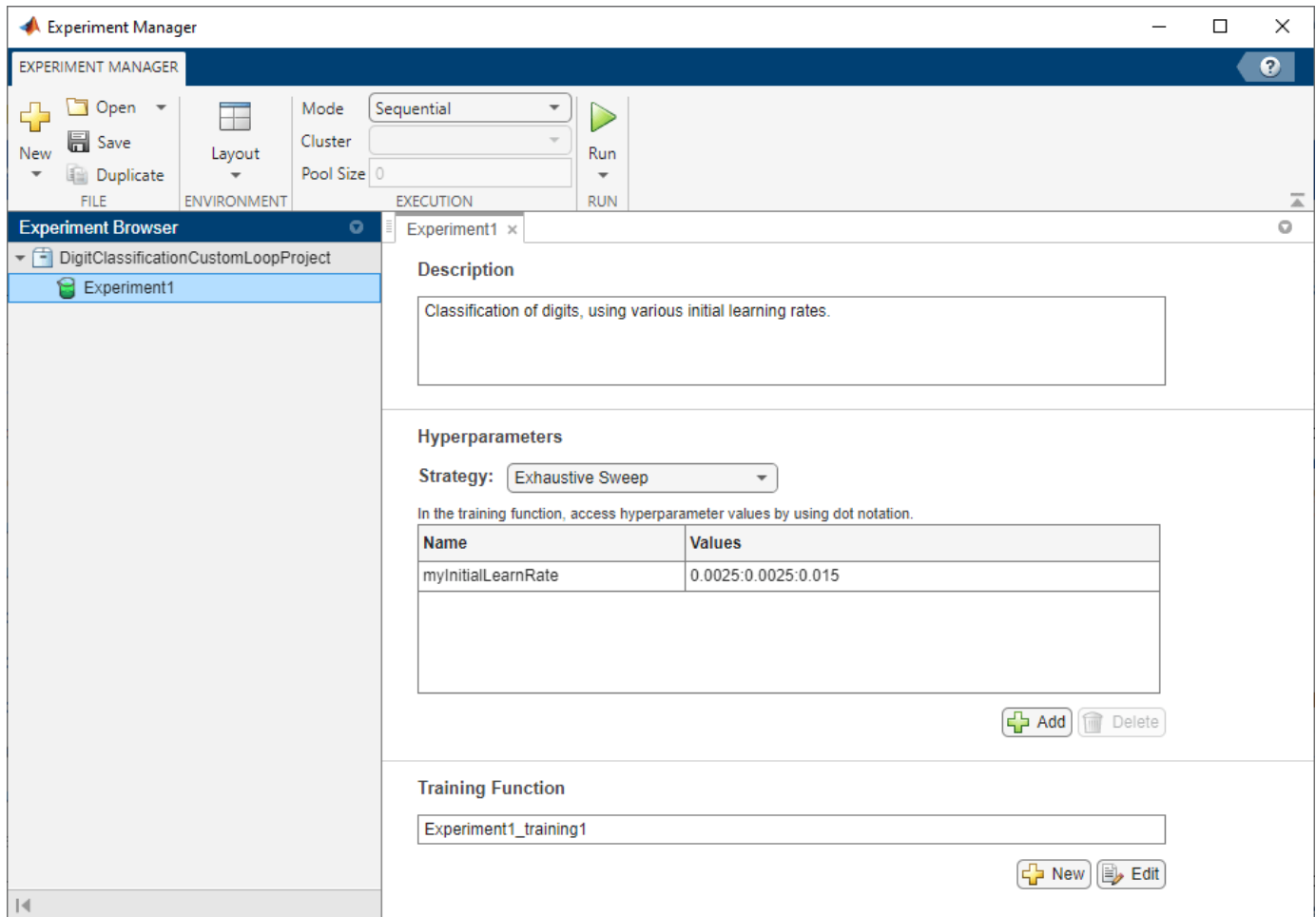
To test the performance of an individual trial, export the trained network or the training information for the trial. On the **Experiment Manager** toolstrip, select **Export > Trained Network** or **Export > Training Information**, respectively. For more information, see “net” on page 1-0 and “info” on page 1-0. To save the contents of the results table as a table array in the MATLAB workspace, select **Export > Results Table**.

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

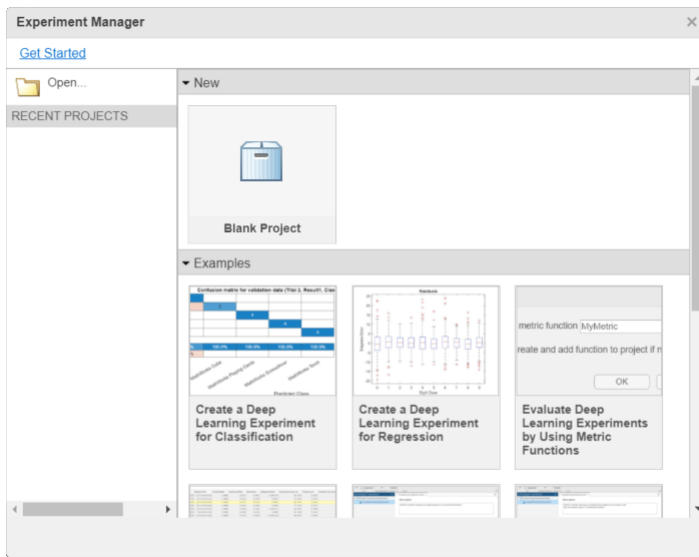
This example shows how to use the training experiment template for image classification using a custom training loop. With this template, you can quickly set up a custom training experiment.

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 **Experiment1**.

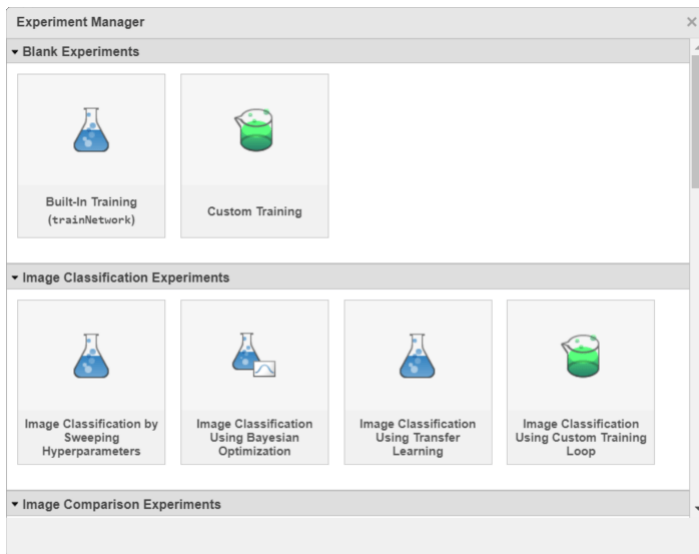


Alternatively, you can configure the experiment yourself by following these steps.

1. Open Experiment Manager. If a project is already open in MATLAB, Experiment Manager prompts you to load the current project. Otherwise, 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 and regression, sequence classification, audio 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`.


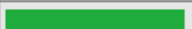
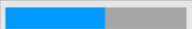
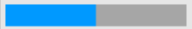
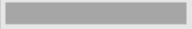
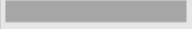
5. Under **Training Function**, click **Edit**. The training function opens in MATLAB Editor. The training function specifies the training data, network architecture, training options, and training procedure used by the experiment. In this experiment, the training function has these sections:

- **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`.

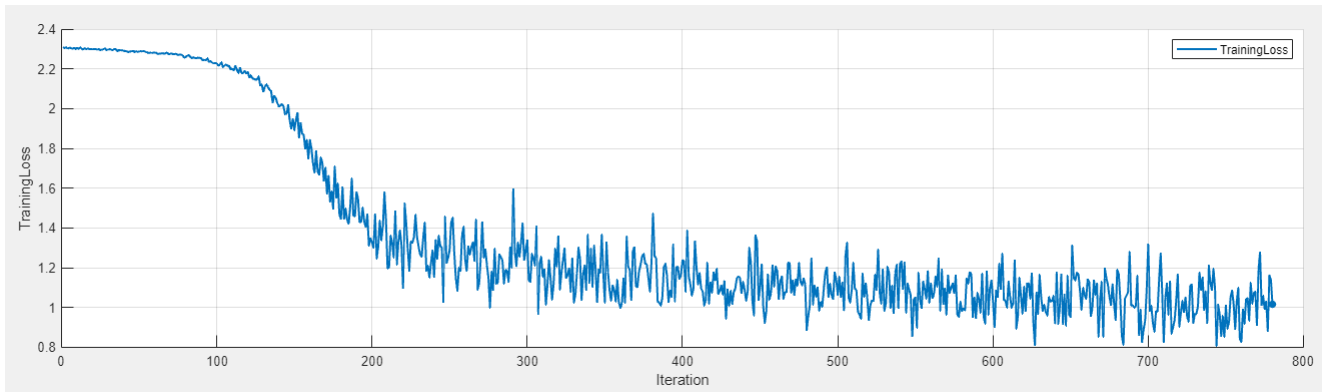
When you run the experiment, Experiment Manager trains the network defined by the training function six times. Each trial uses one of the learning rates specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time or offload your experiment as a batch job in a cluster:

- To run one trial of the experiment at a time, on the **Experiment Manager** toolstrip, under **Mode**, select **Sequential** and click **Run**.
- To run multiple trials at the same time, under **Mode**, select **Simultaneous** and click **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then runs as many simultaneous trials as there are workers in your parallel pool. 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 Computing Requirements” (Parallel Computing Toolbox).
- To offload the experiment as a batch job, under **Mode**, select **Batch Sequential** or **Batch Simultaneous**, specify your **Cluster** and **Pool Size**, and click **Run**. For more information, see “Offload Experiments as Batch Jobs to Cluster”.

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

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	TrainingLoss
1	✔ Complete	☰	 100.0%	0 hr 0 min 55 sec	0.0025	1.0179
2	✔ Complete	☰	 100.0%	0 hr 0 min 56 sec	0.0050	0.4266
3	🔄 Running	☒	 55.0%	0 hr 0 min 42 sec	0.0075	0.3835
4	🔄 Running	☒	 50.0%	0 hr 0 min 42 sec	0.0100	0.1488
5	⏸ Queued	☒	 0.0%		0.0125	
6	⏸ Queued	☒	 0.0%		0.0150	

To display the training plot and track the progress of each trial while the experiment is running, under **Review Results**, click **Training Plot**.



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 1-57.

To test the performance of an individual trial, export the training output for the trial. On the **Experiment Manager** toolstrip, select **Export > Training Output**. In this experiment, the training output is a structure that contains the values of the training loss and the trained network. To save the contents of the results table as a table array in the MATLAB workspace, select **Export > Results Table**.

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 Experiment Manager. Built-in training experiments rely on the `trainNetwork` function and support workflows such as image classification, image regression, sequence classification, and semantic segmentation. The `trainNetwork` function requires Deep Learning Toolbox.

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 and specify the hyperparameters to use for your experiment:

#### Exhaustive Sweep

To sweep through a range of hyperparameter values, set **Strategy** to Exhaustive Sweep. In the hyperparameter table, enter the names and values of the hyperparameters used in the experiment. Hyperparameter names must start with a letter, followed by letters, digits, or underscores.

Hyperparameter values must be scalars or vectors with numeric, logical, or string values, or cell arrays of character vectors. 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"]
- {'squeezeNet' 'googLeNet' 'resnet18'}

When you run the experiment, Experiment Manager trains the network using every combination of the hyperparameter values specified in the table.

### Bayesian Optimization

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:

- **Name** — Enter a valid hyperparameter name. Hyperparameter names must start with a letter, followed by letters, digits, or underscores.
- **Range** — For a real- or integer-valued hyperparameter, enter a two-element vector that gives the lower bound and upper bound of the hyperparameter. For a categorical hyperparameter, enter an array of strings or a cell array of character vectors that lists the possible values of the hyperparameter.
- **Type** — Select **real** for a real-valued hyperparameter, **integer** for an integer-valued hyperparameter, or **categorical** for a categorical hyperparameter.
- **Transform** — Select **none** to use no transform or **log** to use a logarithmic transform. When you select **log**, the hyperparameter values must be positive. With this setting, the Bayesian optimization algorithm models the hyperparameter on a logarithmic scale.

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.

Optionally, specify deterministic constraints, conditional constraints, and an acquisition function for the Bayesian optimization algorithm. Under **Bayesian Optimization Options**, click **Advanced Options** and specify:

- **Deterministic Constraints** - Enter the name of a deterministic constraint function. To run the Bayesian optimization algorithm without deterministic constraints, leave this option blank. For more information, see “Deterministic Constraints — XConstraintFcn” (Statistics and Machine Learning Toolbox).
- **Conditional Constraints** - Enter the name of a conditional constraint function. To run the Bayesian optimization algorithm without conditional constraints, leave this option blank. For more information, see “Conditional Constraints — ConditionalVariableFcn” (Statistics and Machine Learning Toolbox).
- **Acquisition Function Name** - Select an acquisition function from the list. The default value for this option is **expected-improvement-plus**. For more information, see “Acquisition Function Types” (Statistics and Machine Learning Toolbox).

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. Bayesian optimization requires Statistics and Machine Learning Toolbox.

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 <code>images</code> and the training options defined by <code>options</code> .	<code>function [images, layers, options] = Experiment_setup(p</code> ... <code>end</code>
Train a network using the images specified by <code>images</code> and responses specified by <code>responses</code> .	<code>function [images, responses, layers, options] = Experi</code> ... <code>end</code>
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 <code>sequences</code> .	<code>function [sequences, layers, options] = Experiment_setu</code> ... <code>end</code>
Train a network using the sequences specified by <code>sequences</code> and responses specified by <code>responses</code> .	<code>function [sequences, reponses, layers, options] = Experi</code> ... <code>end</code>
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 <code>features</code> .	<code>function [features, layers, options] = Experiment_setup</code> ... <code>end</code>
Train a network using the feature data specified by <code>features</code> and responses specified by <code>responses</code> .	<code>function [features, responses, layers, options] = Experi</code> ... <code>end</code>

**Tip** When writing your setup function, follow these guidelines:

- Load training and validation data from a location that is accessible to all your workers. For example, store your data outside the project and access the data by using an absolute path. Alternatively, create a datastore object that can access the data on another machine by setting up the `AlternateFileSystemRoots` property of the datastore. For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.
- For networks containing batch normalization layers, if the `BatchNormalizationStatistics` training option is `population`, Experiment Manager displays final validation metric values that are often different from the validation metrics evaluated during training. The difference in values is the result of additional operations performed after the network finishes training. For more information, see “Batch Normalization Layer” on page 1-249.
- The execution modes that you can use for your experiment depend on the settings you specify for the training options `ExecutionEnvironment` and `DispatchInBackground`.

Execution Mode	Valid Settings for ExecutionEnvironment	Valid Settings for DispatchInBackground
Sequential	"auto", "cpu", "gpu", "multi-gpu", "parallel"	true, false
Simultaneous	"auto", "cpu", "gpu"	false
Batch Sequential	"auto", "cpu", "gpu", "parallel"	true, false
Batch Simultaneous	"auto", "cpu", "gpu"	false

For more information, see “Use Experiment Manager to Train Networks in Parallel” and “Offload Experiments as Batch Jobs to Cluster”.

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 Experiment Manager. 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 and specify the hyperparameters to use for your experiment:



## Exhaustive Sweep

To sweep through a range of hyperparameter values, set **Strategy** to **Exhaustive Sweep**. In the hyperparameter table, enter the names and values of the hyperparameters used in the experiment. Hyperparameter names must start with a letter, followed by letters, digits, or underscores. Hyperparameter values must be scalars or vectors with numeric, logical, or string values, or cell arrays of character vectors. 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"]
- {'squeezeenet' 'googlenet' 'resnet18'}

When you run the experiment, Experiment Manager trains the network using every combination of the hyperparameter values specified in the table.

## Bayesian Optimization

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:

- **Name** — Enter a valid hyperparameter name. Hyperparameter names must start with a letter, followed by letters, digits, or underscores.
- **Range** — For a real- or integer-valued hyperparameter, enter a two-element vector that gives the lower bound and upper bound of the hyperparameter. For a categorical hyperparameter, enter an array of strings or a cell array of character vectors that lists the possible values of the hyperparameter.
- **Type** — Select **real** for a real-valued hyperparameter, **integer** for an integer-valued hyperparameter, or **categorical** for a categorical hyperparameter.
- **Transform** — Select **none** to use no transform or **log** to use a logarithmic transform. When you select **log**, the hyperparameter values must be positive. With this setting, the Bayesian optimization algorithm models the hyperparameter on a logarithmic scale.

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.

Optionally, specify deterministic constraints, conditional constraints, and an acquisition function for the Bayesian optimization algorithm. Under **Bayesian Optimization Options**, click **Advanced Options** and specify:

- **Deterministic Constraints** - Enter the name of a deterministic constraint function. To run the Bayesian optimization algorithm without deterministic constraints, leave this option blank. For more information, see “Deterministic Constraints — XConstraintFcn” (Statistics and Machine Learning Toolbox).
- **Conditional Constraints** - Enter the name of a conditional constraint function. To run the Bayesian optimization algorithm without conditional constraints, leave this option blank. For more information, see “Conditional Constraints — ConditionalVariableFcn” (Statistics and Machine Learning Toolbox).

- **Acquisition Function Name** - Select an acquisition function from the list. The default value for this option is `expected-improvement-plus`. For more information, see “Acquisition Function Types” (Statistics and Machine Learning Toolbox).

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. Bayesian optimization requires Statistics and Machine Learning Toolbox.

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.

---

**Tip** When writing your training function, follow these guidelines:


- Load training and validation data from a location that is accessible to all your workers. For example, store your data outside the project and access the data by using an absolute path. Alternatively, create a datastore object that can access the data on another machine by setting up the `AlternateFileSystemRoots` property of the datastore. For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.
- 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.
- When the training is complete, the **Review Results** gallery in the toolstrip displays a button for each figure that you create in your training function. To display a figure in the **Visualizations** pane, click the corresponding button in the **Custom Plot** section of the gallery. Specify the name of the button by using the `Name` property of the figure. If you do not name your figure, Experiment Manager derives the name of the button from the axes or figure title.


---

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






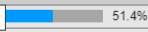






Experiment Manager provides two options for interrupting experiments:


-  **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 results for these trials.












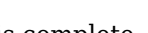
-  **Cancel** marks any running trials as Canceled and discards their results. When the experiment stops, you cannot display the training plot or export the training results for these trials.


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






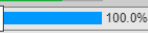





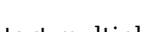
Instead of interrupting the entire experiment, you can stop an individual trial that is running or cancel an individual queued trial. In the **Actions** column of the results table, click the Stop button  or the Cancel button  for the trial.

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete (Max epochs completed)		 100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.9600	1.3936
2	Complete (Max epochs completed)		 100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.5600	1.3986
3	Running	 Stop	 51.4%	0 hr 2 min 2 sec	0.0075	98.4375	0.0962	61.2800	1.3933
4	Running		 51.4%	0 hr 2 min 2 sec	0.0100	98.4375	0.0734	61.3200	1.4084
5	Queued		 0.0%						
6	Queued		 0.0%		0.0150				

To reduce the size of your experiments, discard the results of trials that are no longer relevant. In the **Actions** column of the results table, click the Discard button  for a trial. Experiment Manager deletes the training results, training plot, and confusion matrix for the trial from your project.

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete (Max epochs completed)		 100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.9600	1.3936
2	Complete (Max epochs completed)		 100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.5600	1.3986
3	Stopped	 Discard	 61.0%	0 hr 3 min 4 sec	0.0075	100.0000	0.0321	57.9200	1.4811
4	Complete (Max epochs completed)		 100.0%	0 hr 3 min 57 sec	0.0100	100.0000	0.0143	63.5600	1.4134
5	Complete (Max epochs completed)		 100.0%	0 hr 3 min 22 sec	0.0125	100.0000	0.0128	64.4000	1.4245
6	Canceled		 0.0%		0.0150				

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

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete (Max epochs completed)		 100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.9600	1.3936
2	Complete (Max epochs completed)		 100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.5600	1.3986
3	Stopped	 Restart	 61.0%	0 hr 3 min 4 sec	0.0075	100.0000	0.0321	57.9200	1.4811
4	Discarded		 100.0%	0 hr 3 min 57 sec	0.0100	100.0000	0.0143	63.5600	1.4134
5	Complete (Max epochs completed)		 100.0%	0 hr 3 min 22 sec	0.0125	100.0000	0.0128	64.4000	1.4245
6	Canceled		 0.0%		0.0150				

Alternatively, to restart multiple trials, in the **Experiment Manager** toolstrip, open the **Restart** list, select one or more restarting criteria, and click **Restart** . Restarting criteria include All Canceled, All Stopped, All Error, and All Discarded.

**Note** Stop, cancel, and restart options are not available for all experiment types, strategies, or execution modes.

### 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 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.

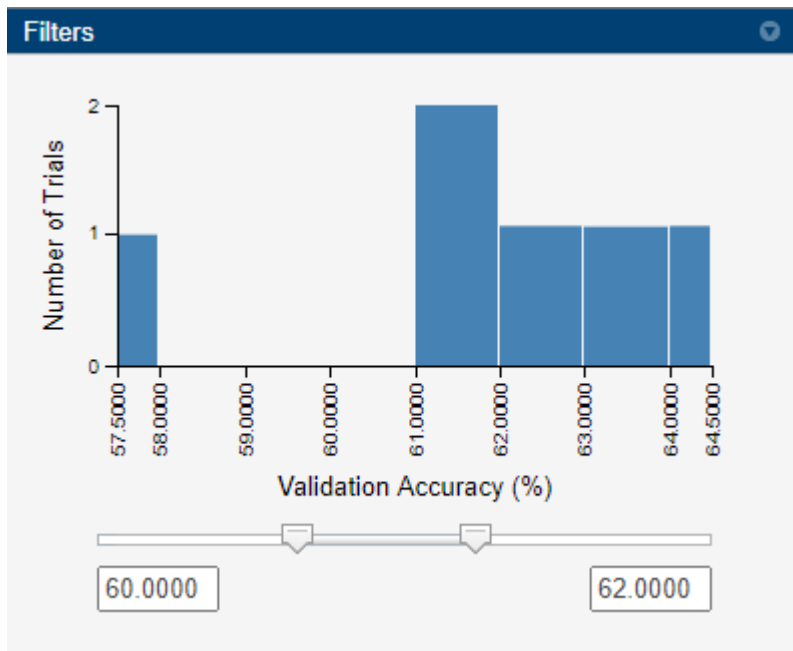
To sort the trials in the results table, use the drop-down list for the column corresponding to a training or validation metric.

- 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**.

Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.96	Sort in Ascending Order
2	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.56	Sort in Descending Order
3	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 3 min 22 sec	0.0075	100.0000	0.0203	62.76	<input type="checkbox"/> Show Filter
4	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 2 min 59 sec	0.0100	100.0000	0.0144	63.4800	1.4136
5	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 3 min 16 sec	0.0125	100.0000	0.0128	64.4400	1.4249
6	✔ Complete (Max epochs completed)	⋮	<div style="width: 100%; height: 10px; background-color: green;"></div> 100.0%	0 hr 3 min 25 sec	0.0150	100.0000	0.0164	61.5600	1.6697

To filter trials from the results table, use the **Filters** pane. This 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 list for the corresponding column and clear the **Show Filter** check box.

- 1 On the **Experiment Manager** toolbar, select **Filters**.
- 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	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
2	✔ Complete (Max epochs completed)		100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.5600	1.3986
6	✔ Complete (Max epochs completed)		100.0%	0 hr 3 min 25 sec	0.0150	100.0000	0.0164	61.5600	1.6697

- 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.

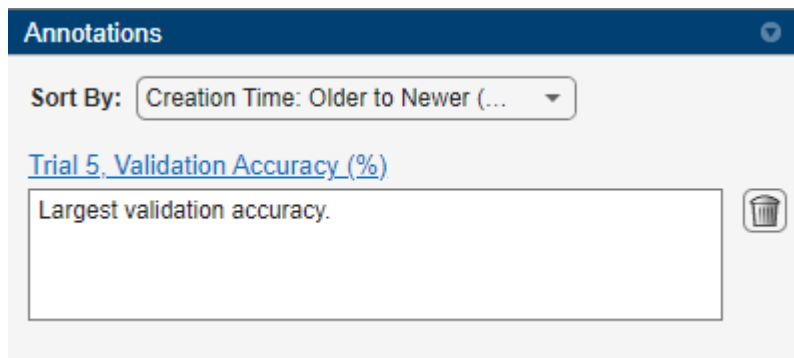
To record observations about the results of your experiment, add an annotation.

- 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** toolbar, select **Annotations > Add Annotation**.

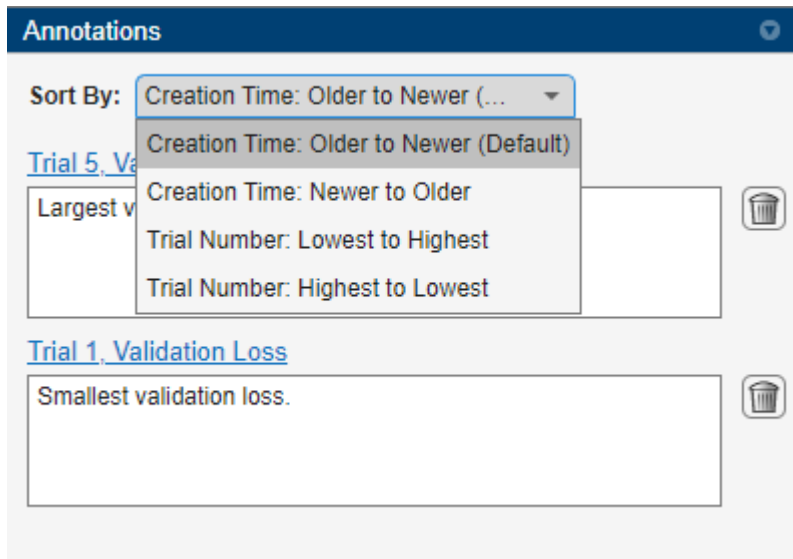
Trial	Status	Actions	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
5	✔ Complete (Max epochs completed)		100.0%	0 hr 3 min 16 sec	0.0125	100.0000	0.0128	64.44	
4	✔ Complete (Max epochs completed)		100.0%	0 hr 2 min 59 sec	0.0100	100.0000	0.0144	63.480	
3	✔ Complete (Max epochs completed)		100.0%	0 hr 3 min 22 sec	0.0075	100.0000	0.0203	62.760	
2	✔ Complete (Max epochs completed)		100.0%	0 hr 4 min 34 sec	0.0050	100.0000	0.0326	61.560	
6	✔ Complete (Max epochs completed)		100.0%	0 hr 3 min 25 sec	0.0150	100.0000	0.0164	61.560	
1	✔ Complete (Max epochs completed)		100.0%	0 hr 4 min 17 sec	0.0025	100.0000	0.0714	57.960	

- Stop
- Restart
- Discard
- Export Training Information
- Export Trained Network
- 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.



- 3 To sort annotations, use the **Sort By** drop-down list. You can sort by creation time or trial number.



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

5

To delete an annotation, click the delete button  to the right of the annotation.

### 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.

- “Generate Experiment Using Deep Network Designer”
- “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”
- “Audio Transfer Learning Using Experiment Manager”
- “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)”
- “Custom Training with Multiple GPUs in Experiment Manager”

## 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 an experiment to find the optimal training options. For more information, see “Generate Experiment Using Deep Network Designer”.
- 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”.

## Version History

### Introduced in R2020a

#### R2023a: Visualizations for custom training experiments

Display visualizations for your custom training experiments directly in the **Experiment Manager** app. When the training is complete, the **Review Results** gallery in the toolstrip displays a button for each figure that you create in your training function. To display a figure in the **Visualizations** pane, click the corresponding button in the **Custom Plot** section of the gallery.

#### R2023a: Debug code before or after running experiment

Diagnose problems in your experiment directly from the **Experiment Manager** app.

- Before running an experiment, you can test your setup and training functions with your choice of hyperparameter values.

- After running an experiment, you can debug your setup and training functions using the same random seed and hyperparameters values you used in one of your trials.



For more information, see “Debug Code Before and After Running Experiments”.

### **R2023a: Ease-of-use enhancements**

- Specify deterministic constraints, conditional constraints, and an acquisition function for experiments that use Bayesian optimization. Under **Bayesian Optimization Options**, click **Advanced Options** and specify:
  - **Deterministic Constraints**
  - **Conditional Constraints**
  - **Acquisition Function Name**
- Load a project that is already open in MATLAB. When you start the Experiment Manager app, a dialog box prompts you to open the current project in Experiment Manager. Alternatively, in the Experiment Manager app, select **New > Project** and, in the dialog box, click **Project from MATLAB**.
- If you have Audio Toolbox, you can set up your built-in or custom training experiments for audio classification by selecting a preconfigured template.

### **R2022b: Ease-of-use enhancements**

*Behavior changed in R2022b*

- In the **Experiment Manager** toolstrip, the **Restart** list replaces the **Restart All Canceled** button. To restart multiple trials of your experiment, open the **Restart** list, select one or more restarting criteria, and click **Restart** . The restarting criteria include All Canceled, All Stopped, All Error, and All Discarded.
- During training, the results table displays the intermediate values for standard training and validation metrics for built-in training experiments. These metrics include loss, accuracy (for classification experiments), and root mean squared error (for regression experiments).
- In built-in training experiments, the **Execution Environment** column of the results table displays whether each trial of a built-in training experiment runs on a single CPU, a single GPU, multiple CPUs, or multiple GPUs.
- To discard the training plot, confusion matrix, and training results for trials that are no longer relevant, in the **Actions** column of the results table, click the Discard button .

### **R2022a: Experiments as batch jobs in a cluster**




If you have Parallel Computing Toolbox and MATLAB Parallel Server, you can send your experiment as a batch job to a remote cluster. If you have only Parallel Computing Toolbox, you can use a local cluster profile to develop and test your experiments on your client machine instead of running them on a network cluster. For more information, see “Offload Experiments as Batch Jobs to Cluster”.

### **R2022a: Ease-of-use enhancements**

*Behavior changed in R2022a*

- In the **Experiment Manager** toolstrip, the **Mode** list replaces the **Use Parallel** button.



- To run one trial of the experiment at a time, select **Sequential** and click **Run**.
- To run multiple trials at the same time, select **Simultaneous** and click **Run**.
- To offload the experiment as a batch job, select **Batch Sequential** or **Batch Simultaneous**, specify your cluster and pool size, and click **Run**.
- Manage experiments using new **Experiment Browser** context menu options:
  - To add a new experiment to a project, right-click the name of the project and select **New Experiment**.
  - To create a copy of an experiment, right-click the name of the experiment and select **Duplicate**.
- Specify hyperparameter values as cell arrays of character vectors. In previous releases, Experiment Manager supported only hyperparameter specifications using scalars and vectors with numeric, logical, or string values.
- To stop, cancel, or restart a trial, in the **Action** column of the results table, click the Stop , Cancel , or Restart  buttons. In previous releases, these buttons were located in the **Progress** column. Alternatively, you can right-click the row for the trial and, in the context menu, select **Stop**, **Cancel**, or **Restart**.
- When an experiment trial ends, the **Status** column of the results table displays one of these reasons for stopping:
  - Max epochs completed
  - Met validation criterion
  - Stopped by OutputFcn
  - Training loss is NaN
- To sort annotations by creation time or trial number, in the **Annotations** pane, use the **Sort By** list.
- After training completes, save the contents of the results table as a `table` array in the MATLAB workspace by selecting **Export > Results Table**.
- To export the training information or trained network for a stopped or completed trial, right-click the row for the trial and, in the context menu, select **Export Training Information** or **Export Trained Network**.

### R2021b: Bayesian optimization in custom training experiments

If you have Statistics and Machine Learning Toolbox, you can use Bayesian optimization to determine the best combination of hyperparameters for a custom training experiment. Previously, custom training experiments supported only sweeping hyperparameters. For more information, see “Use Bayesian Optimization in Custom Training Experiments”.

### R2021b: Experiments in MATLAB Online

Run Experiment Manager in your web browser by using MATLAB Online™. For parallel execution of experiments, you must have access to a Cloud Center cluster.

### R2021b: Ease-of-use enhancements

- In the **Experiment Manager** toolstrip, click **Cancel** to stop an experiment, mark any running trials as Canceled, and discard their results. When the training is complete, click **Restart All Canceled** to restart all the trials that you canceled.
- Use keyboard shortcuts to navigate Experiment Manager when using a mouse is not an option. For more information, see “Keyboard Shortcuts for Experiment Manager”.

## R2021a: Custom training experiments

Create custom training experiments to support workflows such as:

- Using a custom training loop on a dlnetwork, such as a Siamese network or a generative adversarial network (GAN)
- Training a network by using a model function or a custom learning rate schedule
- Updating the learnable parameters of a network by using a custom function

## R2021a: Ease-of-use enhancements

- When you create an experiment, use a preconfigured template as a guide for defining your experiment. Experiment templates support workflows that include image classification, image regression, sequence classification, semantic segmentation, and custom training loops.
- Add annotations to record observations about the results of your experiment. Right-click a cell in the results table and select **Add Annotation**. For more information, see “Sort, Filter, and Annotate Experiment Results” on page 1-57.

## R2020b: Bayesian optimization

If you have Statistics and Machine Learning Toolbox, you can use Bayesian optimization to determine the best combination of hyperparameters for an experiment. For more information, see “Tune Experiment Hyperparameters by Using Bayesian Optimization”.

## R2020b: Parallel execution

If you have Parallel Computing Toolbox, you can run multiple trials of an experiment at the same time by clicking **Use Parallel** and then **Run**. Experiment Manager starts the parallel pool and executes multiple simultaneous trials. For more information, see “Use Experiment Manager to Train Networks in Parallel”.

## See Also

### Apps

**Deep Network Designer**

### Functions

`dlnetwork` | `trainNetwork` | `trainingOptions` | `table`

### Objects

`experiments.Monitor`

**Topics**

- "Generate Experiment Using Deep Network Designer"
- "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"
- "Audio Transfer Learning Using Experiment Manager"
- "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)"
- "Custom Training with Multiple GPUs in Experiment Manager"
- "Use Experiment Manager to Train Networks in Parallel"
- "Offload Experiments as Batch Jobs to Cluster"
- "Debug Code Before and After Running Experiments"
- "Keyboard Shortcuts for Experiment Manager"

## activations

Compute deep learning network layer activations

### Syntax

```
act = activations(net,images,layer)
act = activations(net,sequences,layer)
act = activations(net,features,layer)

act = activations(net,X1,...,XN,layer)
act = activations(net,mixed,layer)

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

### Description

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

To compute activations using a trained `SeriesNetwork` or `DAGNetwork`, use the `activations` function. To compute activations of a `dlnetwork` objects, use the `forward` or `predict` function and specify the `Outputs` option.

`act = activations(net,images,layer)` returns the network activations for the layer with name or number `layer` using the specified images.

`act = activations(net,sequences,layer)` returns the network activations for the layer using the specified sequences.

`act = activations(net,features,layer)` returns the network activations for the layer using the specified feature data.

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

`act = activations(net,mixed,layer)` returns the network activations for the layer using the trained network `net` with multiple inputs of mixed data types.

`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". Use this syntax with any of the input arguments in previous syntaxes. Specify name-value 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 requires only 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)
```



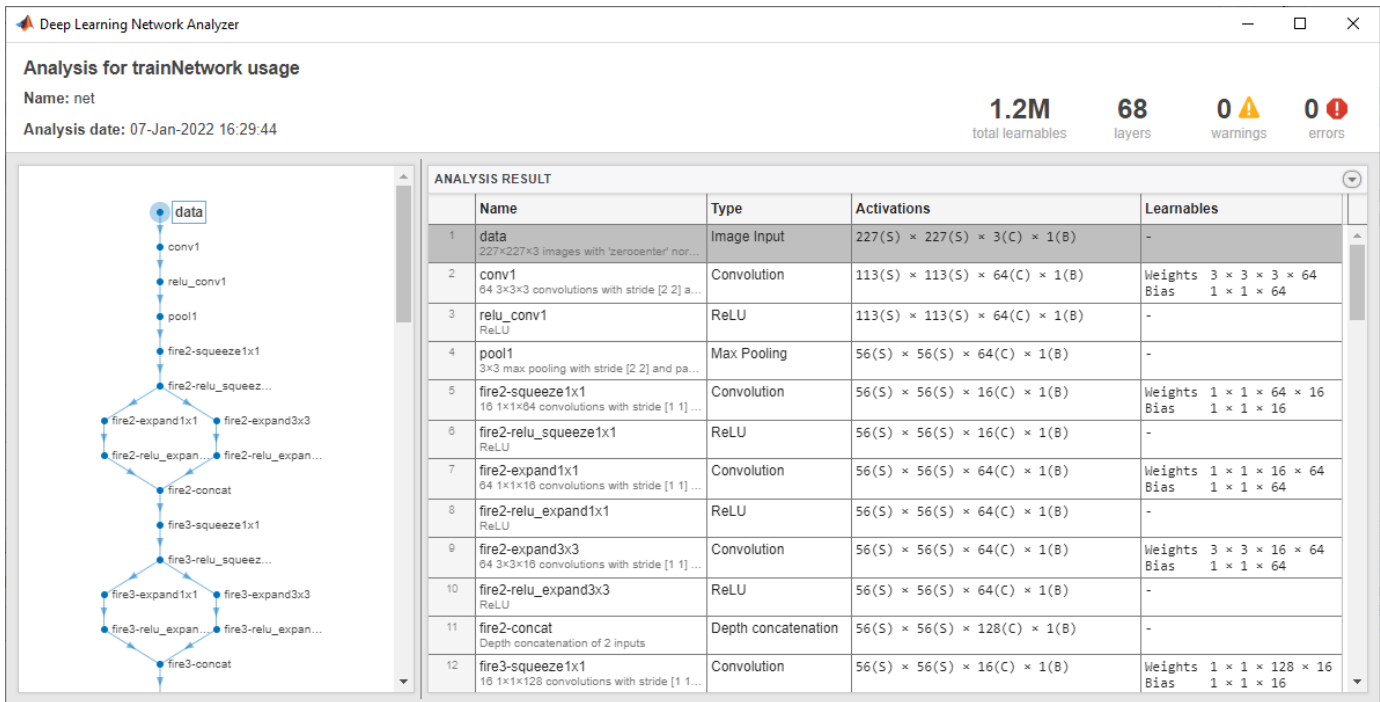
## Load Pretrained Network

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 = 1x3
```

```
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 inputting them 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.

```
TTrain = imdsTrain.Labels;  
TTest = 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,TTrain);
```

### Classify Test Images

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

```
YPred = predict(mdl,featuresTest);
```

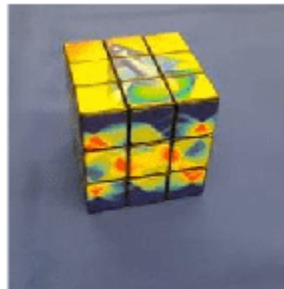
Display four sample test images with their predicted labels.

```
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
```

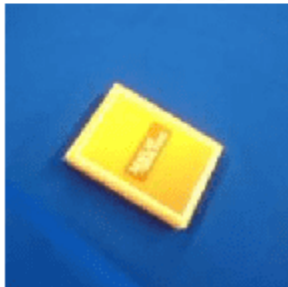
**MathWorks Cap**



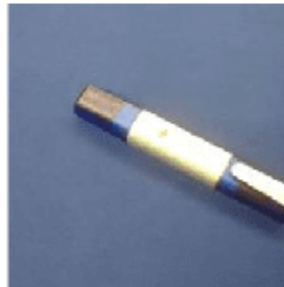
**MathWorks Cube**



**MathWorks Playing Cards**



**MathWorks Screwdriver**





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

```
accuracy = mean(YPred == TTest)
```

```
accuracy = 0.9500
```

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.

### images — Image data

datastore | numeric array | table

Image data, specified as one of the following.

Data Type		Description	Example Usage
Datastore	ImageDatastore	Datastore of images saved on disk	Make predictions with images saved on disk, where the images are the same size.  When the images are different sizes, use an AugmentedImageDatastore object.
	AugmentedImageDatastore	Datastore that applies random affine geometric transformations, including resizing, rotation, reflection, shear, and translation	Make predictions with images saved on disk, where the images are different sizes.
	TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by activations.</li> <li>Apply custom transformations to datastore output.</li> </ul>

Data Type		Description	Example Usage
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions using data in a format that other datastores do not support.  For details, see “Develop Custom Mini-Batch Datastore”.
Numeric array		Images specified as a numeric array	Make predictions using data that fits in memory and does not require additional processing like resizing.
Table		Images specified as a table	Make predictions using data stored in a table.

When you use a datastore with 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. Use datastores when you have data that does not fit in memory or when you want to resize the input data.

These datastores are directly compatible with `activations` for image data.:

- `ImageDatastore`
- `AugmentedImageDatastore`
- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch 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 the `imageDatastore` function for preprocessing or resizing, as this option is usually significantly slower.

You can use other built-in datastores for making predictions by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the format required by `classify`.

The required format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output
Single input	<p>Table or cell array, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom datastores must output tables.</p>	<pre>data = read(ds) data =     4x1 table       Predictors     _____     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}</pre>
		<pre>data = read(ds) data =     4x1 cell array     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}</pre>
Multiple input	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>data = read(ds) data =     4x2 cell array     {224x224x3 double}   {128x128x3 do     {224x224x3 double}   {128x128x3 do     {224x224x3 double}   {128x128x3 do     {224x224x3 double}   {128x128x3 do</pre>

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

Data	Format
2-D images	<i>h</i> -by- <i>w</i> -by- <i>c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively

Data	Format
3-D images	<i>h-by-w-by-d-by-c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively

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.

The size and shape of the numeric array depends on the type of image data.

Data	Format
2-D images	<i>h-by-w-by-c-by-N</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively, and <i>N</i> is the number of images
3-D images	<i>h-by-w-by-d-by-c-by-N</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively, and <i>N</i> is the number of images

**Table**

As an alternative to datastores or numeric arrays, you can also specify images in a table.

When you specify images 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

---

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`

Complex Number Support: Yes

**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	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by activations.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>Make predictions using networks with multiple inputs.</li> <li>Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>
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	Make predictions using data that fits in memory and does not require additional processing like custom transformations.

**Datastore**

Datastores read mini-batches of sequences and responses. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with activations for sequence data:

- CombinedDatastore
- TransformedDatastore
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 activations. For example, you can transform and combine data read from in-memory arrays and CSV files using an ArrayDatastore and an TabularTextDatastore object, respectively.

The datastore must return data in a table or cell array. Custom mini-batch datastores must output tables.

Datastore Output	Example Output
Table	<pre>data = read(ds) data =     4x2 table         Predictors         _____         {12x50 double}         {12x50 double}         {12x50 double}         {12x50 double}</pre>
Cell array	<pre>data = read(ds) data =     4x2 cell array         {12x50 double}         {12x50 double}         {12x50 double}         {12x50 double}</pre>

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

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

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.

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.

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.

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

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

**features – Feature data**

`datastore` | `numeric array` | `table`

Feature data, specified as one of the following.

Data Type	Description	Example Usage
Datastore	TransformedDatastore	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by activations.</li> <li>Apply custom transformations to datastore output.</li> </ul>

Data Type		Description	Example Usage
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>
Table		Feature data specified as a table	Make predictions using data stored in a table.
Numeric array		Feature data specified as numeric array	Make predictions 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. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with `activations` for feature data:

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `activations`. 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	<p>Table or cell array with at least one column, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom mini-batch datastores must output tables.</p>	<p>Table for network with one input:</p> <pre>data = read(ds) data =     4x2 table            Predictors     _____     {24x1 double}     {24x1 double}     {24x1 double}     {24x1 double}</pre>
		<p>Cell array for network with one input:</p> <pre>data = read(ds) data =     4x1 cell array     {24x1 double}     {24x1 double}     {24x1 double}     {24x1 double}</pre>
Multiple input layers	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<p>Cell array for network with two inputs:</p> <pre>data = read(ds) data =     4x3 cell array     {24x1 double}  {28x1 double}     {24x1 double}  {28x1 double}     {24x1 double}  {28x1 double}     {24x1 double}  {28x1 double}</pre>

The predictors must be  $c$ -by-1 column vectors, where  $c$  is the number of features.

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 in the table columns depends on the type of task.

Task	Predictors
Feature classification	Features specified in one or more columns as scalars.

**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.

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** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`  
 Complex Number Support: Yes

**X1, . . . , XN — Numeric or cell arrays for networks with multiple inputs**

`numeric array` | `cell array`

Numeric or cell arrays for networks with multiple inputs.

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`  
 Complex Number Support: Yes

**mixed — Mixed data**

`TransformedDatastore` | `CombinedDatastore` | `custom mini-batch datastore`

Mixed data, specified as one of the following.

Data Type	Description	Example Usage
TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Transform outputs of datastores not supported by activations so they have the required format.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>

You can use other built-in datastores for making predictions 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 activations. For more information, see “Datastores for Deep Learning”.

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.

Datastore Output	Example Output
<p>Cell array with <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>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}</pre>

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To convert a numeric array to a datastore, use `arrayDatastore`.

**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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `MiniBatchSize=256` specifies the mini-batch size as 256.

**OutputAs — Format of output activations**`"channels" (default) | "rows" | "columns"`

Format of output activations, specified as `"channels"`, `"rows"`, or `"columns"`. For descriptions of the output formats, see `act`.

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

**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.

**SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **SequencePaddingValue — Value to pad 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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is ‘sequence’ for each recurrent 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 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 only when you use a GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing

Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

- "none" — Disable all acceleration.

If `Acceleration` is "auto", then MATLAB applies a number of compatible optimizations and does not generate a MEX function.

The "auto" and "mex" options can offer performance benefits 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 supports networks that contain the layers listed on the "Supported Layers" (GPU Coder) page, except for `sequenceInputLayer` objects.

The "mex" option is available when you use a single GPU.

To use the "mex" option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning 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.

For quantized networks, the "mex" option requires a CUDA® enabled NVIDIA® GPU with compute capability 6.1, 6.3, or higher.

MATLAB Compiler™ does not support deploying networks when you use the "mex" option.

### **ExecutionEnvironment — Hardware resource**

"auto" (default) | "gpu" | "cpu" | "multi-gpu" | "parallel"

Hardware resource, specified as one of the following:

- "auto" — Use a GPU if one is available; otherwise, use the CPU.
- "gpu" — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (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".

The "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 Computing Requirements” (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 make predictions in parallel with networks with recurrent layers (by setting `ExecutionEnvironment` to either "multi-gpu" or "parallel"), the `SequenceLength` option must be "shortest" or "longest".

Networks with custom layers that contain `State` parameters do not support making predictions in parallel.

## Output Arguments

### **act** — Activations from 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 specified `OutputAs` option.

#### **Image or Folded Sequence Output**

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

<b>OutputAs</b>	<b>act</b>
"channels"	<p>For 2-D image output, <code>act</code> is an <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>n</math> array, where <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and number of channels for the output of the chosen layer, respectively, and <math>n</math> is the number of images. In this case, <code>act(:, :, :, i)</code> contains the activations for the <math>i</math>th image.</p> <p>For 3-D image output, <code>act</code> is an <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math>-by-<math>n</math> array, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and number of channels for the output of the chosen layer, respectively, and <math>n</math> is the number of images. In this case, <code>act(:, :, :, :, i)</code> contains the activations for the <math>i</math>th image.</p> <p>For folded 2-D image sequence output, <code>act</code> is an <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>(n*s)</math> array, where <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and number of channels for the output of the chosen layer, respectively, <math>n</math> is the number of sequences, and <math>s</math> is the sequence length. In this case, <code>act(:, :, :, (t-1)*n+k)</code> contains the activations for time step <math>t</math> of the <math>k</math>th sequence.</p> <p>For folded 3-D image sequence output, <code>act</code> is an <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math>-by-<math>(n*s)</math> array, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and number of channels for the output of the chosen layer, respectively, <math>n</math> is the number of sequences, and <math>s</math> is the sequence length. In this case, <code>act(:, :, :, :, (t-1)*n+k)</code> contains the activations for time step <math>t</math> of the <math>k</math>th sequence.</p>

OutputAs	act
"rows"	<p>For 2-D and 3-D image output, <code>act</code> is an <math>n</math>-by-<math>m</math> matrix, where <math>n</math> is the number of images and <math>m</math> is the number of output elements from the layer. In this case, <code>act(i, :)</code> contains the activations for the <math>i</math>th image.</p> <p>For folded 2-D and 3-D image sequence output, <code>act</code> is an <math>(n*s)</math>-by-<math>m</math> matrix, where <math>n</math> is the number of sequences, <math>s</math> is the sequence length, and <math>m</math> is the number of output elements from the layer. In this case, <code>act((t-1)*n+k, :)</code> contains the activations for time step <math>t</math> of the <math>k</math>th sequence.</p>
"columns"	<p>For 2-D and 3-D image output, <code>act</code> is an <math>m</math>-by-<math>n</math> matrix, where <math>m</math> is the number of output elements from the chosen layer and <math>n</math> is the number of images. In this case, <code>act(:, i)</code> contains the activations for the <math>i</math>th image.</p> <p>For folded 2-D and 3-D image sequence output, <code>act</code> is an <math>m</math>-by-<math>(n*s)</math> matrix, where <math>m</math> is the number of output elements from the chosen layer, <math>n</math> is the number of sequences, and <math>s</math> is the sequence length. In this case, <code>act(:, (t-1)*n+k)</code> contains the activations for time step <math>t</math> of the <math>k</math>th sequence.</p>

### Sequence Output

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

OutputAs	act
"channels"	<p>For vector sequence output, <code>act</code> is an <math>n</math>-by-1 cell array of <math>c</math>-by-<math>s</math> matrices, where <math>n</math> is the number of sequences, <math>c</math> is the number of features in the sequence, and <math>s</math> is the sequence length.</p> <p>For 2-D image sequence output, <code>act</code> is an <math>n</math>-by-1 cell array of <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>s</math> matrices, where <math>n</math> is the number of sequences, <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and the number of channels of the images, respectively, and <math>s</math> is the sequence length.</p> <p>For 3-D image sequence output, <code>act</code> is an <math>n</math>-by-1 cell array of <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>d</math>-by-<math>s</math> matrices, where <math>n</math> is the number of sequences, <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and the number of channels of the images, respectively, and <math>s</math> is the sequence length.</p> <p>In these cases, <code>act{i}</code> contains the activations of the <math>i</math>th sequence.</p>

### Feature Vector and Single Time Step Output

If `layer` outputs a feature vector or a single time step of a sequence (for example, an LSTM layer with the output mode "last"), then `act` is a numeric array.



OutputAs	act
"channels"	<p>For a feature vector or single time step containing vector data, <code>act</code> is a <math>c</math>-by-<math>n</math> matrix, where <math>n</math> is the number of observations and <math>c</math> is the number of features.</p> <p>For a single time step containing 2-D image data, <code>act</code> is a <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>n</math> array, where <math>n</math> is the number of sequences and <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and the number of channels of the images, respectively.</p> <p>For a single time step containing 3-D image data, <code>act</code> is a <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>d</math>-by-<math>n</math> array, where <math>n</math> is the number of sequences and <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and the number of channels of the images, respectively.</p>
"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, <code>act(i, :)</code> contains the activations for the $i$ th 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, <code>act(:, i)</code> contains the activations for the $i$ th image.

## Algorithms

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Version History

### Introduced in R2016a

#### **R2022b: Prediction functions pad mini-batches to length of longest sequence before splitting when you specify SequenceLength option as an integer**

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch,

then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

## References

- [1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Multidimensional Curve Classification Using Passing-through Regions." *Pattern Recognition Letters* 20, no. 11-13 (November 1999): 1103-11. [https://doi.org/10.1016/S0167-8655\(99\)00077-X](https://doi.org/10.1016/S0167-8655(99)00077-X).
- [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, images, layer)`, where `images` is a numeric array
  - `act = activations(net, sequences, layer)`, where `sequences` is a cell array
  - `act = activations(net, features, layer)`, where `features` is a numeric array
  - `act = activations(__, Name, Value)` using any of the previous syntaxes
- For numeric inputs, the input 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, images, layer)`, where `images` is a numeric array
  - `act = activations(net, sequences, layer)`, where `sequences` is a cell array or numeric array
  - `act = activations(net, features, layer)`, where `features` is a numeric array
  - `act = activations(__, Name, Value)` using any of the previous syntaxes
- For numeric inputs, the input 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™.

- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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”

# 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 `darray` object or structure of `darray` object inputs, the trace depends on the size, format, and underlying datatype of the `darray`. That is, the accelerated function triggers a new trace for `darray` inputs with size, format, or underlying datatype not contained in the cache. Any `darray` 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 `daccelerate` for function calls that:

- are long-running
- have `darray` objects, structures of `darray` 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 `dfeval` 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");
net = s.net;
classNames = s.classNames;
```

Accelerate the model loss function `modelLoss` listed at the end of the example.

```
fun = @modelLoss;
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: @modelLoss
     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");  
X = dlarray(X,"SSCB");  
  
T = categorical(classNames(randi(10,[128 1])));  
T = onehotencode(T,2)';  
T = dlarray(T,"CB");  
  
[loss,gradients,state] = dlfeval(accfun,net,X,T);
```

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

```
accfun.Occupancy
```

```
ans = 2
```

### Model Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding target labels `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)  
  
[Y,state] = forward(net,X);  
loss = crossentropy(Y,T);  
gradients = dlgradient(loss,net.Learnables);  
  
end
```

### Clear Cache of Accelerated Function

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

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

Accelerate the model loss function `modelLoss` listed at the end of the example.

```
fun = @modelLoss;  
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: @modelLoss
    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");
X = dlarray(X,"SSCB");

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

[loss,gradients,state] = dlfeval(accfun,net,X,T);
```

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 Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding target labels `T` and returns the loss, the gradients of the loss with respect to the

learnable parameters in `net`, and the network state. To compute the gradients, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)

[Y,state] = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.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
```

Clear 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);
```

Clear 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 =  
  3×3 dlarray  
  
    1.7922    1.0357    1.6787  
    1.9595    1.8491    1.7577  
    1.6557    1.9340    1.7431
```

```
dLY = accfun2(dlX)
```

```
dLY =  
  3×3 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

## Version History

Introduced in R2021a

### 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”

## adamupdate

Update parameters using adaptive moment estimation (Adam)

### Syntax

```
[netUpdated, averageGrad, averageSqGrad] = adamupdate(net, 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.
- 

`[netUpdated, averageGrad, averageSqGrad] = adamupdate(net, grad, averageGrad, averageSqGrad, iteration)` updates the learnable parameters of the network `net` 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,TTrain] = digitTrain4DArrayData;
classes = categories(TTrain);
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], 'Mean', mean(XTrain,4))
    convolution2dLayer(5,20)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers);
```

### Define Model Loss Function

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

### Specify Training Options

Specify the options to use during training.

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

### Train Network

Initialize the average gradients and squared average gradients.

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

Calculate the total number of iterations for the training progress monitor.

```
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

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 iteration, display the training progress.

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

```
iteration = 0;  
epoch = 0;
```

```
while epoch < numEpochs && ~monitor.Stop  
    epoch = epoch + 1;  
  
    % Shuffle data.  
    idx = randperm(numel(TTrain));  
    XTrain = XTrain(:,:,,idx);  
    TTrain = TTrain(idx);  
  
    i = 0;  
    while i < numIterationsPerEpoch && ~monitor.Stop  
        i = i + 1;  
        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);

T = zeros(numClasses, miniBatchSize,"single");
for c = 1:numClasses
    T(c,TTrain(idx)==classes(c)) = 1;
end

% Convert mini-batch of data to a dlarray.
X = dlarray(single(X),"SSCB");

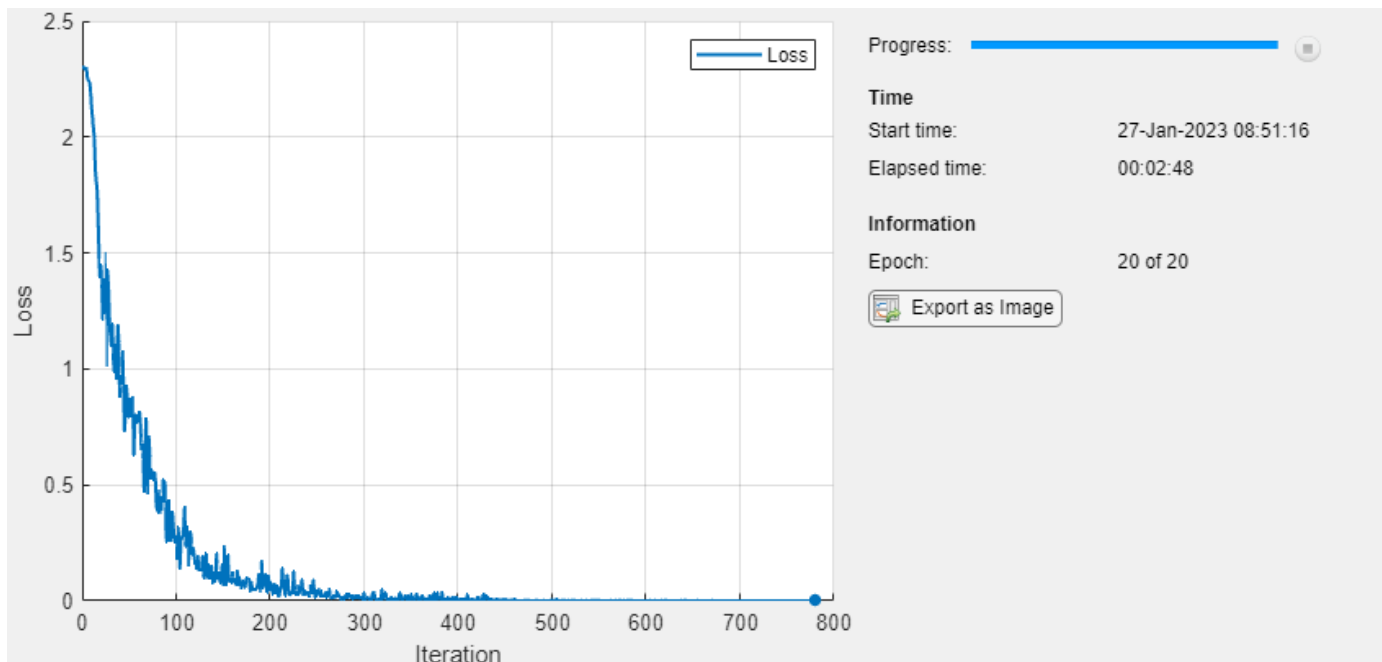
% If training on a GPU, then convert data to a gpuArray.
if canUseGPU
    X = gpuArray(X);
end

% Evaluate the model loss and gradients using dlfeval and the
% modelLoss function.
[loss,gradients] = dlfeval(@modelLoss,net,X,T);

% Update the network parameters using the Adam optimizer.
[net,averageGrad,averageSqGrad] = adamupdate(net,gradients,averageGrad,averageSqGrad,ite

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch + " of " + numEpochs);
monitor.Progress = 100 * iteration/numIterations;
end
end

```



### Test Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

```
[XTest,TTest] = digitTest4DArrayData;
```

Convert the data to a `dLarray` with the dimension format "SSCB" (spatial, spatial, channel, batch). For GPU prediction, also convert the data to a `gpuArray`.

```
XTest = dLarray(XTest,"SSCB");  
if canUseGPU  
    XTest = gpuArray(XTest);  
end
```

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

```
YTest = predict(net,XTest);  
[~,idx] = max(extractdata(YTest),[],1);  
YTest = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YTest==TTest)  
  
accuracy = 0.9908
```

### Model Loss Function

The `modelLoss` helper function takes a `dLnetwork` object `net` and a mini-batch of input data `X` with corresponding labels `T`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `net`. To compute the gradients automatically, use the `dlgradient` function.

```
function [loss,gradients] = modelLoss(net,X,T)  
  
Y = forward(net,X);  
loss = crossentropy(Y,T);  
gradients = dlgradient(loss,net.Learnables);  
  
end
```

## Input Arguments

### **net** — Network

`dLnetwork` object

Network, specified as a `dLnetwork` object.

The function updates the `Learnables` property of the `dLnetwork` object. `net.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 `net.Learnables`.

### **params** — Network learnable parameters

`dLarray` | numeric array | cell array | structure | table

Network learnable parameters, specified as a `darray`, 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 `darray`.

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 `darray` 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`.

### **grad — Gradients of the loss**

`darray` | numeric array | cell array | structure | table

Gradients of the loss, specified as a `darray`, 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`.

<b>Input</b>	<b>Learnable Parameters</b>	<b>Gradients</b>
<code>net</code>	Table <code>net.Learnables</code> containing <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>darray</code> .	Table with the same data type, variables, and ordering as <code>net.Learnables</code> . <code>grad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the gradient of each learnable parameter.
<code>params</code>	<code>darray</code>	<code>darray</code> with the same data type and ordering as <code>params</code>
	Numeric array	Numeric array with the same data type and ordering as <code>params</code>
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code>
	Structure	Structure with the same data types, fields, and ordering as <code>params</code>
	Table with <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>darray</code> .	Table with the same data types, variables, and ordering as <code>params</code> . <code>grad</code> must have a <code>Value</code> 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
net	Table <code>net.Learnables</code> containing <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data type, variables, and ordering as <code>net.Learnables</code> . <code>averageGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average gradient of each learnable parameter.
params	<code>dlarray</code>	<code>dlarray</code> with the same data type and ordering as <code>params</code>
	Numeric array	Numeric array with the same data type and ordering as <code>params</code>
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code>
	Structure	Structure with the same data types, fields, and ordering as <code>params</code>
	Table with <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data types, variables, and ordering as <code>params</code> . <code>averageGrad</code> must have a <code>Value</code> 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
<code>net</code>	Table <code>net.Learnables</code> containing <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data type, variables, and ordering as <code>net.Learnables</code> . <code>averageSqGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.
<code>params</code>	<code>dlarray</code>	<code>dlarray</code> with the same data type and ordering as <code>params</code>
	Numeric array	Numeric array with the same data type and ordering as <code>params</code>
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code>
	Structure	Structure with the same data types, fields, and ordering as <code>params</code>
	Table with <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data types, variables and ordering as <code>params</code> . <code>averageSqGrad</code> must have a <code>Value</code> 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

**netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as a `dlnetwork` object.

The function updates the `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 1-1782 on the `trainingOptions` reference page.

## Version History

Introduced in R2019b

## 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”

## addInputLayer

Add input layer to network

### Syntax

```
netUpdated = addInputLayer(net, layer)
netUpdated = addInputLayer(net, layer, inputName)
netUpdated = addInputLayer( ____, Initialize=tf)
```

### Description

`netUpdated = addInputLayer(net, layer)` adds the input layer `layer` to the network `net` by connecting the input layer to the first unconnected input in `net`. If the updated network supports automatic initialization, then the function automatically initializes the learnable parameters of the network.

`netUpdated = addInputLayer(net, layer, inputName)` also specifies which unconnected input the function connects to the input layer.

`netUpdated = addInputLayer( ____, Initialize=tf)` also specifies whether to initialize the output network using any of the previous syntaxes.

### Examples

#### Add Input Layer to dlnetwork

Create an uninitialized `dlnetwork` object that does not have an input layer.

```
layers = [
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,Stride=2)
    fullyConnectedLayer(10)
    softmaxLayer];

net = dlnetwork(layers,Initialize=false)

net =
    dlnetwork with properties:

        Layers: [5x1 nnet.cnn.layer.Layer]
    Connections: [4x2 table]
    Learnables: [4x3 table]
        State: [0x3 table]
    InputNames: {'conv'}
    OutputNames: {'softmax'}
    Initialized: 0
```

View summary with `summary`.



Create an image input layer with an input size of [28 28 1].

```
layer = imageInputLayer([28 28 1],Normalization="none");
```

Add the input layer to the network.

```
net = addInputLayer(net, layer)
```

```
net =
```

```
  dlnetwork with properties:
```

```
      Layers: [6×1 nnet.cnn.layer.Layer]
Connections: [5×2 table]
Learnables: [4×3 table]
      State: [0×3 table]
InputNames: {'imageinput'}
OutputNames: {'softmax'}
Initialized: 1
```

View summary with `summary`.

View the `Initialized` property. Because the network contains all the information required for initialization, the output network is initialized.

```
net.Initialized
```

```
ans = logical
      1
```

## Connect Input Layers to Specified dlnetwork Inputs

Create an uninitialized `dlnetwork` object that has two unconnected inputs.

```
layers = [
    convolution2dLayer(5,16,Name="conv")
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(50)
    flattenLayer
    concatenationLayer(1,2,Name="cat")
    fullyConnectedLayer(10)
    softmaxLayer];
```

```
net = dlnetwork(layers,Initialize=false);
```

Create an image input layer to connect to the convolution layer.

```
layer = imageInputLayer([28 28 1],Normalization="none");
```

Connect the image input layer to the "conv" layer. Because the network does not contain the information required to initialize the network, the returned network is uninitialized.

```
net = addInputLayer(net, layer, "conv")
```

```
net =  
  dlnetwork with properties:  
  
    Layers: [9×1 nnet.cnn.layer.Layer]  
  Connections: [8×2 table]  
  Learnables: [8×3 table]  
    State: [2×3 table]  
  InputNames: {'imageinput' 'cat/in2'}  
  OutputNames: {'softmax'}  
  Initialized: 0
```

View summary with `summary`.

Create a feature input layer to connect to the second input of the concatenation layer.

```
layer = featureInputLayer(1);
```

Connect the feature input layer to the "in2" input of the "cat" layer. Because the network now contains the information required to initialize the network, the returned network is initialized.

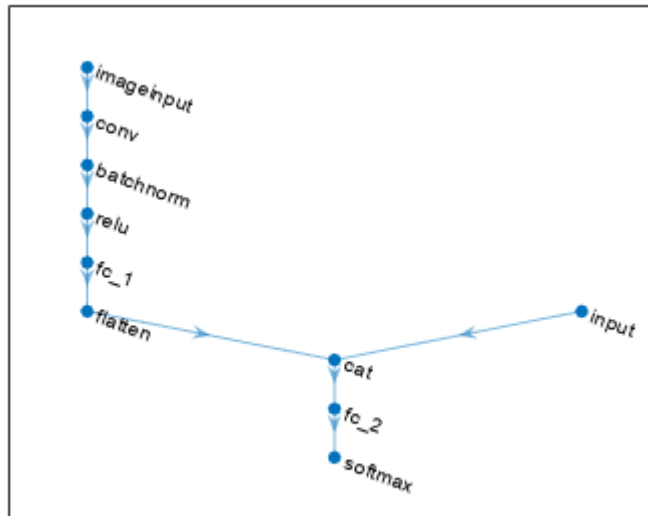
```
net = addInputLayer(net, layer, "cat/in2")
```

```
net =  
  dlnetwork with properties:  
  
    Layers: [10×1 nnet.cnn.layer.Layer]  
  Connections: [9×2 table]  
  Learnables: [8×3 table]  
    State: [2×3 table]  
  InputNames: {'imageinput' 'input'}  
  OutputNames: {'softmax'}  
  Initialized: 1
```

View summary with `summary`.

Visualize the network in a plot.

```
figure  
plot(net)
```



## Input Arguments

### net — Neural network

dlnetwork object

Neural network, specified as a dlnetwork object.

### layer — Input layer to add

Layer object

Input layer to add, specified as a Layer object.

### inputName — Unconnected network input which the function connects the layer

string scalar | character vector | cell array containing a character vector

Unconnected network input which the function connects the layer, specified as a string scalar, a character vector, or a cell array containing a character vector.

Data Types: char | string | cell

### tf — Flag to initialize output network

"auto" (default) | true or 1 | false or 0

Flag to initialize the learnable parameters of the output network, specified as one of these values:

- "auto" — If the network does contains all the required information for initialization, then the function initializes the output network. Otherwise, the function does not initialize the output network.
- 1 (true) — Initialize the output network. If the network does not contain all the required information for initialization, then the function errors.
- 0 (false) — Do not initialize the output network.

## Output Arguments

### **netUpdated** — Updated network

dlnetwork object

Updated network, returned as a dlnetwork object.

The function reorders the layers in the `Layers` property of the network such that the input layer appears immediately before the layer that it is connected to.

## Version History

**Introduced in R2022b**

### See Also

`dlnetwork` | `dlarray` | `dlgradient` | `dlfeval` | `forward` | `predict` | `layerGraph` | `initialize`

### Topics

“Train Network Using Custom Training Loop”

“List of Deep Learning Layers”

“Define Custom Training Loops, Loss Functions, and Networks”

# 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 the name ' '.

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'}
```

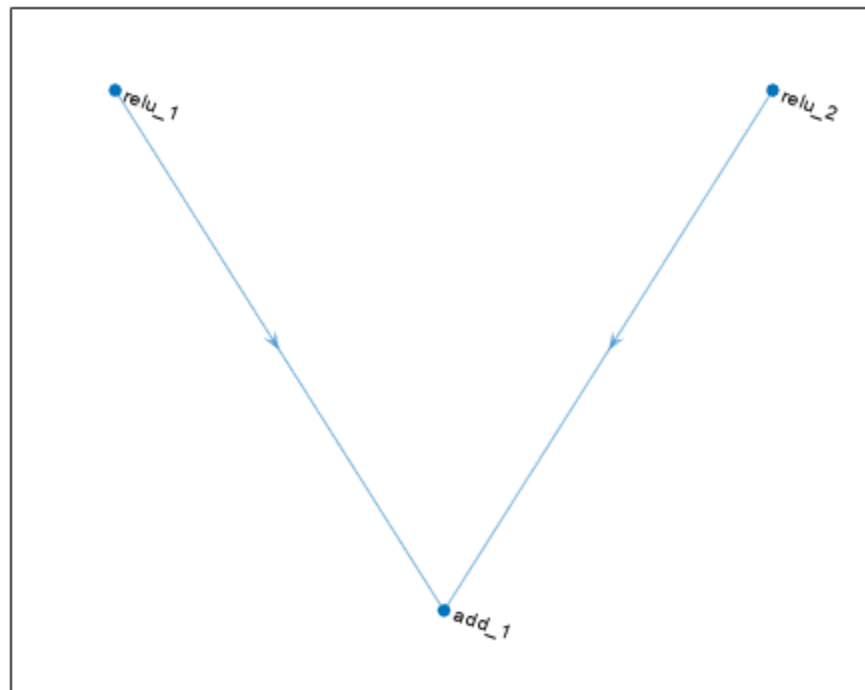
Create two ReLU layers and connect them to the addition layer. The addition layer sums the outputs from the ReLU layers.

```
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];

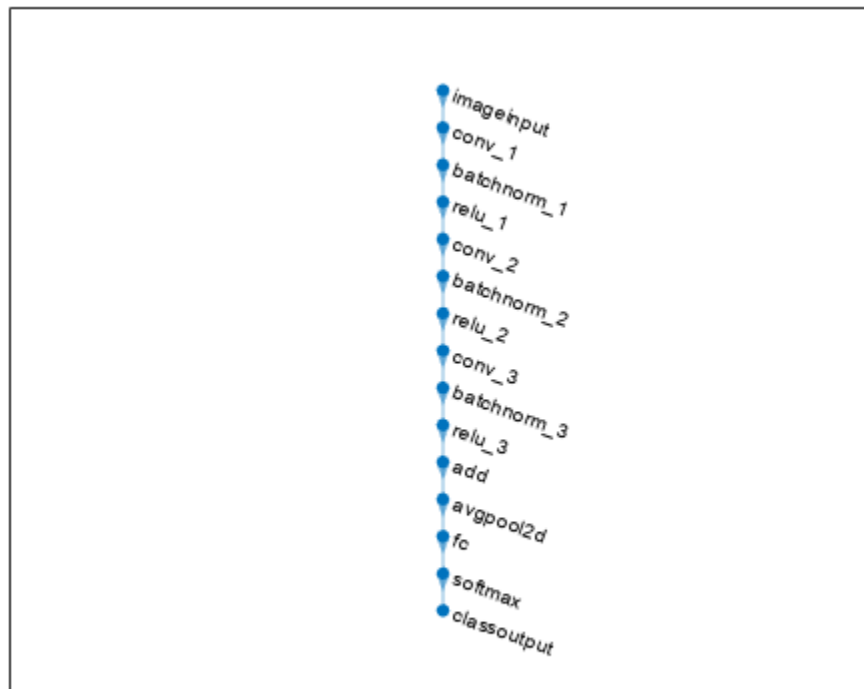
```

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)

```



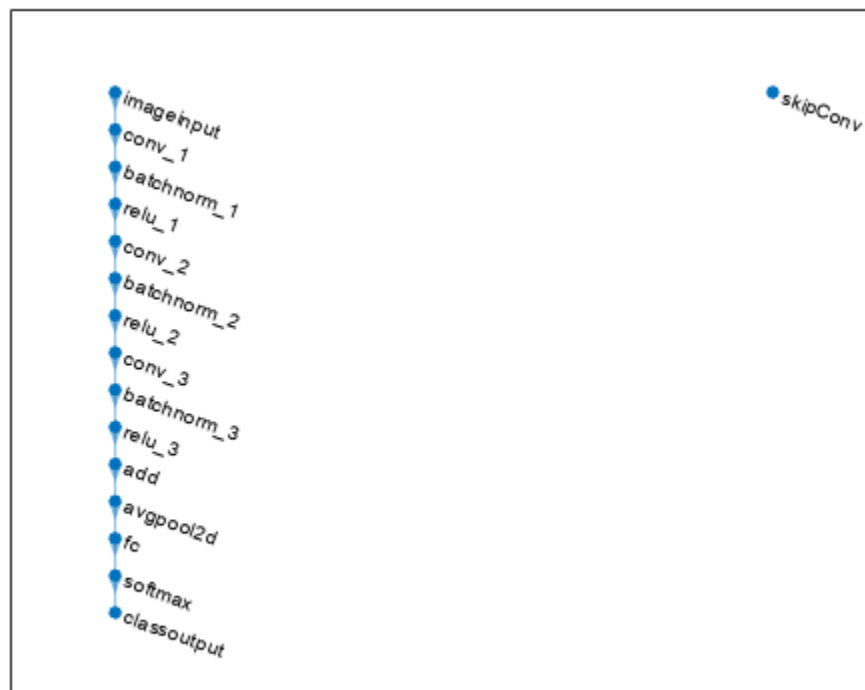
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the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

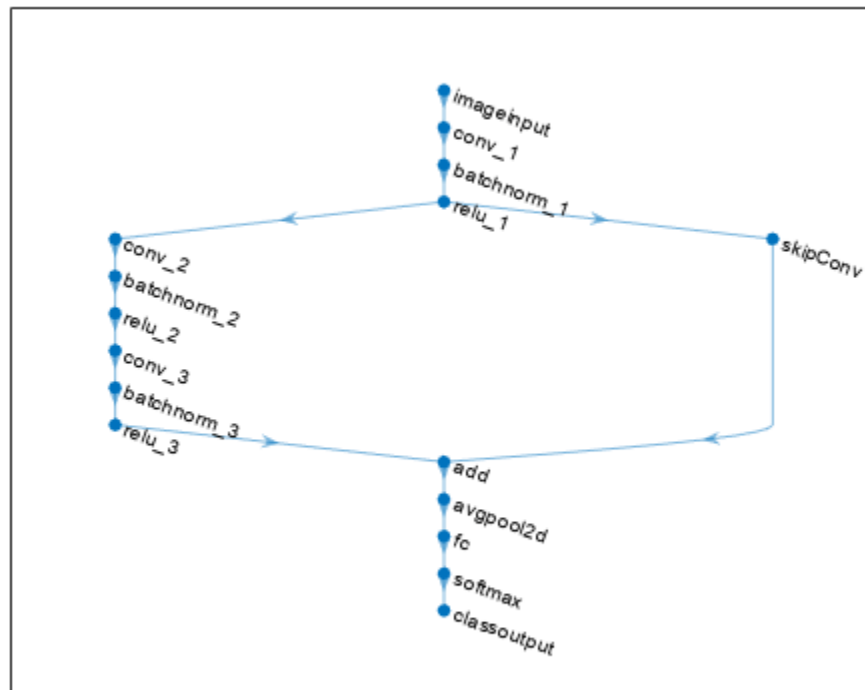
```





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);
```

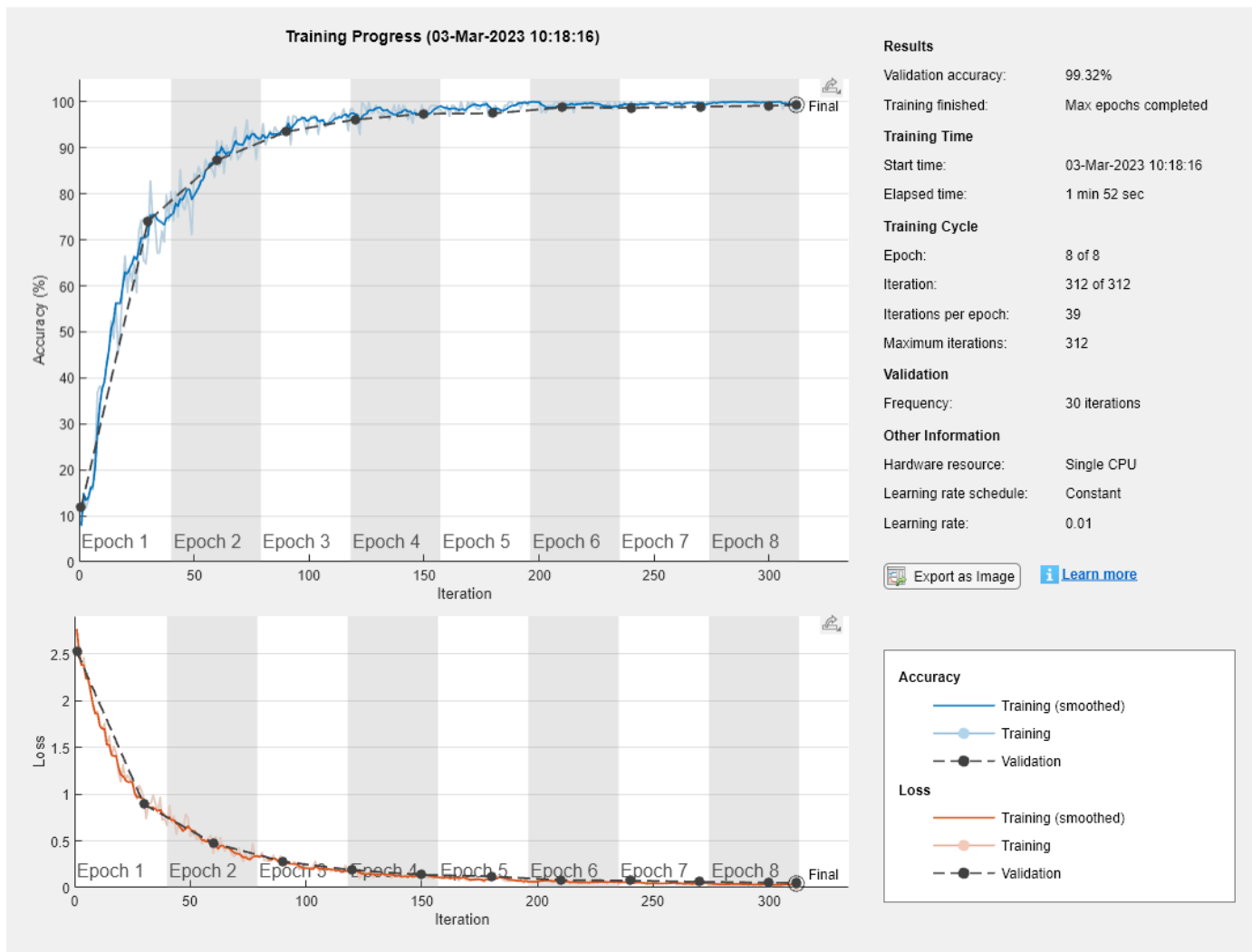


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[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 = 0.9932
```

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `AdditionLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Each input must have data of the same format.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"BU" (batch, unspecified)	"BU" (batch, unspecified)

In `dlnetwork` objects, `AdditionLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2017b

### 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 Neural 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”

## addLayers

**Package:** `nnet.cnn`

Add layers to layer graph or network

### Syntax

```
lgraphUpdated = addLayers(lgraph, larray)
netUpdated = addLayers(net, larray)
```

### Description

`lgraphUpdated = addLayers(lgraph, larray)` adds the network layers in `larray` to the layer graph `lgraph`. The updated layer graph `lgraphUpdated` 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`.

`netUpdated = addLayers(net, larray)` adds the network layers in `larray` to the `dl` network object `net`. The updated network `netUpdated` contains the layers and connections of `net` 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 `net`.

### Examples

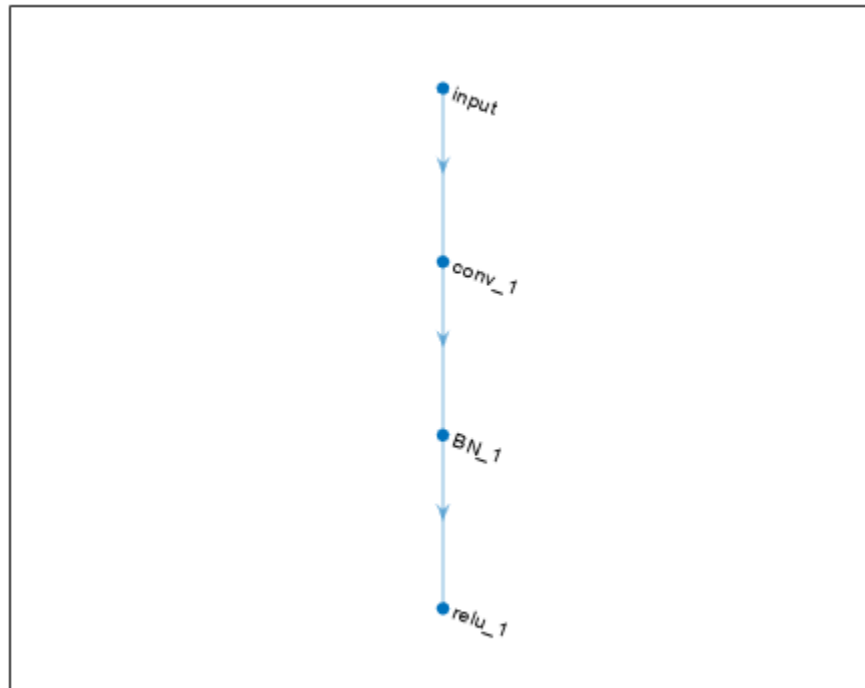
#### Add Layers to Layer Graph

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`.

### **net** — Neural network

dlnetwork object

Neural network, specified as a dlnetwork object.

### **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

### **lgraphUpdated** — Updated layer graph

LayerGraph object

Updated layer graph, returned as a LayerGraph object.

**netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as an uninitialized `dlnetwork` object.

To initialize the learnable parameters of a `dlnetwork` object, use the `initialize` function.

The `addLayers` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

Introduced in R2017b

### See Also

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

### Topics

“Train Deep Learning Network to Classify New Images”

“Train Network with Multiple Outputs”

“Classify Videos Using Deep Learning”



# addMetrics

Compute additional classification performance metrics

## Syntax

```
UpdatedROCObj = addMetrics(rocObj,metrics)
```

## Description

`rocmetrics` computes the false positive rates (FPR), true positive rates (TPR), and additional metrics specified by the `AdditionalMetrics` name-value argument. After creating a `rocmetrics` object, you can compute additional classification performance metrics by using the `addMetrics` function.

`UpdatedROCObj = addMetrics(rocObj,metrics)` computes additional classification performance metrics specified in `metrics` using the classification model information stored in the `rocmetrics` object `rocObj`.

`UpdatedROCObj` contains all the information in `rocObj` plus additional performance metrics computed by `addMetrics`. The function attaches the additional computed metrics (`metrics`) as new variables in the table of the `Metrics` property.

If you compute confidence intervals when you create `rocObj`, the `addMetrics` function computes the confidence intervals for the additional metrics. The new variables in the `Metrics` property contain a three-column matrix in which the first column corresponds to the metric values, and the second and third columns correspond to the lower and upper bounds, respectively. Using confidence intervals requires Statistics and Machine Learning Toolbox.

## Examples

### Compute Additional Metrics

Compute the performance metrics (FPR, TPR, and expected cost) for a multiclass classification problem when you create a `rocmetrics` object. Compute additional metrics, the positive predictive value (PPV) and the negative predictive value (NPV), and add them to the object.

Load a sample of true labels and the prediction scores for a classification problem. For this example, there are five classes: daisy, dandelion, roses, sunflowers, and tulips. The class names are stored in `classNames`. The scores are the softmax prediction scores generated using the `predict` function. `scores` is an N-by-K array where N is the number of observations and K is the number of classes. The column order of scores follows the class order stored in `classNames`.

```
load('flowersDataResponses.mat')

scores = flowersData.scores;
trueLabels = flowersData.trueLabels;

classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels and the classification scores. Specify the column order of scores using `classNames`. By default, `rocmetrics` computes the FPR and TPR. Specify `AdditionalMetrics="ExpectedCost"` to compute the expected cost as well.

```
rocObj = rocmetrics(trueLabels,scores,classNames, ...
  AdditionalMetrics="ExpectedCost");
```

The table in the `Metrics` property of `rocObj` contains performance metric values for each of the classes, vertically concatenated according to the class order. Find and display the top rows for the second class in the table.

```
idx = rocObj.Metrics.ClassName == classNames(2);
head(rocObj.Metrics(idx,:))
```

ClassName	Threshold	FalsePositiveRate	TruePositiveRate	ExpectedCost
dandelion	1	0	0	0.045287
dandelion	1	0	0.23889	0.034469
dandelion	1	0	0.26111	0.033462
dandelion	1	0	0.27222	0.032959
dandelion	1	0	0.28889	0.032204
dandelion	1	0	0.29444	0.031953
dandelion	1	0	0.3	0.031701
dandelion	1	0	0.31111	0.031198

The table in `Metrics` contains the variables for the class names, threshold, false positive rate, true positive rate, and expected cost (the additional metric).

After creating a `rocmetrics` object, you can compute additional metrics using the classification model information stored in the object. Compute the PPV and NPV by using the `addMetrics` function. To overwrite the input argument `rocObj`, assign the output of `addMetrics` to the input.

```
rocObj = addMetrics(rocObj,["PositivePredictiveValue","NegativePredictiveValue"]);
```

Display the `Metrics` property for the top rows.

```
head(rocObj.Metrics(idx,:))
```

ClassName	Threshold	FalsePositiveRate	TruePositiveRate	ExpectedCost	PositiveP
dandelion	1	0	0	0.045287	
dandelion	1	0	0.23889	0.034469	
dandelion	1	0	0.26111	0.033462	
dandelion	1	0	0.27222	0.032959	
dandelion	1	0	0.28889	0.032204	
dandelion	1	0	0.29444	0.031953	
dandelion	1	0	0.3	0.031701	
dandelion	1	0	0.31111	0.031198	

The table in `Metrics` now includes the `PositivePredictiveValue` and `NegativePredictiveValue` variables in the last two columns, in the order you specified. Note that the positive predictive value ( $PPV = TP / (TP + FP)$ ) is NaN for the reject-all threshold (largest threshold), and the negative predictive value ( $NPV = TN / (TN + FN)$ ) is NaN for the accept-all threshold (lowest threshold). TP, FP, TN, and FN represent the number of true positives, false positives, true negatives, and false negatives, respectively.

## Input Arguments

### rocObj — Object evaluating classification performance

rocmetrics object

Object evaluating classification performance, specified as a rocmetrics object.

### metrics — Additional model performance metrics

character vector | string array | function handle | cell array

Additional model performance metrics to compute, specified as a character vector or string scalar of the built-in metric name, string array of names, function handle (@metricName), or cell array of names or function handles. A rocmetrics object always computes the false positive rates (FPR) and the true positive rates (TPR) to obtain a ROC curve. Therefore, you do not have to specify to compute FPR and TPR.

- Built-in metrics — Specify one of the following built-in metric names by using a character vector or string scalar. You can specify more than one by using a string array.

Name	Description
"TruePositives" or "tp"	Number of true positives (TP)
"FalseNegatives" or "fn"	Number of false negatives (FN)
"FalsePositives" or "fp"	Number of false positives (FP)
"TrueNegatives" or "tn"	Number of true negatives (TN)
"SumOfTrueAndFalsePositives" or "tp+fp"	Sum of TP and FP
"RateOfPositivePredictions" or "rpp"	Rate of positive predictions (RPP), $(TP+FP)/(TP+FN+FP+TN)$
"RateOfNegativePredictions" or "rnp"	Rate of negative predictions (RNP), $(TN+FN)/(TP+FN+FP+TN)$
"Accuracy" or "accu"	Accuracy, $(TP+TN)/(TP+FN+FP+TN)$
"FalseNegativeRate", "fnr", or "miss"	False negative rate (FNR), or miss rate, $FN/(TP+FN)$
"TrueNegativeRate", "tnr", or "spec"	True negative rate (TNR), or specificity, $TN/(TN+FP)$
"PositivePredictiveValue", "ppv", or "prec"	Positive predictive value (PPV), or precision, $TP/(TP+FP)$
"NegativePredictiveValue" or "npv"	Negative predictive value (NPV), $TN/(TN+FN)$

Name	Description
"ExpectedCost" or "ecost"	<p>Expected cost, <math>(TP * \text{cost}(P P) + FN * \text{cost}(N P) + FP * \text{cost}(P N) + TN * \text{cost}(N N)) / (TP + FN + FP + TN)</math>, where <code>cost</code> is a 2-by-2 misclassification cost matrix containing <math>[\text{cost}(N P); \text{cost}(P N), 0]</math>. <code>cost(N P)</code> is the cost of misclassifying a positive class (P) as a negative class (N), and <code>cost(P N)</code> is the cost of misclassifying a negative class as a positive class.</p> <p>The software converts the K-by-K matrix specified by the <code>Cost</code> name-value argument of <code>rocmetrics</code> to a 2-by-2 matrix for each one-versus-all binary problem. For details, see "Misclassification Cost Matrix".</p>

The software computes the scale vector using the prior class probabilities (`Prior`) and the number of classes in `Labels`, and then scales the performance metrics according to this scale vector. For details, see "Performance Metrics".

- Custom metric — Specify a custom metric by using a function handle. A custom function that returns a performance metric must have this form:

```
metric = customMetric(C, scale, cost)
```

- The output argument `metric` is a scalar value.
- A custom metric is a function of the confusion matrix (`C`), scale vector (`scale`), and cost matrix (`cost`). The software finds these input values for each one-versus-all binary problem. For details, see "Performance Metrics".
  - `C` is a 2-by-2 confusion matrix consisting of  $[TP, FN; FP, TN]$ .
  - `scale` is a 2-by-1 scale vector.
  - `cost` is a 2-by-2 misclassification cost matrix.

The software does not support cross-validation for a custom metric. Instead, you can specify to use bootstrap when you create a `rocmetrics` object.

Note that the positive predictive value (PPV) is NaN for the reject-all threshold for which  $TP = FP = 0$ , and the negative predictive value (NPV) is NaN for the accept-all threshold for which  $TN = FN = 0$ . For more details, see "Thresholds, Fixed Metric, and Fixed Metric Values".

Example: `["Accuracy", "PositivePredictiveValue"]`

Example: `{"Accuracy", @m1, @m2}` specifies the accuracy metric and the custom metrics `m1` and `m2` as additional metrics. `addMetrics` stores the custom metric values as variables named `CustomMetric1` and `CustomMetric2` in the `Metrics` property.

Data Types: `char` | `string` | `cell` | `function_handle`

## Output Arguments

### UpdatedROCObj — Object evaluating classification performance

`rocmetrics` object

Object evaluating classification performance, returned as a `rocmetrics` object.

To overwrite the input argument `rocObj`, assign the output of `addMetrics` to `rocObj`:

```
rocObj = addMetrics(rocObj,metrics);
```

## **Version History**

**Introduced in R2022b**

## **See Also**

`rocmetrics` | `average` | `plot`

## **Topics**

“ROC Curve and Performance Metrics”

“Compare Deep Learning Models Using ROC Curves”

## 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: [24x24x20x20 dlarray]
fc_1_B: [20x1 dlarray]
fc_2_W: [1x1x20x20 dlarray]
fc_2_B: [20x1 dlarray]
fc_3_W: [1x1x20x10 dlarray]
fc_3_B: [10x1 dlarray]

```

`params.Nonlearnables`

```

ans = struct with fields:
    ConvStride1004: [2x1 dlarray]
    ConvDilationFactor1005: [2x1 dlarray]
    ConvPadding1006: [4x1 dlarray]
    ConvStride1007: [2x1 dlarray]
    ConvDilationFactor1008: [2x1 dlarray]
    ConvPadding1009: [4x1 dlarray]
    ConvStride1010: [2x1 dlarray]
    ConvDilationFactor1011: [2x1 dlarray]
    ConvPadding1012: [4x1 dlarray]
    ConvStride1013: [2x1 dlarray]
    ConvDilationFactor1014: [2x1 dlarray]
    ConvPadding1015: [4x1 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, 'Nonlearnable');
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(Var
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`.

## Version History

**Introduced in R2020b**

### See Also

[importONNXFunction](#) | [ONNXParameters](#) | [removeParameter](#)

# 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:
```

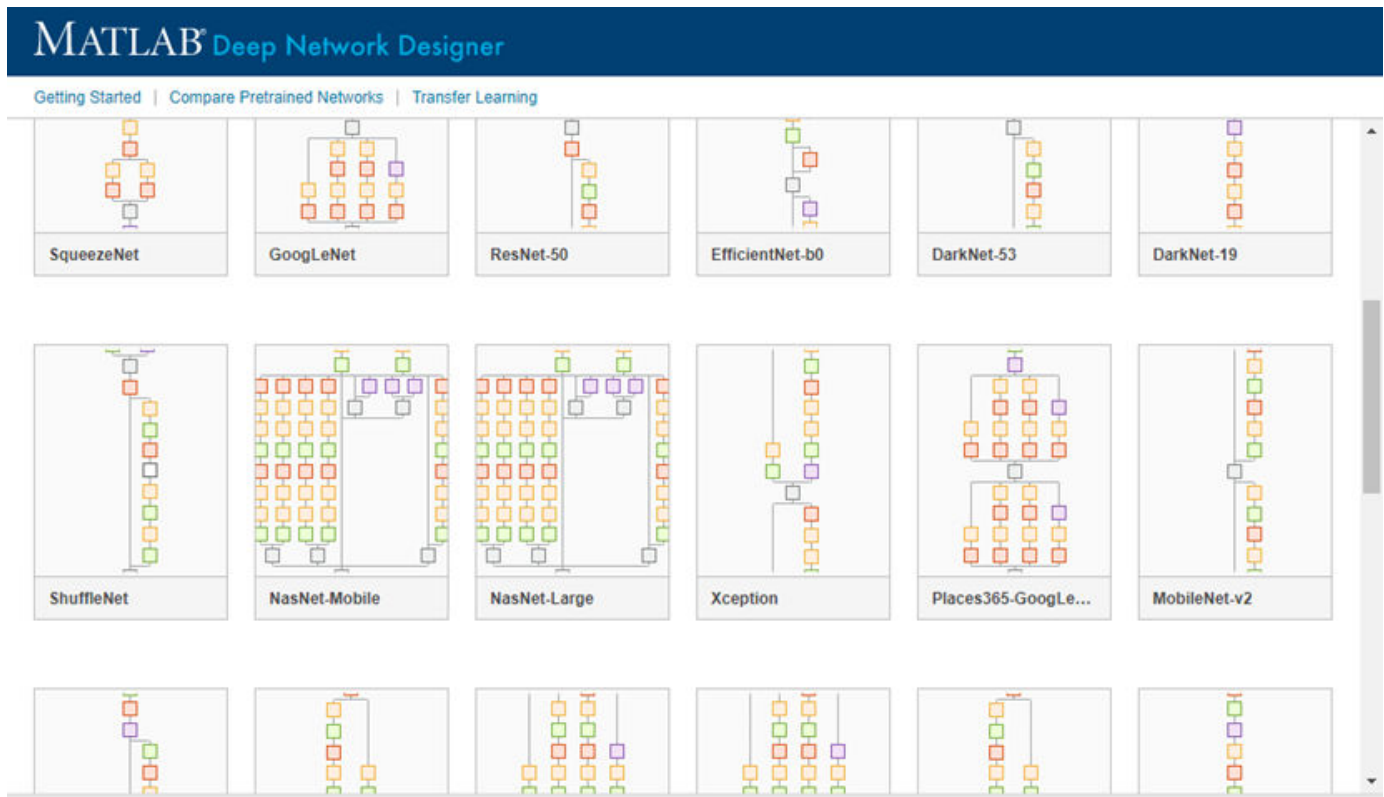
```
Layers: [25x1 nnet.cnn.layer.Layer]
```

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 neural networks in Deep Network Designer by clicking **New**.



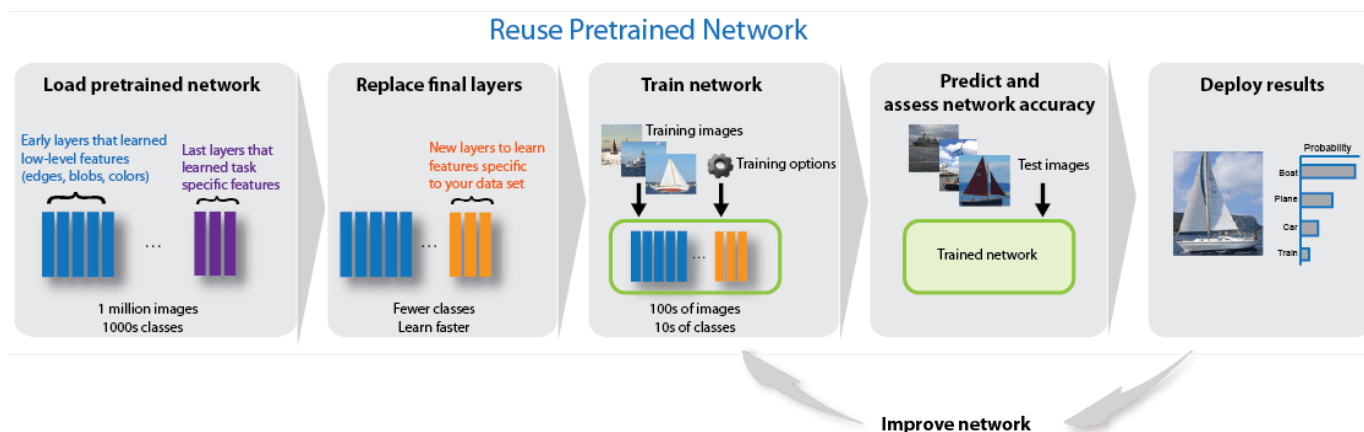
If you need to download a neural network, pause on the desired neural network and click **Install** to open the Add-On Explorer.

## 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
```



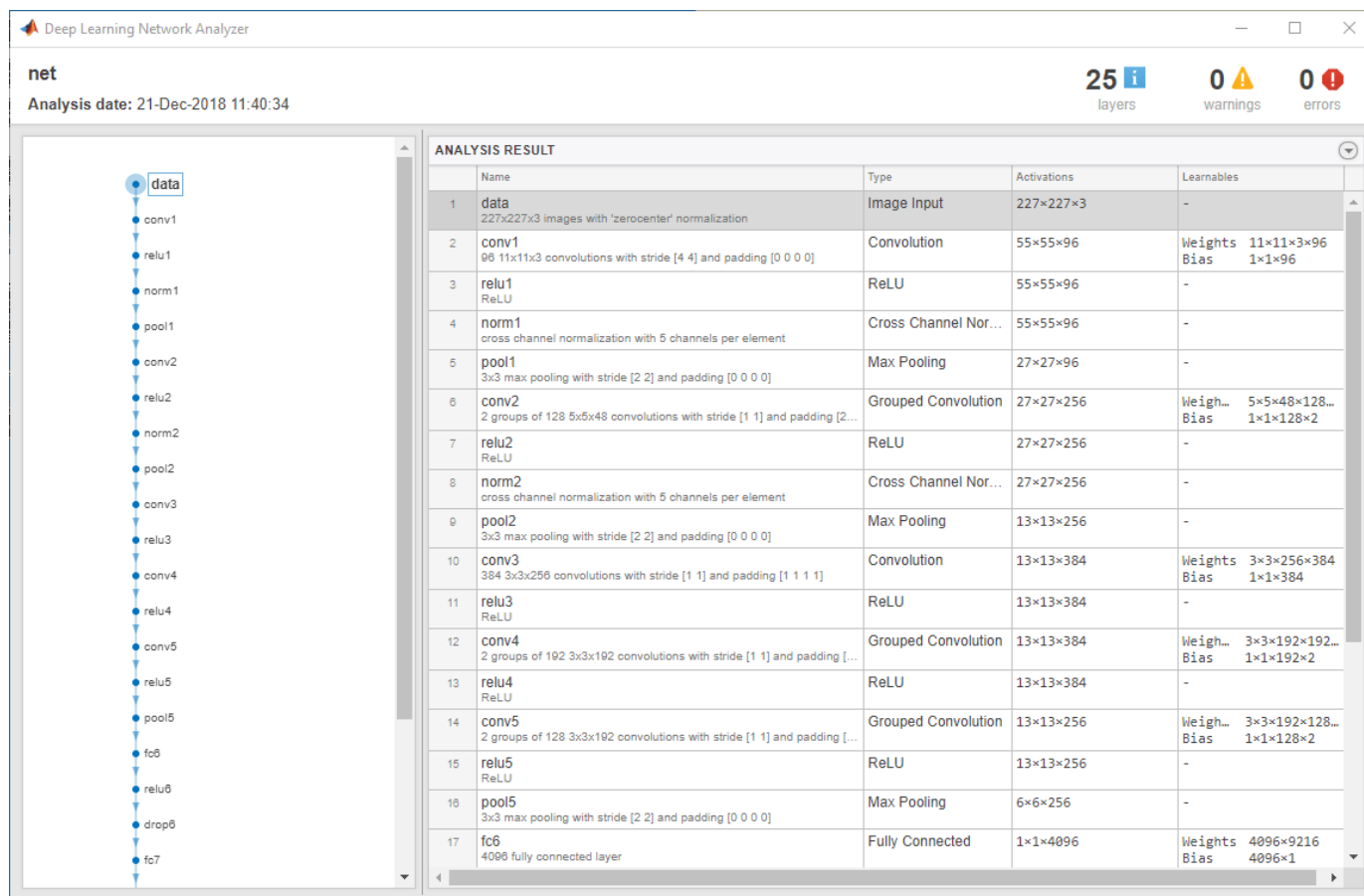
## Load Pretrained Network

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 = 1x3
```

```
    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 datastore have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels 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);
```

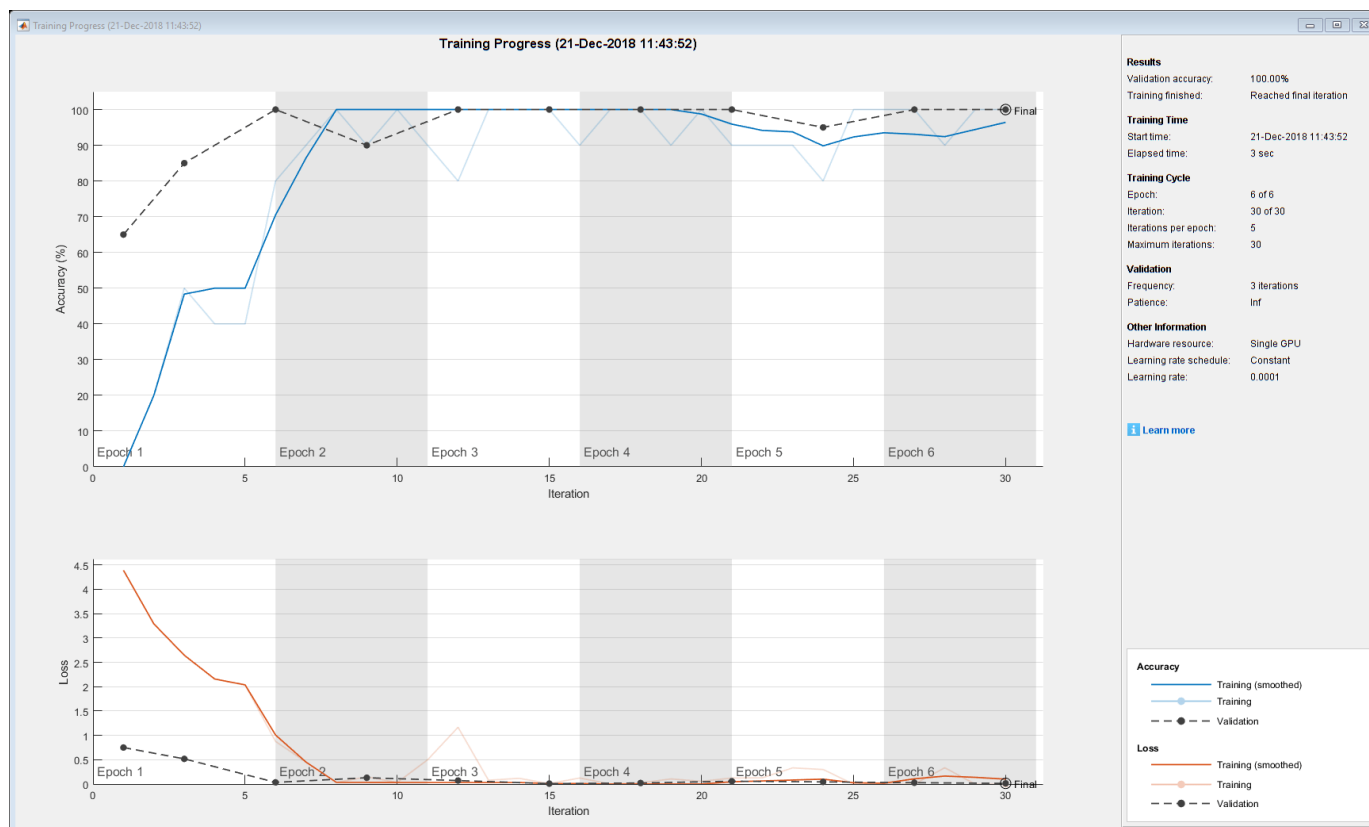
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate 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 Computing Requirements” (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);
```





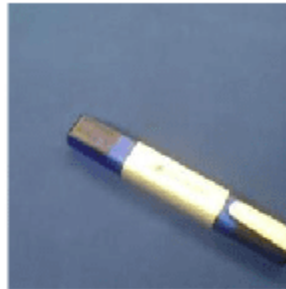
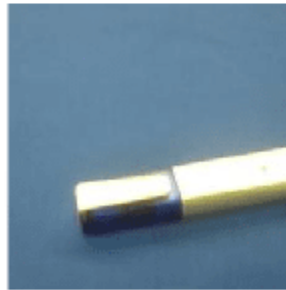
## Classify Validation Images

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
```

**MathWorks Playing Cards****MathWorks Screwdriver****MathWorks Cap****MathWorks Screwdriver**

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)
```

bell pepper



## 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' normalization
2	'conv1'	Convolution	96 11x11x3 convolutions with stride [4 4] and padding
3	'relu1'	ReLU	ReLU
4	'norm1'	Cross Channel Normalization	cross channel normalization with 5 channels per group
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 [3 3] and padding
7	'relu2'	ReLU	ReLU
8	'norm2'	Cross Channel Normalization	cross channel normalization with 5 channels per group
9	'pool2'	Max Pooling	3x3 max pooling with stride [2 2] and padding
10	'conv3'	Convolution	384 3x3x256 convolutions with stride [1 1] and padding
11	'relu3'	ReLU	ReLU
12	'conv4'	Grouped Convolution	2 groups of 192 3x3x192 convolutions with stride [1 1] and padding
13	'relu4'	ReLU	ReLU
14	'conv5'	Grouped Convolution	2 groups of 128 3x3x192 convolutions with stride [1 1] and padding
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 classes

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 = 1x3
```

```
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 datstores 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.

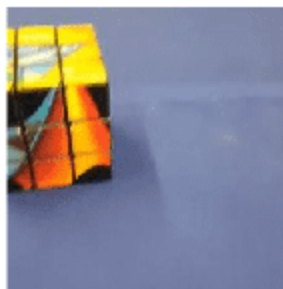
```
YPred = predict(mdl,featuresTest);
```

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
```



**MathWorks Cap****MathWorks Cube****MathWorks Playing Cards****MathWorks Screwdriver**

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.

## Version History

Introduced in R2017a

## 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](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”

# analyzeNetwork

Analyze deep learning network architecture

## Syntax

```
analyzeNetwork(layers)
analyzeNetwork(layers, TargetUsage=target)
analyzeNetwork(layers, X1, ..., Xn, TargetUsage="dlnetwork")
analyzeNetwork(net)
analyzeNetwork(net, X1, ..., Xn)
```

## 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**.

---

`analyzeNetwork(layers)` analyzes the network layers given by the layer array or layer graph `layers` and also detects errors and issues for `trainNetwork` workflows. The function displays an interactive visualization of the network architecture and provides detailed information about the network layers. The layer information includes the layer type, the size and format of the layer activations, and the size and number of learnable and state parameters.

Each activation dimension has one of the following labels: S (spatial), C (channel), B (batch), T (time or sequence), or U (unspecified).

`analyzeNetwork(layers, TargetUsage=target)` analyzes the network layers given by the layer array or layer graph `layers` for the specified target workflow. Use this syntax when analyzing the layers for `dlnetwork` workflows.

`analyzeNetwork(layers, X1, ..., Xn, TargetUsage="dlnetwork")` analyzes the layer array or layer graph `layers` using the example networks inputs `X1, ..., Xn`. The software propagates the example inputs through the network to determine the size and format of layer activations, the size and number of learnable and state parameters, and the total number of learnables. Use this syntax to analyze a network that has one or more inputs that are not connected to an input layer.

`analyzeNetwork(net)` analyzes the `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object `net`.

`analyzeNetwork(net, X1, ..., Xn)` analyzes the `dlnetwork` object `net` using example networks inputs `X1, ..., Xn`. The software propagates the example inputs through the network to determine the size and format of layer activations, the size and number of learnable and state parameters, and the total number of learnables. Use this syntax to analyze an uninitialized `dlnetwork` that has one or more inputs that are not connected to an input layer.

## Examples

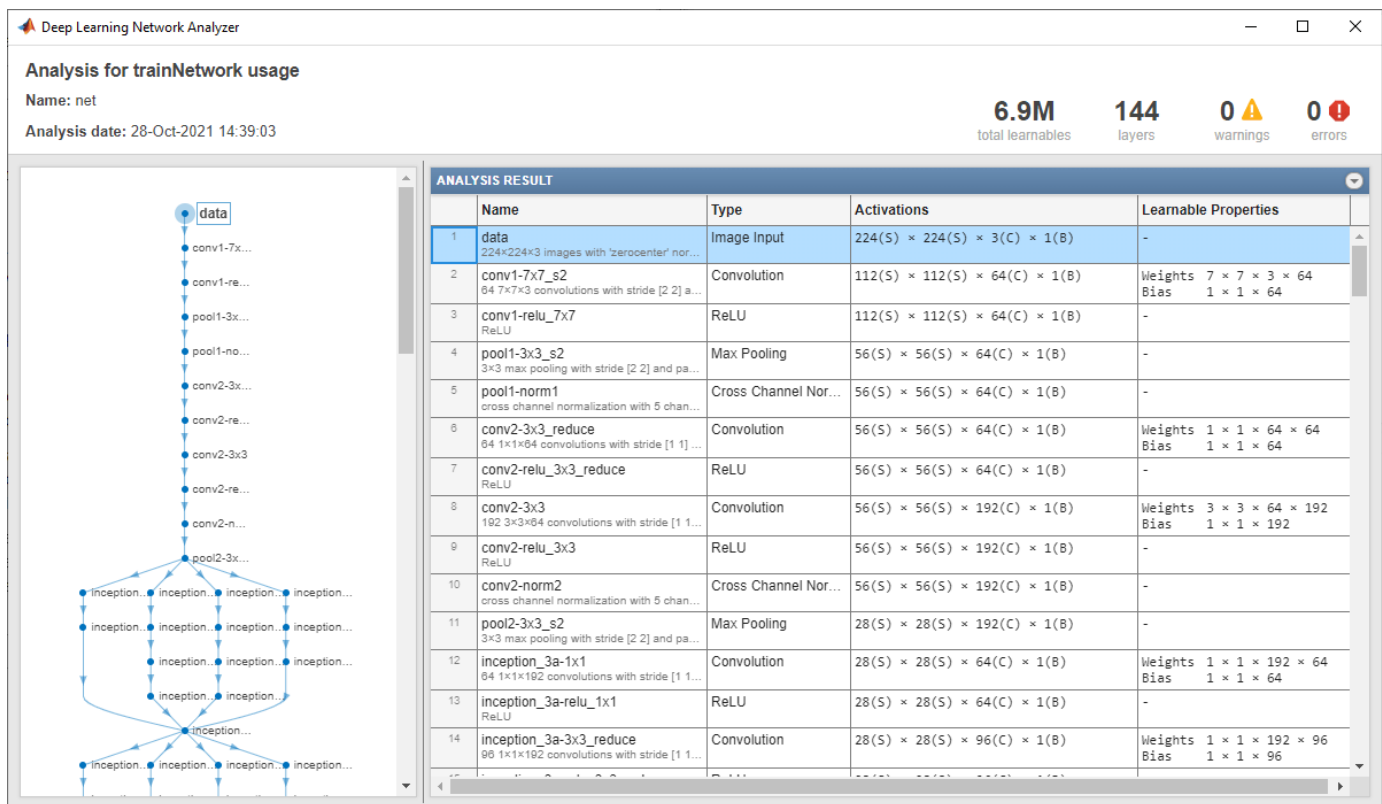
### Analyze Trained Network

Load a pretrained GoogLeNet convolutional neural network.

```
net = googlenet;
```

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

```
analyzeNetwork(net)
```



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. Each activation dimension has one of the following labels:

- S — Spatial
- C — Channel
- B — Batch observations
- T — Time or sequence
- U — Unspecified

View the dimension labels to understand how data propagates through the network and how the layers modify the size and layout of activations.

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.

The Deep Learning Network Analyzer shows the total number of learnable parameters in the network, to one decimal place. To see the exact number of learnable parameters, pause on **total learnables**. To show the number of learnable parameters in each layer, click the arrow in the top-right corner of the layer table and select **Number of 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 number of learnable parameters.

### Fix Errors in Network Architecture

Create a simple convolutional network with some skip connections. Create a layer array containing the main branch of the network.

```
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="add_1")
    convolution2dLayer(3,16,Padding="same",Stride=2)
    reluLayer
    additionLayer(3,Name="add_2")
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

lgraph = layerGraph(layers);
```

Convert the layer array to a layer graph and add the skip connections. One of the skip connections contains a single 1-by-1 convolutional layer `conv_skip`.

```
layer = convolution2dLayer(1,16,Stride=2,Name="conv_skip");
lgraph = addLayers(lgraph,layer);
lgraph = connectLayers(lgraph,"relu_1","add_1/in2");
lgraph = connectLayers(lgraph,"add_1","add_2/in2");
```

Analyze the network architecture using the `analyzeNetwork` function. The function finds issues with three layers in the network.

```
analyzeNetwork(lgraph)
```

Deep Learning Network Analyzer

Analysis for trainNetwork usage  
Name: lgraph  
Analysis date: 28-Oct-2021 13:48:57

3.5k total learnables    13 layers    0 warnings    3 errors

**ISSUES**

Found in	Message
conv_skip	Disconnected layers. All layers in the layer graph must be connected. Detected disconnected layers: layer 'conv_skip'
add_2	Unconnected input. Each layer input must be connected to the output of another layer.
add_1	Input size mismatch. Size of input to this layer is different from the expected input size. Inputs to this layer: from layer 'relu_2' (size 16(S) × 16(S) × 16(C) × 1(B)) from layer 'relu_1' (size 32(S) × 32(S) × 16(C) × 1(B))

**ANALYSIS RESULT**

Name	Type	Activations	Learnable Properties
1 imageinput 32×32×3 images with 'zerocenter' norm...	Image Input	32(S) × 32(S) × 3(C) × 1(B)	-
2 conv_1 16 5×5 convolutions with stride [1 1] and...	Convolution	32(S) × 32(S) × 16(C) × 1(B)	Weights 5 × 5 × 3 × 16 Bias 1 × 1 × 16
3 relu_1 ReLU	ReLU	32(S) × 32(S) × 16(C) × 1(B)	-
4 conv_2 16 3×3 convolutions with stride [2 2] and...	Convolution	16(S) × 16(S) × 16(C) × 1(B)	Weights 3 × 3 × 16 × 16 Bias 1 × 1 × 16
5 relu_2 ReLU	ReLU	16(S) × 16(S) × 16(C) × 1(B)	-
6 add_1 Element-wise addition of 2 inputs	Addition	-	-
7 conv_3 16 3×3 convolutions with stride [2 2] and...	Convolution	-	Weights Bias
8 relu_3 ReLU	ReLU	-	-
9 add_2 Element-wise addition of 3 inputs	Addition	-	-
10 fc 10 fully connected layer	Fully Connected	-	Weights Bias
11 softmax softmax	Softmax	-	-
12 classoutput crossentropyex	Classification Output	-	-
13 conv_skip 16 1×1 convolutions with stride [2 2] and...	Convolution	-	Weights Bias

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

- The layer `conv_skip` is not connected to the rest of the network. It should be a part of the shortcut connection between the `add_1` and `add_2` layers. To fix this error, connect `add_1` to `conv_skip` and `conv_skip` to `add_2`.
- The `add_2` layer is specified to have three inputs, but the layer 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 `add_1` 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 `relu_1`, remove the downsampling by removing the `Stride` value of the `conv_2` layer.

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")
    reluLayer
    additionLayer(2,Name="add_1")
```

```

convolution2dLayer(3,16,Padding="same",Stride=2)
reluLayer
additionLayer(2,Name="add_2")
fullyConnectedLayer(10)
softmaxLayer
classificationLayer];

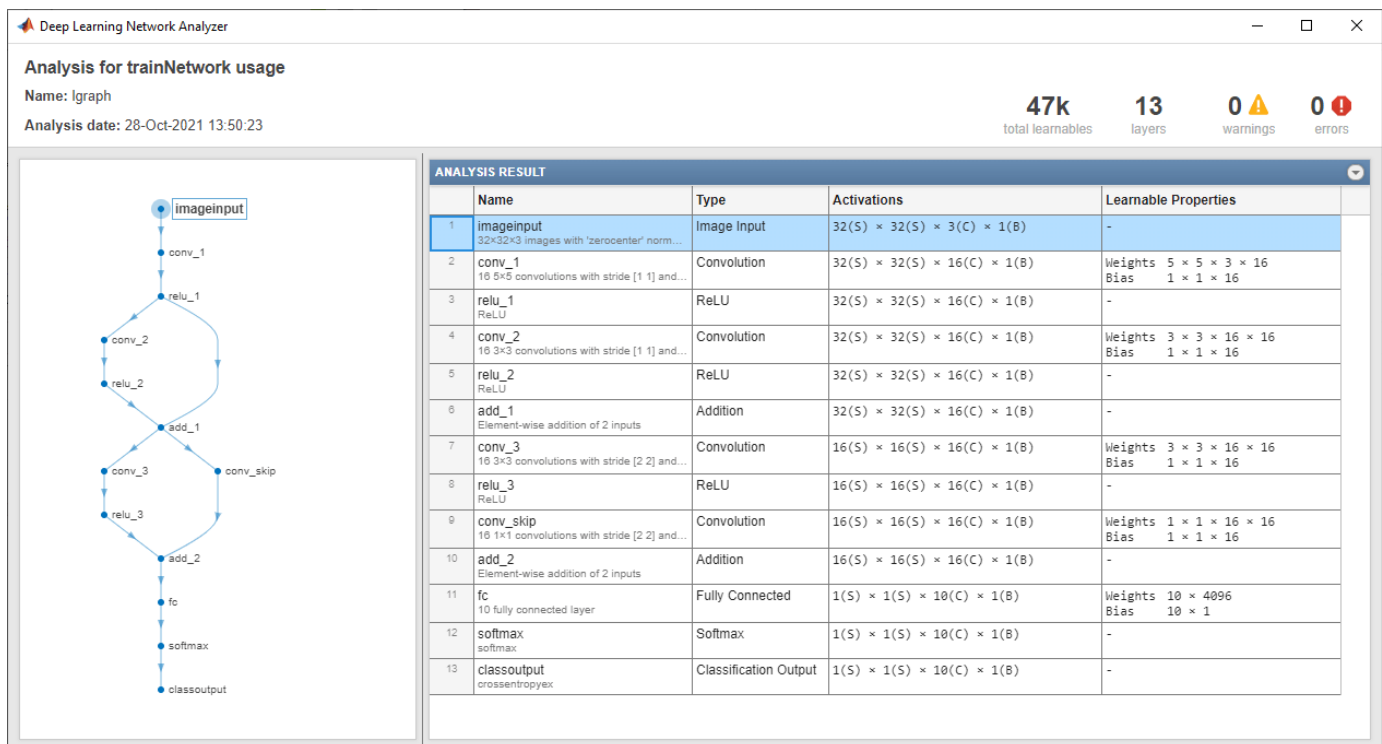
lgraph = layerGraph(layers);

layer = convolution2dLayer(1,16,Stride=2,Name="conv_skip");
lgraph = addLayers(lgraph,layer);
lgraph = connectLayers(lgraph,"relu_1","add_1/in2");
lgraph = connectLayers(lgraph,"add_1","conv_skip");
lgraph = connectLayers(lgraph,"conv_skip","add_2/in2");

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

analyzeNetwork(lgraph)

```



## Analyze Layers for Custom Training Loop

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

```

layers = [
    imageInputLayer([28 28 1],Normalization="none")
    convolution2dLayer(5,20,Padding="same")

```

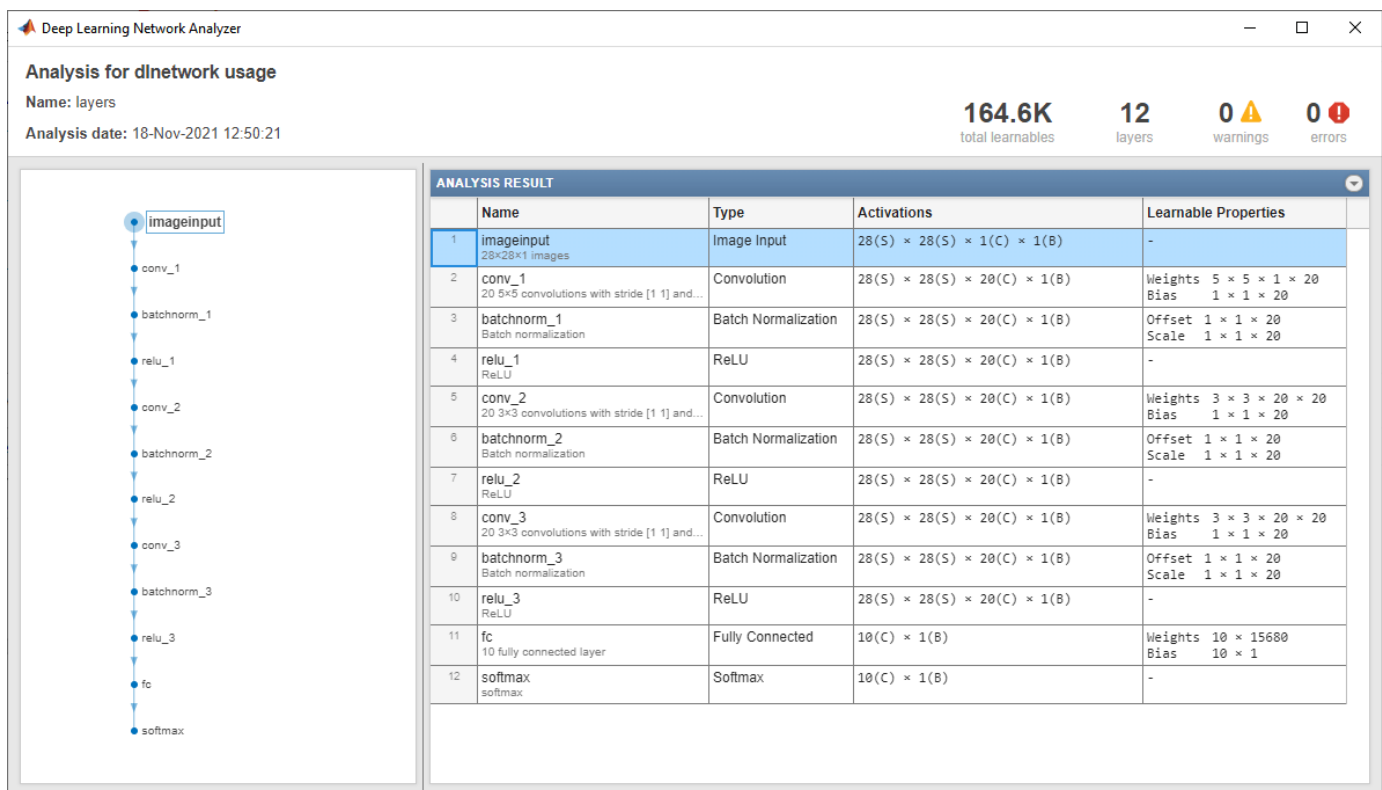
```

batchNormalizationLayer
reluLayer
convolution2dLayer(3,20,Padding="same")
batchNormalizationLayer
reluLayer
convolution2dLayer(3,20,Padding="same")
batchNormalizationLayer
reluLayer
fullyConnectedLayer(10)
softmaxLayer];

```

Analyze the layer array using the `analyzeNetwork` function and set the `TargetUsage` option to "dlnetwork".

```
analyzeNetwork(layers,TargetUsage="dlnetwork")
```



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

### 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 by setting the `TargetUsage` option to "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;
inputSize = [64 64 3];

layersBranch1 = [
    convolution2dLayer(3,6*numFilters,Padding="same",Stride=2)
    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="conv_branch")
    groupNormalizationLayer("all-channels",Name="gn_branch")];

lgraph = layerGraph(layersBranch1);
lgraph = addLayers(lgraph,layersBranch2);
lgraph = connectLayers(lgraph,"gn_branch","add/in2");

```

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

```
net = dlnetwork(lgraph,Initialize=false);
```

View the network input names.

```
net.InputNames
```

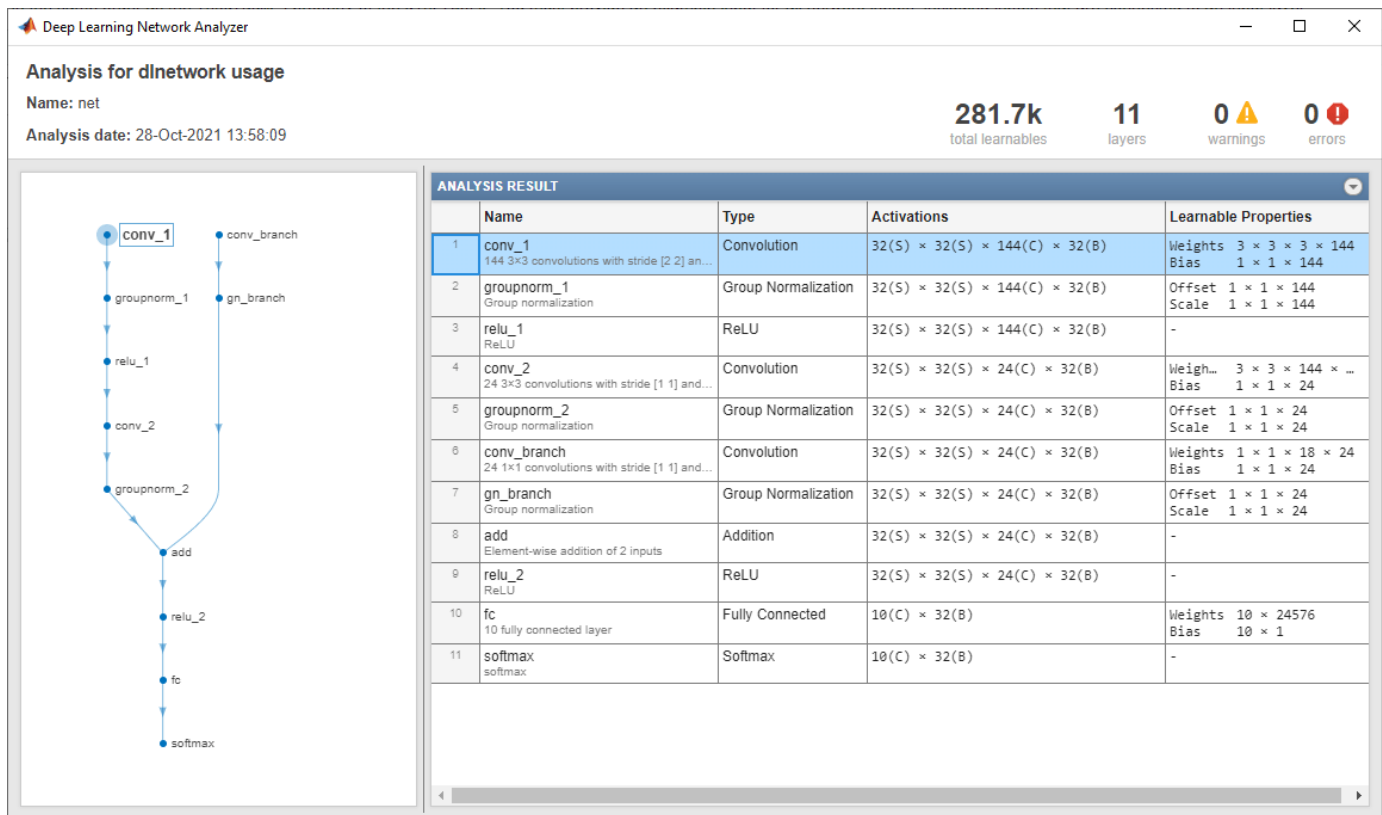
```
ans = 1x2 cell
    {'conv_1'}    {'conv_branch'}
```

Create example network inputs of the same size and format as typical inputs for this network using random `dldarray` objects. 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 "conv\_branch".

```
X1 = dldarray(rand([inputSize 32]),"SSCB");
X2 = dldarray(rand([32 32 18 32]),"SSCB");
```

Analyze the network. Provide the example inputs in the same order as the `InputNames` property of the `dlnetwork`. You must provide an example input for all network inputs, including inputs that are connected to an input layer.

```
analyzeNetwork(net,X1,X2)
```



## Input Arguments

### 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”.

### net — Deep learning network

SeriesNetwork object | DAGNetwork object | dlnetwork object

Deep learning network, specified as a SeriesNetwork, DAGNetwork, or 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.

**X1, . . . , Xn — Example network inputs**`dLarray`

Example network inputs, specified as formatted `dLarray` objects. The software propagates the example inputs through the network to determine the size and format of layer activations, the size and number of learnable and state parameters, and the total number of learnables.

Use example inputs when you want to analyze a network that does not have any input layers or that has inputs that are not connected 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.

## Version History

**Introduced in R2018a**

### See Also

**Deep Network Designer** | `SeriesNetwork` | `DAGNetwork` | `LayerGraph` | `trainNetwork` | `plot` | `assembleNetwork` | `summary`

### Topics

"Create Simple Deep Learning Neural 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"

## 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
```

Create two Gaussian noise layers with the same configurations as the imported Keras layers using the helper `gaussianNoiseLayer` function.

```
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 — Neural network layers

Layer array | LayerGraph object

Neural network layers, specified as a Layer array or a LayerGraph object.

To create a neural network with all layers connected sequentially, you can use a `Layer` array as the input argument. In this case, the returned neural network is a `SeriesNetwork` object.

A directed acyclic graph (DAG) neural network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG neural network, specify the neural network architecture as a `LayerGraph` object and then use that layer graph as the input argument to `assembleNetwork`.

The `assembleNetwork` function supports neural networks with at most one sequence input layer.

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.

## Version History

**Introduced in R2018b**

### 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”

# attention

Dot-product attention

## Syntax

```
Y = attention(queries, keys, values, numHeads)
[Y, weights] = attention(queries, keys, values, numHeads)
[Y, weights] = attention(queries, keys, values, numHeads, DataFormat=FMT)
[Y, weights] = attention(queries, keys, values, numHeads, Name=Value)
```

## Description

The attention operation focuses on parts of the input using weighted multiplication operations.

`Y = attention(queries, keys, values, numHeads)` applies the dot-product attention operation to the specified queries, keys, and values using the number of attention heads `numHeads`. The queries input argument must be a formatted `darray` object.

`[Y, weights] = attention(queries, keys, values, numHeads)` applies the dot-product attention operation and also returns the attention weights..

`[Y, weights] = attention(queries, keys, values, numHeads, DataFormat=FMT)` applies the dot-product attention operation to the unformatted `darray` object `queries` with format specified by `FMT`. For example, `DataFormat="CBT"` specifies data in the format "CBT" (channel, batch, time).

`[Y, weights] = attention(queries, keys, values, numHeads, Name=Value)` specifies additional options using one or more name-value arguments. For example, `DropoutProbability=0.01` specifies a dropout probability of 0.01.

## Examples

### Apply Attention Operation

Specify the sizes of the queries, keys, and values.

```
querySize = 100;
valueSize = 120;
numQueries = 64;
numValues = 80;
numObservations = 32;
```

Create random arrays containing the queries, keys, and values. For the queries, specify the `darray` format "CBT" (channel, batch, time).

```
queries = darray(rand(querySize, numObservations, numQueries), "CBT");
keys = darray(rand(querySize, numObservations, numValues));
values = darray(rand(valueSize, numObservations, numValues));
```

Specify the number of attention heads.

```
numHeads = 5;
```

Apply the attention operation.

```
[Y,weights] = attention(queries,keys,values,numHeads);
```

View the sizes and format of the output.

```
size(Y)
```

```
ans = 1×3
```

```
    120    32    64
```

```
dims(Y)
```

```
ans =  
'CBT'
```

View the sizes and format of the weights.

```
size(weights)
```

```
ans = 1×4
```

```
    80    64     5    32
```

```
dims(weights)
```

```
ans =
```

```
0×0 empty char array
```

### Create Multihead Self Attention Function

You can use the `attention` function to implement the multihead self attention operation [1] that focuses on parts of the input.

Create the `multiheadSelfAttention` function, listed in the Multihead Self Attention Function on page 1-167 section of the example. The `multiheadSelfAttention` function takes as input the data `X`, the number of heads, and the learnable weights for the queries, keys, values, and output data, and returns the multihead attention values.

The `X` input must be an unformatted `darray` object, where the first dimension corresponds to the input channels, the second dimension corresponds to the time or spatial dimension, and the third dimension corresponds to the batch dimension.

Create an array of sequence data.

```
numChannels = 10;
```

```
numObservations = 128;
```

```
numTimeSteps = 100;
```

```
X = rand(numChannels,numObservations,numTimeSteps);
```



```
X = dlarray(X);
size(X)

ans = 1×3

    10    128    100
```

Specify the number of heads for multihead attention.

```
numHeads = 8;
```

Initialize the learnable parameters for multihead attention.

- The learnable query, key, and value weights must be  $(\text{numChannels} \times \text{numHeads})$ -by- $\text{numChannels}$  arrays.
- The learnable output weights must be a  $(\text{numChannels} \times \text{numHeads})$ -by- $(\text{numChannels} \times \text{numHeads})$  array.

```
outputSize = numChannels*numHeads;
```

```
WQ = rand(outputSize,numChannels);
WK = rand(outputSize,numChannels);
WV = rand(outputSize,numChannels);
W0 = rand(outputSize,outputSize);
```

Apply the multihead self attention operation.

```
Y = multiheadSelfAttention(X,numHeads,WQ,WK,WV,W0);
```

View the size of the output. The output has size  $(\text{numChannels} \times \text{numHeads})$ -by- $\text{numObservations}$ -by- $(\text{numTimeSteps})$ .

```
size(Y)

ans = 1×3

    80    128    100
```

### Multihead Self Attention Function

The `multiheadSelfAttention` function takes as input the data `X`, the number of heads, and the learnable weights for the queries, keys, values, and output data, and returns the multihead attention values.

- The `X` input must be an unformatted `dlarray` object, where the first dimension corresponds to the input channels, the second dimension corresponds to the time or spatial dimension, and the third dimension corresponds to the batch dimension.
- The learnable query, key, and value weight matrices are  $(\text{numChannels} \times \text{numHeads})$ -by- $\text{numChannels}$  matrices.
- The learnable output weights matrix is a  $(\text{numChannels} \times \text{numHeads})$ -by- $(\text{numChannels} \times \text{numHeads})$  matrix.

```
function Y = multiheadSelfAttention(X,numHeads,WQ,WK,WV,W0)
```

```
queries = pagetimes(WQ,X);
```

```
keys = pagetimes(WK,X);
values = pagetimes(WV,X);

A = attention(queries,keys,values,numHeads,DataFormat="CTB");

Y = pagetimes(W0,A);

end
```

### Create Luong Attention Function

You can use the `attention` function to create a function that applies the Luong attention operation to its input. Create the `luongAttention` function, listed at the end of the example, that applies the Luong attention operation.

Specify the array sizes.

```
numHiddenUnits = 100;
latentSize = 16;
```

Create random arrays containing the input data.

```
hiddenState = dlarray(rand(numHiddenUnits,1));
Z = dlarray(rand(latentSize,1));
weights = dlarray(rand(numHiddenUnits,latentSize));
```

Apply the `luongAttention` function.

```
[context,attentionScores] = luongAttention(hiddenState,Z,weights);
```

View the sizes of the outputs.

```
size(context)
```

```
ans = 1×2
    16     1
```

```
size(attentionScores)
```

```
ans = 1×2
     1     1
```

### Luong Attention Function

The `luongAttention` function returns the context vector and attention scores according to the Luong "general" scoring [2]. This operation is equivalent to dot-product attention with queries, keys, and values specified as the hidden state, the weighted latent representation, and the latent representation, respectively.

```
function [context,attentionScores] = luongAttention(hiddenState,Z,weights)

numHeads = 1;
queries = hiddenState;
```

```
keys = pagetimes(weights,Z);
values = Z;
```

```
[context,attentionScores] = attention(queries,keys,values,numHeads,Scale=1,DataFormat="CBT");
```

```
end
```

## Input Arguments

### queries – Queries

`darray` object

Queries, specified as a `darray` object.

`queries` can have at most one "S" (spatial) or "T" (time) dimension. Any dimensions in `queries` labeled "U" (unspecified) must be singleton. If `queries` is an unformatted `darray` object, then specify the data format using the `DataFormat` option.

The size of the "C" (channel) dimension in `keys` must match the size of the corresponding dimension in `queries`.

The size of the "B" (batch) dimension in `queries`, `keys`, and `values` must match.

### keys – Keys

`darray` object | numeric array

Keys, specified as a `darray` object or a numeric array.

If `keys` is a formatted `darray` object, then its format must match the format of `queries`. If `keys` is not a formatted `darray` object, then the function uses the same format as `queries`.

The size of any "S" (spatial) or "T" (time) dimensions in `keys` must match the size of the corresponding dimension in `values`.

The size of the "C" (channel) dimension in `keys` must match the size of the corresponding dimension in `queries`.

The size of the "B" (batch) dimension in `queries`, `keys`, and `values` must match.

### values – Values

`darray` object | numeric array

Values, specified as a `darray` object or a numeric array.

If `values` is a formatted `darray` object, then its format must match the format of `queries`. Otherwise, the function uses the same format as `queries`.

The size of any "S" (spatial) or "T" (time) dimensions in `keys` must match the size of the corresponding dimension in `values`.

The size of the "B" (batch) dimension in `queries`, `keys`, and `values` must match.

### numHeads – Number of heads

positive integer

Number of heads, specified as a positive integer. The value of `numHeads` must evenly divide the size of the "C" (channel) dimension of `queries`, `keys`, and `values`.

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `attention(queries, keys, values, numHeads, DataFormat="CBT")` applies the attention operation for unformatted data and specifies the data format "CBT" (channel, batch, time).

### 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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "S" — Spatial
- "C" — Channel
- "B" — Batch (for example, samples and observations)
- "T" — Time (for example, time steps of sequences)
- "U" — Unspecified

You can use the labels "C" and "B" at most once and one dimension labeled either "S" or "T".

You must specify `DataFormat` when the input data is not a formatted `darray` object.

Data Types: `char` | `string`

### Scale — Multiplicative factor for scaled dot-product attention

"auto" (default) | numeric scalar

Multiplicative factor for scaled dot-product attention [1], specified as one of these values:

- "auto" — Multiply the dot-product by  $\lambda = \frac{1}{\sqrt{d_k}}$ , where  $d_k$  denotes the number of channels in the keys divided by the number of heads.
- Numeric scalar — Multiply the dot-product by the specified scale factor.

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

### PaddingMask — Mask indicating padding values

`darray` object | logical array | binary-valued numeric array

Mask indicating which elements of the input correspond to padding values, specified as a `darray` object, a logical array, or a binary-valued numeric array.

The function prevents and allows attention to elements of input data key-value pairs when the corresponding element in `PaddingMask` is 0 and 1, respectively.

If `PaddingMask` is a formatted `darray` object, then its format must match that of `keys`. If `PaddingMask` is not a formatted `darray` object, then the function uses the same format as `keys`. The size of the "S" (spatial), "T" (time), and "B" (batch) dimensions in `PaddingMask` must match the size of the corresponding dimensions in `keys` and `values`.

The default value is a logical array of ones with the same size as `keys`.

### AttentionMask — Attention mask

"none" (default) | "causal" | numeric array | logical array

Attention mask indicating which elements to include when applying the attention operation, specified as one of these values:

- "none" — Do not prevent attention to elements with respect to their positions. If `AttentionMask` is "none", then the software prevents attention using `PaddingMask` only.
- "causal" — Prevent elements in position  $M$  in the "S" (spatial) or "T" (time) dimension of queries from providing attention to the elements in positions  $n$ , where  $n$  is greater than  $M$  in the corresponding dimension of `keys` and `values`. Use this option for auto-regressive models.
- Logical or numeric array — Prevent attention to elements of `keys` and `values` when the corresponding element in the array is 0, where `AttentionMask` is a  $N_k$ -by- $N_q$  matrix or a  $N_k$ -by- $N_q$ -by-`numObservations` array,  $N_k$  is the size of the "S" (spatial) or "T" (time) dimension of `keys`,  $N_q$  is the size of the corresponding dimension in queries, and `numObservations` is the size of the "B" dimension in queries.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string

### DropoutProbability — Dropout probability

0 (default) | scalar in the range [0, 1)

Dropout probability for the attention weights, specified as a scalar in the range [0, 1).

Data Types: single | double

## Output Arguments

### Y — Result of attention operation

`darray` object

Result of attention operation, returned as a `darray` object.

If `queries` is a formatted `darray` object, then `Y` is a formatted `darray` object with the same dimension labels as `queries`. The size of the "C" (channel) dimension of `Y` is the same as the size of the corresponding dimension in `values`. The size of the "S" (spatial) or "T" dimension of `Y` is the same size as the corresponding dimension in queries.

If `queries` is not a formatted `darray` object, then `Y` is an unformatted `darray` object.

### weights — Attention weights

unformatted `darray` object

Attention weights, returned as an unformatted `darray` object.

`weights` is a  $N_k$ -by- $N_q$ -by-`numHeads`-by-`numObservations` array, where  $N_k$  is the size of the "S" (spatial) or "T" (time) dimension of keys,  $N_q$  is the size of the corresponding dimension in queries, and `numObservations` is the size of the "B" (batch) dimension in queries.

## Algorithms

### Dot-Product Attention

The attention operation focuses on parts of the input using weighted multiplication operations.

The single-head dot-product attention operation is given by

$$\text{attention}(Q, K, V) = \text{dropout}\left(\text{softmax}\left(\text{mask}\left(\lambda QK^T, M\right)\right), p\right)V,$$

where:

- $Q$  denotes the queries.
- $K$  denotes the keys.
- $V$  denotes the values.
- $\lambda$  denotes the scaling factor.
- $M$  is a mask array of ones and zeros.
- $p$  is the dropout probability

The mask operation includes and excludes the values of the matrix multiplication setting values of the input to  $-\infty$  for zero-valued mask elements. The mask is the union of the padding and attention masks. The softmax function normalizes the value of the input data across the channel dimension such that it sums to one. The dropout operation sets elements to zero with probability  $p$ .

### Multihead Self-Attention

The multihead self-attention operation for the input  $X$  is given by

$$\text{multiheadSelfAttention}(X, h, W^Q, W^K, W^V, W^O) = \text{concatenate}(\text{head}_1, \dots, \text{head}_h)W^O,$$

where:

- $h$  is the number of heads.
- $W^Q$  is a learnable projection matrix for the queries.
- $W^K$  is a learnable projection matrix for the keys.
- $W^V$  is a learnable projection matrix for the values.
- $W^O$  is a learnable projection matrix for the output.

Each weight matrix is composed of concatenated weight matrices  $W_i$  for each head. Each  $\text{head}_i$  denotes the output of the head operation given by

$$\text{head}_i = \text{attention}\left(XW_i^Q, XW_i^K, XW_i^V\right).$$

## Version History

Introduced in R2022b

## References

- [1] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (December 2017): 6000-6010. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>.
- [2] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. "Effective approaches to attention-based neural machine translation." *arXiv preprint arXiv:1508.04025* (2015).

## 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 these input arguments is a `gpuArray` object or a `dlarray` object with underlying data of type `gpuArray`, this function runs on the GPU.
  - `queries`
  - `keys`
  - `values`

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

### See Also

`padsequences` | `dlarray` | `dlgradient` | `dlfeval` | `lstm` | `gru` | `embed`

### Topics

"Define Custom Training Loops, Loss Functions, and Networks"

"Train Network Using Model Function"

"Sequence-to-Sequence Translation Using Attention"

"Image Captioning Using Attention"

"Multilabel Graph Classification Using Graph Attention Networks"

"Language Translation Using Deep Learning"

"List of Functions with `dlarray` Support"

## 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.

## Version History

Introduced in R2018b

## See Also

`augmentedImageDatastore` | `trainNetwork`

## Topics

“Deep Learning in MATLAB”

“Preprocess Images for Deep Learning”

# 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

```

auids = augmentedImageDatastore(outputSize,uids)
auids = augmentedImageDatastore(outputSize,X,Y)
auids = augmentedImageDatastore(outputSize,X)
auids = augmentedImageDatastore(outputSize,tbl)
auids = augmentedImageDatastore(outputSize,tbl,responseNames)
auids = augmentedImageDatastore( ___,Name,Value)

```

### Description

`auids = augmentedImageDatastore(outputSize,uids)` creates an augmented image datastore for classification problems using images from image datastore `uids`, and sets the `OutputSize` property.

`auids = 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.

`auids = 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 the `imageDatastore` function 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 NaNs, 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 NaNs.

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 NaNs. If there are NaNs 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 FileSet
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.

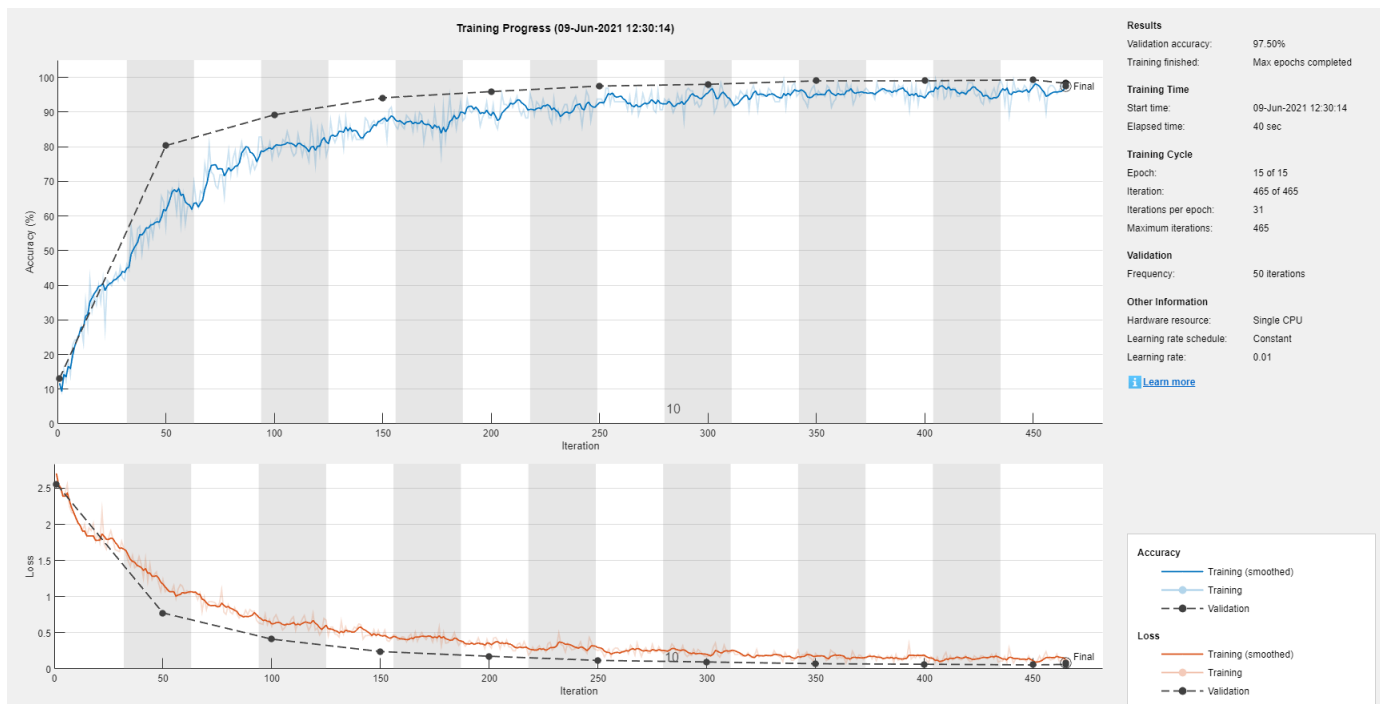
```

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.

## Version History

Introduced in R2018a

## See Also

`imageDataAugmenter` | `imageInputLayer` | `trainNetwork`

## Topics

“Deep Learning in MATLAB”

“Preprocess Images for Deep Learning”

# 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

```

auids = augmentedImageSource(outputSize,imds)
auids = augmentedImageSource(outputSize,X,Y)
auids = augmentedImageSource(outputSize,tbl)
auids = augmentedImageSource(outputSize,tbl,responseNames)
auids = augmentedImageSource( ___,Name,Value)

```

## Description

`auids = augmentedImageSource(outputSize,imds)` creates an augmented image datastore, `auids`, for classification problems using images from image datastore `imds`, with output image size `outputSize`.

`auids = 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.

`auids = augmentedImageSource(outputSize,tbl)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses.

`auids = 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`.

`auids = 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 1-181.

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(aumds, layers, opts);
```

Training on single CPU.  
Initializing image normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning 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, `aumds`.

### 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 the `imageDatastore` function 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 NaNs, 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 NaNs.

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 NaNs. If there are NaNs 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 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.

## **Version History**

**Introduced in R2017b**

### **R2018a: 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.

### **R2018a: 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`



## average

Compute performance metrics for average receiver operating characteristic (ROC) curve in multiclass problem

### Syntax

```
[FPR,TPR,Thresholds,AUC] = average(rocObj,type)
```

### Description

`[FPR,TPR,Thresholds,AUC] = average(rocObj,type)` computes the averages of performance metrics stored in the `rocmetrics` object `rocObj` for a multiclass classification problem using the averaging method specified in `type`. The function returns the average false positive rate (FPR) and the average true positive rate (TPR) for each threshold value in `Thresholds`. The function also returns AUC, the area under the ROC curve composed of FPR and TPR.

### Examples

#### Find Average ROC Curve

Compute the performance metrics for a multiclass classification problem by creating a `rocmetrics` object, and then compute the average values for the metrics by using the `average` function. Plot the average ROC curve using the outputs of `average`.

Load a sample of true labels and the prediction scores for a classification problem. For this example, there are five classes: daisy, dandelion, roses, sunflowers, and tulips. The class names are stored in `classNames`. The scores are the softmax prediction scores generated using the `predict` function. `scores` is an N-by-K array where N is the number of observations and K is the number of classes. The column order of `scores` follows the class order stored in `classNames`.

```
load('flowersDataResponses.mat')

scores = flowersData.scores;
trueLabels = flowersData.trueLabels;

classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels in `trueLabels` and the classification scores in `scores`. Specify the column order of scores using `classNames`.

```
rocObj = rocmetrics(trueLabels,scores,classNames);
```

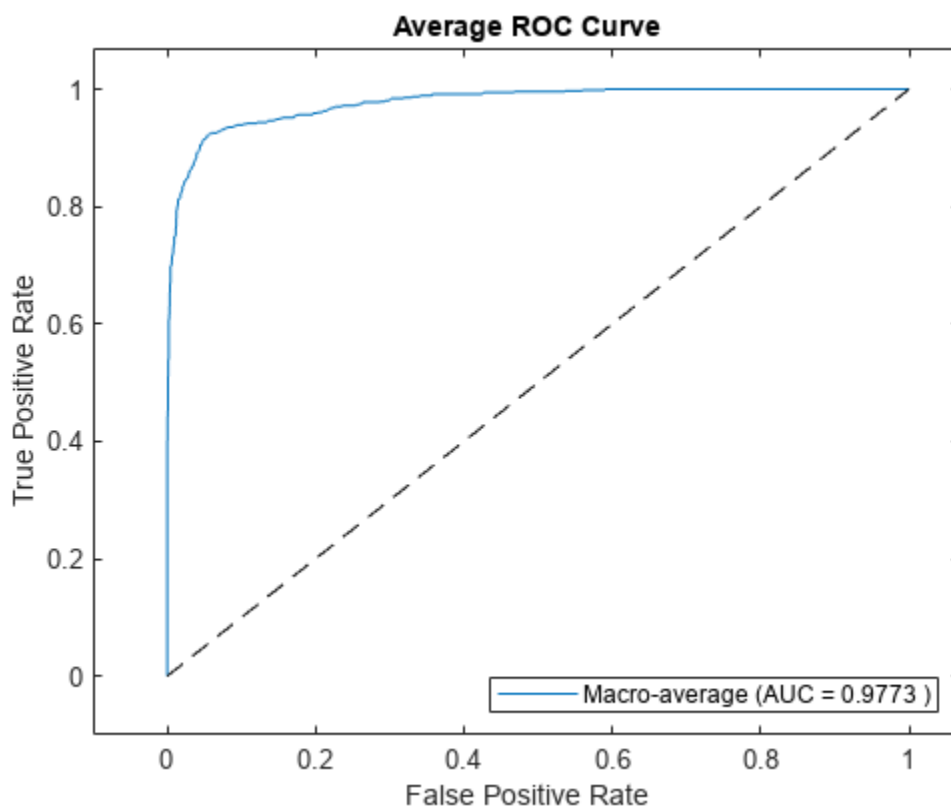
`rocmetrics` computes the FPR and TPR at different thresholds and finds the AUC value for each class.

Compute the average performance metric values, including the FPR and TPR at different thresholds and the AUC value, using the macro-averaging method.

```
[FPR,TPR,Thresholds,AUC] = average(rocObj,"macro");
```

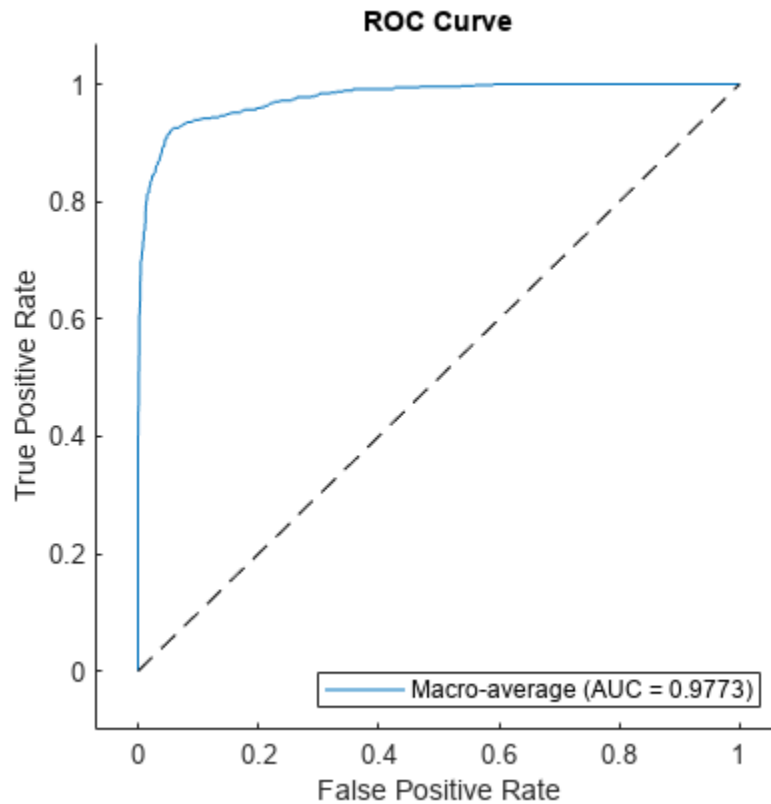
Plot the average ROC curve and display the average AUC value. Include (0,0) so that the curve starts from the origin (0,0).

```
plot([0;FPR],[0;TPR])
xlabel("False Positive Rate")
ylabel("True Positive Rate")
title("Average ROC Curve")
hold on
plot([0,1],[0,1],"k--")
legend(join(["Macro-average (AUC =",AUC,")"]), ...
       Location="southeast")
axis padded
hold off
```



Alternatively, you can create the average ROC curve by using the `plot` function. Specify `AverageROCType="macro"` to compute the metrics for the average ROC curve using the macro-averaging method.

```
plot(rocObj,AverageROCType="macro",ClassNames=[])
```



## Input Arguments

### **rocObj** — Object evaluating classification performance

rocmetrics object

Object evaluating classification performance, specified as a rocmetrics object.

### **type** — Averaging method

"micro" | "macro" | "weighted"

Averaging method, specified as "micro", "macro", or "weighted".

- "micro" (micro-averaging) — `average` finds the average performance metrics by treating all one-versus-all on page 1-195 binary classification problems as one binary classification problem. The function computes the confusion matrix components for the combined binary classification problem, and then computes the average FPR and TPR using the values of the confusion matrix.
- "macro" (macro-averaging) — `average` computes the average values for FPR and TPR by averaging the values of all one-versus-all binary classification problems.
- "weighted" (weighted macro-averaging) — `average` computes the weighted average values for FPR and TPR using the macro-averaging method and using the prior class probabilities (the `Prior` property of rocObj) as weights.

The algorithm type determines the length of the vectors for the output arguments (FPR, TPR, and Thresholds). For more details, see "Average of Performance Metrics".

Data Types: char | string

## Output Arguments

### **FPR — Average false positive rates**

numeric vector

Average false positive rates, returned as a numeric vector.

### **TPR — Average true positive rates**

numeric vector

Average true positive rates, returned as a numeric vector.

### **Thresholds — Thresholds on classification scores**

numeric vector

Thresholds on classification scores at which the function finds each of the average performance metric values (FPR and TPR), returned as a vector.

### **AUC — Area under average ROC curve**

numeric scalar

Area under the average ROC curve on page 1-195 composed of FPR and TPR, returned as a numeric scalar.

## More About

### **Receiver Operating Characteristic (ROC) Curve**

A ROC curve shows the true positive rate versus the false positive rate for different thresholds of classification scores.

The true positive rate and the false positive rate are defined as follows:

- True positive rate (TPR), also known as recall or sensitivity —  $TP / (TP + FN)$ , where TP is the number of true positives and FN is the number of false negatives
- False positive rate (FPR), also known as fallout or 1-specificity —  $FP / (TN + FP)$ , where FP is the number of false positives and TN is the number of true negatives

Each point on a ROC curve corresponds to a pair of TPR and FPR values for a specific threshold value. You can find different pairs of TPR and FPR values by varying the threshold value, and then create a ROC curve using the pairs. For each class, `rocmetrics` uses all distinct adjusted score on page 1-195 values as threshold values to create a ROC curve.

For a multiclass classification problem, `rocmetrics` formulates a set of one-versus-all on page 1-195 binary classification problems to have one binary problem for each class, and finds a ROC curve for each class using the corresponding binary problem. Each binary problem assumes one class as positive and the rest as negative.

For a binary classification problem, if you specify the classification scores as a matrix, `rocmetrics` formulates two one-versus-all binary classification problems. Each of these problems treats one class as a positive class and the other class as a negative class, and `rocmetrics` finds two ROC curves. Use one of the curves to evaluate the binary classification problem.

For more details, see “ROC Curve and Performance Metrics”.

### Area Under ROC Curve (AUC)

The area under a ROC curve (AUC) corresponds to the integral of a ROC curve (TPR values) with respect to FPR from  $FPR = 0$  to  $FPR = 1$ .

The AUC provides an aggregate performance measure across all possible thresholds. The AUC values are in the range 0 to 1, and larger AUC values indicate better classifier performance.

### One-Versus-All (OVA) Coding Design

The one-versus-all (OVA) coding design reduces a multiclass classification problem to a set of binary classification problems. In this coding design, each binary classification treats one class as positive and the rest of the classes as negative. `rocmetrics` uses the OVA coding design for multiclass classification and evaluates the performance on each class by using the binary classification that the class is positive.

For example, the OVA coding design for three classes formulates three binary classifications:

	Binary 1	Binary 2	Binary 3
Class 1	1	-1	-1
Class 2	-1	1	-1
Class 3	-1	-1	1

Each row corresponds to a class, and each column corresponds to a binary classification problem. The first binary classification assumes that class 1 is a positive class and the rest of the classes are negative. `rocmetrics` evaluates the performance on the first class by using the first binary classification problem.

## Algorithms

### Adjusted Scores for Multiclass Classification Problem

For each class, `rocmetrics` adjusts the classification scores (input argument `Scores` of `rocmetrics`) relative to the scores for the rest of the classes if you specify `Scores` as a matrix. Specifically, the adjusted score for a class given an observation is the difference between the score for the class and the maximum value of the scores for the rest of the classes.

For example, if you have  $[s_1, s_2, s_3]$  in a row of `Scores` for a classification problem with three classes, the adjusted score values are  $[s_1 - \max(s_2, s_3), s_2 - \max(s_1, s_3), s_3 - \max(s_1, s_2)]$ .

`rocmetrics` computes the performance metrics using the adjusted score values for each class.

For a binary classification problem, you can specify `Scores` as a two-column matrix or a column vector. Using a two-column matrix is a simpler option because the `predict` function of a classification object returns classification scores as a matrix, which you can pass to `rocmetrics`. If you pass scores in a two-column matrix, `rocmetrics` adjusts scores in the same way that it adjusts scores for multiclass classification, and it computes performance metrics for both classes. You can use the metric values for one of the two classes to evaluate the binary classification problem. The metric values for a class returned by `rocmetrics` when you pass a two-column matrix are equivalent to the metric values returned by `rocmetrics` when you specify classification scores for the class as a column vector.

## Alternative Functionality

- You can use the `plot` function to create the average ROC curve. The function returns a `ROCCurve` object containing the `XData`, `YData`, `Thresholds`, and `AUC` properties, which correspond to the output arguments `FPR`, `TPR`, `Thresholds`, and `AUC` of the `average` function, respectively. For an example, see “Plot ROC Curve” on page 1-1435.

## Version History

Introduced in R2022b

## References

- [1] Sebastiani, Fabrizio. "Machine Learning in Automated Text Categorization." *ACM Computing Surveys* 34, no. 1 (March 2002): 1-47.

## See Also

`rocmetrics` | `addMetrics` | `plot`

## Topics

“ROC Curve and Performance Metrics”

“Compare Deep Learning Models Using ROC Curves”

# 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (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(3,Padding=1)` creates a 1-D average 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $\text{ceil}(\text{inputSize}/\text{Stride})$ , 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 the name ' '.

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,MinLength=40)  
    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	''	1-D 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 0]
5	''	1-D 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 a single dimension.

You can describe data that flows through a deep learning layer or operation using a string of characters representing the different dimensions of the data. The formats can contain one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, you can represent vector sequence data as a 3-D array, in which the first dimension corresponds to the channel dimension, the second dimension corresponds to the batch dimension, and the third dimension corresponds to the time dimension. This representation is in the format "CBT" (channel, batch, time).

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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (spatial) dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, you can represent vector sequence data as a 3-D array, in which the first dimension corresponds to the channel dimension, the second dimension corresponds to the batch dimension, and the third dimension corresponds to the time dimension. This representation is in the format "CBT" (channel, batch, time).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `AveragePooling1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)

In `dlnetwork` objects, `AveragePooling1DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)

## Version History

Introduced in R2021b

### 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 Neural Networks"  
 "List of Deep Learning Layers"  
 "Deep Learning Tips and Tricks"

# 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 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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 the name `' '`.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D Average Pooling	2x2 average pooling with stride [1 1] and padding [0 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 '' 2-D Convolution      20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
    3 '' ReLU                  ReLU
    4 '' 2-D Average Pooling  2x2 average 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 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 '' 2-D Convolution      20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
    3 '' ReLU                  ReLU

```

4	''	2-D Average Pooling	3x2 average pooling with stride [2 2] and padding [0 0]
5	''	Fully Connected	10 fully connected layer
6	''	Softmax	softmax
7	''	Classification Output	crossentropyex

## Algorithms

### 2-D Average Pooling Layer

A 2-D average pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the average of each region.

The dimensions that the layer pools over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `AveragePooling2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSB" (spatial, spatial, batch)	"SSB" (spatial, spatial, batch)

In `dlnetwork` objects, `AveragePooling2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SBT" (spatial, batch, time)	"SBT" (spatial, batch, time)
"SSBT" (spatial, spatial, batch, time)	"SSBT" (spatial, spatial, batch, time)

## Version History

Introduced in R2016a

## 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™.

Usage notes and limitations:

- Code generation that does not depend on third-party libraries can use 'mean' for `PaddingValue` property.
- Code generation using third-party libraries such as MKL - DNN, ARM Compute does not support mean value padding.
- For Simulink models that implement deep learning functionality using MATLAB Function block, simulation errors out if the network contains an average pooling layer with non-zero padding value. In such cases, use the blocks from the **Deep Neural Networks** library instead of a MATLAB Function to implement the deep learning functionality.

### GPU Code Generation

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

Usage notes and limitations:

- Code generation using third-party libraries such as cuDNN, ARM Compute does not support non-zero padding value.
- For Simulink models that implement deep learning functionality using MATLAB Function block, simulation errors out if the network contains an average pooling layer with non-zero padding value. In such cases, use the blocks from the **Deep Neural Networks** library instead of a MATLAB Function to implement the deep learning functionality.

## See Also

[convolution2dLayer](#) | [globalAveragePooling2dLayer](#) | [maxPooling2dLayer](#)

## Topics

[“Create Simple Deep Learning Neural Network for Classification”](#)

[“Train Convolutional Neural Network for Regression”](#)

[“Deep Learning in MATLAB”](#)

[“Specify Layers of Convolutional Neural Network”](#)

[“List of Deep Learning Layers”](#)

# 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.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

## 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

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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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 the name ' '.

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	''	3-D Convolution	20 5x5x5 convolutions with stride [1 1 1] and padding [0 0 0]
3	''	ReLU	ReLU
4	''	3-D Average Pooling	2x2x2 average pooling with stride [2 2 2] and padding [0 0 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`.

## Algorithms

### 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 “2-D Average Pooling Layer” on page 1-209 section of the `averagePooling2dLayer` reference page.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `AveragePooling3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSB" (spatial, spatial, spatial, batch)	"SSSB" (spatial, spatial, spatial, batch)

In `dlnetwork` objects, `AveragePooling3DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)
"SSBT" (spatial, spatial, batch, time)	"SSBT" (spatial, spatial, batch, time)
"SSSBT" (spatial, spatial, spatial, batch, time)	"SSSBT" (spatial, spatial, spatial, batch, time)

## Version History

Introduced in R2019a

### See Also

`averagePooling2dLayer` | `convolution3dLayer` | `maxPooling3dLayer` | `globalAveragePooling3dLayer`

### Topics

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

# avgpool

Pool data to average values over spatial dimensions

## Syntax

```
Y = avgpool(X, poolsize)
Y = avgpool(X, 'global')
Y = avgpool( ____, 'DataFormat', FMT)
Y = 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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` 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`
- 

`Y = avgpool(X, poolsize)` applies the average pooling operation to the formatted `darray` object `X`. 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 `Y` is a formatted `darray` with the same dimension format as `X`.

The function, by default, pools over up to three dimensions of `X` labeled 'S' (spatial). To pool over dimensions labeled 'T' (time), specify a pooling region with a 'T' dimension using the 'PoolFormat' option.

`Y = avgpool(X, 'global')` computes the global average over the spatial dimensions of the input `X`. This syntax is equivalent to setting `poolsize` in the previous syntax to the size of the 'S' dimensions of `X`.

`Y = avgpool( ____, 'DataFormat', FMT)` applies the average pooling operation to the unformatted `darray` object `X` with format specified by `FMT` using any of the previous syntaxes. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. For example,

'DataFormat', 'SSCB' specifies data for 2-D average pooling with format 'SSCB' (spatial, spatial, channel, batch).

`Y = 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

### X — Input data

`dlarray`

Input data, specified as a formatted or unformatted `dlarray` object.

If X 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 X 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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 `X` 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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

### Y — Pooled data

`dlarray`

Pooled data, returned as a `dlarray` with the same underlying data type as `X`.

If the input data `X` is a formatted `dlarray`, then `Y` has the same format as `X`. If the input data is not a formatted `dlarray`, then `Y` 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 “2-D Average Pooling Layer” on page 1-209 section of the `averagePooling2dLayer` reference page.

## Version History

Introduced in R2019b

## 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 `X` 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”

# BaselineDistributionDiscriminator

Baseline distribution discriminator

## Description

The `BaselineDistributionDiscriminator` object is a distribution discriminator that uses the baseline method to compute distribution confidence scores. The object contains a threshold that you can use to separate observations into in-distribution (ID) and out-of-distribution (OOD) data sets.

## Creation

Create an `BaselineDistributionDiscriminator` object using the `networkDistributionDiscriminator` function and setting the method input argument to "baseline".

---

**Note** The object type depends on the method you specify when you use the `networkDistributionDiscriminator` function. The method determines how the software computes the distribution confidence scores. You can set the method argument to either "baseline", "odin", "energy", or "hbos". For more information, see "Distribution Confidence Scores" on page 1-228.

---

## Properties

### Method — Method for computing distribution confidence scores

"baseline"

This property is read-only.

Method for computing the distribution confidence scores, returned as "baseline". For more information, see "Distribution Confidence Scores" on page 1-228.

### Network — Deep learning network

`dlnetwork` object

This property is read-only.

Deep learning network, returned as a `dlnetwork` object.

### Threshold — Distribution threshold

scalar in the range [0, 1]

This property is read-only.

Distribution threshold, returned as a scalar in the range [0, 1]. The threshold separates the ID and OOD data using the distribution confidence scores.

Example: 0.98

## Object Functions

`isInNetworkDistribution` Determine whether data is within the distribution of the network  
`distributionScores` Distribution confidence scores

## Examples

### Create Distribution Discriminator Using Baseline Method

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data. Convert the data to a `darray` object.

```
XID = digitTrain4DArrayData;  
XID = darray(XID,"SSCB");
```

Modify the ID training data to create an OOD set.

```
X00D = XID.*0.3 + 0.1;
```

Create a discriminator using the baseline method.

```
method = "baseline";  
discriminator = networkDistributionDiscriminator(net,XID,X00D,method)
```

```
discriminator =  
  BaselineDistributionDiscriminator with properties:
```

```
    Method: "baseline"  
    Network: [1x1 dlnetwork]  
    Threshold: 0.9743
```

Assess the performance of the discriminator on the OOD data.

```
tf = isInNetworkDistribution(discriminator,X00D);  
accuracy = sum(tf == 0)/numel(tf)
```

```
accuracy = 0.9402
```

## More About

### In-Distribution and Out-of-Distribution Data

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-228.

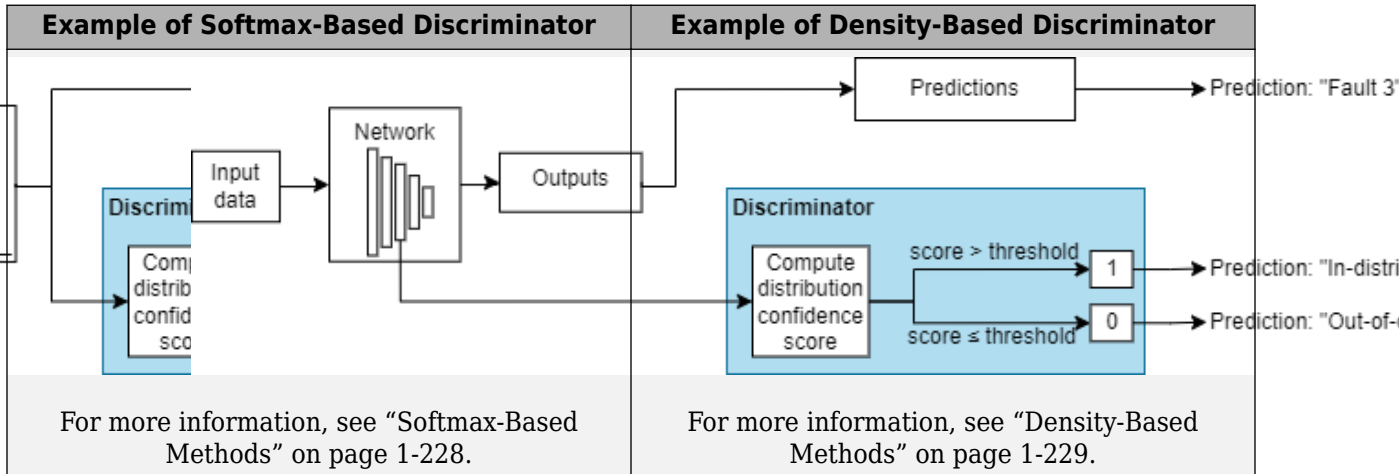
**OOD Data Detection**

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-228.

These images show how a discriminator acts as an additional output of a trained neural network.

**Example Data Discriminators**



**Distribution Confidence Scores**

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.

**Softmax-Based Methods**

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}}$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is  $-\text{Inf}$ .

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. "A Baseline for Detecting Out-of-Distribution Examples in Image Captioning." In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.
- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.



[6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## **See Also**

`networkDistributionDiscriminator` | `isInNetworkDistribution` | `distributionScores`

## **Topics**

"Out-of-Distribution Detection for Deep Neural Networks"

"Verify Robustness of Deep Learning Neural Network"

## batchnorm

Normalize data across all observations for each channel independently

### Syntax

```
Y = batchnorm(X,offset,scaleFactor)
[Y,popMu,popSigmaSq] = batchnorm(X,offset,scaleFactor)
[Y,updatedMu,updatedSigmaSq] = batchnorm(X,offset,scaleFactor,runningMu,
runningSigmaSq)
Y = batchnorm(X,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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .

The `batchnorm` function applies the batch normalization operation to `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** To apply batch normalization within a `layerGraph` object or `Layer` array, use `batchNormalizationLayer`.

---

`Y = batchnorm(X,offset,scaleFactor)` applies the batch normalization operation to the input data `X` 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 `X` for each channel in the 'C' (channel) dimension, independently.

For unformatted input data, use the 'DataFormat' option.

`[Y,popMu,popSigmaSq] = batchnorm(X,offset,scaleFactor)` applies the batch normalization operation and also returns the population mean and variance of the input data `X`.

`[Y,updatedMu,updatedSigmaSq] = batchnorm(X,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.

`Y = batchnorm(X,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 `Y` is an unformatted `dLarray` object with dimensions in the same order as `X`. 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`.

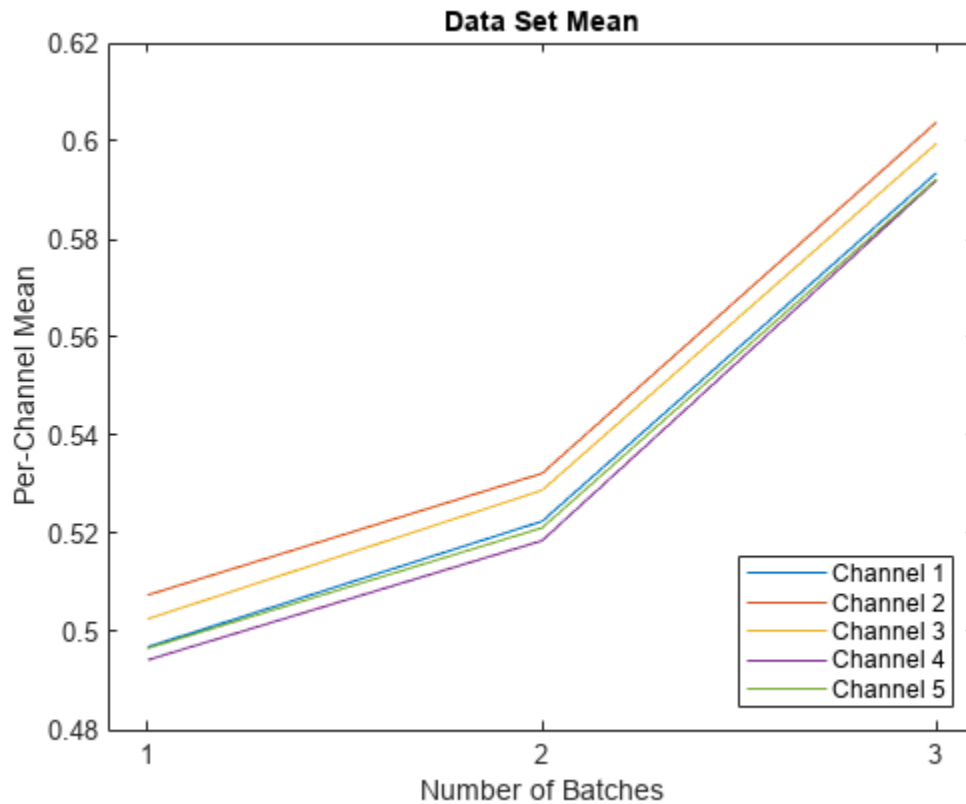
```
[dLY2,datasetMu,datasetSigmaSq] = batchnorm(dLX2,offset,scale,mu,sigmaSq);
```

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

### X — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `X` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `DataFormat` option. If `X` is a numeric array, then either `scaleFactor` or `offset` must be a `dlarray` object.

`X` must have a 'C' (channel) dimension.

### offset — Offset

`dlarray` | numeric array

Offset  $\beta$ , 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 `X`.

If `offset` is a formatted `dlarray` object, then the nonsingleton dimension must have label 'C' (channel).

### scaleFactor — Scale factor

`dlarray` | numeric array

Scale factor  $\gamma$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input `X`.

If `scaleFactor` is a formatted `darray` 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

### **Epsilon — Constant to add to mini-batch variances**

1e-5 (default) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`Epsilon` must be greater than or equal to 1e-5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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`,  $\lambda_\mu$  denotes the mean decay value 'MeanDecay',  $\widehat{\mu}$  denotes the mean of the input data, and  $\mu$  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  $\sigma^{2*}$  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  $\sigma^2$  denotes the current value of the variance `sigmaSq`.

Data Types: `single` | `double`

## Output Arguments

### Y — Normalized data

`dIarray`

Normalized data, returned as a `dIarray` with the same underlying data type as X.

If the input data X is a formatted `dIarray`, then Y has the same format as X. If the input data is not a formatted `dIarray`, then Y is an unformatted `dIarray` with the same dimension order as the input data.

The size of the output Y matches the size of the input X.

### 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  $\mu^*$  denotes the updated mean `updatedMu`,  $\lambda_{\mu}$  denotes the mean decay value 'MeanDecay',  $\widehat{\mu}$  denotes the mean of the input data, and  $\mu$  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  $\sigma^{2*}$  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  $\sigma^2$  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  $\mu_B$  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  $\epsilon$  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  $\beta$  and scale factor  $\gamma$  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.

## Version History

**Introduced in R2019b**

**R2023a: Epsilon supports values less than 1e-5**

The `Epsilon` option also supports positive values less than 1e-5.

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `X`
  - `offset`
  - `scaleFactor`

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

## See Also

`relu` | `fullyconnect` | `dlconv` | `darray` | `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 `darray` Support”

# 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  $\gamma$  and shifts it by a learnable offset  $\beta$ .

## 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 1-244, “Learning Rate and Regularization” on page 1-247, 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 vector

Mean statistic used for prediction, specified as a numeric vector of per-channel mean values.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	

Layer Input	Property Size
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

Data Types: `single` | `double`

### **TrainedVariance — Variance statistic used for prediction**

numeric vector

Variance statistic used for prediction, specified as a numeric vector of per-channel variance values.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

Data Types: `single` | `double`

### **Epsilon — Constant to add to mini-batch variances**

`1e-5` (default) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

Epsilon must be greater than or equal to `1e-5`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NumChannels — Number of input channels**

`'auto'` (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- `'auto'` — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

## **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  $\gamma$ , specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Scale` is nonempty, then the software uses the `Scale` property as the initial value. If `Scale` is empty, then the software uses the initializer specified by `ScaleInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

Data Types: `single` | `double`

### Offset — Channel offsets

`[]` (default) | numeric array

Channel offsets  $\beta$ , specified as a numeric vector.

The channel offsets are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Offset` is nonempty, then the software uses the `Offset` property as the initial value. If `Offset` is empty, then the software uses the initializer specified by `OffsetInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

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  $\mu^*$  denotes the updated mean,  $\lambda_{\mu}$  denotes the mean decay value,  $\widehat{\mu}$  denotes the mean of the layer input, and  $\mu$  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 | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**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  $\sigma^{2*}$  denotes the updated variance,  $\lambda_{\sigma^2}$  denotes the variance decay value,  $\widehat{\sigma^2}$  denotes the variance of the layer input, and  $\sigma^2$  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 | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



## 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ScaleL2Factor — L<sub>2</sub> regularization factor for scale factors

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### OffsetL2Factor — L<sub>2</sub> regularization factor for offsets

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> 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 the name ' '.

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'
```

```
Hyperparameters
```

```

    MeanDecay: 0.1000
    VarianceDecay: 0.1000
    Epsilon: 1.0000e-05

```

```

Learnable Parameters
  Offset: []
  Scale: []

```

```

State Parameters
  TrainedMean: []
  TrainedVariance: []

```

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	''	2-D Convolution	16 3x3 convolutions with stride [1 1] and padding [1 1]
3	''	Batch Normalization	Batch normalization
4	''	ReLU	ReLU
5	''	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6	''	2-D 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

## Algorithms

### 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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .  $\beta$  and  $\gamma$  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`.

The batch normalization operation normalizes the elements  $x_i$  of the input by first calculating the mean  $\mu_B$  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  $\epsilon$  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  $\beta$  and scale factor  $\gamma$  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.

## Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `BatchNormalizationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `BatchNormalizationLayer` objects also support these input and output format combinations.

<b>Input Format</b>	<b>Output Format</b>
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2017b

### R2023a: Epsilon supports values less than 1e-5

The Epsilon option also supports positive values less than 1e-5.

## 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 Neural Network for Classification"

"Train Convolutional Neural Network for Regression"

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

# bilstmLayer

Bidirectional long short-term memory (BiLSTM) layer for recurrent neural network (RNN)

## Description

A bidirectional LSTM (BiLSTM) layer is an RNN layer that learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the RNN 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 1-254, `State` on page 1-255, `Parameters and Initialization` on page 1-256, `Learning Rate and Regularization` on page 1-258, 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

This property is read-only.

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 that the layer remembers between time steps (the hidden state). The hidden state can contain information from all the 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.

The hidden state does not limit the number of time steps that the layer processes in an iteration. To split your sequences into smaller sequences for when you use the `trainNetwork` function, use the `SequenceLength` training option.

The layer outputs data with `NumHiddenUnits` channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**OutputMode — Output mode**

`'sequence'` (default) | `'last'`

This property is read-only.

Output mode, specified as one of these values:

- `'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)

This property is read-only.

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 the 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 the 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)

This property is read-only.

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 the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with the 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 that it computes.

**InputSize — Input size**

`'auto'` (default) | positive integer

This property is read-only.

Input size, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically assigns the input size at training time.

Data Types: `double` | `char`

**Activations****StateActivationFunction — Activation function to update cell and hidden state**

`'tanh'` (default) | `'softsign'`



This property is read-only.

Activation function to update the cell and hidden state, specified as one of these values:

- '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  $\sigma_c$  in the calculations to update the cell and hidden state. For more information on how an LSTM layer uses activation functions, see “Long Short-Term Memory Layer” on page 1-1251.

### GateActivationFunction — Activation function to apply to gates

'sigmoid' (default) | 'hard-sigmoid'

This property is read-only.

Activation function to apply to the gates, specified as one of these values:

- 'sigmoid' — Use the sigmoid function  $\sigma(x) = (1 + e^{-x})^{-1}$ .
- 'hard-sigmoid' — Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}$$

The layer uses this option as the function  $\sigma_g$  in the calculations for the layer gates.

## State

### CellState — Cell state

numeric vector

Cell state to use in the layer operation, specified as a  $2 * \text{NumHiddenUnits}$ -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.

Data Types: single | double

### HiddenState — Hidden state

numeric vector

Hidden state to use in the layer operation, specified as a  $2 * \text{NumHiddenUnits}$ -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.

Data Types: single | double

## 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 these values:

- '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 a standard deviation of 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 you train a neural network using the `trainNetwork` function, if `InputWeights` is nonempty, then the software uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then the software uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is an  $8 \times \text{NumHiddenUnits}$ -by-`InputSize` matrix.

Data Types: single | double

### **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 you train an RNN using the `trainNetwork` function, if `RecurrentWeights` is nonempty, then the software uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then the software uses the initializer specified by `RecurrentWeightsInitializer`.

At training time, `RecurrentWeights` is an  $8 \times \text{NumHiddenUnits}$ -by- $\text{NumHiddenUnits}$  matrix.

Data Types: `single` | `double`

### **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 neural 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  $8 \times \text{NumHiddenUnits}$ -by-1 numeric vector.

Data Types: `single` | `double`

### **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 you specify 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 you specify using 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**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  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the input weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the recurrent weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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.

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 the name `''`.

Data Types: `char` | `string`

### NumInputs — Number of inputs

1 | 3

This property is read-only.

Number of inputs to the layer.

If the `HasStateInputs` property is `0` (`false`), then the layer has one input with the 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 the 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'}`

This property is read-only.

Input names of the layer.

If the `HasStateInputs` property is `0` (`false`), then the layer has one input with the 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 the 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



This property is read-only.

Number of outputs to the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with the 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 that it computes.

Data Types: `double`

### OutputNames – Output names

```
{'out'} | {'out', 'hidden', 'cell'}
```

This property is read-only.

Output names of the layer.

If the `HasStateOutputs` property is `0` (false), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with the 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 that it computes.

## 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 to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `BiLSTMLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `dArray` object with dimensions ordered corresponding to the formats in this table.

Input Format	OutputMode	Output Format
"CB" (channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"CBT" (channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SB" (spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	

In `dlnetwork` objects, `BiLSTMLayer` objects also support these input and output format combinations.

Input Format	OutputMode	Output Format	
"SCB" (spatial, channel, batch)	"sequence"	"CB" (channel, batch)	
	"last"		
"SSCB" (spatial, spatial, channel, batch)	"sequence"		
	"last"		
"SSSCB" (spatial, spatial, spatial, channel, batch)	"sequence"		
	"last"		
"SCBT" (spatial, channel, batch, time)	"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)	
"SC" (spatial, channel)	"sequence"	"CU" (channel, unspecified)	
	"last"		
"SSC" (spatial, spatial, channel)	"sequence"		
	"last"		
"SSSC" (spatial, spatial, spatial, channel)	"sequence"		
	"last"		
"CT" (channel, time)	"sequence"		"CT" (channel, time)
	"last"		"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"sequence"		"CT" (channel, time)
	"last"	"CU" (channel, unspecified)	

Input Format	OutputMode	Output Format
"SSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSB" (spatial, spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"SSSB" (spatial, spatial, spatial, batch)	"sequence"	
	"last"	
"BT" (batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SBT" (spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSBT" (spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSSBT" (spatial, spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

To use these input formats in `trainNetwork` workflows, convert the data to "CB" (channel, batch) or "CBT" (channel, batch, time) format using `flattenLayer`.

If the `HasStateInputs` property is 1 (true), then the layer has two additional inputs with the 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).

## Version History

### Introduced in R2018a

#### R2019a: 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 neural 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'.

#### R2019a: 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 neural 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>
- [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:

For code generation in general, the `HasStateInputs` and `HasStateOutputs` properties must be set to 0 (false).

When generating code with Intel MKL-DNN or ARM Compute Library:

- The `StateActivationFunction` property must be set to 'tanh'.
- The `GateActivationFunction` property must be set to 'sigmoid'.

### 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 Neural Networks"
- "List of Deep Learning Layers"
- "Deep Learning Tips and Tricks"

# calibrate

Simulate and collect ranges of a deep neural network

## Syntax

```
calResults = calibrate(quantObj, calData)
calResults = calibrate(quantObj, calData, Name, Value)
```

## Description

`calResults = 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`.

`calResults = calibrate(quantObj, calData, Name, Value)` calibrates the network with additional options specified by one or more name-value pair arguments.

This function requires Deep Learning Toolbox Model Quantization Library. 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 squeezenetmerch
net
net =
  DAGNetwork with properties:
    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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=121x5 table
    Optimized Layer Name      Network Layer Name      Learnables / Activations      MinValue
```



```

_____
{'conv1_Weights'      }      {'conv1'          }      "Weights"      -0.9198
{'conv1_Bias'         }      {'conv1'          }      "Bias"         -0.0792
{'fire2-squeeze1x1_Weights'}  {'fire2-squeeze1x1'}  "Weights"      -1.3
{'fire2-squeeze1x1_Bias' }  {'fire2-squeeze1x1'}  "Bias"         -0.1164
{'fire2-expand1x1_Weights'}  {'fire2-expand1x1' }  "Weights"      -0.740
{'fire2-expand1x1_Bias' }  {'fire2-expand1x1' }  "Bias"         -0.06005
{'fire2-expand3x3_Weights'}  {'fire2-expand3x3' }  "Weights"      -0.7439
{'fire2-expand3x3_Bias' }  {'fire2-expand3x3' }  "Bias"         -0.05177
{'fire3-squeeze1x1_Weights'}  {'fire3-squeeze1x1'}  "Weights"      -0.771
{'fire3-squeeze1x1_Bias' }  {'fire3-squeeze1x1'}  "Bias"         -0.1013
{'fire3-expand1x1_Weights'}  {'fire3-expand1x1' }  "Weights"      -0.7203
{'fire3-expand1x1_Bias' }  {'fire3-expand1x1' }  "Bias"         -0.06702
{'fire3-expand3x3_Weights'}  {'fire3-expand3x3' }  "Weights"      -0.6144
{'fire3-expand3x3_Bias' }  {'fire3-expand3x3' }  "Bias"         -0.05361
{'fire4-squeeze1x1_Weights'}  {'fire4-squeeze1x1'}  "Weights"      -0.742
{'fire4-squeeze1x1_Bias' }  {'fire4-squeeze1x1'}  "Bias"         -0.1088
:

```

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: [1x1 struct]
    Statistics: [2x2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2x2 table
    NetworkImplementation    MetricOutput
    _____
    {'Floating-Point'}      1
    {'Quantized'           }      1
```

```
valResults.Statistics
```

```
ans=2x2 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 Network for FPGA Deployment

Reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. This example shows how to use Deep Learning Toolbox Model Quantization Library and Deep Learning HDL Toolbox to deploy the `int8` network to a target FPGA board.

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning.

## Load Pretrained Network

Load the pretrained LogoNet network and analyze the network architecture.

```
snet = getLogoNetwork;  
deepNetworkDesigner(snet);
```

## Load Data

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 for calibration and validation. Expedite calibration and validation 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. The FPGA validation workflow has a maximum limit of one image when validating the quantized network. Reduce the FPGA validation data set to a single image.

```
curDir = pwd;  
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_simulation = validationData.subset(1:5);  
validationData_FPGA = validationData.subset(1:1);
```

## Generate Calibration Result File for the Network

Create a `dlquantizer` (Deep Learning HDL Toolbox) object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj_simulation = dlquantizer(snet,'ExecutionEnvironment',"FPGA",'Simulation','on');  
dlQuantObj_FPGA = dlquantizer(snet,'ExecutionEnvironment',"FPGA");
```

Use the `calibrate` (Deep Learning HDL Toolbox) function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases. The `calibrate` function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj_simulation,calibrationData_reduced)
```

```
ans=35x5 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"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045961
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02955
{'imageinput' }	{'imageinput' }	"Activations"	( )
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

```
calibrate(dlQuantObj_FPGA,calibrationData_reduced)
```

```
ans=35x5 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"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045961
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02955
{'imageinput' }	{'imageinput' }	"Activations"	( )
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

## Create Target Object

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 use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\n2020.2\bin\vivado.bat');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, you can also use the JTAG interface.

```
% hTarget = dlhdl.Target('Xilinx', 'Interface', 'JTAG');
```

## Create dlQuantizationOptions Object

Create a `dlquantizationOptions` object. Specify the target bitstream and target board interface. The default metric function is a Top-1 accuracy metric function.

```
options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);  
options_simulation = dlquantizationOptions;
```

To use a custom metric function, specify the metric function in the `dlquantizationOptions` object.

```
options_FPGA = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_FPGA)});  
options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_simulation)});
```

## Validate Quantized Network

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_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulation);
```

```
Compiling leg: conv_1>>relu_4 ...  
Compiling leg: conv_1>>relu_4 ... complete.  
Compiling leg: maxpool_4 ...  
Compiling leg: maxpool_4 ... complete.  
Compiling leg: fc_1>>fc_3 ...  
Compiling leg: fc_1>>fc_3 ... complete.
```

```
prediction_simulation = struct with fields:  
    NumSamples: 5  
    MetricResults: [1x1 struct]  
    Statistics: []
```

For validation on an FPGA, the `validate` function:

- Programs the FPGA board by using the output of the `compile` method and the programming file
- Downloads the network weights and biases

- Compares the performance of the network before and after quantization

```
prediction_FPGA = dlQuantObj_FPGA.validate(validationData_FPGA,options_FPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
  1  'imageinput'    Image Input          227x227x3 images with 'zerocenter' normalization
  2  'conv_1'       Convolution          96 5x5x3 convolutions with stride [1 1] and padding
  3  'relu_1'       ReLU                 ReLU
  4  'maxpool_1'    Max Pooling          3x3 max pooling with stride [2 2] and padding
  5  'conv_2'       Convolution          128 3x3x96 convolutions with stride [1 1] and padding
  6  'relu_2'       ReLU                 ReLU
  7  'maxpool_2'    Max Pooling          3x3 max pooling with stride [2 2] and padding
  8  'conv_3'       Convolution          384 3x3x128 convolutions with stride [1 1] and padding
  9  'relu_3'       ReLU                 ReLU
 10  'maxpool_3'    Max Pooling          3x3 max pooling with stride [2 2] and padding
 11  'conv_4'       Convolution          128 3x3x384 convolutions with stride [2 2] and padding
 12  'relu_4'       ReLU                 ReLU
 13  'maxpool_4'    Max Pooling          3x3 max pooling with stride [2 2] and padding
 14  'fc_1'         Fully Connected      2048 fully connected layer
 15  'relu_5'       ReLU                 ReLU
 16  'dropout_1'    Dropout              50% dropout
 17  'fc_2'         Fully Connected      2048 fully connected layer
 18  'relu_6'       ReLU                 ReLU
 19  'dropout_2'    Dropout              50% dropout
 20  'fc_3'         Fully Connected      32 fully connected layer
 21  'softmax'      Softmax              softmax
 22  'classoutput'  Classification Output crossentropyex with 'adidas' and 31 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
### Compiling layer group: conv_1>>relu_4 ...
### Compiling layer group: conv_1>>relu_4 ... complete.
### Compiling layer group: maxpool_4 ...
### Compiling layer group: maxpool_4 ... complete.
### Compiling layer group: fc_1>>fc_3 ...
### Compiling layer group: fc_1>>fc_3 ... complete.

### Allocating external memory buffers:

      offset_name          offset_address      allocated_space
-----
"InputDataOffset"        "0x00000000"        "12.0 MB"
"OutputResultOffset"    "0x00c00000"        "4.0 MB"
"SchedulerDataOffset"    "0x01000000"        "4.0 MB"
"SystemBufferOffset"     "0x01400000"        "36.0 MB"
"InstructionDataOffset"  "0x03800000"        "8.0 MB"
"ConvWeightDataOffset"  "0x04000000"        "12.0 MB"
"FCWeightDataOffset"    "0x04c00000"        "12.0 MB"
"EndOffset"              "0x05800000"        "Total: 88.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Deep learning network programming has been skipped as the same network is already loaded on the target
```

```
### Finished writing input activations.
### Running single input activation.
```

Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	248358	274080	90.62
DSPs	384	2520	15.24
Block RAM	581	912	63.71

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

```
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	40142478	0.18247	1	40142478
__imageinput_norm	216472	0.00098		
__conv_1	6825671	0.03103		
__maxpool_1	3755088	0.01707		
__conv_2	10440701	0.04746		
__maxpool_2	1447840	0.00658		
__conv_3	9405685	0.04275		
__maxpool_3	1765856	0.00803		
__conv_4	1819636	0.00827		
__maxpool_4	28098	0.00013		
__fc_1	2651288	0.01205		
__fc_2	1696632	0.00771		
__fc_3	89511	0.00041		

\* The clock frequency of the DL processor is: 220MHz

Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	168645	274080	61.53
DSPs	800	2520	31.75
Block RAM	453	912	49.67

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

```
### Finished writing input activations.
### Running single input activation.
```

```
prediction_FPGA = struct with fields:
  NumSamples: 1
  MetricResults: [1x1 struct]
  Statistics: [2x7 table]
```

## View Performance of Quantized Neural Network

Display the accuracy of the quantized network.

```
prediction_simulation.MetricResults.Result
```

```
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}      1
  {'Quantized'      }      1
```

```
prediction_FPGA.MetricResults.Result
```

```
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}      1
  {'Quantized'      }      1
```

Display the performance of the quantized network in frames per second.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
```

```
ans = 19.0828
```

## Quantize a Neural Network for CPU Target

This example shows how to quantize and validate a neural network for a CPU target. This workflow is similar to other execution environments, but before validating you must establish a `raspi` connection and specify it as target using `dlquantizationOptions`.

First, load your network. This example uses the pretrained network `squeezenet`.

```
load squeezenetmerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Then define your calibration and validation data, `calDS` and `valDS` respectively.

```
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 the dlquantizer object and specify a CPU execution environment.

```
dq = dlquantizer(net,'ExecutionEnvironment','CPU')

dq =
    dlquantizer with properties:
        NetworkObject: [1x1 DAGNetwork]
        ExecutionEnvironment: 'CPU'
```

Calibrate the network.

```
calResults = calibrate(dq,aug_calData,'UseGPU','off')
```

```
calResults=122x5 table
    Optimized Layer Name      Network Layer Name      Learnables / Activations      MinValue
    _____            _____            _____            _____
    {'conv1_Weights'         }      {'conv1'                }      "Weights"                -0.9198
    {'conv1_Bias'            }      {'conv1'                }      "Bias"                    -0.0792
    {'fire2-squeeze1x1_Weights'}      {'fire2-squeeze1x1'     }      "Weights"                -1.3
    {'fire2-squeeze1x1_Bias' }      {'fire2-squeeze1x1'     }      "Bias"                    -0.1164
    {'fire2-expand1x1_Weights'}      {'fire2-expand1x1'      }      "Weights"                -0.740
    {'fire2-expand1x1_Bias'  }      {'fire2-expand1x1'      }      "Bias"                    -0.06005
    {'fire2-expand3x3_Weights'}      {'fire2-expand3x3'      }      "Weights"                -0.7439
    {'fire2-expand3x3_Bias'  }      {'fire2-expand3x3'      }      "Bias"                    -0.05177
    {'fire3-squeeze1x1_Weights'}      {'fire3-squeeze1x1'     }      "Weights"                -0.771
    {'fire3-squeeze1x1_Bias' }      {'fire3-squeeze1x1'     }      "Bias"                    -0.1013
    {'fire3-expand1x1_Weights'}      {'fire3-expand1x1'      }      "Weights"                -0.7203
    {'fire3-expand1x1_Bias'  }      {'fire3-expand1x1'      }      "Bias"                    -0.06702
    {'fire3-expand3x3_Weights'}      {'fire3-expand3x3'      }      "Weights"                -0.6144
    {'fire3-expand3x3_Bias'  }      {'fire3-expand3x3'      }      "Bias"                    -0.05361
    {'fire4-squeeze1x1_Weights'}      {'fire4-squeeze1x1'     }      "Weights"                -0.742
    {'fire4-squeeze1x1_Bias' }      {'fire4-squeeze1x1'     }      "Bias"                    -0.1088
    :
```

Use the MATLAB Support Package for Raspberry Pi Hardware function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name or address of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiname','username','password')
```

For example,

```
r = raspi('gpcoder-raspberrypi-7','pi','matlab')

r =
    raspi with properties:
        DeviceAddress: 'gpcoder-raspberrypi-7'
```



```

        Port: 18734
        BoardName: 'Raspberry Pi 3 Model B+'
        AvailableLEDs: {'led0'}
        AvailableDigitalPins: [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
        AvailableSPIChannels: {}
        AvailableI2CBuses: {}
        AvailableWebcams: {}
        I2CBusSpeed:
AvailableCANInterfaces: {}

Supported peripherals

```

Specify `raspi` object as the target for the quantized network.

```

opts = dlquantizationOptions('Target',r)

opts =
dlquantizationOptions with properties:

    MetricFcn: {}
    Bitstream: ''
    Target: [1x1 raspi]

```

Validate the quantized network with the `validate` function.

```

valResults = validate(dq, aug_valData, opts)

### Starting application: 'codegen\lib\validate_predict_int8\pil\validate_predict_int8.elf'
To terminate execution: clear validate_predict_int8_pil
### Launching application validate_predict_int8.elf...
### Host application produced the following standard output (stdout) and standard error (stderr)

valResults = struct with fields:
    NumSamples: 20
    MetricResults: [1x1 struct]
    Statistics: []

```

Examine the validation output to see the performance of the quantized network.

```

valResults.MetricResults.Result

ans=2x2 table
    NetworkImplementation    MetricOutput
    _____    _____
    {'Floating-Point'}      0.95
    {'Quantized' }         0.95

```

## Quantize YOLO v3 Object Detector

This example shows how to quantize a `yolov3objectDetector` (Computer Vision Toolbox) object using preprocessed calibration and validation data.

First, download a pretrained YOLO v3 object detector.

```
detector = downloadPretrainedNetwork();
```

This example uses a small labeled data set that contains one or two labeled instances of a vehicle. Many of these images come from the Caltech Cars 1999 and 2001 data sets, created by Pietro Perona and used with permission.

Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

Add the full path to the local vehicle data folder.

```
vehicleDataset.imageFilename = fullfile(pwd, vehicleDataset.imageFilename);
```

Create an `imageDatastore` for loading the images and a `boxLabelDatastore` (Computer Vision Toolbox) for the ground truth bounding boxes.

```
imds = imageDatastore(vehicleDataset.imageFilename);
blbs = boxLabelDatastore(vehicleDataset(:,2));
```

Use the `combine` function to combine both the datastores into a `CombinedDatastore`.

```
combinedDS = combine(imds, blbs);
```

Split the data into calibration and validation data.

```
calData = combinedDS.subset(1:32);
valData = combinedDS.subset(33:64);
```

Use the `preprocess` (Computer Vision Toolbox) method of `yoloV3ObjectDetector` (Computer Vision Toolbox) object with `transform` function to prepare the data for calibration and validation.

The `transform` function returns a `TransformedDatastore` object.

```
processedCalData = transform(calData, @(data)preprocess(detector,data));
processedValData = transform(valData, @(data)preprocess(detector,data));
```

Create the `dlquantizer` object.

```
dq = dlquantizer(detector, 'ExecutionEnvironment', 'MATLAB');
```

Calibrate the network.

```
calResults = calibrate(dq, processedCalData, 'UseGPU', 'off')
```

*calResults=135x5 table*

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.92189
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.09627
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.375
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1206
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.7527

```

{'fire2-expand1x1_Bias' } {'fire2-expand1x1' } "Bias" -0.059253
{'fire2-expand3x3_Weights' } {'fire2-expand3x3' } "Weights" -0.7527
{'fire2-expand3x3_Bias' } {'fire2-expand3x3' } "Bias" -0.062214
{'fire3-squeeze1x1_Weights' } {'fire3-squeeze1x1' } "Weights" -0.758
{'fire3-squeeze1x1_Bias' } {'fire3-squeeze1x1' } "Bias" -0.1020
{'fire3-expand1x1_Weights' } {'fire3-expand1x1' } "Weights" -0.7156
{'fire3-expand1x1_Bias' } {'fire3-expand1x1' } "Bias" -0.06931
{'fire3-expand3x3_Weights' } {'fire3-expand3x3' } "Weights" -0.6007
{'fire3-expand3x3_Bias' } {'fire3-expand3x3' } "Bias" -0.05804
{'fire4-squeeze1x1_Weights' } {'fire4-squeeze1x1' } "Weights" -0.73
{'fire4-squeeze1x1_Bias' } {'fire4-squeeze1x1' } "Bias" -0.1118
:

```

Validate the quantized network with the `validate` function.

```

valResults = validate(dq, processedValData)

valResults = struct with fields:
    NumSamples: 32
    MetricResults: [1x1 struct]
    Statistics: []

function detector = downloadPretrainedNetwork()
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehicleExample_21aSPKG.zip';
    websave('yolov3SqueezeNetVehicleExample_21aSPKG.zip', pretrainedURL);

    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');

    pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
    detector = pretrained.detector;
end

```

## Input Arguments

### **quantObj** — Network to quantize

dlquantizer object

Network to quantize, specified as a `dlquantizer` object.

### **calData** — Data to use for calibration of quantized network

imageDatastore object | augmentedImageDatastore object | pixelLabelImageDatastore object | CombinedDatastore object | TransformedDatastore object

Data to use for calibration of quantized network, specified as an `imageDatastore` object, an `augmentedImageDatastore` object, a `pixelLabelImageDatastore` object, a `CombinedDatastore` object, or a `TransformedDatastore` object.

You must preprocess the data used for calibration of a `yolov3ObjectDetector` object using the `preprocess` function. For an example of using preprocessed data for calibration of a `yolov3ObjectDetector`, see “Quantize YOLO v3 Object Detector” on page 1-279.

For more information on valid datastores, see “Prepare Data for Quantizing Networks”.

## 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `calResults = calibrate(quantObj,calData,'UseGPU','on')`

### MiniBatchSize — Size of mini-batches

32 (default) | positive integer

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### UseGPU — Whether to use host GPU for calibration

'auto' (default) | 'on' | 'off'

Whether to use host GPU for calibration, specified as one of the following:

- 'auto' — Use host GPU for calibration if one is available. Otherwise, use host CPU for calibration.
- 'on' — Use host GPU for calibration.
- 'off' — Use host CPU for calibration.

Data Types: `char`

## Output Arguments

### calResults — 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.

## Version History

Introduced in R2020a

### R2022b: Calibrate on host GPU or host CPU

You can now choose whether to calibrate your network using the host GPU or host CPU. By default, the `calibrate` function and the **Deep Network Quantizer** app will calibrate on the host GPU if one is available.

In previous versions, it was required that the execution environment was the same as the instrumentation environment used for the calibration step of quantization.

**R2022b: Specify mini-batch size to use for calibration**

Use `MiniBatchSize` to specify the size of mini-batches to use for calibration.

**R2021a: ARM Cortex-A calibration support**

The Deep Learning Toolbox Model Quantization Library now supports calibration of a network for quantization and deployment on ARM Cortex<sup>®</sup>-A microcontrollers.

**See Also****Apps**

**Deep Network Quantizer**

**Functions**

`validate` | `dlquantizer` | `dlquantizationOptions` | `quantize` | `quantizationDetails` | `estimateNetworkMetrics`

**Topics**

“Quantization Workflow Prerequisites”

“Prepare Data for Quantizing Networks”

“Quantization of Deep Neural Networks”

“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”

“Quantize Network for FPGA Deployment” (Deep Learning HDL Toolbox)

“Generate int8 Code for Deep Learning Networks” (MATLAB Coder)

## 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 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.

```
layer = preluLayer;
```

Because the layer has a custom initialize function, initialize the layer using a `networkDataFormat` object that specifies the expected input size and format of a single observation of typical input to the layer.

Specify a valid input size of `[24 24 20]`, where the dimensions correspond to the height, width, and number of channels of the previous layer output.

```
validInputSize = [24 24 20];
layout = networkDataLayout(validInputSize,"SSC");
layer = initialize(layer,layout);
```

Check the layer validity using `checkLayer`. Specify the valid input size as the size as used to initialize the layer. When you pass data through the network, 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.

```
checkLayer(layer,validInputSize)
```

Skipping multi-observation tests. To enable tests with multiple observations, specify the 'ObservationDimension' property. 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: 0.054851 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.

```
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: 0.030498 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
Acceleratable: 0

Learnable Parameters
No properties.

State Parameters
No properties.

Show all properties
```

Check that the layer 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 'ObservationDimension' option.
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, specify the 'CodeGeneration' option.
```

```
Running nnet.checklayer.TestLayerWithoutBackward
.....
Done nnet.checklayer.TestLayerWithoutBackward
```

```
Test Summary:
    12 Passed, 0 Failed, 0 Incomplete, 16 Skipped.
    Time elapsed: 0.21153 seconds.
```

The results show the number of passed, failed, and skipped tests. If you do not specify the `ObservationDimension` 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: 0.16924 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: 0.87346 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.

If `layer` has learnable or state parameters, then the layer must be initialized. If the layer has a custom `initialize` function, then first initialize the layer using the `initialize` function using `networkDataLayout` objects.

The `checkLayer` function does not support layers that inherit from `nnet.layer.Formattable`.

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`.

### 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 1-289.

For large input sizes, the gradient checks take longer to run. To speed up the check, 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 1-289.

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 check, 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
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:

<b>Classification Task</b>	<b>Input Size</b>	<b>Observation Dimension</b>
2-D image classification	1-by-1-by- $K$ -by- $N$ , where $K$ is the number of classes and $N$ is the number of observations	4
3-D image classification	1-by-1-by-1-by- $K$ -by- $N$ , where $K$ is the number of classes and $N$ is the number of observations	5
Sequence-to-label classification	$K$ -by- $N$ , where $K$ is the number of classes and $N$ is the number of observations	2
Sequence-to-sequence classification	$K$ -by- $N$ -by- $S$ , where $K$ is the number of classes, $N$ is the number of observations, and $S$ is the sequence length	2

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  $Y$ .

<b>Regression Task</b>	<b>Input Size</b>	<b>Observation Dimension</b>
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
<code>functionSyntaxesAreCorrect</code>	The syntaxes of the layer functions are correctly defined.
<code>predictDoesNotError</code>	<code>predict</code> function does not error.
<code>forwardDoesNotError</code>	When specified, the <code>forward</code> function does not error.
<code>forwardPredictAreConsistentInSize</code>	When <code>forward</code> is specified, <code>forward</code> and <code>predict</code> output values of the same size.
<code>backwardDoesNotError</code>	When specified, <code>backward</code> does not error.
<code>backwardIsConsistentInSize</code>	When <code>backward</code> is specified, the outputs of <code>backward</code> are consistent in size: <ul style="list-style-type: none"> <li>The derivatives with respect to each input are the same size as the corresponding input.</li> <li>The derivatives with respect to each learnable parameter are the same size as the corresponding learnable parameter.</li> </ul>
<code>predictIsConsistentInType</code>	The outputs of <code>predict</code> are consistent in type with the inputs.
<code>forwardIsConsistentInType</code>	When <code>forward</code> is specified, the outputs of <code>forward</code> are consistent in type with the inputs.
<code>backwardIsConsistentInType</code>	When <code>backward</code> is specified, the outputs of <code>backward</code> are consistent in type with the inputs.
<code>gradientsAreNumericallyCorrect</code>	When <code>backward</code> is specified, the gradients computed in <code>backward</code> are consistent with the numerical gradients.
<code>backwardPropagationDoesNotError</code>	When <code>backward</code> is not specified, the derivatives can be computed using automatic differentiation.
<code>predictReturnsValidStates</code>	For layers with state properties, the <code>predict</code> function returns valid states.

Test	Description
forwardReturnsValidStates	For layers with state properties, the forward function, if specified, returns valid states.
resetStateDoesNotError	For layers with state properties, the resetState function, if specified, does not error and resets the states to valid states.
codegenPragmaDefinedInClassDef	The pragma "%#codegen" for code generation is specified in class file.
layerPropertiesSupportCodegen	The layer properties support code generation.
predictSupportsCodegen	predict is valid for code generation.
doesNotHaveStateProperties	For code generation, the layer does not have state properties.
functionLayerSupportsCodegen	For code generation, the layer function must be a named function on the path and the Formattable property must be 0 (false).

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.

<b>Test</b>	<b>Description</b>
<code>backwardPropagationDoesNotError</code>	When <code>backwardLoss</code> 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`.

## Version History

Introduced in R2018a

### 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”



# 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 '' 2-D Convolution      20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
    3 '' ReLU                  ReLU
    4 '' 2-D 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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D 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 classes

## 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 the name ''.

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  $n$ th sample belongs to the  $i$ th 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  $n$ th input with class  $i$ .

## Version History

Introduced in R2016a

## 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

“Train Sequence Classification Network Using Data With Imbalanced Classes”  
“Deep Learning in MATLAB”

“List of Deep Learning Layers”

# 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 'crossentropy', 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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D 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  ''  2-D Convolution       20 5x5 convolutions with stride [1 1] and padding [0 0 0]
     3  ''  ReLU                  ReLU
     4  ''  2-D 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 classes

```

## 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  $n$ th sample belongs to the  $i$ th 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  $n$ th input with class  $i$ .

## Version History

### Introduced in R2016a

#### R2018b: 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 Neural Network for Classification”

“Train Convolutional Neural Network for Regression”

“Train Sequence Classification Network Using Data With Imbalanced Classes”

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“List of Deep Learning Layers”

# classify

Classify data using trained deep learning neural network

## Syntax

```
Y = classify(net,images)
Y = classify(net,sequences)
Y = classify(net,features)

Y = classify(net,X1,...,XN)
Y = classify(net,mixed)

[Y,scores] = classify(____)
____ = classify(____,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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value argument.

For networks with multiple outputs, use the `predict` function instead and set the `ReturnCategorical` option to `true`.

`Y = classify(net,images)` predicts the class labels of the specified images using the trained network `net`.

`Y = classify(net,sequences)` predicts the class labels of the specified sequences using the trained network `net`.

`Y = classify(net,features)` predicts the class labels of the specified feature data using the trained network `net`.

`Y = classify(net,X1,...,XN)` predicts the class labels for the data in the numeric arrays or cell arrays `X1`, ..., `XN` for the multi-input network `net`. The input `Xi` corresponds to the network input `net.InputNames(i)`.

`Y = classify(net,mixed)` predicts the class labels using the trained network `net` with multiple inputs of mixed data types.

`[Y,scores] = classify(____)` also returns the classification scores corresponding to the class labels using any of the previous input arguments.

`____ = classify(____,Name=Value)` predicts class labels with additional options specified by one or more name-value arguments.

---

**Tip** When you make 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 Convolutional Neural Network

Load the pretrained network `digitsNet`. This network is a classification convolutional neural network that classifies handwritten digits.

```
load digitsNet
```

View the network layers. The output layer of the network is a classification layer.

```
layers = net.Layers
```

```
layers =
```

```
15x1 Layer array with layers:
```

1	'imageinput'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv_1'	2-D Convolution	8 3x3x1 convolutions with stride [1 1] and padding
3	'batchnorm_1'	Batch Normalization	Batch normalization with 8 channels
4	'relu_1'	ReLU	ReLU
5	'maxpool_1'	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding
6	'conv_2'	2-D Convolution	16 3x3x8 convolutions with stride [1 1] and padding
7	'batchnorm_2'	Batch Normalization	Batch normalization with 16 channels
8	'relu_2'	ReLU	ReLU
9	'maxpool_2'	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding
10	'conv_3'	2-D Convolution	32 3x3x16 convolutions with stride [1 1] and padding
11	'batchnorm_3'	Batch Normalization	Batch normalization with 32 channels
12	'relu_3'	ReLU	ReLU
13	'fc'	Fully Connected	10 fully connected layer
14	'softmax'	Softmax	softmax
15	'classoutput'	Classification Output	crossentropyex with '0' and 9 other classes

Load the test images.

```
digitDatasetPath = fullfile(toolboxdir("nnet"),"nndemos","nndatasets","DigitDataset");
imdsTest = imageDatastore(digitDatasetPath,IncludeSubfolders=true);
```

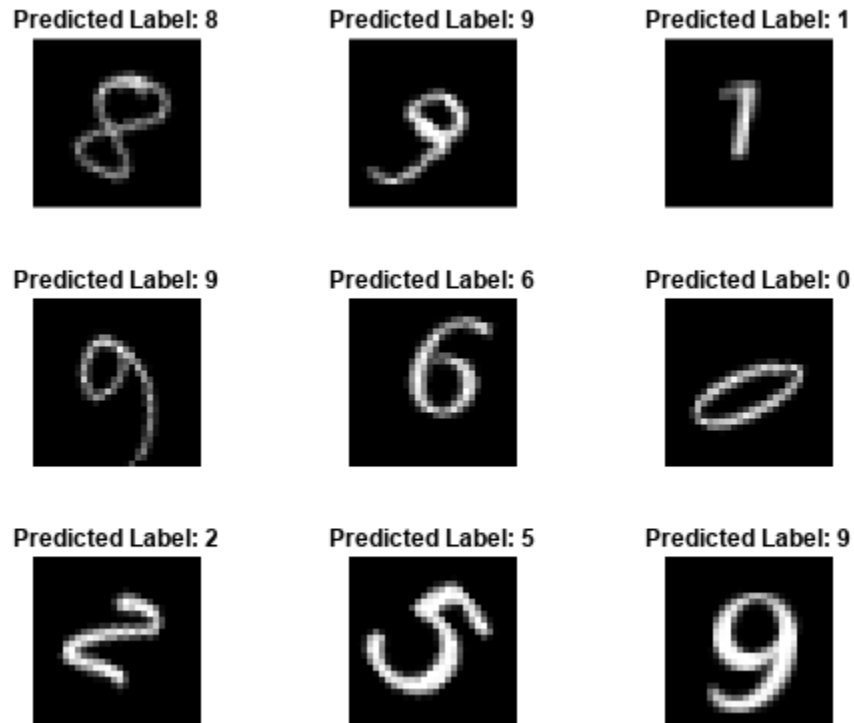
Classify the images using the `classify` function.

```
YTest = classify(net,imdsTest);
```

View some of the test images at random with their predictions.

```
numImages = 9;
idx = randperm(numel(imdsTest.Files),numImages);

figure
tiledlayout("flow")
for i = 1:numImages
    nexttile
    imshow(imdsTest.Files{idx(i)});
    title("Predicted Label: " + string(YTest(idx(i))))
end
```



### Classify Sequences Using Trained LSTM Network

Load the pretrained network `JapaneseVowelsNet`. This network is a pretrained LSTM network trained on the Japanese Vowels data set 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.

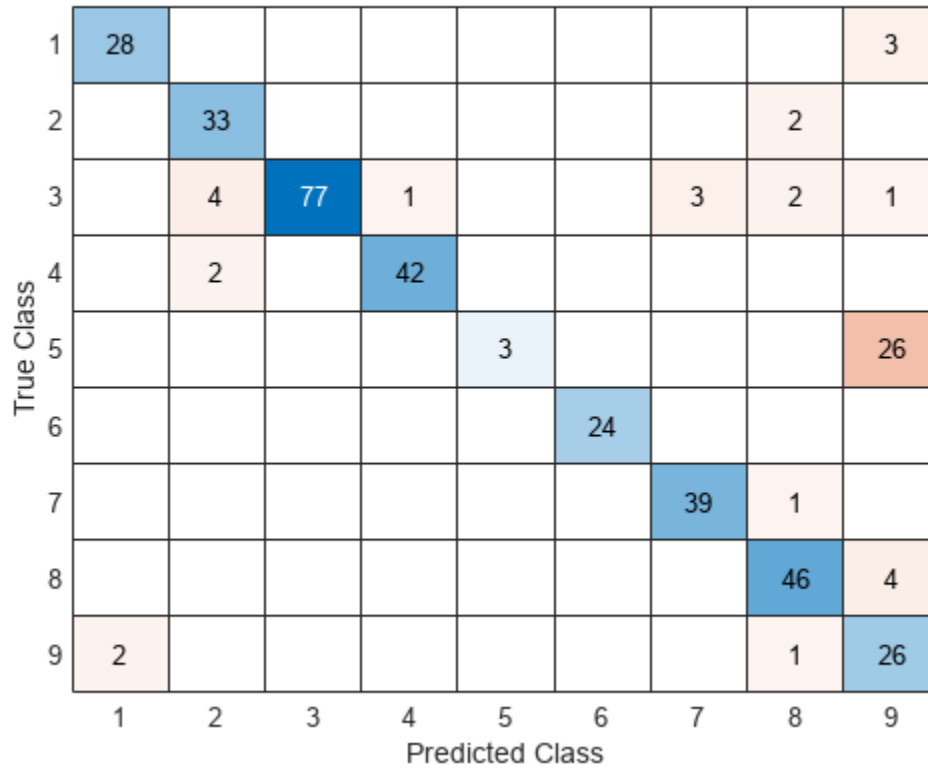
```
[XTest,TTest] = japaneseVowelsTestData;
```

Classify the test data.

```
YTest = classify(net,XTest);
```

View the predictions in a confusion chart.

```
figure
confusionchart(TTest,YTest)
```



Calculate the classification accuracy of the predictions.

```
accuracy = mean(YTest == TTest)
```

```
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
```

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. This data set has two categorical features named "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 insertion of 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)
```

SigMean	SigMedian	SigRMS	SigVar	SigPeak	SigPeak2Peak	SigSkewness	SigKurtosis
-0.94876	-0.9722	1.3726	0.98387	0.81571	3.6314	-0.041525	2.7111
-0.97537	-0.98958	1.3937	0.99105	0.81571	3.6314	-0.023777	2.7111
1.0502	1.0267	1.4449	0.98491	2.8157	3.6314	-0.04162	2.7111
1.0227	1.0045	1.4288	0.99553	2.8157	3.6314	-0.016356	2.7111

1.0123	1.0024	1.4202	0.99233	2.8157	3.6314	-0.014701	2.7
1.0275	1.0102	1.4338	1.0001	2.8157	3.6314	-0.02659	2.7
1.0464	1.0275	1.4477	1.0011	2.8157	3.6314	-0.042849	2.7
1.0459	1.0257	1.4402	0.98047	2.8157	3.6314	-0.035405	2.7

Extract the test labels from the table.

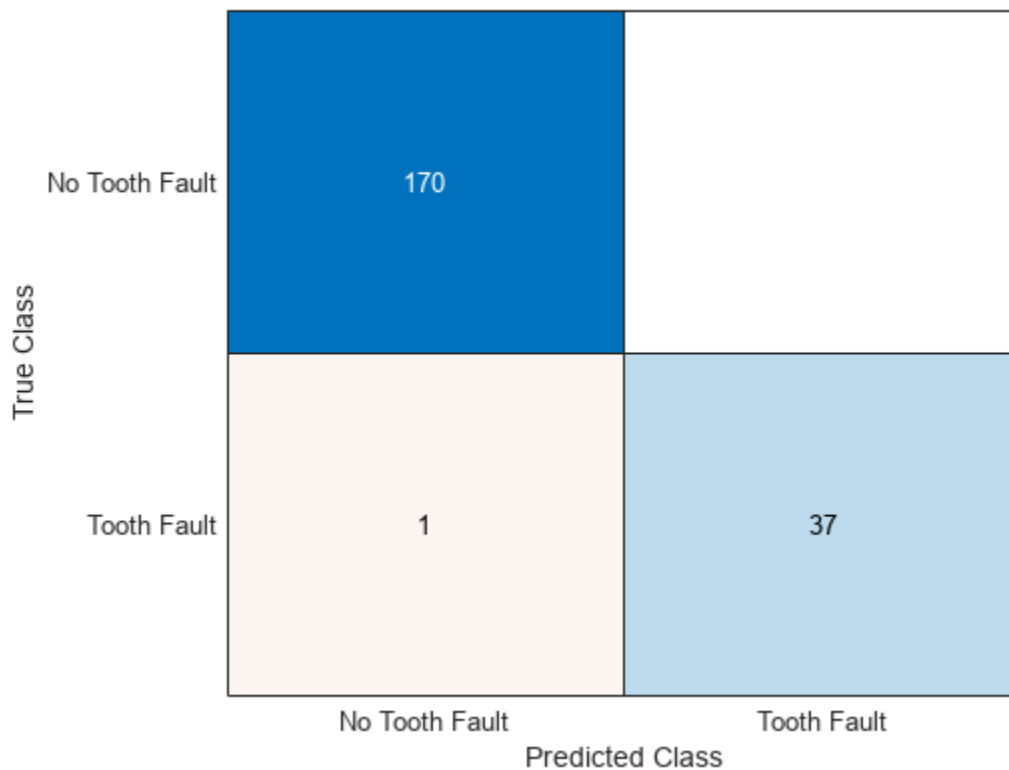
```
TTest = tbl{:,labelName};
```

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

```
YTest = classify(net,tbl(:,1:end-1));
```

Visualize the predictions in a confusion matrix.

```
figure
confusionchart(TTest,YTest)
```



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

```
accuracy = mean(YTest == TTest)
```

```
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.

### images — Image data

datastore | numeric array | table

Image data, specified as one of the following.

Data Type		Description	Example Usage
Datastore	ImageDatastore	Datastore of images saved on disk	Make predictions with images saved on disk, where the images are the same size.  When the images are different sizes, use an AugmentedImageDatastore object.
	AugmentedImageDatastore	Datastore that applies random affine geometric transformations, including resizing, rotation, reflection, shear, and translation	Make predictions with images saved on disk, where the images are different sizes.
	TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by classify.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>Make predictions using networks with multiple inputs.</li> <li>Combine predictors from different data sources.</li> </ul>

Data Type		Description	Example Usage
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions using data in a format that other datastores do not support.  For details, see “Develop Custom Mini-Batch Datastore”.
Numeric array		Images specified as a numeric array	Make predictions using data that fits in memory and does not require additional processing like resizing.
Table		Images specified as a table	Make predictions using data stored in a table.

When you use a datastore with 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. Use datastores when you have data that does not fit in memory or when you want to resize the input data.

These datastores are directly compatible with `classify` for image data.:

- `ImageDatastore`
- `AugmentedImageDatastore`
- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch 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 the `imageDatastore` function for preprocessing or resizing, as this option is usually significantly slower.

---

You can use other built-in datastores for making predictions by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the format required by `classify`.

The required format of the datastore output depends on the network architecture.

Network Architecture	Datastore Output	Example Output
Single input	<p>Table or cell array, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom datastores must output tables.</p>	<pre>data = read(ds) data =     4x1 table         Predictors     _____     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}</pre> <pre>data = read(ds) data =     4x1 cell array         {224x224x3 double}         {224x224x3 double}         {224x224x3 double}         {224x224x3 double}</pre>
Multiple input	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>data = read(ds) data =     4x2 cell array         {224x224x3 double} {128x128x3 do         {224x224x3 double} {128x128x3 do         {224x224x3 double} {128x128x3 do         {224x224x3 double} {128x128x3 do</pre>

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

Data	Format
2-D images	<i>h-by-w-by-c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively
3-D images	<i>h-by-w-by-d-by-c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively

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.

The size and shape of the numeric array depends on the type of image data.

<b>Data</b>	<b>Format</b>
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 in a table.

When you specify images 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

---

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`

Complex Number Support: Yes

**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	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by <code>classify</code>.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>Make predictions using networks with multiple inputs.</li> <li>Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>
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	Make predictions using data that fits in memory and does not require additional processing like custom transformations.

### Datastore

Datastores read mini-batches of sequences and responses. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with `classify` for sequence data:

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `classify`. For example, you can transform and combine data read from in-memory arrays and CSV files using an `ArrayDatastore` and an `TabularTextDatastore` object, respectively.

The datastore must return data in a table or cell array. Custom mini-batch datastores must output tables.

Datastore Output	Example Output
Table	<pre>data = read(ds) data =     4x2 table         Predictors         _____         {12x50 double}         {12x50 double}         {12x50 double}         {12x50 double}</pre>
Cell array	<pre>data = read(ds) data =     4x2 cell array         {12x50 double}         {12x50 double}         {12x50 double}         {12x50 double}</pre>

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

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

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.

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.

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.

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** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

### features – Feature data

`datastore` | `numeric array` | `table`

Feature data, specified as one of the following.

Data Type	Description	Example Usage
Datastore	TransformedDatastore	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by <code>classify</code>.</li> <li>Apply custom transformations to datastore output.</li> </ul>

Data Type		Description	Example Usage
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>
Table		Feature data specified as a table	Make predictions using data stored in a table.
Numeric array		Feature data specified as numeric array	Make predictions 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. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with `classify` for feature data:

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `classify`. 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	<p>Table or cell array with at least one column, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom mini-batch datastores must output tables.</p>	<p>Table for network with one input:</p> <pre>data = read(ds) data =     4x2 table         Predictors     _____     {24x1 double}     {24x1 double}     {24x1 double}     {24x1 double}</pre>
		<p>Cell array for network with one input:</p> <pre>data = read(ds) data =     4x1 cell array         {24x1 double}         {24x1 double}         {24x1 double}         {24x1 double}</pre>
Multiple input layers	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<p>Cell array for network with two inputs:</p> <pre>data = read(ds) data =     4x3 cell array         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}</pre>

The predictors must be *c*-by-1 column vectors, where *c* is the number of features.

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 in the table columns depends on the type of task.

Task	Predictors
Feature classification	Features specified in one or more columns as scalars.

**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.

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** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`  
 Complex Number Support: Yes

**X1, . . . , XN — Numeric or cell arrays for networks with multiple inputs**

`numeric array` | `cell array`

Numeric or cell arrays for networks with multiple inputs.

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`  
 Complex Number Support: Yes

**mixed — Mixed data**

`TransformedDatastore` | `CombinedDatastore` | `custom mini-batch datastore`

Mixed data, specified as one of the following.

Data Type	Description	Example Usage
TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Transform outputs of datastores not supported by <code>classify</code> so they have the required format.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>

You can use other built-in datastores for making predictions 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 `classify`. For more information, see “Datastores for Deep Learning”.

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.

Datastore Output	Example Output
<p>Cell array with <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>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}</pre>

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To convert a numeric array to a datastore, use `arrayDatastore`.

## 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in 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 you make 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Acceleration — Performance optimization

"auto" (default) | "mex" | "none"

Performance optimization, specified as 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 only when you use a GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- "none" — Disable all acceleration.

If `Acceleration` is "auto", then MATLAB applies a number of compatible optimizations and does not generate a MEX function.

The "auto" and "mex" options can offer performance benefits 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 supports networks that contain the layers listed on the "Supported Layers" (GPU Coder) page, except for `sequenceInputLayer` objects.

The "mex" option is available when you use a single GPU.

To use the "mex" option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning 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.

For quantized networks, the "mex" option requires a CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3, or higher.

MATLAB Compiler does not support deploying networks when you use the "mex" option.

### ExecutionEnvironment — Hardware resource

"auto" (default) | "gpu" | "cpu" | "multi-gpu" | "parallel"

Hardware resource, specified as one of the following:

- "auto" — Use a GPU if one is available; otherwise, use the CPU.
- "gpu" — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (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".

The "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 Computing Requirements" (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 make predictions in parallel with networks with recurrent layers (by setting ExecutionEnvironment to either "multi-gpu" or "parallel"), the SequenceLength option must be "shortest" or "longest".

Networks with custom layers that contain State parameters do not support making predictions in parallel.

### SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is `'sequence'` for each recurrent 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**

`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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Output Arguments**

### **Y — 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 Y 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	
Sequence-to-sequence classification	<p><math>N</math>-by-1 cell array of categorical sequences of labels, where <math>N</math> is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after the <code>SequenceLength</code> option is applied to each mini-batch independently.</p> <p>For sequence-to-sequence classification tasks with one observation, <code>sequences</code> can be a matrix. In this case, <math>Y</math> is a categorical sequence of labels.</p>

### 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 the <code>SequenceLength</code> option is applied 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

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

## Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Alternatives

To classify data using a network with multiple output layers, use the `predict` function and set the `ReturnCategorical` option to 1 (true).

To compute the predicted classification scores, you can also use the `predict` function.

To compute the activations from a network layer, use the `activations` function.

For recurrent networks such as LSTM networks, you can make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.

## Version History

### Introduced in R2016a

### **R2022b: Prediction functions pad mini-batches to length of longest sequence before splitting when you specify `SequenceLength` option as an integer**

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

## References

- [1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Multidimensional Curve Classification Using Passing-through Regions." *Pattern Recognition Letters* 20, no. 11–13 (November 1999): 1103–11. [https://doi.org/10.1016/S0167-8655\(99\)00077-X](https://doi.org/10.1016/S0167-8655(99)00077-X).
- [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:
  - `[Y,scores] = classify(net,images)`, where `images` is a numeric array
  - `[Y,scores] = classify(net,sequences)`, where `sequences` is a cell array
  - `[Y,scores] = classify(net,features)`, where `features` is a numeric array
  - `[Y,scores] = classify(__,Name=Value)` using any of the previous syntaxes.
- 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:
  - `Y = classify(net,images)`, where `images` is a numeric array
  - `Y = classify(net,sequences)`, where `sequences` is a cell array or numeric array
  - `Y = classify(net,features)`, where `features` is a numeric array
  - `[Y,scores] = classify(__)` using any of the previous syntaxes.
  - `[Y,scores] = classify(__,Name=Value)` using any of the previous syntaxes.
- 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™.

- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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"

# classifyAndUpdateState

Classify data using a trained recurrent neural network and update the network state

## Syntax

```
[updatedNet,Y] = classifyAndUpdateState(recNet,sequences)
[updatedNet,Y] = classifyAndUpdateState(recNet,X1,...,XN)
[updatedNet,Y] = classifyAndUpdateState(recNet,mixed)
[updatedNet,Y,scores] = classifyAndUpdateState(____)
____ = classifyAndUpdateState(____,Name=Value)
```

## Description

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value argument.

For networks with multiple outputs, use the `predictAndUpdateState` function instead and set the `ReturnCategorical` option to `true`.

`[updatedNet,Y] = 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 such as an LSTM layer or a custom layer with state parameters.

`[updatedNet,Y] = classifyAndUpdateState(recNet,X1,...,XN)` predicts the class labels for the data in the numeric arrays or cell arrays `X1, ..., XN` for the multi-input network `recNet`. The input `Xi` corresponds to the network input `recNet.InputNames(i)`.

`[updatedNet,Y] = classifyAndUpdateState(recNet,mixed)` predicts the class labels for the multi-input network `recNet` with data of mixed data types.

`[updatedNet,Y,scores] = classifyAndUpdateState(____)` also returns the classification scores corresponding to the class labels using any of the previous syntaxes.

`____ = classifyAndUpdateState(____,Name=Value)` predicts class labels with additional options specified by one or more name-value arguments using any of the previous syntaxes. For example, `MiniBatchSize=27` classifies data using mini-batches of size 27.

---

**Tip** When you make 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.

---

“Classify and Update Network State” on page 1-330

## 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, TTest] = 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.

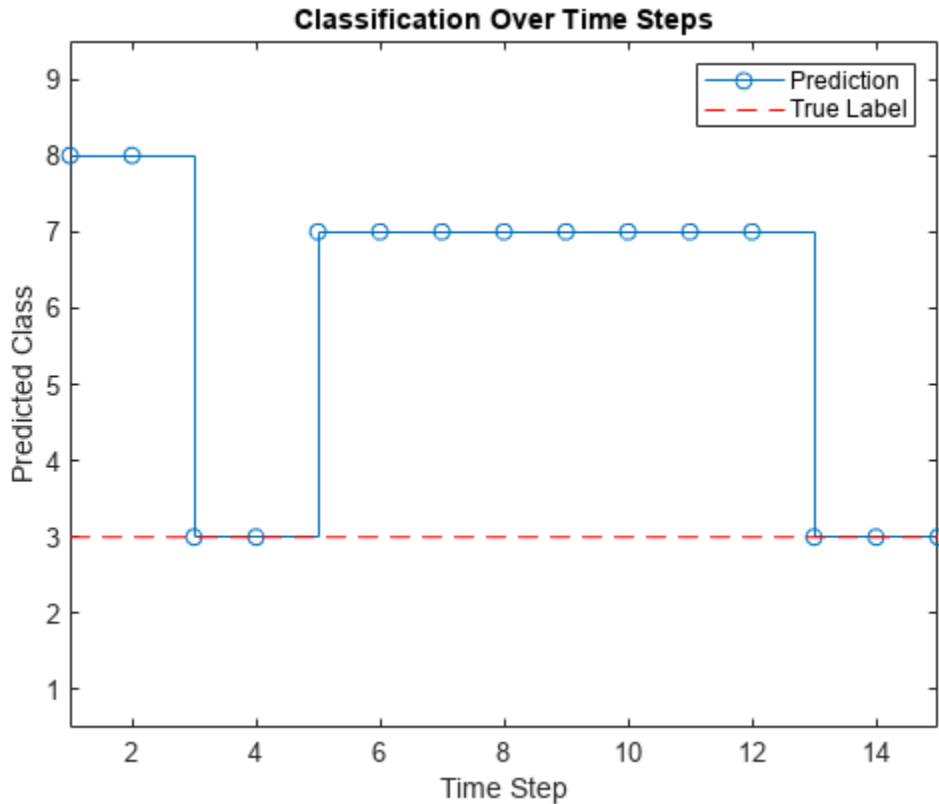
```
trueLabel = TTest(94)

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.

---

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

**X1, . . . , XN — Numeric or cell arrays for networks with multiple inputs**

numeric array | cell array

Numeric or cell arrays for networks with multiple inputs.

For sequence predictor input, the input must be a numeric array representing a single sequence or a cell array of sequences, where the format of the predictors match the formats described in the `sequences` argument description. For image and feature predictor input, the input must be a numeric array and the format of the predictors must match the one of the following:

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.

Data	Format
Feature data	$c$ -by-1 column vectors, where $c$ is the number of features.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

**Tip** To input complex-valued data into a network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`  
 Complex Number Support: Yes

**mixed — Mixed data**

`TransformedDatastore` | `CombinedDatastore` | custom mini-batch datastore

Mixed data, specified as one of the following.

Data Type	Description	Example Usage
<code>TransformedDatastore</code>	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Transform outputs of datastores not supported by <code>classifyAndUpdateState</code> so they have the required format.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
<code>CombinedDatastore</code>	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions using data in a format that other datastores do not support.  For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `classifyAndUpdateState`. For more information, see “Datastores for Deep Learning”.

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.

Datastore Output	Example Output
<p>Cell array with <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>data = read(ds) data =     4x3 cell array     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}</pre>

For sequence predictor input, the input must be a numeric array representing a single sequence or a cell array of sequences, where the format of the predictors match the formats described in the `sequences` argument description. For image and feature predictor input, the input must be a numeric array and the format of the predictors must match the one of the following:

Data	Format
2-D images	<i>h</i> -by- <i>w</i> -by- <i>c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively.
3-D images	<i>h</i> -by- <i>w</i> -by- <i>d</i> -by- <i>c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively.
Feature data	<i>c</i> -by-1 column vectors, where <i>c</i> is the number of features.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To convert a numeric array to a datastore, use `ArrayDatastore`.

---

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `[updatedNet,Y] = 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 you make 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 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.

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.

### ExecutionEnvironment — Hardware resource

"auto" (default) | "gpu" | "cpu"

Hardware resource, specified as one of the following:

- "auto" — Use a GPU if one is available; otherwise, use the CPU.
- "gpu" — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- "cpu" — Use the CPU.

### SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

**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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is `'sequence'` for each recurrent 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**`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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Output Arguments****updatedNet — Updated network**`SeriesNetwork` object | `DAGNetwork` object

Updated network. `updatedNet` is the same type of network as the input network.

**Y — 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 Y depends on the type of problem.

The following table describes the format of Y.

Task	Format
Sequence-to-label classification	$N$ -by-1 categorical vector of labels, where $N$ is the number of observations.

Task	Format
Sequence-to-sequence classification	<p><math>N</math>-by-1 cell array of categorical sequences of labels, where <math>N</math> is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the <code>SequenceLength</code> option to each mini-batch independently.</p> <p>For sequence-to-sequence classification problems with one observation, <code>sequences</code> can be a matrix. In this case, <code>YPred</code> is a categorical sequence of labels.</p>

### 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 <code>SequenceLength</code> 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

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Alternatives

To classify data using a recurrent neural network with multiple output layers and update the network state, use the `predictAndUpdateState` function and set the `ReturnCategorical` option to 1 (true).

To compute the predicted classification scores and update the network state of a recurrent neural network, you can also use the `predictAndUpdateState` function.

To compute the activations of a network layer, use the `activations` function. The `activations` function does not update the network state.

To make predictions without updating the network state, use the `classify` function or the `predict` function.

## Version History

### Introduced in R2017b

### **R2022b: Prediction functions pad mini-batches to length of longest sequence before splitting when you specify `SequenceLength` option as an integer**

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

## 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, Y] = classifyAndUpdateState(recNet, sequences)`, where `sequences` is cell array or numeric array.
  - `[updatedNet, Y, scores] = classifyAndUpdateState(recNet, sequences)`, where `sequences` is cell array.
  - `__ = classifyAndUpdateState(recNet, sequences, Name=Value)` using any of the previous syntaxes
- 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 arguments are supported for code generation. All name-value arguments must be compile-time constants.
- Only the "longest" and "shortest" option of the `SequenceLength` name-value argument 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, Y] = classifyAndUpdateState(recNet, sequences)`, where `sequences` is cell array.
  - `[updatedNet, Y, scores] = classifyAndUpdateState(recNet, sequences)`, where `sequences` is cell array.
  - `__ = classifyAndUpdateState(__, Name=Value)` using any of the previous syntaxes
- GPU code generation for the `classifyAndUpdateState` function is only supported for recurrent neural networks targeting cuDNN and TensorRT libraries.
- GPU code generation does not support `gpuArray` inputs to the `classifyAndUpdateState` function.
- 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 arguments are supported for code generation. All name-value arguments must be compile-time constants.
- Only the "longest" and "shortest" options of the `SequenceLength` name-value argument is supported for code generation.

### GPU Arrays

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

- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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 Neural Networks"

"Specify Layers of Convolutional Neural Network"

"Set Up Parameters and Train Convolutional Neural Network"

"Deep Learning in MATLAB"

## 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");
net = s.net;
classNames = s.classNames;
```

Accelerate the model loss function `modelLoss` listed at the end of the example.

```
fun = @modelLoss;
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: @modelLoss
    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");
X = dlarray(X,"SSCB");

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

[loss,gradients,state] = dlfeval(accfun,net,X,T);
```

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 Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding target labels `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)

[Y,state] = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);

end
```

## Input Arguments

### **accfun** — Accelerated function

`AcceleratedFunction` object

Accelerated function, specified as an `AcceleratedFunction` object.

## Version History

Introduced in R2021a



**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”](#)

## 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:

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

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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	Clipped ReLU	Clipped ReLU with ceiling 10
4	''	2-D 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

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

`ClippedReLU` objects apply an element-wise operation and supports input data of any format. The layer does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

**Introduced in R2017b**

## 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 Neural 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"](#)

## compressNetworkUsingProjection

Compress neural network using projection

### Syntax

```
netProjected = compressNetworkUsingProjection(net,mbq)
netProjected = compressNetworkUsingProjection(net,X1,...,XN)
netProjected = compressNetworkUsingProjection(net,npca)
[netProjected, info] = compressNetworkUsingProjection( ___ )
[netProjected, info] = compressNetworkUsingProjection( ___ ,Name=Value)
```

### Description

The `compressNetworkUsingProjection` function reduces the number of learnable parameters of layers by performing principal component analysis (PCA) of the neuron activations using a data set representative of the training data and then projects the learnable parameters into the subspace that maintains the highest variance in neuron activations.

Forward passes of a projected deep neural network are typically faster when you deploy the network to embedded hardware using library-free C or C++ code generation.

If you prune or quantize your network, then use compression using projection after pruning and before quantization. Network compression using projection supports projecting LSTM layers only.

`netProjected = compressNetworkUsingProjection(net,mbq)` compresses the `dlnetwork` object `net` by replacing layers with projected layers. The function compresses layers by performing principal component analysis (PCA) of the neuron activations using the data in the `minibatchqueue` object `mbq` and projects learnable parameters into the subspace that maintains the highest variance in neuron activations. This function requires the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

`netProjected = compressNetworkUsingProjection(net,X1,...,XN)` compresses the network using the data in the `dlarray` objects `X1,...,XN`, where `N` is the number of network inputs.

`netProjected = compressNetworkUsingProjection(net,npca)` compresses the network using the `neuronPCA` object `npca`. The PCA step can be computationally intensive. If you expect to compress the same network multiple times (for example, when exploring different levels of compression), then you can perform the PCA step up front using a `neuronPCA` object.

`[netProjected, info] = compressNetworkUsingProjection( ___ )` also returns the structure `info` that contains information about the reduction of learnable parameters and the explained variance achieved during compression.

`[netProjected, info] = compressNetworkUsingProjection( ___ ,Name=Value)` specifies additional options using one or more name-value arguments.

### Examples

## Compress Network Using Projection

Load the pretrained network in `dlnetJapaneseVowels` and the training data.

```
load dlnetJapaneseVowels
XTrain = japaneseVowelsTrainData;
```

Create a mini-batch queue containing the training data. To create a mini-batch queue from in-memory data, convert the sequences to an array datastore.

```
adsXTrain = arrayDatastore(XTrain,OutputType="same");
```

Create the `minibatchqueue` object.

- Specify a mini-batch size of 16.
- Preprocess the mini-batches using the `preprocessMiniBatchPredictors` function, listed in the Mini-Batch Predictors Preprocessing Function on page 1-349 section of the example.
- Specify that the output data has format "CTB" (channel, time, batch).

```
miniBatchSize = 16;
```

```
mbq = minibatchqueue(adsXTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatchPredictors, ...
    MiniBatchFormat="CTB");
```

Compress the network.

```
netProjected = compressNetworkUsingProjection(net,mbq);
```

Compressed network has 82.4% fewer learnable parameters.  
Projected layers explain on average 96.6% of layer activation variance.

### Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the sequence data from the input cell array and truncating them along the second dimension so that they have the same length.

**Note:** Do not pad sequence data when doing the PCA step for projection as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

```
function X = preprocessMiniBatchPredictors(dataX)
X = padsequences(dataX,2,Length="shortest");
end
```

### Determine Maximum Possible Compression

To determine the maximum possible compression, set the `LearnablesReductionGoal` option to 1 or the `ExplainedVarianceGoal` option to 0.

Load the pretrained network in `dlnetJapaneseVowels` and the training data.

```
load dlnetJapaneseVowels
XTrain = japaneseVowelsTrainData;
```

Create a mini-batch queue containing the training data. To create a mini-batch queue from in-memory data, convert the sequences to an array datastore.

```
adsXTrain = arrayDatastore(XTrain,OutputType="same");
```

Create the minibatchqueue object.

- Specify a mini-batch size of 16.
- Preprocess the mini-batches using the `preprocessMiniBatchPredictors` function, listed in the Mini-Batch Predictors Preprocessing Function on page 1-350 section of the example.
- Specify that the output data has format "CTB" (channel, time, batch).

```
miniBatchSize = 16;
```

```
mbq = minibatchqueue(adsXTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatchPredictors, ...
    MiniBatchFormat="CTB");
```

Compress the network. To determine the maximum possible compression, set the `LearnablesReductionGoal` option to 1.

```
[netProjected,info] = compressNetworkUsingProjection(net,mbq,LearnablesReductionGoal=1);
```

```
Compressed network has 95.2% fewer learnable parameters.
Projected layers explain on average 33.8% of layer activation variance.
```

View the proportion of total number of network learnables removed by inspecting the `LearnablesReduction` property of the information structure.

```
info.LearnablesReduction
```

```
ans = 0.9518
```

### Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the sequence data from the input cell array and truncating them along the second dimension so that they have the same length.

**Note:** Do not pad sequence data when doing the PCA step for projection as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

```
function X = preprocessMiniBatchPredictors(dataX)
```

```
X = padsequences(dataX,2,Length="shortest");
```

```
end
```

### Input Arguments

**net** — Neural network

initialized dlnetwork object



Neural network, specified as an initialized `dlnetwork` object.

### **mbq — Mini-batch queue**

`minibatchqueue` object

Mini-batch queue that outputs data for each input of the network, specified as a `minibatchqueue` object.

The PCA step typically works best when using the full training set. However, any dataset that is representative of the training data distribution suffices. The input data must contain two or more observations and sequences must contain two or more time steps.

---

**Note** Do not pad sequence as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

---

### **X1, . . . , XN — Input data**

formatted `dlarray`

Input data, specified as a formatted `dlarray`.

For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

The PCA step typically works best when using the full training set. However, any dataset that is representative of the training data distribution suffices. The input data must contain two or more observations and sequences must contain two or more time steps.

---

**Note** Do not pad sequence as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

---

### **npca — Neuron principal component analysis**

`neuronPCA` object

Neuron principal component analysis, specified as a `neuronPCA` object.

### **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: `netProjected = compressNetworkUsingProjection(net,mbq,VerbosityLevel="off")` compresses the network using projection and disables the command line display.

### **LayerNames — Names of layers to compress**

string array | cell array of character vectors | character vector

Names of layers to compress, specified as a string array, cell array of character vectors, or a character vector containing a single layer name.

The software, by default, compress all the layers in the network that support projection. Network compression using projection supports projecting LSTM layers only.

Data Types: `string` | `cell`

**ExplainedVarianceGoal — Target proportion of neuron activation variance explained by principal components**

0.95 (default) | nonnegative scalar less than or equal to 1

Target proportion of neuron activation variance explained by the remaining principal components of each projected layer, specified as a value between 0 (maximum compression) and 1 (project layers with minimal compression).

If you specify the `ExplainedVarianceGoal` option, then you must not specify the `LearnablesReductionGoal` option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LearnablesReductionGoal — Target proportion of total number of network learnables to remove**

nonnegative scalar less than or equal to 1

Target proportion of total number of network learnables to remove, specified as a nonnegative scalar less than or equal to 1.

If you specify the `LearnablesReductionGoal` option, then you must not specify the `ExplainedVarianceGoal` option. If you do not specify the `LearnablesReductionGoal` option, then the function compresses the network using the `ExplainedVarianceGoal` option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**VerbosityLevel — Verbosity level**

"summary" (default) | "steps" | "iterations" | "off"

Verbosity level, specified as one of these values:

- "summary" — Display a summary of the compression algorithm.
- "steps" — Display information about the steps of the compression algorithm.
- "iterations" — Display information about the iterations of the compression algorithm.
- "off" — Do not display information.

## Output Arguments

**netProjected — Projected network**

`dlnetwork` object

Projected network, returned as a `dlnetwork` object.

After you compress the network using projection, you can fine-tune the network to help regain predictive accuracy lost by the compression process. For an example, see “Compress Neural Network Using Projection”.

**info — Projection information**

structure

Projection information, returned as a structure with these fields:

- `LearnablesReduction` — proportion of total number of network learnables removed
- `ExplainedVariance` — proportion of neuron activation variance explained by principal components

## Version History

Introduced in R2022b

## 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 any of these conditions are met:
  - Any of the values of the network learnable parameters inside `net.Learnables.Value` are `dLarray` objects with underlying data of type `gpuArray`
  - The input argument `mbq` outputs `dLarray` objects with underlying data of type `gpuArray`
  - The input arguments `X1`, . . . , `XN` are `dLarray` objects with underlying data of type `gpuArray`

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

## See Also

`dlnetwork` | `minibatchqueue` | `dLarray` | `neuronPCA` | `lstmProjectedLayer`

### Topics

“Compress Neural Network Using Projection”

“Parameter Pruning and Quantization of Image Classification Network”

“Quantization of Deep Neural Networks”

# 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. The layer supports concatenating over channel, batch, and spatial dimensions only.

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 the name ' '.

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')
```

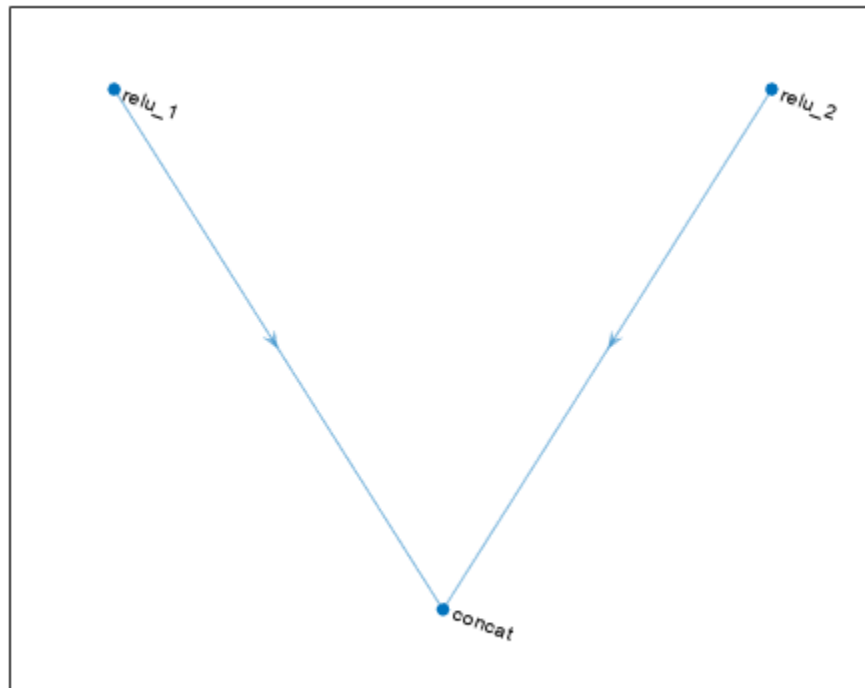
```
concat =
```

```
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)
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time

- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `ConcatenationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

The layer supports concatenation over "S" (spatial), "C" (channel), and "B" (batch) dimensions only. The output format matches the input format. Each input must have data of the same format.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `ConcatenationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2019a

## **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”



# 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');
```

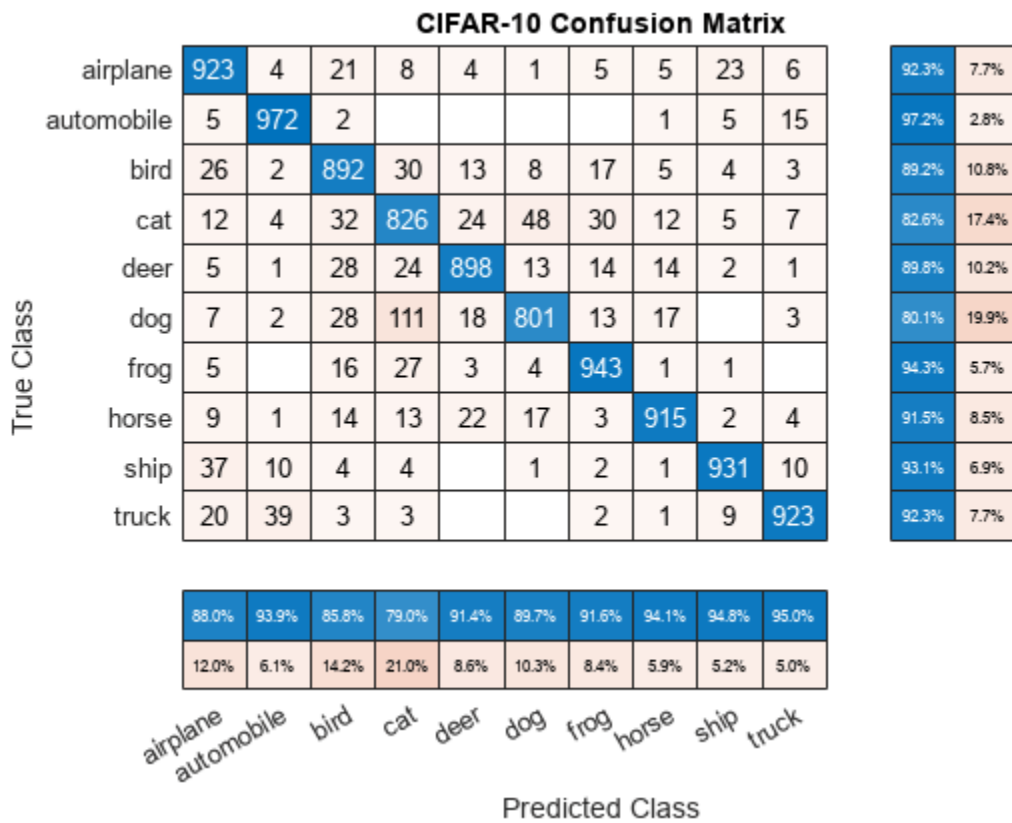
Create a confusion matrix chart.

```
figure
cm = confusionchart(trueLabels,predictedLabels);
```

True Class	airplane	923	4	21	8	4	1	5	5	23	6
	automobile	5	972	2					1	5	15
	bird	26	2	892	30	13	8	17	5	4	3
	cat	12	4	32	826	24	48	30	12	5	7
	deer	5	1	28	24	898	13	14	14	2	1
	dog	7	2	28	111	18	801	13	17		3
	frog	5		16	27	3	4	943	1	1	
	horse	9	1	14	13	22	17	3	915	2	4
	ship	37	10	4	4		1	2	1	931	10
	truck	20	39	3	3			2	1	9	923
		airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
		Predicted Class									

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 = 10x10
```

```

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

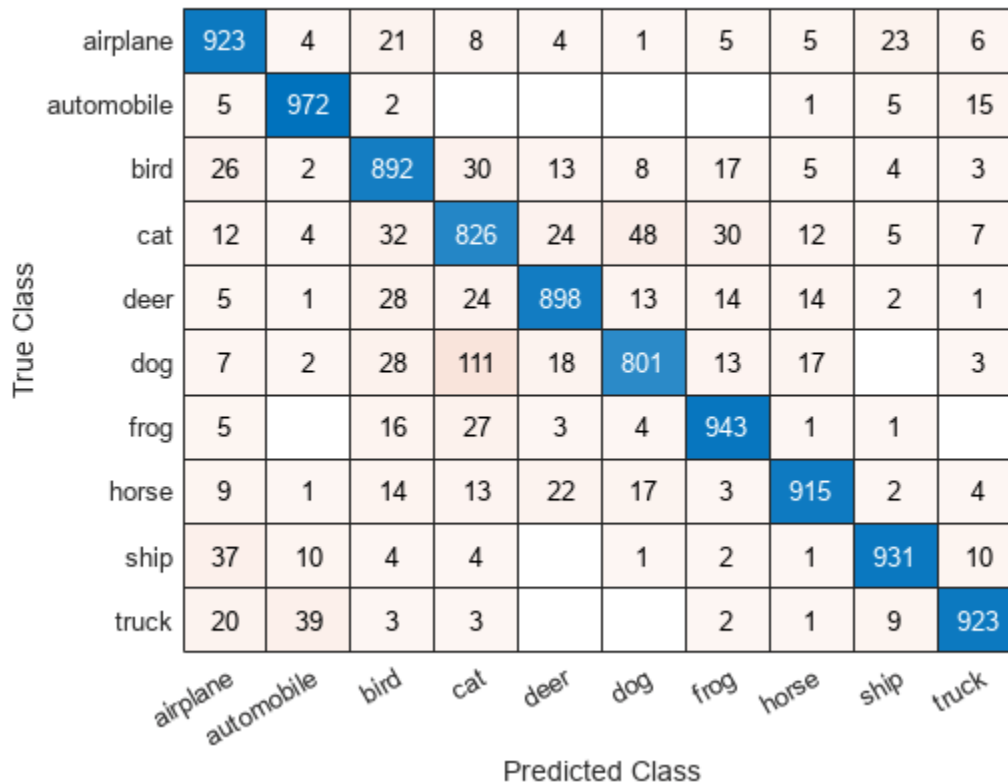
```

```
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);
```

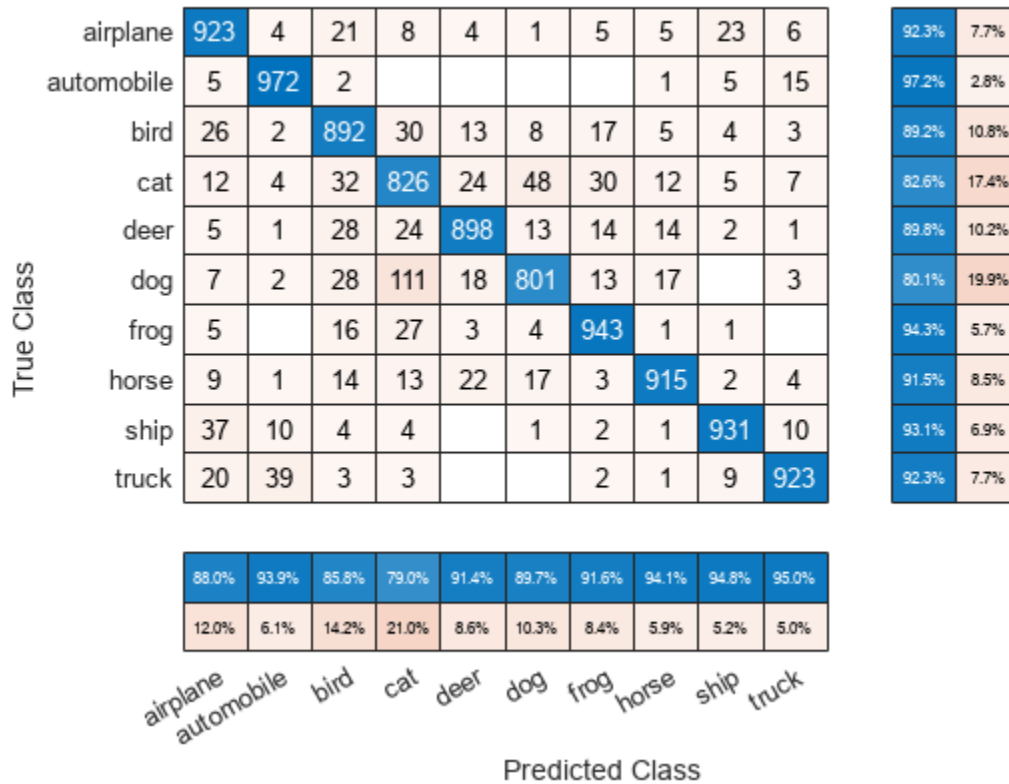


### Sort Classes by Precision or Recall

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');
```



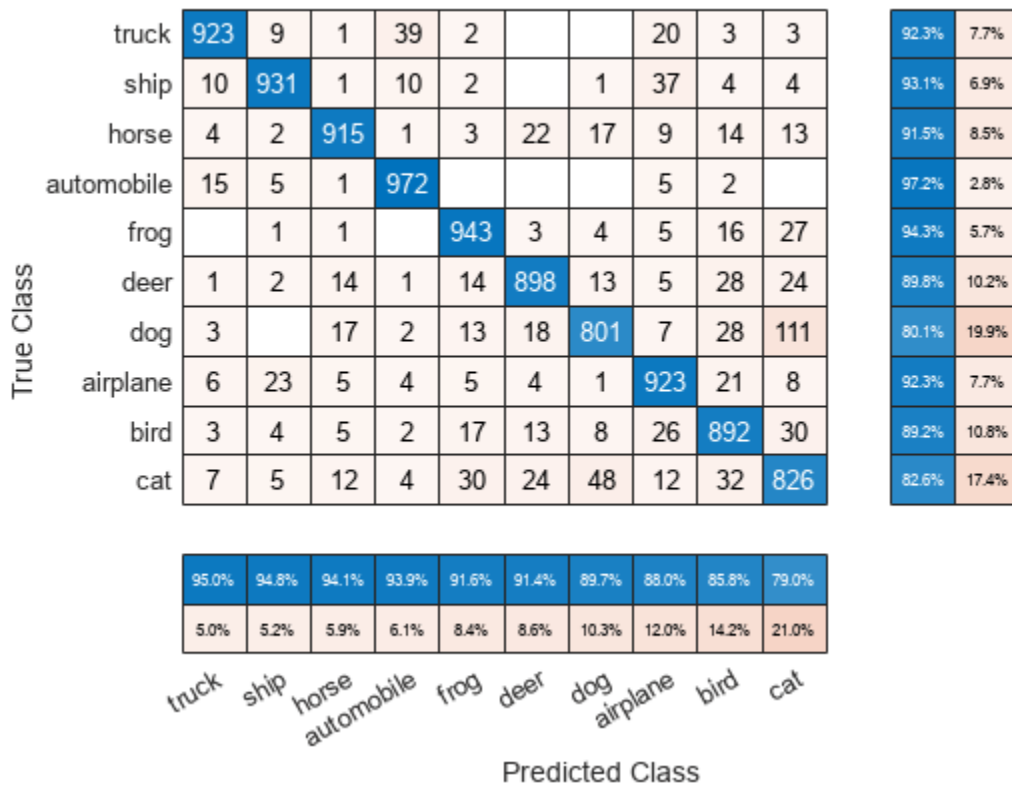
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. 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';
```

True Class	automobile	972		5	5	15	1		2			97.2%	2.8%								
	frog		943	1	5		1	3	16	27	4	94.3%	5.7%								
	ship	10	2	931	37	10	1		4	4	1	93.1%	6.9%								
	airplane	4	5	23	923	6	5	4	21	8	1	92.3%	7.7%								
	truck	39	2	9	20	923	1		3	3		92.3%	7.7%								
	horse	1	3	2	9	4	915	22	14	13	17	91.5%	8.5%								
	deer	1	14	2	5	1	14	898	28	24	13	89.8%	10.2%								
	bird	2	17	4	26	3	5	13	892	30	8	89.2%	10.8%								
	cat	4	30	5	12	7	12	24	32	826	48	82.6%	17.4%								
	dog	2	13		7	3	17	18	28	111	801	80.1%	19.9%								
												93.9%	91.6%	94.8%	88.0%	95.0%	94.1%	91.4%	85.8%	79.0%	89.7%
												6.1%	8.4%	5.2%	12.0%	5.0%	5.9%	8.6%	14.2%	21.0%	10.3%
												automobile	frog	ship	airplane	truck	horse	deer	bird	cat	dog
												Predicted Class									

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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 1-368. 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).

## Version History

Introduced in R2018b

## See Also

### Functions

`categorical` | `sortClasses` | `classify` | `confusionmat`

### Properties

`ConfusionMatrixChart` Properties

### Topics

“Compare Deep Learning Models Using ROC Curves”  
“Deep Learning in MATLAB”

## 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 = 10x10
```

```
  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 the confusion matrix as a confusion matrix chart.

```

figure
cm = confusionchart(m,order);

```

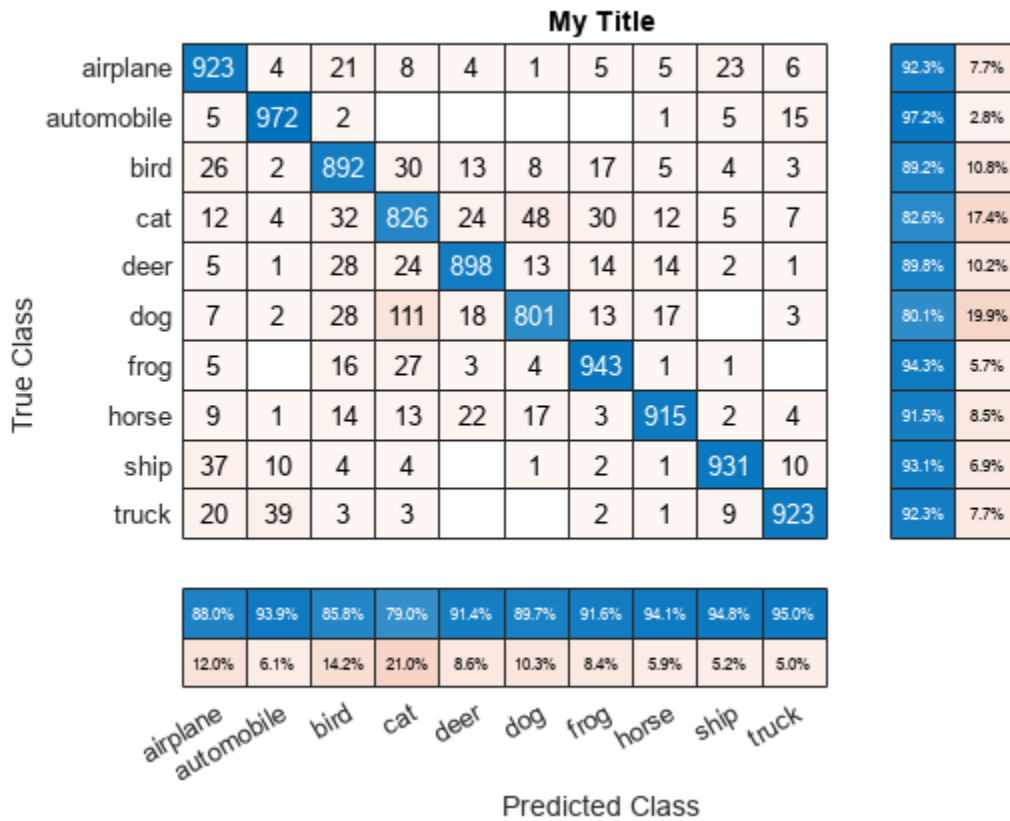
True Class	airplane	923	4	21	8	4	1	5	5	23	6
	automobile	5	972	2					1	5	15
	bird	26	2	892	30	13	8	17	5	4	3
	cat	12	4	32	826	24	48	30	12	5	7
	deer	5	1	28	24	898	13	14	14	2	1
	dog	7	2	28	111	18	801	13	17		3
	frog	5		16	27	3	4	943	1	1	
	horse	9	1	14	13	22	17	3	915	2	4
	ship	37	10	4	4		1	2	1	931	10
	truck	20	39	3	3			2	1	9	923
		airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
		Predicted Class									

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 = 10x10

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' | ...





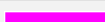
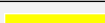


Color for diagonal cells, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color of each diagonal cell is proportional to the cell value and the `DiagonalColor`

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**






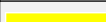


`[0.8510 0.3255 0.0980]` (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

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: <ul style="list-style-type: none"> <li>• Character width = width of letter x.</li> <li>• Character height = distance between the baselines of two lines of text.</li> </ul>
'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: <ul style="list-style-type: none"> <li>• On Windows systems, a pixel is 1/96th of an inch.</li> <li>• On Macintosh systems, a pixel is 1/72nd of an inch.</li> </ul> On Linux systems, the size of a pixel is determined by your system resolution.

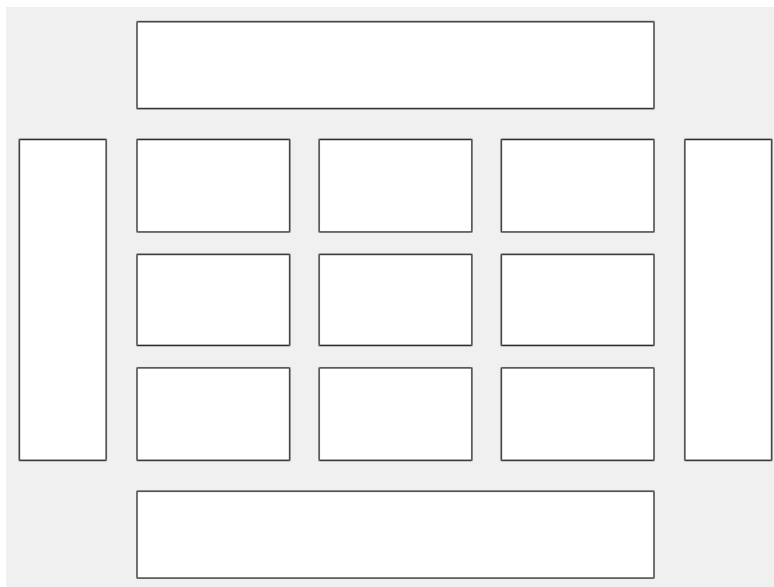
When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units for, such as `OuterPosition`.

### 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.

## **Version History**

**Introduced in R2018b**

## **See Also**

### **Functions**

`categorical` | `confusionchart` | `sortClasses`

### **Topics**

“Deep Learning in MATLAB”

## connectLayers

**Package:** `nnet.cnn`

Connect layers in layer graph or network

### Syntax

```
lgraphUpdated = connectLayers(lgraph,s,d)
netUpdated = connectLayers(net,s,d)
```

### Description

`lgraphUpdated = connectLayers(lgraph,s,d)` connects the source layer `s` to the destination layer `d` in the layer graph `lgraph`. The updated layer graph, `lgraphUpdated`, contains the same layers as `lgraph` and includes the new connection.

`netUpdated = connectLayers(net,s,d)` connects the source layer `s` to the destination layer `d` in the `dlnetwork` object `net`. The updated network, `netUpdated`, contains the same layers as `net` 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'}
```

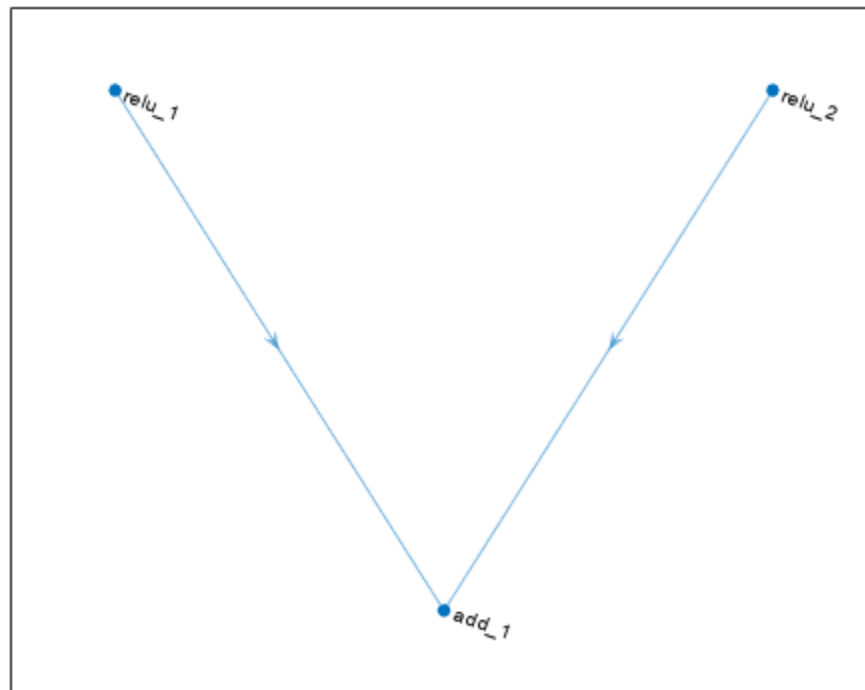
Create two ReLU layers and connect them to the addition layer. The addition layer sums the outputs from the ReLU layers.

```
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];

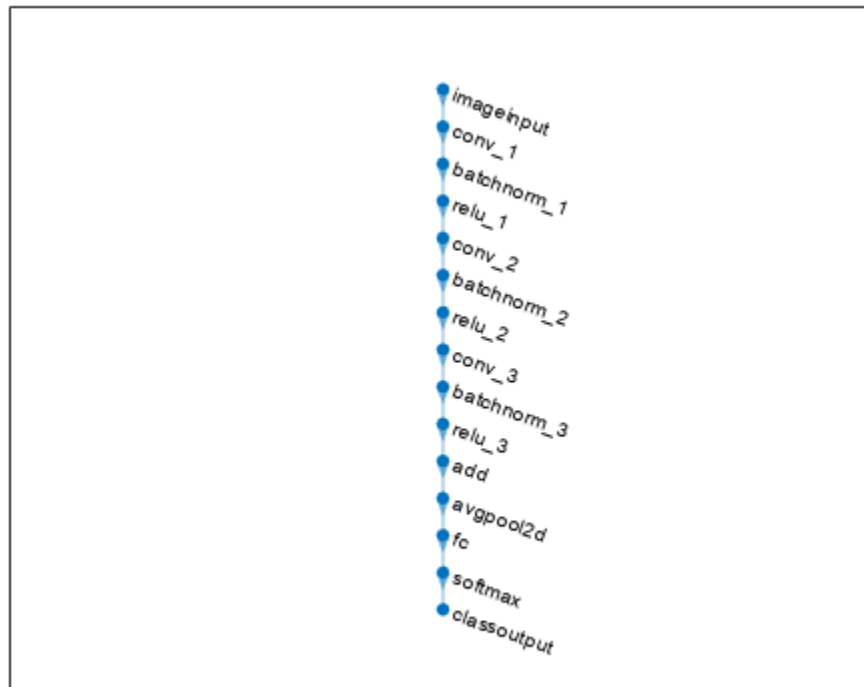
```

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)

```

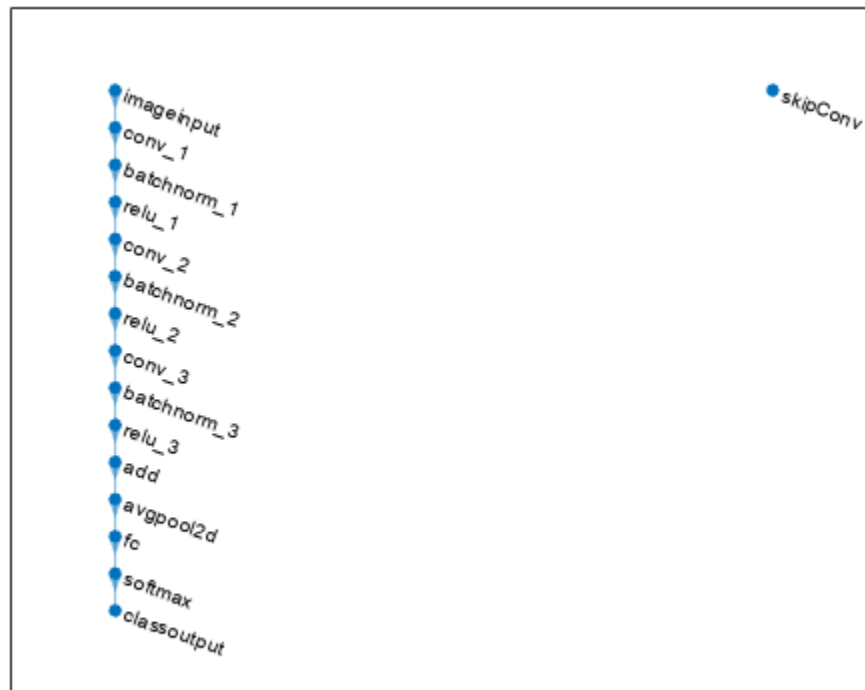


Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the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

```

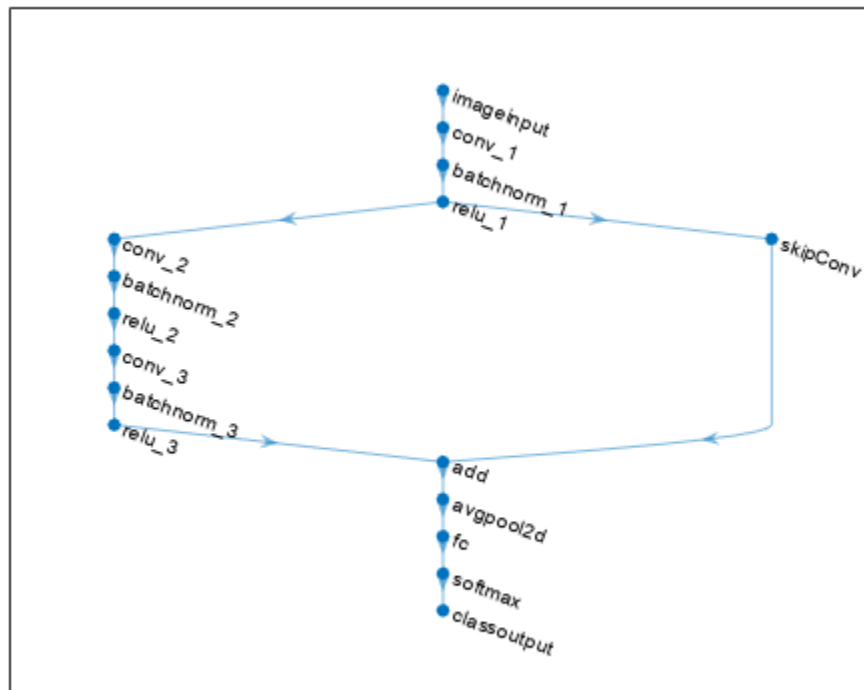
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)

```



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);
```



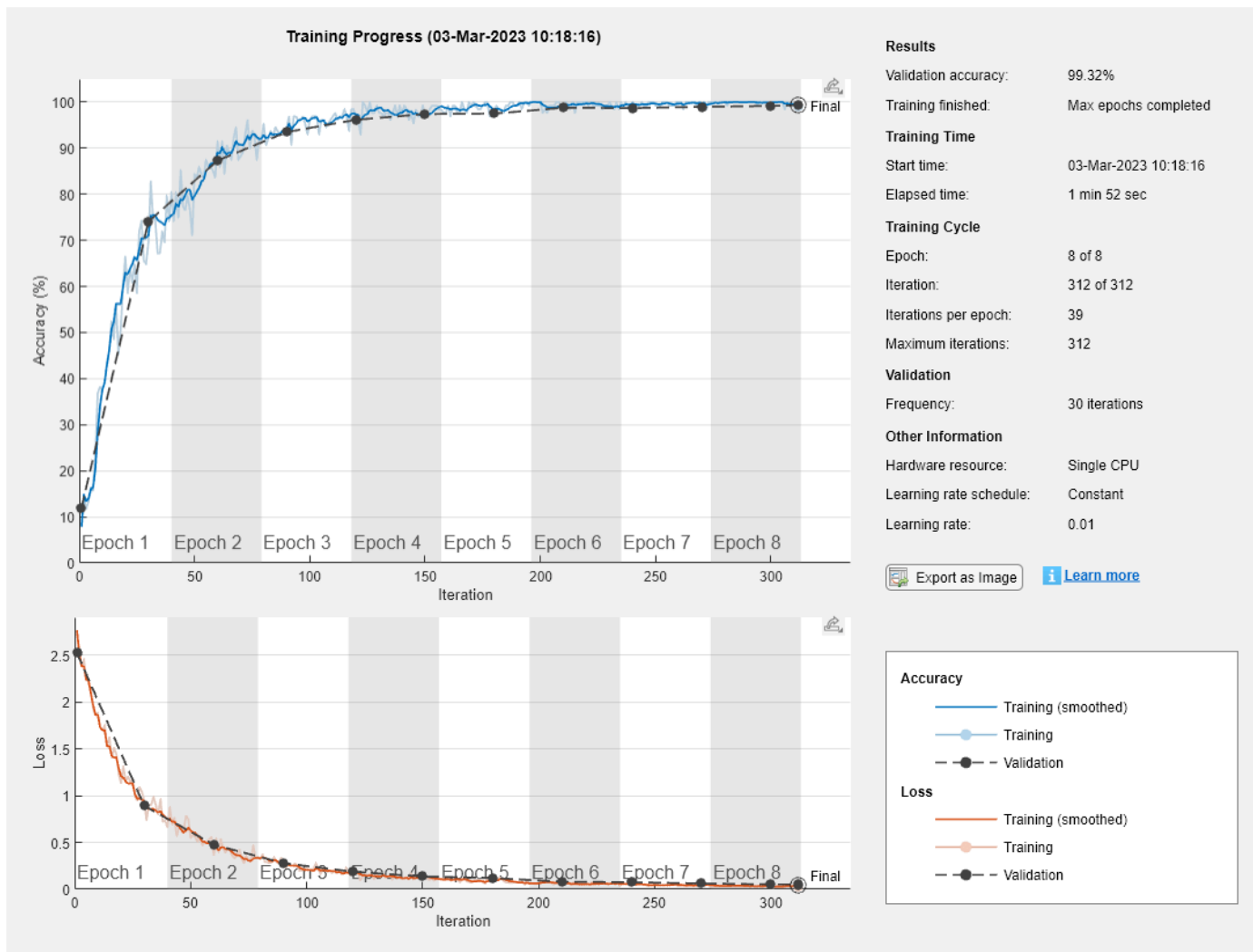
Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[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 = 0.9932
```

## Input Arguments

### **lgraph** — Layer graph

LayerGraph object

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

### **net** — Neural network

dlnetwork object

Neural network, specified as a dlnetwork object.

### **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

### **lgraphUpdated** — Updated layer graph

LayerGraph object

Updated layer graph, returned as a LayerGraph object.

### **netUpdated** — Updated network

dlnetwork object

Updated network, returned as an uninitialized dlnetwork object.

To initialize the learnable parameters of a dlnetwork object, use the `initialize` function.

The `connectLayers` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

Introduced in R2017b

### See Also

layerGraph | addLayers | removeLayers | replaceLayer | disconnectLayers | plot | assembleNetwork | dlnetwork

### Topics

“Train Deep Learning Network to Classify New Images”

“Train Network with Multiple Outputs”

“Classify Videos Using Deep Learning”

# 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 1-397, “Learning Rate and Regularization” on page 1-398, 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $(\text{FilterSize} - 1) \cdot \text{DilationFactor}$ . 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 layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the layer output.

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  $(\text{FilterSize} - 1) \cdot \text{DilationFactor} + 1$ . 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  $\text{ceil}(\text{inputSize}/\text{Stride})$ , 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  $(\text{FilterSize} - 1) \cdot \text{DilationFactor}$ .

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.	<code>[3 1 4] → [0 0 3 1 4 0 0]</code>
<code>'symmetric-include-edge'</code>	Pad using mirrored values of the input, including the edge values.	<code>[3 1 3] → [1 3 3 1 4 4 1]</code>
<code>'symmetric-exclude-edge'</code>	Pad using mirrored values of the input, excluding the edge values.	<code>[3 1 4] → [4 1 3 1 4 1 3]</code>
<code>'replicate'</code>	Pad using repeated border elements of the input.	<code>[3 1 3] → [3 3 3 1 4 4 4]</code>

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **NumChannels — Number of input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

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  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize} * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with a mean of zero and a variance of  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$ .
- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.

- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

### **Weights — Layer weights**

`[]` (default) | numeric array

Layer weights for the transposed convolution operation, specified as a `FilterSize-by-NumChannels-by-numFilters` numeric array or `[]`.

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 transposed convolutional operation, specified as a `1-by-NumFilters` numeric array or `[]`.

The layer biases are learnable parameters. When you train a neural 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`.

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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

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(3,MinLength=20)  
    convolution1dLayer(11,96)  
    reluLayer  
    globalMaxPooling1dLayer  
    fullyConnectedLayer(10)
```

```

softmaxLayer
classificationLayer]

layers =
  7x1 Layer array with layers:

   1  ''  Sequence Input           Sequence input with 3 dimensions
   2  ''  1-D 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 to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, you can represent vector sequence data as a 3-D array, in which the first dimension corresponds to the channel dimension, the second dimension corresponds to the batch dimension, and the third dimension corresponds to the time dimension. This representation is in the format "CBT" (channel, batch, time).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `Convolution1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)

In `dlnetwork` objects, `Convolution1DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)

## Version History

Introduced in R2021b

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `biLstmLayer` | `gruLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer` | `transposedConv1dLayer`

## 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 Neural Networks"  
 "List of Deep Learning Layers"

“Deep Learning Tips and Tricks”

# 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.

The dimensions that the layer convolves over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer convolves over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the two spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer convolves over the spatial and time dimensions.

## 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 1-408, “Learning Rate and Regularization” on page 1-410, 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

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.

Before R2021a, use commas to separate each name and value, and enclose Name in 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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

### 2-D 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 layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the layer output.

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  $(Filter\ Size - 1) \cdot Dilation\ Factor + 1$ . 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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.	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 4 & 0 \\ 0 & 0 & 1 & 5 & 9 & 0 \\ 0 & 0 & 2 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
'symmetric-include-edge'	Pad using mirrored values of the input, including the edge values.	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 1 & 1 & 5 & 9 & 9 & 5 \\ 1 & 3 & 3 & 1 & 4 & 4 & 1 \\ 1 & 3 & 3 & 1 & 4 & 4 & 1 \\ 5 & 1 & 1 & 5 & 9 & 9 & 5 \\ 6 & 2 & 2 & 6 & 5 & 5 & 6 \\ 6 & 2 & 2 & 6 & 5 & 5 & 6 \\ 5 & 1 & 1 & 5 & 9 & 9 & 5 \end{bmatrix}$

PaddingValue	Description	Example
'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 input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. NumChannels and the number of channels in the layer input data must match. For example, if the input is an RGB image, then NumChannels must be 3. If the input is the output of a convolutional layer with 16 filters, then NumChannels must be 16.

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 [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2 / (\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance  $2 / \text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$ .

- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name `''`.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D 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
```

## Algorithms

### 2-D 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 dimensions that the layer convolves over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer convolves over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the two spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer convolves over the spatial and time dimensions.

The convolutional layer consists of various components.<sup>1</sup>

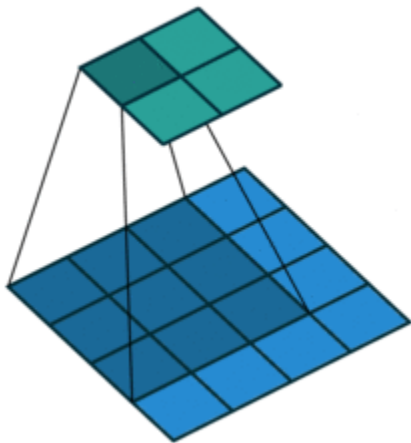
### Filters and Stride

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.

<sup>1</sup> Image credit: Convolution arithmetic (License)

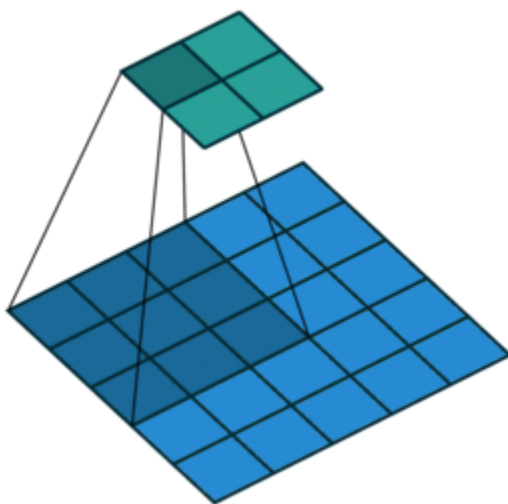
For each region, the `trainNetwork` function computes a dot product of the weights and the input, and then adds a bias term. A set of weights that is applied to a region in the image is called a *filter*. The filter moves along the input image vertically and horizontally, repeating the same computation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

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.

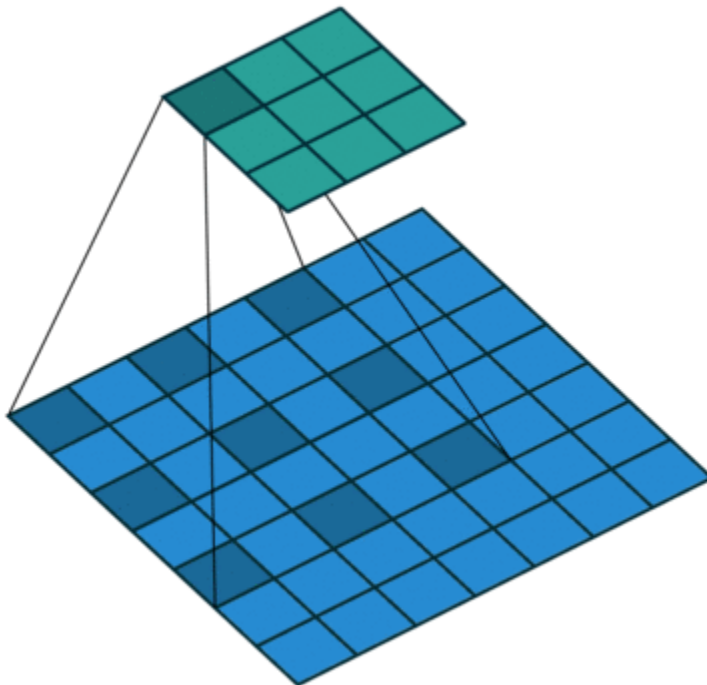
### Dilated Convolutions

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.



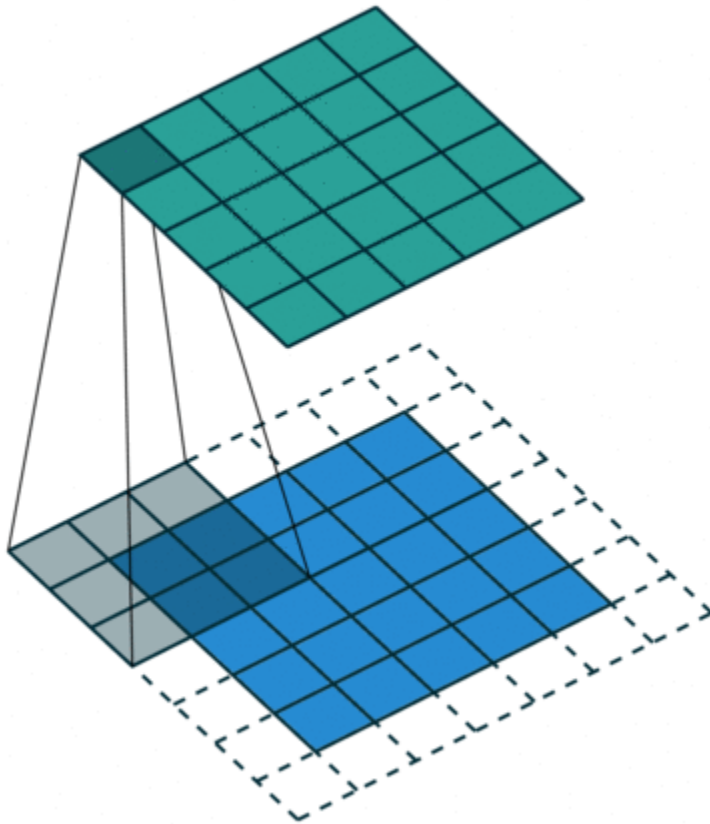
### Feature Maps

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.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `Convolution2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)

In `dlnetwork` objects, `Convolution2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSC" (spatial, spatial, channel)	"SSCB" (spatial, spatial, channel)
"SCT" (spatial, channel, time)	"SCBT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

### Introduced in R2016a

#### R2019a: 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## 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 Neural 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”](#)

# 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.

The dimensions that the layer convolves over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer convolves over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the spatial and time dimensions.

## 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 1-426, “Learning Rate and Regularization” on page 1-427, 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

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.



Before R2021a, use commas to separate each name and value, and enclose Name in 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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

### 3-D 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 layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the layer output.

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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.	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 4 & 0 \\ 0 & 0 & 1 & 5 & 9 & 0 \\ 0 & 0 & 2 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
'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

**NumChannels — Number of input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. NumChannels and the number of channels in the layer input data must match. For example, if the input is an RGB image, then NumChannels must be 3. If the input is the output of a convolutional layer with 16 filters, then NumChannels must be 16.

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 Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2 / (\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2 / \text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$ .
- '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  $\text{weights} = \text{func}(\text{sz})$ , where  $\text{sz}$  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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

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	''	3-D 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  $(\text{imageSize}(i) - \text{filterSize}(i) + \text{padding}(i)) / \text{stride}(i) + 1$ .

- 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
```

## Algorithms

### 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 1-415 on the `convolution2dLayer` reference page.

The dimensions that the layer convolves over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer convolves over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves over the spatial and time dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `Convolution3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)

In `dlnetwork` objects, `Convolution3DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2019a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## **See Also**

`convolution2dLayer` | `globalAveragePooling3dLayer` | `maxPooling3dLayer` | `image3dInputLayer` | `averagePooling3dLayer`

## **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”

## 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.

Mode	Description
'custom'	The location of the cropping window is based on the <code>Location</code> property. This value is automatically set when the <code>Location</code> 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 <code>Mode</code> 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 the name ' '.

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:  
  
    InputNames: {'image'}  
    OutputNames: {1x0 cell}  
    Layers: [2x1 nnet.cnn.layer.Layer]  
    Connections: [1x2 table]
```

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:  
  
    InputNames: {'image'}  
    OutputNames: {1x0 cell}  
    Layers: [2x1 nnet.cnn.layer.Layer]  
    Connections: [2x2 table]
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `Crop2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)

In `dlnetwork` objects, `Crop2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

Introduced in R2017b

### 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”

# 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 the name ' '.

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'	3-D Convolution	16 3x3x3 convolutions with stride [1 1 1] and padding 'same'

```

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.

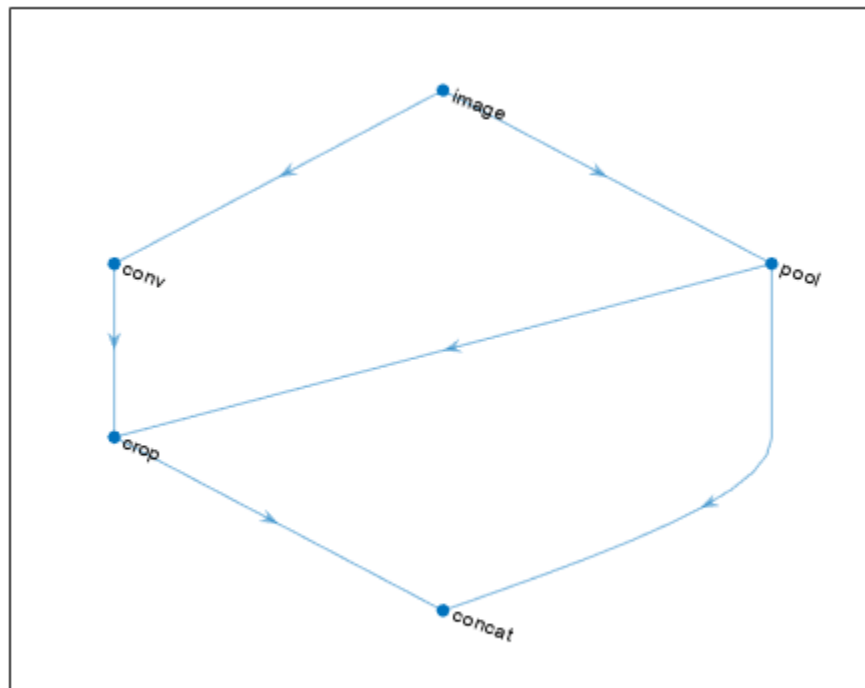
```
lgraph = connectLayers(lgraph,'pool','crop/ref');
```

Concatenate the crop layer output and the max pooling layer output.

```
lgraph = connectLayers(lgraph,'pool','concat/in2');
```

Display the layer graph.

```
plot(lgraph)
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `Crop3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, spatial, channel, batch)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `Crop3DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2019b

### 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”

# 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 less than or equal to 16

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 less than or equal to 16.

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 — $\alpha$ hyperparameter in normalization

0.0001 (default) | numeric scalar

$\alpha$  hyperparameter in the normalization (the multiplier term), specified as a numeric scalar.

Example: 0.0002



**Beta —  $\beta$  hyperparameter in normalization**

0.75 (default) | numeric scalar

$\beta$  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 the name ' '.

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 '' 2-D Convolution 20 5x5 convolutions with stride [1 1] and padding [
      3 '' ReLU ReLU
      4 '' Cross Channel Normalization cross channel normalization with 3 channels per elem
      5 '' Fully Connected 10 fully connected layer
      6 '' Softmax softmax
      7 '' Classification Output crossentropyex
```

## Algorithms

### Cross Channel Normalization Layer

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  $K$ ,  $\alpha$ , and  $\beta$  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`.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `CrossChannelNormalizationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSB" (spatial, spatial, spatial, batch)	"SSSB" (spatial, spatial, spatial, batch)

In `dlnetwork` objects, `CrossChannelNormalizationLayer` objects also support these input and output format combinations.

<b>Input Format</b>	<b>Output Format</b>
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSBT" (spatial, spatial, spatial, batch, time)	"SSSBT" (spatial, spatial, spatial, batch, time)

## Version History

Introduced in R2016a

## 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 Neural Network for Classification"

"Train Convolutional Neural Network for Regression"

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

# crosschannelnorm

Cross channel square-normalize using local responses

## Syntax

```
Y = crosschannelnorm(X,windowSize)
Y = crosschannelnorm(X,windowSize,'DataFormat',FMT)
Y = 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 `darray` data. If you want to apply cross-channel normalization within a `layerGraph` object or `Layer` array, use the following layer:

- `crossChannelNormalizationLayer`
- 

`Y = crosschannelnorm(X,windowSize)` normalizes each element of `X` with respect to local values in the same position in nearby channels. The normalized elements in `Y` are calculated from the elements in `X` using the following formula.

$$y = \frac{x}{\left(K + \frac{\alpha * ss}{windowSize}\right)^\beta}$$

where  $y$  is an element of `Y`,  $x$  is the corresponding element of `X`,  $ss$  is the sum of the squares of the elements in the channel region defined by `windowSize`, and  $\alpha$ ,  $\beta$ , and  $K$  are hyperparameters in the normalization.

`Y = crosschannelnorm(X,windowSize,'DataFormat',FMT)` also specifies the dimension format `FMT` when `X` is an unformatted `darray`, in addition to the input arguments the previous syntax. The output `Y` is an unformatted `darray` with the same dimension order as `X`.

`Y = 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  $\beta$  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);
X = dlarray(X, 'SSCB');
```

Compute the cross-channel normalization using a channel window size of three.

```
Y = crosschannelnorm(X,3);
```

Each value in each observation of  $X$  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);
d1X = dlarray(X);
```

Normalize the data using a channel-window size of 3, an  $\alpha$  of 1, a  $\beta$  of 1, and a  $K$  of  $1e-5$ . Specify a data format of 'SSC'.

```
d1Y = crosschannelnorm(d1X,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.

```
d1X = reshape(d1X,2,6)
```

```
d1X =
  2x6 dlarray
    1    1    1    1    1    1
    1    1    1    1    1    1
```

```
d1Y = reshape(d1Y,2,6)
```

```
d1Y =
  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);
dIX = dIarray(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,numGroups);
params.conv.bias = 0;
```

Apply the `modelFunction` to the data `dIX`.

```
dIY = modelFunction(dIX,params);
```

```
function dIY = modelFunction(dIX,params)
```

```
dIY = dlconv(dIX,params.conv.weights,params.conv.bias);
dIY = relu(dIY);
dIY = crosschannelnorm(dIY,5);
```

```
end
```

## Input Arguments

### **X — Input data**

`dlarray`

Input data, specified as a `dlarray` with or without data format. When X 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 X 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  $\text{floor}((\text{windowSize}-1)/2)$  channels and the following  $\text{floor}((\text{windowSize})/2)$  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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 these options:

- "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 `darray`.

Data Types: `char` | `string`

### **Alpha — Normalization constant ( $\alpha$ )**

`1e-4` (default) | numeric scalar

Normalization constant ( $\alpha$ ) 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 ( $\beta$ )**

`0.75` (default) | numeric scalar greater than or equal to `0.01`

Contrast constant ( $\beta$ ), 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**

### **Y — Normalized data**

`darray`

Normalized data, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, Y has the same dimension labels as X. If the input data is an unformatted `darray`, Y is an unformatted `darray` 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 “Cross Channel Normalization Layer” on page 1-448 on the `crossChannelNormalizationLayer` reference page.

## **Version History**

**Introduced in R2020a**

## **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 `X` is a `darray` 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**

`darray` | `dlgradient` | `dlfeval` | `avgpool` | `dlconv` | `maxpool`

## **Topics**

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Network Using Model Function”

“List of Functions with `darray` Support”

# crossentropy

Cross-entropy loss for classification tasks

## Syntax

```
loss = crossentropy(Y,targets)
loss = crossentropy(Y,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` 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(Y,targets)` returns the categorical cross-entropy loss between the formatted `darray` object `Y` containing the predictions and the target values `targets` for single-label classification tasks. The output `loss` is an unformatted `darray` scalar.

For unformatted input data, use the 'DataFormat' option.

`loss = crossentropy(Y,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 `Y` is not a formatted `darray`.

`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 =  
    1×1 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 =
    1×1 single darray
    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 = darray(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 =  
1×1 dLarray
```

```
1.1261
```

## Input Arguments

### Y — Predictions

dLarray | numeric array

Predictions, specified as a formatted dLarray, an unformatted dLarray, or a numeric array. When Y is not a formatted dLarray, you must specify the dimension format using the DataFormat option.

If Y 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 Y. For example, if Y 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 Y, or the same as DataFormat if Y is unformatted.

If `targets` is an unformatted `darray` or a numeric array, then the function applies the format of `Y` or the value of `DataFormat` to `targets`.

---

**Tip** Formatted `darray` 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 `Y` and `targets` are consistent, when `Y` is a formatted `darray`, also specify `targets` as a formatted `darray`.

---

## weights — Weights

`darray` | numeric array

Weights, specified as a `darray` 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 `Y`. Specify the 'C' (channel) dimension of the class weights by using a formatted `darray` 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 `Y`. Specify the 'B' (batch) dimension of the class weights by using a formatted `darray` 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 `Y`. In this case, if `weights` is not a formatted `darray` object, then the function uses the same format as `Y`. Alternatively, specify the weights format using 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `Y` 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 `Y`, averaged over the number of observations.
- 'independent' — Multi-label classification. Each observation in the predictions `Y` 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 `Y` 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**`darray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `darray` object, a logical array, or a numeric array with the same size as `Y`.

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.

If `Mask` is a formatted `darray` object, then its format must match that of `Y`. If `Mask` is not a formatted `darray` object, then the function uses the same format as `Y`.

If you specify the `DataFormat` option, then the function also uses the specified format for the mask.

The size of each dimension of `Mask` must match the size of the corresponding dimension in `Y`. The default value is a logical array of ones.

---

**Tip** Formatted `darray` 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, `darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

**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 `darray` object with the same size as `Y`.

**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 `Y`.
- `"all-elements"` — Normalize the loss by dividing it by the number of elements of `Y`.
- `"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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `Y` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `Y` are vectors, then default value of `WeightsFormat` is the same as the format of `Y`.

Data Types: `char` | `string`

## **Output Arguments**

### **loss — Cross-entropy loss**

`darray`

Cross-entropy loss, returned as an unformatted `darray`. The output `loss` is an unformatted `darray` with the same underlying data type as the input `Y`.

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.	$\text{loss} = -\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K T_{ni} \ln Y_{ni},$ <p>where <math>N</math> and <math>K</math> are the numbers of observations, and classes, respectively.</p>
Multi-label classification	Cross-entropy loss for independent classes. This is useful when observations can have multiple labels.	$\text{loss} = -\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K (T_{ni} \ln(Y_{ni}) + (1 - T_{ni}) \ln(1 - Y_{ni})),$ <p>where <math>N</math> and <math>K</math> are the numbers of observations and classes, respectively.</p>

Task	Description	Loss
Single-label classification with weighted classes	Cross-entropy loss with class weights. This is useful for datasets with imbalanced classes.	<p>loss =</p> $-\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K w_i T_{ni} \ln Y_{ni},$ <p>where <math>N</math> and <math>K</math> are the numbers of observations and classes, respectively, and <math>w_i</math> denotes the weight for class <math>i</math>.</p>
Sequence-to-sequence classification	Cross-entropy loss with masked time-steps. This is useful for ignoring loss values that correspond to padded data.	<p>loss =</p> $-\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^S m_{nt} \sum_{i=1}^K T_{nti} \ln Y_{nti},$ <p>where <math>N</math>, <math>S</math>, and <math>K</math> are the numbers of observations, time steps, and classes, <math>m_{nt}</math> denotes the mask value for time step <math>t</math> of observation <math>n</math>.</p>

## Version History

Introduced in R2019b

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `Y`
  - `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”](#)

## ctc

Connectionist temporal classification (CTC) loss for unaligned sequence classification

### Syntax

```
loss = ctc(Y,targets,YMask,targetsMask)
loss = ctc(Y,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

`loss = ctc(Y,targets,YMask,targetsMask)` returns the CTC loss between the formatted `darray` object `Y` containing the predictions and the target values `targets` using the prediction and target masks `YMask` and `targetsMask`, respectively. The function reduces the loss values by taking the mean across the batch dimension.

For unformatted input data, use the 'DataFormat' option.

`loss = ctc(Y,targets,YMask,targetsMask,'DataFormat',FMT)` also specifies the dimension format `FMT` when `Y` is not a formatted `darray`.

`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];
```

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);
```

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);
```

Pad the prediction sequences in the second dimension using the `padsequences` function and also return the corresponding mask.

```
[Y, YMask] = padsequences(Y, 2);
```

Convert the padded prediction sequences and mask to `darray` with format "CTB" (channel, time, batch). Because formatted `darray` objects automatically sort the dimensions, keep the dimensions of the targets and mask consistent by also converting them to a formatted `darray` objects with the same formats.

```
Y = darray(Y, "CTB");  
YMask = darray(YMask, "CTB");
```

The `ctc` function expects output from a softmax operation or layer. Apply the softmax operation to the predictions.

```
Y = softmax(Y);
```

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);
```

Similarly, convert the padded target sequences and mask to `darray` with format "TB" (time, batch).

```
targets = darray(targets, "TB");  
targetsMask = darray(targetsMask, "TB");
```

Compute the CTC loss between the predictions and the targets using the `ctc` function.

```
loss = ctc(Y, targets, YMask, targetsMask)
```

```
loss =  
  1x1 darray  
  35.5857
```

## Input Arguments

### Y — Predictions

`darray` | numeric array

Predictions, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. When Y is not a formatted `darray`, you must specify the dimension format using the `'DataFormat'` option.

The function computes the CTC loss assuming that Y is the output of a softmax operation or layer.

The predictions Y must have a 'B' (batch), 'C' (channel), and 'T' (time) dimension and can have different sequence lengths to the corresponding targets in `targets`.

If Y is a numeric array, then `targets`, `YMask`, or `targetsMask` must be a `darray`.

### targets — Target sequences

`darray` | numeric array

Target sequences, specified as a formatted or unformatted `darray` 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 `darray` 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 Y and must not include the blank index.

If `targets` is a formatted `darray`, then its format must be the same as the format of Y, or the same as `DataFormat` if Y is unformatted.

If `targets` is an unformatted `darray` or a numeric array, then the function applies the format of Y or the value of `DataFormat` to `targets`.

---

**Tip** Formatted `darray` 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 Y and `targets` are consistent, when Y is a formatted `darray`, also specify `targets` as a formatted `darray`.

---

### YMask — Mask indicating which prediction elements to include for loss computation

`darray` | logical array | numeric array

Mask indicating which prediction elements to include for loss computation, specified as a `darray` object, a logical array, or a numeric array with the same size as Y.

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 `darray` 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,

`darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

### **targetsMask — Mask indicating which target elements to include for loss computation**

`darray` | logical array | numeric array

Mask indicating which target elements to include for loss computation, specified as a `darray` object, a logical array, or a numeric array with the same size as `targets`.

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 `darray` 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, `darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

### **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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 be between 1 and the number of channels of `Y` 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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

## Output Arguments

### Loss — CTC loss

`darray`

CTC loss, returned as an unformatted `darray` scalar with the same underlying data type as the input `Y`.

## Version History

Introduced in R2021a

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `Y`
  - `targets`
  - `YMask`
  - `targetsMask`

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

## See Also

`darray` | `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”

# 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 1-1718. 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 1-983.
  - 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 1-1097.
  - For a Caffe model, use `importCaffeNetwork`. For an example, see “Import Caffe Network” on page 1-960.
  - For an ONNX model, use `importONNXNetwork`. For an example, see “Import ONNX Network as DAGNetwork” on page 1-1061.
- 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**

<code>activations</code>	Compute deep learning network layer activations
<code>classify</code>	Classify data using trained deep learning neural network
<code>predict</code>	Predict responses using trained deep learning neural network
<code>plot</code>	Plot neural network architecture

## **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];

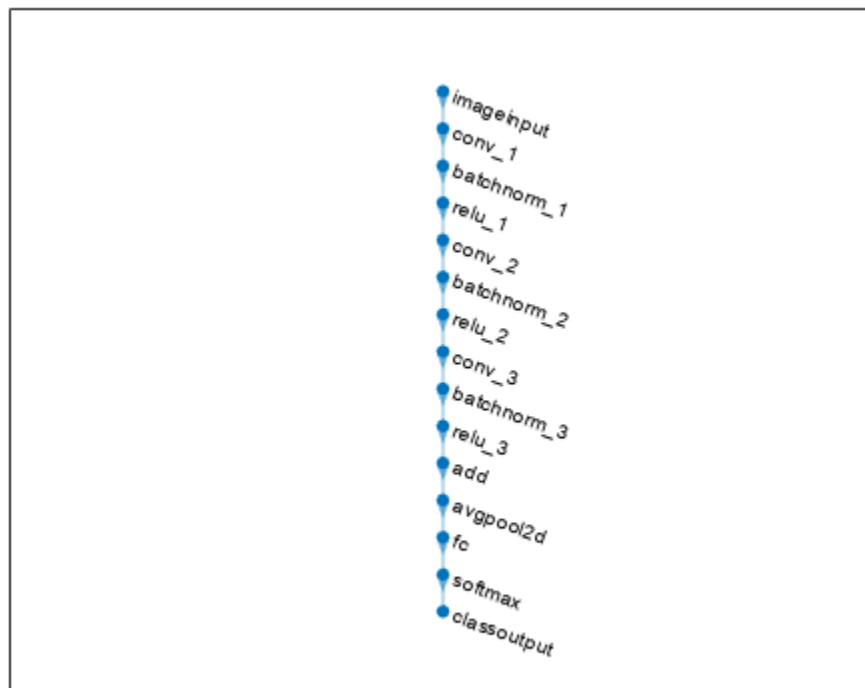
```

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)

```

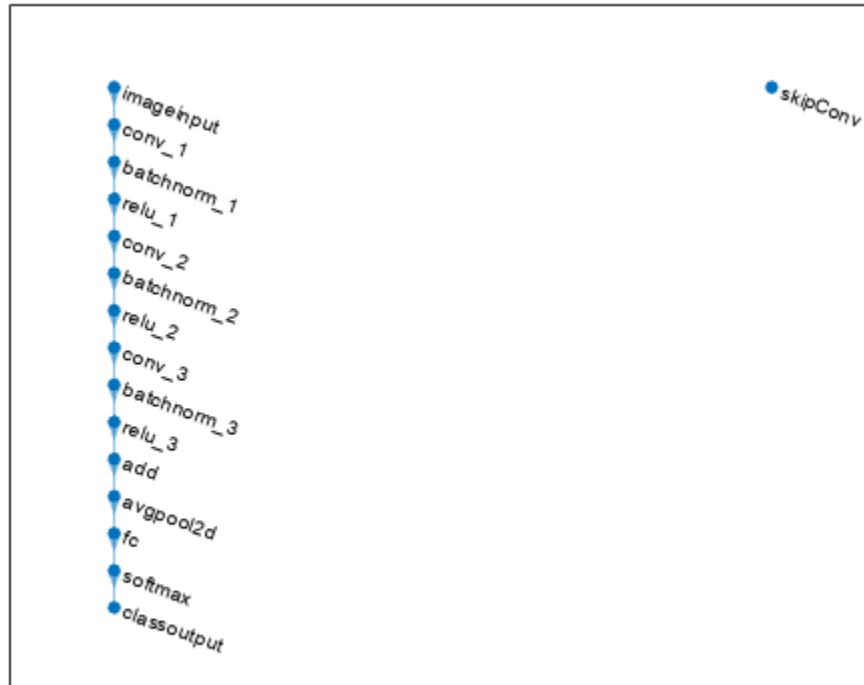


Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the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

```

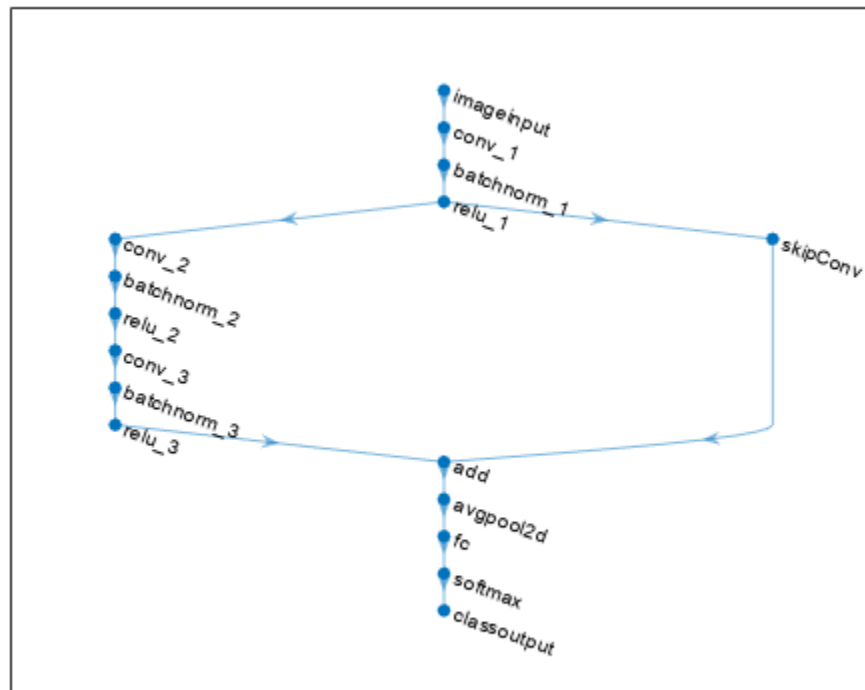


Create the shortcut connection from the 'relu\_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The third ReLU layer is already connected to the 'in1' input. Connect the 'relu\_1' 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);

```

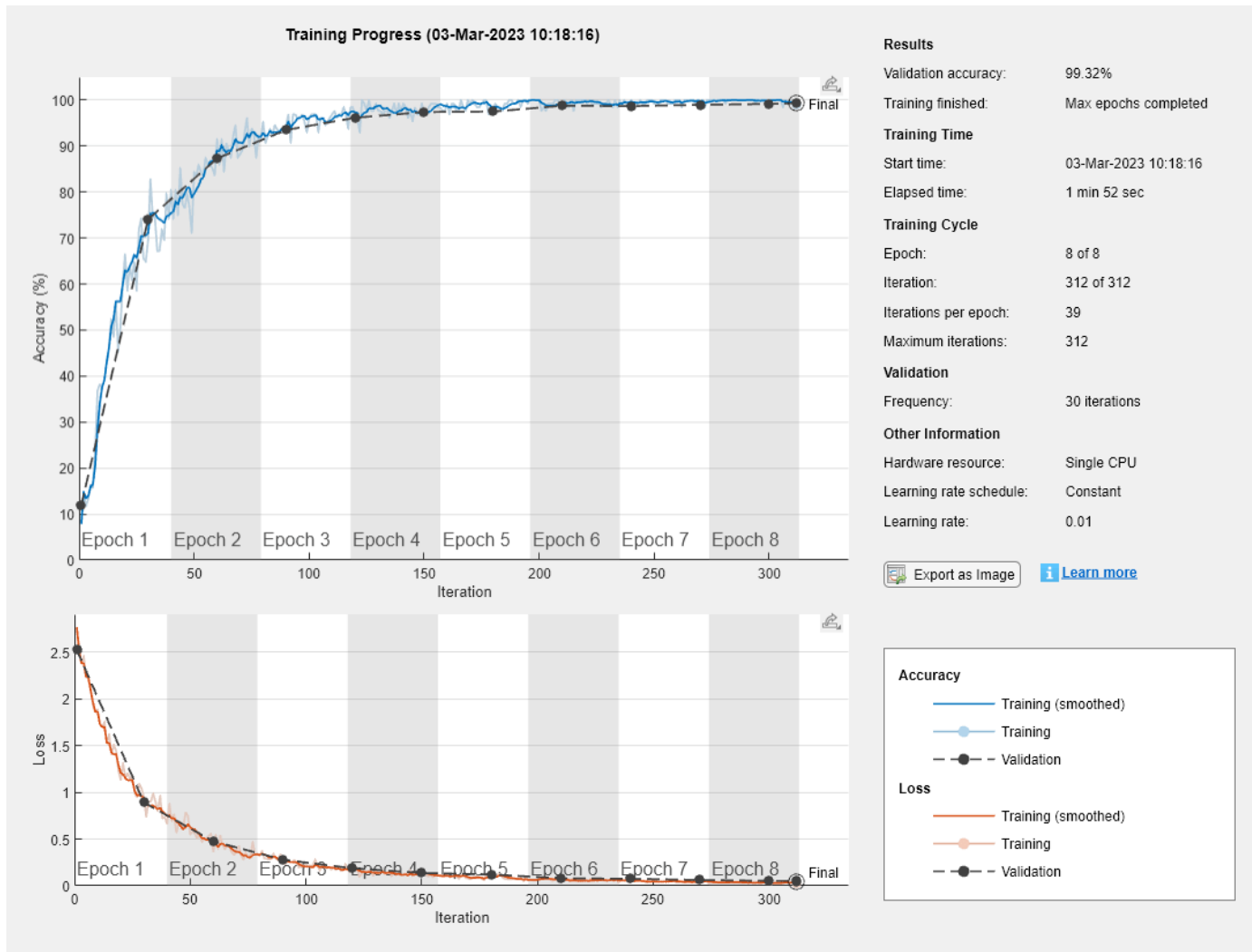


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[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 = 0.9932
```



## Version History

Introduced in R2017b

### 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”

## 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 =

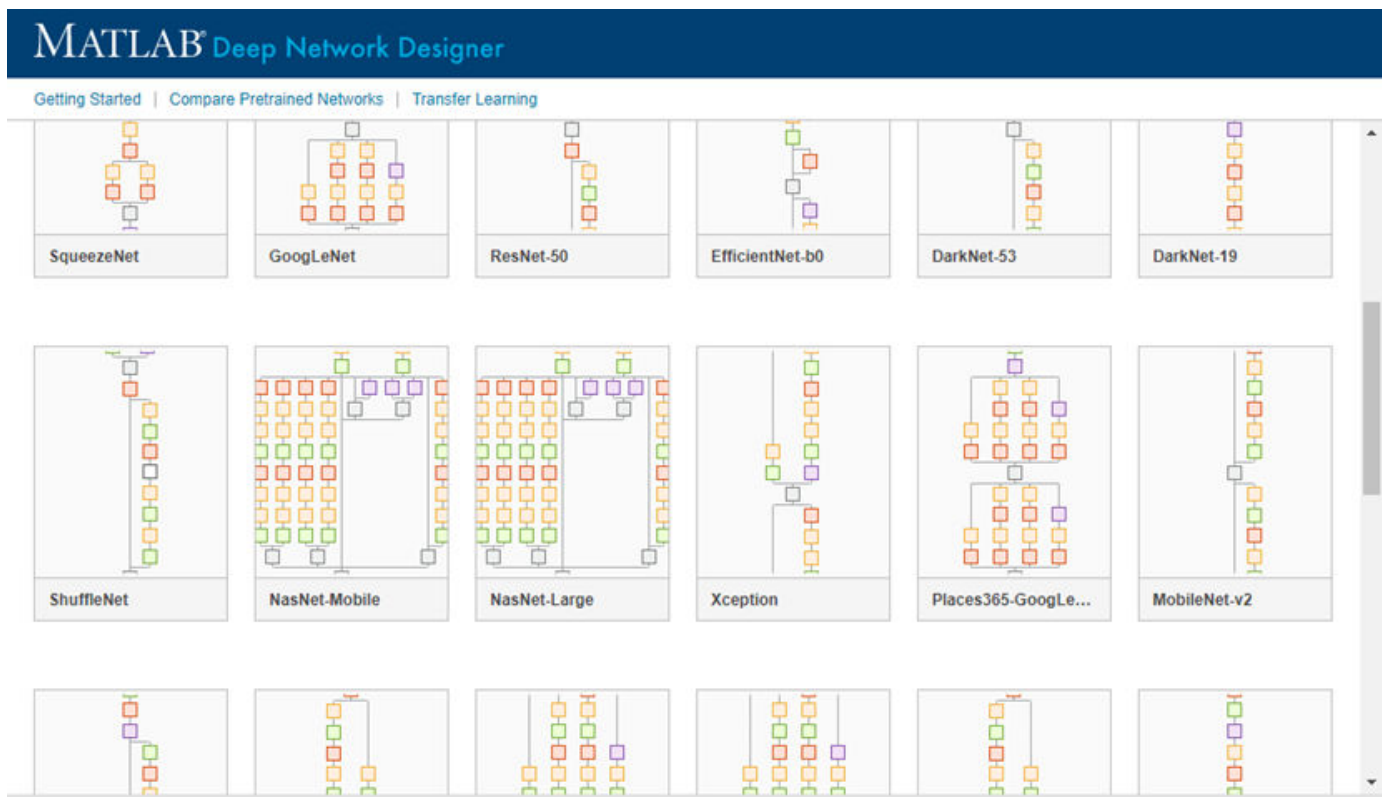
SeriesNetwork with properties:

```
Layers: [64x1 nnet.cnn.layer.Layer]
InputNames: {'input'}
OutputNames: {'output'}
```

Visualize the network using Deep Network Designer.

`deepNetworkDesigner(darknet19)`

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

**Introduced in R2020a**

## 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”

## 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 =
```

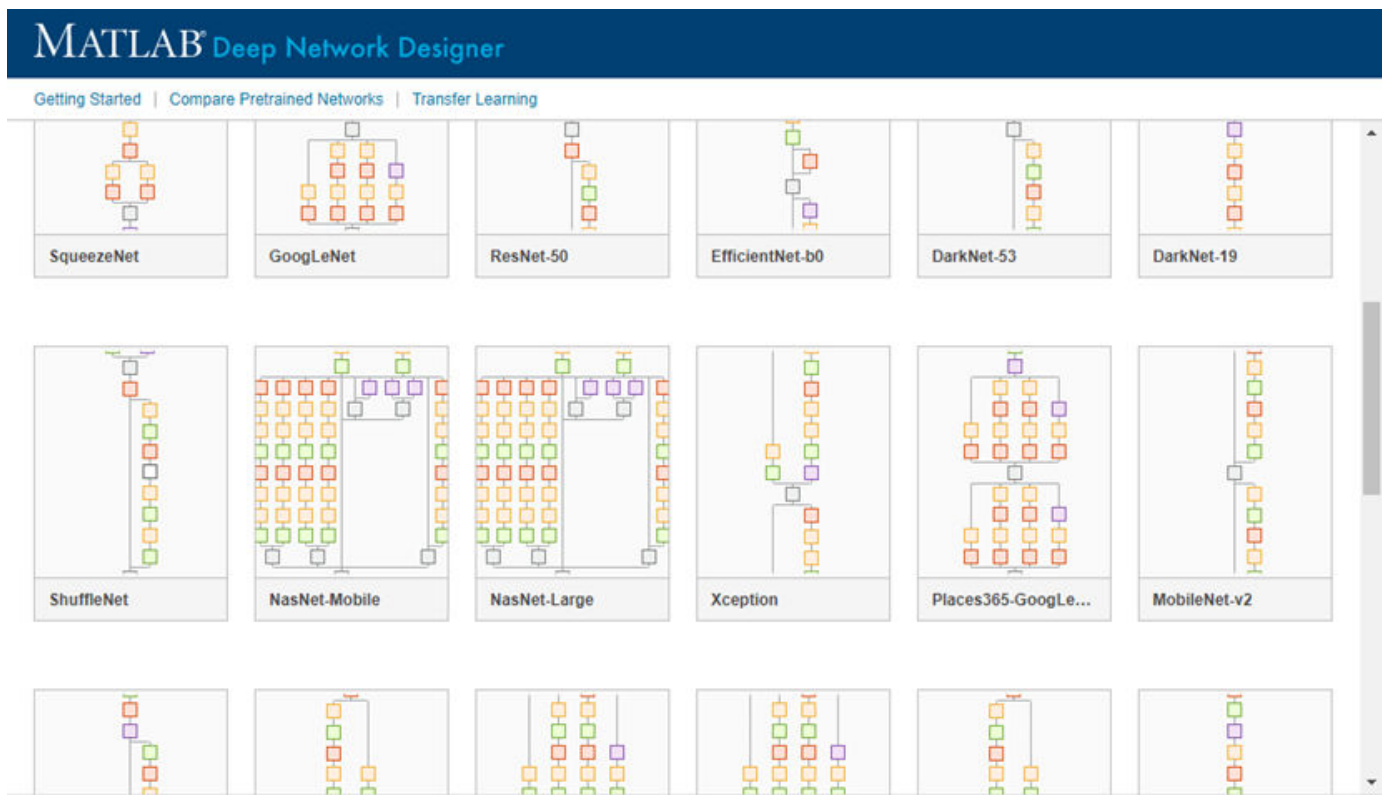
```
DAGNetwork with properties:
```

```
    Layers: [184x1 nnet.cnn.layer.Layer]
 Connections: [206x2 table]
  InputNames: {'input'}
 OutputNames: {'output'}
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(darknet53)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

**Introduced in R2020a**

## 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”

## 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.

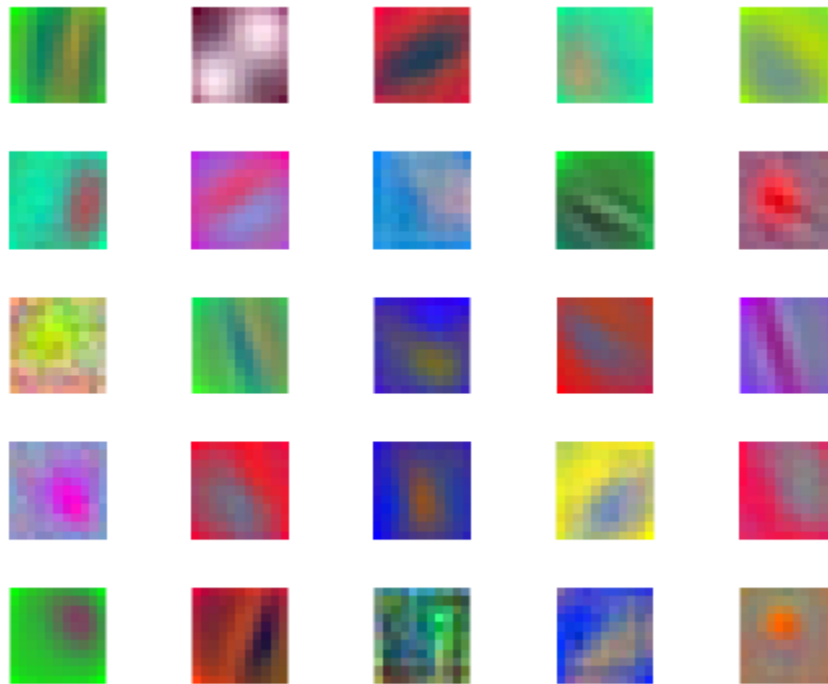
```
net = alexnet;
```

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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example:

```
deepDreamImage(net, layer, channels, 'NumIterations', 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 &gt; 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, $I(:, :, :, \text{channel})$ , 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 one of the following:

- "auto" — Use a GPU if one is available; otherwise, use the CPU.
- "gpu" — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- "cpu" — Use the 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 neural 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 neural networks using both CPUs and GPUs.

## Version History

**Introduced in R2017a**

## 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”

# 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 =
```

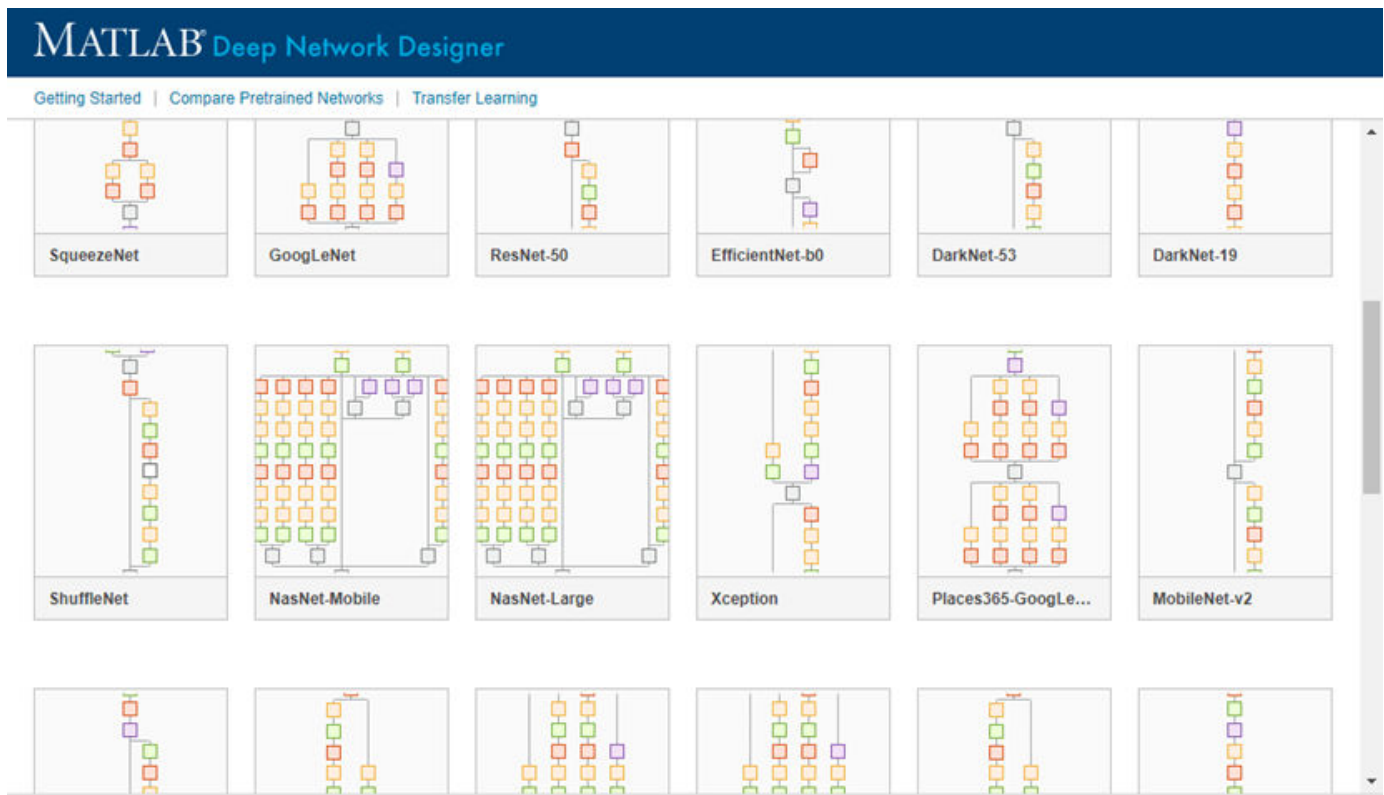
```
DAGNetwork with properties:
```

```
    Layers: [709x1 nnet.cnn.layer.Layer]  
    Connections: [806x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(densenet201)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2018a

## 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”

# depthConcatenationLayer

Depth concatenation layer

## Description

A depth concatenation layer takes inputs that have the same height and width and concatenates them along 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 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 the name ' '.

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'}
```

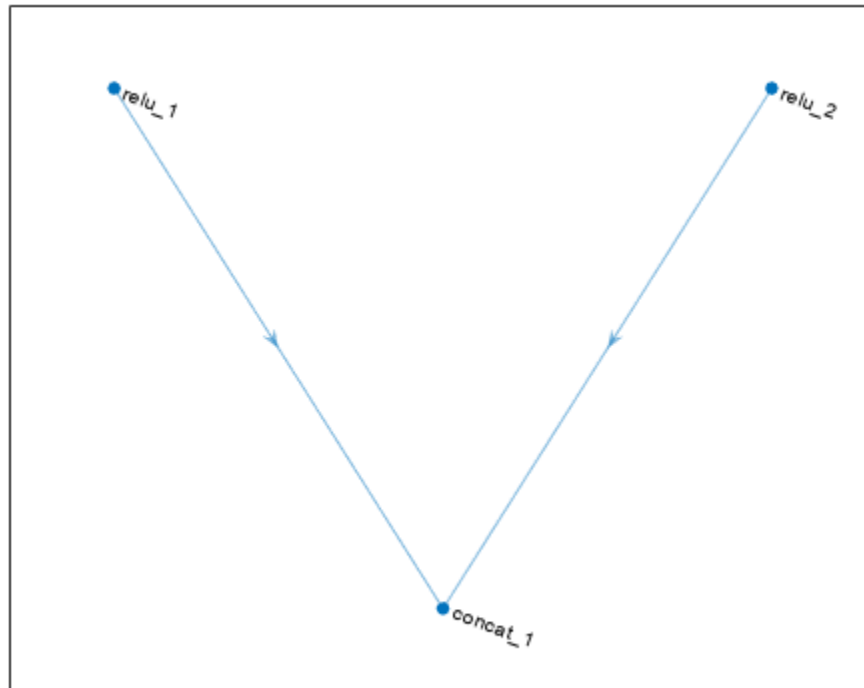
Create two ReLU layers and connect them to the depth concatenation layer. The depth concatenation layer concatenates the outputs from the ReLU layers.

```
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)
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `DepthConcatenationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

The output format matches the input format.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `DepthConcatenationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2017b

## 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 Neural 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"

# dims

Dimension labels of `darray`

## Syntax

```
d = dims(X)
```

## Description

`d = dims(X)` returns the data format of `X` as a character array. The data format provides the dimension labels for each dimension in `X`.

## Examples

### Obtain Dimension Labels

Obtain the dimension labels of a formatted `darray`.

```
d1X = darray(randn(3,4), 'TS');  
d = dims(d1X)
```

```
d =  
'ST'
```

Obtain the labels of an unformatted `darray`.

```
y = stripdims(d1X);  
d = dims(y)
```

```
d =
```

```
0x0 empty char array
```

## Input Arguments

### X — Input data

`darray` object

Input data, specified as a formatted or unformatted `darray` object.

## Output Arguments

### d — Dimension labels

character vector

Dimension labels, returned as a character vector. If the input `X` is unformatted, `d` is empty.

## **Version History**

**Introduced in R2019b**

### **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`



# 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 `darray` objects, structures of `darray` 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");  
net = s.net;  
classNames = s.classNames;
```

Accelerate the model loss function `modelLoss` listed at the end of the example.

```
fun = @modelLoss;  
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: @modelLoss  
    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");  
X = dlarray(X,"SSCB");  
  
T = categorical(classNames(randi(10,[128 1])));  
T = onehotencode(T,2)';  
T = dlarray(T,"CB");  
  
[loss,gradients,state] = dlfeval(accfun,net,X,T);
```

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

```
accfun.Occupancy
```

```
ans = 2
```

### Model Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding target labels `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)
```

```
[Y,state] = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);

end
```

### Clear Cache of Accelerated Function

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

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

Accelerate the model loss function `modelLoss` listed at the end of the example.

```
fun = @modelLoss;
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: @modelLoss
    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");
X = dlarray(X,"SSCB");

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

[loss,gradients,state] = dlfeval(accfun,net,X,T);
```

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 Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding target labels `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)
```

```
[Y,state] = forward(net,X);  
loss = crossentropy(Y,T);  
gradients = dlgradient(loss,net.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
```

Clear 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.

```
d1X = dlarray(ones(3,3));
d1Y = accfun(d1X)
```

```
d1Y =
  3×3 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.

```
d1Y = accfun(d1X)
```

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

```
d1Y =
  3×3 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 dIarray 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 dIarray object (a dIarray object that depends on an input dIarray 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 dIarray object.

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

Clear 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.

```
dIY = accfun2(dIX)
```

```
dIY =
  3x3 dIarray
    1.7922    1.0357    1.6787
    1.9595    1.8491    1.7577
    1.6557    1.9340    1.7431
```

```
dIY = accfun2(dIX)
```

```
dIY =
  3x3 dIarray
    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: `@modelLoss`

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)

Because the caching process requires extra computation, acceleration can lead to longer running code in some cases. This scenario can happen when the software spends time creating new caches that do not get reused often. For example, when you pass multiple mini-batches of different sequence lengths to the function, the software triggers a new trace for each unique sequence length.

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"`. 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` function with the `GradientMode` option set to `"direct"`.

## **Version History**

**Introduced in R2021a**

### **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”



# dlarray

Deep learning array for customization

## 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 neural 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 neural 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 1-514. 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 string scalar or a character vector. Each character in the string must be one of the following dimension labels:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "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 the input data. If `fmt` is not in the listed order ('S' followed by 'C' and so on), then `darray` 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 the input data. If you specify more than that number of dimension labels, `darray` 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	<i>h</i> -by- <i>w</i> -by- <i>c</i> -by- <i>n</i> numeric array, where <i>h</i> , <i>w</i> , <i>c</i> and <i>n</i> are the height, width, number of channels of the images, and number of observations, respectively.	"SSCB"
3-D images	<i>h</i> -by- <i>w</i> -by- <i>d</i> -by- <i>c</i> -by- <i>n</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , <i>c</i> and <i>n</i> are the height, width, depth, number of channels of the images, and number of image observations, respectively.	"SSSCB"
Vector sequences	<i>c</i> -by- <i>s</i> -by- <i>n</i> matrix, where <i>c</i> is the number of features of the sequence, <i>s</i> is the sequence length, and <i>n</i> 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"

**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

<code>avgpool</code>	Pool data to average values over spatial dimensions
<code>batchnorm</code>	Normalize data across all observations for each channel independently
<code>crossentropy</code>	Cross-entropy loss for classification tasks
<code>dims</code>	Dimension labels of <code>dlarray</code>
<code>dlconv</code>	Deep learning convolution
<code>dlgradient</code>	Compute gradients for custom training loops using automatic differentiation
<code>dltransconv</code>	Deep learning transposed convolution
<code>extractdata</code>	Extract data from <code>dlarray</code>
<code>finddim</code>	Find dimensions with specified label
<code>fullyconnect</code>	Sum all weighted input data and apply a bias
<code>gru</code>	Gated recurrent unit
<code>leakyrelu</code>	Apply leaky rectified linear unit activation
<code>lstm</code>	Long short-term memory
<code>maxpool</code>	Pool data to maximum value
<code>maxunpool</code>	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”.

## Version History

### Introduced in R2019b

#### R2022b: Code generation behavior change for `dlarray` inputs and outputs

*Behavior changed in R2022b*

In R2022b, the generated code creates structures for the `dlarray` inputs and outputs of entry-point functions. `Data` is a public field that you can directly access it.

In previous releases, the generated code uses `class` to represent the `dlarray` inputs and outputs of entry-point functions. In these releases, you use the initializing function `init` to access the `Data` field.

## 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)”

“Train Network Using Custom Training Loop”

“Define Custom Training Loops, Loss Functions, and Networks”

“Automatic Differentiation Background”

“Use Automatic Differentiation In Deep Learning Toolbox”

“List of Functions with `dlarray` Support”



# dlconv

Deep learning convolution

## Syntax

```
Y = dlconv(X,weights,bias)
Y = dlconv(X,weights,bias,DataFormat=FMT)
Y = 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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` 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`
- 

`Y = dlconv(X,weights,bias)` applies the deep learning convolution operation to the formatted `darray` object `X`. The function uses sliding convolutional filters defined by `weights` and adds the constant `bias`. The output `Y` is a formatted `darray` object with the same format as `X`.

The function, by default, convolves over up to three dimensions of `X` labeled "S" (spatial). To convolve over dimensions labeled "T" (time), specify `weights` with a "T" dimension using a formatted `darray` object or by using the `WeightsFormat` option.

For unformatted input data, use the `DataFormat` option.

`Y = dlconv(X,weights,bias,DataFormat=FMT)` applies the deep learning convolution operation to the unformatted `darray` object `X` with format specified by `FMT`. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. For example, `DataFormat="SSCB"` specifies data for 2-D convolution with format "SSCB" (spatial, spatial, channel, batch).

`Y = 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);  
X = dlarray(X,"SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
```

```
    28    28     3   128
```

```
dims(X)
```

```
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.

```
Y = dlconv(X,weights,bias);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
```

```
    26    26    64   128
```

```
dims(Y)
```

```
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);
X = darray(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.

```
Y = dlconv(X,weights,bias);
size(Y)
```

```
ans = 1×4
```

```
    93    93    12    10
```

```
dims(Y)
```

```
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 `darray`.

```
height = 64;
width = 64;
numChannels = 10;
```

```
X = rand(height,width,numChannels);
X = darray(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 = numChannels;
```

```
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.

```
Y = dlconv(X,weights,bias,DataFormat="SSC");
size(Y)
```

```
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 `darray` 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);
X = darray(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.

```
Y = dlconv(X,weights,bias,WeightsFormat="TCU");
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×3
```

```
    64    128    510
```

```
dims(Y)
```

```
ans =  
'CBT'
```

## Input Arguments

### X — Input data

`dlarray` | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

If `X` is an unformatted `dlarray` or a numeric array, then you must specify the format using the `DataFormat` option. If `X` 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 `X` 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, numFiltersPerGroup is the number of filters per group, and numGroups is the number of groups.  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)	"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	and <code>filterSize(2)</code> are the height and width of the 2-D filters, respectively, <code>numChannels</code> is the number of channels of the input data, and <code>numFilters</code> is the number of filters.	
2-D grouped convolution	First "S" (spatial)	Filter height	<code>filterSize(1)-by-filterSize(2)-by-numChannelsPerGroup-by-numFiltersPerGroup-by-numGroups</code> array, where <code>filterSize(1)</code> and <code>filterSize(2)</code> are the height and width of the 2-D filters, respectively, <code>numChannelsPerGroup</code> is the number of channels per group of the input data, <code>numFiltersPerGroup</code> is the number of filters per group, and <code>numGroups</code> is the number of groups.  <code>numChannelsPerGroup</code> must equal the number of the channels of the input data divided by <code>numGroups</code> .	"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	<code>filterSize(1)-by-</code>	"SSSCU" (spatial, spatial, spatial,

Task	Required Dimensions	Size	Example	
			Weights	Format
	Second "S" (spatial)	Filter width	<code>filterSize(2)-by-filterSize(3)-by-numChannels-by-numFilters</code> array, where <code>filterSize(1)</code> , <code>filterSize(2)</code> , and <code>filterSize(3)</code> are the height, width, and depth of the 3-D filters, respectively, <code>numChannels</code> is the number of channels of the input data, and <code>numFilters</code> is the number of filters.	channel, unspecified)
	Third "S" (spatial) or "T" (time)	Filter depth		
	"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 X labeled "S" (spatial). To convolve over dimensions labeled "T" (time), specify `weights` with a "T" dimension using a formatted `darray` object or by using the `WeightsFormat` option.

---

### **bias — Bias constant**

`darray` | numeric vector | numeric scalar

Bias constant, specified as a formatted `darray`, an unformatted `darray`, 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 `darray`, then the nonsingleton dimension must be a channel dimension with label 'C' (channel).

### **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.



Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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 these options:

- "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.

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 X 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.

---

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 a numeric scalar or numeric vector.

To use the same step size for all convolution dimensions, specify the stride as a scalar. To specify a different value for each convolution dimension, specify the stride as a vector with elements ordered corresponding to the dimensions labels in the data format.

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 specified as a numeric scalar or numeric vector.

To use the dilation factor all convolution dimensions, specify the dilation factor as a scalar. To specify a different value for each convolution dimension, specify the dilation factor as a vector with elements ordered corresponding to the dimensions labels in the data format.

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  $\text{filterSize} + (\text{filterSize}-1) * (\text{dilationFactor}-1)$ .

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 one of the following:

- "same" — Apply padding such that the output dimension sizes are  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $(\text{FilterSize} - 1) * \text{DilationFactor}$ . 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 *i*th "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 *i*th "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.

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.	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 5 & 9 & 0 & 0 \\ 0 & 0 & 2 & 6 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
'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

### Y — Convolved feature map

darray

Convolved feature map, returned as a darray with the same underlying data type as X.

If the input data X is a formatted darray, then Y has the same format as X. If the input data is not a formatted darray, then Y is an unformatted darray with the same dimension order as the input data.

The size of the "C" (channel) dimension of Y 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 1-415 on the `convolution2dLayer` reference page.

## Version History

Introduced in R2019b

## 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.
  - `X`
  - `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"

## 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 neural 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 neural 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,  $x_1$  and  $x_2$ .

```
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;
```

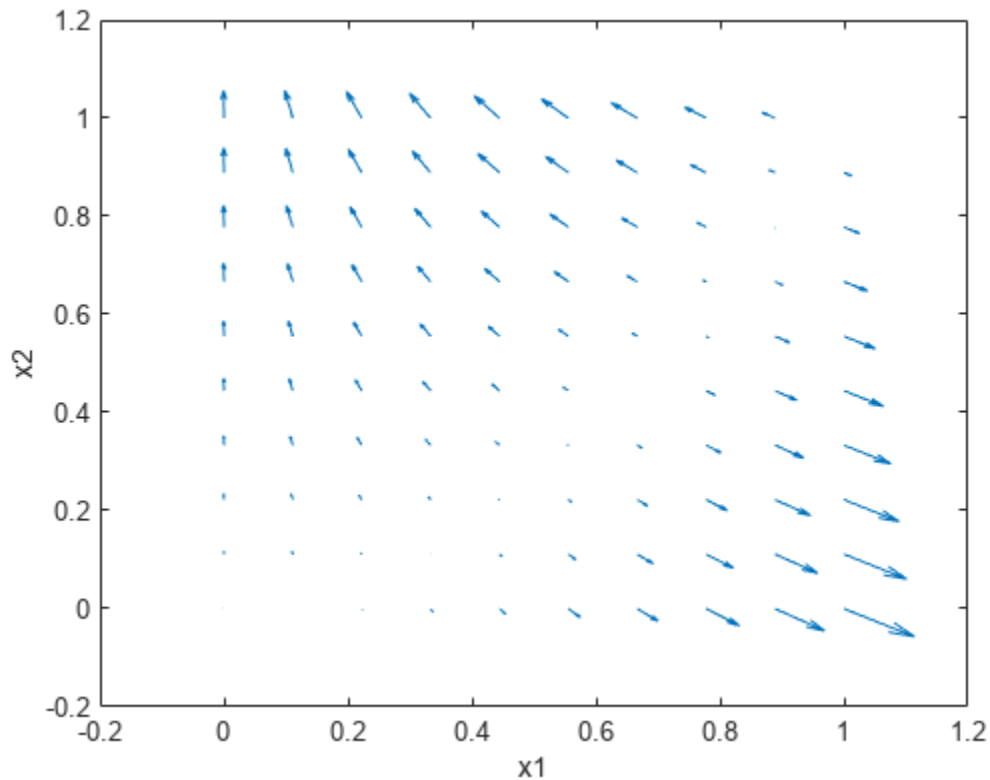
Evaluate the function in a loop. Plot the result using `quiver`.

```
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. ';
```

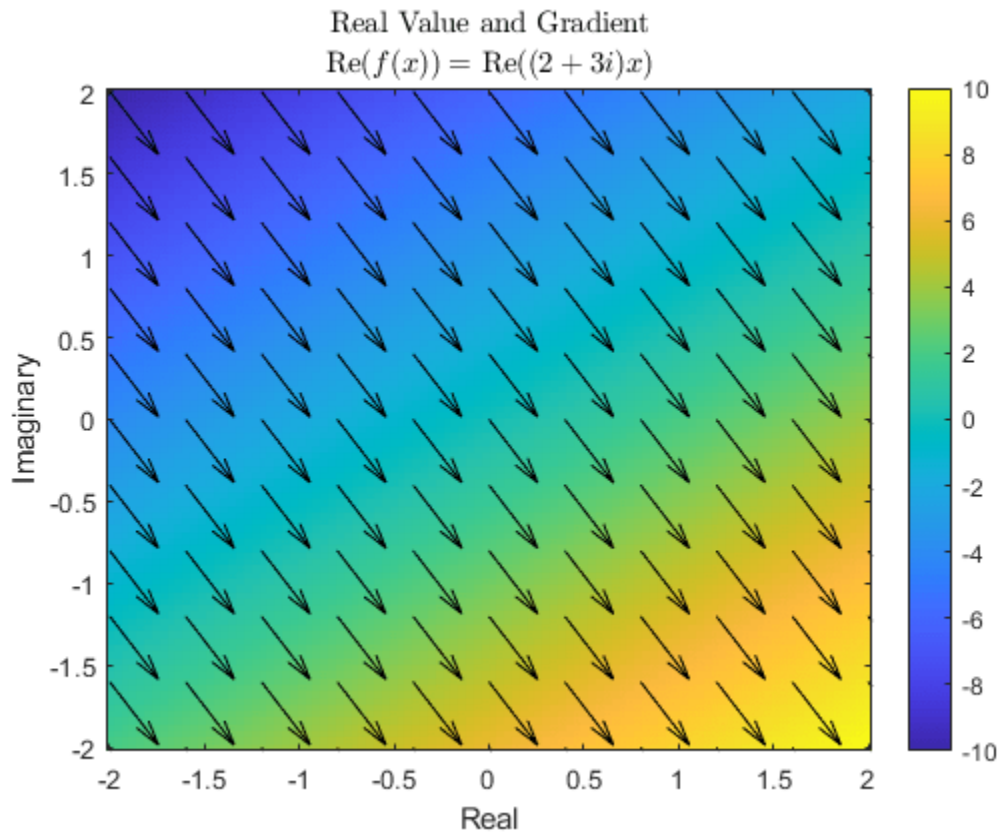
Extract the gradient values at these sample points.

```
[~,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 differentiation.

```
grad(1,1)

ans =
    1x1 dlarray

    2.0000 - 3.0000i
```

By inspection, the complex derivative of the function has the value

$$\frac{df(x)}{dx} = 2 + 3i$$

However, the function  $\text{Re}(f(x))$  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  $\text{Re}(f(x))$ :  $\mathbf{C} \rightarrow \mathbf{R}$  as a function  $\text{Re}(f(x_R + ix_I))$ :  $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ .

```
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. Quantized `dlnetwork` objects are not supported.

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”.

## Algorithms

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Version History

Introduced in R2019b

## 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  $x_1, \dots, x_n$  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)”

## 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 neural 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 neural 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 1-538 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 1-545.

### 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
```

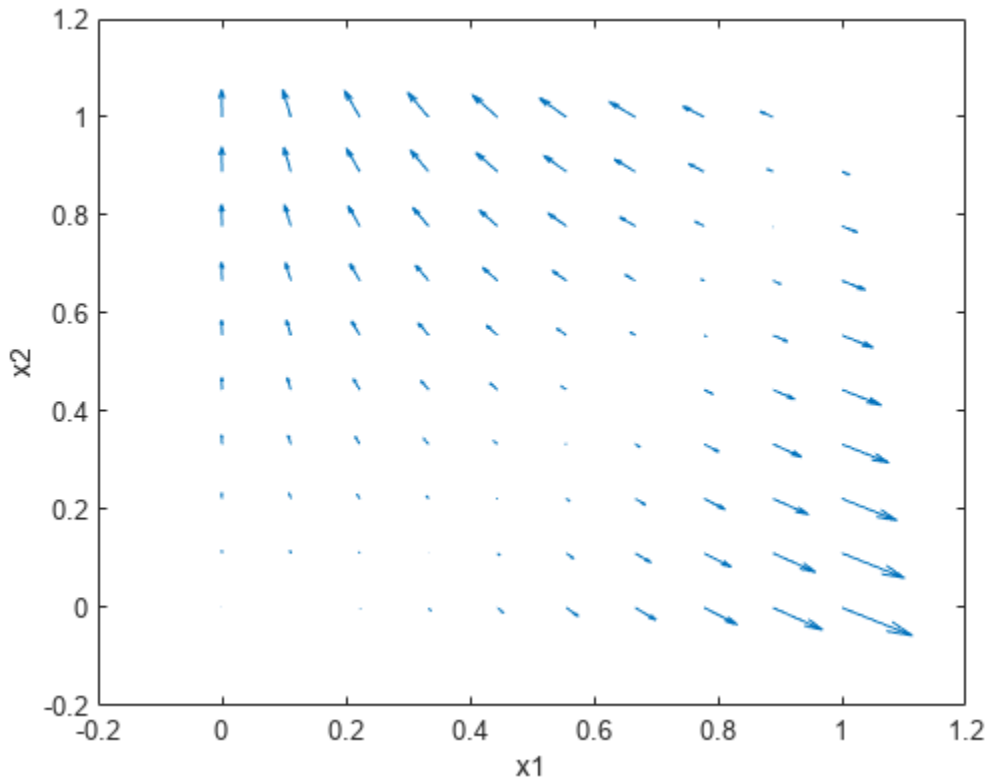
```
  200
```

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 = darray(X1(:));
X2 = darray(X2(:));
Y = darray(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.

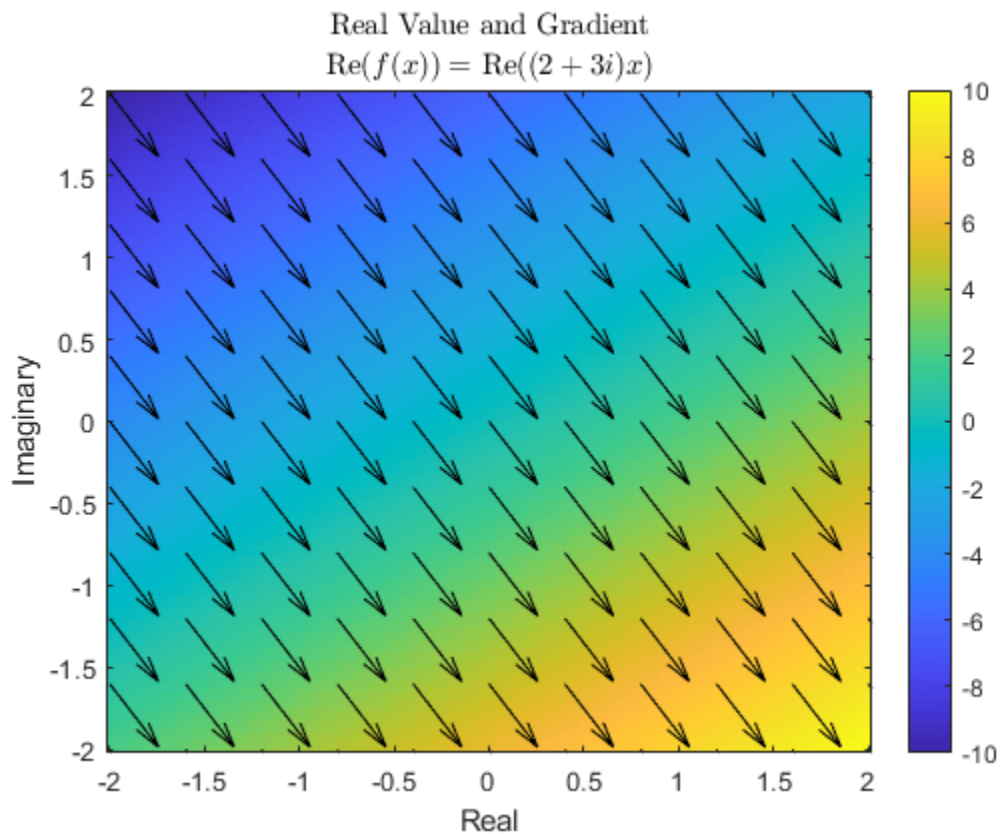
```
gradientRes = linspace(-2,2,11);
xGrad = gradientRes + 1i*gradientRes.';
```

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 differentiation.

```
grad(1,1)
```

```
ans =  
1x1 dlarray  
2.0000 - 3.0000i
```

By inspection, the complex derivative of the function has the value

$$\frac{df(x)}{dx} = 2 + 3i$$

However, the function  $\text{Re}(f(x))$  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  $\text{Re}(f(x))$ :  $\mathbf{C} \rightarrow \mathbf{R}$  as a function  $\text{Re}(f(x_R + ix_I))$ :  $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ .

```
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 1-545) 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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 `darray` discards the derivative trace immediately after computing a derivative. When this argument is `true`, a `darray` retains the derivative trace until the end of the `dfeval` function call that evaluates the `dgradient`. The `true` setting is useful only when the `dfeval` call contains more than one `dgradient` call. The `true` setting causes the software to use more memory, but can save time when multiple `dgradient` 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 = dgradient(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 `dgradient` 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 `dgradient` 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.

## Version History

Introduced in R2019b

### 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)”

# dlmtimes

(Not recommended) Batch matrix multiplication for deep learning

---

**Note** `dlmtimes` is not recommended. Use `pagemtimes` instead. For more information, see “Compatibility Considerations”

---

## Syntax

```
d1C = dlmtimes(d1A,d1B)
```

## Description

`d1C = dlmtimes(d1A,d1B)` computes matrix multiplication for each page of `d1A` and `d1B`. For 3-D inputs `d1A` and `d1B`, `d1C` is calculated as

$$d1C(:,:,i) = d1A(:,:,i) * d1B(:,:,i)$$

Similarly, for n-dimensional inputs `d1A` and `d1B`, `d1C` is calculated as

$$d1C(:,:,i1,\dots,in) = d1A(:,:,i1,\dots,in) * d1B(:,:,i1,\dots,in)$$

If one of `d1A` or `d1B` 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);
```

```
d1A = d1array(A);
d1B = d1array(B);
```

Calculate the batch matrix multiplication of `d1A` and `d1B`.

```
d1C = dlmtimes(d1A,d1B);
size(d1C)
```

```
ans = 1x4
      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);  
dIA = dlarray(A);
```

Create a random matrix of size 20-by-15.

```
B = rand(20,15);
```

Multiply `dIA` and `B` using `dlmtimes`.

```
dIC = dlmtimes(dIA,B);  
size(dIC)
```

```
ans = 1×4  
     15     15     3    128
```

## Input Arguments

### `dIA`, `dIB` — Operands

scalars | vectors | matrices | arrays

Operands, specified as scalars, vectors, matrices, or N-D arrays. At least one of `dIA` or `dIB` must be a `dlarray`. The inputs `dIA` or `dIB` must not be formatted unless one of `dIA` or `dIB` is an unformatted scalar.

The number of columns of `dIA` must match the number of rows of `dIB`. If one of `dIA` or `dIB` is a two-dimensional matrix, this matrix multiplies each page of the other input. Otherwise, the size of `dIA` and `dIB` for each dimension greater than two must match.

## Output Arguments

### `dIC` — Product

scalar | vector | matrix | array

Product, returned as a scalar, vector, matrix, or an N-D array.

Array `dIC` has the same number of rows as input `dIA` and the same number of columns as input `dIB`, unless one of `dIA` or `dIB` is a scalar. The size of the other dimensions of `dIC` match the size of the dimensions greater than two of both `dIA` and `dIB`. If `dIA` or `dIB` is a matrix, the size of the other dimensions matches the size of the other (non-matrix) input. If one of `dIA` or `dIB` is a scalar, `dIC` has the same size as the non-scalar input.

## Version History

Introduced in R2020a

### **R2020b: `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”](#)

# 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 neural 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 neural 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

```
net = dlnetwork(layers)
net = dlnetwork(layers,X1,...,Xn)
net = dlnetwork(layers,'Initialize',tf)
net = dlnetwork( ____, 'OutputNames',names)
net = dlnetwork(prunableNet)
```

### Description

`net = 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 `net` are initialized for training with initial values based on the input size defined by the network input layer.

`net = dlnetwork(layers,X1,...,Xn)` creates an initialized `dlnetwork` object using network data layout objects or example inputs `X1, ..., Xn`. The learnable parameters and state values of `net` are initialized with initial values based on the size and format defined by `X1, ..., Xn`. Use this syntax to create an initialized `dlnetwork` with inputs that are not connected to an input layer.

`net = 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.

`net = 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.

`net = dlnetwork(prunableNet)` removes filters selected for pruning from the convolution layers of `prunableNet` and returns a compressed `dlnetwork` object that has fewer learnable parameters and is smaller in size.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

## 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 1-565.

### X1, . . . , Xn — Example network inputs or data layouts

formatted `dlnetwork` object | formatted `networkDataLayout` object

Example network inputs or data layouts, specified as formatted `dlnetwork` objects or formatted `networkDataLayout` objects. The software propagates `X1, . . . , Xn` through the network to determine the appropriate sizes and formats of the learnable and state parameters of the `dlnetwork`.

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.

---

**Note** Automatic initialization uses only the size and format information of the input data. For initialization that depends on the values on the input data, you must initialize the learnable parameters manually.

---

### tf — Flag to return initialized dlnetwork

1 (true) (default) | 0 (false)

Flag to return initialized `dlnetwork`, specified as a numeric or logical 1 (true) or 0 (false).

If `tf` is 1, then the software initializes the learnable and state parameters of `net` with initial values for training, according to the network input layer or the example inputs provided.

If `tf` is 0, then the software does not initialize the learnable and state parameters. Before you use an uninitialized network, you must first initialize it using the `initialize` function. Example inputs are not supported when `tf` is false.

### **prunableNet — Network for pruning by using first-order Taylor approximation**

TaylorPrunableNetwork object

Network for pruning by using first-order Taylor approximation, specified as a TaylorPrunableNetwork object.

## **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 `darray` 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

<code>predict</code>	Compute deep learning network output for inference
<code>forward</code>	Compute deep learning network output for training
<code>initialize</code>	Initialize learnable and state parameters of a <code>dlnetwork</code>
<code>layerGraph</code>	Graph of network layers for deep learning
<code>setL2Factor</code>	Set L2 regularization factor of layer learnable parameter
<code>setLearnRateFactor</code>	Set learn rate factor of layer learnable parameter
<code>getLearnRateFactor</code>	Get learn rate factor of layer learnable parameter
<code>getL2Factor</code>	Get L2 regularization factor of layer learnable parameter
<code>resetState</code>	Reset state parameters of neural network
<code>plot</code>	Plot neural network architecture
<code>addInputLayer</code>	Add input layer to network
<code>addLayers</code>	Add layers to layer graph or network
<code>removeLayers</code>	Remove layers from layer graph or network
<code>connectLayers</code>	Connect layers in layer graph or network
<code>disconnectLayers</code>	Disconnect layers in layer graph or network
<code>replaceLayer</code>	Replace layer in layer graph or network
<code>summary</code>	Print network summary

## 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

```

View summary with `summary`.

## Create Initialized `dlnetwork` with Unconnected Inputs

Use network data layout objects to create a multi-input `dlnetwork` that is ready for training. The software uses the size and format information 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)
    groupNormalizationLayer("all-channels")
    reluLayer
    convolution2dLayer(3,numFilters,Padding="same")
    groupNormalizationLayer("channel-wise")
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer];

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

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

```

Create network data layout objects that represent the size and format of 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 convolution layer in the first branch. Use an input of size 64-by-64 with 18 channels for the convolution layer in the second branch.

```

X1 = dlarray(rand([64 64 3 32]),"SSCB");
X2 = 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`.

```
net = dlnetwork(lgraph,X1,X2);
```

Check that the network is initialized and ready for training by inspecting the `Initialized` property of the network.

```
net.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.

You can train most types of neural networks using the `trainNetwork` and `trainingOptions` functions. If the `trainingOptions` function does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using `dlarray` and `dlnetwork` objects for automatic differentiation. For an example showing how to retrain a pretrained deep learning network using the `trainNetwork` function, see “Transfer Learning Using Pretrained Network”.

Training a deep neural network is an optimization task. By considering a neural network as a function  $f(X; \theta)$ , where  $X$  is the network input, and  $\theta$  is the set of learnable parameters, you can optimize  $\theta$  so that it minimizes some loss value based on the training data. For example, optimize the learnable parameters  $\theta$  such that for a given inputs  $X$  with a corresponding targets  $T$ , they minimize the error between the predictions  $Y = f(X; \theta)$  and  $T$ .

The loss function used depends on the type of task. For example:

- For classification tasks, you can minimize the cross entropy error between the predictions and targets.
- For regression tasks, you can minimize the mean squared error between the predictions and targets.

You can optimize the objective using gradient descent: minimize the loss  $L$  by iteratively updating the learnable parameters  $\theta$  by taking steps towards the minimum using the gradients of the loss with respect to the learnable parameters. Gradient descent algorithms typically update the learnable parameters by using a variant of an update step of the form  $\theta_{t+1} = \theta_t - \rho \nabla L$ , where  $t$  is the iteration number,  $\rho$  is the learning rate, and  $\nabla L$  denotes the gradients (the derivatives of the loss with respect to the learnable parameters).

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  $\rho_t = \frac{\rho_0}{1+kt}$ , where  $t$  is the iteration number,  $\rho_0$  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=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);
```

Determine the number of classes in the training data.

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

## Define Network

Define the network for image classification.

- For image input, specify an image input layer with input size matching the training data.
- Do not normalize the image input, set the `Normalization` option of the input layer to `"none"`.
- Specify three convolution-batchnorm-ReLU blocks.
- Pad the input to the convolution layers such that the output has the same size by setting the `Padding` option to `"same"`.
- For the first convolution layer specify 20 filters of size 5. For the remaining convolution layers specify 20 filters of size 3.
- For classification, specify a fully connected layer with size matching the number of classes
- To map the output to probabilities, include a softmax layer.

When training a network using a custom training loop, do not include an output layer.

```
layers = [
    imageInputLayer(inputSize,Normalization="none")
    convolution2dLayer(5,20,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,20,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,20,Padding="same")
```

```
batchNormalizationLayer
reluLayer
fullyConnectedLayer(numClasses)
softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers)
```

```
net =
  dlnetwork with properties:

    Layers: [12x1 nnet.cnn.layer.Layer]
  Connections: [11x2 table]
  Learnables: [14x3 table]
    State: [6x3 table]
  InputNames: {'imageinput'}
  OutputNames: {'softmax'}
  Initialized: 1
```

View summary with `summary`.

### Define Model Loss Function

Training a deep neural network is an optimization task. By considering a neural network as a function  $f(X; \theta)$ , where  $X$  is the network input, and  $\theta$  is the set of learnable parameters, you can optimize  $\theta$  so that it minimizes some loss value based on the training data. For example, optimize the learnable parameters  $\theta$  such that for a given inputs  $X$  with a corresponding targets  $T$ , they minimize the error between the predictions  $Y = f(X; \theta)$  and  $T$ .

Create the function `modelLoss`, listed in the Model Loss Function on page 1-562 section of the example, that takes as input the `dlnetwork` object, a mini-batch of input data with corresponding targets, and returns the loss, the gradients of the loss with respect to the learnable parameters, and the network state.

### 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 format 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 Computing Requirements” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(augimdsTrain,...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB" ""]);
```

Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Calculate the total number of iterations for the training progress monitor.

```
numObservationsTrain = numel(imdsTrain.Files);
numIterationsPerEpoch = ceil(numObservationsTrain / miniBatchSize);
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info=["Epoch","LearnRate"],XLabel="Iteration");
```

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 loss, gradients, and state using the `dlfeval` and `modelLoss` 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.
- Update the loss, learn rate, and epoch values in the training progress monitor.
- Stop if the `Stop` property is true. The `Stop` property value of the `TrainingProgressMonitor` object changes to true when you click the Stop button.

```
epoch = 0;
iteration = 0;

% Loop over epochs.
while epoch < numEpochs && ~monitor.Stop

    epoch = epoch + 1;

    % Shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq) && ~monitor.Stop

        iteration = iteration + 1;
```

```

% Read mini-batch of data.
[X,T] = next(mbq);

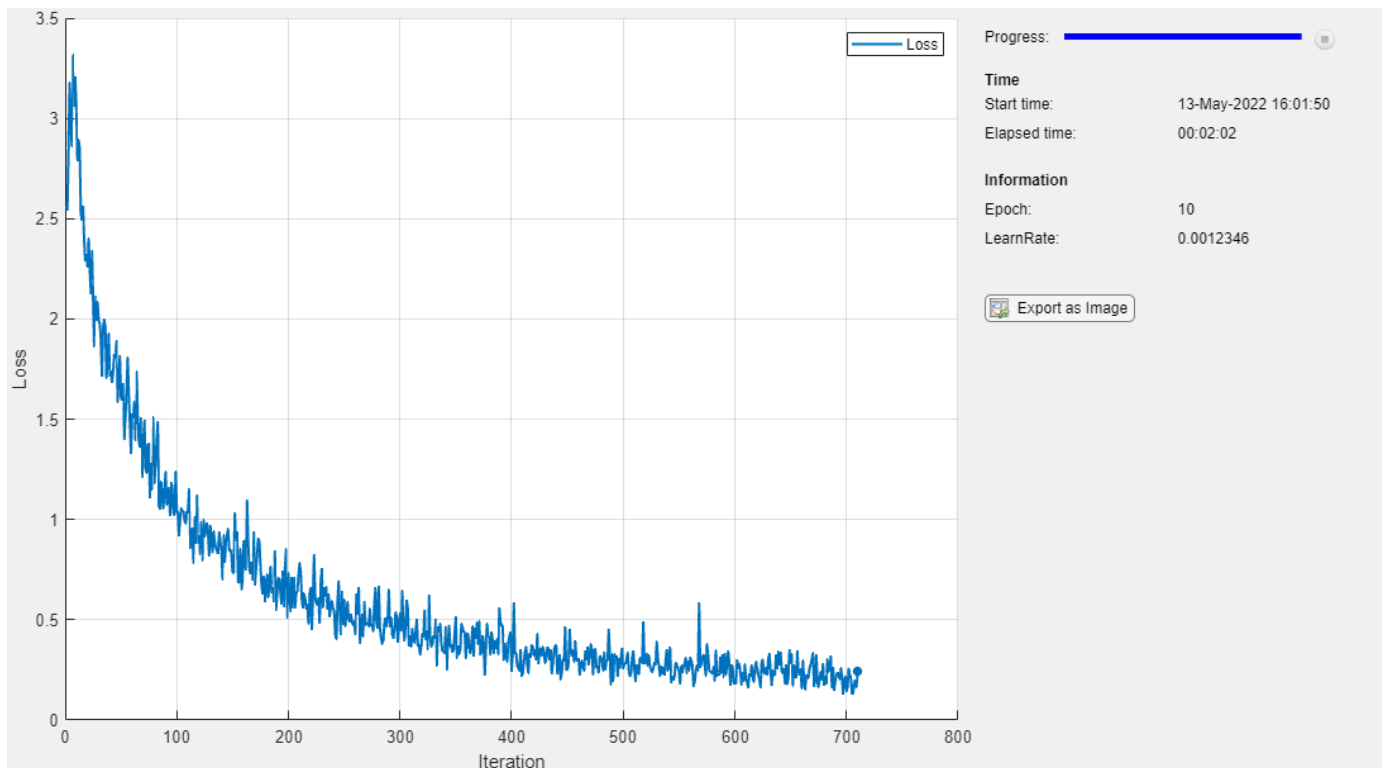
% Evaluate the model gradients, state, and loss using dlfeval and the
% modelLoss function and update the network state.
[loss,gradients,state] = dlfeval(@modelLoss,net,X,T);
net.State = state;

% Determine learning rate for time-based decay learning rate schedule.
learnRate = initialLearnRate/(1 + decay*iteration);

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

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch,LearnRate=learnRate);
monitor.Progress = 100 * iteration/numIterations;
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.

```
YTest = modelPredictions(net,mbqTest,classes);
```

Evaluate the classification accuracy.

```
TTest = imdsValidation.Labels;  
accuracy = mean(TTest == YTest)
```

```
accuracy = 0.9750
```

Visualize the predictions in a confusion chart.

```
figure  
confusionchart(TTest,YTest)
```

0	100									
1		95	4						1	
2			98	2						
3				100						
4					100					
5				1		99				
6			1			1	97		1	
7		1				1		98		
8			3	1		3	1		92	
9			1	3						96
	0	1	2	3	4	5	6	7	8	9

Large values on the diagonal indicate accurate predictions for the corresponding class. Large values on the off-diagonal indicate strong confusion between the corresponding classes.

### Supporting Functions

#### Model Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding targets `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients automatically, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)

% Forward data through network.
[Y,state] = forward(net,X);

% Calculate cross-entropy loss.
loss = crossentropy(Y,T);

% Calculate gradients of loss with respect to learnable parameters.
gradients = dlgradient(loss,net.Learnables);

end
```

## Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `net`, 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 Y = modelPredictions(net,mbq,classes)

Y = [];

% Loop over mini-batches.
while hasdata(mbq)
    X = next(mbq);

    % Make prediction.
    scores = predict(net,X);

    % Decode labels and append to output.
    labels = onehotdecode(scores,classes,1)';
    Y = [Y; labels];
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,T] = preprocessMiniBatch(dataX,dataT)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(dataX);

% Extract label data from cell and concatenate.
T = cat(2,dataT{1:end});

% One-hot encode labels.
T = onehotencode(T,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(dataX)

% Concatenate.
X = cat(4,dataX{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)
```

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
 <code>imageInputLayer</code>	An image input layer inputs 2-D images to a neural network and applies data normalization.

Layer	Description
 image3dInputLayer	A 3-D image input layer inputs 3-D images or volumes to a neural network and applies data normalization.
 sequenceInputLayer	A sequence input layer inputs sequence data to a neural network.
 featureInputLayer	A feature input layer inputs feature data to a neural 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.
 transposedConv2dLayer	A transposed 2-D convolution layer upsamples two-dimensional 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 neural network.
 lstmLayer	An LSTM layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data.
 lstmLayer	An LSTM projected layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data using projected learnable weights.




Layer	Description
 bilstmLayer	A bidirectional LSTM (BiLSTM) layer is an RNN layer that learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the RNN to learn from the complete time series at each time step.
 gruLayer	A GRU layer is an RNN layer that 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.
 wordEmbeddingLayer	A word embedding layer maps word indices to vectors.





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 <i>clipping ceiling</i> 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.
 geluLayer	A Gaussian error linear unit (GELU) layer weights the input by its probability under a Gaussian distribution.
 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).


Layer	Description
 functionLayer	A function layer applies a specified function to the layer input.

**Normalization 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.



**Utility Layers**

Layer	Description
 dropoutLayer	A dropout layer randomly sets input elements to zero with a given probability.
 crop2dLayer	A 2-D crop layer applies 2-D cropping to the input.
 stftLayer	An STFT layer computes the short-time Fourier transform of the input.
 cwtLayer	A CWT layer computes the CWT of the input.





Layer	Description
 modwtLayer	A MODWT layer computes the MODWT and MODWT multiresolution analysis (MRA) 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 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 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.

## Version History

### Introduced in R2019b

#### R2021a: 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,net.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.
- The `Initialized` property of the `dlnetwork` object must be set to `true`.
- You can generate code for `dlnetwork` that have vector or image 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:
  - `dlnetwork` containing vector sequences that have 'CT' or 'CBT' data formats.
  - `dlnetwork` containing image sequences that have 'SSCT' or 'SSCBT' data formats.
  - Multi-input `dlnetwork` with heterogeneous input layers. For RNN networks, multiple input is not supported.
  - `dlnetwork` inputs with a variable-size time (T) dimension. You use such `dlnetwork` objects to represent time series data of variable sequence length.
- Code generation supports only the `predict` object function. The `dlnetwork` input to the `predict` method must be a single datatype.
- 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.
- The `Initialized` property of the `dlnetwork` object must be set to `true`.
- You can generate code for `dlnetwork` that have vector and image sequence inputs. Code generation support includes:
  - `dlnetwork` containing vector sequences that have 'CT' or 'CBT' data formats.
  - `dlnetwork` containing image sequences that have 'SSCT' or 'SSCBT' data formats.
  - Multi-input `dlnetwork` with heterogeneous input layers. For RNN networks, multiple input is not supported.
  - `dlnetwork` inputs with a variable-size time (T) dimension. You use such `dlnetwork` objects to represent time series data of variable sequence length.
- Code generation supports only the `predict` object function. The `dlnetwork` 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 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` | `TaylorPrunableNetwork`

## **Topics**

“Train Network Using Custom Training Loop”

“Train Generative Adversarial Network (GAN)”

“Automatic Differentiation Background”

“Define Custom Training Loops, Loss Functions, and Networks”

# predict

Compute deep learning network output for inference

## Syntax

```
Y = predict(net,X)
Y = predict(net,X1,...,XM)
[Y1,...,YN] = predict(____)
[Y1,...,YK] = 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`.

---

`Y = predict(net,X)` returns the network output `Y` during inference given the input data `X` and the network `net` with a single input and a single output.

`Y = predict(net,X1,...,XM)` returns the network output `Y` during inference given the `M` inputs `X1, ..., XM` and the network `net` that has `M` inputs and a single output.

`[Y1,...,YN] = predict(____)` returns the `N` outputs `Y1, ..., YN` during inference for networks that have `N` outputs using any of the previous syntaxes.

`[Y1,...,YK] = predict(____,'Outputs',layerNames)` returns the outputs `Y1, ..., YK` 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 Computing Requirements” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(imds, ...
    "MiniBatchSize", miniBatchSize, ...
    "MiniBatchFcn", @preprocessMiniBatch, ...
    "MiniBatchFormat", "SSCB");
```

Loop over the minibatches of data and make predictions using the `predict` function. Use the `onehotdecode` function to determine the class labels. Store the predicted class labels.

```
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.

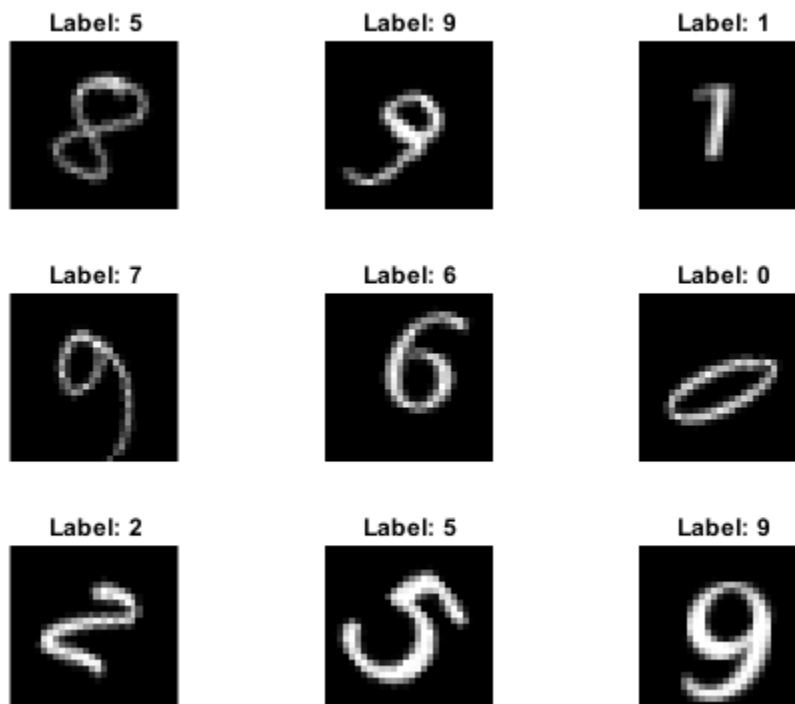
```

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

#### **net** — Network for custom training loops or custom pruning loops

`dlnetwork` object | `TaylorPrunableNetwork` object

This argument can represent either of these:

- Network for custom training loops, specified as a `dlnetwork` object.
- Network for custom pruning loops, specified as a `TaylorPrunableNetwork` object.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

#### **X** — 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 Computing Requirements” (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 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.

For quantized networks, the `'mex'` option requires a CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3, or higher.

Example: `'Acceleration','mex'`

## Output Arguments

### **Y** — 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.

## Algorithms

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Version History

### Introduced in R2019b

#### **R2021a: 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,net.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:
  - `Y = predict(net,X)`
  - `Y = predict(net,X1,...,XM)`
  - `[Y1,...,YN] = predict(__)`
  - `[Y1,...,YK] = predict(__,'Outputs',layerNames)`
- The input data `X` 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:
  - `Y = predict(net,X)`
  - `Y = predict(net,X1,...,XM)`
  - `[Y1,...,YN] = predict(__)`
  - `[Y1,...,YK] = predict(__,'Outputs',layerNames)`
- The input data `X` 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 `[Y1,...,YK] = 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 `net.Learnables.Value` are `dlarray` objects with underlying data of type `gpuArray`
  - The input argument `X` 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` | `TaylorPrunableNetwork`

### Topics

“Train Generative Adversarial Network (GAN)”

“Automatic Differentiation Background”

“Define Custom Training Loops, Loss Functions, and Networks”

## dlode45

Deep learning solution of nonstiff ordinary differential equation (ODE)

### Syntax

```
Y = dlode45(net,tspan,Y0)
Y = dlode45(odefun,tspan,Y0,theta)
Y = dlode45( ____,DataFormat=FMT)
Y = dlode45( ____,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` 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`.

---

`Y = dlode45(net,tspan,Y0)` integrates the system of ODEs characterized by the `dlnetwork` object `net` on the time interval defined by the first and last elements of `tspan` and with the initial conditions `Y0`.

`Y = dlode45(odefun,tspan,Y0,theta)` integrates the system of ODEs given by the function handle `odefun` on the time interval defined by the first and last elements of `tspan`, with the initial conditions `Y0` and parameters `theta`.

`Y = dlode45( ____,DataFormat=FMT)` specifies the data format for the unformatted initial conditions `Y0`. The format must contain "S" (spatial), "C" (channel), and "B" (batch) dimension labels only.

`Y = dlode45( ____,Name=Value)` specifies additional options using one or more name-value arguments. For example, `Y = dlode45(odefun,tspan,Y0,theta,GradientMode="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 `darray` object containing 128 observations of numeric feature data with 4 channels. Specify the format "CB" (channel, batch).

```

miniBatchSize = 128;
numChannels = 4;
Y0 = rand(numChannels,miniBatchSize);
Y0 = dldarray(Y0,"CB");

```

View the size and format of the initial conditions.

```

size(Y0)
ans = 1×2
     4    128

```

```

dims(Y0)

```

```

ans =
'CB'

```

Create a `dlnetwork` object that characterizes the ODE function given by two fully connect operations with a tanh activation between them.

```

layers = [
    featureInputLayer(4)
    fullyConnectedLayer(10)
    tanhLayer
    fullyConnectedLayer(4)];
net = dlnetwork(layers);

```

Specify an interval of integration of [0 0.1].

```

tspan = [0 0.1];

```

Apply the neural ODE operation.

```

Y = dlode45(net,tspan,Y0);

```

View the size and format of the output.

```

size(Y)
ans = 1×2
     4    128

```

```

dims(Y)

```

```

ans =
'CB'

```

### Specify Custom Neural ODE Function

For the initial conditions, create a formatted `dldarray` 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);
Y0 = darray(Y0,"SSCB");
```

View the size and format of the initial conditions.

```
size(Y0)
ans = 1×4
    28    28    64   128
```

```
dims(Y0)
```

```
ans =
'SSCB'
```

Specify the ODE function. Define the function `odeModel`, listed in the ODE Function on page 1-582 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 = darray(rand(filterSize(1),filterSize(2),numChannels,numFilters));
parameters.Bias = darray(zeros(1,numFilters));
```

Specify an interval of integration of [0 0.1].

```
tspan = [0 0.1];
```

Apply the neural ODE operation.

```
Y = dlode45(odefun,tspan,Y0,parameters);
```

View the size and format of the output.

```
size(Y)
ans = 1×4
    28    28    64   128
```

```
dims(Y)
```

```
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

### **net** — Neural network characterizing neural ODE function

initialized `dlnetwork` object

Neural network characterizing neural ODE function, specified as an initialized `dlnetwork` object.

If `net` has one input, then `predict(net,Y)` defines the ODE system. If `net` has two inputs, then `predict(net,T,Y)` defines the ODE system, where `T` is a time step repeated over the batch dimension.

When the first argument is a `dlnetwork` object and `GradientMode` is "adjoint", the network `State` property must be empty. To use a network with a nonempty `State` property, set `GradientMode` to "direct".

### **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 `darray` 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 `Y0` 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 `dode45` 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.

### **Y0 — Initial conditions**

`darray` object

Initial conditions, specified as a formatted or unformatted `darray` object.

If `Y0` is an unformatted `darray`, 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**

`dlnetwork` object | `darray` object | cell array | structure | table

Parameters of ODE function, specified as one of these values:

- `dlnetwork` object
- `darray` object
- Cell array of one or more of these values:
  - `dlnetwork` object
  - `darray` object
- Structure or nested structures of one or more of these values:

- `dlnetwork` object
- `dlarray` object
- 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: `Y = dlode45(odefun,tspan,Y0,theta,GradientMode="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 these options:

- "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 `Y0` 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.

When the first argument is a `dlnetwork` object and `GradientMode` is "adjoint", the network `State` property must be empty. To use a network with a nonempty `State` property, set `GradientMode` to "direct".

The `dlaccelerate` function does not support accelerating the `dlode45` function when the `GradientMode` option is "direct". 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` function with the `GradientMode` option set to "direct".

---

**Warning** When `GradientMode` is "adjoint", `odefun` must support function acceleration. Otherwise, the function can return unexpected results.

When `GradientMode` is "adjoint", the software traces the ODE function input to determine the computation graph used for automatic differentiation. This tracing process can take some time and can end up recomputing the same trace. By optimizing, caching, and reusing the traces, the software can speed up the gradient computation.

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)

For more information on deep learning function acceleration, see "Deep Learning Function Acceleration for Custom Training Loops".

---

### **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**

### **Y — 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 `Y0`.

If `Y0` is a formatted `dlarray` and `tspan` contains exactly two elements, then `Y` has the same format as `Y0`. If `Y0` is not a formatted `dlarray` and `tspan` contains exactly two elements, then `Y` is an unformatted `dlarray` with the same dimension order as `Y0`.

If `Y0` is a formatted `dlarray` and `tspan` contains more than two elements, then `Y` has the same format as `Y0` with an additional appended "T" (time) dimension. If `Y0` is not a formatted `dlarray` and `tspan` contains more than two elements, then `Y` is an unformatted `dlarray` with the same dimension order as `Y0` 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  $\theta$  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].

## **Version History**

**Introduced in R2021b**

### **R2023a: Specify system of ODEs or learnable parameters as a neural network**

Specify a system of ODEs as a neural network by specifying a `dlnetwork` object as the first input argument `net`.

Specify learnable parameters `theta` as one of these values:

- `dlnetwork` object
- Cell array of `dlnetwork` and `dlarray` objects
- Structure of `dlnetwork` and `dlarray` objects or nested structures of `dlnetwork` and `dlarray` objects

## 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](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"

# 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.

This object requires Deep Learning Toolbox Model Quantization Library. 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 — Metric function to use for validation of quantized network

cell array of function handles

Metric function to use for validation of quantized network, specified as a cell array of one or more function handles.

```
Example: options = dlquantizationOptions('MetricFcn',
{@(x)hComputeModelAccuracy(x,net,groundTruth)});
```

Data Types: cell

### Execution Environment Options

#### Bitstream — Name of FPGA bitstream

'zcu102\_int8' | 'zc706\_int8' | 'arria10soc\_int8'

*This property is valid only when the 'ExecutionEnvironment' property of the `dlquantizer` object is set to 'FPGA'.*

Name of the FPGA bitstream specified, as one of these values:

Bitstream	Target Board
'zcu102_int8'	Xilinx Zynq UltraScale™ ZCU102
'zc706_int8'	Xilinx Zynq-7000 ZC706
'arria10soc_int8'	Intel Arria 10 SoC development kit

Example: `quantOpts = dlquantizationOptions('Bitstream','zcu102_int8')`

### Target – Target for quantized network

`dlhdl.Target` object | `raspi` object

Target for quantized network, specified as one of the following:

Target	Execution Environment for Quantized Network	Example
Target CPU board specified as a <code>raspi</code> object	Set Target property as a <code>raspi</code> object only when 'ExecutionEnvironment' property of the <code>dlquantizer</code> object is set to 'CPU'	<pre>r = raspi('hostname', 'User Name','Password' ); quantOpts = dlquantizationOptions('Target',r )</pre>
Target FPGA board vendor name and interface, specified as a <code>dlhdl.Target</code> object	Set Target property as a <code>dlhdl.Target</code> object only when 'ExecutionEnvironment' property of the <code>dlquantizer</code> object is set to 'FPGA'	<pre>hTarget = dlhdl.Target('In tel','Interface' ,'JTAG'); quantOpts = dlquantizationOptions('Target',h Target)</pre>

## 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 squeezeNetmerch
net

net =
  DAGNetwork with properties:
```



```

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 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=121x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.9198
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.0792
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.3
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1164
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.740
{'fire2-expand1x1_Bias' }	{'fire2-expand1x1' }	"Bias"	-0.06005
{'fire2-expand3x3_Weights' }	{'fire2-expand3x3' }	"Weights"	-0.7439
{'fire2-expand3x3_Bias' }	{'fire2-expand3x3' }	"Bias"	-0.05177
{'fire3-squeeze1x1_Weights' }	{'fire3-squeeze1x1' }	"Weights"	-0.771
{'fire3-squeeze1x1_Bias' }	{'fire3-squeeze1x1' }	"Bias"	-0.1013
{'fire3-expand1x1_Weights' }	{'fire3-expand1x1' }	"Weights"	-0.7203
{'fire3-expand1x1_Bias' }	{'fire3-expand1x1' }	"Bias"	-0.06702
{'fire3-expand3x3_Weights' }	{'fire3-expand3x3' }	"Weights"	-0.6144
{'fire3-expand3x3_Bias' }	{'fire3-expand3x3' }	"Bias"	-0.05361
{'fire4-squeeze1x1_Weights' }	{'fire4-squeeze1x1' }	"Weights"	-0.742
{'fire4-squeeze1x1_Bias' }	{'fire4-squeeze1x1' }	"Bias"	-0.1088
:			

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: [1x1 struct]
Statistics: [2x2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

ans=2x2 table

NetworkImplementation	MetricOutput
{'Floating-Point' }	1
{'Quantized' }	1

```
valResults.Statistics
```

```
ans=2x2 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 Network for FPGA Deployment

Reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. This example shows how to use Deep Learning Toolbox Model Quantization Library and Deep Learning HDL Toolbox to deploy the `int8` network to a target FPGA board.

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning.

## Load Pretrained Network

Load the pretrained LogoNet network and analyze the network architecture.

```
snet = getLogoNetwork;
deepNetworkDesigner(snet);
```

## Load Data

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 for calibration and validation. Expedite calibration and validation 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. The FPGA validation workflow has a maximum limit of one image when validating the quantized network. Reduce the FPGA validation data set to a single image.

```
curDir = pwd;
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_simulation = validationData.subset(1:5);
validationData_FPGA = validationData.subset(1:1);
```

### Generate Calibration Result File for the Network

Create a `dlquantizer` (Deep Learning HDL Toolbox) object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj_simulation = dlquantizer(snet,'ExecutionEnvironment',"FPGA",'Simulation','on');
dlQuantObj_FPGA = dlquantizer(snet,'ExecutionEnvironment',"FPGA");
```

Use the `calibrate` (Deep Learning HDL Toolbox) function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases. The `calibrate` function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj_simulation,calibrationData_reduced)
```

ans=35x5 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"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045962
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05016
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050706
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02951
{'imageinput' }	{'imageinput' }	"Activations"	( )
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:	:	:	:

```
calibrate(dlQuantObj_FPGA,calibrationData_reduced)
```

ans=35x5 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"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045962
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05016

```

{'fc_2_Bias'           }      {'fc_2'           }      "Bias"           -0.001756
{'fc_3_Weights'       }      {'fc_3'           }      "Weights"        -0.05070
{'fc_3_Bias'          }      {'fc_3'           }      "Bias"           -0.0295
{'imageinput'         }      {'imageinput'}     "Activations"    (
{'imageinput_normalization'}  {'imageinput'}     "Activations"    -139.3
:

```

## Create Target Object

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 use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado
\2020.2\bin\vivado.bat');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, you can also use the JTAG interface.

```
% hTarget = dlhdl.Target('Xilinx', 'Interface', 'JTAG');
```

## Create dlQuantizationOptions Object

Create a `dlquantizationOptions` object. Specify the target bitstream and target board interface. The default metric function is a Top-1 accuracy metric function.

```
options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);
options_simulation = dlquantizationOptions;
```

To use a custom metric function, specify the metric function in the `dlquantizationOptions` object.

```
options_FPGA = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_FPGA)});
options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_simulation)});
```

## Validate Quantized Network

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_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulation);
```

```

Compiling leg: conv_1>>relu_4 ...
Compiling leg: conv_1>>relu_4 ... complete.
Compiling leg: maxpool_4 ...
Compiling leg: maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
Compiling leg: fc_1>>fc_3 ... complete.

```

```

prediction_simulation = struct with fields:
    NumSamples: 5
    MetricResults: [1x1 struct]

```

Statistics: []

For validation on an FPGA, the validate function:

- Programs the FPGA board by using the output of the compile method and the programming file
- Downloads the network weights and biases
- Compares the performance of the network before and after quantization

```
prediction_FPGA = dlQuantObj_FPGA.validate(validationData_FPGA,options_FPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
```

```
### Targeting FPGA bitstream zcu102_int8.
```

```
### The network includes the following layers:
```

1	'imageinput'	Image Input	227x227x3 images with 'zerocenter' normalization
2	'conv_1'	Convolution	96 5x5x3 convolutions with stride [1 1] and padding
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	3x3 max pooling with stride [2 2] and padding
5	'conv_2'	Convolution	128 3x3x96 convolutions with stride [1 1] and padding
6	'relu_2'	ReLU	ReLU
7	'maxpool_2'	Max Pooling	3x3 max pooling with stride [2 2] and padding
8	'conv_3'	Convolution	384 3x3x128 convolutions with stride [1 1] and padding
9	'relu_3'	ReLU	ReLU
10	'maxpool_3'	Max Pooling	3x3 max pooling with stride [2 2] and padding
11	'conv_4'	Convolution	128 3x3x384 convolutions with stride [2 2] and padding
12	'relu_4'	ReLU	ReLU
13	'maxpool_4'	Max Pooling	3x3 max pooling with stride [2 2] and padding
14	'fc_1'	Fully Connected	2048 fully connected layer
15	'relu_5'	ReLU	ReLU
16	'dropout_1'	Dropout	50% dropout
17	'fc_2'	Fully Connected	2048 fully connected layer
18	'relu_6'	ReLU	ReLU
19	'dropout_2'	Dropout	50% dropout
20	'fc_3'	Fully Connected	32 fully connected layer
21	'softmax'	Softmax	softmax
22	'classoutput'	Classification Output	crossentropyex with 'adidas' and 31 other classes

```
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
```

```
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
```

```
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software.
```

```
### Compiling layer group: conv_1>>relu_4 ...
```

```
### Compiling layer group: conv_1>>relu_4 ... complete.
```

```
### Compiling layer group: maxpool_4 ...
```

```
### Compiling layer group: maxpool_4 ... complete.
```

```
### Compiling layer group: fc_1>>fc_3 ...
```

```
### Compiling layer group: fc_1>>fc_3 ... complete.
```

```
### Allocating external memory buffers:
```

offset_name	offset_address	allocated_space
"InputDataOffset"	"0x00000000"	"12.0 MB"
"OutputResultOffset"	"0x00c00000"	"4.0 MB"
"SchedulerDataOffset"	"0x01000000"	"4.0 MB"
"SystemBufferOffset"	"0x01400000"	"36.0 MB"
"InstructionDataOffset"	"0x03800000"	"8.0 MB"

```

"ConvWeightDataOffset"    "0x04000000"    "12.0 MB"
"FCWeightDataOffset"     "0x04c00000"    "12.0 MB"
"EndOffset"              "0x05800000"    "Total: 88.0 MB"

```

```
### Network compilation complete.
```

```

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Deep learning network programming has been skipped as the same network is already loaded on the target
### Finished writing input activations.
### Running single input activation.

```

#### Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	248358	274080	90.62
DSPs	384	2520	15.24
Block RAM	581	912	63.71

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP blocks

```

### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer and a normalization layer
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software

```

#### Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
Network	40142478	0.18247	1	40142478
__imageinput_norm	216472	0.00098		
__conv_1	6825671	0.03103		
__maxpool_1	3755088	0.01707		
__conv_2	10440701	0.04746		
__maxpool_2	1447840	0.00658		
__conv_3	9405685	0.04275		
__maxpool_3	1765856	0.00803		
__conv_4	1819636	0.00827		
__maxpool_4	28098	0.00013		
__fc_1	2651288	0.01205		
__fc_2	1696632	0.00771		
__fc_3	89511	0.00041		

\* The clock frequency of the DL processor is: 220MHz

#### Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	168645	274080	61.53
DSPs	800	2520	31.75
Block RAM	453	912	49.67

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP blocks

```

### Finished writing input activations.
### Running single input activation.

prediction_FPGA = struct with fields:
    NumSamples: 1
    MetricResults: [1x1 struct]
    Statistics: [2x7 table]

```

### View Performance of Quantized Neural Network

Display the accuracy of the quantized network.

```
prediction_simulation.MetricResults.Result
```

ans=2x2 table

NetworkImplementation	MetricOutput
{'Floating-Point'}	1
{'Quantized' }	1

```
prediction_FPGA.MetricResults.Result
```

ans=2x2 table

NetworkImplementation	MetricOutput
{'Floating-Point'}	1
{'Quantized' }	1

Display the performance of the quantized network in frames per second.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
```

ans = 19.0828

### Quantize a Neural Network for CPU Target

This example shows how to quantize and validate a neural network for a CPU target. This workflow is similar to other execution environments, but before validating you must establish a `raspi` connection and specify it as target using `dlquantizationOptions`.

First, load your network. This example uses the pretrained network `squeezenet`.

```
load squeezenetmerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```



Then define your calibration and validation data, `calDS` and `valDS` respectively.

```
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 the `dlquantizer` object and specify a CPU execution environment.

```
dq = dlquantizer(net,'ExecutionEnvironment','CPU')

dq =
    dlquantizer with properties:
```

```
    NetworkObject: [1x1 DAGNetwork]
    ExecutionEnvironment: 'CPU'
```

Calibrate the network.

```
calResults = calibrate(dq,aug_calData,'UseGPU','off')
```

`calResults=122x5 table`

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.9198
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.0792
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.3
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1164
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.740
{'fire2-expand1x1_Bias' }	{'fire2-expand1x1' }	"Bias"	-0.06005
{'fire2-expand3x3_Weights' }	{'fire2-expand3x3' }	"Weights"	-0.7439
{'fire2-expand3x3_Bias' }	{'fire2-expand3x3' }	"Bias"	-0.05177
{'fire3-squeeze1x1_Weights' }	{'fire3-squeeze1x1' }	"Weights"	-0.771
{'fire3-squeeze1x1_Bias' }	{'fire3-squeeze1x1' }	"Bias"	-0.1013
{'fire3-expand1x1_Weights' }	{'fire3-expand1x1' }	"Weights"	-0.7203
{'fire3-expand1x1_Bias' }	{'fire3-expand1x1' }	"Bias"	-0.06702
{'fire3-expand3x3_Weights' }	{'fire3-expand3x3' }	"Weights"	-0.6144
{'fire3-expand3x3_Bias' }	{'fire3-expand3x3' }	"Bias"	-0.05361
{'fire4-squeeze1x1_Weights' }	{'fire4-squeeze1x1' }	"Weights"	-0.742
{'fire4-squeeze1x1_Bias' }	{'fire4-squeeze1x1' }	"Bias"	-0.1088
:			

Use the MATLAB Support Package for Raspberry Pi Hardware function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name or address of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiname','username','password')
```

For example,

```
r = raspi('gpcoder-raspberrypi-7','pi','matlab')

r =
  raspi with properties:
      DeviceAddress: 'gpcoder-raspberrypi-7'
          Port: 18734
      BoardName: 'Raspberry Pi 3 Model B+'
      AvailableLEDs: {'led0'}
  AvailableDigitalPins: [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
  AvailableSPIChannels: {}
      AvailableI2CBuses: {}
      AvailableWebcams: {}
          I2CBusSpeed:
  AvailableCANInterfaces: {}

  Supported peripherals
```

Specify `raspi` object as the target for the quantized network.

```
opts = dlquantizationOptions('Target',r)

opts =
  dlquantizationOptions with properties:
      MetricFcn: {}
      Bitstream: ''
      Target: [1x1 raspi]
```

Validate the quantized network with the `validate` function.

```
valResults = validate(dq, aug_valData, opts)

### Starting application: 'codegen\lib\validate_predict_int8\pil\validate_predict_int8.elf'
  To terminate execution: clear validate_predict_int8_pil
### Launching application validate_predict_int8.elf...
### Host application produced the following standard output (stdout) and standard error (stderr)

valResults = struct with fields:
  NumSamples: 20
  MetricResults: [1x1 struct]
  Statistics: []
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result

ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}    0.95
  {'Quantized' }        0.95
```

## Version History

Introduced in R2020a

### **R2023a: Specify Raspberry Pi as quantization target**

*Behavior changed in R2023a*

You can now specify a `raspi` object as the target for quantization using the `Target` property when `dlquantizer Execution Environment` is set to CPU.

## See Also

### **Apps**

**Deep Network Quantizer**

### **Functions**

`dlquantizer` | `calibrate` | `validate` | `quantize` | `quantizationDetails` | `estimateNetworkMetrics`

### **Topics**

“Quantization of Deep Neural Networks”

“Quantize Layers in Object Detectors and Generate CUDA Code”

“Classify Images on FPGA Using Quantized Neural Network” (Deep Learning HDL Toolbox)

“Generate INT8 Code for Deep Learning Network on Raspberry Pi” (MATLAB Coder)

# 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. You can create and verify the behavior of a quantized network for GPU, FPGA, CPU deployment, or explore the quantized network in MATLAB.

For CPU and GPU deployment, the software generates code for a convolutional deep neural network by quantizing the weights, biases, and activations of the convolution layers to 8-bit scaled integer data types. The quantization is performed by providing the calibration result file produced by the `calibrate` function to the `codegen` command.

Code generation does not support quantized deep neural networks produced by the `quantize` function.

This object requires Deep Learning Toolbox Model Quantization Library. To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

## Creation

### Syntax

```
quantObj = dlquantizer(net)  
quantObj = dlquantizer(net,Name,Value)
```

### Description

`quantObj = dlquantizer(net)` creates a `dlquantizer` object for the specified deep neural network, `net`.

`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.

### Input Arguments

#### **net** — Pretrained neural network

DAGNetwork object | dlnetwork object | SeriesNetwork object | yolov2objectDetector object | yolov3objectDetector object | yolov4objectDetector object | ssdobjectDetector object

Pretrained neural network, specified as a `DAGNetwork`, `dlnetwork`, `SeriesNetwork`, `yolov2objectDetector`, `yolov3objectDetector`, `yolov4objectDetector`, or a `ssdobjectDetector` object.

## Properties

### NetworkObject — Pretrained neural network

DAGNetwork object | dlnetwork object | SeriesNetwork object | yolov20objectDetector object | yolov30objectDetector | yolov40objectDetector | ssd0objectDetector object

This property is read-only.

Pre-trained neural network, specified as a DAGNetwork, dlnetwork, SeriesNetwork, yolov20objectDetector, yolov30objectDetector, yolov40objectDetector, or a ssd0objectDetector object.

### ExecutionEnvironment — Execution environment

'GPU' (default) | 'FPGA' | 'CPU' | 'MATLAB'

Execution environment for the quantized network, specified as 'GPU', 'FPGA', 'CPU', or 'MATLAB'. How the network is quantized depends on the choice of execution environment.

The 'MATLAB' execution environment indicates a target-agnostic quantization of the neural network will be performed. This option does not require you to have target hardware in order to explore the quantized network in MATLAB.

Example: 'ExecutionEnvironment','FPGA'

### Simulation — Validate behavior of network quantized for FPGA environment using simulation

'off' (default) | 'on'

Whether to validate behavior of network quantized for FPGA using simulation, specified as one of these values:

- 'on' — Validate the behavior of the quantized network by simulating the quantized network in MATLAB and comparing the prediction results of the original single-precision network to the simulated prediction results of the quantized network.
- 'off' — Generate code and validate the behavior of the quantized network on the target hardware.

---

**Note** This option is only valid when ExecutionEnvironment is set to 'FPGA'.

---



---

**Note** Alternatively, you can use the `quantize` method to create a simulatable quantized network. The simulatable quantized network enables visibility of the quantized layers, weights, and biases of the network, as well as simulatable quantized inference behavior.

---

Example: 'Simulation','on'

## Object Functions

calibrate Simulate and collect ranges of a deep neural network  
 validate Quantize and validate a deep neural network  
 quantize Quantize 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 squeezeNetMerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
  Connections: [75x2 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=121x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.9198
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.0792
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.3
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1164
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.740
{'fire2-expand1x1_Bias' }	{'fire2-expand1x1' }	"Bias"	-0.06005
{'fire2-expand3x3_Weights' }	{'fire2-expand3x3' }	"Weights"	-0.7439
{'fire2-expand3x3_Bias' }	{'fire2-expand3x3' }	"Bias"	-0.05177
{'fire3-squeeze1x1_Weights' }	{'fire3-squeeze1x1' }	"Weights"	-0.771
{'fire3-squeeze1x1_Bias' }	{'fire3-squeeze1x1' }	"Bias"	-0.1013
{'fire3-expand1x1_Weights' }	{'fire3-expand1x1' }	"Weights"	-0.7203
{'fire3-expand1x1_Bias' }	{'fire3-expand1x1' }	"Bias"	-0.06702
{'fire3-expand3x3_Weights' }	{'fire3-expand3x3' }	"Weights"	-0.6144
{'fire3-expand3x3_Bias' }	{'fire3-expand3x3' }	"Bias"	-0.05361
{'fire4-squeeze1x1_Weights' }	{'fire4-squeeze1x1' }	"Weights"	-0.742
{'fire4-squeeze1x1_Bias' }	{'fire4-squeeze1x1' }	"Bias"	-0.1088
:	:	:	:

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 CPU Target

This example shows how to quantize and validate a neural network for a CPU target. This workflow is similar to other execution environments, but before validating you must establish a `raspi` connection and specify it as target using `dlquantizationOptions`.

First, load your network. This example uses the pretrained network `squeezenet`.

```
load squeezenetmerch
net

net =
  DAGNetwork with properties:

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



Then define your calibration and validation data, `calDS` and `valDS` respectively.

```
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 the `dlquantizer` object and specify a CPU execution environment.

```
dq = dlquantizer(net,'ExecutionEnvironment','CPU')
```

```
dq =
  dlquantizer with properties:
    NetworkObject: [1x1 DAGNetwork]
    ExecutionEnvironment: 'CPU'
```

Calibrate the network.

```
calResults = calibrate(dq,aug_calData,'UseGPU','off')
```

`calResults=122x5 table`

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.9198
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.0792
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.3
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1164
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.740
{'fire2-expand1x1_Bias' }	{'fire2-expand1x1' }	"Bias"	-0.06005
{'fire2-expand3x3_Weights' }	{'fire2-expand3x3' }	"Weights"	-0.7439
{'fire2-expand3x3_Bias' }	{'fire2-expand3x3' }	"Bias"	-0.05177
{'fire3-squeeze1x1_Weights' }	{'fire3-squeeze1x1' }	"Weights"	-0.771
{'fire3-squeeze1x1_Bias' }	{'fire3-squeeze1x1' }	"Bias"	-0.1013
{'fire3-expand1x1_Weights' }	{'fire3-expand1x1' }	"Weights"	-0.7203
{'fire3-expand1x1_Bias' }	{'fire3-expand1x1' }	"Bias"	-0.06702
{'fire3-expand3x3_Weights' }	{'fire3-expand3x3' }	"Weights"	-0.6144
{'fire3-expand3x3_Bias' }	{'fire3-expand3x3' }	"Bias"	-0.05361
{'fire4-squeeze1x1_Weights' }	{'fire4-squeeze1x1' }	"Weights"	-0.742
{'fire4-squeeze1x1_Bias' }	{'fire4-squeeze1x1' }	"Bias"	-0.1088
:			

Use the MATLAB Support Package for Raspberry Pi Hardware function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiName` with the name or address of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiName','username','password')
```

For example,

```
r = raspi('gpcoder-raspberrypi-7','pi','matlab')
r =
  raspi with properties:
    DeviceAddress: 'gpcoder-raspberrypi-7'
    Port: 18734
    BoardName: 'Raspberry Pi 3 Model B+'
    AvailableLEDs: {'led0'}
    AvailableDigitalPins: [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
    AvailableSPIChannels: {}
    AvailableI2CBuses: {}
    AvailableWebcams: {}
    I2CBusSpeed:
    AvailableCANInterfaces: {}

  Supported peripherals
```

Specify `raspi` object as the target for the quantized network.

```
opts = dlquantizationOptions('Target',r)
opts =
  dlquantizationOptions with properties:
    MetricFcn: {}
    Bitstream: ''
    Target: [1x1 raspi]
```

Validate the quantized network with the `validate` function.

```
valResults = validate(dq, aug_valData, opts)
### Starting application: 'codegen\lib\validate_predict_int8\pil\validate_predict_int8.elf'
  To terminate execution: clear validate_predict_int8_pil
### Launching application validate_predict_int8.elf...
### Host application produced the following standard output (stdout) and standard error (stderr)

valResults = struct with fields:
  NumSamples: 20
  MetricResults: [1x1 struct]
  Statistics: []
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}    0.95
  {'Quantized' }        0.95
```

## Quantize Network for FPGA Deployment

Reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. This example shows how to use Deep Learning Toolbox Model Quantization Library and Deep Learning HDL Toolbox to deploy the `int8` network to a target FPGA board.

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning.

### Load Pretrained Network

Load the pretrained LogoNet network and analyze the network architecture.

```
snet = getLogoNetwork;
deepNetworkDesigner(snet);
```

### Load Data

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 for calibration and validation. Expedite calibration and validation 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. The FPGA validation workflow has a maximum limit of one image when validating the quantized network. Reduce the FPGA validation data set to a single image.

```
curDir = pwd;
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_simulation = validationData.subset(1:5);
validationData_FPGA = validationData.subset(1:1);
```

### Generate Calibration Result File for the Network

Create a `dlquantizer` (Deep Learning HDL Toolbox) object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj_simulation = dlquantizer(snet,'ExecutionEnvironment',"FPGA",'Simulation','on');
dlQuantObj_FPGA = dlquantizer(snet,'ExecutionEnvironment',"FPGA");
```

Use the `calibrate` (Deep Learning HDL Toolbox) function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases. The `calibrate` function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj_simulation,calibrationData_reduced)
```

ans=35x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'conv_1' }	"Weights"	-0.048978
{'conv_1_Bias' }	{'conv_1' }	"Bias"	0.99999
{'conv_2_Weights' }	{'conv_2' }	"Weights"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.0006117
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045947
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013999
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045967
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051394
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.0005231
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02955
{'imageinput' }	{'imageinput' }	"Activations"	0
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

calibrate(dlQuantObj\_FPGA,calibrationData\_reduced)

ans=35x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'conv_1' }	"Weights"	-0.048978
{'conv_1_Bias' }	{'conv_1' }	"Bias"	0.99999
{'conv_2_Weights' }	{'conv_2' }	"Weights"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.0006117
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045947
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013999
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045967
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051394
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.0005231
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02955
{'imageinput' }	{'imageinput' }	"Activations"	0
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

### Create Target Object

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 use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, you can also use the JTAG interface.

```
% hTarget = dlhdl.Target('Xilinx', 'Interface', 'JTAG');
```

### Create dlQuantizationOptions Object

Create a `dlquantizationOptions` object. Specify the target bitstream and target board interface. The default metric function is a Top-1 accuracy metric function.

```
options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);
options_simulation = dlquantizationOptions;
```

To use a custom metric function, specify the metric function in the `dlquantizationOptions` object.

```
options_FPGA = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_FPGA)});
options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_simulation)});
```

### Validate Quantized Network

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_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulation);
```

```
Compiling leg: conv_1>>relu_4 ...
Compiling leg: conv_1>>relu_4 ... complete.
Compiling leg: maxpool_4 ...
Compiling leg: maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
Compiling leg: fc_1>>fc_3 ... complete.
```

```
prediction_simulation = struct with fields:
    NumSamples: 5
    MetricResults: [1x1 struct]
    Statistics: []
```

For validation on an FPGA, the `validate` function:

- Programs the FPGA board by using the output of the `compile` method and the programming file
- Downloads the network weights and biases
- Compares the performance of the network before and after quantization

```
prediction_FPGA = dlQuantObj_FPGA.validate(validationData_FPGA,options_FPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
```

```
### Targeting FPGA bitstream zcu102_int8.
```

```
### The network includes the following layers:
```

1	'imageinput'	Image Input	227×227×3 images with 'zerocenter' normalization
2	'conv_1'	Convolution	96 5×5×3 convolutions with stride [1 1] and padding
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	3×3 max pooling with stride [2 2] and padding
5	'conv_2'	Convolution	128 3×3×96 convolutions with stride [1 1] and padding

```

6 'relu_2'      ReLU      ReLU
7 'maxpool_2'  Max Pooling 3x3 max pooling with stride [2 2] and padding
8 'conv_3'     Convolution 384 3x3x128 convolutions with stride [1 1] and
9 'relu_3'     ReLU      ReLU
10 'maxpool_3' Max Pooling 3x3 max pooling with stride [2 2] and padding
11 'conv_4'     Convolution 128 3x3x384 convolutions with stride [2 2] and
12 'relu_4'     ReLU      ReLU
13 'maxpool_4' Max Pooling 3x3 max pooling with stride [2 2] and padding
14 'fc_1'       Fully Connected 2048 fully connected layer
15 'relu_5'     ReLU      ReLU
16 'dropout_1' Dropout   50% dropout
17 'fc_2'       Fully Connected 2048 fully connected layer
18 'relu_6'     ReLU      ReLU
19 'dropout_2' Dropout   50% dropout
20 'fc_3'       Fully Connected 32 fully connected layer
21 'softmax'    Softmax   softmax
22 'classoutput' Classification Output crossentropyex with 'adidas' and 31 other classes

```

```

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
### Compiling layer group: conv_1>>relu_4 ...
### Compiling layer group: conv_1>>relu_4 ... complete.
### Compiling layer group: maxpool_4 ...
### Compiling layer group: maxpool_4 ... complete.
### Compiling layer group: fc_1>>fc_3 ...
### Compiling layer group: fc_1>>fc_3 ... complete.

```

### Allocating external memory buffers:

offset_name	offset_address	allocated_space
"InputDataOffset"	"0x00000000"	"12.0 MB"
"OutputResultOffset"	"0x00c00000"	"4.0 MB"
"SchedulerDataOffset"	"0x01000000"	"4.0 MB"
"SystemBufferOffset"	"0x01400000"	"36.0 MB"
"InstructionDataOffset"	"0x03800000"	"8.0 MB"
"ConvWeightDataOffset"	"0x04000000"	"12.0 MB"
"FCWeightDataOffset"	"0x04c00000"	"12.0 MB"
"EndOffset"	"0x05800000"	"Total: 88.0 MB"

### Network compilation complete.

```

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Deep learning network programming has been skipped as the same network is already loaded on the target
### Finished writing input activations.
### Running single input activation.

```

#### Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	248358	274080	90.62
DSPs	384	2520	15.24
Block RAM	581	912	63.71

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

```

### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software

```

#### Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
	-----	-----	-----	-----
Network	40142478	0.18247	1	40142478
___imageinput_norm	216472	0.00098		
___conv_1	6825671	0.03103		
___maxpool_1	3755088	0.01707		
___conv_2	10440701	0.04746		
___maxpool_2	1447840	0.00658		
___conv_3	9405685	0.04275		
___maxpool_3	1765856	0.00803		
___conv_4	1819636	0.00827		
___maxpool_4	28098	0.00013		
___fc_1	2651288	0.01205		
___fc_2	1696632	0.00771		
___fc_3	89511	0.00041		

\* The clock frequency of the DL processor is: 220MHz

#### Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
-----	-----	-----	-----
LUTs (CLB/ALM)*	168645	274080	61.53
DSPs	800	2520	31.75
Block RAM	453	912	49.67

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive Logic Devices(ALD) in Intel devices

```

### Finished writing input activations.
### Running single input activation.

```

```

prediction_FPGA = struct with fields:
  NumSamples: 1
  MetricResults: [1x1 struct]
  Statistics: [2x7 table]

```

### View Performance of Quantized Neural Network

Display the accuracy of the quantized network.

```
prediction_simulation.MetricResults.Result
```

```

ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}    1
  {'Quantized' }       1

```

```
prediction_FPGA.MetricResults.Result
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}    1
  {'Quantized'      }    1
```

Display the performance of the quantized network in frames per second.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
ans = 19.0828
```

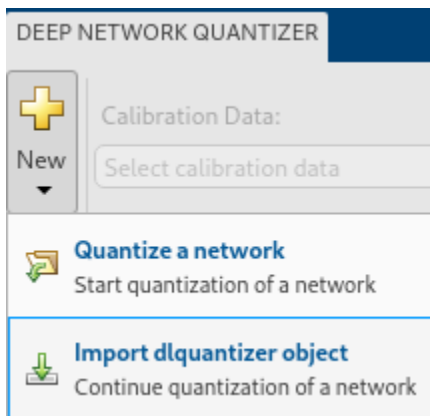
### Import a dlquantizer Object into the Deep Network Quantizer App

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.

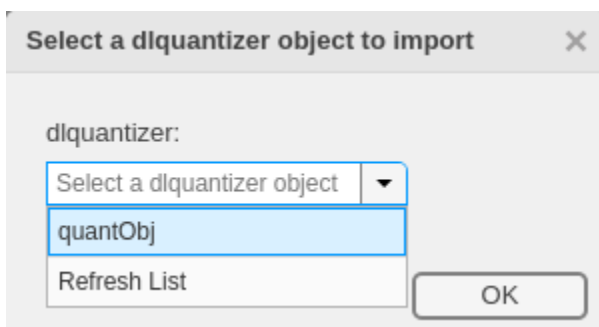
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

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



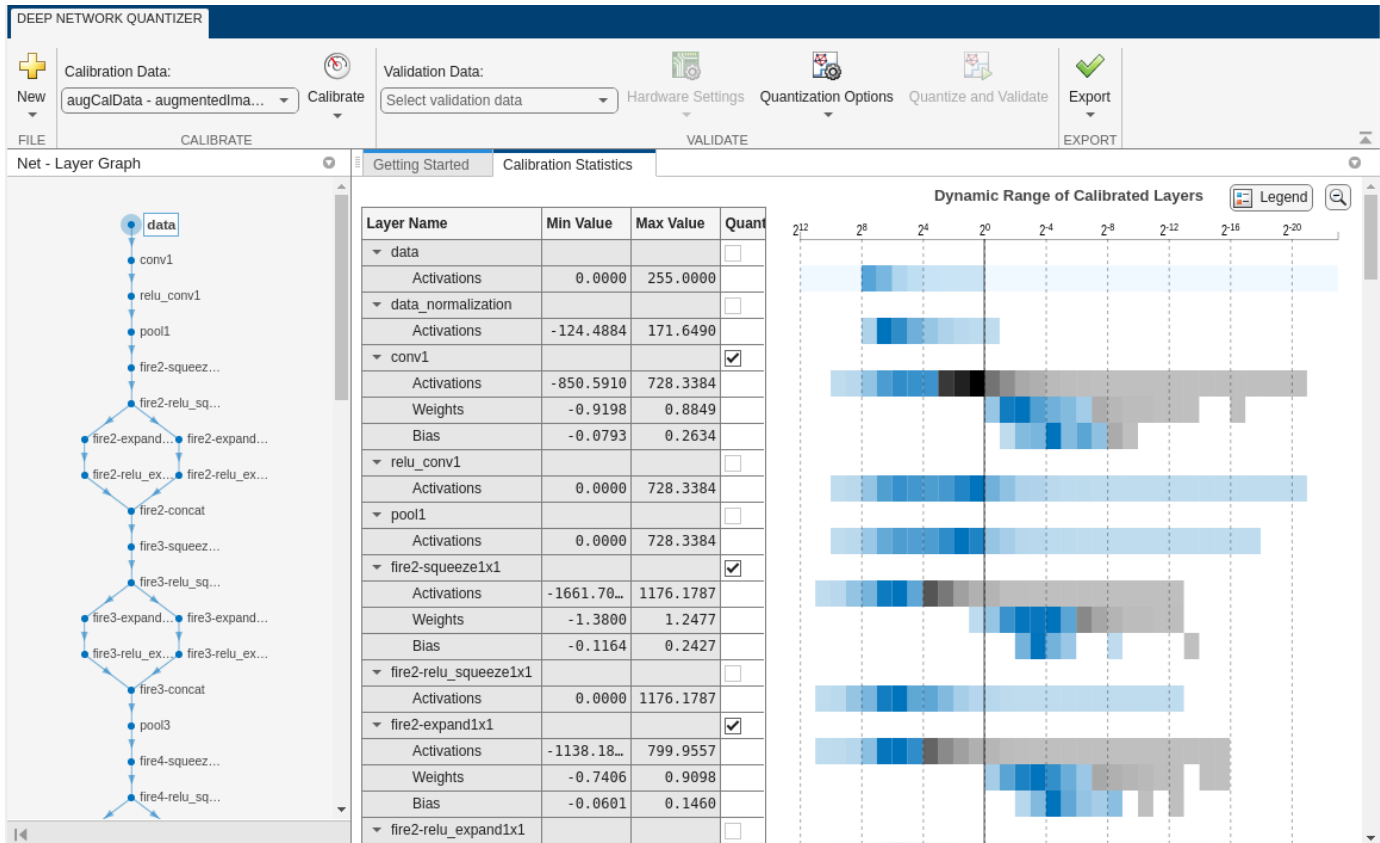
In the dialog, select the `dlquantizer` object to import from the base workspace. For this example, use `quantObj` that you create in the above example `Quantize a Neural Network for GPU Target`.





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”.



## Emulate Target Agnostic Quantized Network

This example shows how to create a target agnostic, simulatable quantized deep neural network in MATLAB.

Target agnostic quantization allows you to see the effect quantization has on your neural network without target hardware or target-specific quantization schemes. Creating a target agnostic quantized network is useful if you:

- Do not have access to your target hardware.
- Want to preview whether or not your network is suitable for quantization.
- Want to find layers that are sensitive to quantization.

Quantized networks emulate quantized behavior for quantization-compatible layers. Network architecture like layers and connections are the same as the original network, but inference behavior uses limited precision types. Once you have quantized your network, you can use the `quantizationDetails` function to retrieve details on what was quantized.

Load the pretrained network. `net` is a SqueezeNet network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch
net
net =
  DAGNetwork with properties:
    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

You can use the `quantizationDetails` function to see that the network is not quantized.

```
qDetailsOriginal = quantizationDetails(net)
qDetailsOriginal = struct with fields:
    IsQuantized: 0
    TargetLibrary: ""
    QuantizedLayerNames: [0x0 string]
    QuantizedLearnables: [0x3 table]
```

Unzip and load the `MerchData` images as an image datastore.

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

Define calibration and validation data to use for quantization. The output size of the images are changed for both calibration and validation data according to network requirements.

```
[calData,valData] = splitEachLabel(imds,0.7,'randomized');
augCalData = augmentedImageDatastore([227 227],calData);
augValData = augmentedImageDatastore([227 227],valData);
```

Create `dlquantizer` object and specify the network to quantize. Set the execution environment to MATLAB. How the network is quantized depends on the execution environment. The MATLAB execution environment is agnostic to the target hardware and allows you to prototype quantized behavior.

```
quantObj = dlquantizer(net,'ExecutionEnvironment','MATLAB');
```

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, augCalData);
```

Use the `quantize` method to quantize the network object and return a simulatable quantized network.

```
qNet = quantize(quantObj)
```

```
qNet =
Quantized DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
  InputNames: {'data'}
  OutputNames: {'new_classoutput'}
```

Use the `quantizationDetails` method to extract quantization details.

You can use the `quantizationDetails` function to see that the network is now quantized.

```
qDetailsQuantized = quantizationDetails(qNet)
```

```
qDetailsQuantized = struct with fields:
    IsQuantized: 1
    TargetLibrary: "none"
QuantizedLayerNames: [26x1 string]
QuantizedLearnables: [52x3 table]
```

Make predictions using the original, single-precision floating-point network, and the quantized INT8 network.

```
predOriginal = classify(net, augValData);           % Predictions for the non-quantized network
predQuantized = classify(qNet, augValData);        % Predictions for the quantized network
```

Compute the relative accuracy of the quantized network as compared to the original network.

```
ccrQuantized = mean(predQuantized == valData.Labels)*100
```

```
ccrQuantized = 100
```

```
ccrOriginal = mean(predOriginal == valData.Labels)*100
```

```
ccrOriginal = 100
```

For this validation data set, the quantized network gives the same predictions as the floating-point network.

## Version History

Introduced in R2020a

### R2023a: dlnetwork support

dlquantizer now supports dlnetwork objects for quantization using the `quantize` function.

**R2022b: dlnetwork support**

`dlquantizer` now supports `dlnetwork` objects for quantization using the `calibrate` and `validate` functions.

**R2022a: Validate the performance of quantized network for CPU target**

You can now use the `dlquantizer` object and the `validate` function to quantize a network and generate code for CPU targets.

**R2022a: Quantize neural networks without a specific target**

Specify `MATLAB` as the `ExecutionEnvironment` to quantize your neural networks without generating code or committing to a specific target for code deployment. This can be useful if you:

- Do not have access to your target hardware.
- Want to inspect your quantized network without generating code.

Your quantized network implements `int8` data instead of `single` data. It keeps the same layers and connections as the original network, and it has the same inference behavior as it would when running on hardware.

Once you have quantized your network, you can use the `quantizationDetails` function to inspect your quantized network. Additionally, you also have the option to deploy the code to a GPU target.

**R2023a: Quantize yolov3objectDetector and yolov4objectDetector using dlquantizer**

`yolov3objectDetector` and `yolov4objectDetector` objects can now be quantized using `dlquantizer`.

**See Also****Apps**

**Deep Network Quantizer**

**Functions**

`calibrate` | `quantize` | `validate` | `dlquantizationOptions` | `quantizationDetails` | `estimateNetworkMetrics`

**Topics**

“Quantization of Deep Neural Networks”

“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”

“Quantize Layers in Object Detectors and Generate CUDA Code”

“Quantize Network for FPGA Deployment” (Deep Learning HDL Toolbox)

“Generate INT8 Code for Deep Learning Network on Raspberry Pi” (MATLAB Coder)

“Parameter Pruning and Quantization of Image Classification Network”

# dlupdate

Update parameters using custom function

## Syntax

```
netUpdated = dlupdate(fun,net)
params = dlupdate(fun,params)
[ ___ ] = dlupdate(fun, ___ A1,...,An)
[ ___ ,X1,...,Xm] = dlupdate(fun, ___ )
```

## Description

`netUpdated = dlupdate(fun,net)` updates the learnable parameters of the `dlnetwork` object `net` 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.

```
d1X = 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,d1X,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,TTrain] = digitTrain4DArrayData;
classes = categories(TTrain);
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], 'Mean', mean(XTrain,4))
    convolution2dLayer(5,20)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers);
```

### Define Model Loss Function

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

### Define Stochastic Gradient Descent Function

Create the helper function `sgdFunction`, listed at the end of this example. The function takes the parameters and the gradients of the loss with respect to the parameters, and returns the updated parameters 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,  $\theta$  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(TTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Specify the learning rate.

```
learnRate = 0.01;
```

### Train Network

Calculate the total number of iterations for the training progress monitor.

```
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

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.

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 Computing Requirements” (Parallel Computing Toolbox).

```
iteration = 0;
epoch = 0;
```

```
while epoch < numEpochs && ~monitor.Stop
    epoch = epoch + 1;
```

```
    % Shuffle data.
```

```
idx = randperm(numel(TTrain));
XTrain = XTrain(:,:,idx);
TTrain = TTrain(idx);

i = 0;
while i < numIterationsPerEpoch && ~monitor.Stop
    i = i + 1;
    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);

    T = zeros(numClasses, miniBatchSize,"single");
    for c = 1:numClasses
        T(c,TTrain(idx)==classes(c)) = 1;
    end

    % Convert mini-batch of data to dlarray.
    X = dlarray(single(X),"SSCB");

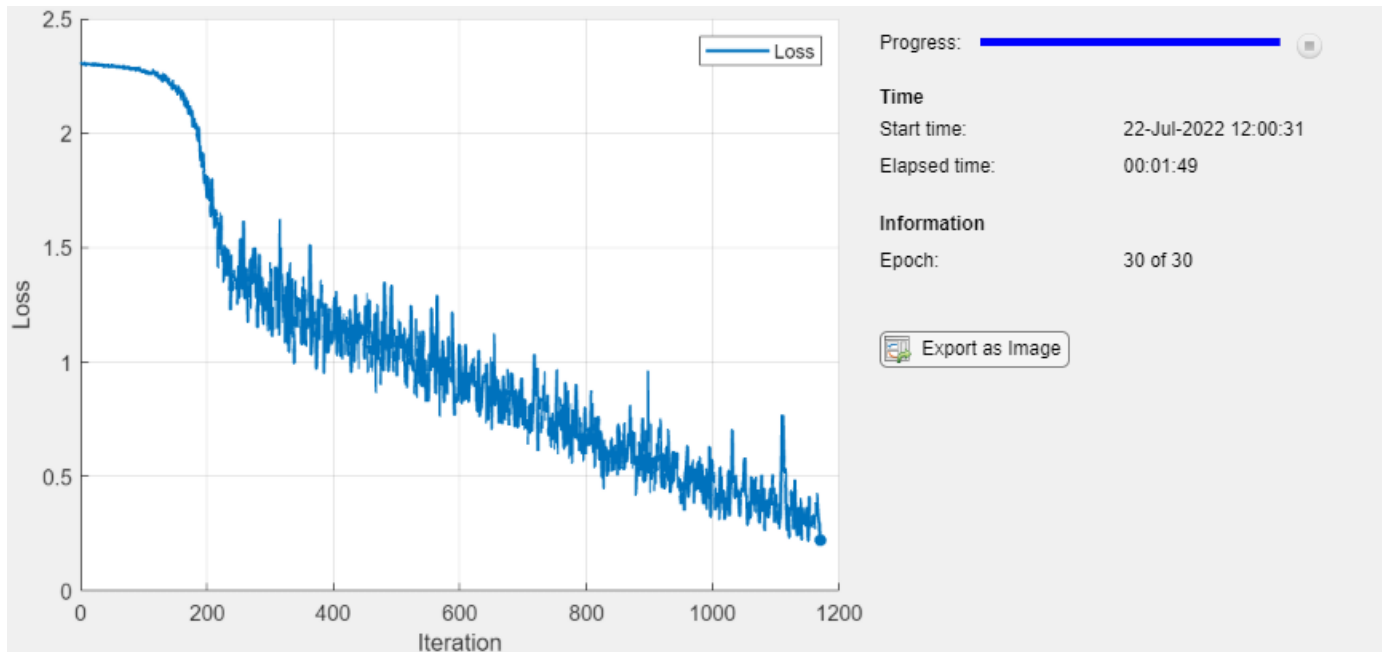
    % If training on a GPU, then convert data to a gpuArray.
    if canUseGPU
        X = gpuArray(X);
    end

    % Evaluate the model loss and gradients using dlfeval and the
    % modelLoss function.
    [loss,gradients] = dlfeval(@modelLoss,net,X,T);

    % Update the network parameters using the SGD algorithm defined in
    % the sgdFunction helper function.
    updateFcn = @(net,gradients) sgdFunction(net,gradients,learnRate);
    net = dlupdate(updateFcn,net,gradients);

    % Update the training progress monitor.
    recordMetrics(monitor,iteration,Loss=loss);
    updateInfo(monitor,Epoch=epoch + " of " + numEpochs);
    monitor.Progress = 100 * iteration/numIterations;
end
end
```





### Test Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

```
[XTest,TTest] = digitTest4DArrayData;
```

Convert the data to a `dlarray` with the dimension format "SSCB" (spatial, spatial, channel, batch). For GPU prediction, also convert the data to a `gpuArray`.

```
XTest = dlarray(XTest,"SSCB");
if canUseGPU
    XTest = gpuArray(XTest);
end
```

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

```
YTest = predict(net,XTest);
[~,idx] = max(extractdata(YTest),[],1);
YTest = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YTest==TTest)
```

```
accuracy = 0.9040
```

### Model Loss Function

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

```
function [loss,gradients] = modelLoss(net,X,T)

Y = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);

end
```

### Stochastic Gradient Descent Function

The helper function `sgdFunction` takes the learnable parameters `parameters`, the gradients of the loss with respect to the learnable parameters, and the learning rate `learnRate`, and returns the updated parameters 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,  $\theta$  is the parameter vector, and  $E(\theta)$  is the loss function.

```
function parameters = sgdFunction(parameters,gradients,learnRate)

parameters = parameters - learnRate .* gradients;

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 `net` or `params`.

### net — Network

dlnetwork object

Network, specified as a `dlnetwork` object.

The function updates the `Learnables` property of the `dlnetwork` object. `net.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 `dIarray`.

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 `dIarray` 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**

`dIarray` | numeric array | cell array | structure | table

Additional input arguments to `fun`, specified as `dIarray` 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 `dIupdate`.

<b>Input</b>	<b>Learnable Parameters</b>	<b>A1, . . . , An</b>
<code>net</code>	Table <code>net.Learnables</code> containing <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dIarray</code> .	Table with the same data type, variables, and ordering as <code>net.Learnables</code> . <code>A1, . . . , An</code> must have a <code>Value</code> variable consisting of cell arrays that contain the additional input arguments for the function <code>fun</code> to apply to each learnable parameter.
<code>params</code>	<code>dIarray</code>	<code>dIarray</code> with the same data type and ordering as <code>params</code> .
	Numeric array	Numeric array with the same data type and ordering as <code>params</code> .
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code> .
	Structure	Structure with the same data types, fields, and ordering as <code>params</code> .

Input	Learnable Parameters	A1, . . . , An
	Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a darray.	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

### netUpdated — Updated network

dlnetwork object

Network, returned as a dlnetwork object.

The function updates the Learnables property of the dlnetwork object.

### params — Updated network learnable parameters

darray | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a darray, 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

darray | 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 darray 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 dupdate.

Input	Learnable parameters	X1, . . . , Xm
net	Table net.Learnables containing Layer, Parameter, and Value variables. The Value variable consists of cell arrays that contain each learnable parameter as a darray.	Table with the same data type, variables, and ordering as net.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.
params	darray	darray with the same data type and ordering as params.
	Numeric array	Numeric array with the same data type and ordering as params.

Input	Learnable parameters	X1, . . . , Xm
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code> .
	Structure	Structure with the same data types, fields, and ordering as <code>params</code> .
	Table with <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>darray</code> .	Table with the same data types, variables, and ordering as <code>params</code> . X1, . . . , Xm has a <code>Value</code> variable consisting of cell arrays that contain the additional output argument of the function <code>fun</code> applied to each learnable parameter.

## Version History

Introduced in R2019b

## 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 `darray` 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` | `darray` | `adamupdate` | `rmspropupdate` | `sgdupdate` | `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”

## dltranspconv

Deep learning transposed convolution

### Syntax

```
Y = dltranspconv(X,weights,bias)
Y = dltranspconv(X,weights,bias,DataFormat=FMT)
Y = dltranspconv( ___ Name=Value)
```

### Description

The transposed convolution operation upsamples feature maps.

The `dltranspconv` function applies the deep learning transposed convolution operation to `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** This function applies the deep learning transposed convolution operation to `darray` data. If you want to apply transposed convolution within a `layerGraph` object or `Layer` array, use one of the following layers:

- `transposedConv1dLayer`
  - `transposedConv2dLayer`
  - `transposedConv3dLayer`
- 

`Y = dltranspconv(X,weights,bias)` computes the deep learning transposed convolution of the input `X` using the filters defined by `weights`, and adds the constant `bias`. The input `X` must be a formatted `darray`. The output `Y` is a formatted `darray` with the same dimension format as `X`.

The function, by default, convolves over up to three dimensions of `X` labeled "S" (spatial). To convolve over dimensions labeled "T" (time), specify `weights` with a "T" dimension using a formatted `darray` object or by using the `WeightsFormat` option.

For unformatted input data, use the `DataFormat` option.

`Y = dltranspconv(X,weights,bias,DataFormat=FMT)` applies the deep learning transposed convolution operation to the unformatted `darray` object `X` with format specified by `FMT`. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`.

`Y = 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

### Perform 2-D Transposed Convolution

Create a formatted `darray` 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);
X = darray(X,"SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
```

```
    28    28     3   128
```

```
dims(X)
```

```
ans =
'SSCB'
```

Initialize the weights and bias for 2-D transposed 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),numFilters,numChannels);
bias = zeros(1,numFilters);
```

Apply 2-D transposed convolution using the `dltranspconv` function.

```
Y = dltranspconv(X,weights,bias);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
```

```
    30    30    64   128
```

```
dims(Y)
```

```
ans =
'SSCB'
```

## Perform Grouped Transposed Convolution

Create a formatted `darray` object containing a batch of 128 28-by-28 images with 16 channels. Specify the format "SSCB" (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 16;
X = rand(inputSize(1),inputSize(2),numChannels,miniBatchSize);
X = darray(X,"SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
```

```
    28    28    16   128
```

```
dims(X)
```

```
ans =
'SSCB'
```

Initialize the weights and bias for 2-D grouped transposed convolution. For the weights, specify two groups of 64 3-by-3 filters. For the bias, specify a vector of zeros.

The number of channels per group is given by the number of channels of the input data divided by the number of groups. The size of the bias vector is the number of filters per group multiplied by the number of groups.

```
filterSize = [3 3];
numFiltersPerGroup = 64;
numGroups = 2;
numChannelsPerGroup = numChannels / numGroups;
```

```
weights = rand(filterSize(1),filterSize(2),numFiltersPerGroup,numChannelsPerGroup,numGroups);
bias = zeros(1,numFiltersPerGroup*numGroups);
```

Apply 2-D grouped transposed convolution using the `dltranspconv` function.

```
Y = dltranspconv(X,weights,bias);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
```

```
    30    30   128   128
```

```
dims(Y)
```

```
ans =
'SSCB'
```



## Input Arguments

### X — Input data

dlarray | numeric array

Input data, specified as a formatted dlarray, an unformatted dlarray, or a numeric array.

If X is an unformatted dlarray or a numeric array, then you must specify the format using the DataFormat option. If X 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 X 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 — Filters

dlarray | numeric array

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 transposed convolution	"S" (spatial) or "T" (time)	Filter size	filterSize-by-numFilters-by-numChannels array, where filterSize is the size of the 1-D filters, numFilters is the number of filters, and numChannels is the number of channels of the input data.	"SCU" (spatial, channel, unspecified)
	"C" (channel)	Number of channels		
	"U" (unspecified)	Number of filters		
1-D grouped transposed convolution	"S" (spatial) or "T" (time)	Filter size	filterSize-by-numFiltersPerGroup-by-numChannelsPerGroup-by-numGroups array, where filterSize is the	"SCUU" (spatial, channel, unspecified, unspecified)
	"C" (channel)	Number of channels per group		
	First "U" (unspecified)	Number of filters per group		

Task	Required Dimensions	Size	Example	
			Weights	Format
	Second "U" (unspecified)	Number of groups	size of the 1-D filters, numFiltersPerGroup is the number of filters per group, numChannelsPerGroup is the number of channels per group of the input data, and numGroups is the number groups.  numChannelsPerGroup must equal the number of the channels of the input data divided by numGroups.	
2-D transposed convolution	First "S" (spatial)	Filter height	filterSize(1)-by-filterSize(2)-by-numFilters-by-numChannels array, where filterSize(1) and filterSize(2) are the height and width of the 2-D filters, respectively, numFilters is the number of filters, and numChannels is the number of channels of the input data.	"SSCU" (spatial, spatial, channel, unspecified)
	Second "S" (spatial) or "T" (time)	Filter width		
	"C" (channel)	Number of channels		
	"U" (unspecified)	Number of filters		
2-D grouped transposed convolution	First "S" (spatial)	Filter height	filterSize(1)-by-filterSize(2)-by-numFiltersPerGroup-by-numChannelsPerGroup-by-numGroups array,	"SSCUU" (spatial, spatial, channel, unspecified, unspecified)
	Second "S" (spatial) or "T" (time)	Filter width		
	"C" (channel)	Number of channels per group		

Task	Required Dimensions	Size	Example	
			Weights	Format
	First "U" (unspecified)	Number of filters per group	<p>where <code>filterSize(1)</code> and <code>filterSize(2)</code> are the height and width of the 2-D filters, respectively, <code>numFiltersPerGroup</code> is the number of filters per group, <code>numChannelsPerGroup</code> is the number of channels per group of the input data, and <code>numGroups</code> is the number of groups.</p> <p><code>numChannelsPerGroup</code> must equal the number of the channels of the input data divided by <code>numGroups</code>.</p>	
	Second "U" (unspecified)	Number of groups		
3-D transposed convolution	First "S" (spatial)	Filter height	<code>filterSize(1)-by-filterSize(2)-by-filterSize(3)-by-numFilters-by-numChannels</code> array, where <code>filterSize(1)</code> , <code>filterSize(2)</code> , and <code>filterSize(3)</code> are the height, width, and depth of the 3-D filters, respectively, <code>numFilters</code> is the number of filters, and <code>numChannels</code> is the number of channels of the input data.	"SSSCU" (spatial, spatial, spatial, channel, unspecified)
	Second "S" (spatial)	Filter width		
	Third "S" (spatial) or "T" (time)	Filter depth		
	"C" (channel)	Number of channels		
	"U" (unspecified)	Number of filters		

---

**Tip** The function, by default, convolves over up to three dimensions of X labeled "S" (spatial). To convolve over dimensions labeled "T" (time), specify `weights` with a "T" dimension using a formatted `darray` object or by using the `WeightsFormat` option.

---

### **bias — Bias constant**

`darray` vector | `darray` scalar | numeric vector | numeric scalar

Bias constant, specified as a formatted or unformatted `darray` vector or `darray` 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 `weights`.

If `bias` is a formatted `darray`, the nonsingleton dimension must be a channel dimension labeled "C".

### **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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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.

The default value of `WeightsFormat` depends on the task:

Task	Default
1-D transposed convolution	"SCU" (spatial, channel, unspecified)
1-D grouped transposed convolution	"SCUU" (spatial, channel, unspecified, unspecified)
2-D transposed convolution	"SSCU" (spatial, spatial, channel, unspecified)
2-D grouped transposed convolution	"SSCUU" (spatial, spatial, channel, unspecified, unspecified)
3-D transposed 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 `X` 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.

---

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 a numeric scalar or numeric vector.

To use the same step size for all convolution dimensions, specify the stride as a scalar. To specify a different value for each convolution dimension, specify the stride as a vector with elements ordered corresponding to the dimensions labels in the data format.

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 specified as a numeric scalar or numeric vector.

To use the dilation factor all convolution dimensions, specify the dilation factor as a scalar. To specify a different value for each convolution dimension, specify the dilation factor as a vector with elements ordered corresponding to the dimensions labels in the data format.

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  $\text{filterSize} + (\text{filterSize}-1) * (\text{dilationFactor}-1)$ .

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 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  $\text{inputSize} \times \text{stride}$ , where  $\text{inputSize}$  is the size of the input along the convolution dimension.
- Numeric scalar — The same cropping value is applied to both ends of the convolution dimensions.
- Numeric vector — A different cropping value is applied along each convolution dimension. Use a vector of size  $d$ , where  $d$  is the number of convolution 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 convolution dimension.
- Numeric matrix — A different cropping value is applied to the start and end of each convolution dimension. Use a matrix of size 2-by- $d$ , where  $d$  is the number of convolution dimensions of the input data. The element  $(1, d)$  specifies the cropping applied to the start of convolution dimension  $d$ . The element  $(2, d)$  specifies the cropping applied to the end of convolution dimension  $d$ . For example, in 2-D the format is `[top, left; bottom, right]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### Y — Feature map

`dlarray`

Feature map, returned as a `dlarray`. The output  $Y$  has the same underlying data type as the input  $X$ .

If the input data  $X$  is a formatted `dlarray`, then  $Y$  has the same format as  $X$ . If the input data is not a formatted `dlarray`, then  $Y$  is an unformatted `dlarray` or numeric array with the same dimension order as the input data.

The size of the "C" (channel) dimension of  $Y$  depends on the size of the `weights` input. The size of the "C" (channel) 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" (channel) dimension and the second "U" (unspecified) dimension.

## Algorithms

### Transposed Convolution

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution

derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

## Version History

Introduced in R2019b

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU.
  - `X`
  - `weights`
  - `bias`

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

## See Also

`darray` | `avgpool` | `dlconv` | `maxunpool` | `maxpool` | `dlgradient` | `dlfeval`

### Topics

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Network Using Model Function”

“List of Functions with `darray` Support”

## disconnectLayers

**Package:** `nnet.cnn`

Disconnect layers in layer graph or network

### Syntax

```
lgraphUpdated = disconnectLayers(lgraph,s,d)
netUpdated = disconnectLayers(net,s,d)
```

### Description

`lgraphUpdated = disconnectLayers(lgraph,s,d)` disconnects the source layer `s` from the destination layer `d` in the layer graph `lgraph`. The updated layer graph, `lgraphUpdated`, contains the same layers as `lgraph`, but excludes the connection between `s` and `d`.

`netUpdated = disconnectLayers(net,s,d)` disconnects the source layer `s` from the destination layer `d` in the `dlnetwork` object `net`. The updated network, `netUpdated`, contains the same layers as `net`, but excludes the connection between `s` and `d`.

### Examples

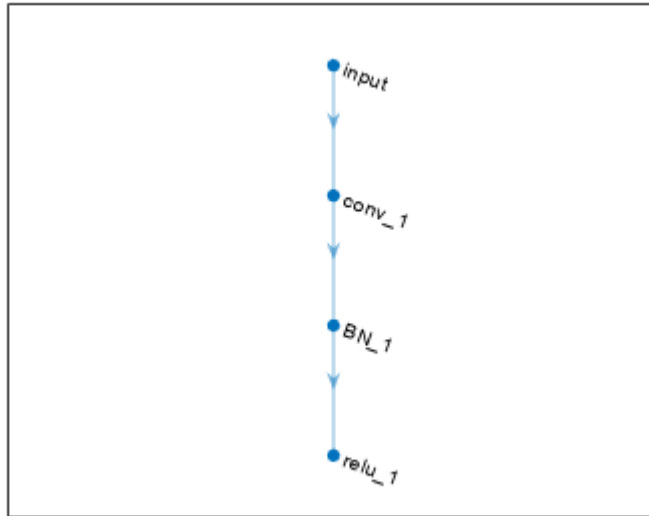
#### Disconnect Layers in Layer Graph

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')];

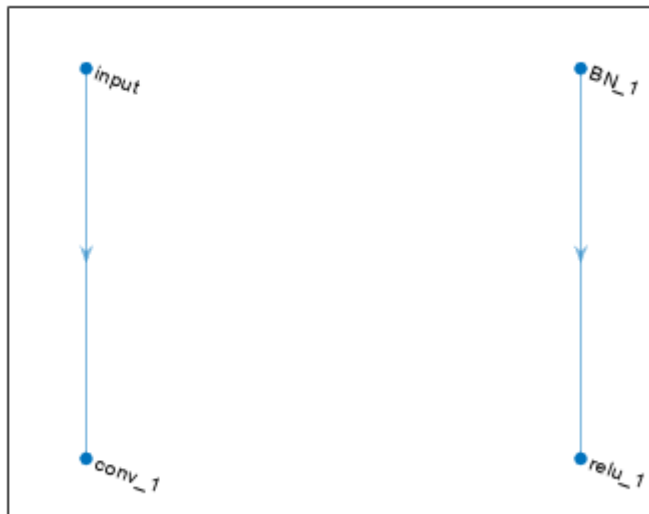
lgraph = layerGraph(layers);
figure
plot(lgraph)
```





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`.

**net — Neural network**

`dlnetwork` object

Neural network, specified as a `dlnetwork` object.

**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

**lgraphUpdated — Updated layer graph**

`LayerGraph` object

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

**netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as an uninitialized `dlnetwork` object.

To initialize the learnable parameters of a `dlnetwork` object, use the `initialize` function.

The `disconnectLayers` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

**Introduced in R2017b**

## See Also

layerGraph | addLayers | removeLayers | replaceLayer | connectLayers | plot | assembleNetwork | dlnetwork

## Topics

“Train Deep Learning Network to Classify New Images”

## distributionScores

**Package:** aivnv.ood

Distribution confidence scores

### Syntax

```
scores = distributionScores(discriminator,X)
```

### Description

`scores = distributionScores(discriminator,X)` returns the distribution confidence score for each observation in `X` using the method you specify in the `Method` property of `discriminator`. You can use the scores to separate data into in-distribution (ID) and out-of-distribution (OOD) data sets. For example, you can classify any observation with distribution confidence score less than or equal to the `Threshold` property of `discriminator` as OOD. For more information about how the software computes the distribution confidence scores, see “Distribution Confidence Scores” on page 1-646.

### Examples

#### Find Distribution Scores

Load a pretrained classification network.

```
load('digitsClassificationMLPNet.mat');
```

Load ID data. Convert the data to a `darray` object.

```
XID = digitTrain4DArrayData;  
XID = darray(XID,"SSCB");
```

Modify the ID training data to create an OOD set.

```
XOOD = XID.*0.3 + 0.1;
```

Create a discriminator using the `networkDistributionDiscriminator` function.

```
method = "baseline";  
discriminator = networkDistributionDiscriminator(net,XID,XOOD,method)
```

```
discriminator =  
  BaselineDistributionDiscriminator with properties:  
    Method: "baseline"  
   Network: [1x1 dlnetwork]  
  Threshold: 0.9743
```

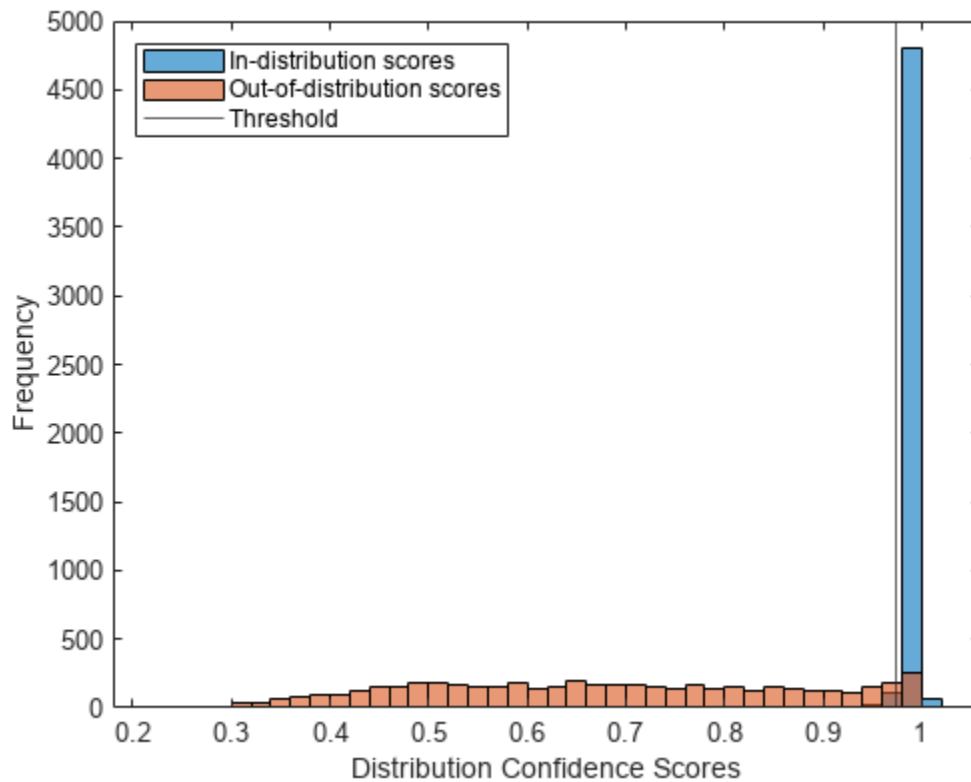
The discriminator object contains a threshold for separating the ID and OOD confidence scores.

Use the `distributionScores` function to find the distribution scores for the ID and OOD data. You can use the distribution confidence scores to separate the data into ID and OOD. The algorithm the software uses to compute the scores is set when you create the discriminator. In this example, the software computes the scores using the baseline method.

```
scoresID = distributionScores(discriminator,XID);
scoresOOD = distributionScores(discriminator,XOOD);
```

Plot the distribution confidence scores for the ID and OOD data. Add the threshold separating the ID and OOD confidence scores.

```
figure
histogram(scoresID,BinWidth=0.02)
hold on
histogram(scoresOOD,BinWidth=0.02)
xline(discriminator.Threshold)
legend(["In-distribution scores","Out-of-distribution scores","Threshold"],Location="northwest")
xlabel("Distribution Confidence Scores")
ylabel("Frequency")
hold off
```



### Assess Performance Using ROC Curve

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data and convert the data to a `dlarray` object.

```
XID = digitTrain4DArrayData;  
XID = dlarray(XID, "SSCB");
```

Modify the ID training data to create an OOD set.

```
XOOD = XID.*0.3 + 0.1;
```

Create a discriminator.

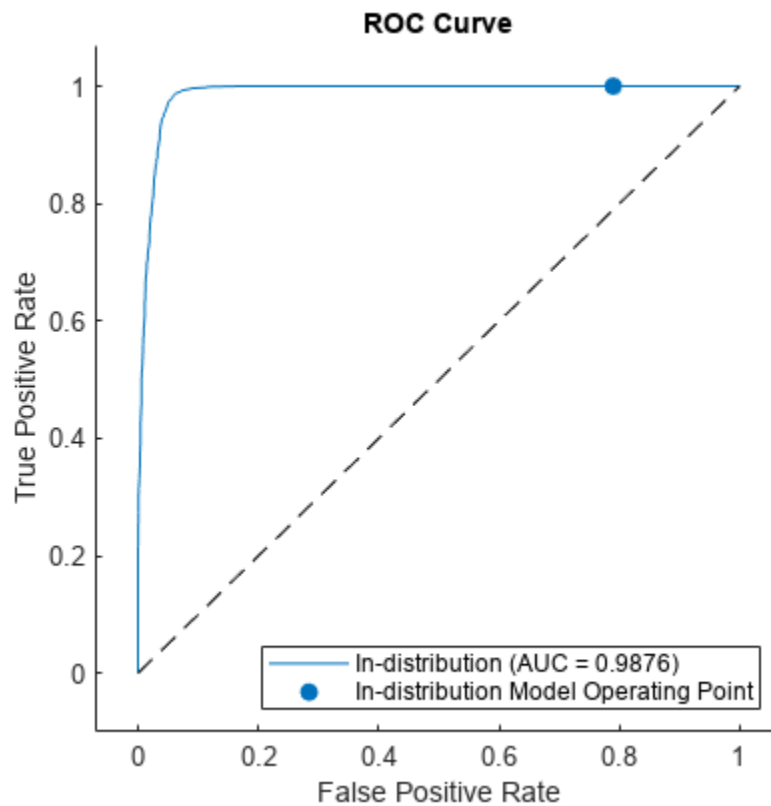
```
method = "baseline";  
discriminator = networkDistributionDiscriminator(net,XID,XOOD,method);
```

Use the `distributionScores` function to find the distribution scores for the ID and OOD data. You can use the distribution scores to separate the data into ID and OOD.

```
scoresID = distributionScores(discriminator,XID);  
scoresOOD = distributionScores(discriminator,XOOD);
```

Use `rocmetrics` to plot a ROC curve to show how well the model performs at separating the data into ID and OOD.

```
labels = [  
    repelem("In-distribution",numel(scoresID)), ...  
    repelem("Out-of-distribution",numel(scoresOOD))];  
scores = [scoresID',scoresOOD'];  
  
rocObj = rocmetrics(labels,scores,"In-distribution");  
figure  
plot(rocObj)
```



## Input Arguments

### **discriminator** — Distribution discriminator

BaselineDistributionDiscriminator object | ODINDistributionDiscriminator object | EnergyDistributionDiscriminator object | HBOSDistributionDiscriminator object

Distribution discriminator, specified as a BaselineDistributionDiscriminator, ODINDistributionDiscriminator, EnergyDistributionDiscriminator, or HBOSDistributionDiscriminator object. To create this object, use the networkDistributionDiscriminator function.

### **X** — Input data

formatted darray

Input data, specified as a formatted darray. For more information about darray formats, see the fmt input argument of darray.

## More About

### **In-Distribution and Out-of-Distribution Data**

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-646.

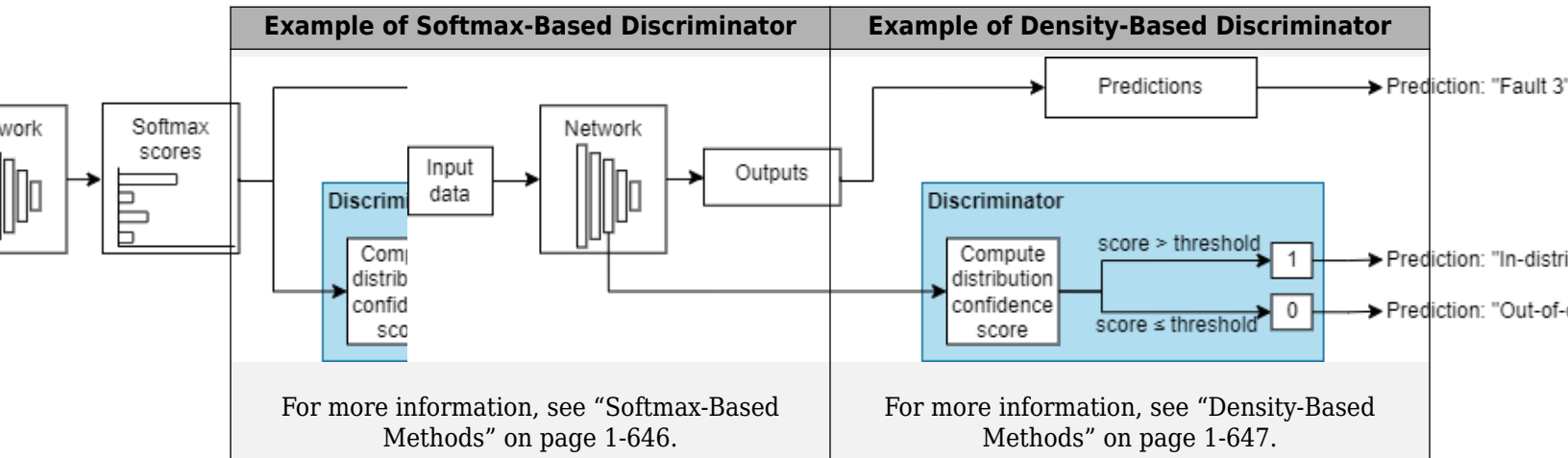
### OOD Data Detection

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-646.

These images show how a discriminator acts as an additional output of a trained neural network.

### Example Data Discriminators



### Distribution Confidence Scores

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.

#### Softmax-Based Methods

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.



Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}},$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is `-Inf`.

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. “A Baseline for Detecting Out-of-Distribution Examples in Image Captioning.” In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.
- [2] Shiyu Liang, Yixuan Li, and R. Srikant, “Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks” arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, “Energy-based Out-of-distribution Detection” arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.

- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## 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 its inputs meet either or both of these conditions:
  - Any values of the network learnable parameters are `darray` objects with underlying data of type `gpuArray`. To see the learnables for a network `net`, call `net.Learnables.Value`.
  - Any of the data is a `darray` object with underlying data of type `gpuArray`.

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

## See Also

`networkDistributionDiscriminator` | `isInNetworkDistribution` | `rocmetrics`

### Topics

"Out-of-Distribution Detection for Deep Neural Networks"

"Out-of-Distribution Data Discriminator for YOLO v4 Object Detector"

"Verify Robustness of Deep Learning Neural Network"

"Compare Deep Learning Models Using ROC Curves"

# 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  $\text{rand}(\text{size}(X)) < \text{Probability}$ , where  $X$  is the layer input and then scales the remaining elements by  $1/(1-\text{Probability})$ . 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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	Dropout	50% dropout
5	''	Fully Connected	10 fully connected layer
6	''	Softmax	softmax
7	''	Classification Output	crossentropyex

## Algorithms

### 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  $\text{rand}(\text{size}(X)) < \text{Probability}$ , where  $X$  is the layer input and then scales the remaining elements by  $1/(1-\text{Probability})$ . 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.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel

- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

DropoutLayer objects apply an element-wise operation and supports input data of any format. The layer and does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

Introduced in R2016a

## 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 Neural Network for Classification"
- "Train Convolutional Neural Network for Regression"
- "Deep Learning in MATLAB"
- "Specify Layers of Convolutional Neural Network"
- "List of Deep Learning Layers"

## 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 =
```

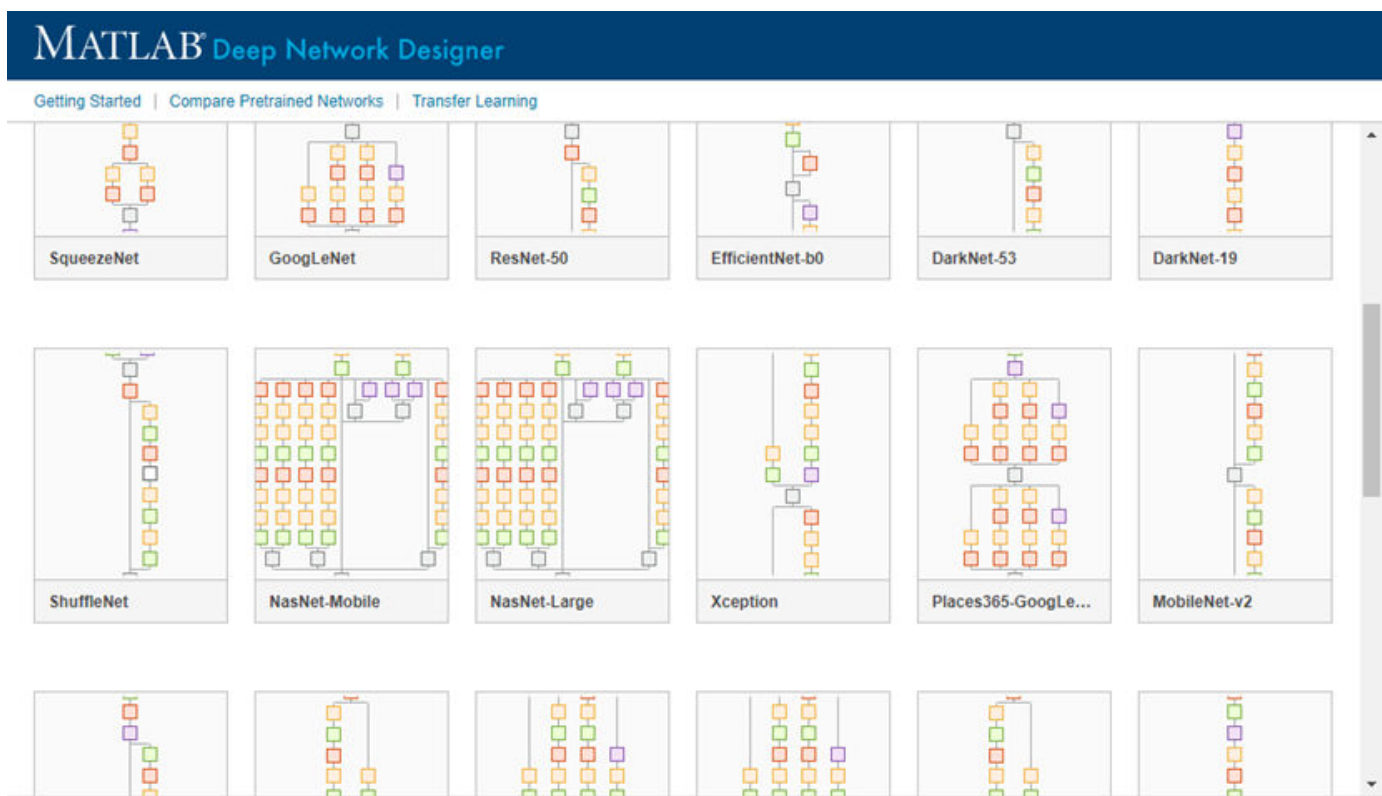
```
DAGNetwork with properties:
```

```
    Layers: [290x1 nnet.cnn.layer.Layer]
 Connections: [363x2 table]
 InputNames: {'ImageInput'}
 OutputNames: {'classification'}
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(efficientnetb0)
```

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



If you need to download a neural network, pause on the desired neural 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.

## **Version History**

**Introduced in R2020b**

## **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”

## 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  $\alpha$  is 1. Specify a value of  $\alpha$  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  $\alpha$ , 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  $\alpha$ .

#### 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 the name ' '.

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	''	2-D Convolution	16 3x3 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	Batch Normalization	Batch normalization
4	''	ELU	ELU with Alpha 1
5	''	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6	''	2-D Convolution	32 3x3 convolutions with stride [1 1] and padding [0 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

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

ELULayer objects apply an element-wise operation and supports input data of any format. The layer and does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

Introduced in R2019a

## 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 Neural 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"

## embed

Embed discrete data

### Syntax

```
Y = embed(X,weights)
Y = embed(X,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 `darray` data. If you want to apply the `embed` operation within a `LayerGraph` object or `Layer` array, use a `wordEmbeddingLayer` object.

---

`Y = embed(X,weights)` returns the embedding vectors in `weights` corresponding to the numeric indices in the formatted `darray` object `X`.

`Y = embed(X,weights,'DataFormat',FMT)` also specifies dimension format `FMT` when `X` is not a formatted `darray` object. The output `Y` is an unformatted `darray` with the same dimension order as `X`.

### 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 `darray` input, the `embed` function expands into a singleton 'C' (channel) dimension with size 1. Create a formatted `darray` object containing the data. To specify that the rows correspond to observations, specify the format 'BC' (batch, channel).

```
d1X = darray(X, 'BC')
```

```
d1X =
  1(C) x 5(B) darray
      2   1   2   1   1
```

Embed the numeric indices using the `embed` function. The `embed` function expands into the 'C' dimension.

```
d1Y = embed(d1X, weights)
```

```
d1Y =
  10(C) x 5(B) darray
      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=2x1 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 = 2x11
```

```
    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 `darray`. Because the rows and columns of `X` correspond to observations and time steps, respectively, specify the format `'BT'`.

```
d1X = darray(X,'BT')
```

```
d1X =
    2(B) x 11(T) darray
```

```
    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.

```
d1Y = embed(d1X,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 `d1Y(:,n,t)` corresponds to the embedding vector of time-step `t` of observation `n`.

## Input Arguments

### X — Input data

`darray` object | numeric array

Input data, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. The elements of `X` must be nonnegative integers or `NaN`.

The function returns the embedding vectors in `weights` corresponding to the numeric indices in `X`. If any values in `X` are zero, `NaN`, or greater than the vocabulary size, then the function returns the out-of-vocabulary vector for that element.

When `X` is not a formatted `darray` object, you must specify the dimension label format using the `'DataFormat'` option. Also, if `X` is a numeric array, then `weights` must be a `darray` 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

`darray` object | numeric array

Embedding weights, specified as a formatted `darray`, an unformatted `darray`, 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 `darray` 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 `darray` 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 `X`. If any values in `X` 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 `darray`.

Example: 'DataFormat', 'SSCB'

Data Types: `char` | `string`

## Output Arguments

### Y — Embedding vectors

`darray`

Embedding vectors, returned as a `darray` object. The output Y has the same underlying data type as the input X.

The function returns the embedding vectors in `weights` corresponding to the numeric indices in X. If any values in X 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 Y depend on the input data:

- If X is a formatted `darray` 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 X is a formatted `darray` 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 X is not a formatted `darray` 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 X is not a formatted `darray` 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.

## Version History

Introduced in R2020b

## 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.
  - `X`
  - `weights`

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

### See Also

`dlarray` | `dlfeval` | `dlgradient` | `lstm` | `attention`

### 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”

# EnergyDistributionDiscriminator

Energy distribution discriminator

## Description

The `EnergyDistributionDiscriminator` object is a distribution discriminator that uses the `energy` method to compute distribution confidence scores. The object contains a threshold that you can use to separate observations into in-distribution (ID) and out-of-distribution (OOD) data sets.

## Creation

Create an `EnergyDistributionDiscriminator` object using the `networkDistributionDiscriminator` function and setting the `method` input argument to `"energy"`.

---

**Note** The object type depends on the method you specify when you use the `networkDistributionDiscriminator` function. The method determines how the software computes the distribution confidence scores. You can set the `method` argument to either `"baseline"`, `"odin"`, `"energy"`, or `"hbos"`. For more information, see “Distribution Confidence Scores” on page 1-670.

---

## Properties

### Method — Method for computing distribution confidence scores

`"energy"`

This property is read-only.

Method for computing the distribution confidence scores, returned as `"energy"`. For more information, see “Distribution Confidence Scores” on page 1-670.

### Network — Deep learning network

`dlnetwork` object

This property is read-only.

Deep learning network, returned as a `dlnetwork` object.

### Temperature — Temperature scaling

positive scalar

This property is read-only.

Temperature scaling, returned as a positive scalar. The temperature controls the scaling of the softmax scores when the software computes the distribution confidence scores. For more information, see “Distribution Confidence Scores” on page 1-670.

Example: 1000

**Threshold — Distribution threshold**

positive scalar

This property is read-only.

Distribution threshold, returned as a positive scalar. The threshold separates the ID and OOD data using the distribution confidence scores.

Example: 1.2

**Object Functions**

`isInNetworkDistribution` Determine whether data is within the distribution of the network  
`distributionScores` Distribution confidence scores

**Examples****Create Distribution Discriminator Using Energy Method**

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data. Convert the data to a `dlarray` object.

```
XID = digitTrain4DArrayData;  
XID = dlarray(XID,"SSCB");
```

Modify the ID training data to create an OOD set.

```
X00D = XID.*0.3 + 0.1;
```

Create a discriminator using the energy method. Set the temperature scaling to 0.1.

```
method = "energy";  
discriminator = networkDistributionDiscriminator(net,XID,X00D,method, ...  
    Temperature=0.1)
```

```
discriminator =  
    EnergyDistributionDiscriminator with properties:
```

```
        Method: "energy"  
        Network: [1x1 dlnetwork]  
    Temperature: 0.1000  
        Threshold: 8.7246
```

Assess the performance of the discriminator on the OOD data.

```
tf = isInNetworkDistribution(discriminator,X00D);  
accuracy = sum(tf == 0)/numel(tf)
```

```
accuracy = 0.9066
```

## More About

### In-Distribution and Out-of-Distribution Data

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-670.

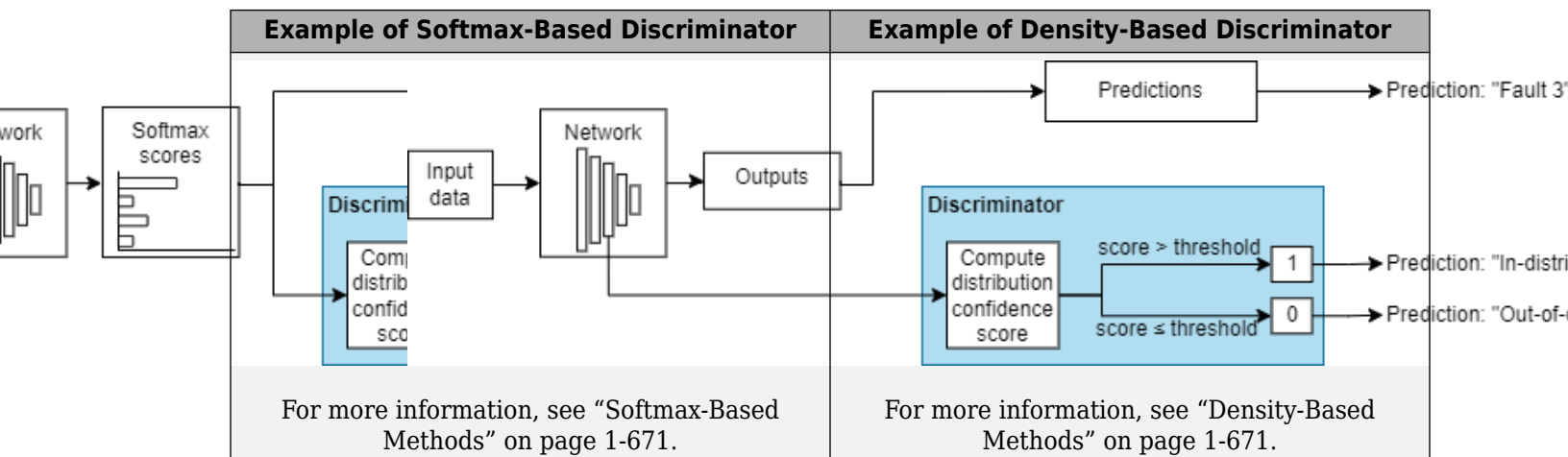
### OOD Data Detection

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-670.

These images show how a discriminator acts as an additional output of a trained neural network.

### Example Data Discriminators



### Distribution Confidence Scores

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.



### Softmax-Based Methods

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}},$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is `-Inf`.

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. “A Baseline for Detecting Out-of-Distribution Examples in Image Captioning.” In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.

- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## See Also

`networkDistributionDiscriminator` | `isInNetworkDistribution` | `distributionScores`

## Topics

"Out-of-Distribution Detection for Deep Neural Networks"

"Verify Robustness of Deep Learning Neural Network"

## equalizeLayers

Equalize layer parameters of deep neural network

### Syntax

```
equalizedNet = equalizeLayers(net)
```

### Description

`equalizedNet = equalizeLayers(net)` equalizes layer parameters in the deep neural network `net` and returns an updated network of the same type.

This function requires Deep Learning Toolbox Model Quantization Library.

### Examples

#### Use equalizeLayers to Improve Performance of Quantized Network

This example shows how to improve the performance of a quantized deep learning model by equalizing layer parameters in the network. Use the `equalizeLayers` function to adjust the compatible network parameters of compute layers in order to make the layers more suitable for quantization.

The network in this example has a MobileNet-v2 backbone. Transfer learning was used to train the network to classify images in the CIFAR-10 dataset.

#### Load Pretrained Network and Data

Download the CIFAR-10 data set [1]. The data set contains 60,000 images. Each image is 32-by-32 in size and has three color channels (RGB). The size of the data set is 175 MB. Depending on your internet connection, the download process can take some time.

```
datadir = tempdir;  
downloadCIFARData(datadir);
```

Load the trained network for CIFAR-10 using a MobileNet-v2 backbone. The batch normalization layers have been folded into the convolution and grouped convolution layers.

```
net = load("CIFARMobilenet.mat").trainedNet;
```

Load the CIFAR-10 training and test images as 4-D arrays. The training set contains 50,000 images and the test set contains 10,000 images. Create an `augmentedImageDatastore` object to use for network training and validation.

```
[XTrain,TTrain,XTest,TTest] = loadCIFARData(datadir);  
inputSize = net.Layers(1).InputSize;  
augimdsTrain = augmentedImageDatastore(inputSize,XTrain,TTrain);  
augimdsTest = augmentedImageDatastore(inputSize,XTest,TTest);  
classes = categories(TTest);
```

Calculate the accuracy of the trained network on the test data.

```
YTest = classify(net, augimdsTest);
accuracyOfnet = mean(YTest == TTest)*100;
display("Accuracy of original network on validation data: " + accuracyOfnet + "%");

"Accuracy of original network on validation data: 85.92%"
```

You can use the `deepNetworkDesigner` function to open the Deep Network Designer app and display the network diagram and the total number of learnable parameters in the network.

```
deepNetworkDesigner(net)
```

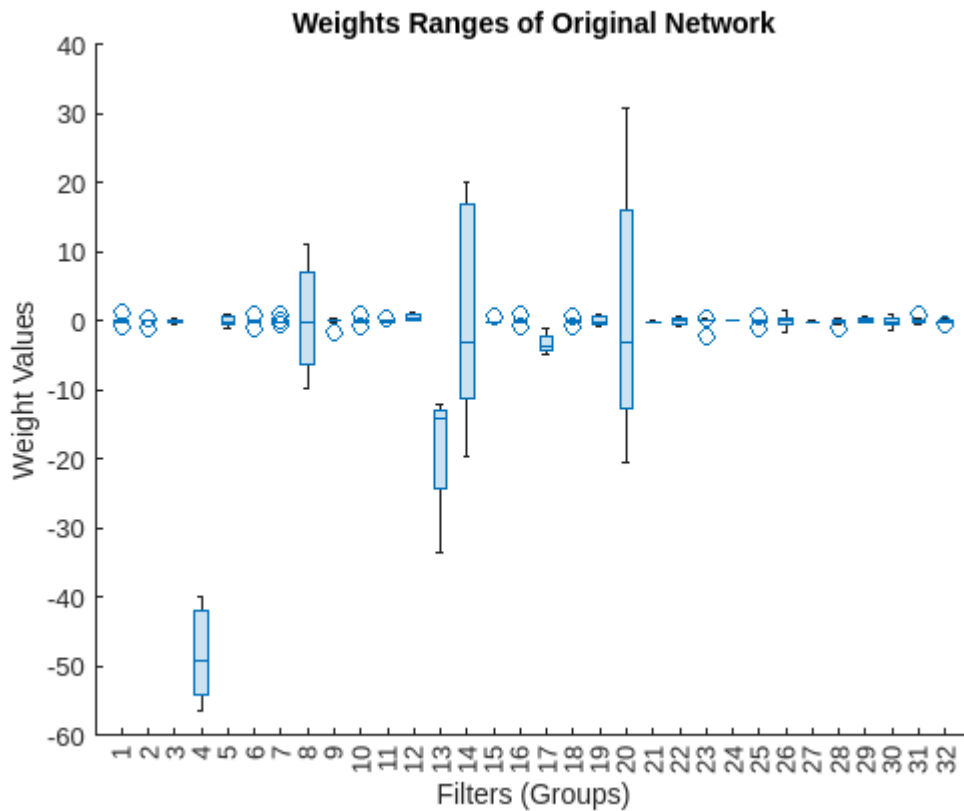
When you quantize then network to `int8` format, it produces an accuracy of 16.8%. This is nearly a 70% degradation compared to the accuracy of the floating-point version of the network.

### Analyze Network

One possible cause of major performance degradation during quantization is the significant range variation in the weight tensor of the convolution and grouped convolution layers. To investigate this variation, you can view the layer weights at the filter level for the first depth-wise separable convolution layer.

For a more detailed view, plot the weight tensor as a boxplot across filters.

```
layerWeights = net.Layers(4).Weights;
layerWeights = reshape(layerWeights,[9 32]);
boxchart(layerWeights)
xlabel("Filters (Groups)")
ylabel("Weight Values")
title("Weights Ranges of Original Network")
```



Due to the significant range variation across the filters, many filter groups are not representable after quantization to `int8` format. Nonrepresentable filter groups cause severe network degradation when the error produced by nonrepresentable values propagates through the network.

To view the histogram bins across different layers, import a calibrated `dlquantizer` object into the Deep Network Quantizer app. The app displays histograms of the dynamic ranges of the parameters as well as the minimum and maximum values across each network layer parameter tensor. “Quantization of Deep Neural Networks” explains this process in detail.

### Use `equalizeLayers` to Adjust Network Parameters

You can use the `equalizeLayers` function to improve the network quantization behavior. The `equalizeLayers` function can be used on networks that have linear activation functions that exhibit a positive scale equivariance property [1]. The `equalizeLayers` function tries to find compatible layers and modifies the weights to make the network more suitable for quantization. The function modifies the layer weights in such a way that the overall numerical output of the network is unchanged.

Equalize the layers in the network.

```
equalizedNet = equalizeLayers(net);
```

### Quantize Equalized Network

After you equalize the layers, quantize the network. Compare the accuracy of the quantized network with and without equalized layers.

Create a `dlquantizer` object for the equalized network.

```
dlQuantObj = dlquantizer(equalizedNet);
```

Define calibration data to use for quantization.

```
numberOfCalibrationImages = 500;
augimdsCalibration = subset(augimdsTrain,1:numberOfCalibrationImages);
```

Calibrate the `dlquantizer` object.

```
calibStats = calibrate(dlQuantObj,augimdsCalibration)
```

Attempt to calibrate with host GPU errored with the message:

Unable to find a supported GPU device. For more information on GPU support, see GPU Support by Reverting to use host CPU.

*calibStats=158x5 table*

Optimized Layer Name	Network Layer Name	Learnables / Activation
{'Conv1_Weights' }	{'Conv1' }	"Weights"
{'Conv1_Bias' }	{'Conv1' }	"Bias"
{'expanded_conv_depthwise_Weights' }	{'expanded_conv_depthwise' }	"Weights"
{'expanded_conv_depthwise_Bias' }	{'expanded_conv_depthwise' }	"Bias"
{'expanded_conv_project_Weights' }	{'expanded_conv_project' }	"Weights"
{'expanded_conv_project_Bias' }	{'expanded_conv_project' }	"Bias"
{'block_1_expand_Weights' }	{'block_1_expand' }	"Weights"
{'block_1_expand_Bias' }	{'block_1_expand' }	"Bias"
{'block_1_depthwise_Weights' }	{'block_1_depthwise' }	"Weights"
{'block_1_depthwise_Bias' }	{'block_1_depthwise' }	"Bias"
{'block_1_project_Weights' }	{'block_1_project' }	"Weights"
{'block_1_project_Bias' }	{'block_1_project' }	"Bias"
{'block_2_expand_Weights' }	{'block_2_expand' }	"Weights"
{'block_2_expand_Bias' }	{'block_2_expand' }	"Bias"
{'block_2_depthwise_Weights' }	{'block_2_depthwise' }	"Weights"
{'block_2_depthwise_Bias' }	{'block_2_depthwise' }	"Bias"
:		

Use the `quantize` function to create a simulatable quantized network from the `dlquantizer` object.

```
qNetEqualized = quantize(dlQuantObj);
```

Evaluate the accuracy of the quantized network. The table compares the accuracy of the original floating-point network, the quantized network without equalized layers, and the quantized network with equalized layers.

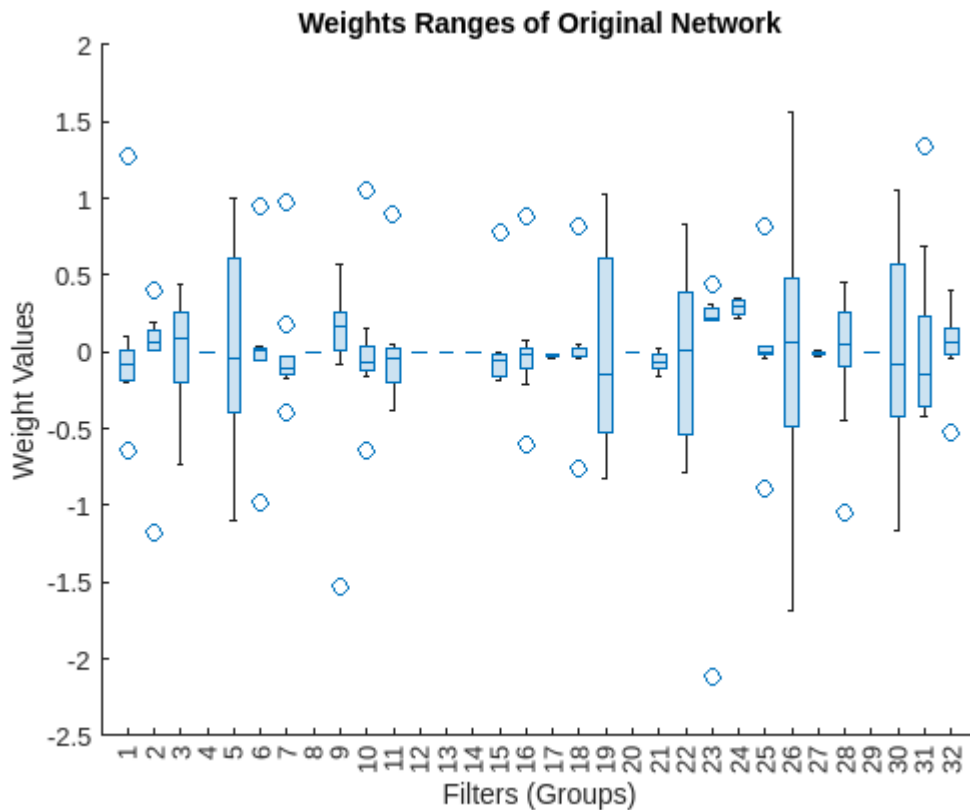
```
YTest = classify(qNetEqualized,augimdsTest);
accuracyOfQuantizedNet = mean(YTest == TTest)*100;
createComparisonTable(accuracyOfnet,accuracyOfQuantizedNet)
```

*ans=1x3 table*

	Original Network	Quantized Original Network	Quantized Equalized
Test Data Accuracy (%)	85.92	16.81	76.44

Plot the new weight tensor as a boxplot across the filters. The weight ranges for the equalized network are better distributed and the quantized equalized network performance is significantly better than the quantized network without layer equalization.

```
newLayerWeights = equalizedNet.Layers(4).Weights;
newLayerWeights = reshape(newLayerWeights,[9 32]);
boxchart(newLayerWeights)
xlabel("Filters (Groups)")
ylabel("Weight Values")
title("Weights Ranges of Original Network")
```



### Helper Functions

The `createComparisonTable` helper function prints a table comparing the accuracy of the original floating-point network, the quantized network without equalized layers, and the quantized network with equalized layers.

```
function comparisonTable = createComparisonTable(originalAccuracy,accuracyQuantizedEqualizedNet)
% Create the summary table
comparisonTable = table(originalAccuracy,16.81,accuracyQuantizedEqualizedNet, ...
    VariableNames=["Original Network", "Quantized Original Network", "Quantized Equalized Network"])
```



```
    RowNames="Test Data Accuracy (%)";  
end
```

## Input Arguments

### **net** — Pretrained neural network

DAGNetwork object | SeriesNetwork object | dlnetwork object | yolov20objectDetector object  
| ssd0objectDetector object

Pretrained neural network, specified as a DAGNetwork, SeriesNetwork, dlnetwork, yolov20objectDetector, or ssd0objectDetector object.

## Output Arguments

### **equalizedNet** — Equalized network

DAGNetwork object | SeriesNetwork object | dlnetwork object | yolov20objectDetector object  
| ssd0objectDetector object

Equalized network, returned as a DAGNetwork, SeriesNetwork, dlnetwork, yolov20objectDetector, or ssd0objectDetector object. The type of network that the function returns is the same as the net input.

## Version History

**Introduced in R2022b**

## References

- [1] Nagel, Markus, Mart Van Baalen, Tijmen Blankevoort, and Max Welling. "Data-Free Quantization Through Weight Equalization and Bias Correction." In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 1325-34. Seoul, Korea (South): IEEE, 2019. <https://doi.org/10.1109/ICCV.2019.00141>.

## See Also

dlquantizer | calibrate | quantize | validate | dlquantizationOptions |  
quantizationDetails

## estimateNetworkMetrics

Estimate network metrics for specific layers of a neural network

### Syntax

```
dataTable = estimateNetworkMetrics(net)
[dataTable1,dataTable2,...,dataTablen] = estimateNetworkMetrics(net1,net2,
...,netn)
```

### Description

`dataTable = estimateNetworkMetrics(net)` returns a table containing these estimated layer-wise metrics for a deep neural network:

- `LayerName` — Name of layer
- `LayerType` — Type of layer
- `NumberOfLearnables` — Number of learnable parameters (weights and biases) in the network
- `NumberOfOperations` — Total number of multiplications and additions
- `ParameterMemory (MB)` — Memory required to store all of the learnable parameters
- `NumberOfMACs` — Number of multiply-accumulate operations
- `ArithmeticIntensity` — Amount of data reuse of data that is being fetched from memory. A high arithmetic intensity indicates more data reuse.

This function estimates network metrics for learnable layers, which have weights and bias, in the network. Estimated metrics are provided for 2-D convolutional layers, 2-D grouped convolutional layers, and fully connected layers.

`[dataTable1,dataTable2,...,dataTablen] = estimateNetworkMetrics(net1,net2, ...,netn)` returns metrics for multiple networks.

This function requires Deep Learning Toolbox Model Quantization Library. To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

### Examples

#### Estimate Metrics for Neural Network Layers

This example shows how to estimate layer-wise metrics for a neural network.

Load the pretrained network. `net` is a SqueezeNet convolutional neural network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch
net
net =
  DAGNetwork with properties:
```

```

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
InputNames: {'data'}
OutputNames: {'new_classoutput'}

```

Use the `estimateNetworkMetrics` function to estimate metrics for the 2-D convolutional layers, 2-D grouped convolutional layers, and fully connected layers in your network.

```
estNet = estimateNetworkMetrics(net)
```

```
estNet=26x7 table
```

LayerName	LayerType	NumberOfLearnables	NumberOfOperations	Parameter...
"conv1"	"Convolution"	1792	4.413e+07	0.000
"fire2-squeeze1x1"	"Convolution"	1040	6.4225e+06	0.003
"fire2-expand1x1"	"Convolution"	1088	6.4225e+06	0.004
"fire2-expand3x3"	"Convolution"	9280	5.7803e+07	0
"fire3-squeeze1x1"	"Convolution"	2064	1.2845e+07	0.007
"fire3-expand1x1"	"Convolution"	1088	6.4225e+06	0.004
"fire3-expand3x3"	"Convolution"	9280	5.7803e+07	0
"fire4-squeeze1x1"	"Convolution"	4128	6.4225e+06	0.03
"fire4-expand1x1"	"Convolution"	4224	6.4225e+06	0.03
"fire4-expand3x3"	"Convolution"	36992	5.7803e+07	0.3
"fire5-squeeze1x1"	"Convolution"	8224	1.2845e+07	0.03
"fire5-expand1x1"	"Convolution"	4224	6.4225e+06	0.03
"fire5-expand3x3"	"Convolution"	36992	5.7803e+07	0.3
"fire6-squeeze1x1"	"Convolution"	12336	4.8169e+06	0.04
"fire6-expand1x1"	"Convolution"	9408	3.6127e+06	0.03
"fire6-expand3x3"	"Convolution"	83136	3.2514e+07	0.3
:				

## Compare Metrics for Floating-Point and Quantized Neural Network

This example shows how to estimate the metrics for a floating-point and quantized neural network.

Load the pretrained network. `net` is a SqueezeNet convolutional neural network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch
net
```

```
net =
DAGNetwork with properties:
```

```

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
InputNames: {'data'}
OutputNames: {'new_classoutput'}

```

Unzip and load the MerchData images as an image datastore. Define an augmentedImageDatastore object to resize the data for the network, and split the data into calibration and validation data sets to use for quantization.

```
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. Set the execution environment to MATLAB.

```
quantObj = dlquantizer(net, 'ExecutionEnvironment', 'MATLAB');
```

Use the calibrate function to exercise the network with sample inputs and collect range information.

```
calResults = calibrate(quantObj, aug_calData);
```

Attempt to calibrate with host GPU errored with the message:  
Unable to find a supported GPU device. For more information on GPU support, see GPU Support by Reverting to use host CPU.

Use the quantize method to quantize the network object and return a simulatable quantized network.

```
qNet = quantize(quantObj)
```

```
qNet =
Quantized DAGNetwork with properties:
```

```
    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
  InputNames: {'data'}
OutputNames: {'new_classoutput'}
```

Use the quantizationDetails method to extract quantization details.

Use the estimateNetworkMetrics function to compare metrics for the floating-point and quantized networks.

```
[dataTableFloat, dataTableQuantized] = estimateNetworkMetrics(net, qNet)
```

```
dataTableFloat=26x7 table
    LayerName    LayerType    NumberOfLearnables    NumberOfOperations    Parameter...
```

LayerName	LayerType	NumberOfLearnables	NumberOfOperations	Parameter...
"conv1"	"Convolution"	1792	4.413e+07	0.000
"fire2-squeeze1x1"	"Convolution"	1040	6.4225e+06	0.000
"fire2-expand1x1"	"Convolution"	1088	6.4225e+06	0.000
"fire2-expand3x3"	"Convolution"	9280	5.7803e+07	0
"fire3-squeeze1x1"	"Convolution"	2064	1.2845e+07	0.000
"fire3-expand1x1"	"Convolution"	1088	6.4225e+06	0.000
"fire3-expand3x3"	"Convolution"	9280	5.7803e+07	0
"fire4-squeeze1x1"	"Convolution"	4128	6.4225e+06	0.000
"fire4-expand1x1"	"Convolution"	4224	6.4225e+06	0.000
"fire4-expand3x3"	"Convolution"	36992	5.7803e+07	0.000

"fire5-squeeze1x1"	"Convolution"	8224	1.2845e+07	0.03
"fire5-expand1x1"	"Convolution"	4224	6.4225e+06	0.03
"fire5-expand3x3"	"Convolution"	36992	5.7803e+07	0.3
"fire6-squeeze1x1"	"Convolution"	12336	4.8169e+06	0.09
"fire6-expand1x1"	"Convolution"	9408	3.6127e+06	0.03
"fire6-expand3x3"	"Convolution"	83136	3.2514e+07	0.3
:				

dataTableQuantized=26x7 table

LayerName	LayerType	NumberOfLearnables	NumberOfOperations	ParameterMemory
"conv1"	"Convolution"	1792	4.413e+07	0.03
"fire2-squeeze1x1"	"Convolution"	1040	6.4225e+06	0.0003
"fire2-expand1x1"	"Convolution"	1088	6.4225e+06	0.0003
"fire2-expand3x3"	"Convolution"	9280	5.7803e+07	0.0003
"fire3-squeeze1x1"	"Convolution"	2064	1.2845e+07	0.0003
"fire3-expand1x1"	"Convolution"	1088	6.4225e+06	0.0003
"fire3-expand3x3"	"Convolution"	9280	5.7803e+07	0.0003
"fire4-squeeze1x1"	"Convolution"	4128	6.4225e+06	0.0003
"fire4-expand1x1"	"Convolution"	4224	6.4225e+06	0.0003
"fire4-expand3x3"	"Convolution"	36992	5.7803e+07	0.0003
"fire5-squeeze1x1"	"Convolution"	8224	1.2845e+07	0.0003
"fire5-expand1x1"	"Convolution"	4224	6.4225e+06	0.0003
"fire5-expand3x3"	"Convolution"	36992	5.7803e+07	0.0003
"fire6-squeeze1x1"	"Convolution"	12336	4.8169e+06	0.0003
"fire6-expand1x1"	"Convolution"	9408	3.6127e+06	0.0003
"fire6-expand3x3"	"Convolution"	83136	3.2514e+07	0.0003
:				

The quantized network has significantly lower parameter memory requirements than the floating-point version of the network.

## Input Arguments

### net — Neural network

DAGNetwork object | SeriesNetwork object | dlnetwork object

Neural network, specified as a DAGNetwork object, SeriesNetwork object, or dlnetwork object.

### net1, net2, ..., netn — Neural networks

DAGNetwork object | SeriesNetwork object | dlnetwork object

Neural networks, specified as a comma-separated list of DAGNetwork objects, SeriesNetwork objects, or dlnetwork objects.

## Version History

Introduced in R2022a

## See Also

### Apps

**Deep Network Quantizer**

### Functions

`dlquantizer` | `dlquantizationOptions` | `calibrate` | `quantize` | `validate` | `quantizationDetails`

### Topics

“Quantization of Deep Neural Networks”

“Quantize Layers in Object Detectors and Generate CUDA Code”

“Classify Images on FPGA Using Quantized Neural Network” (Deep Learning HDL Toolbox)

“Generate INT8 Code for Deep Learning Network on Raspberry Pi” (MATLAB Coder)

# estimateNetworkOutputBounds

Estimate output bounds of deep learning network

## Syntax

```
[YLower,YUpper] = estimateNetworkOutputBounds(net,XLower,XUpper)
```

## Description

`[YLower,YUpper] = estimateNetworkOutputBounds(net,XLower,XUpper)` estimates the lower and upper output bounds, `YLower` and `YUpper`, respectively, of the network `net` for input within the bounds specified by `XLower` and `XUpper`.

The function estimates the range of output values that the network returns when the input is between the specified lower and upper bounds. Use this function to estimate how sensitive the network predictions are to input perturbation.

The `estimateNetworkOutputBounds` function requires the Deep Learning Toolbox Verification Library support package. If this support package is not installed, use the **Add-On Explorer**. To open the **Add-On Explorer**, go to the MATLAB Toolstrip and click **Add-Ons > Get Add-Ons**.

## Examples

### Estimate Output Bounds

Estimate the output bounds for an image regression network.

Load a pretrained regression network. This network is a `dlnetwork` object that has been trained to predict the rotation angle of images of handwritten digits.

```
load("digitsRegressionMLPNet.mat");
```

Load the test data.

```
[XTest,~,TTest] = digitTest4DArrayData;
```

Select the first ten images.

```
X = XTest(:,:, :, 1:10);
T = TTest(1:10);
```

Convert the test images to `dlarray` objects.

```
X = dlarray(X, "SSCB");
```

Estimate the output bounds for an input perturbation between  $-0.01$  and  $0.01$  for each pixel. Create lower and upper bounds for the input.

```
perturbation = 0.01;
XLower = X - perturbation;
XUpper = X + perturbation;
```

Estimate the output bounds for each input.

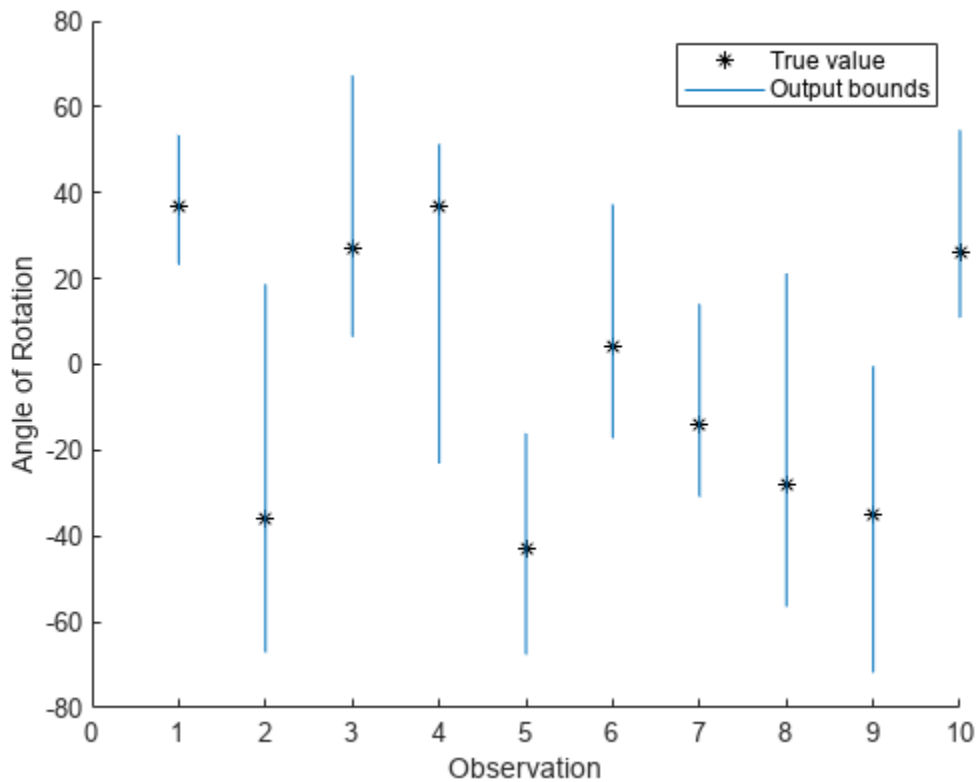
```
[YLower,YUpper] = estimateNetworkOutputBounds(net,XLower,XUpper);
```

The output bounds are dIarray objects. To plot the output bounds, first extract the data using extractdata.

```
YLower = extractdata(YLower);
YUpper = extractdata(YUpper);
```

Visualize the output bounds.

```
figure
hold on
for i = 1:10
    plot(i,T(i),"k*")
    line([i i],[YLower(i) YUpper(i)])
end
hold off
xlim([0 10])
xlabel("Observation")
ylabel("Angle of Rotation")
legend(["True value","Output bounds"])
```





## Input Arguments

### **net** — Network

initialized `dlnetwork` object

Network, specified as an initialized `dlnetwork` object. To initialize a `dlnetwork` object, use the `initialize` function.

The function supports networks with these layers:

- `imageInputLayer` with `Normalization="none"`
- `featureInputLayer` with `Normalization="none"`
- `fullyConnectedLayer`
- `convolution2dLayer` with `Dilation` set to `[1 1]` or `1` and `PaddingValue` set to `0`
- `reluLayer`
- `tanhLayer`
- `batchNormalizationLayer`
- `averagePooling2dLayer` with `PaddingValue` set to `0`
- `globalAveragePooling2dLayer`
- `dropoutLayer`
- `sigmoidLayer`

The function does not support networks with multiple inputs and multiple outputs.

The function estimates the output bounds using the final layer in the network. For most applications, use the final fully connected layer when computing the output bounds. If your network has a different layer as its final layer, remove the layer before calling the function.

### **XLower** — Input lower bound

formatted `darray` object

Input lower bound, specified as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The lower and upper bounds, `XLower` and `XUpper`, must have the same size and format. The function computes the results across the batch ("B") dimension of the input lower and upper bounds.

### **XUpper** — Input upper bound

formatted `darray` object

Input upper bound, specified as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The lower and upper bounds, `XLower` and `XUpper`, must have the same size and format. The function computes the results across the batch ("B") dimension of the input lower and upper bounds.

## Output Arguments

### **YLower** — Output lower bound

formatted `darray` object

Output lower bound, returned as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The function estimates the output bounds for each observation across the batch ("B") dimension. If you supply `k` upper bounds and lower bounds, then `YLower` contains `k` output lower bounds. For more information, see "Algorithms" on page 1-1973.

### **YUpper — Output upper bound**

formatted `darray` object

Output upper bound, returned as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The function estimates the output bounds for each observation across the batch ("B") dimension. If you supply `k` upper bounds and lower bounds, then `YUpper` contains `k` output upper bounds. For more information, see "Algorithms" on page 1-1973.

## **Algorithms**

Let  $X$  be the input for which you want to estimate the output bounds. To use the `estimateNetworkOutputBounds` function, you must specify a lower and upper bound for the input. For example, let  $\epsilon$  be a small perturbation. You can define a lower and upper bound for the input as  $X_{\text{lower}} = X - \epsilon$  and  $X_{\text{upper}} = X + \epsilon$ , respectively.

To estimate the output bounds for the network `net` and the input bounds  $X_{\text{lower}}$  and  $X_{\text{upper}}$ , the function performs these steps.

- 1 Create an input set using the lower and upper input bounds.
- 2 Pass the input set through the network and return an output set. To reduce computational overhead, the function performs abstract interpretation by approximating the output of each layer using the DeepPoly [2] method.
- 3 Return the minimum and maximum estimated output values for the input set.

If you specify multiple pairs of input lower and upper bounds, then the function estimates the output bounds for each pair of input bounds.

---

**Note** Because of floating-point round-off error, the output bounds might be slightly different when working with networks produced using C/C++ code generation.

---

## **Version History**

**Introduced in R2022b**

## **References**

- [1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples." Preprint, submitted March 20, 2015. <https://arxiv.org/abs/1412.6572>.
- [2] Singh, Gagandeep, Timon Gehr, Markus Püschel, and Martin Vechev. "An Abstract Domain for Certifying Neural Networks". *Proceedings of the ACM on Programming Languages* 3, no. POPL (January 2, 2019): 1-30. <https://doi.org/10.1145/3290354>.

## **See Also**

`verifyNetworkRobustness` | `dlarray` | `dlnetwork`

## **Topics**

“Verify Robustness of Deep Learning Neural Network”

“Generate Untargeted and Targeted Adversarial Examples for Image Classification”

“Train Image Classification Network Robust to Adversarial Examples”

“Train Network Using Custom Training Loop”

“Out-of-Distribution Detection for Deep Neural Networks”

# 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 1-54.

## 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

### 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`.

Example: `monitor.Metrics = ["TrainingLoss", "ValidationLoss"];`

Data Types: char | string

### 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.

Example: `monitor.Info = ["GradientDecayFactor", "SquaredGradientDecayFactor"];`

Data Types: char | string

### Stop — Request to stop trial

false or 0 (default) | true or 1

This property is read-only.

Request 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`

### **Progress — Training progress**

0 (default) | numeric scalar | `darray`

Training progress percentage, specified as a numeric scalar or `darray` object between 0 and 100.

Example: `monitor.Progress = 17;`

### **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`

### **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`

### **MetricData — Metric column values**

structure

This property is read-only.

Metric column values, specified as a structure. Use the `Metrics` property to specify the field names for the structure. Each field is a matrix that contains the custom training loop step values and metric values recorded by the `recordMetrics` function.

Data Types: `struct`

### **InfoData — Information column values**

structure

This property is read-only.

Information column values, specified as a structure. Use the `Info` property to specify the field names for the structure. Each field is a column vector that contains the values updated by the `updateInfo` function.

Data Types: `struct`

## **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 = 100 * (iteration/numIterations);
```

## 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.
- An `experiments.Monitor` object has the same properties and object functions as a `TrainingProgressMonitor` object. Therefore, you can easily adapt your custom training loop plotting code for use in an **Experiment Manager** setup script. For more information, see “Prepare Plotting Code for Custom Training Experiment” on page 1-1836.

## Version History

Introduced in R2021a

### **R2022b: InfoData and MetricData properties**

The properties `InfoData` and `MetricData` store the values of the information and metric columns for your trial.

### **See Also**

#### **Apps**

**Experiment Manager**

#### **Objects**

`trainingProgressMonitor`

#### **Topics**

“Use Bayesian Optimization in Custom Training Experiments”

“Run a Custom Training Experiment for Image Comparison”

“Use Experiment Manager to Train Generative Adversarial Networks (GANs)”

“Custom Training with Multiple GPUs in Experiment Manager”

## exportNetworkToTensorFlow

Export Deep Learning Toolbox network or layer graph to TensorFlow

### Syntax

```
exportNetworkToTensorFlow(net,modelPackage)
exportNetworkToTensorFlow(lgraph,modelPackage)
```

### Description

`exportNetworkToTensorFlow(net,modelPackage)` exports the MATLAB deep learning network `net` and saves it as a TensorFlow model in the Python® package `modelPackage`. For information on how to load the TensorFlow model in Python, see “Load Exported TensorFlow Model” on page 1-704.

The `exportNetworkToTensorFlow` function requires the Deep Learning Toolbox Converter for TensorFlow Models. If this support package is not installed, then `exportNetworkToTensorFlow` provides a download link.

`exportNetworkToTensorFlow(lgraph,modelPackage)` exports the MATLAB deep learning layer graph `lgraph` and saves it as a TensorFlow model in the Python package `modelPackage`.

If the MATLAB network or layer graph contains a custom or built-in MATLAB layer that `exportNetworkToTensorFlow` cannot convert to a TensorFlow layer, the `exportNetworkToTensorFlow` function exports this layer as a custom TensorFlow layer. For more information on which MATLAB layers `exportNetworkToTensorFlow` can convert to TensorFlow layers, see “Layers Supported for Exporting to TensorFlow” on page 1-702. For an example, see “Export Layer Graph with Custom Layer to TensorFlow” on page 1-698.

### Examples

#### Export Network to TensorFlow

Save a MATLAB deep learning network as a TensorFlow model by using the `exportNetworkToTensorFlow` function.

Download and install the Deep Learning Toolbox Converter for TensorFlow Models support package. You can enter `exportNetworkToTensorFlow` at the command prompt to check whether the support package is installed. If the 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**.

Load the pretrained squeezenet convolutional neural network as a `DAGNetwork` object.

```
net = squeezenet
net =
  DAGNetwork with properties:
    Layers: [68x1 nnet.cnn.layer.Layer]
```



```

Connections: [75x2 table]
InputNames: {'data'}
OutputNames: {'ClassificationLayer_predictions'}

```

Export the network `net` to TensorFlow. The `exportNetworkToTensorFlow` function saves the TensorFlow model in the Python package `myModel`.

```
exportNetworkToTensorFlow(net, "myModel")
```

Run this code in Python to load the exported TensorFlow model from the `myModel` package.

```
import myModel
model = myModel.load_model()
```

Save the exported model in the TensorFlow `SavedModel` format. Saving `model` in `SavedModel` format is optional. You can perform deep learning workflows directly with `model`. For an example that shows how to classify an image with the exported TensorFlow model, see “Export Network to TensorFlow and Classify Image” on page 1-695.

```
model.save("myModelTF")
```

## Export Network to TensorFlow and Classify Image

Use a MATLAB network to classify an image. Save the network as a TensorFlow model and use the TensorFlow model to classify the same image.

### Classify Image in MATLAB

Load the pretrained `squeezenet` convolutional network as a `DAGNetwork` object.

```
net = squeezenet
```

```
net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
InputNames: {'data'}
OutputNames: {'ClassificationLayer_predictions'}
```

Specify the class names.

```
classNames = net.Layers(end).Classes;
```

Read the image you want to classify. Resize the image to the input size of the network.

```
Im = imread("peppers.png");
```

```
InputSize = net.Layers(1).InputSize;
Im = imresize(Im,InputSize(1:2));
```

Predict class labels and classification scores.

```
[label,score] = classify(net,Im);
```

Show the image with the classification label.

```
imshow(Im)  
title(ClassNames(label),FontSize=12)
```

**bell pepper**



### **Export Network and Image Data**

Export the network `net` to TensorFlow. The `exportNetworkToTensorFlow` function saves the TensorFlow model in the Python package `myModel`.

```
exportNetworkToTensorFlow(net, "myModel")
```

Permute the 2-D image data from the Deep Learning Toolbox™ ordering (HWCN) to the TensorFlow ordering (NHWC), where H, W, and C are the height, width, and number of channels of the image, respectively, and N is the number of images. Save the image in a MAT file.

```
ImTF = permute(Im, [4,1,2,3]);
```

```
filename = "peppers.mat";  
save(filename, "ImTF")
```

### **Classify Image with Exported TensorFlow Model**

Run this code in Python to load the exported TensorFlow model and use the model for image classification.

Load the exported model from the Python package `myModel`.

```
import myModel  
model = myModel.load_model()
```

Classify the image with the exported model. For more information on how to compare prediction results between MATLAB and TensorFlow, see “Inference Comparison Between TensorFlow and Imported Networks for Image Classification”.

```
score_tf = model.predict(ImTF)
```

## Export Untrained Layer Graph to TensorFlow

Export an untrained layer graph to TensorFlow and train the exported TensorFlow model.

### Create Layer Graph

Create a long short-term memory (LSTM) network to classify sequence data. An LSTM network takes sequence data as input and makes predictions based on the individual time steps of the sequence data.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [
    sequenceInputLayer(inputSize)
    biLstmLayer(numHiddenUnits,OutputMode="last")
    fullyConnectedLayer(numClasses)
    softmaxLayer];

lgraph = layerGraph(layers);
```

### Create Training Data Set

Load the Japanese Vowels training data set. XTrain is a cell array containing 270 sequences of dimension 12 and varying length. YTrain is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
[XTrain,YTrain] = japaneseVowelsTrainData;
```

Prepare the sequence data in XTrain for padding. For more information, see "Sequence Classification Using Deep Learning".

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

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

Pad XTrain along the second dimension.

```
XTrain = padsequences(XTrain,2);
```

Permute the sequence data from the Deep Learning Toolbox™ ordering (CSN) to the TensorFlow ordering (NSC), where C is the number of features of the sequence, S is the sequence length, and N is the number of sequence observations. Save the training data to a MAT file.

```
XTrain = permute(XTrain,[3,2,1]);
YTrain = double(YTrain)-1;
```

```
filename = "training_data.mat";  
save(filename, "XTrain", "YTrain")
```

### Export Layer Graph to TensorFlow

Export the layer graph `lgraph` to TensorFlow. The `exportNetworkToTensorFlow` function saves the TensorFlow model in the Python package `myModel`.

```
exportNetworkToTensorFlow(lgraph, "myModel")
```

### Train Exported TensorFlow Model

Run this code in Python to load the exported model from the Python package `myModel`. You can compile and train the exported model in Python. To train `model`, use the training data in `training_data.mat`. For more information on how to load data from a MAT file into Python, see “Inference Comparison Between TensorFlow and Imported Networks for Image Classification”.

```
import myModel  
model = myModel.load_model()
```

### Export Layer Graph with Custom Layer to TensorFlow

Export a layer graph, which contains a MATLAB custom layer, to TensorFlow.

#### Create Layer Graph

Create a PReLU layer by defining the custom layer `preluLayer`. Display the definition of the custom layer.

type `preluLayer.m`

```
classdef preluLayer < nnet.layer.Layer  
    % Example custom PReLU layer.  
  
    properties (Learnable)  
        % Layer learnable parameters.  
  
        % Scaling coefficient.  
        Alpha  
    end  
  
    methods  
        function layer = preluLayer(args)  
            % layer = preluLayer creates a PReLU layer.  
            %  
            % layer = preluLayer(numChannels,Name=name) also specifies the  
            % layer name.  
  
            arguments  
                args.Name = "";  
            end  
  
            % Set layer name.  
            layer.Name = args.Name;  
  
            % Set layer description.
```

```

        layer.Description = "PReLU";
    end

    function layer = initialize(layer,layout)
        % layer = initialize(layer,layout) initializes the learnable
        % parameters of the layer for the specified input layout.

        % Skip initialization of nonempty parameters.
        if ~isempty(layer.Alpha)
            return
        end

        % Input data size.
        sz = layout.Size;
        ndims = numel(sz);

        % Find number of channels.
        idx = finddim(layout,"C");
        numChannels = sz(idx);

        % Initialize Alpha.
        szAlpha = ones(1,ndims);
        szAlpha(idx) = numChannels;
        layer.Alpha = rand(szAlpha);
    end

    function Z = predict(layer, X)
        % Z = predict(layer, X) forwards the input data X through the
        % layer and outputs the result Z.

        Z = max(0, X) + layer.Alpha .* min(0, X);
    end
end
end

```

Create a layer graph.

```

layers = [
    imageInputLayer([31 53 3],Name="image",Normalization="none")
    preluLayer(Name="prelu")
    regressionLayer];

lgraph = layerGraph(layers);

```

### Export Layer Graph to TensorFlow

Export the layer graph `lgraph` to TensorFlow. The `exportNetworkToTensorFlow` function saves the TensorFlow model in the Python package `myModel` and the definition of the custom layer in the `customLayers` folder of the `myModel` package.

```
exportNetworkToTensorFlow(lgraph,"myModel")
```

Warning: Layer 'prelu': Layer class 'preluLayer' was exported into an incomplete TensorFlow custom layer.

Display the definition of the TensorFlow custom layer `preluLayer.py`.

```
type ./myModel/customLayers/preluLayer.py
```

```

# This file was created by
# MATLAB Deep Learning Toolbox Converter for TensorFlow Models.

```

```
# 03-Mar-2023 11:09:11

import tensorflow as tf
import sys      # Remove this line after completing the layer definition.

class preluLayer(tf.keras.layers.Layer):
    # Add any additional layer hyperparameters to the constructor's
    # argument list below.
    def __init__(self, Alpha_Shape_=None, name=None):
        super(preluLayer, self).__init__(name=name)
        # Learnable parameters: These have been exported from MATLAB and will be loaded automatically
        self.Alpha = tf.Variable(name="Alpha", initial_value=tf.zeros(Alpha_Shape_), trainable=True)

    def call(self, input1):
        # Add code to implement the layer's forward pass here.
        # The input tensor format(s) are: BSSC
        # The output tensor format(s) are: BSSC
        # where B=batch, C=channels, T=time, S=spatial(in order of height, width, depth,...)

        # Remove the following 3 lines after completing the custom layer definition:
        print("Warning: load_model(): Before you can load the model, you must complete the definition of preluLayer.py")
        print("Exiting...")
        sys.exit("See the warning message above.")

        return output1
```

### Load Exported Layer Graph

This section describes the steps that you must perform in Python to load the exported TensorFlow model.

Edit the definition of `preluLayer.py` by implementing the forward computation in `call`.

```
def call(self, input1):
    output1 = tf.math.maximum(input1,0.0) + self.Alpha * tf.math.minimum(0.0,input1)
    return output1
```

Delete the lines in `preluLayer.py`, as instructed by the comments in the file. View the updated custom layer `preluLayer.py`.

```
import tensorflow as tf

class preluLayer(tf.keras.layers.Layer):
    # Add any additional layer hyperparameters to the constructor's
    # argument list below.
    def __init__(self, Alpha_Shape_=None, name=None):
        super(preluLayer, self).__init__(name=name)
        # Learnable parameters: These have been exported from MATLAB and will be loaded automatically
        self.Alpha = tf.Variable(name="Alpha", initial_value=tf.zeros(Alpha_Shape_), trainable=True)

    def call(self, input1):
        output1 = tf.math.maximum(input1,0.0) + self.Alpha * tf.math.minimum(0.0,input1)
        return output1
```

In this example, you only have to edit `preluLayer.py`. In other cases, you might have to edit `model.py` to pass arguments to custom layer calls.

Before loading the model, you might have to restart your Python kernel for the changes to take effect. Load the model from the Python package `myModel`.

```
import myModel
model = myModel.load_model()
```

## Input Arguments

### **net** — Deep Learning Toolbox network

SeriesNetwork object | DAGNetwork object | dlnetwork object

Deep Learning Toolbox network, specified as a SeriesNetwork object, DAGNetwork object, or dlnetwork object.

You can get a trained network by:

- Using a Deep Learning Toolbox function to load a pretrained network. For example, use the `efficientnetb0` function.
- Downloading a pretrained network from the MATLAB Deep Learning Model Hub.
- Training your own network. Use `trainNetwork` to train a DAGNetwork or SeriesNetwork object. Use a custom training loop to train a dlnetwork object.

You can also export an initialized dlnetwork object to TensorFlow.

### **modelPackage** — Name of Python package containing exported model

string scalar | character vector

Name of the Python package containing the exported TensorFlow model, specified as a string scalar or character vector. The `modelPackage` package contains:

- The `_init_.py` file, which defines the `modelPackage` folder as a regular Python package.
- The `model.py` file, which contains the code that defines the untrained TensorFlow-Keras model.
- The `README.txt` file, which provides instructions on how to load the TensorFlow model and save it in HDF5 or SavedModel format. For more details, see “Load Exported TensorFlow Model” on page 1-704 and “Save Exported TensorFlow Model in Standard Format” on page 1-704.
- The `weights.h5` file, which contains the model weights in HDF5 format.
- The `customLayers` folder, which contains one file for each exported custom layer. Each file is an incomplete definition of a TensorFlow custom layer. You must edit or replace each of these files before you can load the model in Python. The software creates the `customLayers` folder only when the MATLAB network or layer graph contains a custom or built-in MATLAB layer that `exportNetworkToTensorFlow` cannot convert to a TensorFlow layer.

Example: `"myModel"`

### **lgraph** — Deep Learning Toolbox layer graph

LayerGraph object | Layer array

Deep Learning Toolbox layer graph, specified as a LayerGraph object or Layer array.

## Limitations

- To load an exported TensorFlow model, you must have:
  - TensorFlow version r2.0 or later
  - Python version 3.0 or later
  - The TensorFlow module `tfa` for a MATLAB network or layer graph that contains one or more of the following layers:
    - `groupNormalizationLayer`
    - `instanceNormalizationLayer`
    - `layerNormalizationLayer` with `OperationDimension` set to "batch-excluded"

## More About

### Layers Supported for Exporting to TensorFlow

The `exportNetworkToTensorFlow` function supports these Deep Learning Toolbox layers for export as TensorFlow layers.

- **Deep Learning Toolbox Layers**

<b>Input Layers</b>
<code>imageInputLayer</code>
<code>image3dInputLayer</code>
<code>featureInputLayer</code>
<code>sequenceInputLayer</code>

<b>Convolution and Fully Connected Layers</b>
<code>convolution1dLayer</code>
<code>convolution2dLayer</code>
<code>convolution3dLayer</code>
<code>groupedConvolution2dLayer</code>
<code>fullyConnectedLayer</code>
<code>transposedConv2dLayer</code>
<code>transposedConv3dLayer</code>

<b>Sequence Layers</b>
<code>averagePooling1dLayer</code>
<code>biLstmLayer</code>
<code>convolution1dLayer</code>
<code>flattenLayer</code>
<code>globalMaxPooling1dLayer</code>
<code>gruLayer</code>



<b>Sequence Layers</b>
lstmLayer
lstmProjectedLayer
maxPooling1dLayer
selfAttentionLayer
sequenceFoldingLayer
sequenceInputLayer
sequenceUnfoldingLayer
transposedConv1dLayer

<b>Activation Layers</b>
clippedReluLayer
eluLayer
geluLayer
functionLayer
leakyReluLayer
reluLayer
swishLayer
tanhLayer

<b>Normalization, Dropout, and Cropping Layers</b>
batchNormalizationLayer
crop2dLayer
crop3dLayer
crossChannelNormalizationLayer
dropoutLayer
groupNormalizationLayer
instanceNormalizationLayer
layerNormalizationLayer

<b>Pooling and Unpooling Layers</b>
averagePooling1dLayer
averagePooling2dLayer
averagePooling3dLayer
globalAveragePooling1dLayer
globalAveragePooling2dLayer
globalAveragePooling3dLayer
maxPooling1dLayer
maxPooling2dLayer

<b>Pooling and Unpooling Layers</b>
maxPooling3dLayer
globalMaxPooling1dLayer
globalMaxPooling2dLayer
globalMaxPooling3dLayer

<b>Combination Layers</b>
additionLayer
multiplicationLayer
depthConcatenationLayer
concatenationLayer

<b>Output Layers</b>
softmaxLayer
sigmoidLayer
classificationLayer
regressionLayer

- **Image Processing Toolbox™ Layers**

depthToSpace2dLayer
resize2dLayer
resize3dLayer
spaceToDepthLayer

- **Lidar Toolbox™ Layers**

pointCloudInputLayer
----------------------

### Load Exported TensorFlow Model

This section describes how to load a TensorFlow model in Python from the package `modelPackage`, which the `exportNetworkToTensorFlow` creates. For an example, see “Export Network to TensorFlow” on page 1-694.

Load the exported TensorFlow model with weights.

```
import modelPackage
model = modelPackage.load_model()
```

Load the exported TensorFlow model without weights.

```
import modelPackage
model = modelPackage.load_model(load_weights=False)
```

### Save Exported TensorFlow Model in Standard Format

Optionally, you can save the exported TensorFlow model in `SavedModel` or `HDF5` format. You must first load the exported TensorFlow model by following the instructions in “Load Exported TensorFlow

Model” on page 1-704. For an example that shows how to save an exported model to SavedModel format, see “Export Network to TensorFlow” on page 1-694.

Save the loaded TensorFlow model in SavedModel format.

```
model.save("modelName")
```

Save the loaded TensorFlow model in HDF5 format.

```
model.save("modelName", save_format="h5")
```

## Tips

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.

## Version History

**Introduced in R2022b**

### See Also

[importTensorFlowNetwork](#) | [importTensorFlowLayers](#) | [importNetworkFromPyTorch](#) | [exportONNXNetwork](#)

### Topics

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Pretrained Deep Neural Networks”

“Inference Comparison Between TensorFlow and Imported Networks for Image Classification”

“Define Custom Deep Learning Layer with Learnable Parameters”

# 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 arguments.

## Examples

### Export Network to ONNX Format

Load the pretrained SqueezeNet convolutional neural network.

```
net = squeezeNet
```

```
net =
  DAGNetwork with properties:
    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

Export the network `net` as an ONNX format file called `squeezenet.onnx`. Save the file to the current folder. If the Deep Learning Toolbox Converter for ONNX Model Format support package is not installed, then `exportONNXNetwork` 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: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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)
```

Deep Learning Network Analyzer

**lgraph1** 68 layers 0 warnings 0 errors

Analysis date: 04-Jan-2021 23:51:10

ANALYSIS RESULT				
	Name	Type	Activations	Learnables
57	fire9-relu_squeeze... ReLU	ReLU	14×14×64	-
58	fire9-expand1x1 256 1×1×64 convolutio...	Convolution	14×14×256	Weights 1×1×64×256 Bias 1×1×256
59	fire9-relu_expand1x1 ReLU	ReLU	14×14×256	-
60	fire9-expand3x3 256 3×3×64 convolutio...	Convolution	14×14×256	Weights 3×3×64×256 Bias 1×1×256
61	fire9-relu_expand3x3 ReLU	ReLU	14×14×256	-
62	fire9-concat Depth concatenation of...	Depth concatenation	14×14×512	-
63	drop9 50% dropout	Dropout	14×14×512	-
64	conv10 1000 1×1×512 convolu...	Convolution	14×14×1000	Weigh... 1×1×512×10... Bias 1×1×1000
65	relu_conv10 ReLU	ReLU	14×14×1000	-
66	pool10 Global average pooling	Global Average Po...	1×1×1000	-
67	prob softmax	Softmax	1×1×1000	-
68	ClassificationLayer... crossentropyex with 'te...	Classification Output	1×1×1000	-

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: [67x1 nnet.cnn.layer.Layer]
        Connections: [74x2 table]
        InputNames: {'data'}
        OutputNames: {1x0 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)
```

The screenshot shows the 'Deep Learning Network Analyzer' window. The title bar includes the application name and standard window controls. The main window is titled 'lgraph2' and shows an analysis date of '04-Jan-2021 23:56:17'. On the right side, there are statistics: 67 layers, 0 warnings, and 2 errors. The left pane displays a layer graph starting with an input node 'data' and followed by several layers including 'conv1', 'relu\_conv1', 'pool1', and multiple 'fire' (fire2 and fire3) blocks. The right pane is divided into two sections: 'ISSUES' and 'ANALYSIS RESULT'. The 'ISSUES' section contains two error messages: 'Missing output layer. The network must have at least one output layer.' and 'Unconnected output. Each layer output must be connected to the input of another layer.' The 'ANALYSIS RESULT' section is a table with columns for Name, Type, Activations, and Learnables, listing layers 61 through 67.

ISSUES		Found in	Message
!	Network		Missing output layer. The network must have at least one output layer.
!	prob		Unconnected output. Each layer output must be connected to the input of another layer.

ANALYSIS RESULT				
	Name	Type	Activations	Learnables
61	fire9-relu_expand3x3 ReLU	ReLU	14×14×256	-
62	fire9-concat Depth concatenation of...	Depth concatenation	14×14×512	-
63	drop9 50% dropout	Dropout	14×14×512	-
64	conv10 1000 1×1×512 convolu...	Convolution	14×14×1000	Weigh... 1×1×512×10... Bias 1×1×1000
65	relu_conv10 ReLU	ReLU	14×14×1000	-
66	pool10 Global average pooling	Global Average Po...	1×1×1000	-
67	! prob softmax	Softmax	1×1×1000	-

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 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: `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: NetworkName="my\_squeezenet"

**OpsetVersion — Version of ONNX operator set**

8 (default) | 6 | 7 | 9 | 10 | 11 | 12 | 13

Version of ONNX operator set to use in the exported model, specified as a positive integer in the range [6 13]. 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: OpsetVersion=6

**BatchSize — Batch size of ONNX network**

[] (default) | positive integer

Batch size of the ONNX network, specified as [] or as a positive integer. If you specify BatchSize as [], the ONNX network has a dynamic batch size. If you specify BatchSize as a positive integer  $k$ , the ONNX network has a fixed batch size of  $k$ .

Example: BatchSize=10

**Limitations**

- exportONNXNetwork supports ONNX versions as follows:
  - The function supports ONNX intermediate representation version 7.
  - The function supports ONNX operator sets 6 to 14.
- 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 1-711), 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:

<b>ONNX Exporter Supported Layers</b>
<b>Deep Learning Toolbox Layers</b>
additionLayer
averagePooling1dLayer
averagePooling2dLayer
averagePooling3dLayer
batchNormalizationLayer
bilstmLayer
ClassificationOutputLayer
clippedReluLayer
concatenationLayer
convolution1dLayer
convolution2dLayer
convolution3dLayer
crop2dLayer
CrossChannelNormalizationLayer
depthConcatenationLayer
dropoutLayer
eluLayer
featureInputLayer
flattenLayer
fullyConnectedLayer
globalAveragePooling1dLayer
globalAveragePooling2dLayer
globalAveragePooling3dLayer
globalMaxPooling1dLayer
globalMaxPooling2dLayer
globalMaxPooling3dLayer

<b>ONNX Exporter Supported Layers</b>
groupedConvolution2dLayer
groupNormalizationLayer
gruLayer
imageInputLayer
image3dInputLayer
leakyReluLayer
lstmLayer
lstmProjectedLayer
maxPooling1dLayer
maxPooling2dLayer
maxPooling3dLayer
maxUnpooling2dLayer
multiplicationLayer
RegressionOutputLayer
reluLayer
sequenceInputLayer
sigmoidLayer
softmaxLayer
swishLayer
tanhLayer
transposedConv2dLayer
transposedConv3dLayer
<b>ONNX Importer Custom Layers</b>
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
<b>Keras Importer Custom Layers</b>
nnet.keras.layer.FlattenCStyleLayer
nnet.keras.layer.GlobalAveragePooling2dLayer
nnet.keras.layer.TanhLayer

<b>ONNX Exporter Supported Layers</b>
nnet.keras.layer.ZeroPadding2dLayer
<b>Caffe Importer Custom Layers</b>
nnet.caffe.layer.TanhLayer
<b>Computer Vision Toolbox™ Layers</b>
pixelClassificationLayer
rcnnBoxRegressionLayer
roiInputLayer
roiMaxPooling2dLayer
<b>Image Processing Toolbox Layers</b>
depthToSpace2dLayer
resize2dLayer
resize3dLayer
spaceToDepthLayer
<b>Lidar Toolbox Layers</b>
pointCloudInputLayer
<b>Text Analytics Toolbox™ Layers</b>
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".

## Version History

Introduced in R2018a

## 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**

`importONNXNetwork` | `importONNXLayers` | `importTensorFlowNetwork` |  
`importTensorFlowLayers` | `importNetworkFromPyTorch` | `exportNetworkToTensorFlow` |  
`layerGraph`

### **Topics**

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Pretrained Deep Neural Networks”

“Deep Learning in MATLAB”

## extractdata

Extract data from dLarray

### Syntax

```
Y = extractdata(X)
```

### Description

`Y = extractdata(X)` returns the data in the dLarray `X`. The output `Y` has the same data type as the underlying data in `X`.

### 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

#### X — Input dLarray

dLarray object

Input dLarray, specified as a dLarray object.

Example: `X = 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 X.

## Tips

- If X 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”.

## Version History

Introduced in R2019b

## 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`

# featureInputLayer

Feature input layer

## Description

A feature input layer inputs feature data to a neural 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  $Y = \text{func}(X)$ , where X is the input data and the output Y is the normalized data.

---

**Tip** The software, by default, automatically calculates the normalization statistics when using the `trainNetwork` function. To save time when training, specify the required statistics for normalization and set the `ResetInputNormalization` option in `trainingOptions` to 0 (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 1-by-numFeatures 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 `trainNetwork` function calculates the mean. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the Mean property to a numeric scalar or a numeric array.

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 1-by-numFeatures 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 `trainNetwork` function calculates the standard deviation. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the StandardDeviation property to a numeric scalar or a numeric array.

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 1-by-`numFeatures` 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 `trainNetwork` function calculates the minima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Min` property to a numeric scalar or a numeric array.

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 1-by-`numFeatures` 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 `trainNetwork` function calculates the maxima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Max` property to a numeric scalar or a numeric array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SplitComplexInputs — Flag to split input data into real and imaginary components**

0 (false) (default) | 1 (true)

This property is read-only.

Flag to split input data into real and imaginary components specified as one of these values:

- 0 (false) - Do not split input data.
- 1 (true) - Split data into real and imaginary components.

When `SplitComplexInputs` is 1, then the layer outputs twice as many channels as the input data. For example, if the input data is complex-values with `numChannels` channels, then the layer outputs data with  $2 \times \text{numChannels}$  channels, where channels 1 through `numChannels` contain the real components of the input data and `numChannels+1` through  $2 \times \text{numChannels}$  contain the imaginary components of the input data. If the input data is real, then channels `numChannels+1` through  $2 \times \text{numChannels}$  are all zero.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

For an example showing how to train a network with complex-valued data, see “Train Network with Complex-Valued Data”.

**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 the name ' '.

Data Types: `char` | `string`

**NumInputs — Number of inputs**

0 (default)

This property is read-only.

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

**InputNames — Input names**

{ } (default)

This property is read-only.

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  
SplitComplexInputs: 0
```

```
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 flatten layer and 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')
    flattenLayer('name', 'flatten')
    concatenationLayer(1, 2, 'Name', 'concat')
    fullyConnectedLayer(numClasses, 'Name', 'fc2')
    softmaxLayer('Name', 'softmax')];
```

Convert the layers to a layer graph.

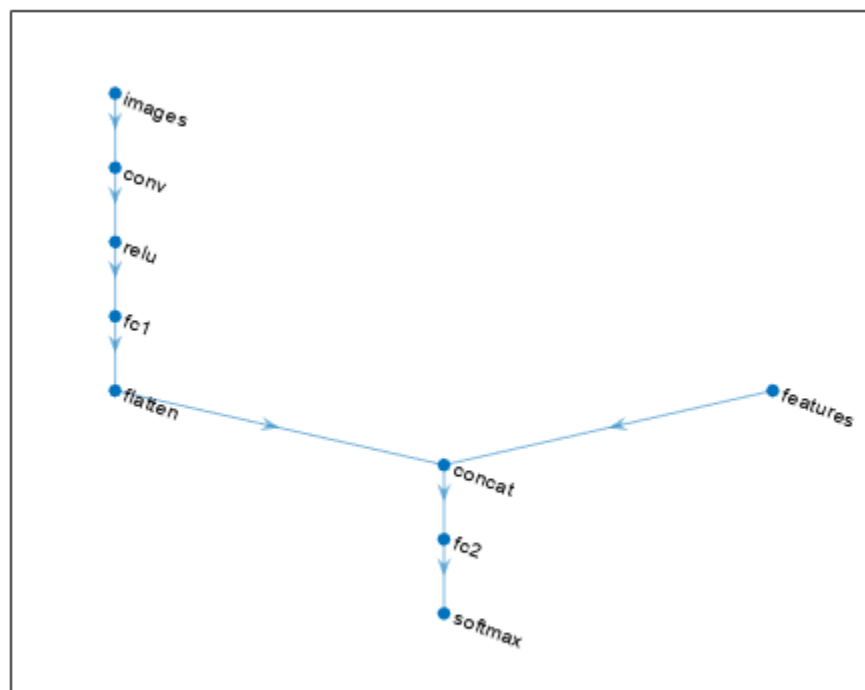
```
lgraph = layerGraph(layers);
```

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: [9x1 nnet.cnn.layer.Layer]  
  Connections: [8x2 table]  
  Learnables: [6x3 table]  
    State: [0x3 table]  
  InputNames: {'images' 'features'}  
  OutputNames: {'softmax'}  
  Initialized: 1
```

[View summary with summary.](#)

## Version History

Introduced in R2020b

### Extended Capabilities

#### C/C++ Code Generation

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

- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

#### 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).

- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

### See Also

[trainNetwork](#) | [fullyConnectedLayer](#) | [image3dInputLayer](#) | [Deep Network Designer](#) | [imageInputLayer](#) | [sequenceInputLayer](#) | [dlnetwork](#)

#### Topics

“Train Network with Numeric Features”

“Create Simple Deep Learning Neural Network for Classification”

“Train Convolutional Neural Network for Regression”

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“List of Deep Learning Layers”

## finddim

Find dimensions with specified label

### Syntax

```
dim = finddim(X,label)
dim = finddim(layout,label)
```

### Description

`dim = finddim(X,label)` returns the dimensions in the `dLarray` object `X` that have the specified dimension label.

`dim = finddim(layout,label)` returns the dimensions in the `networkDataLayout` object `layout` that have the specified dimension label.

### Examples

#### Find Dimensions of dLarray Object

Create a formatted `dLarray` with format "SSCBT" (spatial, spatial, channel, batch, time).

```
X = dLarray(randn(6,5,4,3,2), "SSCBT");
```

View the dimension labels of `X`.

```
dims(X)
```

```
ans =
'SSCBT'
```

Find the dimensions with the label "S".

```
dim = finddim(X, "S")
```

```
dim = 1×2
```

```
    1    2
```

#### Find Dimensions of Network Data Layout Object

Create a formatted network data layout object representing 2-D image sequences. Specify the format "SSCBT" (spatial, spatial, channel, batch, time).

```
layout = networkDataLayout([227 227 3 NaN 100], "SSCBT");
```

Find the dimensions with the label "S".

```
dim = finddim(layout, "S")
dim = 1x2
     1     2
```

## Input Arguments

### **X** — Input data

`dLarray` object

Input data, specified as a formatted or unformatted `dLarray` object.

### **layout** — Network data layout

`networkDataLayout` object

Network data layout, specified as a `networkDataLayout` object.

### **label** — Dimension label

"S" | "C" | "B" | "T" | "U"

Dimension label, specified as one of the following `dLarray` dimension labels:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

## Output Arguments

### **dim** — Dimension

vector of positive integers

Dimension, returned as a vector of positive integers.

If the input is unformatted or has no dimension labels that match `label`, then `dim` is empty.

## Version History

Introduced in R2019b

## Extended Capabilities

### **C/C++ Code Generation**

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

Usage notes and limitations:

- The input data must be a `dlarray` object.
- 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 input data must be a `dlarray` object.
- The label argument must be a compile-time constant.

### **See Also**

`dims` | `stripdims` | `dlarray` | `networkDataLayout`



# 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. 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:
    InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
    Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
```

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'         2-D Convolution      20 7x7 convolutions with s
    3  'conv2d_1_relu'    ReLU                 ReLU
    4  'conv2d_2'         2-D Convolution      20 3x3 convolutions with s
    5  'conv2d_2_relu'    ReLU                 ReLU
    6  'gaussian_noise_1' PLACEHOLDER LAYER    Placeholder for 'GaussianN
    7  'gaussian_noise_2' PLACEHOLDER LAYER    Placeholder for 'GaussianN
    8  'max_pooling2d_1'  2-D Max Pooling      2x2 max pooling with strid
    9  'max_pooling2d_2'  2-D Max Pooling      2x2 max pooling with strid
   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.

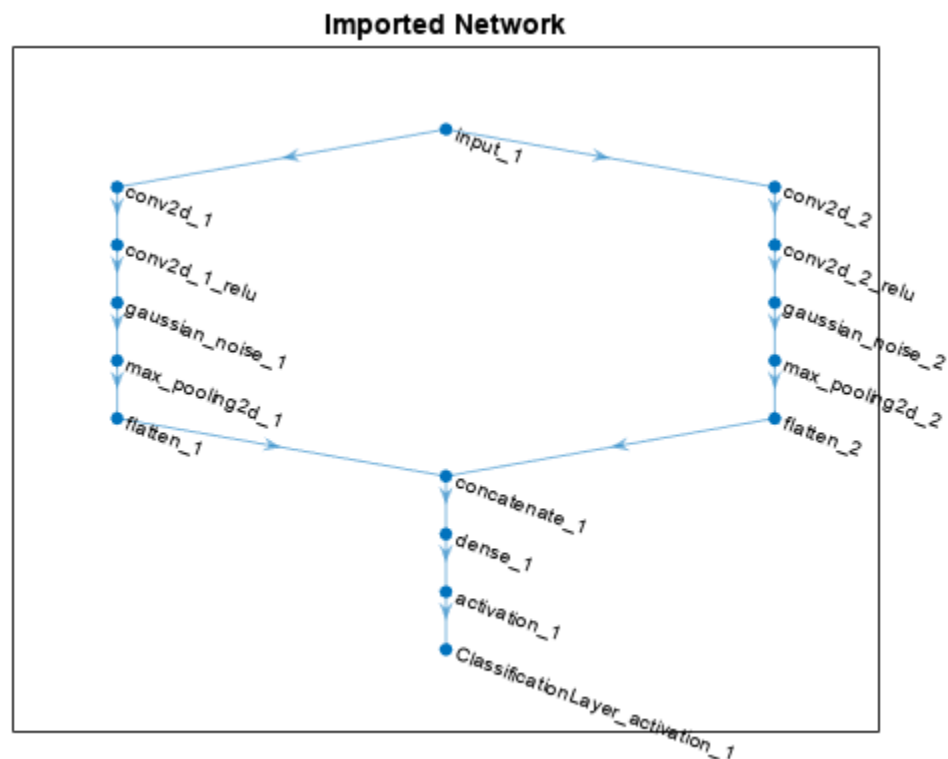
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename, 'ImportWeights', true);
```

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
```

Create two Gaussian noise layers with the same configurations as the imported Keras layers using the helper `gaussianNoiseLayer` function.

```
gnLayer1 = gaussianNoiseLayer(1.5, 'new_gaussian_noise_1');  
gnLayer2 = gaussianNoiseLayer(0.7, 'new_gaussian_noise_2');
```

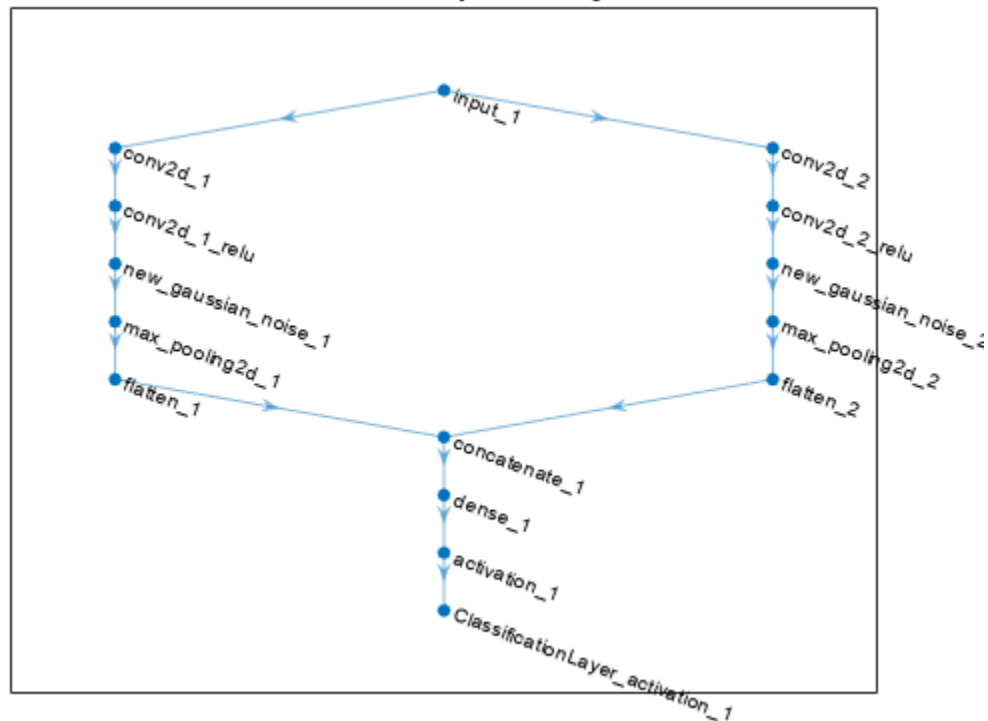
Replace the placeholder layers with the custom layers using `replaceLayer`.

```
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")
```

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'	2-D Convolution	20 7x7x1 convolutions with
3	'conv2d_1_relu'	ReLU	ReLU
4	'conv2d_2'	2-D 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'	2-D Max Pooling	2x2 max pooling with strid
9	'max_pooling2d_2'	2-D Max Pooling	2x2 max pooling with strid
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  
Layer array | LayerGraph object

Network architecture imported from Keras or ONNX, 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.

## Version History

**Introduced in R2017b**

## See Also

`importKerasLayers` | `PlaceholderLayer` | `replaceLayer` | `assembleNetwork` | `importONNXLayers` | `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”

# 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.

## 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 the name ' '.

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')

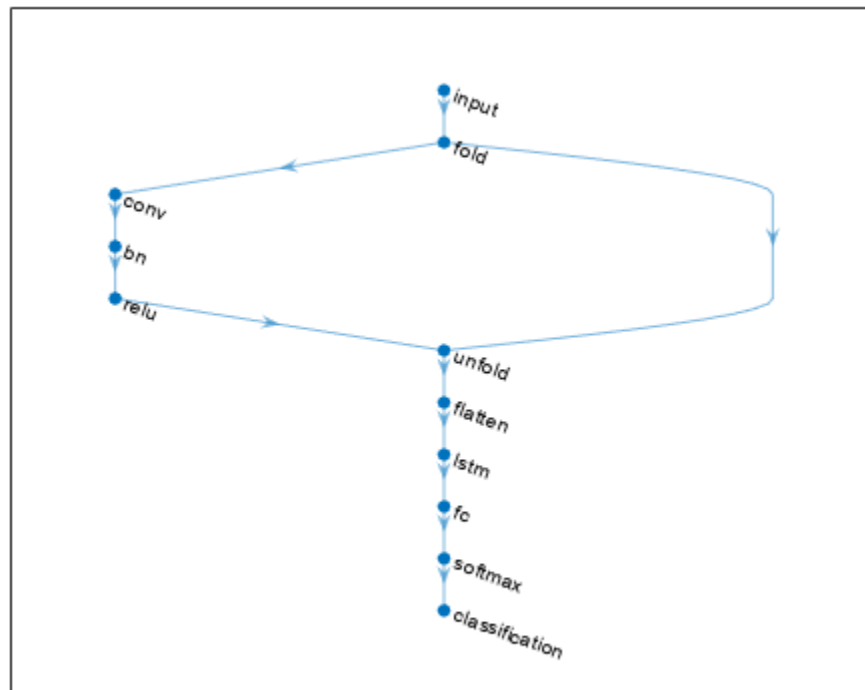
    fullyConnectedLayer(numClasses, 'Name', 'fc')
    softmaxLayer('Name', 'softmax')
    classificationLayer('Name', 'classification')];
```

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)
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `FlattenLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `dLarray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	
"SSCB" (spatial, spatial, channel, batch)	
"SSSCB" (spatial, spatial, channel, batch)	
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	
"SSCBT" (spatial, spatial, channel, batch, time)	
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	
"SSC" (spatial, spatial, channel)	
"SSSC" (spatial, spatial, spatial, channel)	

In `dlnetwork` objects, `FlattenLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	
"SSCT" (spatial, spatial, channel, time)	
"SSSCT" (spatial, spatial, spatial, channel, time)	

## Version History

Introduced in R2019a

## 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

`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 Neural Networks"
- "Deep Learning in MATLAB"
- "List of Deep Learning Layers"

## forward

Compute deep learning network output for training

### Syntax

```
Y = forward(net,X)
Y = forward(net,X1,...,XM)
[Y1,...,YN] = forward(____)
[Y1,...,YK] = forward(____,'Outputs',layerNames)
[____] = forward(____,'Acceleration',acceleration)
[____,state] = forward(____)
[____,state,pruningActivations] = 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.

`Y = forward(net,X)` returns the network output `Y` during training given the input data `X`.

`Y = forward(net,X1,...,XM)` returns the network output `Y` during training given the `M` inputs `X1, ..., XM` and the network `net` that has `M` inputs and a single output.

`[Y1,...,YN] = forward(____)` returns the `N` outputs `Y1, ..., YN` during training for networks that have `N` outputs using any of the previous syntaxes.

`[Y1,...,YK] = forward(____,'Outputs',layerNames)` returns the outputs `Y1, ..., YK` 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.

`[____,state,pruningActivations] = forward(____)` also returns a cell array of activations of the pruning layers. This syntax is applicable only if `net` is a `TaylorPrunableNetwork` object.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

### 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.

You can train most types of neural networks using the `trainNetwork` and `trainingOptions` functions. If the `trainingOptions` function does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using `dlarray` and `dlnetwork` objects for automatic differentiation. For an example showing how to retrain a pretrained deep learning network using the `trainNetwork` function, see “Transfer Learning Using Pretrained Network”.

Training a deep neural network is an optimization task. By considering a neural network as a function  $f(X; \theta)$ , where  $X$  is the network input, and  $\theta$  is the set of learnable parameters, you can optimize  $\theta$  so that it minimizes some loss value based on the training data. For example, optimize the learnable parameters  $\theta$  such that for a given inputs  $X$  with a corresponding targets  $T$ , they minimize the error between the predictions  $Y = f(X; \theta)$  and  $T$ .

The loss function used depends on the type of task. For example:

- For classification tasks, you can minimize the cross entropy error between the predictions and targets.
- For regression tasks, you can minimize the mean squared error between the predictions and targets.

You can optimize the objective using gradient descent: minimize the loss  $L$  by iteratively updating the learnable parameters  $\theta$  by taking steps towards the minimum using the gradients of the loss with respect to the learnable parameters. Gradient descent algorithms typically update the learnable parameters by using a variant of an update step of the form  $\theta_{t+1} = \theta_t - \rho \nabla L$ , where  $t$  is the iteration number,  $\rho$  is the learning rate, and  $\nabla L$  denotes the gradients (the derivatives of the loss with respect to the learnable parameters).

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  $\rho_t = \frac{\rho_0}{1 + kt}$ , where  $t$  is the iteration number,  $\rho_0$  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=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);
```

Determine the number of classes in the training data.

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

### Define Network

Define the network for image classification.

- For image input, specify an image input layer with input size matching the training data.
- Do not normalize the image input, set the `Normalization` option of the input layer to "none".
- Specify three convolution-batchnorm-ReLU blocks.
- Pad the input to the convolution layers such that the output has the same size by setting the `Padding` option to "same".
- For the first convolution layer specify 20 filters of size 5. For the remaining convolution layers specify 20 filters of size 3.
- For classification, specify a fully connected layer with size matching the number of classes
- To map the output to probabilities, include a softmax layer.

When training a network using a custom training loop, do not include an output layer.

```
layers = [  
    imageInputLayer(inputSize, Normalization="none")  
    convolution2dLayer(5, 20, Padding="same")  
    batchNormalizationLayer  
    reluLayer  
    convolution2dLayer(3, 20, Padding="same")  
    batchNormalizationLayer  
    reluLayer  
    convolution2dLayer(3, 20, Padding="same")  
    batchNormalizationLayer  
    reluLayer  
    fullyConnectedLayer(numClasses)  
    softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers)
```



```
net =
  dlnetwork with properties:

    Layers: [12x1 nnet.cnn.layer.Layer]
    Connections: [11x2 table]
    Learnables: [14x3 table]
    State: [6x3 table]
    InputNames: {'imageinput'}
    OutputNames: {'softmax'}
    Initialized: 1
```

View summary with `summary`.

## Define Model Loss Function

Training a deep neural network is an optimization task. By considering a neural network as a function  $f(X; \theta)$ , where  $X$  is the network input, and  $\theta$  is the set of learnable parameters, you can optimize  $\theta$  so that it minimizes some loss value based on the training data. For example, optimize the learnable parameters  $\theta$  such that for a given inputs  $X$  with a corresponding targets  $T$ , they minimize the error between the predictions  $Y = f(X; \theta)$  and  $T$ .

Create the function `modelLoss`, listed in the Model Loss Function on page 1-747 section of the example, that takes as input the `dlnetwork` object, a mini-batch of input data with corresponding targets, and returns the loss, the gradients of the loss with respect to the learnable parameters, and the network state.

## 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 format 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 Computing Requirements" (Parallel Computing Toolbox).

```
mbq = minibatchqueue(augimdsTrain,...  
    MiniBatchSize=miniBatchSize,...  
    MiniBatchFcn=@preprocessMiniBatch,...  
    MiniBatchFormat=["SSCB" ""]);
```

Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Calculate the total number of iterations for the training progress monitor.

```
numObservationsTrain = numel(imdsTrain.Files);  
numIterationsPerEpoch = ceil(numObservationsTrain / miniBatchSize);  
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info=["Epoch","LearnRate"],XLabel="Iteration");
```

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 loss, gradients, and state using the `dlfeval` and `modelLoss` 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.
- Update the loss, learn rate, and epoch values in the training progress monitor.
- Stop if the `Stop` property is true. The `Stop` property value of the `TrainingProgressMonitor` object changes to true when you click the `Stop` button.

```
epoch = 0;  
iteration = 0;
```

```
% Loop over epochs.
```

```
while epoch < numEpochs && ~monitor.Stop
```

```
    epoch = epoch + 1;
```

```
    % Shuffle data.
```

```
    shuffle(mbq);
```

```
    % Loop over mini-batches.
```

```
    while hasdata(mbq) && ~monitor.Stop
```

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

```
        % Read mini-batch of data.
```

```
        [X,T] = next(mbq);
```

```
        % Evaluate the model gradients, state, and loss using dlfeval and the  
        % modelLoss function and update the network state.
```

```
        [loss,gradients,state] = dlfeval(@modelLoss,net,X,T);  
        net.State = state;
```

```
        % Determine learning rate for time-based decay learning rate schedule.
```

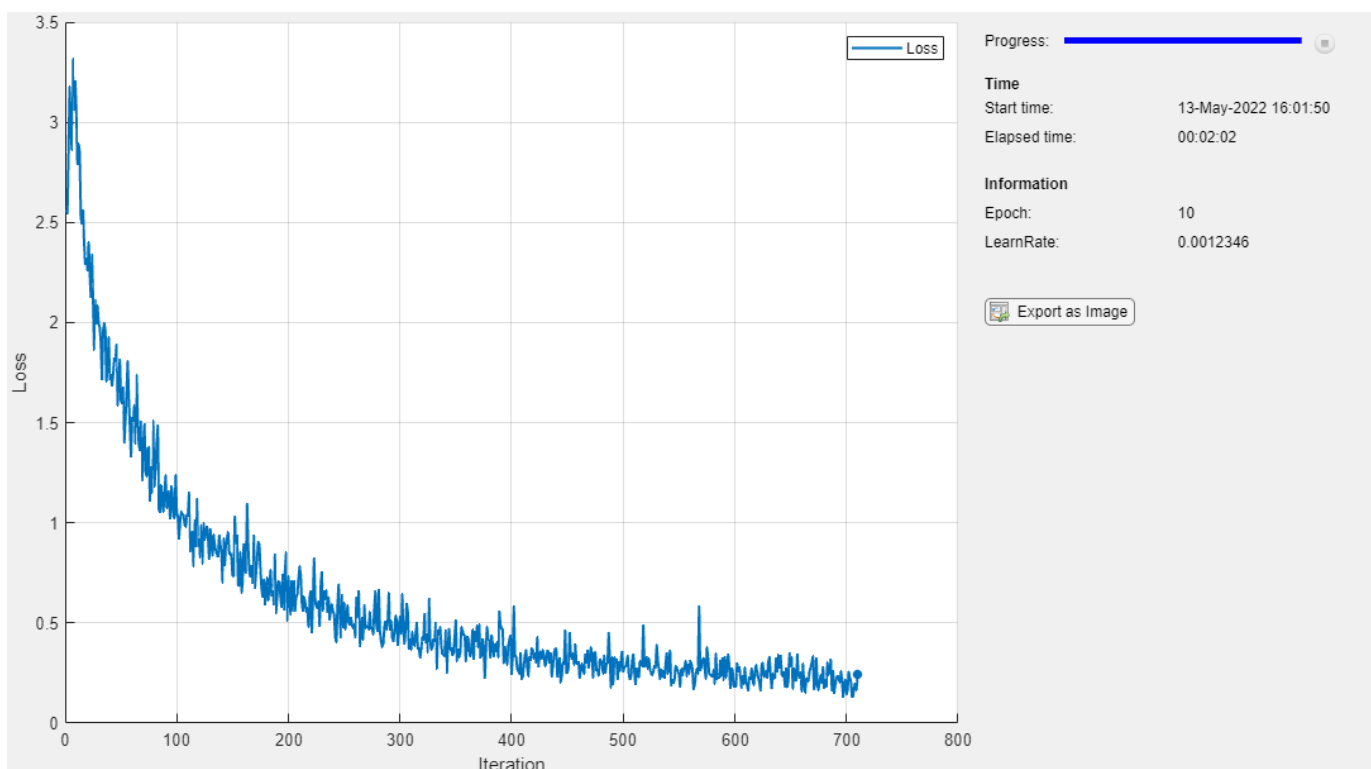
```

learnRate = initialLearnRate/(1 + decay*iteration);

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

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch,LearnRate=learnRate);
monitor.Progress = 100 * iteration/numIterations;
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.

```
YTest = modelPredictions(net,mbqTest,classes);
```

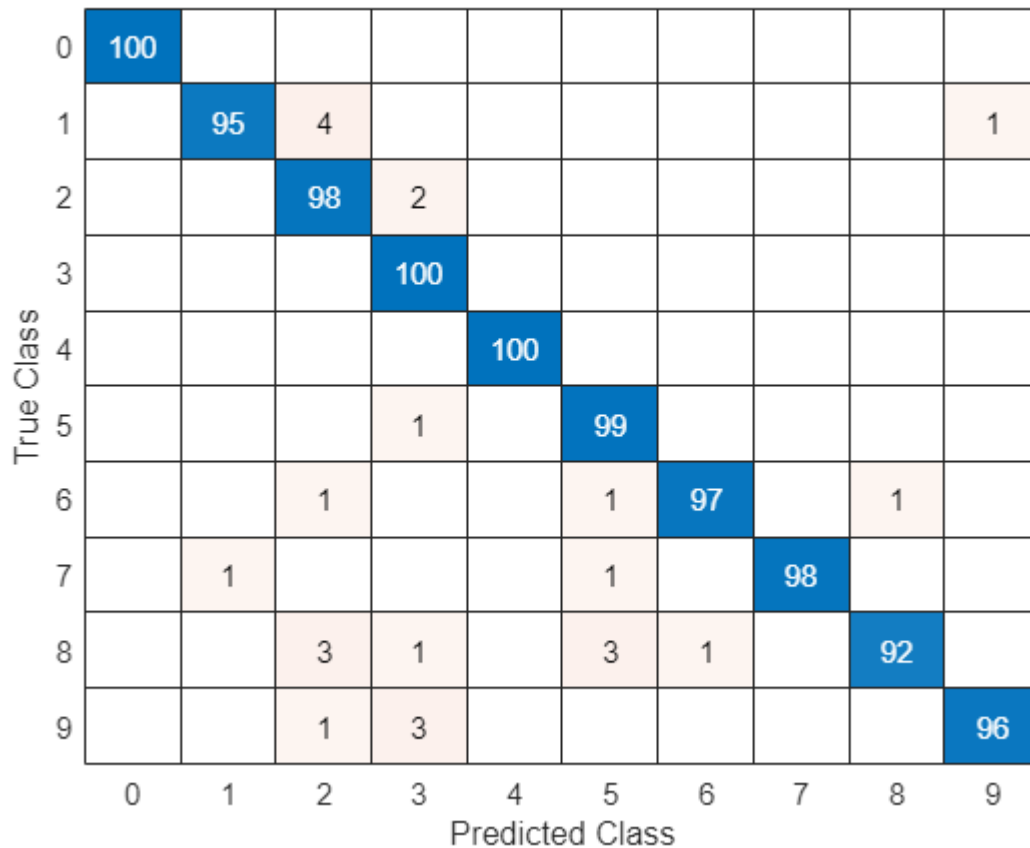
Evaluate the classification accuracy.

```
TTest = imdsValidation.Labels;
accuracy = mean(TTest == YTest)
```

```
accuracy = 0.9750
```

Visualize the predictions in a confusion chart.

```
figure
confusionchart(TTest,YTest)
```



Large values on the diagonal indicate accurate predictions for the corresponding class. Large values on the off-diagonal indicate strong confusion between the corresponding classes.

## Supporting Functions

### Model Loss Function

The `modelLoss` function takes a `dlnetwork` object `net`, a mini-batch of input data `X` with corresponding targets `T` and returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients automatically, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)

% Forward data through network.
[Y,state] = forward(net,X);

% Calculate cross-entropy loss.
loss = crossentropy(Y,T);

% Calculate gradients of loss with respect to learnable parameters.
gradients = dlgradient(loss,net.Learnables);

end
```

### Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `net`, 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 Y = modelPredictions(net,mbq,classes)

Y = [];

% Loop over mini-batches.
while hasdata(mbq)
    X = next(mbq);

    % Make prediction.
    scores = predict(net,X);

    % Decode labels and append to output.
    labels = onehotdecode(scores,classes,1)';
    Y = [Y; labels];
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,T] = preprocessMiniBatch(dataX,dataT)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(dataX);

% Extract label data from cell and concatenate.
T = cat(2,dataT{1:end});

% One-hot encode labels.
T = onehotencode(T,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(dataX)

% Concatenate.
X = cat(4,dataX{1:end});

end
```

## Input Arguments

### **net** — Network for custom training loops or custom pruning loops

`dlnetwork` object | `TaylorPrunableNetwork` object

This argument can represent either of these:

- Network for custom training loops, specified as a `dlnetwork` object.
- Network for custom pruning loops, specified as a `TaylorPrunableNetwork` object.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

### **X** — 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**

### **Y — 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.

### **pruningActivations — Activations of the pruning layers**

cell array containing `dLarray` objects

Cell array of activations of the pruning layers, if the input network is a `TaylorPrunableNetwork` object.

## Version History

### Introduced in R2019b

#### **R2021a: 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,net.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 `net.Learnables.Value` are `dlarray` objects with underlying data of type `gpuArray`
  - The input argument `X` 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` | `TaylorPrunableNetwork`

### Topics

“Train Generative Adversarial Network (GAN)”

“Automatic Differentiation Background”

“Define Custom Training Loops, Loss Functions, and Networks”



# 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 1-755.

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 = onhotencode(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")
```

Function containing the imported ONNX network architecture was saved to the file `squeezenetFcn.m`. To learn how to use this function, type: `help squeezenetFcn`.

```
params =
  ONNXParameters with properties:

    Learnables: [1x1 struct]
    Nonlearnables: [1x1 struct]
    State: [1x1 struct]
    NumDimensions: [1x1 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: [1x1x512x1000 darray]
```

```
conv10_B: [1000x1 darray]
```

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");
```

The network is ready for training. Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
```

```
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Calculate the total number of iterations for the training progress monitor.

```
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
numIterations = numEpochs*numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object immediately after the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

Train the network.

```
epoch = 0;
iteration = 0;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
while epoch < numEpochs && ~monitor.Stop

    epoch = epoch + 1;

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

    % Loop over mini-batches.
    i = 0;
    while i < numIterationsPerEpoch && ~monitor.Stop
        i = i + 1;
        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;

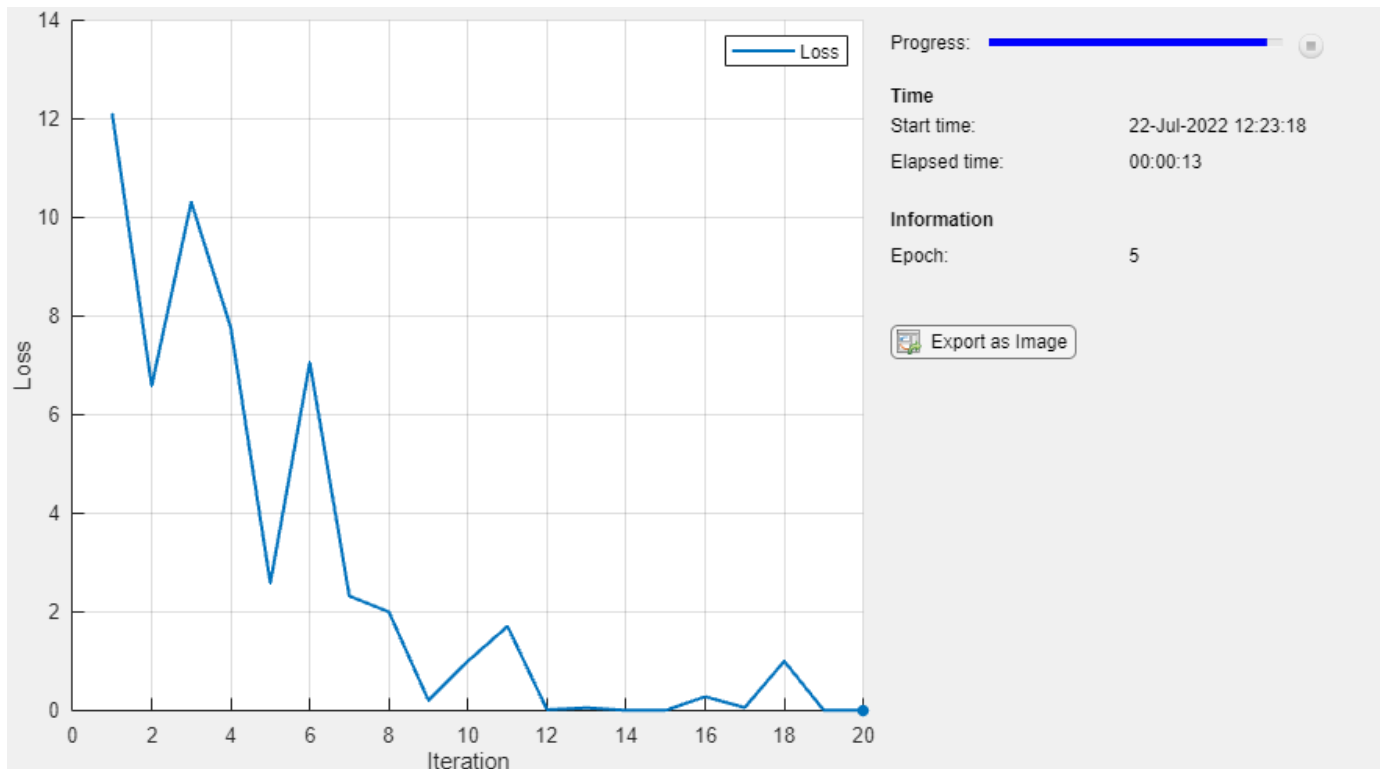
        % Determine the learning rate for the time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);
    end
end
```

```

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch);
monitor.Progress = 100 * iteration/numIterations;
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`.

## Version History

**Introduced in R2020b**

## See Also

[importONNXFunction](#) | [ONNXParameters](#) | [unfreezeParameters](#)

## fullyconnect

Sum all weighted input data and apply a bias

### Syntax

```
Y = fullyconnect(X,weights,bias)
Y = fullyconnect(X,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 `darray` data. If you want to apply the fully connect operation within a `layerGraph` object or `Layer` array, use the following layer:

- `fullyConnectedLayer`
- 

`Y = fullyconnect(X,weights,bias)` computes the weighted sum of the spatial, channel, and unspecified data in `X` using the weights specified by `weights`, and adds a bias. The input `X` must be a formatted `darray`. The output `Y` is a formatted `darray`.

`Y = fullyconnect(X,weights,bias,'DataFormat',FMT)` also specifies the dimension format `FMT` when `X` is not a formatted `darray`. The output `Y` is an unformatted `darray`.

### 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);
X = darray(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.

```
Y = fullyconnect(X,weights,bias);
```

```
Y =
    10(C) × 1(B) dIarray
    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 Y is a 2-D dIarray with one channel dimension of size ten and one singleton batch dimension.

## Input Arguments

### X — Input data

dIarray | numeric array

Input data, specified as a formatted dIarray, an unformatted dIarray, or a numeric array. When X is not a formatted dIarray, you must specify the dimension label format using 'DataFormat', FMT. If X is a numeric array, at least one of weights or bias must be a dIarray.

The fullyconnect operation sums over the 'S', 'C', and 'U' dimensions of X for each output feature specified by weights. The size of each 'B' or 'T' dimension of X is preserved.

Data Types: single | double

### weights — Weights

dIarray | numeric array

Weights, specified as a formatted dIarray, an unformatted dIarray, or a numeric array.

If weights is an unformatted dIarray or a numeric array, the first dimension of weights must match the number of output features. If weights is a formatted dIarray, 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 X multiplied by the number of output features.

Data Types: single | double

### bias — Bias constant

dIarray vector | numeric vector

Bias constant, specified as a formatted dIarray, an unformatted dIarray, 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 `darray`, 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 `darray`.

Example: 'DataFormat', 'SSCB'

Data Types: `char` | `string`

## **Output Arguments**

### **Y — Weighted output features**

`darray`

Weighted output features, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input X is a formatted `darray`, the output Y has one dimension labeled 'C' representing the output features, and the same number of 'B' or 'T' dimensions as the input X, if either or both are present. If X has no 'B' or 'T' dimensions, Y has the format 'CB', where the 'B' dimension is singleton.

If the input X is not a formatted `darray`, output Y is unformatted. The first dimension of Y contains the output features. Other dimensions of Y correspond to the 'B' and 'T' dimensions of X, if either or both are present, and are provided in the same order as in FMT. If X has no 'B' or 'T' dimensions, the first dimension of Y 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 1-768 on the `fullyConnectedLayer` reference page.

## Version History

Introduced in R2019b

## 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:
  - `X`
  - `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”

# 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 1-762, “Learning Rate and Regularization” on page 1-764, 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  $2/(\text{InputSize} + \text{OutputSize})$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2/\text{InputSize}$ .
- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software

determines the global  $L_2$  regularization factor based on the settings you specify 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 the name `''`.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D 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  $X$ .

Fully connected layers flatten the output. They encode the spatial data in the channel dimension by reshaping the output data. Fully connected layers in `SeriesNetwork` and `DAGNetwork` objects output data with the same number of the spatial dimensions as the input by outputting data with spatial dimensions of size one. Fully connected layers in `dlnetwork` objects remove the spatial dimensions of the output.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `FullyConnectedLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format	
	SeriesNetwork and DAGNetwork Objects	dlnetwork Objects
"CB" (channel, batch)	"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)	
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)	
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)	
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SC" (spatial, channel)	"CB" (channel, batch)	"CB" (channel, batch)
"SSC" (spatial, spatial, channel)		
"SSSC" (spatial, spatial, spatial, channel, batch)		

In `dlnetwork` objects, `FullyConnectedLayer` objects also support the following input and output format combinations.

Input Format	Output Format
"SCBT" (spatial, channel, batch, time)	"CBT" (channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	
"SSCT" (spatial, spatial, channel, time)	
"SSSCT" (spatial, spatial, spatial, channel, time)	

To use these input formats in `trainNetwork` workflows, first convert the data to "CBT" (channel, batch, time) format using `flattenLayer`.

## Version History

### Introduced in R2016a

#### R2019a: 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>
- [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 Neural 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”](#)

# 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 1-772 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 the syntax  $[Y1, \dots, YM] = \text{fun}(X1, \dots, XN)$ , where the inputs and outputs are `darray` objects, and  $M$  and  $N$  correspond to the `NumOutputs` and `NumInputs` properties, respectively.

The inputs  $X1, \dots, XN$  correspond to the layer inputs with names given by `InputNames`. The outputs  $Y1, \dots, YM$  correspond to the layer outputs with names given by `OutputNames`.

For a list of functions that support `darray` input, see “List of Functions with `darray` 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 whether the layer function operates on formatted `dlarray` objects, specified as 0 (false) or 1 (true).

Data Types: `logical`

**Acceleratable — Flag indicating that function supports acceleration**

0 (false) (default) | 1 (true)

This property is read-only.

Flag indicating whether the layer function supports acceleration using `dlaccelerate`, specified as 0 (false) or 1 (true).

---

**Tip** Setting `Acceleratable` to 1 (true) can significantly improve the performance of training and inference (prediction) using a `dlnetwork`.

Most simple functions support acceleration using `dlaccelerate`. For more information, see “Deep Learning Function Acceleration for Custom Training Loops”.

---

Data Types: `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 the name ' '.

Data Types: `char` | `string`

**Description — One-line description of 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(PredictFcn)`.

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
    Acceleratable: 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 the 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  ''  2-D Convolution       20 5x5 convolutions with stride [1 1] and padding [0 0 0]
   3  ''  Function              softsign
   4  ''  2-D 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 the format "CB" (channel, batch) to have the format "SBC" (spatial, batch, channel). To specify that the layer operates on formatted data, set the `Formattable` option to `true`. To specify that the layer function supports acceleration using `dlaccelerate`, set the `Acceleratable` option to `true`.

```
layer = functionLayer(@(X) dlarray(X, "SBC"),Formattable=true,Acceleratable=true)
```

```
layer =
  FunctionLayer with properties:
      Name: ''
      PredictFcn: @(X)dlarray(X,"SBC")
      Formattable: 1
      Acceleratable: 1

  Learnable Parameters
  No properties.

  State Parameters
  No properties.

  Show all properties
```

Include a function layer that reformats the input to have the format "SB" in a layer array. Set the layer description to "channel to spatial".

```
layers = [
  featureInputLayer(10)
  functionLayer(@(X) dlarray(X, "SBC"),Formattable=true,Acceleratable=true,Description="channel
  convolution1dLayer(3,16)]
```

```
layers =
  3x1 Layer array with layers:

   1  ''  Feature Input      10 features
   2  ''  Function          channel to spatial
   3  ''  1-D 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 operation 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 the 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 = 1x3
      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 supported.
```

```
Warning: Unable to import layer. Keras layer 'Activation' with the specified settings is not supported.
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning Toolbox.
```

```
layers =
    13x1 Layer array with layers:
```

1	'ImageInputLayer'	Image Input	28x28x1 images
2	'conv2d'	2-D Convolution	8 3x3x1 convolutions with stride [2 2]
3	'conv2d_softsign'	PLACEHOLDER LAYER	Placeholder for 'Activation' Keras layer
4	'max_pooling2d'	2-D Max Pooling	2x2 max pooling with stride [2 2]
5	'conv2d_1'	2-D Convolution	16 3x3x8 convolutions with stride [2 2]
6	'conv2d_1_softsign'	PLACEHOLDER LAYER	Placeholder for 'Activation' Keras layer
7	'max_pooling2d_1'	2-D Max Pooling	2x2 max pooling with stride [2 2]
8	'flatten'	Keras Flatten	Flatten activations into 1-D array
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'              2-D Convolution 8 3x3x1 convolutions with stride
   3  'layer'              Function         softsign
   4  'max_pooling2d'      2-D Max Pooling 2x2 max pooling with stride [2
   5  'conv2d_1'          2-D Convolution 16 3x3x8 convolutions with stri
   6  'layer_1'           Function         softsign
   7  'max_pooling2d_1'   2-D 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)
```

0	498			1					1	
1		489	1	1		3		6		
2	1	1	498							
3		1		493		1			4	1
4	1				496		3			
5				5		495				
6	2	2				1	491		4	
7		1	1	1				497		
8			2	1		1			496	
9				1					2	497
	0	1	2	3	4	5	6	7	8	9

Predicted Class

## Version History

Introduced in R2021b

### Extended Capabilities

#### C/C++ Code Generation

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

Usage notes and limitations:

- The layer function `fun` must be a named function on the path.
- The `Formattable` property must be `0` (false)

#### GPU Code Generation

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

Usage notes and limitations:

- The layer function `fun` must be a named function on the path.
- The `Formattable` property must be `0` (false)

**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”](#)

## functionToLayerGraph

(To be removed) Convert deep learning model function to a layer graph

---

**Note** `functionToLayerGraph` will be removed in a future release. Construct layer graphs manually instead. For a list of layers, see “List of Deep Learning Layers”. For information about developing custom layers, see “Define Custom Deep Learning Layers”.

---

### 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 1-782.

`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
X1 = dlarray(rand(10), 'CB');
X2 = dlarray(zeros(10,1), 'CB');
fun = @(x) simplemodel(x,X1,X2);
```

Call `functionToLayerGraph` using a `dlarray` for the input data `X`.

```
X = dlarray(ones(10,1), 'CB');
lgraph = functionToLayerGraph(fun,X)
```

```
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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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.

## **Limitations**

`functionToLayerGraph` does not support these deep learning operations:

- `gru`
- `lstm`

If you use `functionToLayerGraph` with a function that contains these operations, then the resulting layer graph contains placeholder layers.

## **Version History**

**Introduced in R2019b**

**R2023a: To be removed**

*Warns starting in R2023a*

`functionToLayerGraph` will be removed in a future release. Construct layer graphs manually instead. For a list of layers, see “List of Deep Learning Layers”. For information about developing custom layers, see “Define Custom Deep Learning Layers”.

## **See Also**

`layerGraph` | `findPlaceholderLayers` | `PlaceholderLayer` | `dlarray`

**Topics**

“List of Deep Learning Layers”

“Define Custom Deep Learning Layers”

“List of Functions with dlarray Support”

## gelu

Apply Gaussian error linear unit (GELU) activation

### Syntax

```
Y = gelu(X)
Y = gelu(X, Approximation=method)
```

### Description

The Gaussian error linear unit (GELU) activation operation weights the input by its probability under a Gaussian distribution.

This operation is given by

$$\text{GELU}(x) = \frac{x}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right),$$

where erf denotes the error function.

---

**Note** This function applies the GELU operation to `darray` data. If you want to apply the GELU activation within a `layerGraph` object or `Layer` array, use the following layer:

- `geluLayer`
- 

`Y = gelu(X)` applies the GELU activation to the input data `X`.

`Y = gelu(X, Approximation=method)` also specifies the approximation method for the GELU operation. For example, `Approximation="tanh"` specifies the tanh approximation of the underlying error function.

## Examples

### Apply GELU Operation

Create a formatted `darray` object containing a batch of 128 28-by-28 images with three channels. Specify the format "SSCB" (spatial, spatial, channel, batch).

```
miniBatchSize = 128;
inputSize = [28 28];
numChannels = 3;
X = rand(inputSize(1), inputSize(2), numChannels, miniBatchSize);
X = darray(X, "SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
      28      28      3      128
```

```
dims(X)
```

```
ans =
'SSCB'
```

Apply the GELU activation.

```
Y = gelu(X);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
      28      28      3      128
```

```
dims(Y)
```

```
ans =
'SSCB'
```

## Input Arguments

### X — Input data

d<sub>l</sub>array object

Input data, specified as a formatted or unformatted d<sub>l</sub>array object.

### method — Approximation method

"none" (default) | "tanh"

Approximation method, specified as one of these values:

- "none" — Do not use approximation.
- "tanh" — Approximate the underlying error function using

$$\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \approx \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right).$$

---

**Tip** In MATLAB, computing the tanh approximation is typically less accurate, and, for large input sizes, slower than computing the GELU activation without using an approximation. Use the tanh approximation when you want to reproduce models that use this approximation, such as BERT and GPT-2.

---

## Output Arguments

### Y — GELU activations

`darray` object

GELU activations, returned as a `darray` object. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray` object, then Y has the same dimension format as X. If the input data is not a formatted `darray` object, then Y is an unformatted `darray` object with the same dimension order as the input data.

## Algorithms

### Gaussian Error Linear Unit Activation

The Gaussian error linear unit (GELU) activation operation weights the input by its probability under a Gaussian distribution.

This operation is given by

$$\text{GELU}(x) = \frac{x}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right),$$

where `erf` denotes the error function given by

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

When the Approximation option is "tanh", the software approximates the error function using

$$\text{erf} \left( \frac{x}{\sqrt{2}} \right) \approx \tanh \left( \sqrt{\frac{2}{\pi}} (x + 0.044715x^3) \right).$$

## Version History

**Introduced in R2022b**

## References

[1] Hendrycks, Dan, and Kevin Gimpel. "Gaussian error linear units (GELUs)." Preprint, submitted June 27, 2016. <https://arxiv.org/abs/1606.08415>

## 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 X is a `darray` 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](#) | [relu](#) | [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”

## geluLayer

Gaussian error linear unit (GELU) layer

### Description

A Gaussian error linear unit (GELU) layer weights the input by its probability under a Gaussian distribution.

This operation is given by

$$\text{GELU}(x) = \frac{x}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right),$$

where erf denotes the error function.

### Creation

#### Syntax

```
layer = geluLayer  
layer = geluLayer(Name=Value)
```

#### Description

`layer = geluLayer` returns a GELU layer.

`layer = geluLayer(Name=Value)` sets the optional `Approximation` and `Name` properties using name-value arguments. For example, `geluLayer(Name="gelu")` creates a GELU layer with the name "gelu".

### Properties

#### GELU

##### Approximation — Approximation method for GELU operation

'none' (default) | 'tanh'

Approximation method for the GELU operation, specified as one of these values:

- 'none' — Do not use approximation.
- 'tanh' — Approximate the underlying error function using

$$\text{erf} \left( \frac{x}{\sqrt{2}} \right) \approx \tanh \left( \sqrt{\frac{2}{\pi}} (x + 0.044715x^3) \right).$$

---

**Tip** In MATLAB, computing the tanh approximation is typically less accurate, and, for large input sizes, slower than computing the GELU activation without using an approximation. Use the tanh



approximation when you want to reproduce models that use this approximation, such as BERT and GPT-2.

---

## 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 the name ' '.

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 GELU Layer

Create a GELU layer.

```

layer = geluLayer

layer =
  GELULayer with properties:
      Name: ''
      Hyperparameters
      Approximation: 'none'
  
```

Include a GELU layer in a Layer array.

```

layers = [
  imageInputLayer([28 28 1])
  convolution2dLayer(5,20)
  geluLayer
  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 0]
3	''	GELU	GELU
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

## Algorithms

### Gaussian Error Linear Unit Activation

The Gaussian error linear unit (GELU) activation operation weights the input by its probability under a Gaussian distribution.

This operation is given by

$$\text{GELU}(x) = \frac{x}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right),$$

where erf denotes the error function given by

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

When the Approximation option is "tanh", the software approximates the error function using

$$\text{erf} \left( \frac{x}{\sqrt{2}} \right) \approx \tanh \left( \sqrt{\frac{2}{\pi}} (x + 0.044715x^3) \right).$$

## Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

GELULayer objects apply an element-wise operation and supports input data of any format. The layer does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

Introduced in R2022b

## References

[1] Hendrycks, Dan, and Kevin Gimpel. "Gaussian error linear units (GELUs)." Preprint, submitted June 27, 2016. <https://arxiv.org/abs/1606.08415>

## See Also

`trainNetwork` | `fullyConnectedLayer` | `imageInputLayer` | `sequenceInputLayer`

## Topics

"Train Network with Numeric Features"

"Create Simple Deep Learning Neural Network for Classification"

"Train Convolutional Neural Network for Regression"

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

## getL2Factor

**Package:** `nnet.cnn.layer`

Get L2 regularization factor of layer learnable parameter

### Syntax

```
factor = getL2Factor(layer, parameterName)
factor = getL2Factor(layer, parameterPath)

factor = getL2Factor(net, layerName, parameterName)
factor = getL2Factor(net, 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(net, 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(net, 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
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
      Network: [1x1 dlnetwork]
```

```
    Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:
```

1	'conv_1'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
2	'batchnorm_1'	Batch Normalization	Batch normalization
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
5	'batchnorm_2'	Batch Normalization	Batch normalization
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the L2 regularization factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the setL2Factor function.

```
factor = 2;
layer = setL2Factor(layer, 'Network/conv_1/Weights', factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(layer, 'Network/conv_1/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/conv_1/Weights"	{3x3x32x32 dlarray}
"res1"	"Network/conv_1/Bias"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_1/Offset"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_1/Scale"	{1x1x32 dlarray}
"res1"	"Network/conv_2/Weights"	{3x3x32x32 dlarray}
"res1"	"Network/conv_2/Bias"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_2/Offset"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_2/Scale"	{1x1x32 dlarray}

For the layer "res1", set the L2 regularization factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the setL2Factor function.

```

factor = 2;
dlnet = setL2Factor(dlnet,'res1/Network/conv_1/Weights',factor);

```

Get the updated L2 regularization factor using the getL2Factor function.

```
factor = getL2Factor(dlnet,'res1/Network/conv_1/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 net.

Data Types: char | string

**net — Neural network**`dlnetwork` object

Neural network, 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.

## Version History

**Introduced in R2017b**

### See Also

`setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `trainNetwork` | `trainingOptions`

### Topics

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“Define Custom Deep Learning Layers”

## getLearnRateFactor

**Package:** `nnet.cnn.layer`

Get learn rate factor of layer learnable parameter

### Syntax

```
factor = getLearnRateFactor(layer,parameterName)
factor = getLearnRateFactor(layer,parameterPath)

factor = getLearnRateFactor(net,layerName,parameterName)
factor = getLearnRateFactor(net,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(net,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(net,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
```

```
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
    Network: [1x1 dlnetwork]
```

```
Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:
```

1	'conv_1'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
2	'batchnorm_1'	Batch Normalization	Batch normalization
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
5	'batchnorm_2'	Batch Normalization	Batch normalization
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the learning rate factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the setLearnRateFactor function.

```
factor = 2;
layer = setLearnRateFactor(layer, 'Network/conv_1/Weights', factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(layer, 'Network/conv_1/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	'conv_1'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding [1 1]
2	'batchnorm_1'	Batch Normalization	Batch normalization with 32 channels
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding [1 1]
5	'batchnorm_2'	Batch Normalization	Batch normalization with 32 channels
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the learning rate factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the setLearnRateFactor function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'res1/Network/conv_1/Weights',factor);
```

Get the updated learning rate factor using the getLearnRateFactor function.

```
factor = getLearnRateFactor(dlnet,'res1/Network/conv_1/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 net.

Data Types: char | string

### **net — Neural network**

`dlnetwork` object

Neural network, 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.

## Version History

**Introduced in R2017b**

### See Also

`setLearnRateFactor` | `setL2Factor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

### Topics

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“Define Custom Deep Learning Layers”

# 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (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 the name ' '.

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,MinLength=20)
  convolution1dLayer(11,96)
  reluLayer
  globalAveragePooling1dLayer
  fullyConnectedLayer(10)
  softmaxLayer
  classificationLayer]
layers =
  7x1 Layer array with layers:
```

1	''	Sequence Input	Sequence input with 12 dimensions
2	''	1-D 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (spatial) dimension.

Global pooling layers remove the "T" (time) dimension when the pool over the "T" (time) dimension. Global pooling layers do not remove the "S" (spatial) dimension when the pool over the "S" (spatial) dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalAveragePooling1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SB" (spatial, batch)	"SB" (spatial, batch)

In `dlnetwork` objects, `GlobalAveragePooling1dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"BT" (batch, time)	"BU" (batch, unspecified)
"SBT" (spatial, batch, time)	"SBT" (spatial, batch, time)

## Version History

Introduced in R2021b

### 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 Neural Networks"  
 "List of Deep Learning Layers"  
 "Deep Learning Tips and Tricks"

# globalAveragePooling2dLayer

2-D 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.

The dimensions that the layer pools over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

## 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 the name ' '.

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 '' 2-D 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 number of channels in the input of the global average pooling layer must match the number of classes in the classification task.

## Algorithms

### 2-D Global Average Pooling Layer

A 2-D global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input.

The dimensions that the layer pools over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

Global pooling layers remove the "T" (time) dimension when the pool over the "T" (time) dimension. Global pooling layers do not remove the "S" (spatial) dimension when the pool over the "S" (spatial) dimension.

## Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalAveragePooling2dLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SCBT" (spatial, channel, batch, time)	"SCB" (spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSB" (spatial, spatial, batch)	"SSB" (spatial, spatial, batch)

In `dlnetwork` objects, `GlobalAveragePooling2dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SCT" (spatial, channel, time)	"SC" (spatial, channel)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SBT" (spatial, batch, time)	"SB" (spatial, batch)
"SSBT" (spatial, spatial, batch, time)	"SSBT" (spatial, spatial, batch, time)

## Version History

Introduced in R2019b

## **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 Neural Network for Classification”](#)

[“Train Convolutional Neural Network for Regression”](#)

[“Deep Learning in MATLAB”](#)

[“Specify Layers of Convolutional Neural Network”](#)

[“List of Deep Learning Layers”](#)



# 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.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

## 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 the name ' '.

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 '' 3-D Convolution          20 5x5x5 convolutions with stride [1 1 1] and padding
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 number of channels in the input of the global average pooling layer must match the number of classes in the classification task.

## Algorithms

### 3-D Global Average Pooling Layer

A 3-D global average pooling layer performs downsampling by computing the mean of the height, width, and depth dimensions of the input.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

Global pooling layers remove the "T" (time) dimension when the pool over the "T" (time) dimension. Global pooling layers do not remove the "S" (spatial) dimension when the pool over the "S" (spatial) dimension.

## Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalAveragePooling3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCB" (spatial, spatial, channel, batch)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"SSSB" (spatial, spatial, spatial, batch)	"SSSB" (spatial, spatial, spatial, batch)

In `dlnetwork` objects, `GlobalAveragePooling3DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCT" (spatial, spatial, channel, time)	"SSC" (spatial, spatial, channel)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)
"SSBT" (spatial, spatial, batch, time)	"SSB" (spatial, spatial, batch)
"SSSBT" (spatial, spatial, spatial, batch, time)	"SSSBT" (spatial, spatial, spatial, batch, time)

## Version History

Introduced in R2019b

## See Also

[averagePooling3dLayer](#) | [globalAveragePooling2dLayer](#) | [convolution3dLayer](#) | [maxPooling3dLayer](#) | [globalMaxPooling3dLayer](#)

## Topics

[“Deep Learning in MATLAB”](#)

[“Specify Layers of Convolutional Neural Network”](#)

[“List of Deep Learning Layers”](#)

# 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (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 the name ' '.

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,MinLength=20)
  convolution1dLayer(11,96)
  reluLayer
  globalMaxPooling1dLayer
  fullyConnectedLayer(10)
  softmaxLayer
  classificationLayer]
layers =
  7x1 Layer array with layers:
```

1	''	Sequence Input	Sequence input with 12 dimensions
2	''	1-D 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (spatial) dimension.

Global pooling layers remove the "T" (time) dimension when the pool over the "T" (time) dimension. Global pooling layers do not remove the "S" (spatial) dimension when the pool over the "S" (spatial) dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalMaxPooling1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)
"SB" (spatial, batch)	"SB" (spatial, batch)

In `dlnetwork` objects, `GlobalMaxPooling1dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"BT" (batch, time)	"BU" (batch, unspecified)
"SBT" (spatial, batch, time)	"SBT" (spatial, batch, time)

## Version History

Introduced in R2021b

### 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 Neural Networks"  
 "List of Deep Learning Layers"  
 "Deep Learning Tips and Tricks"

# 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 the name ' '.

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  ''  2-D 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

## Algorithms

### 2-D Global Max Pooling Layer

A 2-D global max pooling layer performs downsampling by computing the maximum of the height and width dimensions of the input.

The dimensions that the layer pools over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalMaxPooling2dLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)

In `dlnetwork` objects, `GlobalMaxPooling2dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

Introduced in R2020a

## 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 Neural Network for Classification”  
 “Train Convolutional Neural Network for Regression”  
 “Deep Learning in MATLAB”  
 “Specify Layers of Convolutional Neural Network”  
 “List of Deep Learning Layers”

# 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 the name ' '.

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  ''  3-D Convolution           20 5x5x5 convolutions with stride [1 1 1] and padding [
   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 `globalAveragePooling3dLayer` 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

## Algorithms

### 3-D Global Max Pooling Layer

A 3-D global max pooling layer performs downsampling by computing the maximum of the height, width, and depth dimensions of the input.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GlobalMaxPooling3dLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `GlobalMaxPooling3dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2020a

### See Also

[averagePooling3dLayer](#) | [globalMaxPooling2dLayer](#) | [convolution3dLayer](#) | [maxPooling3dLayer](#) | [globalAveragePooling3dLayer](#)

### Topics

"Deep Learning in MATLAB"  
 "Specify Layers of Convolutional Neural Network"  
 "List of Deep Learning Layers"

## 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 =
```

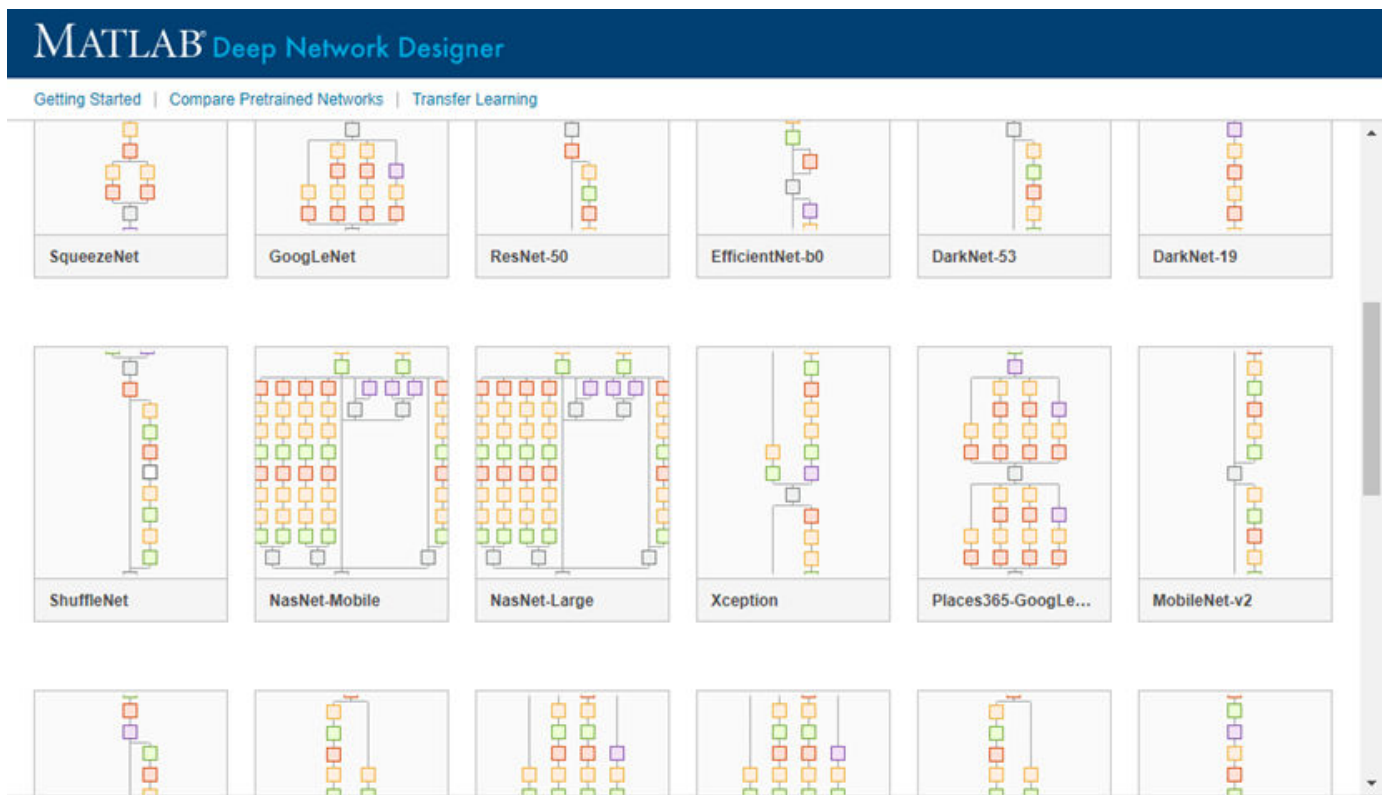
```
DAGNetwork with properties:
```

```
    Layers: [144x1 nnet.cnn.layer.Layer]  
    Connections: [170x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(googlenet)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2017b

## 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](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”

## 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 input `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 the data.

The Grad-CAM interpretability technique uses the gradients of the classification score with respect to the final convolutional feature map. The parts of an observation 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 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

#### Grad-CAM for Image Classification

Use `gradCAM` to visualize which parts of an image are important to the classification decision of a network.

Import the pretrained network SqueezeNet.

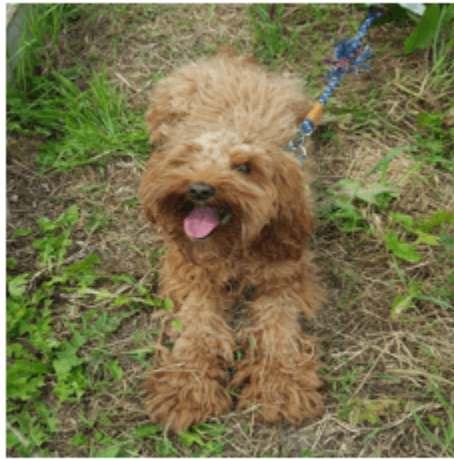
```
net = squeezeNet;
```

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)
```

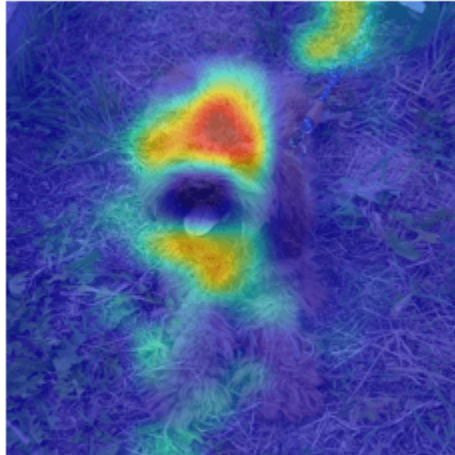
```
label = categorical  
toy poodle
```

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.

### Grad-CAM for Image Regression

Use Grad-CAM to visualize which parts of an image are most important to the predictions of an image regression network.

Load the pretrained network `digitsRegressionNet`. This network is a regression convolutional neural network that predicts the angle of rotation of handwritten digits.

load `digitsRegressionNet`

View the network layers. The output layer of the network is a regression layer.

```
layers = net.Layers
```

```
layers =
```

```
18x1 Layer array with layers:
```

1	'imageinput'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv_1'	2-D Convolution	8 3x3x1 convolutions with stride [1 1] and padding [1 1]
3	'batchnorm_1'	Batch Normalization	Batch normalization with 8 channels
4	'relu_1'	ReLU	ReLU
5	'avgpool2d_1'	2-D Average Pooling	2x2 average pooling with stride [2 2] and padding [1 1]
6	'conv_2'	2-D Convolution	16 3x3x8 convolutions with stride [1 1] and padding [1 1]
7	'batchnorm_2'	Batch Normalization	Batch normalization with 16 channels
8	'relu_2'	ReLU	ReLU
9	'avgpool2d_2'	2-D Average Pooling	2x2 average pooling with stride [2 2] and padding [1 1]
10	'conv_3'	2-D Convolution	32 3x3x16 convolutions with stride [1 1] and padding [1 1]
11	'batchnorm_3'	Batch Normalization	Batch normalization with 32 channels
12	'relu_3'	ReLU	ReLU
13	'conv_4'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding [1 1]



14	'batchnorm_4'	Batch Normalization	Batch normalization with 32 channels
15	'relu_4'	ReLU	ReLU
16	'dropout'	Dropout	20% dropout
17	'fc'	Fully Connected	1 fully connected layer
18	'regressionoutput'	Regression Output	mean-squared-error with response 'Response'

Load the test images.

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

Evaluate the performance of the network on a test image.

```
testIdx = 501;
testDigit = XTest(:,:,,testIdx);
```

Use `predict` to predict the angle of rotation and compare the predicted rotation to the true rotation.

```
predRotation = predict(net,testDigit)
```

```
predRotation = single
26.5635
```

```
trueRotation = YTest(testIdx)
```

```
trueRotation = 29
```

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_4';
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.

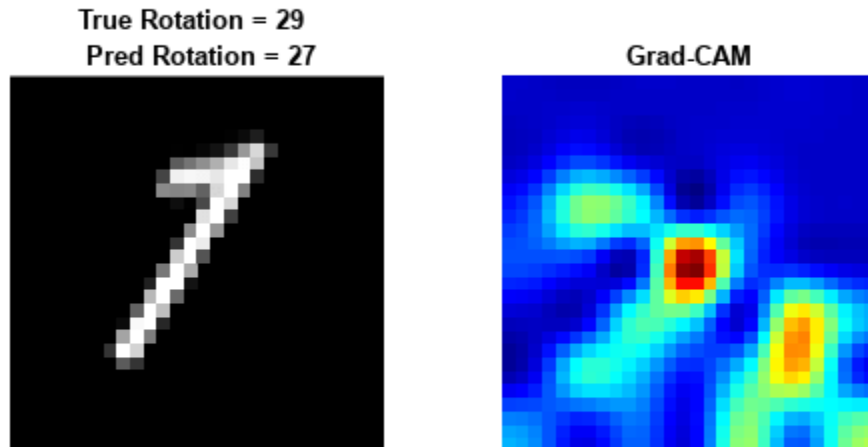
```
scoreMap = gradCAM(net,testDigit,reductionFcn, ...
'ReductionLayer',reductionLayer, ...
'FeatureLayer',featureLayer);
```

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(rescale(scoreMap))
colormap(ax(2),'jet')
```

```
title("Grad-CAM")
hold off
```



### Grad-CAM for Sequence Classification

Use gradCAM to visualize which parts of a sequence are important to the classification decision of a network.

Load the pretrained network `JapaneseVowelsConvNet`. This network is a pretrained 1-D convolutional neural network trained on the Japanese Vowels data set as described in [1] and [2].

```
load JapaneseVowelsConvNet
```

View the network architecture.

```
net.Layers
```

```
ans =
```

```
11x1 Layer array with layers:
```

1	'sequenceinput'	Sequence Input	Sequence input with 12 dimensions
2	'conv1d_1'	1-D Convolution	32 3x12 convolutions with stride 1 and
3	'relu_1'	ReLU	ReLU
4	'layernorm_1'	Layer Normalization	Layer normalization with 32 channels
5	'conv1d_2'	1-D Convolution	64 3x32 convolutions with stride 1 and

6	'relu_2'	ReLU	ReLU
7	'layernorm_2'	Layer Normalization	Layer normalization with 64 channels
8	'globalavgpool1d'	1-D Global Average Pooling	1-D global average pooling
9	'fc'	Fully Connected	9 fully connected layer
10	'softmax'	Softmax	softmax
11	'classoutput'	Classification Output	crossentropyex with '1' and 8 other cl

Load the test data. `XTest` is a cell array containing sequences of dimension 12 of varying length. `TTest` is a categorical vector of labels "1","2",..., "9", which correspond to the nine speakers.

```
[XTest,TTest] = japaneseVowelsTestData;
```

Select the first time series observation. This sequence has 19 time steps.

```
testSequence = XTest{1};
testSize = size(testSequence)
```

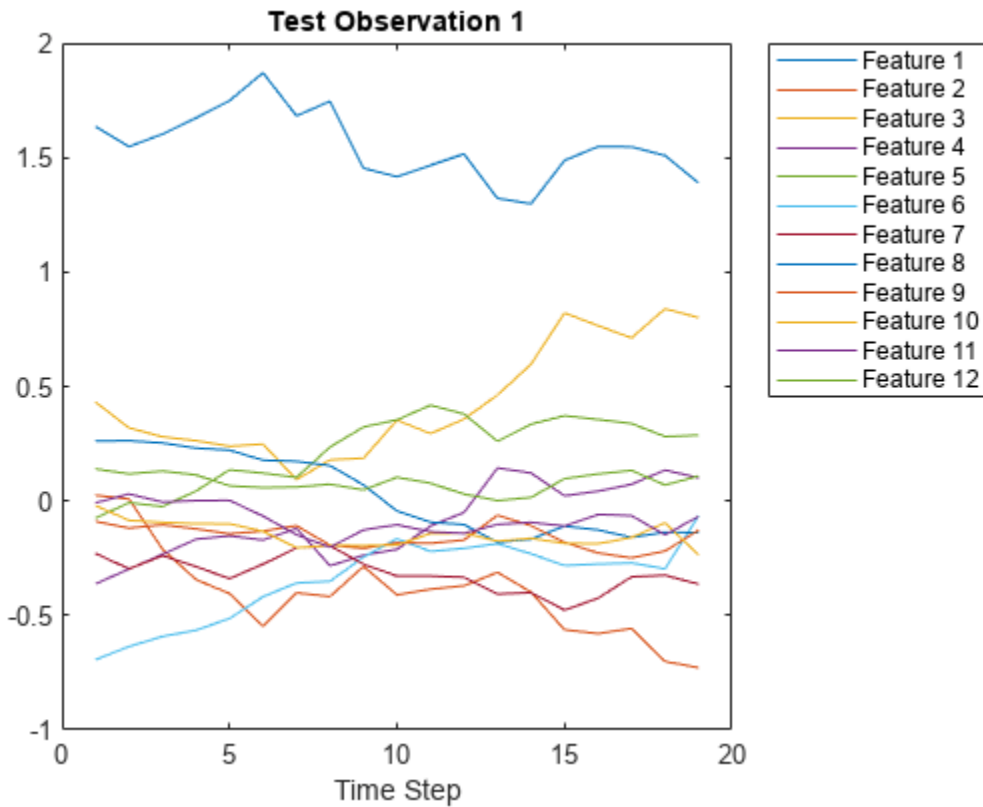
```
testSize = 1×2
```

```
    12    19
```

```
numFeatures = testSize(1);
```

Visualize the first time series in a plot. Each line corresponds to a feature.

```
figure
plot(testSequence')
title("Test Observation 1")
xlabel("Time Step")
legend("Feature " + string(1:numFeatures), 'Location', 'northeastoutside')
```



Classify the test observation.

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

```
label = categorical
      1
```

Use gradCAM to determine which parts of the sequence are important to the classification result.

```
map = gradCAM(net, testSequence, label);
```

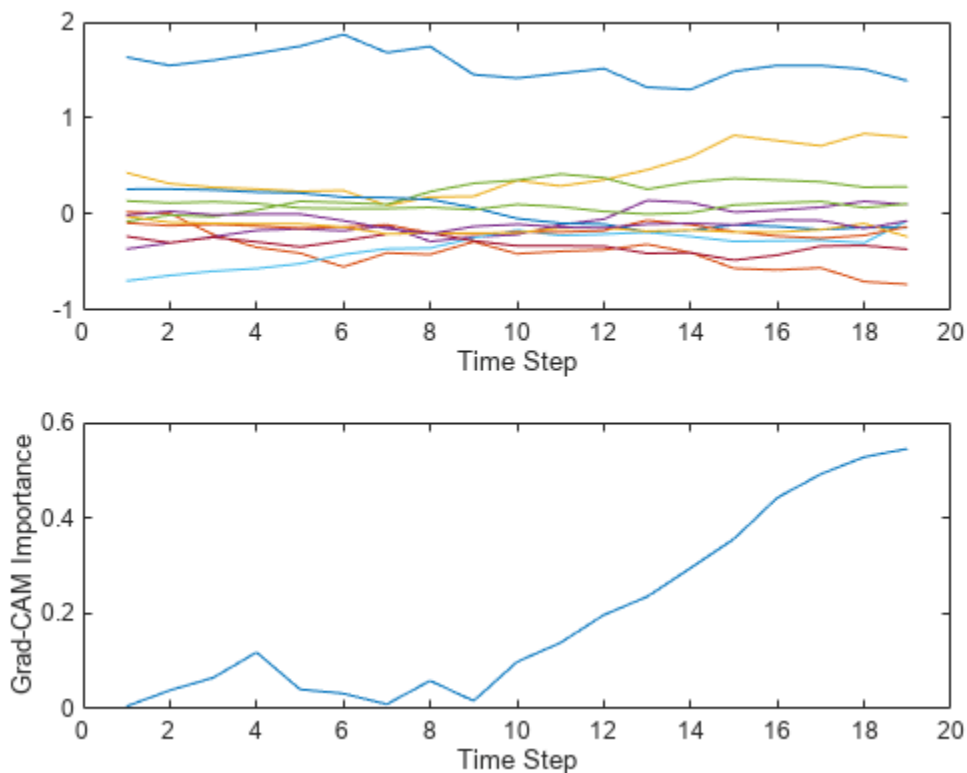
map is a 1-by-19 vector containing the importance value for each of the 19 time steps.

Plot the results to see which time steps contribute most to the classification score.

```
figure
```

```
subplot(2,1,1)
plot(testSequence')
xlabel("Time Step")
```

```
subplot(2,1,2)
plot(map)
xlabel("Time Step")
ylabel("Grad-CAM Importance")
```



The Grad-CAM map suggests that the network is focusing on the later time steps to make the classification decision.

## 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, image3dInputLayer, or sequenceInputLayer. For networks with a sequenceInputLayer, the InputSize property must be scalar.

### **X** — Input data

numeric array | dlarray

Input data, specified as a numeric array or formatted dlarray object.

- For 2-D image data, X must be a  $h$ -by- $w$ -by- $c$  array, where  $h$ ,  $w$ , and  $c$  are the height, width, and number of channels of the network input layer, respectively.

- For 3-D image data, `X` must be a  $h$ -by- $w$ -by- $d$ -by- $c$  array, where  $h$ ,  $w$ ,  $d$ , and  $c$  are the height, width, depth, and number of the network input layer, respectively.
- For vector sequence data, `X` must be a  $c$ -by- $t$  array, where  $c$  is the input size of the network net and  $t$  is the number of time steps in the sequence and can be any positive integer.

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 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` contains the map for each class in `label`. For more information, see `scoreMap`.

The `gradCAM` function sums over the spatial and time 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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 data 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 or time 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 or time dimensions in the output.

The default value is the final layer with nonsingleton spatial or time 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
- T — Time

For more information, see `dlarray`.

The `gradCAM` function supports at most one singleton batch dimension.

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.
- `'nearest'` — Use nearest-neighbor interpolation to expand the map.
- `'none'` — Use no upsampling. The map can be smaller or larger than the input data.

If `'OutputUpsampling'` is `'nearest'` or `'bicubic'`, the Grad-CAM map is the same size as the spatial and temporal dimensions of the input.

- For 2-D image data, the map is size  $h$ -by- $w$ , where  $h$  and  $w$  are the height and width of the image, respectively
- For 3-D image data, the map is size  $h$ -by- $w$ -by- $d$ , where  $h$ ,  $w$ , and  $d$  are the height, width, and depth of the image, respectively.
- For vector sequence input, the map is size 1-by- $t$ , where  $t$  is the number of time steps in the sequence.

If `'OutputUpsampling'` is `'nearest'` or `'bicubic'`, the computed map is upsampled using the `imresize` function for 2-D image and vector sequence data, and the `imresize3` function for 3-D image data. For 3-D image 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 Computing Requirements” (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**

row vector | numeric matrix | numeric array

Grad-CAM importance map, returned as a row vector, 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.



The size of `scoreMap` depends on the type of data.

- For 2-D image data, `scoreMap` is a matrix. `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(:, :, n)` corresponds to the map for `label(n)`.
- For 3-D image data, `scoreMap` is a 3-D array. `scoreMap(i, j, k)` corresponds to the Grad-CAM importance at the spatial location  $(i, j, k)$ . If you provide `label` as a vector of categoricals, character vectors, or strings, then `scoreMap(:, :, :, n)` corresponds to the map for `label(n)`.
- For vector sequence data, `scoreMap` is a row vector. `scoreMap(:, i)` corresponds to the Grad-CAM importance at time step  $i$ . If you provide `label` as a vector of categoricals, character vectors, or strings, then `scoreMap(:, :, n)` corresponds to the map for `label(n)`.

### **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 or time 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 [3]. 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 pooled over space and time dimensions 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 a 2-D 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),  $A^k_{ij}$ , where  $ij$  indexes the pixels. The neuron importance weight is

$$\alpha_k^c = \frac{1}{N} \sum_I \sum_J \frac{\partial y^c}{\partial A_{i,j}^k},$$

Global average pooling  
Gradients  
via  
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:

$$M = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right).$$

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 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  $y_{ij}^c$  is 1 if pixel  $(i,j)$  is predicted to be class  $c$ , and 0 otherwise [4]. 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 `darray` object. The function must reduce this output to a scalar `darray`, 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.
- To display the Grad-CAM map as an overlay on a grayscale image, you must first rescale the map to be in the range  $[0, 1]$ . For example, `scoreMapRescaled = rescale(scoreMap);`.

## Version History

Introduced in R2021a

## References

- [1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Multidimensional Curve Classification Using Passing-through Regions." *Pattern Recognition Letters* 20, no. 11-13 (November 1999): 1103-11. [https://doi.org/10.1016/S0167-8655\(99\)00077-X](https://doi.org/10.1016/S0167-8655(99)00077-X).
- [2] UCI Machine Learning Repository: Japanese Vowels Dataset. <https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels>.

- [3] 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/ICCV2017.74>.
- [4] 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"
- "Explore Quantized Semantic Segmentation Network Using Grad-CAM"
- "Understand Network Predictions Using LIME"
- "Understand Network Predictions Using Occlusion"
- "Multilabel Image Classification Using Deep Learning"

# 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 1-854, “Learning Rate and Regularization” on page 1-855, 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 1-854, “Learning Rate and Regularization” on page 1-855, 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $\text{FiltersPerGroup} * \text{NumGroups}$ .

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  $(\text{Filter Size} - 1) * \text{Dilation Factor} + 1$ . 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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.	$\begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 4 & 0 \\ 0 & 0 & 1 & 5 & 9 & 0 \\ 0 & 0 & 2 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
'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  $2 / (\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannelsPerGroup}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumFiltersPerGroup}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2 / \text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannelsPerGroup}$ .
- '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  $\text{weights} = \text{func}(\text{sz})$ , where  $\text{sz}$  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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle



Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

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];

```

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GroupedConvolution2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)

In `dlnetwork` objects, `GroupedConvolution2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

Introduced in R2019a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## 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 Neural 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”](#)

## groupnorm

Normalize data across grouped subsets of channels for each observation independently

### Syntax

```
Y = groupnorm(X,numGroups,offset,scaleFactor)
Y = groupnorm(X,numGroups,offset,scaleFactor,'DataFormat',FMT)
Y = 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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .

The `groupnorm` function applies the group normalization operation to `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** To apply group normalization within a `LayerGraph` object or `Layer` array, use `groupNormalizationLayer`.

---

`Y = groupnorm(X,numGroups,offset,scaleFactor)` applies the group normalization operation to the input data `X` 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 `X` for each observation in the 'B' (batch) dimension, independently.

For unformatted input data, use the 'DataFormat' option.

`Y = groupnorm(X,numGroups,offset,scaleFactor,'DataFormat',FMT)` applies the group normalization operation to the unformatted `darray` object `X` with format specified by `FMT`. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. For example, 'DataFormat', 'SSCB' specifies data for 2-D image input with format 'SSCB' (spatial, spatial, channel, batch).

`Y = 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);
X = darray(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;
Y = groupnorm(X,numGroups,offset,scaleFactor);
```

## Input Arguments

### X — Input data

`darray` | numeric array

Input data, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

If `X` is an unformatted `darray` or a numeric array, then you must specify the format using the `DataFormat` option. If `X` is a numeric array, then either `scaleFactor` or `offset` must be a `darray` object.

`X` 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.

<b>numGroups</b>	<b>Description</b>
'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 <code>layernorm</code> .
'channel-wise'	Treat all incoming channels as separate groups. This operation is also known as instance normalization. Alternatively, use <code>instancenorm</code> .

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

### **offset** — Offset

`darray` | numeric array

Offset  $\beta$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input  $X$ .

If `offset` is a formatted `darray` object, then the nonsingleton dimension must have label 'C' (channel).

### **scaleFactor** — Scale factor

`darray` | numeric array

Scale factor  $\gamma$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input  $X$ .

If `scaleFactor` is a formatted `darray` object, then the nonsingleton dimension must have label 'C' (channel).

### **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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

### **Epsilon — Constant to add to mini-batch variances**

1e-5 (default) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

Epsilon must be greater than or equal to 1e-5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Output Arguments**

### **Y — Normalized data**

`darray`

Normalized data, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, Y has the same dimension labels as X. If the input data is not a formatted `darray`, Y is an unformatted `darray` 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  $\mu_G$  and variance  $\sigma_G^2$  over spatial, time, and grouped subsets of the channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\hat{x}_i = \frac{x_i - \mu_G}{\sqrt{\sigma_G^2 + \epsilon}},$$

where  $\epsilon$  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

$$y_i = \gamma \hat{x}_i + \beta,$$

where the offset  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

## Version History

Introduced in R2020b

### R2023a: Epsilon supports values less than 1e-5

The Epsilon option also supports positive values less than 1e-5.

## 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:
  - `X`
  - `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"

# 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  $\gamma$  and shifts it by a learnable offset  $\beta$ .

## 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 1-868, "Learning Rate and Regularization" on page 1-870, 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) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

Epsilon must be greater than or equal to 1e-5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **NumChannels — Number of input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### 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  $\gamma$ , specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Scale` is nonempty, then the software uses the `Scale` property as the initial value. If `Scale` is empty, then the software uses the initializer specified by `ScaleInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

Data Types: `single` | `double`

### Offset — Channel offsets

`[]` (default) | numeric array

Channel offsets  $\beta$ , specified as a numeric vector.

The channel offsets are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Offset` is nonempty, then the software uses the `Offset` property as the initial value. If `Offset` is empty, then the software uses the initializer specified by `OffsetInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 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.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### ScaleL2Factor — L<sub>2</sub> regularization factor for scale factors

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> regularization factor using the `trainingOptions` function.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### OffsetL2Factor — L<sub>2</sub> 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.

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 the name `' '`.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	Group Normalization	Group normalization
4	''	ReLU	ReLU
5	''	2-D 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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .

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

### Group Normalization Layer

The group normalization operation normalizes the elements  $x_i$  of the input by first calculating the mean  $\mu_G$  and variance  $\sigma_G^2$  over spatial, time, and grouped subsets of the channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\hat{x}_i = \frac{x_i - \mu_G}{\sqrt{\sigma_G^2 + \epsilon}},$$

where  $\epsilon$  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

$$y_i = \gamma \hat{x}_i + \beta,$$

where the offset  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the

images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GroupNormalizationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `GroupNormalizationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

### Introduced in R2020b

#### R2023a: Epsilon supports values less than 1e-5

The `Epsilon` option also supports positive values less than 1e-5.

## R2023a: Layer supports 1-D image sequence data

GroupNormalizationLayer objects support normalizing 1-D image sequence data (data with one spatial and one time dimension).

## 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 Neural Network for Classification"

"Train Convolutional Neural Network for Regression"

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

## groupSubPlot

**Package:** experiments

Group metrics in experiment training plot

### Syntax

```
groupSubPlot(monitor, groupName, metricNames)
```

### Description

`groupSubPlot(monitor, groupName, metricNames)` groups the specified metrics in a single training subplot with the y-axis label `groupName`. 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 = 100 * (iteration/numIterations);
```

## 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.

### **groupName** — Name of subplot group

string | character vector

Name of subplot group, specified as a string or character vector. Experiment Manager groups the specified metrics in a single training subplot with the y-axis label `groupName`.

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`.

## Version History

Introduced in R2021a

## See Also

### **Apps**

Experiment Manager

### **Objects**

experiments.Monitor

### **Functions**

recordMetrics | updateInfo

### **Topics**

“Use Experiment Manager to Train Generative Adversarial Networks (GANs)”

## groupSubPlot

**Package:** deep

Group metrics in training plot

### Syntax

```
groupSubPlot(monitor, groupName, metricNames)
```

### Description

`groupSubPlot(monitor, groupName, metricNames)` groups the specified metrics in a single training subplot with the y-axis label `groupName`. By default, the software plots each ungrouped metric in its own training subplot.

### Examples

#### Track Progress and Produce Training Plots

Use a `TrainingProgressMonitor` object to track training progress and produce training plots for custom training loops.

Create a `TrainingProgressMonitor` object. The monitor automatically tracks the start time and the elapsed time. The timer starts when you create the object.

---

**Tip** To ensure that the elapsed time accurately reflects the training time, make sure you create the `TrainingProgressMonitor` object close to the start of your custom training loop.

---

```
monitor = trainingProgressMonitor;
```

Before you start the training, specify names for the information and metric values.

```
monitor.Info = ["LearningRate", "Epoch", "Iteration"];  
monitor.Metrics = ["TrainingLoss", "ValidationLoss", "TrainingAccuracy", "ValidationAccuracy"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot, and group the training and validation accuracy in the same plot.

```
monitor.XLabel = "Iteration";  
groupSubPlot(monitor, "Loss", ["TrainingLoss", "ValidationLoss"]);  
groupSubPlot(monitor, "Accuracy", ["TrainingAccuracy", "ValidationAccuracy"]);
```

During training:

- Evaluate the `Stop` property at the start of each step in your custom training loop. When you click the **Stop** button in the Training Progress window, the `Stop` property changes to 1. Training stops if your training loop exits when the `Stop` property is 1.



- Update the information values. The updated values appear in the Training Progress window.
- Record the metric values. The recorded values appear in the training plot.
- Update the training progress percentage based on the fraction of iterations completed.

---

**Note** The following example code is a template. You must edit this training loop to compute your metric and information values. For a complete example that you can run in MATLAB, see “Monitor Custom Training Loop Progress During Training”.

---

```

epoch = 0;
iteration = 0;

monitor.Status = "Running";

while epoch < maxEpochs && ~monitor.Stop
    epoch = epoch + 1;

    while hasData(mbq) && ~monitor.Stop
        iteration = iteration + 1;

        % Add code to calculate metric and information values.
        % lossTrain = ...

        updateInfo(monitor, ...
            LearningRate=LearnRate, ...
            Epoch=string(epoch) + " of " + string(maxEpochs), ...
            Iteration=string(iteration) + " of " + string(numIterations));

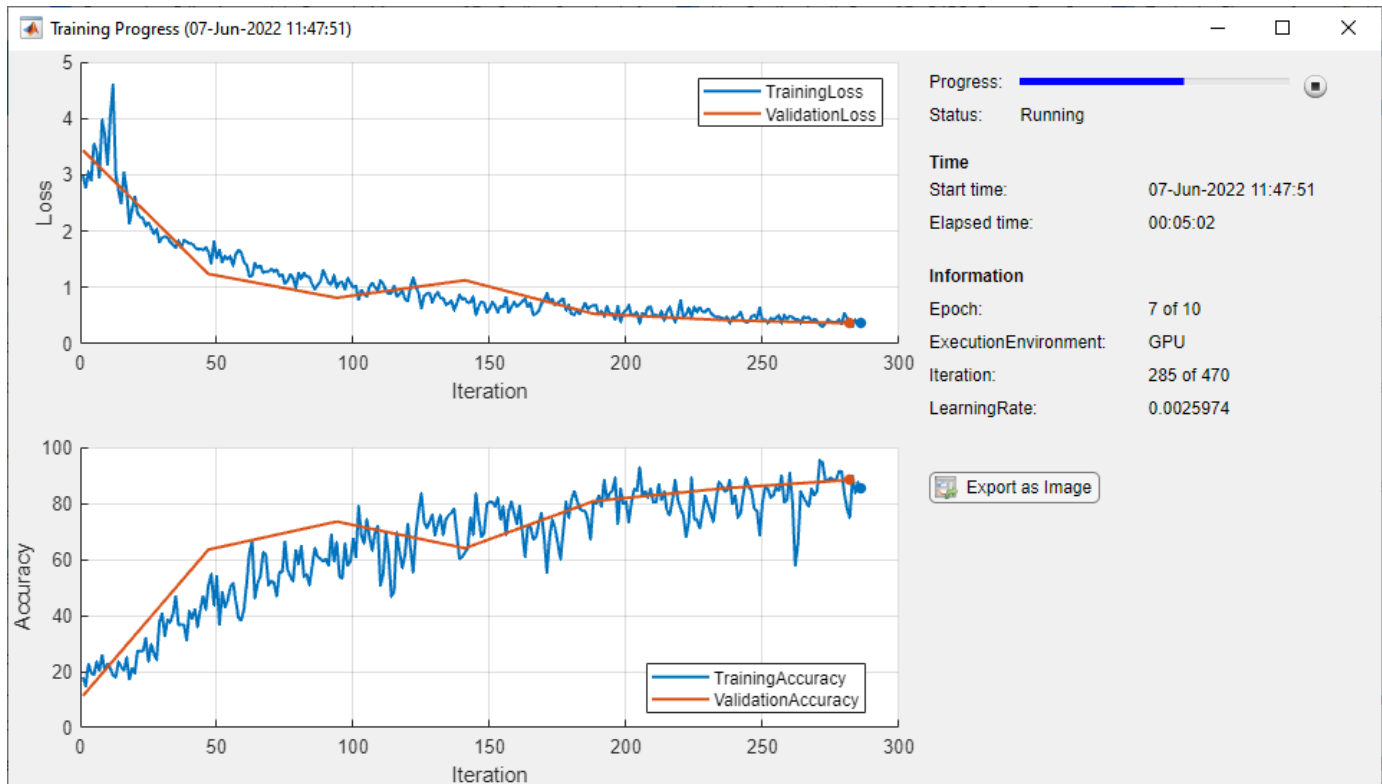
        recordMetrics(monitor, iteration, ...
            TrainingLoss=lossTrain, ...
            TrainingAccuracy=accuracyTrain, ...
            ValidationLoss=lossValidation, ...
            ValidationAccuracy=accuracyValidation);

        monitor.Progress = 100*iteration/numIterations;
    end
end

```

The Training Progress window shows animated plots of the metrics, and the information values, training progress bar, and elapsed time.

- The training plots update each time you call `recordMetrics`.
- The values under **Information** update each time you call `updateInfo`.
- The elapsed time updates each time you call `recordMetrics` or `updateInfo`, and when you update the property.



## Input Arguments

### **monitor** — Training progress monitor

TrainingProgressMonitor object

Training progress monitor, specified as a TrainingProgressMonitor object.

### **groupName** — Name of subplot group

string scalar | character vector

Name of the subplot group, specified as a string scalar or character vector. The software groups the specified metrics in a single training subplot with the y-axis label groupName.

Data Types: char | string

### **metricNames** — Metric names

string scalar | character vector | string array | cell array of character vectors

Metric names, specified as a string scalar, character vector, string array, or cell array of character vectors. Each metric name must be an element of the Metrics property of monitor and can only appear in one subplot.

Data Types: char | string | cell

## Version History

Introduced in R2022b

## **See Also**

trainingProgressMonitor | updateInfo | recordMetrics

## **Topics**

“Monitor Custom Training Loop Progress”

“Train Network Using Custom Training Loop”

“Detect Issues During Deep Neural Network Training”

## gru

Gated recurrent unit

### Syntax

```
Y = gru(X,H0,weights,recurrentWeights,bias)
[Y,hiddenState] = gru(X,H0,weights,recurrentWeights,bias)
___ = gru(X,H0,weights,recurrentWeights,bias,DataFormat=FMT)
___ = gru(X,H0,weights,recurrentWeights,bias,Name=Value)
```

### 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 `darray` data. If you want to apply a GRU operation within a `layerGraph` object or `Layer` array, use the following layer:

- `gruLayer`
- 

`Y = gru(X,H0,weights,recurrentWeights,bias)` applies a gated recurrent unit (GRU) calculation to input `X` using the initial hidden state `H0`, and parameters `weights`, `recurrentWeights`, and `bias`. The input `X` must be a formatted `darray`. The output `Y` is a formatted `darray` with the same dimension format as `X`, 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.

`[Y,hiddenState] = gru(X,H0,weights,recurrentWeights,bias)` also returns the hidden state after the GRU operation.

\_\_\_ = `gru(X,H0,weights,recurrentWeights,bias,DataFormat=FMT)` also specifies the dimension format `FMT` when `X` is not a formatted `darray`. The output `Y` is an unformatted `darray` with the same dimension order as `X`, except for any "S" dimensions.

\_\_\_ = `gru(X,H0,weights,recurrentWeights,bias,Name=Value)` specifies additional options using one or more name-value arguments.

### 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);
X = 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.

```

[Y,hiddenState] = gru(X,H0,weights,recurrentWeights,bias);

```

View the size and dimension format of the output.

```

size(Y)

ans = 1×3

    100    32    64

```

Y.dims

```

ans =
'CBT'

```

View the size of the hidden state.

```

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

### X — Input data

dlarray | numeric array

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `X` is not a formatted `dlarray`, you must specify the dimension label format using the `DataFormat`

name-value argument. If  $X$  is a numeric array, at least one of  $H_0$ , `weights`, `recurrentWeights`, or `bias` must be a `darray`.

$X$  must contain a sequence dimension labeled "T". If  $X$  has any spatial dimensions labeled "S", they are flattened into the "C" channel dimension. If  $X$  does not have a channel dimension, then one is added. If  $X$  has any unspecified dimensions labeled "U", they must be singleton.

Data Types: `single` | `double`

### **$H_0$ — Initial hidden state vector**

`darray` | numeric array

Initial hidden state vector, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

If  $H_0$  is a formatted `darray`, it must contain a channel dimension labeled "C" and optionally a batch dimension labeled "B" with the same size as the "B" dimension of  $X$ . If  $H_0$  does not have a "B" dimension, the function uses the same hidden state vector for each observation in  $X$ .

If  $H_0$  is a formatted `darray`, 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**

`darray` | numeric array

Weights, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

Specify `weights` as a matrix of size  $3 \times \text{NumHiddenUnits}$ -by- $\text{InputSize}$ , where  $\text{NumHiddenUnits}$  is the size of the "C" dimension of  $H_0$ , and  $\text{InputSize}$  is the size of the "C" dimension of  $X$  multiplied by the size of each "S" dimension of  $X$ , where present.

If `weights` is a formatted `darray`, it must contain a "C" dimension of size  $3 \times \text{NumHiddenUnits}$  and a "U" dimension of size  $\text{InputSize}$ .

Data Types: `single` | `double`

### **`recurrentWeights` — Recurrent weights**

`darray` | numeric array

Recurrent weights, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

Specify `recurrentWeights` as a matrix of size  $3 \times \text{NumHiddenUnits}$ -by- $\text{NumHiddenUnits}$ , where  $\text{NumHiddenUnits}$  is the size of the "C" dimension of  $H_0$ .

If `recurrentWeights` is a formatted `darray`, it must contain a "C" dimension of size  $3 \times \text{NumHiddenUnits}$  and a "U" dimension of size  $\text{NumHiddenUnits}$ .

Data Types: `single` | `double`

### **`bias` — Bias**

`darray` vector | numeric vector

Bias, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

Specify `bias` as a vector of length  $3 * \text{NumHiddenUnits}$ , where `NumHiddenUnits` is the size of the "C" dimension of `H0`.

If `bias` is a formatted `darray`, the nonsingleton dimension must be labeled with "C".

Data Types: `single` | `double`

### 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `Y = gru(X,H0,weights,recurrentWeights,bias,DataFormat="CTB")` applies the GRU operation and specifies that the data has format "CTB" (channel, time, batch).

### 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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

### 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 the "Gated Recurrent Unit Layer" on page 1-898 definition on the `gruLayer` reference page.

## Output Arguments

### **Y — GRU output**

`darray`

GRU output, returned as a `darray`. The output `Y` has the same underlying data type as the input `X`.

If the input data `X` is a formatted `darray`, `Y` has the same dimension format as `X`, except for any "S" dimensions. If the input data is not a formatted `darray`, `Y` is an unformatted `darray` with the same dimension order as the input data.

The size of the "C" dimension of `Y` is the same as the number of hidden units, specified by the size of the "C" dimension of `H0`.

### **hiddenState — Hidden state vector**

`darray` | numeric array

Hidden state vector for each observation, returned as a `darray` or a numeric array with the same data type as `H0`.

If the input `H0` is a formatted `darray`, then the output `hiddenState` is a formatted `darray` with the format "CB".

## 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 1-898 definition on the `gruLayer` reference page.

## Version History

**Introduced in R2020a**

### **R2023a: Specify reset gate mode**

Specify the reset gate mode using the `ResetGateMode` name-value argument.

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `X`
  - `H0`
  - `weights`
  - `recurrentWeights`
  - `bias`

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

## See Also

`darray` | `fullyconnect` | `softmax` | `dlgradient` | `dlfeval` | `lstm` | `attention`

## 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 `darray` Support”

## gruLayer

Gated recurrent unit (GRU) layer for recurrent neural network (RNN)

### Description

A GRU layer is an RNN layer that 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 1-890, “State” on page 1-890, “Parameters and Initialization” on page 1-891, “Learning Rate and Regularization” on page 1-893, 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

This property is read-only.

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 that the layer remembers between time steps (the hidden state). The hidden state can contain information from all the 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.

The hidden state does not limit the number of time steps that the layer processes in an iteration. To split your sequences into smaller sequences for when you use the `trainNetwork` function, use the `SequenceLength` training option.

The layer outputs data with `NumHiddenUnits` channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputMode — Output mode**

`'sequence'` (default) | `'last'`

This property is read-only.

Output mode, specified as one of these values:

- `'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 1-898.

`dlnetwork` objects support GRU layers with the `ResetGateMode` set to `"after-multiplication"` only.

**InputSize — Input size**

'auto' (default) | positive integer

This property is read-only.

Input size, specified as a positive integer or 'auto'. If InputSize is 'auto', then the software automatically assigns the input size at training time.

Data Types: double | char

**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  $\sigma_s$  in the calculations to update the hidden state.

**GateActivationFunction — Activation function to apply to gates**

'sigmoid' (default) | 'hard-sigmoid'

This property is read-only.

Activation function to apply to the gates, specified as one of these values:

- 'sigmoid' — Use the sigmoid function  $\sigma(x) = (1 + e^{-x})^{-1}$ .
- 'hard-sigmoid' — Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}$$

The layer uses this option as the function  $\sigma_g$  in the calculations for the layer gates.

**State****HiddenState — Hidden state**

[] (default) | 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 you set this property manually, calls to the `resetState` function set the hidden state to this value.

If HasStateInputs is 1 (true), then the HiddenState property must be empty.

Data Types: single | double

## 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 you train a neural network using the `trainNetwork` function, if `InputWeights` is nonempty, then the software uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then the software uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is a  $3 \times \text{NumHiddenUnits}$ -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 you train an RNN using the `trainNetwork` function, if `RecurrentWeights` is nonempty, then the software uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then the software uses the initializer specified by `RecurrentWeightsInitializer`.

At training time `RecurrentWeights` is a  $3 \times \text{NumHiddenUnits}$ -by- $\text{NumHiddenUnits}$  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  $3 \times \text{NumHiddenUnits}$ -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  $6 \times \text{NumHiddenUnits}$ -by-1 numeric vector.

The layer biases are learnable parameters. When you train a neural 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 1-898.

## **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 you specify 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 you specify using 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  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the input weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using 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  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the recurrent weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name `' '`.

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 is an RNN layer that 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-multiplication' 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

$$\mathbf{h}_t = (1 - z_t) \odot \tilde{\mathbf{h}}_t + z_t \odot \mathbf{h}_{t-1}.$$

The following formulas describe the components at time step  $t$ .

Component	ResetGateMode	Formula
Reset gate	'after-multiplication'	$r_t = \sigma_g(W_r \mathbf{x}_t + b_{W_r} + R_r \mathbf{h}_{t-1})$
	'before-multiplication'	
	'recurrent-bias-after-multiplication'	$r_t = \sigma_g(W_r \mathbf{x}_t + b_{W_r} + R_r \mathbf{h}_{t-1} + b_{R_r})$
Update gate	'after-multiplication'	$z_t = \sigma_g(W_z \mathbf{x}_t + b_{W_z} + R_z \mathbf{h}_{t-1})$
	'before-multiplication'	
	'recurrent-bias-after-multiplication'	$z_t = \sigma_g(W_z \mathbf{x}_t + b_{W_z} + R_z \mathbf{h}_{t-1} + b_{R_z})$
Candidate state	'after-multiplication'	$\tilde{h}_t = \sigma_s(W_{\tilde{h}} \mathbf{x}_t + b_{W_{\tilde{h}}} + r_t \odot (R_{\tilde{h}} \mathbf{h}_{t-1}))$
	'before-multiplication'	
	'recurrent-bias-after-multiplication'	$\tilde{h}_t = \sigma_s(W_{\tilde{h}} \mathbf{x}_t + b_{W_{\tilde{h}}} + r_t \odot (R_{\tilde{h}} \mathbf{h}_{t-1} + b_{R_{\tilde{h}}}))$

In these calculations,  $\sigma_g$  and  $\sigma_s$  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 to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `GRULayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	OutputMode	Output Format
"CB" (channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"CBT" (channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SB" (spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	

In `dlnetwork` objects, `GRULayer` objects also support these input and output format combinations.

Input Format	OutputMode	Output Format	
"SCB" (spatial, channel, batch)	"sequence"	"CB" (channel, batch)	
	"last"		
"SSCB" (spatial, spatial, channel, batch)	"sequence"		
	"last"		
"SSSCB" (spatial, spatial, spatial, channel, batch)	"sequence"		
	"last"		
"SCBT" (spatial, channel, batch, time)	"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)	
"SC" (spatial, channel)	"sequence"	"CU" (channel, unspecified)	
	"last"		
"SSC" (spatial, spatial, channel)	"sequence"		
	"last"		
"SSSC" (spatial, spatial, spatial, channel)	"sequence"		
	"last"		
"CT" (channel, time)	"sequence"		"CT" (channel, time)
	"last"		"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"sequence"		"CT" (channel, time)

Input Format	OutputMode	Output Format
	"last"	"CU" (channel, unspecified)
"SSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSB" (spatial, spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"SSSB" (spatial, spatial, spatial, batch)	"sequence"	
	"last"	
"BT" (batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SBT" (spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSBT" (spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSSBT" (spatial, spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

To use these input formats in `trainNetwork` workflows, convert the data to "CB" (channel, batch) or "CBT" (channel, batch, time) format using `flattenLayer`.

If the `HasStateInputs` property is 1 (true), then the layer has two additional inputs with the 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).

## Version History

Introduced in R2020a

### R2023a: Specify reset gate mode for GRU layers in `dlnetwork` objects

For GRU layers in `dlnetwork` objects, specify the reset gate mode using the `ResetGateMode` property.

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>
- [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:

For code generation in general, the `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

When generating code with Intel MKL-DNN or ARM Compute Library:

- 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'`.

When generating generic C/C++ code:

- The `ResetGateMode` property can be set to `'after-multiplication'`, `'before-multiplication'` or `'recurrent-bias-after-multiplication'`.

### 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 Neural Networks"
- "List of Deep Learning Layers"
- "Deep Learning Tips and Tricks"

# 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** — Mini-batch queue

`minibatchqueue` object

Mini-batch queue, 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.

## Version History

Introduced in R2020b

## 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”

# HBOSDistributionDiscriminator

HBOS distribution discriminator

## Description

The `HBOSDistributionDiscriminator` object is a distribution discriminator that uses the histogram based outlier scores (HBOS) method to compute distribution confidence scores. The object contains a threshold that you can use to separate observations into in-distribution (ID) and out-of-distribution (OOD) data sets.

## Creation

Create a `HBOSDistributionDiscriminator` object using the `networkDistributionDiscriminator` function and setting the method input argument to "hbos".

---

**Note** The object type depends on the method you specify when you use the `networkDistributionDiscriminator` function. The method determines how the software computes the distribution confidence scores. You can set the method argument to either "baseline", "odin", "energy", or "hbos". For more information, see "Distribution Confidence Scores" on page 1-910.

---

## Properties

### Method — Method for computing distribution confidence scores

"hbos"

This property is read-only.

Method for computing the distribution confidence scores, returned as "hbos". For more information, see "Distribution Confidence Scores" on page 1-910.

### Network — Deep learning network

`dlnetwork` object

This property is read-only.

Deep learning network, returned as a `dlnetwork` object.

### LayerNames — Layer names

string array

This property is read-only.

Layer names for the HBOS method, returned as a string array.

The software computes the distribution confidence scores using the principal components from all of the specified layers. For more information, see “Density-Based Methods” on page 1-911.

---

**Note** To compute the histograms, the software must first compute the principal component features for each layer. Computing the principal components can be slow for layers with a large output size and can cause you to run out of memory. If you run out of memory, try specifying layers with a smaller output size. For more information, see “Density-Based Methods” on page 1-911.

---

Example: ["out\_1" "out\_2"]

### **VarianceCutoff — Variance cutoff**

scalar in the range [0, 1]

This property is read-only.

Variance cutoff for the HBOS method, returned as a scalar in the range [0, 1].

The variance cutoff controls the number of principal component features that the software uses to compute the distribution confidence scores. Using a greater number of features takes more computation time. The closer the `VarianceCutoff` value is to 1, the fewer principal components the software uses to compute the scores. For more information, see “Density-Based Methods” on page 1-911.

Example: 0.01

### **Threshold — Distribution threshold**

scalar in the range [-Inf, 0]

Distribution threshold, returned as a scalar in the range [-Inf, 0]. The threshold separates the ID and OOD data using the distribution confidence scores.

Example: -10

## **Object Functions**

<code>isInNetworkDistribution</code>	Determine whether data is within the distribution of the network
<code>distributionScores</code>	Distribution confidence scores

## **Examples**

### **Create Distribution Discriminator Using HBOS Method**

Load a pretrained regression network.

```
load("digitsRegressionMLPNet.mat")
```

Load ID data. Convert the data to a `darray` object.

```
XID = digitTrain4DArrayData;  
XID = darray(XID, "SSCB");
```

Modify the ID training data to create an OOD set.

```
XOOD = XID.*0.3 + 0.1;
```

Create a discriminator using the HBOS method. Set the variance cutoff to 0.0001 and use the penultimate layer to compute the HBOS distribution scores.

```
method = "hbos";
discriminator = networkDistributionDiscriminator(net,XID,XOOD,method, ...
    VarianceCutoff=0.0001, ...
    LayerNames="relu_2")

discriminator =
    HBOSDistributionDiscriminator with properties:

        Method: "hbos"
        Network: [1x1 dlnetwork]
        LayerNames: "relu_2"
        VarianceCutoff: 1.0000e-04
        Threshold: -54.0601
```

Assess the performance of the discriminator on the OOD data.

```
tf = isInNetworkDistribution(discriminator,XOOD);
accuracy = sum(tf == 0)/numel(tf)

accuracy = 0.9820
```

## More About

### In-Distribution and Out-of-Distribution Data

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-909.

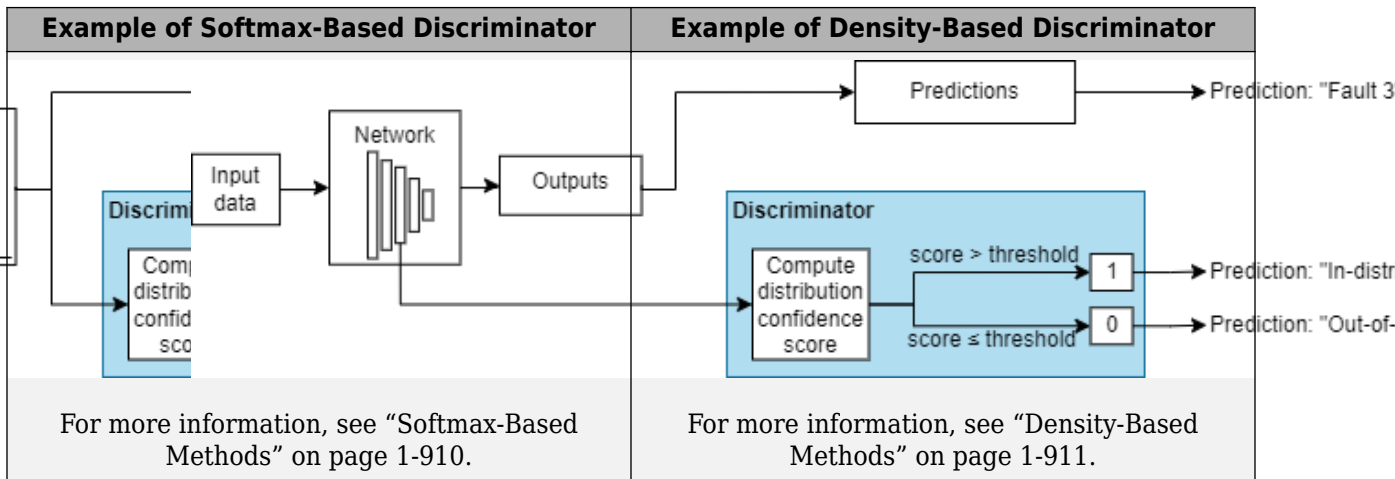
### OOD Data Detection

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-910.

These images show how a discriminator acts as an additional output of a trained neural network.

**Example Data Discriminators**



**Distribution Confidence Scores**

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.

**Softmax-Based Methods**

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}}$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$



- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is  $-\text{Inf}$ .

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. "A Baseline for Detecting Out-of-Distribution Examples in Image Captioning." In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.
- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## See Also

`networkDistributionDiscriminator` | `isInNetworkDistribution` | `distributionScores`

## Topics

"Out-of-Distribution Detection for Deep Neural Networks"

"Out-of-Distribution Data Discriminator for YOLO v4 Object Detector"

"Verify Robustness of Deep Learning Neural Network"

# huber

Huber loss for regression tasks

## Syntax

```
loss = huber(Y,targets)
loss = huber(Y,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

`loss = huber(Y,targets)` returns the Huber loss between the formatted `darray` object `Y` containing the predictions and the target values `targets` for regression tasks. The input `Y` is a formatted `darray`. The output `loss` is an unformatted `darray` scalar.

For unformatted input data, use the 'DataFormat' option.

`loss = huber(Y,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 `Y` is not a formatted `darray`.

`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 =  
    1×1 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=12x1 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=12x1 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 = 1x3
```

```
    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 =
    1×1 darray
    8.1834
```

## Input Arguments

### Y — Predictions

`darray` | numeric array

Predictions, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. When Y is not a formatted `darray`, you must specify the dimension format using the `DataFormat` option.

If Y is a numeric array, `targets` must be a `darray`.

### targets — Target responses

`darray` | numeric array

Target responses, specified as a formatted or unformatted `darray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of Y.

If `targets` is a formatted `darray`, then its format must be the same as the format of Y, or the same as `DataFormat` if Y is unformatted.

If `targets` is an unformatted `darray` or a numeric array, then the function applies the format of Y or the value of `DataFormat` to `targets`.

---

**Tip** Formatted `darray` 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 Y and `targets` are consistent, when Y is a formatted `darray`, also specify `targets` as a formatted `darray`.

---

**weights — Weights**

darray | numeric array

Weights, specified as a darray 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 Y. Specify the 'C' (channel) dimension of the response weights by using a formatted darray 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 Y. Specify the 'B' (batch) dimension of the class weights by using a formatted darray 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 Y. In this case, if weights is not a formatted darray object, then the function uses the same format as Y. Alternatively, specify the weights format using 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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<sub>1</sub> loss*.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Mask — Mask indicating which elements to include for loss computation**

darray | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a darray object, a logical array, or a numeric array with the same size as Y.

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.

If Mask is a formatted darray object, then its format must match that of Y. If Mask is not a formatted darray object, then the function uses the same format as Y.

If you specify the DataFormat option, then the function also uses the specified format for the mask.

The size of each dimension of Mask must match the size of the corresponding dimension in Y. The default value is a logical array of ones.

---

**Tip** Formatted darray 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,

`darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

### **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 `darray` object with the same size as `Y`.

### **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 `Y`.
- "all-elements" — Normalize the loss by dividing it by the number of elements of `Y`.
- "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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `Y` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `Y` are vectors, then default value of `WeightsFormat` is the same as the format of `Y`.

Data Types: `char` | `string`

## Output Arguments

### loss — Huber loss

`darray`

Huber loss, returned as an unformatted `darray`. The output `loss` is an unformatted `darray` with the same underlying data type as the input `Y`.

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

$$\text{loss}_j = \begin{cases} \frac{1}{2}(Y_j - T_j)^2 & \text{if } |Y_j - T_j| \leq \delta \\ \delta|Y_j - T_j| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

where  $T_j$  is the corresponding target value to the prediction  $Y_j$  and  $\delta$  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:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

## Version History

Introduced in R2021a

## 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:
  - `Y`
  - `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”

# 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 1-921. 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

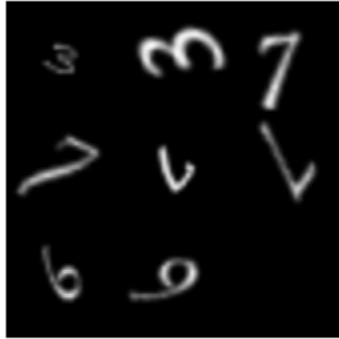
```

Preview the random transformations applied to the first eight images in the image datastore.

```

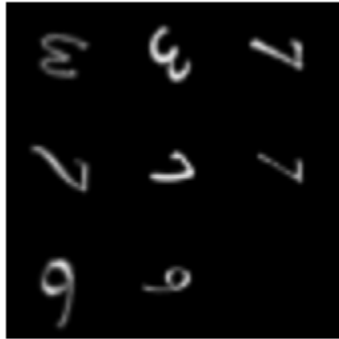
minibatch = preview(auimds);
imshow(imtile(minibatch.input));

```



Preview different random transformations applied to the same set of images.

```
minibatch = preview(aumds);  
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.

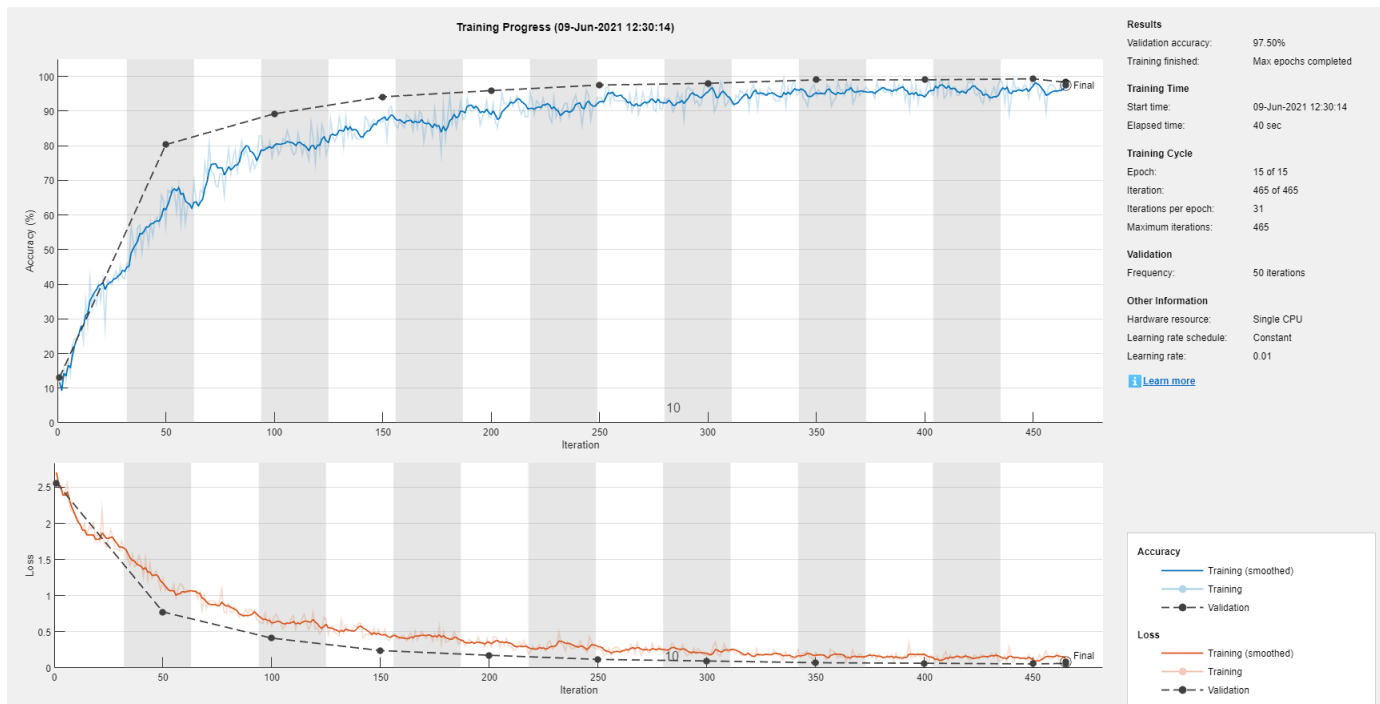
```

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.

## Version History

Introduced in R2017b

## See Also

`augmentedImageDatastore` | `imageInputLayer` | `trainNetwork`

## Topics

“Deep Learning in MATLAB”

“Preprocess Images for Deep Learning”

“Create Function Handle”

# image3dInputLayer

3-D image input layer

## Description

A 3-D image input layer inputs 3-D images or volumes to a neural 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

This property is read-only.

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  $Y = \text{func}(X)$ , where  $X$  is the input data and the output  $Y$  is the normalized data.

---

**Tip** The software, by default, automatically calculates the normalization statistics when using the `trainNetwork` function. To save time when training, specify the required statistics for normalization and set the `ResetInputNormalization` option in `trainingOptions` to `0` (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 `trainNetwork` function calculates the mean. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the Mean property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the standard deviation. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `StandardDeviation` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the minima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Min` property to a numeric scalar or a numeric array.

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 `Max` property, then `Normalization` must be `'rescale-symmetric'` or `'rescale-zero-one'`. If `Max` is [], then the `trainNetwork` function calculates the maxima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Max` property to a numeric scalar or a numeric array.

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 the name ''.

Data Types: `char` | `string`

### **NumInputs — Number of inputs**

0 (default)

This property is read-only.

Number of inputs of the layer. The layer has no inputs.

Data Types: double

### **InputNames — Input names**

{ } (default)

This property is read-only.

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 '' 3-D 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
```

## Version History

### Introduced in R2019a

#### **R2019b: 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.

#### **R2019b: 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”](#)



# imageInputLayer

Image input layer

## Description

An image input layer inputs 2-D images to a neural 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 `Normalization`, `NormalizationDimension`, `Mean`, `StandardDeviation`, `Min`, `Max`, `SplitComplexInputs`, and `Name` properties using one or more name-value arguments. Enclose the property names in quotes.

## Properties

### Image Input

#### **InputSize** — Size of the input

row vector of integers

This property is read-only.

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

This property is read-only.

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  $Y = \text{func}(X)$ , where  $X$  is the input data and the output  $Y$  is the normalized data.

---

**Tip** The software, by default, automatically calculates the normalization statistics when using the `trainNetwork` function. To save time when training, specify the required statistics for normalization and set the `ResetInputNormalization` option in `trainingOptions` to 0 (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 `trainNetwork` function calculates the mean. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Mean` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the standard deviation. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `StandardDeviation` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the minima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Min` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the maxima. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Max` property to a numeric scalar or a numeric array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SplitComplexInputs — Flag to split input data into real and imaginary components**

0 (false) (default) | 1 (true)

This property is read-only.

Flag to split input data into real and imaginary components specified as one of these values:

- 0 (false) - Do not split input data.
- 1 (true) - Split data into real and imaginary components.

When `SplitComplexInputs` is 1, then the layer outputs twice as many channels as the input data. For example, if the input data is complex-values with `numChannels` channels, then the layer outputs

data with  $2 \times \text{numChannels}$  channels, where channels 1 through  $\text{numChannels}$  contain the real components of the input data and  $\text{numChannels}+1$  through  $2 \times \text{numChannels}$  contain the imaginary components of the input data. If the input data is real, then channels  $\text{numChannels}+1$  through  $2 \times \text{numChannels}$  are all zero.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

For an example showing how to train a network with complex-valued data, see “Train Network with Complex-Valued Data”.

### **DataAugmentation — Data augmentation transforms**

'none' (default) | 'randcrop' | 'randfliplr' | cell array of 'randcrop' and 'randfliplr'

This property is read-only.

---

**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 the name ' '.

Data Types: `char` | `string`

#### **NumInputs — Number of inputs**

0 (default)

This property is read-only.

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

#### **InputNames — Input names**

{ } (default)

This property is read-only.

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]
      SplitComplexInputs: 0

  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  ''  2-D Convolution       20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
   3  ''  ReLU                  ReLU
   4  ''  2-D 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
```

## Version History

### Introduced in R2016a

#### **R2019b: 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.

#### **R2019b: 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.
- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

**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.
- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

**See Also**

[trainNetwork](#) | [convolution2dLayer](#) | [fullyConnectedLayer](#) | [maxPooling2dLayer](#) | [augmentedImageDatastore](#) | [image3dInputLayer](#) | **Deep Network Designer** | [featureInputLayer](#)

**Topics**

[“Create Simple Deep Learning Neural Network for Classification”](#)

[“Train Convolutional Neural Network for Regression”](#)

[“Deep Learning in MATLAB”](#)

[“Specify Layers of Convolutional Neural Network”](#)

[“List of Deep Learning Layers”](#)

## 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)
```

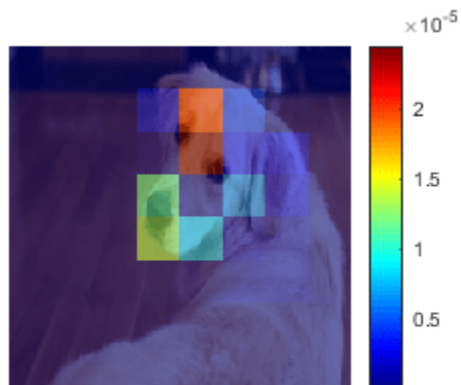
```
label = categorical  
golden retriever
```

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','NumFeatures',3072);
```

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
```

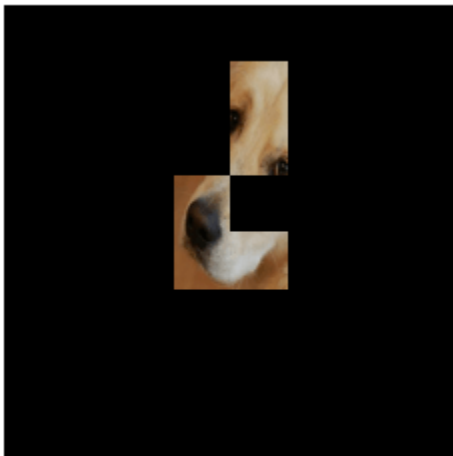


Use the feature importance to find the indices of the most important five features.

```
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;

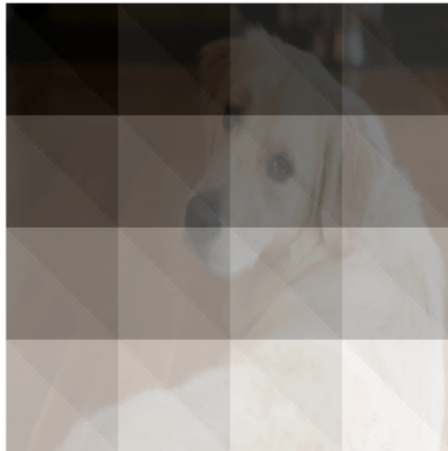
% Set the diagonal elements to alternate values 1 and 2.
segmentationSubset(1:(blockSize+1):end) = repmat([1 2],1,blockSize/2)';
```

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
```

**Custom Segmentation Map**

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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 Computing Requirements” (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 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.

## **Version History**

**Introduced in R2020b**

## **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”](#)

# 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 'ipl' 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

### Generate Code for Imported Network Architecture

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).
- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importCaffeLayers` returns the network architecture layers as a `Layer` or `LayerGraph` object. For code generation, you must first convert the imported `Layer` or `LayerGraph` object to a network. Convert a `Layer` or `LayerGraph` object to a `DAGNetwork` or `SeriesNetwork` object by using `assembleNetwork`. Convert a `Layer` or `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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 layers 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.
 

```
net = dlupdate(@gpuArray,net)
```
- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 `profile` specifies with the include-phase TEST. The function ignores any layers that `profile` specifies with the include-phase TRAIN.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.

## Version History

Introduced in R2017a

## References

[1] *Caffe*. <https://caffe.berkeleyvision.org/>.

[2] *Caffe Model Zoo*. [https://caffe.berkeleyvision.org/model\\_zoo.html](https://caffe.berkeleyvision.org/model_zoo.html).

## See Also

`importCaffeNetwork` | `importONNXLayers` | `importTensorFlowLayers` | `importONNXNetwork` | `importTensorFlowNetwork` | `importNetworkFromPyTorch` | `assembleNetwork` | `exportONNXNetwork` | `exportNetworkToTensorFlow`

## Topics

“Deep Learning in MATLAB”



“Pretrained Deep Neural Networks”  
“List of Deep Learning Layers”

# 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: [7x1 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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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**

### **Generate Code for Imported Network**

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).
- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importCaffeNetwork` returns the network `net` as a `DAGNetwork` or `SeriesNetwork` object. Both these objects support code generation. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

## 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox).

## Tips

- `importCaffeNetwork` can import networks with the following Caffe layer types, with some limitations:

Caffe Layer	Deep Learning Toolbox Layer
<code>BatchNormLayer</code>	<code>batchNormalizationLayer</code>
<code>ConcatLayer</code>	<code>depthConcatenationLayer</code>
<code>ConvolutionLayer</code>	<code>convolution2dLayer</code>
<code>DeconvolutionLayer</code>	<code>transposedConv2dLayer</code>
<code>DropoutLayer</code>	<code>dropoutLayer</code>
<code>EltwiseLayer</code> (only sum)	<code>additionLayer</code>
<code>EuclideanLossLayer</code>	<code>RegressionOutputLayer</code>
<code>InnerProductLayer</code>	<code>fullyConnectedLayer</code>
<code>InputLayer</code>	<code>imageInputLayer</code>
<code>LRNLayer</code> (Local Response Normalization)	<code>crossChannelNormalizationLayer</code>
<code>PoolingLayer</code>	<code>maxPooling2dLayer</code> or <code>averagePooling2dLayer</code>
<code>ReLULayer</code>	<code>reluLayer</code> or <code>leakyReluLayer</code>
<code>ScaleLayer</code>	<code>batchNormalizationLayer</code>
<code>SigmoidLayer</code>	<code>nnet.caffe.layer.SigmoidLayer</code>
<code>SoftmaxLayer</code>	<code>softmaxLayer</code>

Caffe Layer	Deep Learning Toolbox Layer
TanHLayer	tanhLayer

If the network contains any other type of layer, then the software returns an error.

The function imports only the layers that `profile` specifies with the include-phase TEST. The function ignores any layers that `profile` specifies with the include-phase TRAIN.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.

## Version History

### Introduced in R2017a

#### R2018b: '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](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` | `importONNXNetwork` | `importTensorFlowNetwork` | `importNetworkFromPyTorch` | `importONNXLayers` | `importTensorFlowLayers` | `assembleNetwork` | `exportONNXNetwork` | `exportNetworkToTensorFlow`

### Topics

“Deep Learning in MATLAB”

“Pretrained Deep Neural Networks”

# 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:

        InputNames: {'input_1'}
        OutputNames: {'ClassificationLayer_activation_1'}
        Layers: [13x1 nnet.cnn.layer.Layer]
        Connections: [13x2 table]
```

### Import Layers from Keras Network and Plot Architecture

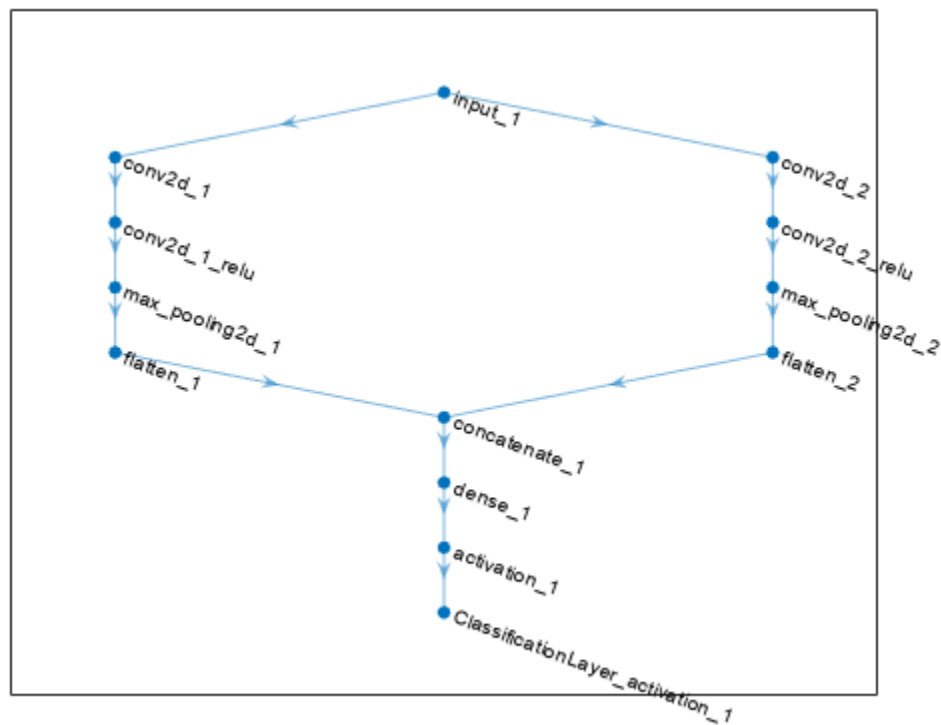
Import the network layers from the model file `digitsDAGnet.h5`.

```
modelfile = 'digitsDAGnet.h5';
layers = importKerasLayers(modelfile)

layers =
  LayerGraph with properties:
    InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
    Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
```

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:
    InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
    Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [13x2 table]
```

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 (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	15.62%	31.6977	0.0010
1	50	00:00:13	63.28%	1.2113	0.0010
2	100	00:00:24	85.16%	0.4127	0.0010
3	150	00:00:36	96.88%	0.1745	0.0010
4	200	00:00:47	99.22%	0.0457	0.0010
5	250	00:00:57	100.00%	0.0364	0.0010
6	300	00:01:07	96.88%	0.1215	0.0010
7	350	00:01:16	100.00%	0.0086	0.0010
7	400	00:01:26	100.00%	0.0165	0.0010
8	450	00:01:36	100.00%	0.0097	0.0010
9	500	00:01:45	100.00%	0.0046	0.0010
10	550	00:01:55	100.00%	0.0031	0.0010
10	580	00:02:00	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:  
  
    InputNames: {'input_1'}  
    OutputNames: {'ClassificationLayer_activation_1'}  
    Layers: [13x1 nnet.cnn.layer.Layer]  
    Connections: [13x2 table]
```

View the size of the weights in the second layer.

```
weights = layers.Layers(2).Weights;  
size(weights)
```

```
ans = 1x4
```

```
    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:

        InputNames: {'input_1'}
        OutputNames: {'ClassificationLayer_activation_1'}
        Layers: [13x1 nnet.cnn.layer.Layer]
        Connections: [13x2 table]
```

### 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.

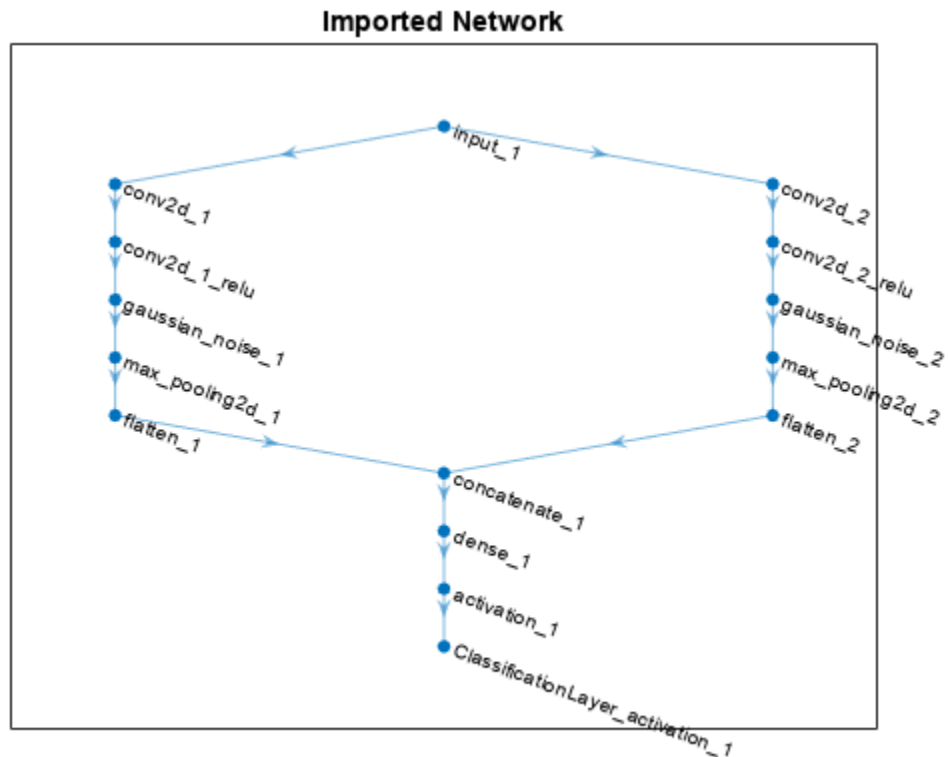
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

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
```

Create two Gaussian noise layers with the same configurations as the imported Keras layers using the helper `gaussianNoiseLayer` function.

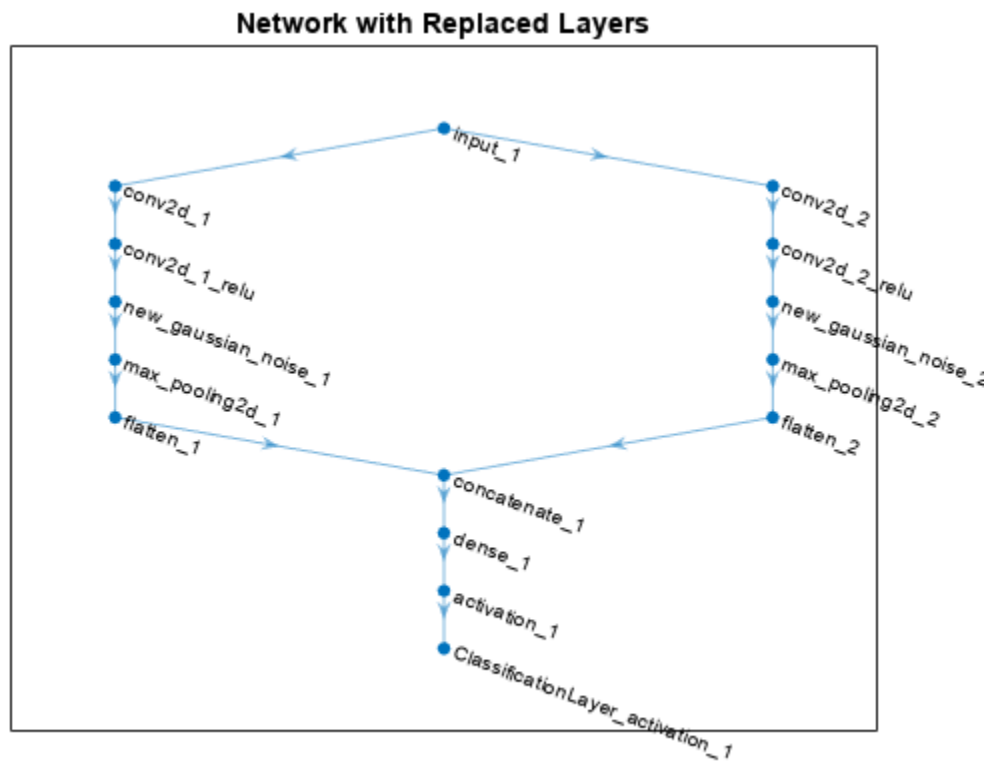
```
gnLayer1 = gaussianNoiseLayer(1.5, 'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7, 'new_gaussian_noise_2');
```

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'	2-D Convolution	20 7x7x1 convolutions with
3	'conv2d_1_relu'	ReLU	ReLU
4	'conv2d_2'	2-D Convolution	20 3x3x1 convolutions with
5	'conv2d_2_relu'	ReLU	ReLU
6	'new_gaussian_noise_1'	Gaussian Noise	Gaussian noise with standa
7	'new_gaussian_noise_2'	Gaussian Noise	Gaussian noise with standa
8	'max_pooling2d_1'	2-D Max Pooling	2x2 max pooling with stride
9	'max_pooling2d_2'	2-D 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

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ a_i x_i & \text{if } x_i \leq 0 \end{cases}$$

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 'Alpha'. During import
```

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'         2-D Convolution      20 7x7x1 convolutions with stri
    3  'conv2d_2'         2-D Convolution      20 3x3x1 convolutions with stri
```

4	'p_re_lu_1'	PReLU	PReLU layer
5	'p_re_lu_2'	PReLU	PReLU layer
6	'max_pooling2d_1'	2-D Max Pooling	2x2 max pooling with stride [2
7	'max_pooling2d_2'	2-D Max Pooling	2x2 max pooling with stride [2
8	'flatten_1'	Keras Flatten	Flatten activations into 1-D as
9	'flatten_2'	Keras Flatten	Flatten activations into 1-D as
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	crossentropy

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 =
  PReLU layer 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 =
  PReLU layer 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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	<code>additionLayer</code>
Activation, with activation names: <ul style="list-style-type: none"> <li>• <code>elu</code></li> <li>• <code>gelu</code></li> <li>• <code>relu</code></li> <li>• <code>linear</code></li> <li>• <code>softmax</code></li> <li>• <code>sigmoid</code></li> <li>• <code>swish</code></li> <li>• <code>tanh</code></li> </ul>	Layers: <ul style="list-style-type: none"> <li>• <code>eluLayer</code></li> <li>• <code>geluLayer</code></li> <li>• <code>reluLayer</code> or <code>clippedReluLayer</code></li> <li>• None</li> <li>• <code>softmaxLayer</code></li> <li>• <code>sigmoidLayer</code></li> <li>• <code>swishLayer</code></li> <li>• <code>tanhLayer</code></li> </ul>
Advanced activations: <ul style="list-style-type: none"> <li>• ELU</li> <li>• Softmax</li> <li>• ReLU</li> <li>• LeakyReLU</li> <li>• PReLU*</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• <code>eluLayer</code></li> <li>• <code>softmaxLayer</code></li> <li>• <code>reluLayer</code>, <code>clippedReluLayer</code>, or <code>leakyReluLayer</code></li> <li>• <code>leakyReluLayer</code></li> <li>• <code>nnet.keras.layer.PreluLayer</code></li> </ul>
<code>AveragePooling1D</code>	<code>averagePooling1dLayer</code> with <code>PaddingValue</code> specified as <code>'mean'</code>
<code>AveragePooling2D</code>	<code>averagePooling2dLayer</code> with <code>PaddingValue</code> specified as <code>'mean'</code>

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, `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 1-973.

## Supported Keras Loss Functions

`importKerasLayers` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`
- `binary_crossentropy`

## Generate Code for Imported Network Architecture

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).
- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importKerasLayers` returns the network architecture layers as a `Layer` or `LayerGraph` object. For code generation, you must first convert the imported `Layer` or `LayerGraph` object to a network. Convert a `Layer` or `LayerGraph` object to a `DAGNetwork` or `SeriesNetwork` object by using `assembleNetwork`. Convert a `Layer` or `LayerGraph` object to a `dlnetwork` object by using `dlnetwork`. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

## 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 layers 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.

```
net = dlupdate(@gpuArray,net)
```

- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 1-977), 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 1-969.
- 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 1-1047. 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 format to RGB format.
  - 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”.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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.

## Version History

Introduced in R2017b

## References

[1] *Keras: The Python Deep Learning library*. <https://keras.io>.

## See Also

`importKerasNetwork` | `importTensorFlowLayers` | `importONNXLayers` |  
`importTensorFlowNetwork` | `importNetworkFromPyTorch` | `findPlaceholderLayers` |  
`replaceLayer` | `assembleNetwork` | `importONNXNetwork` | `exportNetworkToTensorFlow` |  
`exportONNXNetwork`

## Topics

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Tips on Importing Models from TensorFlow, PyTorch, and ONNX”

“Pretrained Deep Neural Networks”

“Define Custom Deep Learning Layers”

## 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), w
```

```
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.

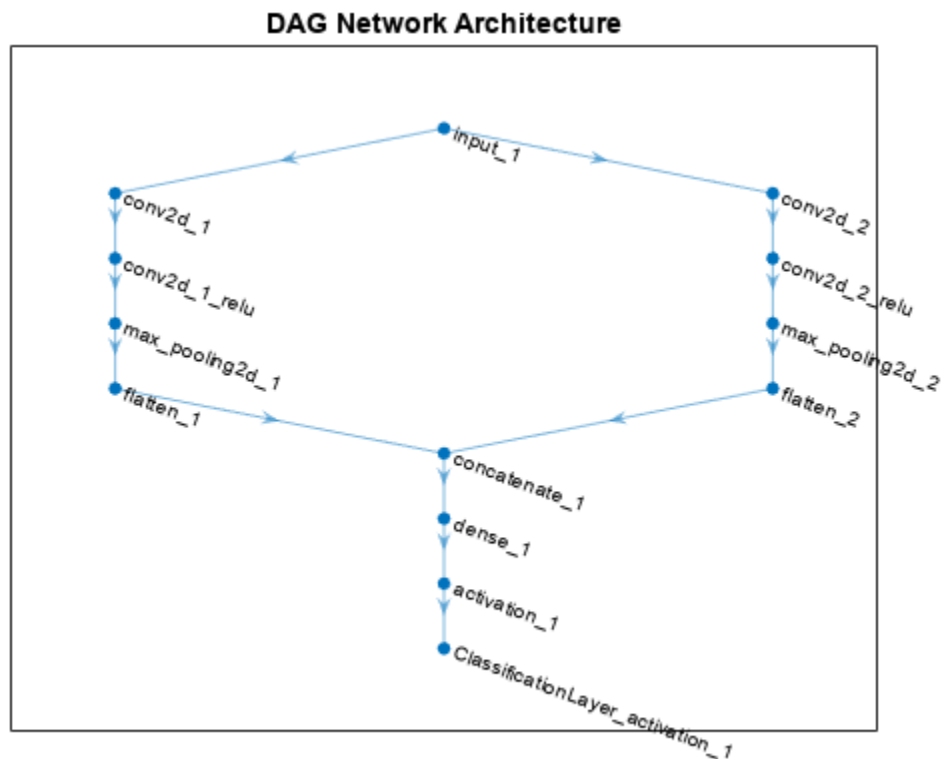
```
net = importKerasNetwork(modelfile)
```

Warning: Saved Keras networks do not include classes. Classes will be set to `categorical(1:N)`, where `N` is the number of classes.

```
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)])
```

Classification result: 5



## 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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	<code>additionLayer</code>

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
Activation, with activation names: <ul style="list-style-type: none"> <li>• elu</li> <li>• gelu</li> <li>• relu</li> <li>• linear</li> <li>• softmax</li> <li>• sigmoid</li> <li>• swish</li> <li>• tanh</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• geluLayer</li> <li>• reluLayer or clippedReluLayer</li> <li>• None</li> <li>• softmaxLayer</li> <li>• sigmoidLayer</li> <li>• swishLayer</li> <li>• tanhLayer</li> </ul>
Advanced activations: <ul style="list-style-type: none"> <li>• ELU</li> <li>• Softmax</li> <li>• ReLU</li> <li>• LeakyReLU</li> <li>• PReLU*</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• softmaxLayer</li> <li>• reluLayer, clippedReluLayer, or leakyReluLayer</li> <li>• leakyReluLayer</li> <li>• nnet.keras.layer.PreluLayer</li> </ul>
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

TensorFlow-Keras Layer	Corresponding Deep Learning Toolbox Layer
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, `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 1-973.

### Supported Keras Loss Functions

`importKerasNetwork` supports the following Keras loss functions:

- `mean_squared_error`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`
- `binary_crossentropy`

### Generate Code for Imported Network

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).
- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importKerasNetwork` returns the network `net` as a `DAGNetwork` or `SeriesNetwork` object. Both these objects support code generation. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 1-987), 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 1-1047. 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 format to RGB format.

- 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”.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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.

## Version History

### Introduced in R2017b

#### R2018b: '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

`importKerasLayers` | `importTensorFlowNetwork` | `importNetworkFromPyTorch` | `importONNXNetwork` | `importTensorFlowLayers` | `importONNXLayers` | `exportNetworkToTensorFlow` | `exportONNXNetwork`

## Topics

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Tips on Importing Models from TensorFlow, PyTorch, and ONNX”  
“Pretrained Deep Neural Networks”

# importNetworkFromPyTorch

Import PyTorch model as MATLAB network

## Syntax

```
net = importNetworkFromPyTorch(modelfile)
net = importNetworkFromPyTorch(modelfile,PackageName=CustomLayersPackage)
```

## Description

`net = importNetworkFromPyTorch(modelfile)` imports a pretrained and traced PyTorch® model from the file `modelfile`. The function returns the network `net` as an uninitialized `dlnetwork` object.

`importNetworkFromPyTorch` requires the Deep Learning Toolbox Converter for PyTorch Models support package. If this support package is not installed, then `importNetworkFromPyTorch` provides a download link.

---

**Note** The `importNetworkFromPyTorch` function might generate a custom layer when converting a PyTorch layer. For more information, see “Algorithms” on page 1-1015. The function saves the generated custom layers in the package `+modelfile`.

---

`net = importNetworkFromPyTorch(modelfile,PackageName=CustomLayersPackage)` imports a pretrained network from PyTorch and saves the generated custom layers and associated functions in the package `+CustomLayersPackage`.

## Examples

### Import Network from PyTorch and Add Input Layer

Import a pretrained and traced PyTorch model as an uninitialized `dlnetwork` object. Then, add an input layer to the imported network.

This example imports the MNASNet (Copyright© Soumith Chintala 2016) PyTorch model. MNASNet is an image classification model that is trained with images from the ImageNet database. Download the `mnasnet1_0.pt` file, which is approximately 17 MB in size, from the MathWorks website.

```
modelfile = matlab.internal.examples.downloadSupportFile("nnet", ...
    "data/PyTorchModels/mnasnet1_0.pt");
```

Import the MNASNet model by using the `importNetworkFromPyTorch` function. The function imports the model as an uninitialized `dlnetwork` object without an input layer. The software displays a warning that provides you information on the number of input layers, what type of input layer to add, and how to add an input layer.

```
net = importNetworkFromPyTorch(modelfile)
```

Warning: Network was imported as an uninitialized `dlnetwork`. Before using the network, add input

```
inputLayer1 = imageInputLayer(<inputSize>, Normalization="none");  
net = addInputLayer(net, inputLayer1, Initialize=true);
```

```
net =  
  dlnetwork with properties:  
  
    Layers: [152x1 nnet.cnn.layer.Layer]  
  Connections: [163x2 table]  
  Learnables: [210x3 table]  
    State: [104x3 table]  
  InputNames: {'TopLevelModule_layers_0'}  
  OutputNames: {'aten__linear12'}  
  Initialized: 0
```

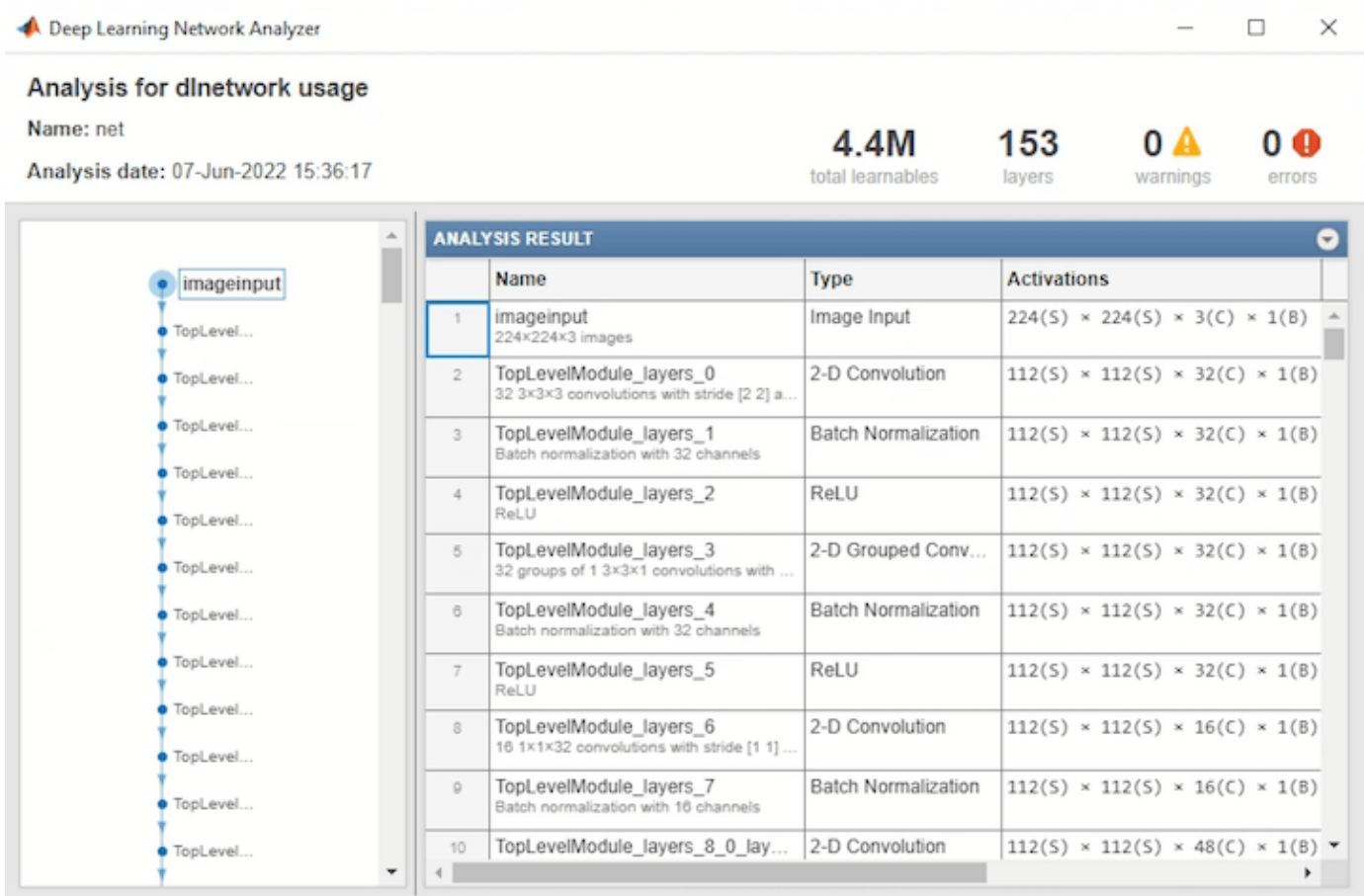
View summary with `summary`.

Specify the input size of the imported network and create an image input layer. Then, add the image input layer to the imported network and initialize the network by using the `addInputLayer` function.

```
InputSize = [224 224 3];  
inputLayer = imageInputLayer(InputSize, Normalization="none");  
net = addInputLayer(net, inputLayer, Initialize=true);
```

Analyze the imported network and view the input layer. The network is ready to use for prediction.

```
analyzeNetwork(net)
```



## Import Network from PyTorch and Initialize

Import a pretrained and traced PyTorch model as an uninitialized `dlnetwork` object. Then, initialize the imported network.

This example imports the MNASNet (Copyright© Soumith Chintal 2016) PyTorch model. MNASNet is an image classification model that is trained with images from the ImageNet database. Download the `mnasnet1_0.pt` file, which is approximately 17 MB in size, from the MathWorks website.

```
modelfile = matlab.internal.examples.downloadSupportFile("nnet", ...
    "data/PyTorchModels/mnasnet1_0.pt");
```

Import the MNASNet model by using the `importNetworkFromPyTorch` function. The function imports the model as an uninitialized `dlnetwork` object.

```
net = importNetworkFromPyTorch(modelfile)
```

Warning: Network was imported as an uninitialized `dlnetwork`. Before using the network, add input

```
inputLayer1 = imageInputLayer(<inputSize>, Normalization="none");
net = addInputLayer(net, inputLayer1, Initialize=true);
```

```
net =
  dlnetwork with properties:

    Layers: [152x1 nnet.cnn.layer.Layer]
    Connections: [163x2 table]
    Learnables: [210x3 table]
    State: [104x3 table]
    InputNames: {'TopLevelModule_layers_0'}
    OutputNames: {'aten__linear12'}
    Initialized: 0

View summary with summary.
```

Specify the input size of the imported network. Then, create a random `dLarray` object that represents the input to the network. The data format of the `dLarray` object must have the dimensions "SSCB" (spatial, spatial, channel, batch) to represent a 2-D image input. For more information, see "Data Formats for Prediction with `dlnetwork`".

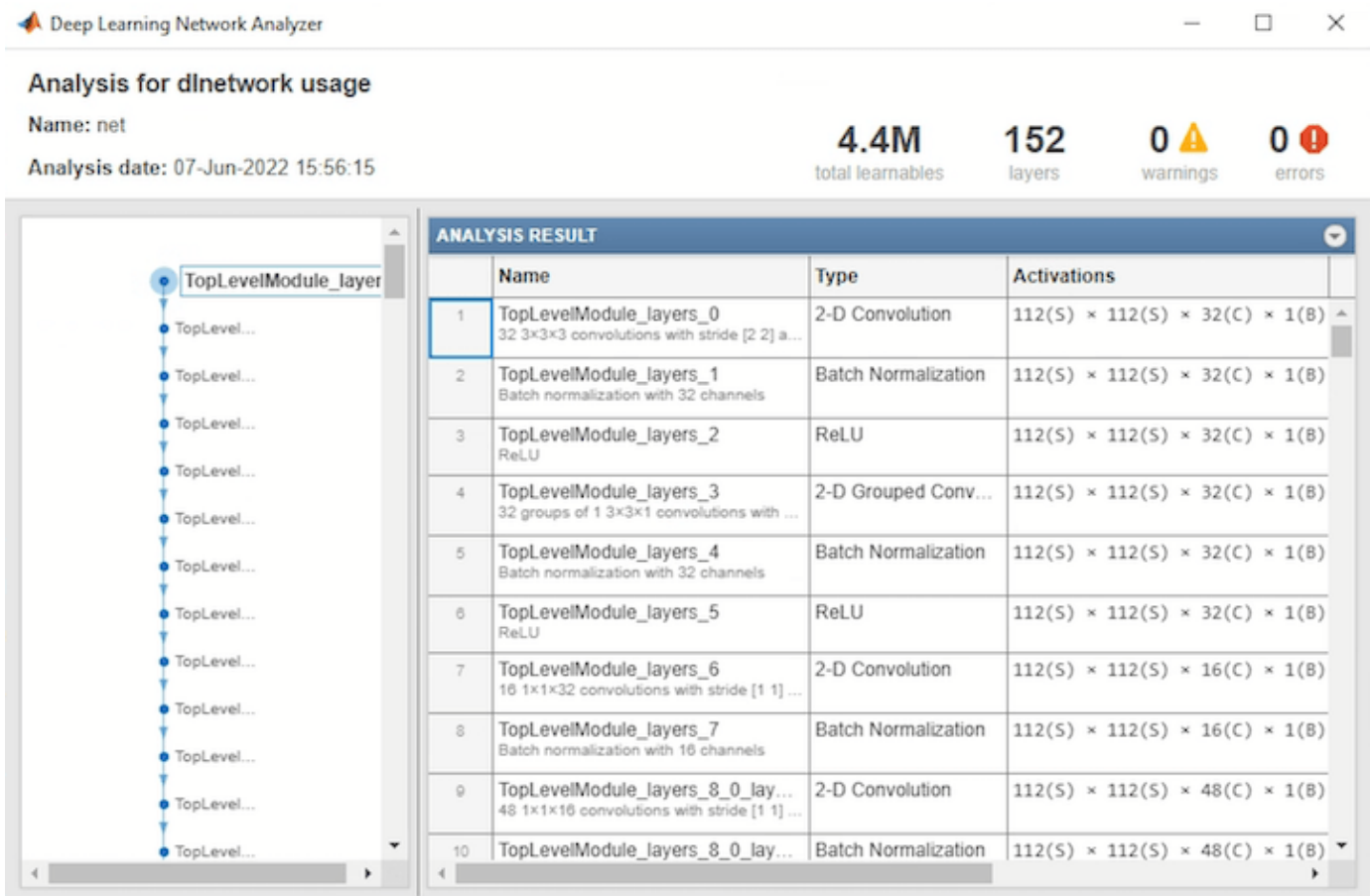
```
InputSize = [224 224 3];
X = dLarray(rand(InputSize), "SSCB");
```

Initialize the learnable parameters of the imported network by using the `initialize` function.

```
net = initialize(net,X);
```

Now the imported network is ready to use for prediction. Analyze the imported network.

```
analyzeNetwork(net)
```



## Import Network from PyTorch and Classify Image

Import a pretrained and traced PyTorch model as an uninitialized `dlnetwork` object to classify an image.

This example imports the MNASNet (Copyright© Soumith Chintala 2016) PyTorch model. MNASNet is an image classification model that is trained with images from the ImageNet database. Download the `mnasnet1_0.pt` file, which is approximately 17 MB in size, from the MathWorks website.

```
modelfile = matlab.internal.examples.downloadSupportFile("nnet", ...
    "data/PyTorchModels/mnasnet1_0.pt");
```

Import the MNASNet model by using the `importNetworkFromPyTorch` function. The function imports the model as an uninitialized `dlnetwork` object.

```
net = importNetworkFromPyTorch(modelfile)
```

Warning: Network was imported as an uninitialized `dlnetwork`. Before using the network, add input

```
inputLayer1 = imageInputLayer(<inputSize1>, Normalization="none");
net = addInputLayer(net, inputLayer1, Initialize=true);
```

```
net =  
  dlnetwork with properties:  
  
    Layers: [152x1 nnet.cnn.layer.Layer]  
    Connections: [163x2 table]  
    Learnables: [210x3 table]  
    State: [104x3 table]  
    InputNames: {'TopLevelModule_layers_0'}  
    OutputNames: {'aten__linear12'}  
    Initialized: 0
```

View summary with `summary`.

Specify the input size of the imported network and create an image input layer. Then, add the image input layer to the imported network and initialize the network by using the `addInputLayer` function.

```
InputSize = [224 224 3];  
inputLayer = imageInputLayer(InputSize,Normalization="none");  
net = addInputLayer(net,inputLayer,Initialize=true);
```

Read the image you want to classify.

```
Im = imread("peppers.png");
```

Resize the image to the input size of the network. Show the image.

```
InputSize = [224 224 3];  
Im = imresize(Im,InputSize(1:2));  
imshow(Im)
```



The inputs to MNASNet require further preprocessing. Rescale the image. Then, normalize the image by subtracting the training images mean and dividing by the training images standard deviation. For more information, see “Input Data Preprocessing”.

```
Im = rescale(Im,0,1);
```



```
meanIm = [0.485 0.456 0.406];
stdIm = [0.229 0.224 0.225];
Im = (Im - reshape(meanIm,[1 1 3]))./reshape(stdIm,[1 1 3]);
```

Convert the image to a `dIarray` object. Format the image with the dimensions "SSCB" (spatial, spatial, channel, batch).

```
Im_dIarray = dIarray(single(Im), "SSCB");
```

Get the class names from `squeezenet`, which is also trained with ImageNet images.

```
squeezeNet = squeezenet;
ClassNames = squeezeNet.Layers(end).Classes;
```

Classify the image and find the predicted label.

```
prob = predict(net, Im_dIarray);
[~, label_ind] = max(prob);
```

Display the classification result.

```
ClassNames(label_ind)
```

```
ans = categorical
      bell pepper
```

## Import Network from PyTorch and Find Generated Custom Layers

Import a pretrained and traced PyTorch model as an uninitialized `dInetwork` object. Then, find the custom layers that the software generates.

This example uses the `findCustomLayers` on page 1-1001 helper function.

This example imports the MNASNet (Copyright© Soumith Chintala 2016) PyTorch model. MNASNet is an image classification model that is trained with images from the ImageNet database. Download the `mnasnet1_0.pt` file, which is approximately 17 MB in size, from the MathWorks website.

```
modelfile = matlab.internal.examples.downloadSupportFile("nnet", ...
    "data/PyTorchModels/mnasnet1_0.pt");
```

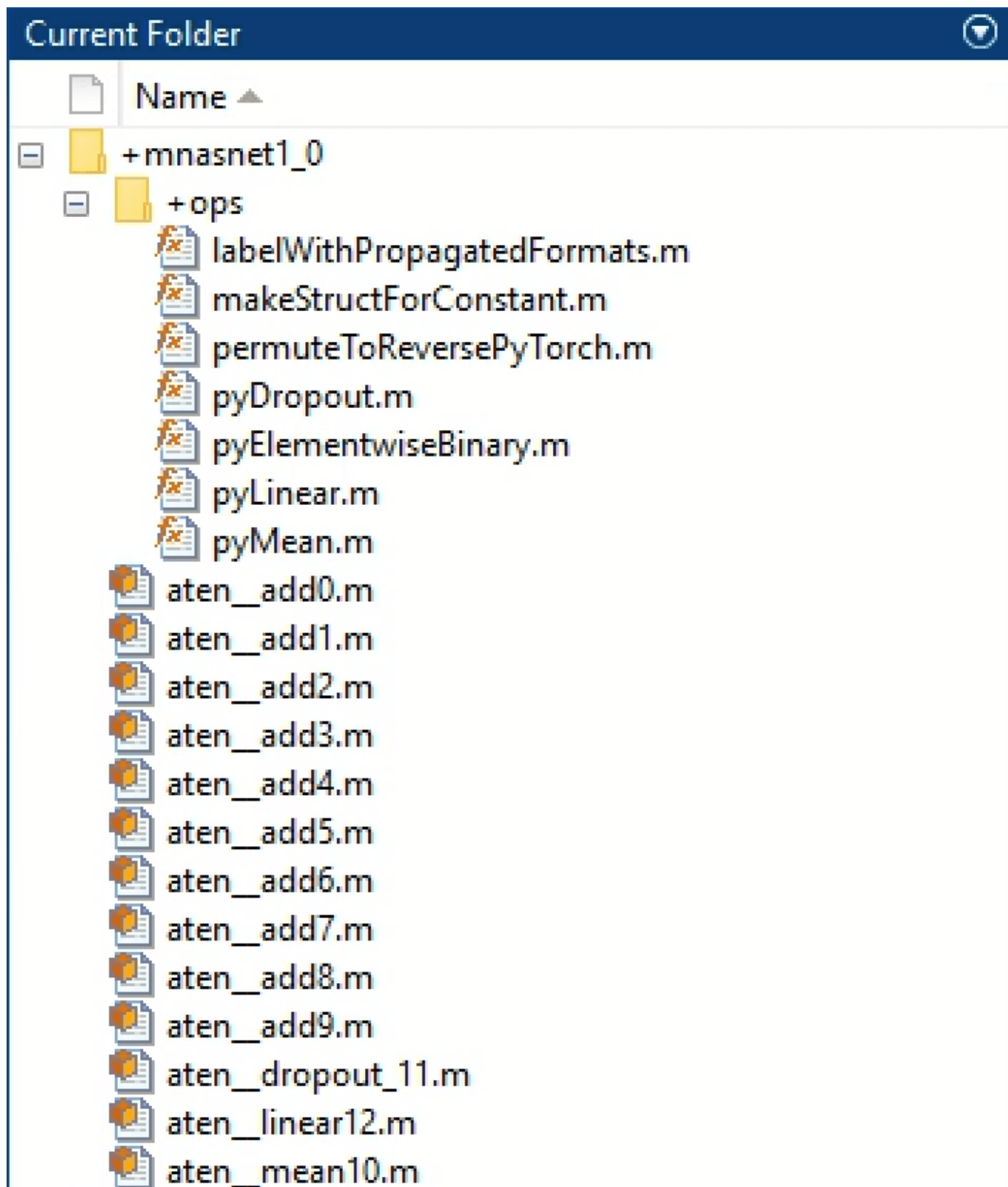
Import the MNASNet model by using the `importNetworkFromPyTorch` function. The function imports the model as an uninitialized `dInetwork` object.

```
net = importNetworkFromPyTorch(modelfile);
```

Warning: Network was imported as an uninitialized `dInetwork`. Before using the network, add input

```
inputLayer1 = imageInputLayer(<inputSizeI>, Normalization="none");
net = addInputLayer(net, inputLayer1, Initialize=true);
```

The `importNetworkFromPyTorch` function generates custom layers for the PyTorch layers that the function cannot convert to built-in MATLAB layers or functions. For more information, see "Algorithms" on page 1-1015. The software saves the automatically generated custom layers to the package `+mnasnet1_0` in the current folder and the associated functions to the subpackage `+ops`. To see the custom layers and associated functions, inspect the package.



You can also find the indices of the generated custom layers by using the `findCustomLayers` helper function. Display the custom layers.

```
ind = findCustomLayers(net.Layers, '+mnasnet1_0');
net.Layers(ind)

ans =
  13x1 Layer array with layers:

   1  'aten__add0'      Custom Layer  mnasnet1_0.aten__add0
   2  'aten__add1'      Custom Layer  mnasnet1_0.aten__add1
   3  'aten__add2'      Custom Layer  mnasnet1_0.aten__add2
   4  'aten__add3'      Custom Layer  mnasnet1_0.aten__add3
   5  'aten__add4'      Custom Layer  mnasnet1_0.aten__add4
   6  'aten__add5'      Custom Layer  mnasnet1_0.aten__add5
   7  'aten__add6'      Custom Layer  mnasnet1_0.aten__add6
   8  'aten__add7'      Custom Layer  mnasnet1_0.aten__add7
   9  'aten__add8'      Custom Layer  mnasnet1_0.aten__add8
  10  'aten__add9'      Custom Layer  mnasnet1_0.aten__add9
  11  'aten_dropout_11' Custom Layer  mnasnet1_0.aten_dropout_11
  12  'aten_linear12'   Custom Layer  mnasnet1_0.aten_linear12
  13  'aten__mean10'   Custom Layer  mnasnet1_0.aten__mean10
```

## Helper Function

This section provides the `findCustomLayers` helper function, which returns the indices of the custom layers that `importNetworkFromPyTorch` 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
```

## Train Network Imported from PyTorch to Classify New Images

This example shows how to import a network from PyTorch and train the network to classify new images. Use the `importNetworkFromPytorch` function to import the network as a uninitialized `dlnetwork` object. Train the network by using a custom training loop.

This example uses the `modelLoss` on page 1-1009, `modelPredictions` on page 1-1010, and `preprocessMiniBatchPredictors` on page 1-1011 helper functions.

This example provides the supporting file `new_fcLayer.m`. To access the supporting file, open the example in Live Editor.

## Load Data

Unzip the MerchData data set, which contains 75 images. Load the new images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation.

```
unzip("MerchData.zip");
imds = imageDatastore("MerchData", ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7);
```

The network used in this example requires input images of size 224-by-224-by-3. 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 30 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 = [224 224 3];

pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter(...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange, ...
    RandXScale=scaleRange, ...
    RandYScale=scaleRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    DataAugmentation=imageAugmenter);
```

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

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

Determine the number of classes in the training data.

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

## Import Network

Download the MNASNet (Copyright© Soumith Chintala 2016) PyTorch model. MNASNet is an image classification model that is trained with images from the ImageNet database. Download the `mnasnet1_0.pt` file, which is approximately 17 MB in size, from the MathWorks website.

```
modelfile = matlab.internal.examples.downloadSupportFile("nnet", ...
    "data/PyTorchModels/mnasnet1_0.pt");
```

Import the MNASNet model as an uninitialized `dlnetwork` object, by using the `importNetworkFromPyTorch` function.

```
net = importNetworkFromPyTorch(modelfile)
```

Warning: Network was imported as an uninitialized `dlnetwork`. Before using the network, add input

```
inputLayer1 = imageInputLayer(<inputSize1>, Normalization="none");
net = addInputLayer(net, inputLayer1, Initialize=true);
```

```
net =
  dlnetwork with properties:

    Layers: [152x1 nnet.cnn.layer.Layer]
    Connections: [163x2 table]
    Learnables: [210x3 table]
    State: [104x3 table]
    InputNames: {'TopLevelModule_layers_0'}
    OutputNames: {'aten__linear12'}
    Initialized: 0
```

View summary with `summary`.

Display the final layer of the imported network.

```
net.Layers(end)
```

```
ans =
  aten__linear12 with properties:

    Name: 'aten__linear12'
    NumInputs: 2
    InputNames: {'in' 'in_rank'}

    Learnable Parameters
    TopLevelModule_classifier_1_weight: [1280x1000 single]
    TopLevelModule_classifier_1_bias: [0.0493 -0.0804 -0.0906 -0.1006 0.1332 -0.0767 -0.0788 ...]

    State Parameters
    No properties.

    Show all properties
```

The `aten__linear12` layer is a custom layer generated by the `importNetworkFromPyTorch` function and the last learnable layer of the imported network. This layer contains information on how to combine the features that the network extracts into class probabilities and a loss value.

### Replace Final Layer

To retrain the imported network to classify new images, replace the final layers with a new fully connected layer. The new layer `new_fcLayer` is adapted to the new data set and must also be a custom layer because it has two inputs.

Initialize the `new_fcLayer` layer and replace the `aten__linear12` layer with `new_fcLayer`.

```
newLayer = new_fcLayer("fc1", "Custom Layer", ...
  {'in' 'in_rank'}, {'out'}, numClasses);
net = replaceLayer(net, "aten__linear12", newLayer);
```

Add a softmax layer to the network and connect the softmax layer to the new fully connected layer.

```
net = addLayers(net, softmaxLayer(Name="sm1"));
net = connectLayers(net, "fc1", "sm1");
```

### Add Input Layer

Add an image input layer to the network and initialize the network.

```
inputLayer = imageInputLayer(inputSize,Normalization="none");
net = addInputLayer(net,inputLayer,Initialize=true);
```

Analyze the network. View the first layer and the final layers.

```
analyzeNetwork(net)
```

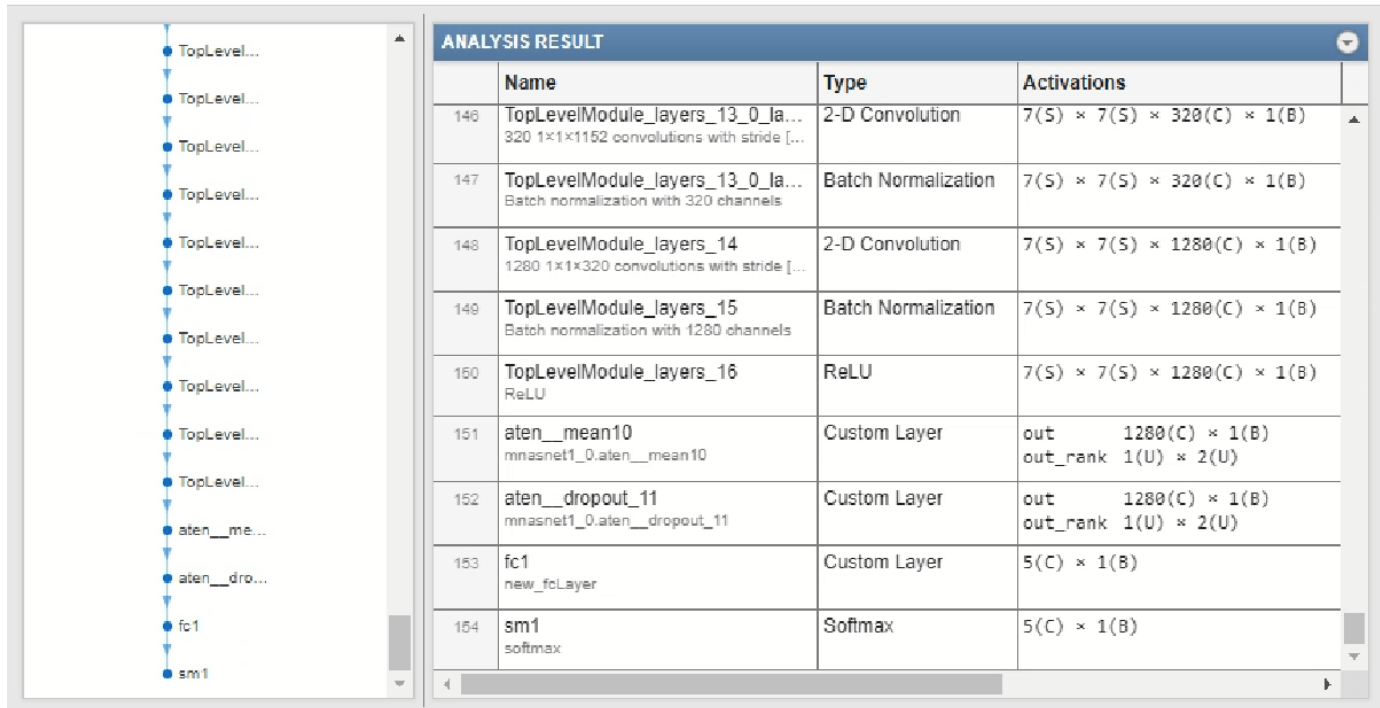
The screenshot shows the 'Deep Learning Network Analyzer' window. At the top, it displays 'Analysis for dlnetwork usage' for a network named 'net', analyzed on 13-Jun-2022 at 00:05:41. Summary statistics are shown: 3.1M total learnables, 154 layers, 0 warnings, and 0 errors. On the left, a network diagram shows an 'imageinput' layer at the top, followed by four 'TopLevel...' layers. On the right, an 'ANALYSIS RESULT' table provides details for the first three layers.

ANALYSIS RESULT			
	Name	Type	Activations
1	imageinput 224x224x3 images	Image Input	224(S) × 224(S) × 3(C) × 1(B)
2	TopLevelModule_layers_0 32 3x3 convolutions with stride [2 2] a...	2-D Convolution	112(S) × 112(S) × 32(C) × 1(B)
3	TopLevelModule_layers_1 Batch normalization with 32 channels	Batch Normalization	112(S) × 112(S) × 32(C) × 1(B)

## Analysis for dlnetwork usage

Name: net

Analysis date: 13-Jun-2022 00:05:41

3.1M  
total learnables154  
layers0 ⚠  
warnings0 ❗  
errors


	Name	Type	Activations
146	TopLevelModule_layers_13_0_la... 320 1×1×1152 convolutions with stride [...]	2-D Convolution	7(S) × 7(S) × 320(C) × 1(B)
147	TopLevelModule_layers_13_0_la... Batch normalization with 320 channels	Batch Normalization	7(S) × 7(S) × 320(C) × 1(B)
148	TopLevelModule_layers_14 1280 1×1×320 convolutions with stride [...]	2-D Convolution	7(S) × 7(S) × 1280(C) × 1(B)
149	TopLevelModule_layers_15 Batch normalization with 1280 channels	Batch Normalization	7(S) × 7(S) × 1280(C) × 1(B)
150	TopLevelModule_layers_16 ReLU	ReLU	7(S) × 7(S) × 1280(C) × 1(B)
151	aten__mean10 mnasnet1_0.aten__mean10	Custom Layer	out 1280(C) × 1(B) out_rank 1(U) × 2(U)
152	aten__dropout_11 mnasnet1_0.aten__dropout_11	Custom Layer	out 1280(C) × 1(B) out_rank 1(U) × 2(U)
153	fc1 new_fcLayer	Custom Layer	5(C) × 1(B)
154	sm1 softmax	Softmax	5(C) × 1(B)

### Define Model Loss Function

Training a deep neural network is an optimization task. By considering a neural network as a function  $f(X; \theta)$ , where  $X$  is the network input, and  $\theta$  is the set of learnable parameters, you can optimize  $\theta$  so that it minimizes some loss value based on the training data. For example, optimize the learnable parameters  $\theta$  such that, for given inputs  $X$  with corresponding targets  $T$ , they minimize the error between the predictions  $Y = f(X; \theta)$  and  $T$ .

Create the `modelLoss` function, listed in the Model Loss Function on page 1-1009 section of the example, which takes as input the `dlnetwork` object and a mini-batch of input data with corresponding targets. The function returns the loss, the gradients of the loss with respect to the learnable parameters, and the network state.

### Specify Training Options

Train for 15 epochs with a mini-batch size of 20.

```
numEpochs = 15;
miniBatchSize = 20;
```

Specify the options for SGDM optimization. Specify an initial learn rate of 0.001 with a decay of 0.005, and a momentum of 0.9.

```
initialLearnRate = 0.001;
decay = 0.005;
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 `darray` objects with the underlying type `single`. Do not format the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` object if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox).

```
mbq = minibatchqueue(augimdsTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    MiniBatchFormat=["SSCB" ""]);
```

Initialize the velocity parameter for the gradient descent with momentum (SGDM) solver.

```
velocity = [];
```

Calculate the total number of iterations for the training progress monitor.

```
numObservationsTrain = numel(imdsTrain.Files);
numIterationsPerEpoch = ceil(numObservationsTrain/miniBatchSize);
numIterations = numEpochs*numIterationsPerEpoch;
```

Initialize the `trainingProgressMonitor` object. Because the timer starts when you create the monitor object, create the object immediately after the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss", Info=["Epoch", "LearnRate"], XLabel="Iteration");
```

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 loss, gradients, and state using the `dlfeval` and `modelLoss` on page 1-1009 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.
- Update the loss, learn rate, and epoch values in the training progress monitor.
- Stop if the `Stop` property is true. The `Stop` property value of the `TrainingProgressMonitor` object changes to true when you click the **Stop** button.

```
epoch = 0;
iteration = 0;
```

```
% Loop over epochs.
while epoch < numEpochs && ~monitor.Stop
```



```
epoch = epoch + 1;

% Shuffle data.
shuffle(mbq);

% Loop over mini-batches.
while hasdata(mbq) && ~monitor.Stop

    iteration = iteration + 1;

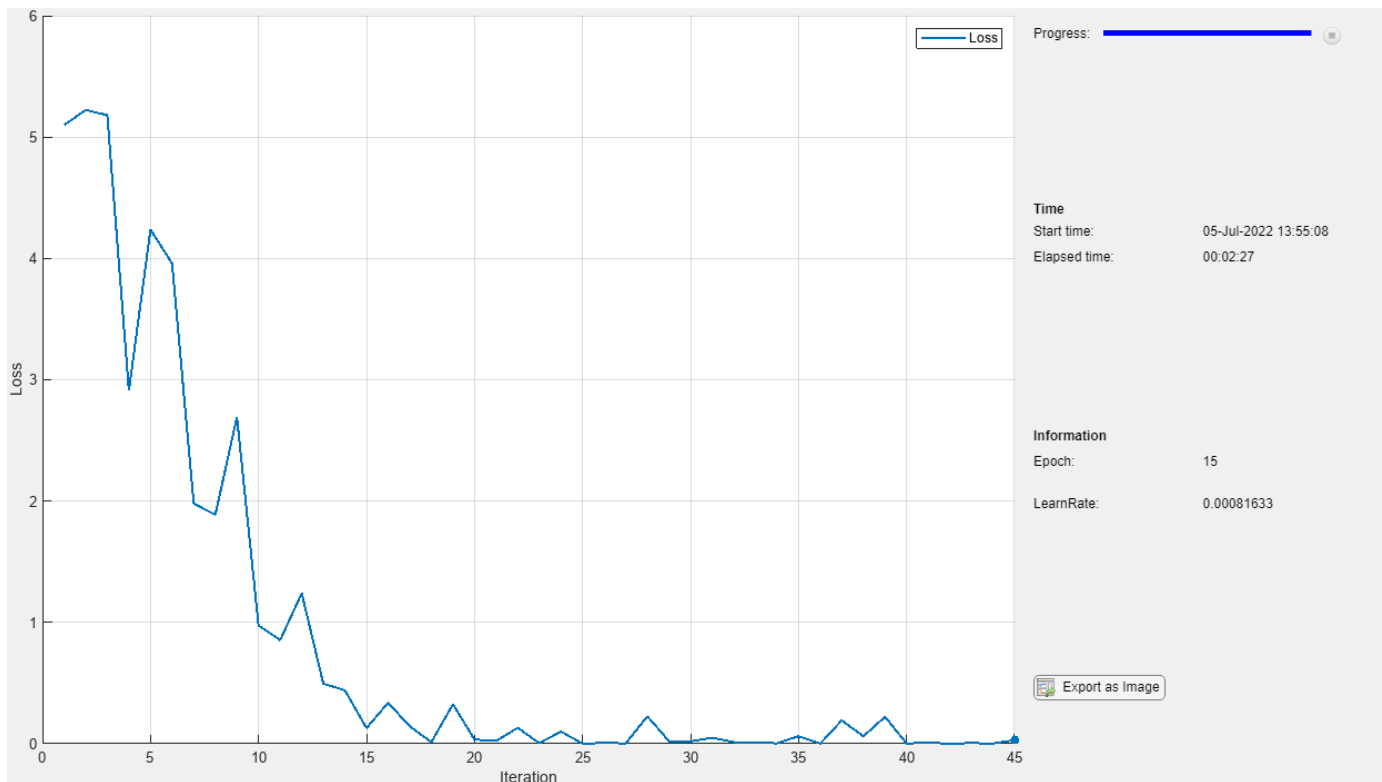
    % Read mini-batch of data.
    [X,T] = next(mbq);

    % Evaluate the model gradients, state, and loss using dlfeval and the
    % modelLoss function and update the network state.
    [loss,gradients,state] = dlfeval(@modelLoss,net,X,T);
    net.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.
    [net,velocity] = sgdmupdate(net,gradients,velocity,learnRate,momentum);

    % Update the training progress monitor.
    recordMetrics(monitor,iteration,Loss=loss);
    updateInfo(monitor,Epoch=epoch,LearnRate=learnRate);
    monitor.Progress = 100*iteration/numIterations;
end
end
```



### Classify Validation Images

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 a `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 the `modelPredictions` function, listed at the end of the example.

```
YTest = modelPredictions(net,mbqTest,classes);
```

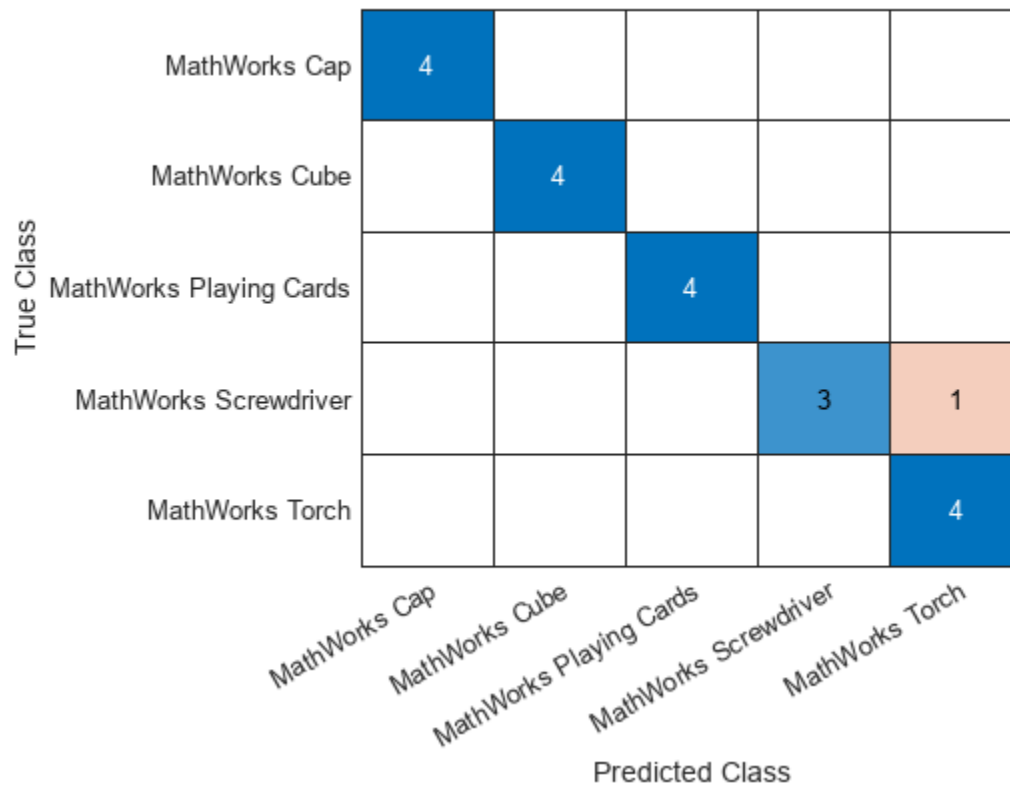
Evaluate the classification accuracy.

```
TTest = imdsValidation.Labels;
accuracy = mean(TTest == YTest)
```

```
accuracy = 0.9500
```

Visualize the predictions in a confusion chart. Large values on the diagonal indicate accurate predictions for the corresponding class. Large values on the off-diagonal indicate strong confusion between the corresponding classes.

```
figure
confusionchart(TTest,YTest)
```



## Helper Functions

### Model Loss Function

The `modelLoss` function takes as input a `dlnetwork` object `net` and a mini-batch of input data `X` with corresponding targets `T`. The function returns the loss, the gradients of the loss with respect to the learnable parameters in `net`, and the network state. To compute the gradients automatically, use the `dlgradient` function.

```
function [loss,gradients,state] = modelLoss(net,X,T)
```

```
% Forward data through network.
[Y,state] = forward(net,X);
```

```
% Calculate cross-entropy loss.
loss = crossentropy(Y,T);
```

```
% Calculate gradients of loss with respect to learnable parameters.  
gradients = dlgradient(loss,net.Learnables);
```

```
end
```

### Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `net`, a `minibatchqueue` of input data `mbq`, and the network classes. The function computes the model predictions by iterating over all the data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function Y = modelPredictions(net,mbq,classes)
```

```
Y = [];
```

```
% Loop over mini-batches.
```

```
while hasdata(mbq)  
    X = next(mbq);
```

```
    % Make prediction.
```

```
    scores = predict(net,X);
```

```
    % Decode labels and append to output.
```

```
    labels = onehotdecode(scores,classes,1)';  
    Y = [Y; labels];
```

```
end
```

```
end
```

### Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using these steps:

- 1 Preprocess the images using the `preprocessMiniBatchPredictors` function.
- 2 Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.
- 3 One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,T] = preprocessMiniBatch(dataX,dataT)
```

```
% Preprocess predictors.
```

```
X = preprocessMiniBatchPredictors(dataX);
```

```
% Extract label data from cell and concatenate.
```

```
T = cat(2,dataT{1:end});
```

```
% One-hot encode labels.
```

```
T = onehotencode(T,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 concatenating 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(dataX)

% Concatenate.
X = cat(4,dataX{1:end});

end
```

## Input Arguments

### **modelfile** — Name of PyTorch model file

character vector | string scalar

Name of the PyTorch 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 of the file. The PyTorch model must be pretrained and traced over one inference iteration.

For information on how to trace a PyTorch model, see <https://pytorch.org/docs/stable/generated/torch.jit.trace.html>.

Example: "mobilenet\_v3.pt"

### **CustomLayersPackage** — Name of custom layers package

character vector | string scalar

Name of the package in which `importNetworkFromPyTorch` saves custom layers, specified as a character vector or string scalar. `importNetworkFromPyTorch` saves the custom layers package `+CustomLayersPackage` in the current folder. If you do not specify `CustomLayersPackage`, then `importNetworkFromPyTorch` saves the custom layers in a package named `+modelfile` in the current folder. For more information on packages, see "Packages Create Namespaces".

See "Algorithms" on page 1-1015 about information on when the `importNetworkFromPyTorch` function generates a custom layer. The function saves each generated custom layer to a separate program file in `+CustomLayersPackage`. To view or edit a custom layer, open the associated program file. For more information on custom layers, see "Custom Layers".

The package `+CustomLayersPackage` can also contain the subpackage `+ops`. This subpackage contains MATLAB functions that the automatically generated custom layers use.

`importNetworkFromPyTorch` saves each MATLAB function in a separate program file in the subpackage `+ops`. The object functions of `dlnetwork`, such as the `predict` function, use these functions when interacting with the custom layers. The subpackage `+ops` might also contain "Placeholder Functions" on page 1-1014.

Example: "mobilenet\_v3"

## Output Arguments

### net — Pretrained PyTorch network

`dlnetwork` object

Pretrained PyTorch network, returned as an uninitialized `dlnetwork` object. Before using the imported network, you must add an input layer or initialize the network. For examples, see “Import Network from PyTorch and Add Input Layer” on page 1-993 and “Import Network from PyTorch and Initialize” on page 1-995.

## Limitations

- The `importNetworkFromPyTorch` function fully supports PyTorch version 1.10.0. The function can import most models created in other PyTorch versions.
- The `importNetworkFromPyTorch` function can import only image classification and segmentation models.
- You can run `importNetworkFromPyTorch` on a Windows or Mac OS platform.

## More About

### Conversion of PyTorch Layers and Functions into Built-In MATLAB Layers and Functions

The `importNetworkFromPyTorch` function supports the PyTorch layers, functions, and operators listed in this section for conversion into built-in MATLAB layers and functions with `dlarray` support. For more information on functions that operate on `dlarray` objects, see “List of Functions with `dlarray` Support”. You might observe limitations in the conversion.

#### Conversion of PyTorch Layers

This table shows the correspondence between PyTorch layers and Deep Learning Toolbox layers. In some cases, when `importNetworkFromPyTorch` cannot convert a PyTorch layer into a MATLAB layer, the software converts the PyTorch layer into a Deep Learning Toolbox function with `dlarray` support.

PyTorch Layer	Corresponding Deep Learning Toolbox Layer	Alternative Deep Learning Toolbox Function
<code>torch.nn.AdaptiveAvgPool2d</code>	<code>nnet.pytorch.layer.AdaptiveAveragePoolingLayer</code>	<code>pyAdaptiveAvgPool2d</code>
<code>torch.nn.AvgPool2d</code>	<code>averagePooling2dLayer</code>	Not applicable
<code>torch.nn.BatchNorm2d</code>	<code>batchNormalizationLayer</code>	Not applicable
<code>torch.nn.Conv1d</code>	<code>convolution1dLayer</code>	<code>pyConvolution</code>
<code>torch.nn.Conv2d</code>	<code>convolution2dLayer</code>	Not applicable
<code>torch.nn.ConvTranspose1d</code>	<code>transposedConv1dLayer</code>	<code>pyConvolution</code>
<code>torch.nn.Dropout</code>	<code>dropoutLayer</code>	Not applicable
<code>torch.nn.GroupNorm</code>	<code>groupNormalizationLayer</code>	Not applicable
<code>torch.nn.LayerNorm</code>	<code>layerNormalizationLayer</code>	Not applicable
<code>torch.nn.Linear</code>	<code>fullyConnectedLayer</code>	<code>pyLinear</code>

PyTorch Layer	Corresponding Deep Learning Toolbox Layer	Alternative Deep Learning Toolbox Function
<code>torch.nn.MaxPool2d</code>	<code>maxPooling2dLayer</code>	Not applicable
<code>torch.nn.PReLU</code>	<code>nnet.pytorch.layer.PReLU Layer</code>	Not applicable
<code>torch.nn.ReLU</code>	<code>reluLayer</code>	<code>relu</code>
<code>torch.nn.SiLU</code>	<code>swishLayer</code>	<code>pySilu</code>
<code>torch.nn.Sigmoid</code>	<code>sigmoidLayer</code>	<code>pySigmoid</code>
<code>torch.nn.Softmax</code>	<code>nnet.pytorch.layer.SoftmaxLayer</code>	<code>pySoftmax</code>
<code>torch.nn.Upsample</code>	<code>resize2dLayer</code>	<code>pyUpsample</code> (requires Image Processing Toolbox)
<code>torch.nn.UpsamplingNearest2d</code>	<code>resize2dLayer</code>	<code>pyUpsample</code> (requires Image Processing Toolbox)
<code>torch.nn.UpsamplingBilinear2d</code>	<code>resize2dLayer</code>	<code>pyUpsample</code> (requires Image Processing Toolbox)

### Conversion of PyTorch Functions

This table shows the correspondence between PyTorch functions and Deep Learning Toolbox functions.

PyTorch Function	Corresponding Deep Learning Toolbox Function
<code>torch.nn.functional.avg_pool2d</code>	<code>pyAvgPool2d</code>
<code>torch.nn.functional.conv1d</code>	<code>pyConvolution</code>
<code>torch.nn.functional.conv2d</code>	<code>pyConvolution</code>
<code>torch.nn.functional.hardsigmoid</code>	<code>pyAdaptiveAvgPool2d</code>
<code>torch.nn.functional.dropout</code>	<code>pyDropout</code>
<code>torch.nn.functional.hardsigmoid</code>	<code>pyHardSigmoid</code>
<code>torch.nn.functional.hardswish</code>	<code>pyHardSwish</code>
<code>torch.nn.functional.linear</code>	<code>pyLinear</code>
<code>torch.nn.functional.log_softmax</code>	<code>pyLogSoftmax</code>
<code>torch.nn.functional.max_pool2d</code>	<code>pyMaxPool2d</code>
<code>torch.nn.functional.relu</code>	<code>pyHardTanh</code>
<code>torch.nn.functional.silu</code>	<code>pySilu</code>
<code>torch.nn.functional.softmax</code>	<code>pySoftmax</code>

### Conversion of PyTorch Mathematical Operators

This table shows the correspondence between PyTorch mathematical operators and Deep Learning Toolbox functions. For the `cat` PyTorch operator, `importNetworkFromPyTorch` first tries to convert it to a concatenation layer and alternatively to a function.

PyTorch Operator	Corresponding Deep Learning Toolbox Layer or Function	Alternative Deep Learning Toolbox Function
<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>	<code>pyElementwiseBinary</code>	Not applicable
<code>torch.argmax</code>	<code>pyArgMax</code>	Not applicable
<code>torch.bmm</code>	<code>pyMatMul</code>	Not applicable
<code>torch.cat</code>	<code>concatenationLayer</code>	<code>pyConcat</code>
<code>torch.chunk</code>	<code>pyChunk</code>	Not applicable
<code>torch.concat</code>	<code>pyConcat</code>	Not applicable
<code>torch.matmul</code>	<code>pyMatMul</code>	Not applicable
<code>torch.max</code>	<code>pyMaxBinary/pyMaxUnary</code>	Not applicable
<code>torch.mean</code>	<code>pyMean</code>	Not applicable
<code>torch.permute</code>	<code>pyPermute</code>	Not applicable
<code>torch.pow</code>	<code>pyElementwiseBinary</code>	Not applicable
<code>torch.reshape</code>	<code>pyView</code>	Not applicable
<code>torch.size</code>	<code>pySize</code>	Not applicable
<code>torch.split</code>	<code>pySplitWithSizes</code>	Not applicable
<code>torch.stack</code>	<code>pyStack</code>	Not applicable
<code>torch.sum</code>	<code>pySum</code>	Not applicable
<code>torch.squeeze</code>	<code>pySqueeze</code>	Not applicable
<code>torch.transpose</code>	<code>pyTranspose</code>	Not applicable
<code>torch.unsqueeze</code>	<code>pyUnsqueeze</code>	Not applicable
<code>torch.zeros</code>	<code>pyZeros</code>	Not applicable

### Conversion of PyTorch Matrix Operators

This table shows the correspondence between PyTorch matrix operators and Deep Learning Toolbox functions.

PyTorch Operator	Corresponding Deep Learning Toolbox Function or Operator
Indexing (for example, <code>X[:,1]</code> )	<code>pySlice</code>
<code>torch.tensor.contiguous</code>	<code>=</code>
<code>torch.tensor.expand</code>	<code>pyExpand</code>
<code>torch.tensor.expand_as</code>	<code>pyExpandAs</code>
<code>torch.tensor.select</code>	<code>pySlice</code>
<code>torch.tensor.view</code>	<code>pyView</code>

### Placeholder Functions

When the `importNetworkFromPyTorch` function cannot convert a PyTorch layer into a built-in MATLAB layer or generate a custom layer with associated MATLAB functions, the function creates a



custom layer with a placeholder function. You must complete the placeholder function before you can use the network.

This code snippet shows the definition of a custom layer with the placeholder function `pyAtenUnsupportedOperator`.

```
classdef UnsupportedOperator < nnet.layer.Layer

function [output] = predict(obj, arg1)
% Placeholder function for aten::<unsupportedOperator>
output= pyAtenUnsupportedOperator(arg1,params);
end

end
```

## 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 format to RGB format.
  - 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` are not accessible if the package parent folder is not on the MATLAB path. For more information, see “Packages and the MATLAB Path”.
- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## Algorithms

The `importNetworkFromPyTorch` function imports a PyTorch layer into MATLAB by trying these steps in order:

- 1 The function tries to import the PyTorch layer as a built-in MATLAB layer. For more information, see “Conversion of PyTorch Layers” on page 1-1012.
- 2 The function tries to import the PyTorch layer as a built-in MATLAB function. For more information, see “Conversion of PyTorch Layers” on page 1-1012.
- 3 The function tries to import the PyTorch layer as a custom layer. `importNetworkFromPyTorch` saves the generated custom layers and the associated functions in the package `+modelfile`. For an example, see “Import Network from PyTorch and Find Generated Custom Layers” on page 1-999.
- 4 The function imports the PyTorch layer as a custom layer with a placeholder function. For more information, see “Placeholder Functions” on page 1-1014.

In the first three cases, the imported network is ready for prediction after you initialize it.

## **Version History**

**Introduced in R2022b**

### **See Also**

`importTensorFlowNetwork` | `importONNXNetwork` | `exportNetworkToTensorFlow` | `exportONNXNetwork` | `dlnetwork` | `dlarray` | `addInputLayer`

### **Topics**

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Tips on Importing Models from TensorFlow, PyTorch, and ONNX”

“Pretrained Deep Neural Networks”

“Deploy Imported TensorFlow Model with MATLAB Compiler”

# 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 1-1028.

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 1-740).

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 as Function

Import an ONNX network as a function. 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";
```

Import the network as a function to generate a model function that you can readily use for deep learning tasks.

```
params = importONNXFunction(modelfile,"shufflenetFcn")
```

Function containing the imported ONNX network architecture was saved to the file `shufflenetFcn.m`. To learn how to use this function, type: `help shufflenetFcn`.

```
params =  
    ONNXParameters with properties:  
  
        Learnables: [1x1 struct]  
        Nonlearnables: [1x1 struct]  
        State: [1x1 struct]  
        NumDimensions: [1x1 struct]  
        NetworkFunctionName: 'shufflenetFcn'
```

`importONNXFunction` returns the `ONNXParameters` object `params`, which contains the network parameters, and the model function `shufflenetFcn`, 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`.

Deep Learning Toolbox Converter for ONNX Model Format also provides the `importONNXNetwork` and `importONNXLayers` functions, which you can use to import a pretrained ONNX network. For information on which import function best suits different scenarios, see “Select Function to Import ONNX Pretrained Network”.

## 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
```

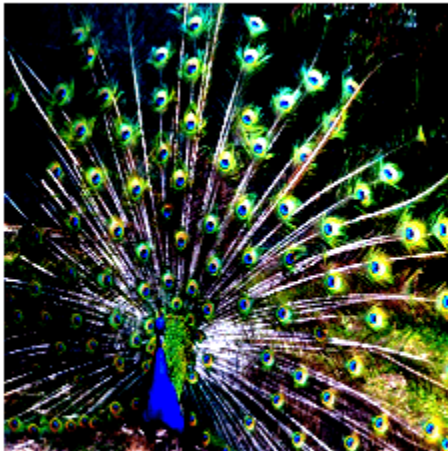
Resize the image to the input size of the network. Show the image.

```
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 = 1x1 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 1-1025.

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 = onhotencode(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")
```

Function containing the imported ONNX network architecture was saved to the file `squeezenetFcn.m`. To learn how to use this function, type: `help squeezenetFcn`.

```
params =
  ONNXParameters with properties:
    Learnables: [1x1 struct]
    Nonlearnables: [1x1 struct]
    State: [1x1 struct]
    NumDimensions: [1x1 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: [1x1x512x1000 darray]
```

```
conv10_B: [1000x1 darray]
```



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");
```

The network is ready for training. Specify the training options.

```
velocity = [];
numEpochs = 5;
miniBatchSize = 16;
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;
```

Calculate the total number of iterations for the training progress monitor.

```
numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
numIterations = numEpochs*numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object immediately after the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

Train the network.

```
epoch = 0;
iteration = 0;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.
```

```
% Loop over epochs.
```

```
while epoch < numEpochs && ~monitor.Stop
```

```
    epoch = epoch + 1;
```

```
    % Shuffle data.
```

```
    idx = randperm(numObservations);
    XTrain = XTrain(:,:,,idx);
    YTrain = YTrain(:,idx);
```

```
    % Loop over mini-batches.
```

```
    i = 0;
    while i < numIterationsPerEpoch && ~monitor.Stop
        i = i + 1;
        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

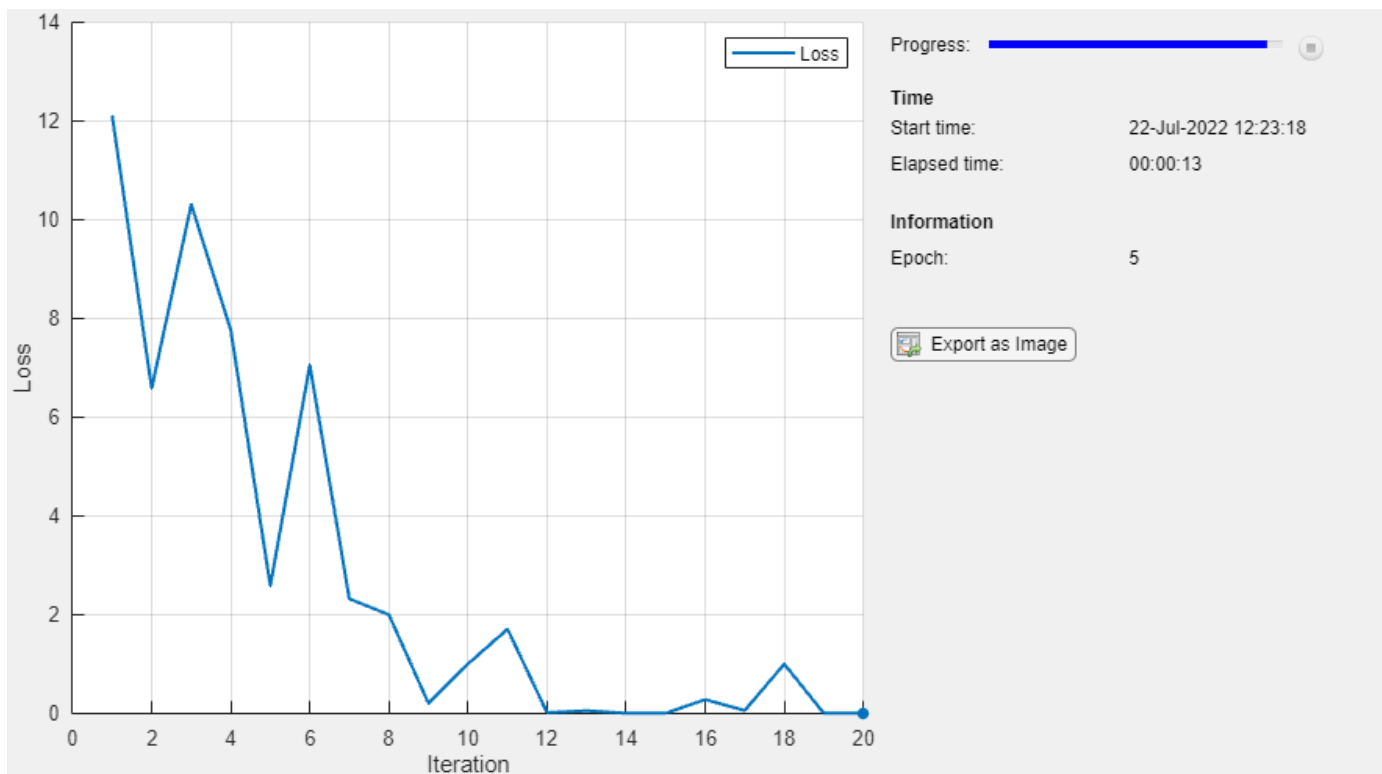
% Evaluate the model gradients and loss using dlfeval and the
% modelGradients function.
[gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
params.State = state;

% 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);

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch);
monitor.Progress = 100 * iteration/numIterations;
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 1-1027.

`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 `'hwc'`, `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, ~, iTtoInv] = 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(iTtoInv);
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 1-1028.

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 <code>darray</code> .
params	Network parameters, specified as an <code>ONNXParameters</code> object.

### Name-Value Pair Arguments

Argument name	Description
'Training'	<p>Training option, specified as 'false' (default) or 'true'.</p> <ul style="list-style-type: none"> <li>• Set value to 'false' to use <code>ONNXFunction</code> to predict. For an example, see “Predict Using Imported ONNX Function” on page 1-1018.</li> <li>• Set value to 'true' to use <code>ONNXFunction</code> to train. For an example, see “Train Imported ONNX Function Using Custom Training Loop” on page 1-1020.</li> </ul>
'InputDataPermutation'	<p>Permutation applied to the dimension ordering of input X, specified as 'auto' (default), 'none', a numeric vector, or a cell array.</p> <p>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.</p> <ul style="list-style-type: none"> <li>• 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 1-1030.</li> <li>• Assign the value 'none' to pass X in the original ordering.</li> <li>• Assign a numeric vector value to customize the input dimension ordering; for example, <code>[4 3 1 2]</code>. For an example, see “Sequence Classification Using Imported ONNX Function” on page 1-1025.</li> <li>• Assign a cell array value for multiple inputs; for example, <code>{[3 2 1], 'none'}</code>.</li> </ul>

Argument name	Description
'OutputDataPermutation'	<p>Permutation applied to the dimension ordering of output Y, specified as 'auto' (default), 'none', a numeric vector, or a cell array.</p> <p>Assign a value to the name-value pair argument 'OutputDataPermutation' to match the dimension ordering of the imported ONNX model.</p> <ul style="list-style-type: none"> <li>• Assign the value 'auto' to return Y in Deep Learning Toolbox ordering. For more details, see “Automatic Output Data Permutation” on page 1-1031.</li> <li>• Assign the value 'none' to return Y in ONNX ordering. For an example, see “Sequence Classification Using Imported ONNX Function” on page 1-1025.</li> <li>• Assign a numeric vector value to customize the output dimension ordering; for example, [3 4 2 1].</li> <li>• Assign a cell array value for multiple outputs; for example, {[3 2 1], 'none'}.</li> </ul>

**Output Arguments**

Argument	Description
Y	<p>Output data, returned as an array or dlarray.</p> <ul style="list-style-type: none"> <li>• If X is an array or you use ONNXFunction to predict, Y is a array.</li> <li>• If X is a dlarray or you use ONNXFunction for training, Y is a dlarray.</li> </ul>
state	<p>Updated network state, specified as a structure.</p> <p>The network state contains information remembered by the network between iterations and updated across multiple training batches.</p>

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.

<b>ONNX Operators Supported by importONNXFunction</b>	<b>importONNXNetwork and importONNXLayers Support</b>
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
GRU	Yes
Hardmax	No
Identity	Yes
If	No
InstanceNormalization	Yes

<b>ONNX Operators Supported by importONNXFunction</b>	<b>importONNXNetwork and importONNXLayers Support</b>
LeakyRelu	Yes
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

<b>ONNX Operators Supported by importONNXFunction</b>	<b>importONNXNetwork and importONNXLayers Support</b>
Shape	No
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”.

## Version History

Introduced in R2020b

## 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”](#)

# 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 1-1052.

`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 1-1058.

---

`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

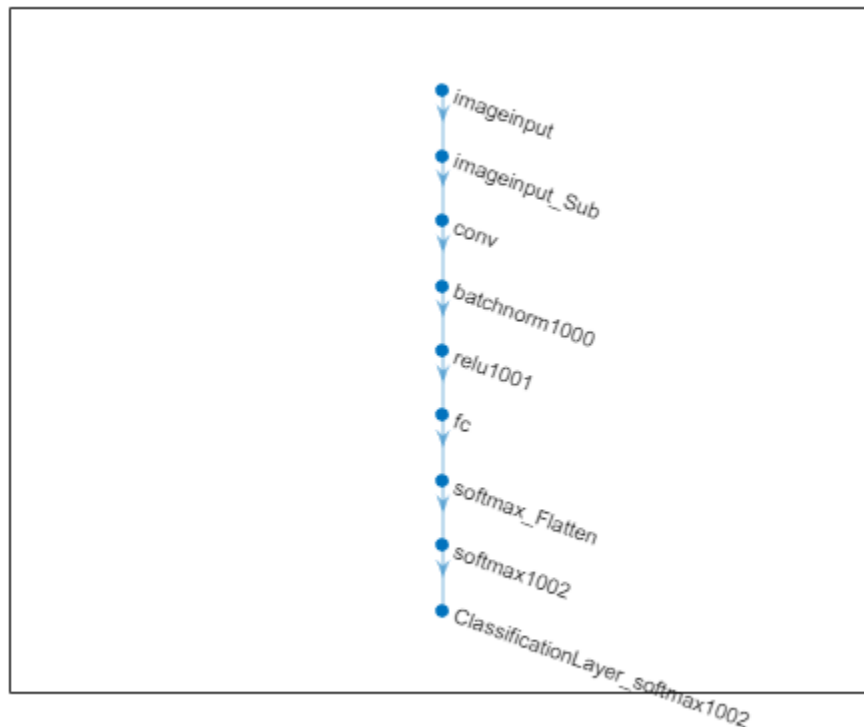
package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file "simplenet.onnx" at the command line. If the support package is installed, then the function returns a LayerGraph object.

```
modelfile = "simplenet.onnx";
lgraph = importONNXLayers(modelfile)

lgraph =
  LayerGraph with properties:
    Layers: [9x1 nnet.cnn.layer.Layer]
    Connections: [8x2 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: [70x1 nnet.cnn.layer.Layer]
        Connections: [77x2 table]
        InputNames: {'data'}
        OutputNames: {'ClassificationLayer_prob'}
```

Analyze the imported network architecture.

```
analyzeNetwork(lgraph)
```

The screenshot shows the 'Deep Learning Network Analyzer' window. The title bar reads 'Deep Learning Network Analyzer'. Below the title bar, the analysis is for 'trainNetwork usage' with the name 'lgraph' and an analysis date of '08-Jul-2021 22:43:35'. On the right side, there are three summary statistics: 70 layers, 0 warnings, and 0 errors.

The main area is divided into two panes. The left pane shows a network diagram starting with a 'data' input node, followed by 'data\_Sub', 'conv1', 'relu\_conv1', 'pool1', 'fire2\_squeeze1x1', 'fire2\_relu\_squeeze1x1', 'fire2\_expand1x1', 'fire2\_concat', 'fire3\_squeeze1x1', 'fire3\_relu\_squeeze1x1', and 'fire3\_expand1x1'. The right pane is titled 'ANALYSIS RESULT' and contains a table with the following data:

	Name	Type	Activations	Learnables
1	data 227×227×3 images	Image Input	227×227×3	-
2	data_Sub Applies an elementwis...	Elementwise Affine	227×227×3	-
3	conv1 64 3×3×3 convolutions ...	Convolution	113×113×64	Weights 3×3×3×64 Bias 1×1×64
4	relu_conv1 ReLU	ReLU	113×113×64	-
5	pool1 3×3 max pooling with s...	Max Pooling	56×56×64	-
6	fire2_squeeze1x1 16 1×1×64 convolution...	Convolution	56×56×16	Weights 1×1×64×16 Bias 1×1×16
7	fire2_relu_squeeze1x1 ReLU	ReLU	56×56×16	-
8	fire2_expand1x1 64 1×1×16 convolution...	Convolution	56×56×64	Weights 1×1×16×64 Bias 1×1×64
9	fire2_relu_expand1x1 ReLU	ReLU	56×56×64	-
10	fire2_expand3x3 64 3×3×16 convolution...	Convolution	56×56×64	Weights 3×3×16×64 Bias 1×1×64
11	fire2_relu_expand3x3 ReLU	ReLU	56×56×64	-



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: [70x1 nnet.cnn.layer.Layer]
      Connections: [77x2 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 = 1x3
    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: [70x1 nnet.cnn.layer.Layer]
```

```
Connections: [77x2 table]
InputNames: {'data'}
OutputNames: {1x0 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 = 1x3
```

```
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 `darray`. 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_darray = darray(single(I), "SSCB");
```

Classify the sample image and find the predicted label.

```
prob = predict(dlnet,I_darray);
[~,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 1-1044.

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: [173x1 nnet.cnn.layer.Layer]  
    Connections: [188x2 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 =  
    16x1 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: [173x1 nnet.cnn.layer.Layer]
      Connections: [188x2 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 placeholder 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 replaced by the following operators:

```
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: [6x1 nnet.cnn.layer.Layer]
    Connections: [5x2 table]
    InputNames: {'sequenceinput'}
    OutputNames: {1x0 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 with
    2 'lstm1000'              LSTM                   LSTM with 100 hid
    3 'fc_MatMul'            Fully Connected        9 fully connected
    4 'fc_Add'                Elementwise Affine     Applies an element
    5 'Flatten_To_SoftmaxLayer1005' lstmNet.Flatten_To_SoftmaxLayer1005 lstmNet.Flatten_To
    6 'OutputLayer_softmax1001' PLACEHOLDER LAYER     Placeholder for 'a
```

```
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 with
    2 'lstm1000'              LSTM                   LSTM with 100 hid
    3 'fc_MatMul'            Fully Connected        9 fully connected
    4 'fc_Add'                Elementwise Affine     Applies an element
    5 'Flatten_To_SoftmaxLayer1005' lstmNet.Flatten_To_SoftmaxLayer1005 lstmNet.Flatten_To
    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";
```

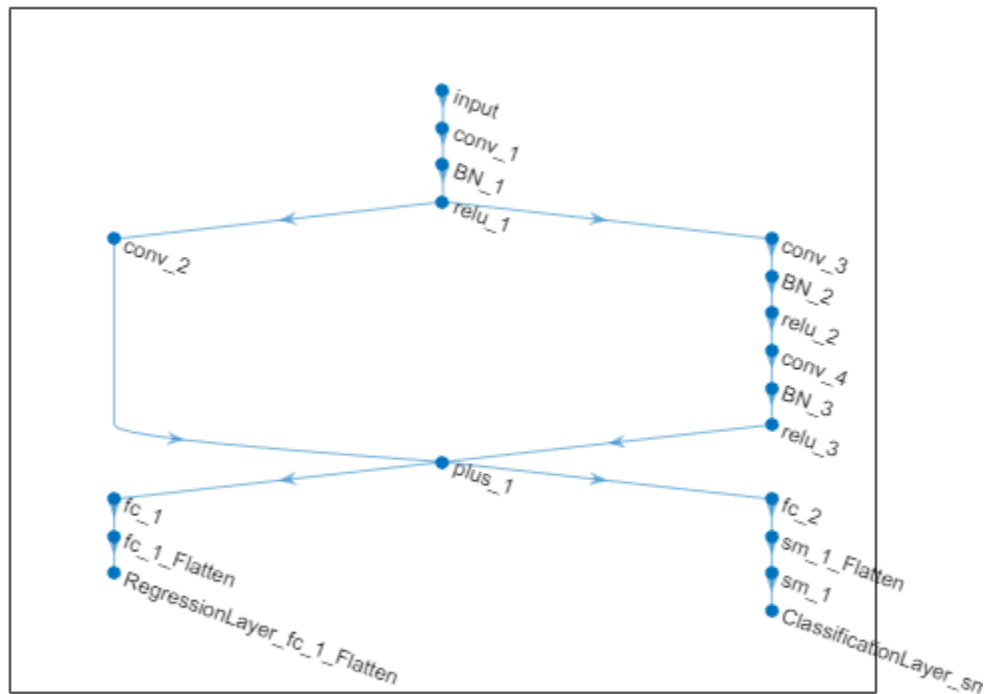
Import the layers and weights from `modelfile`. The network in `digitsMIMO.onnx` has two output layers: one classification layer (`ClassificationLayer_sm_1`) to classify digits and one regression layer (`RegressionLayer_fc_1_Flatten`) to compute the mean squared error for the predicted angles of the 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 =

19x1 Layer array with layers:

1	'input'	Image Input	28x28x1 images
2	'conv_1'	Convolution	16 5x5x1 convolutions with stride 1
3	'BN_1'	Batch Normalization	Batch normalization with 16 channels
4	'relu_1'	ReLU	ReLU
5	'conv_2'	Convolution	32 1x1x16 convolutions with stride 1
6	'conv_3'	Convolution	32 3x3x16 convolutions with stride 1
7	'BN_2'	Batch Normalization	Batch normalization with 32 channels
8	'relu_2'	ReLU	ReLU
9	'conv_4'	Convolution	32 3x3x32 convolutions with stride 1
10	'BN_3'	Batch Normalization	Batch normalization with 32 channels
11	'relu_3'	ReLU	ReLU
12	'plus_1'	Addition	Element-wise addition of 2 inputs
13	'fc_1'	Convolution	1 14x14x32 convolutions with stride 1
14	'fc_2'	Convolution	10 14x14x32 convolutions with stride 1
15	'sm_1_Flatten'	ONNX Flatten	Flatten activations into 1-D as vector
16	'sm_1'	Softmax	softmax
17	'fc_1_Flatten'	ONNX Flatten	Flatten activations into 1-D as vector
18	'ClassificationLayer_sm_1'	Classification Output	crossentropy
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: [19x1 nnet.cnn.layer.Layer]
  Connections: [19x2 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 "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 1-1054.

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 `InputDataFormats` 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 1-1054.

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 1-1054.

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 1-1052), then `importONNXLayers` inserts a placeholder layer in place of each unsupported layer. For more information, see “Tips” on page 1-1058.

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 7.
  - The function supports ONNX operator sets 6 to 14.

---

**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.

<b>ONNX Operator</b>	<b>Deep Learning Toolbox Layer</b>
Add	additionLayer or nnet.onnx.layer.ElementwiseAffineLayer
AveragePool	averagePooling1dLayer or averagePooling2dLayer
BatchNormalization	batchNormalizationLayer
Concat	concatenationLayer
Constant	None (Imported as weights)
Conv*	convolution1dLayer or convolution2dLayer
ConvTranspose	transposedConv2dLayer
Dropout	dropoutLayer
Elu	eluLayer
Gemm	fullyConnectedLayer if ONNX network is recurrent, otherwise nnet.onnx.layer.FlattenLayer followed by convolution2dLayer
GlobalAveragePool	globalAveragePooling1dLayer or globalAveragePooling2dLayer
GlobalMaxPool	globalMaxPooling1dLayer or 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	maxPooling1dLayer or maxPooling2dLayer
Mul	multiplicationLayer
Relu	reluLayer or clippedReluLayer
Sigmoid	sigmoidLayer
Softmax	softmaxLayer
Sum	additionLayer
Tanh	tanhLayer

\* If importONNXLayers imports the Conv ONNX operator as a convolution2dLayer and the Conv operator is a vector with only two elements [p1, p2], importONNXLayers sets the Padding option of convolution2dLayer to [p1, p2, p1, p2].

ONNX Operator	ONNX Importer Custom Layer
Clip	<code>nnet.onnx.layer.ClipLayer</code>
Div	<code>nnet.onnx.layer.ElementwiseAffineLayer</code>
Flatten	<code>nnet.onnx.layer.FlattenLayer</code> or <code>nnet.onnx.layer.Flatten3dLayer</code>
Identity	<code>nnet.onnx.layer.IdentityLayer</code>
ImageScaler	<code>nnet.onnx.layer.ElementwiseAffineLayer</code>
PReLU	<code>nnet.onnx.layer.PReLULayer</code>
Reshape	<code>nnet.onnx.layer.FlattenLayer</code>
Sub	<code>nnet.onnx.layer.ElementwiseAffineLayer</code>

ONNX Operator	Image Processing Toolbox
DepthToSpace	<code>depthToSpace2dLayer</code>
Resize	<code>resize2dLayer</code> or <code>resize3dLayer</code>
SpaceToDepth	<code>spaceToDepthLayer</code>
Upsample	<code>resize2dLayer</code> or <code>resize3dLayer</code>

### 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- $c$ -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	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	<i>h</i> -by- <i>w</i> -by- <i>c</i> -by- <i>s</i> -by- <i>n</i> array, where <i>h</i> , <i>w</i> , <i>c</i> and <i>n</i> correspond to the height, width, and number of channels of the image, respectively, <i>s</i> is the sequence length, and <i>n</i> is the number of image sequence observations	2-D image sequence	sequenceInputLayer
TBCSSS	SSSCBT	<i>h</i> -by- <i>w</i> -by- <i>d</i> -by- <i>c</i> -by- <i>s</i> -by- <i>n</i> array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> correspond to the height, width, depth, and number of channels of the image, respectively, <i>s</i> is the sequence length, and <i>n</i> 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

**Generate Code for Imported Network Architecture**

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or

C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).

- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importONNXLayers` returns the network architecture `lgraph` as a `LayerGraph` object. For code generation, you must first convert the imported `LayerGraph` object to a network. Convert a `LayerGraph` object to a `DAGNetwork` or `dlnetwork` object by using `assembleNetwork` or `dlnetwork`. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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.
 

```
net = dlupdate(@gpuArray,net)
```
- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 1-1052) 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 1-1047.
- 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 format to RGB format.
  - 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”.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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”.

## Version History

### Introduced in R2018a

#### **R2021b: 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.

## **R2021b: 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.

## **R2021b: Names of imported layers might differ**

*Behavior changed in R2021b*

The layer names of an imported layer graph might differ from previous releases. To update your code, replace the existing name of a layer with the new name or lgraph.Layers(n).Name.

## **References**

[1] *Open Neural Network Exchange*. <https://github.com/onnx/>.

[2] *ONNX*. <https://onnx.ai/>.

## **See Also**

importONNXNetwork | importONNXFunction | importTensorFlowLayers | importTensorFlowNetwork | importNetworkFromPyTorch | exportONNXNetwork | exportNetworkToTensorFlow | findPlaceholderLayers | replaceLayer | assembleNetwork

## **Topics**

"Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX"

"Tips on Importing Models from TensorFlow, PyTorch, and ONNX"

"Pretrained Deep Neural Networks"

"Define Custom Deep Learning Layers"

"Assemble Network from Pretrained Keras Layers"

"Select Function to Import ONNX Pretrained Network"

# 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 1-1073.

`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 1-1079.

---

`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.

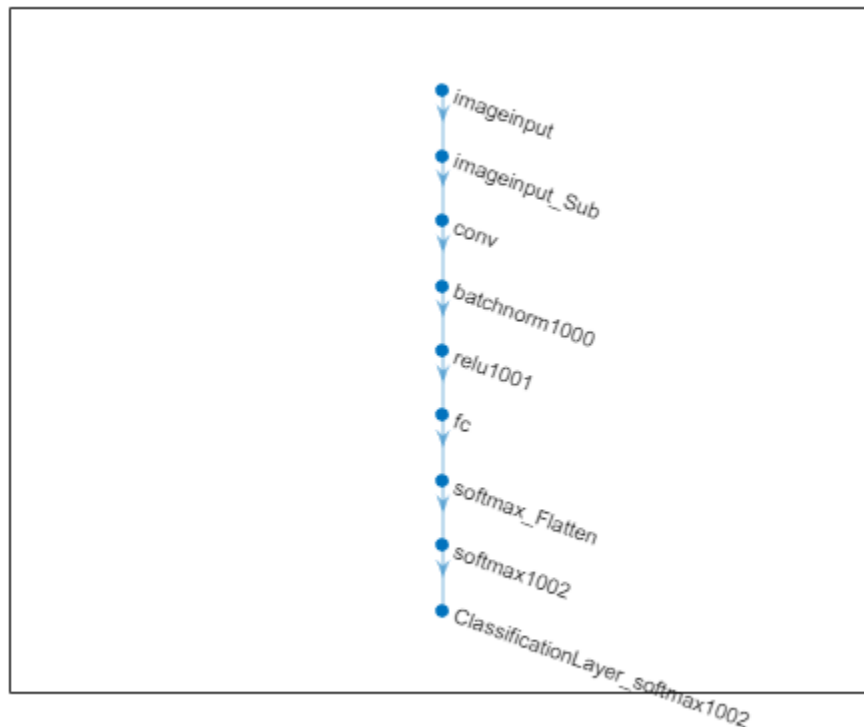
```
modelfile = "simplenet.onnx";
net = importONNXNetwork(modelfile)

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;
```

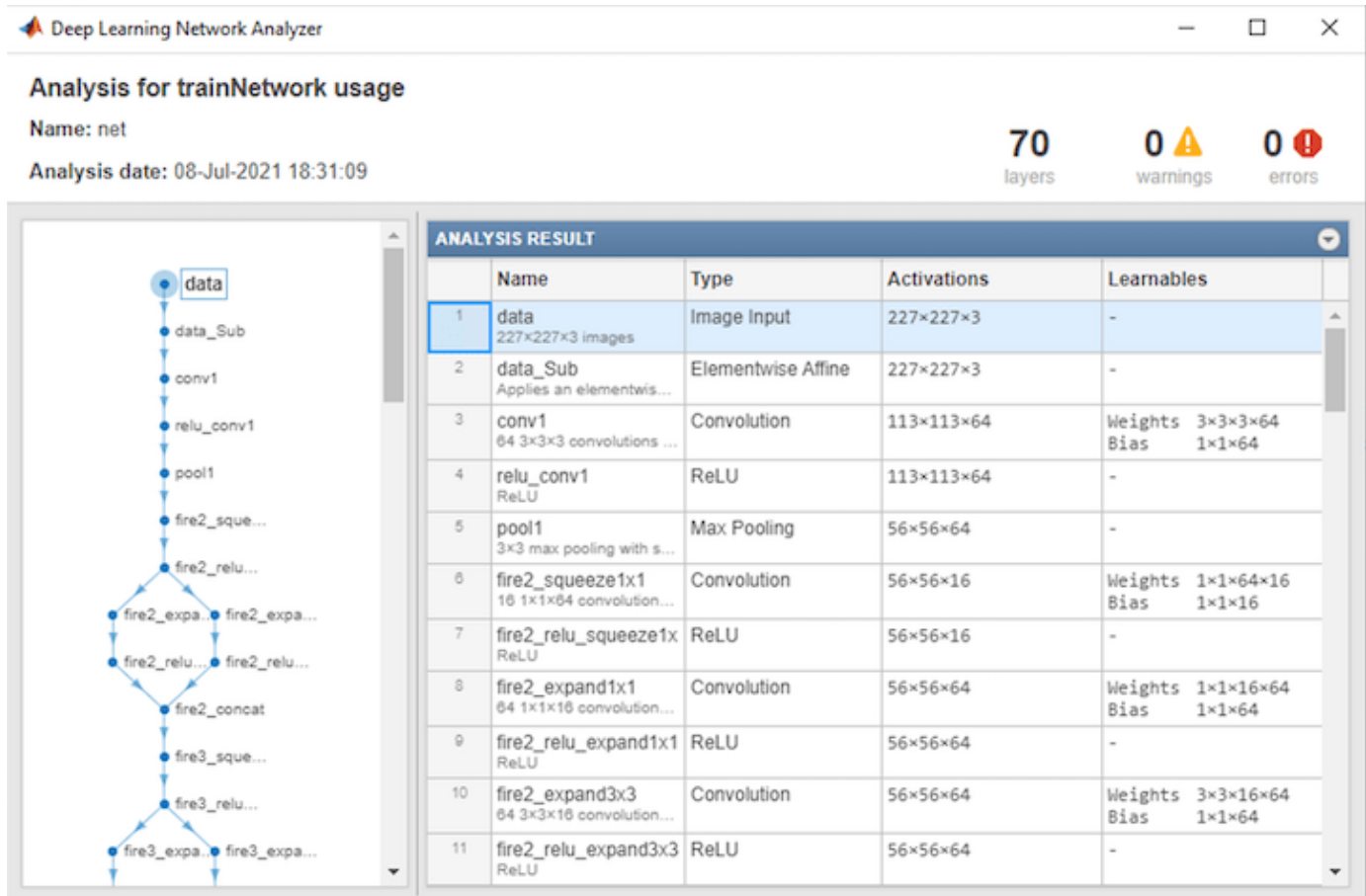
Import the pretrained squeezeNet.onnx model, and specify the classes. By default, importONNXNetwork imports the network as a DAGNetwork object.

```
net = importONNXNetwork("squeezeNet.onnx", Classes=ClassNames)
```

```
net =
  DAGNetwork with properties:
    Layers: [70x1 nnet.cnn.layer.Layer]
    Connections: [77x2 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: [70x1 nnet.cnn.layer.Layer]  
 Connections: [77x2 table]  
 Learnables: [52x3 table]  
    State: [0x3 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 = 1x3  
  
    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 `darray`. 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_darray = darray(single(I), "SSCB");
```

Classify the sample image and find the predicted label.

```
prob = predict(net, I_darray);
[~, 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 1-1068.

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: [173x1 nnet.cnn.layer.Layer]
Connections: [188x2 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 =
    16x1 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 = 1x3
      792      1056      3
```

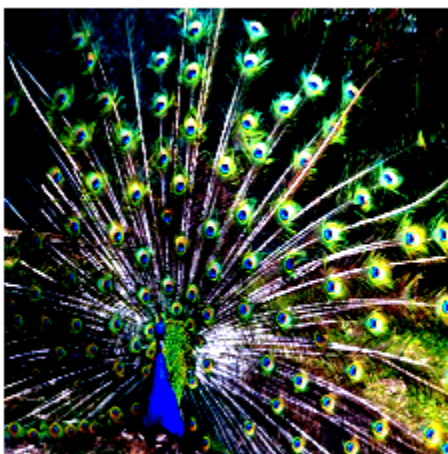
Resize the image to the input size of the network. Show the image.

```
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

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.

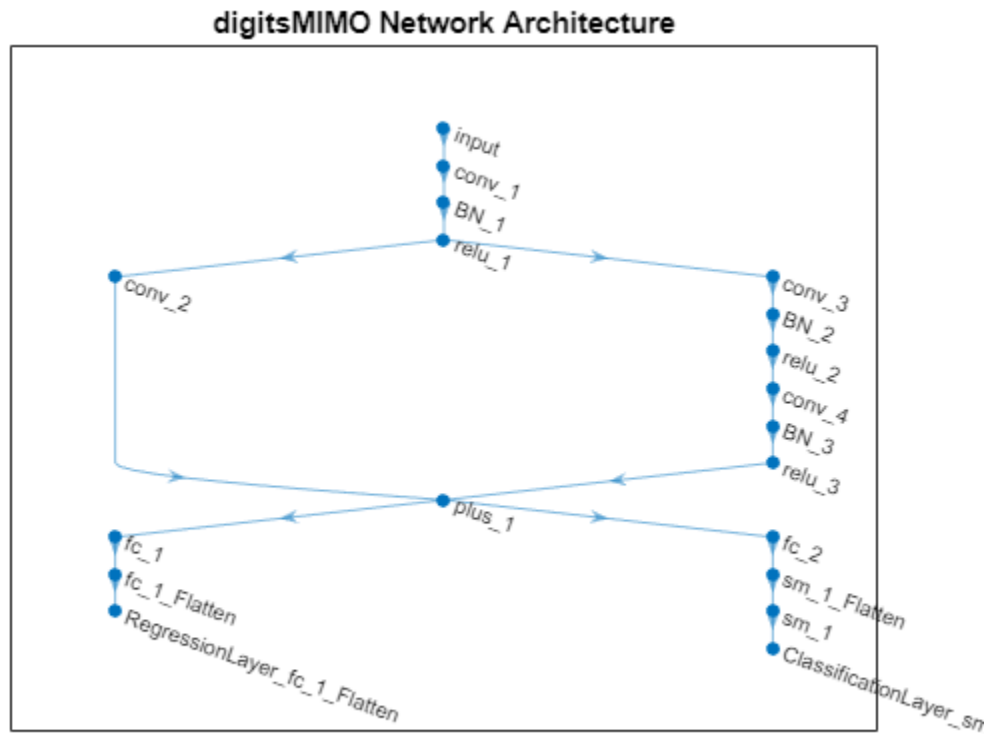
```
modelfile = "digitsMIMO.onnx";
net = importONNXNetwork(modelfile)

net =
    DAGNetwork with properties:

        Layers: [19x1 nnet.cnn.layer.Layer]
    Connections: [19x2 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 “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 1-1074.

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 1-1074.

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 1-1074.

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 1-1073), 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 1-1079.

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 7.
  - The function supports ONNX operator sets 6 to 14.

---

**Note** If you import an exported network, layers of the reimplemented 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	averagePooling1dLayer or averagePooling2dLayer
BatchNormalization	batchNormalizationLayer
Concat	concatenationLayer
Constant	None (Imported as weights)
Conv*	convolution1dLayer or convolution2dLayer
ConvTranspose	transposedConv2dLayer
Dropout	dropoutLayer
Elu	eluLayer
Gemm	fullyConnectedLayer if ONNX network is recurrent, otherwise nnet.onnx.layer.FlattenLayer followed by convolution2dLayer
GlobalAveragePool	globalAveragePooling1dLayer or globalAveragePooling2dLayer
GlobalMaxPool	globalMaxPooling1dLayer or globalMaxPooling2dLayer
GRU	gruLayer

ONNX Operator	Deep Learning Toolbox Layer
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	maxPooling1dLayer or maxPooling2dLayer
Mul	multiplicationLayer
Relu	reluLayer or clippedReluLayer
Sigmoid	sigmoidLayer
Softmax	softmaxLayer
Sum	additionLayer
Tanh	tanhLayer

\* If importONNXNetwork imports the Conv ONNX operator as a convolution2dLayer and the Conv operator is a vector with only two elements [p1,p2], importONNXNetwork sets the Padding option of convolution2dLayer to [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- $c$ -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	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

### Generate Code for Imported Network

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or

C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).

- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importONNXNetwork` returns the network `net` as a `DAGNetwork` or `dlnetwork` object. Both these objects support code generation. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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.

```
net = dlupdate(@gpuArray,net)
```
- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 format to RGB format.
  - 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”.

- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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 1-1073) 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”.

## Version History

### Introduced in R2018a

#### **R2021b: 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`.

#### **R2021b: 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.

### **R2021b: Layer names of imported network might differ**

*Behavior changed in R2021b*

The layer names of an imported network might differ from previous releases. To update your code, replace the existing name of a layer with the new name or `net.Layers(n).Name`.

## **References**

[1] *Open Neural Network Exchange*. <https://github.com/onnx/>.

[2] *ONNX*. <https://onnx.ai/>.

## **See Also**

`importONNXLayers` | `importONNXFunction` | `importTensorFlowNetwork` |  
`importTensorFlowLayers` | `importNetworkFromPyTorch` | `exportONNXNetwork` |  
`exportNetworkToTensorFlow`

## **Topics**

"Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX"

"Tips on Importing Models from TensorFlow, PyTorch, and ONNX"

"Pretrained Deep Neural Networks"

"Inference Comparison Between ONNX and Imported Networks for Image Classification"

"Select Function to Import ONNX Pretrained Network"

# 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 1-1089.

`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 1-1095.

---

`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: [13x1 nnet.cnn.layer.Layer]
  Connections: [13x2 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: [13x1 nnet.cnn.layer.Layer]
  Connections: [13x2 table]
   InputNames: {'input_1'}
  OutputNames: {'ClassificationLayer_activation_1'}
```

Read the image you want to classify.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'ndemos','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}])
```

**Classification result 5**



### 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: [12x1 nnet.cnn.layer.Layer]
    Connections: [12x2 table]
    InputNames: {'input_1'}
    OutputNames: {1x0 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 = 1x2
     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 = 1x3
     28     28     1
```

Convert the image to a `darray`. 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_darray = darray(single(I),'SSCB');
```

Classify the sample image and find the predicted label.

```
prob = predict(dlnet,I_darray);
[~,label] = max(prob);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' classNames{label}])
```

## Classification result 5



### 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 1-1086.

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 =
    2x1 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: [15x1 nnet.cnn.layer.Layer]
 Connections: [15x2 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 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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

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 “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 1-1091) 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 to 2.6.

## 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: <ul style="list-style-type: none"> <li>• elu</li> <li>• gelu</li> <li>• relu</li> <li>• linear</li> <li>• softmax</li> <li>• sigmoid</li> <li>• swish</li> <li>• tanh</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• geluLayer</li> <li>• reluLayer or clippedReluLayer</li> <li>• None</li> <li>• softmaxLayer</li> <li>• sigmoidLayer</li> <li>• swishLayer</li> <li>• tanhLayer</li> </ul>

<b>TensorFlow-Keras Layer</b>	<b>Corresponding Deep Learning Toolbox Layer</b>
Advanced activations: <ul style="list-style-type: none"> <li>• ELU</li> <li>• Softmax</li> <li>• ReLU</li> <li>• LeakyReLU</li> <li>• PReLU*</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• softmaxLayer</li> <li>• reluLayer, clippedReluLayer, or leakyReluLayer</li> <li>• leakyReluLayer</li> <li>• nnet.keras.layer.PreluLayer</li> </ul>
AveragePooling1D	averagePooling1dLayer with PaddingValue specified as 'mean'
AveragePooling2D	averagePooling2dLayer with PaddingValue specified as 'mean'
BatchNormalization	batchNormalizationLayer
Bidirectional(LSTM(__))	bilstmLayer
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, `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 1-973.

### 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 `dldarray` support.

TensorFlow Operator	Corresponding MATLAB Function
Add	<code>tfAdd</code>
AddN	<code>tfAddN</code>
AddV2	<code>tfAdd</code>
All	<code>tfAll</code>
Assert	<code>assert</code>
AvgPool	<code>tfAvgPool</code>
BatchMatMulV2	<code>tfBatchMatMulV2</code>
BiasAdd	<code>tfBiasAdd</code>
BroadcastTo	<code>tfBroadcastTo</code>
Cast	<code>tfCast</code>
ConcatV2	<code>tfCat</code>
Const	None (translated to weights in custom layer)
Conv2D	<code>tfConv2D</code>
DepthToSpace	<code>depthToSpace</code>
DepthwiseConv2dNative	<code>tfDepthwiseConv2D</code>

TensorFlow Operator	Corresponding MATLAB Function
Equal	==
Exp	exp
ExpandDims	tfExpandDims
Fill	tfFill
FusedBatchNormV3	tfBatchnorm
GatherV2	tfGather
Greater	gt>
GreaterEqual	ge, >=
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, -
NonMaxSuppressionV5	tfNonMaxSuppressionV5
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

TensorFlow Operator	Corresponding MATLAB Function
Relu6	relu and min
Reshape	tfReshape
ResizeBilinear	dlresize
ResizeNearestNeighbor	dlresize
Rsqrt	sqrt
Select	tfSelect
Shape	tfShape
Sigmoid	sigmoid
Size	tfSize
Slice	tfSlice
Softmax	softmax
SpaceToDepth	spaceToDepth
Split	tfSplit
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
Sum	tfSum
Tanh	tanh
Tile	tfTile
Transpose	tfTranspose
Unpack	tfUnpack
Where	tfWhere

For more information on functions that operate on `dlarray` objects, see “List of Functions with `dlarray` Support”.

### Generate Code for Imported Network Architecture

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or

C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).

- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importTensorFlowLayers` returns the network architecture `lgraph` as a `LayerGraph` object. For code generation, you must first convert the imported `LayerGraph` object to a network. Convert a `LayerGraph` object to a `DAGNetwork` or `dlnetwork` object by using `assembleNetwork` or `dlnetwork`. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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.

```
net = dlupdate(@gpuArray,net)
```
- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 1-1089) 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 format to RGB format.
  - 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”.
- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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.

## Version History

### Introduced in R2021a

#### **R2023a: `importTensorFlowLayers` creates a custom layer for concatenation**

*Behavior changed in R2023a*

The `importTensorFlowLayers` function returns an autogenerated custom layer for the TensorFlow-Keras Concatenate layer instead of the built-in Deep Learning Toolbox layer `concatenationLayer` or `depthConcatenationLayer`. Previously, the function used `concatenationLayer` or `depthConcatenationLayer` and assumed a concatenation axis value based on the input layer of the network being imported. However, for some complex networks this assumption might not be true. Therefore, the software uses an autogenerated custom layer to perform the concatenation. The autogenerated custom layer ensures the correct translation of the TensorFlow-Keras Concatenate layer into MATLAB at runtime.

## References

[1] *TensorFlow*. <https://www.tensorflow.org/>.

[2] *Using the SavedModel format*. [https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model).

## See Also

`importTensorFlowNetwork` | `exportNetworkToTensorFlow` | `importNetworkFromPyTorch` | `importKerasLayers` | `importKerasNetwork` | `importONNXLayers` | `importONNXNetwork` | `exportONNXNetwork` | `layerGraph` | `assembleNetwork` | `findPlaceholderLayers` | `replaceLayer`

## Topics

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Tips on Importing Models from TensorFlow, PyTorch, and ONNX”

“Pretrained Deep Neural Networks”

“Assemble Network from Pretrained Keras Layers”

“Define Custom Deep Learning Layers”

“Make Predictions Using `dlnetwork` Object” on page 1-573

# 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 1-1107.

`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 1-1112.

---

`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]
```

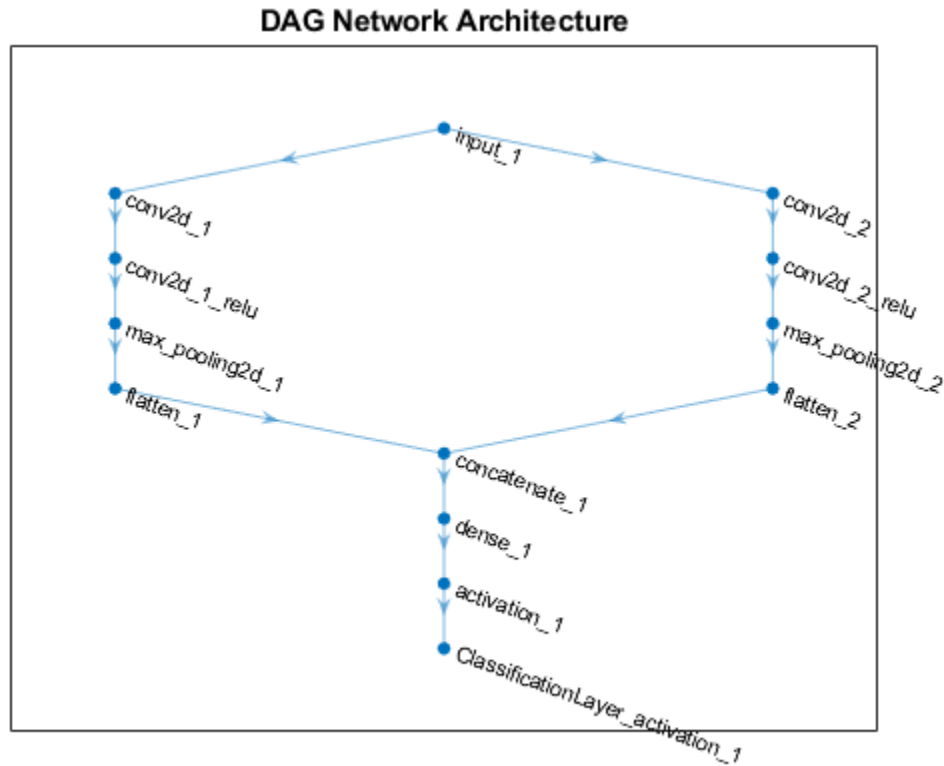
```
 Connections: [13×2 table]
```

```
  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 = 1x2
    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 = 1x3
    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)])
```

Classification result 5



### 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: [12x1 nnet.cnn.layer.Layer]
Connections: [12x2 table]
  Learnables: [6x3 table]
    State: [0x3 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 `darray`. 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_darray = darray(single(I), 'SSCB');
```

Classify the sample image and find the predicted label.

```
prob = predict(net, I_darray);
[~, label] = max(prob);
```

Display the image and the classification result.

```
imshow(I)
title(['Classification result ' classNames{label}])
```

**Classification result 5**



## 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 1-1104.

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 =
```

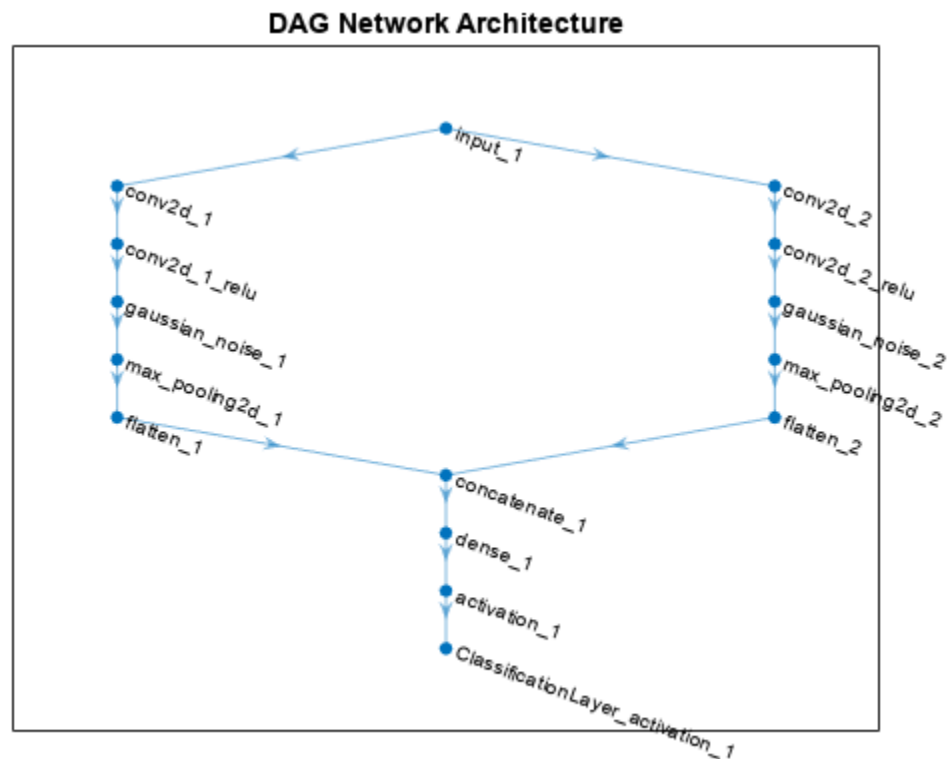
```
3x1 Layer array with layers:
```

1	'concatenate_1'	Concatenate	digitsDAGnetwithnoise.kConcatenate1Layer3826
2	'gaussian_noise_1'	GaussianNoise	digitsDAGnetwithnoise.kGaussianNoise1Layer3766
3	'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'),'ndemos','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)])
```

Classification result 5

**5**

## 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(['.' filesep 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
importTensorFlowNetwork(modelFolder, 'TargetNetwork', 'dagnetwork', 'OutputLayer  
Type', '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 “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 1-1108) 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 to 2.6.

## 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: <ul style="list-style-type: none"> <li>• elu</li> <li>• gelu</li> <li>• relu</li> <li>• linear</li> <li>• softmax</li> <li>• sigmoid</li> <li>• swish</li> <li>• tanh</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• geluLayer</li> <li>• reluLayer or clippedReluLayer</li> <li>• None</li> <li>• softmaxLayer</li> <li>• sigmoidLayer</li> <li>• swishLayer</li> <li>• tanhLayer</li> </ul>
Advanced activations: <ul style="list-style-type: none"> <li>• ELU</li> <li>• Softmax</li> <li>• ReLU</li> <li>• LeakyReLU</li> <li>• PReLU*</li> </ul>	Layers: <ul style="list-style-type: none"> <li>• eluLayer</li> <li>• softmaxLayer</li> <li>• reluLayer, clippedReluLayer, or leakyReluLayer</li> <li>• leakyReluLayer</li> <li>• nnet.keras.layer.PreluLayer</li> </ul>
AveragePooling1D	averagePooling1dLayer with PaddingValue specified as 'mean'
AveragePooling2D	averagePooling2dLayer with PaddingValue specified as 'mean'
BatchNormalization	batchNormalizationLayer
Bidirectional(LSTM(__))	bilstmLayer
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 1-973.

### 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 `darray` support.

TensorFlow Operator	Corresponding MATLAB Function
Add	tfAdd
AddN	tfAddN
AddV2	tfAdd
All	tfAll
Assert	assert
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
Equal	==
Exp	exp
ExpandDims	tfExpandDims
Fill	tfFill
FusedBatchNormV3	tfBatchnorm
GatherV2	tfGather
Greater	gt>
GreaterEqual	ge, >=
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

<b>TensorFlow Operator</b>	<b>Corresponding MATLAB Function</b>
Neg	minus, -
NonMaxSuppressionV5	tfNonMaxSuppressionV5
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
ResizeBilinear	dlresize
ResizeNearestNeighbor	dlresize
Rsqrt	sqrt
Select	tfSelect
Shape	tfShape
Sigmoid	sigmoid
Size	tfSize
Slice	tfSlice
Softmax	softmax
SpaceToDepth	spaceToDepth
Split	tfSplit
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



TensorFlow Operator	Corresponding MATLAB Function
Sum	tfSum
Tanh	tanh
Tile	tfTile
Transpose	tfTranspose
Unpack	tfUnpack
Where	tfWhere

For more information on functions that operate on `dlarray` objects, see “List of Functions with `dlarray` Support”.

### Generate Code for Imported Network

You can use MATLAB Coder or GPU Coder together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for an imported network. For more information, see “Code Generation”.

- Use MATLAB Coder with Deep Learning Toolbox to generate MEX or standalone CPU code that runs on desktop or embedded targets. You can deploy generated standalone code that uses the Intel MKL-DNN library or the ARM Compute library. Alternatively, you can generate generic C or C++ code that does not call third-party library functions. For more information, see “Deep Learning with MATLAB Coder” (MATLAB Coder).
- Use GPU Coder with Deep Learning Toolbox to generate CUDA MEX or standalone CUDA code that runs on desktop or embedded targets. You can deploy generated standalone CUDA code that uses the CUDA deep neural network library (cuDNN), the TensorRT high performance inference library, or the ARM Compute library for Mali GPU. For more information, see “Deep Learning with GPU Coder” (GPU Coder).

`importTensorFlowNetwork` returns the network `net` as a `DAGNetwork` or `dlnetwork` object. Both these objects support code generation. For more information on MATLAB Coder and GPU Coder support for Deep Learning Toolbox objects, see “Supported Classes” (MATLAB Coder) and “Supported Classes” (GPU Coder), respectively.

You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see “Supported Layers” (MATLAB Coder) and “Supported Layers” (GPU Coder), respectively. For more information on the code generation capabilities and limitations of each built-in MATLAB layer, see the Extended Capabilities section of the layer. For example, see “Code Generation” on page 1-940 and “GPU Code Generation” on page 1-941 of `imageInputLayer`.

### 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.

```
net = dlupdate(@gpuArray,net)
```

- 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (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 1-1107) 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 format to RGB format.
  - 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”.
- MATLAB uses one-based indexing, whereas Python uses zero-based indexing. In other words, the first element in an array has an index of 1 and 0 in MATLAB and Python, respectively. For more information on MATLAB indexing, see “Array Indexing”. In MATLAB, to use an array of indices (`ind`) created in Python, convert the array to `ind+1`.
- For more tips, see “Tips on Importing Models from TensorFlow, PyTorch, and ONNX”.

## 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.

## Version History

**Introduced in R2021a**

### **R2023a: importTensorFlowNetwork creates a custom layer for concatenation**

*Behavior changed in R2023a*

The `importTensorFlowNetwork` function returns an autogenerated custom layer for the TensorFlow-Keras Concatenate layer instead of the built-in Deep Learning Toolbox layer `concatenationLayer` or `depthConcatenationLayer`. Previously, the function used `concatenationLayer` or `depthConcatenationLayer` and assumed a concatenation axis value based on the input layer of the network being imported. However, for some complex networks this assumption might not be true. Therefore, the software uses an autogenerated custom layer to perform the concatenation. The autogenerated custom layer ensures the correct translation of the TensorFlow-Keras Concatenate layer into MATLAB at runtime.

## References

[1] *TensorFlow*. <https://www.tensorflow.org/>.

[2] *Using the SavedModel format*. [https://www.tensorflow.org/guide/saved\\_model](https://www.tensorflow.org/guide/saved_model).

## See Also

`importTensorFlowLayers` | `exportNetworkToTensorFlow` | `importNetworkFromPyTorch` | `importKerasNetwork` | `importKerasLayers` | `importONNXNetwork` | `importONNXLayers` | `exportONNXNetwork`

## Topics

“Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX”

“Tips on Importing Models from TensorFlow, PyTorch, and ONNX”

“Inference Comparison Between TensorFlow and Imported Networks for Image Classification”

“Pretrained Deep Neural Networks”

“Make Predictions Using `dlnetwork` Object” on page 1-573

“Deploy Imported TensorFlow Model with MATLAB Compiler”

## 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 =
```

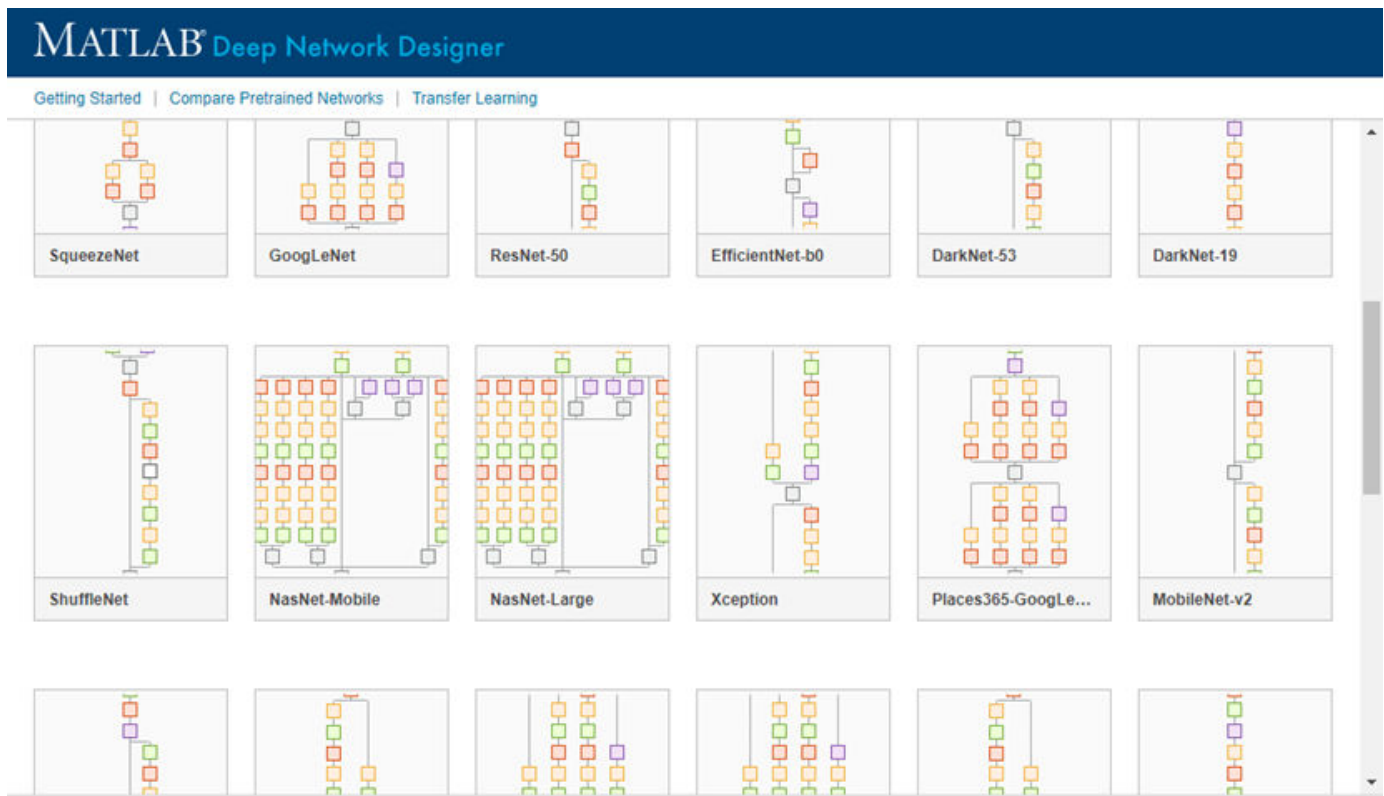
```
    DAGNetwork with properties:
```

```
Layers: [825x1 nnet.cnn.layer.Layer]
Connections: [922x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(inceptionresnetv2)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2017b

## 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”

# 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 =
```

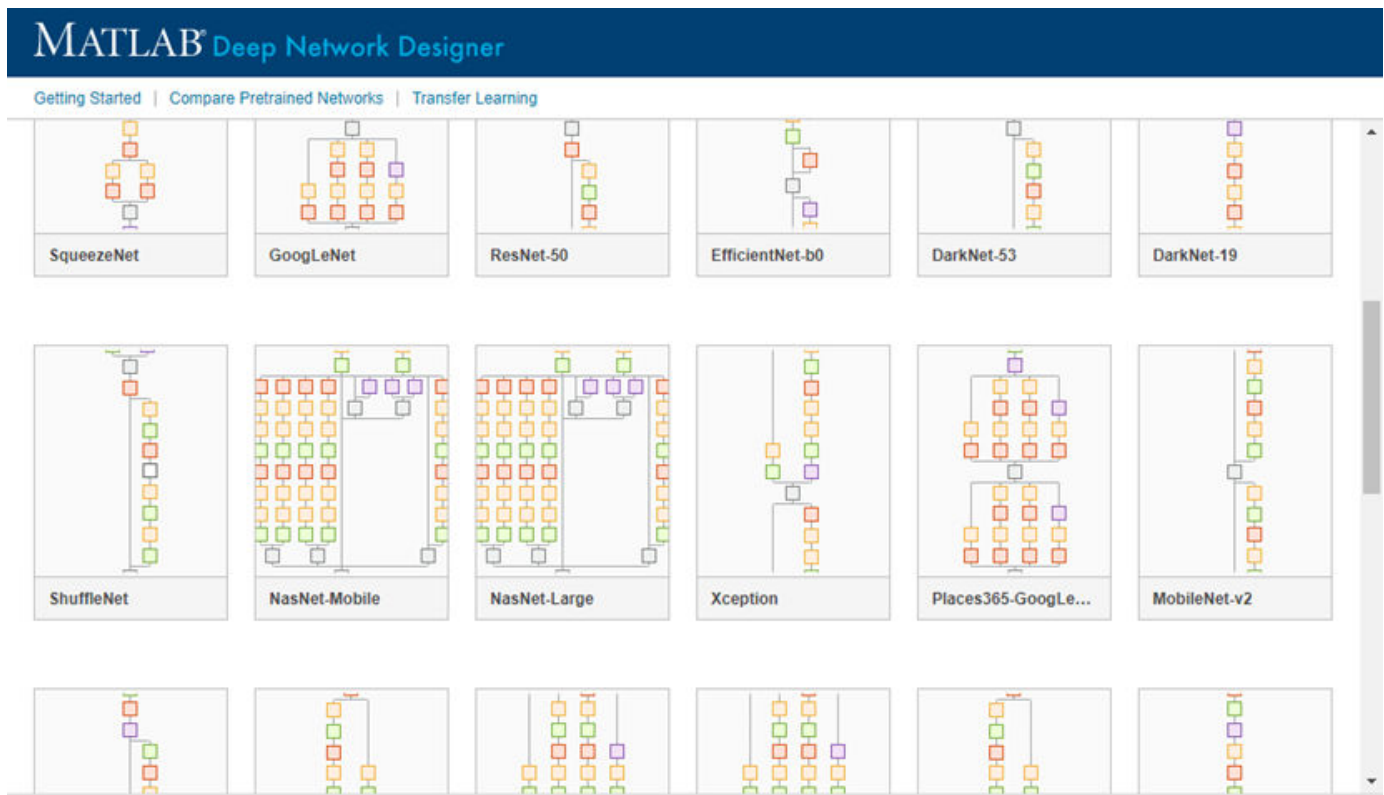
```
DAGNetwork with properties:
```

```
Layers: [316x1 nnet.cnn.layer.Layer]  
Connections: [350x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(inceptionv3)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2017b

## 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”

# initialize

Initialize learnable and state parameters of a `dlnetwork`

## Syntax

```
netUpdated = initialize(net)
netUpdated = initialize(net,X1,...,Xn)
```

## 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.

---

`netUpdated = initialize(net)` initializes any unset learnable parameters and state values of `net` 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.

`netUpdated = initialize(net,X1,...,Xn)` initializes any unset learnable parameters and state values of `net` based on the example network inputs or network data layout objects `X1, . . . , Xn`. Use this syntax when the network has inputs that are not connected to an input layer.

## Examples

### Initialize `dlnetwork` Containing Input Layer

Define a simple image classification network as a layer array.

```
layers = [
    imageInputLayer([28 28 1],Normalization="none")
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer];
```

Convert the layer graph to a `dlnetwork` object. Create an uninitialized `dlnetwork` object by setting the `Initialize` option to `false`.

```
net = dlnetwork(layers,Initialize=false);
```

View the learnable parameters of the network. Because the network is not initialized, the values are empty.

```
net.Learnables
```

```
ans=6x3 table
```

Layer	Parameter	Value
"conv"	"Weights"	{0x0 double}
"conv"	"Bias"	{0x0 double}
"batchnorm"	"Offset"	{0x0 double}
"batchnorm"	"Scale"	{0x0 double}
"fc"	"Weights"	{0x0 double}
"fc"	"Bias"	{0x0 double}

Initialize the learnable parameters of the network using the `initialize` function.

```
net = initialize(net);
```

View the learnable parameters of the network. Because the network is now initialized, the values are nonempty with sizes inferred using the size of the input layer.

```
net.Learnables
```

```
ans=6x3 table
```

Layer	Parameter	Value
"conv"	"Weights"	{ 5x5x1x20 dlarray}
"conv"	"Bias"	{ 1x1x20 dlarray}
"batchnorm"	"Offset"	{ 1x1x20 dlarray}
"batchnorm"	"Scale"	{ 1x1x20 dlarray}
"fc"	"Weights"	{10x11520 dlarray}
"fc"	"Bias"	{10x1 dlarray}

### Initialize dlnetwork Not Containing Input Layer

Define a multi-input image classification network.

```
numFilters = 24;
```

```
layersBranch1 = [
    convolution2dLayer(3,6*numFilters,Padding="same",Stride=2)
    groupNormalizationLayer("all-channels")
    reluLayer
    convolution2dLayer(3,numFilters,Padding="same")
    groupNormalizationLayer("channel-wise")
    additionLayer(2,Name="add")
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer];

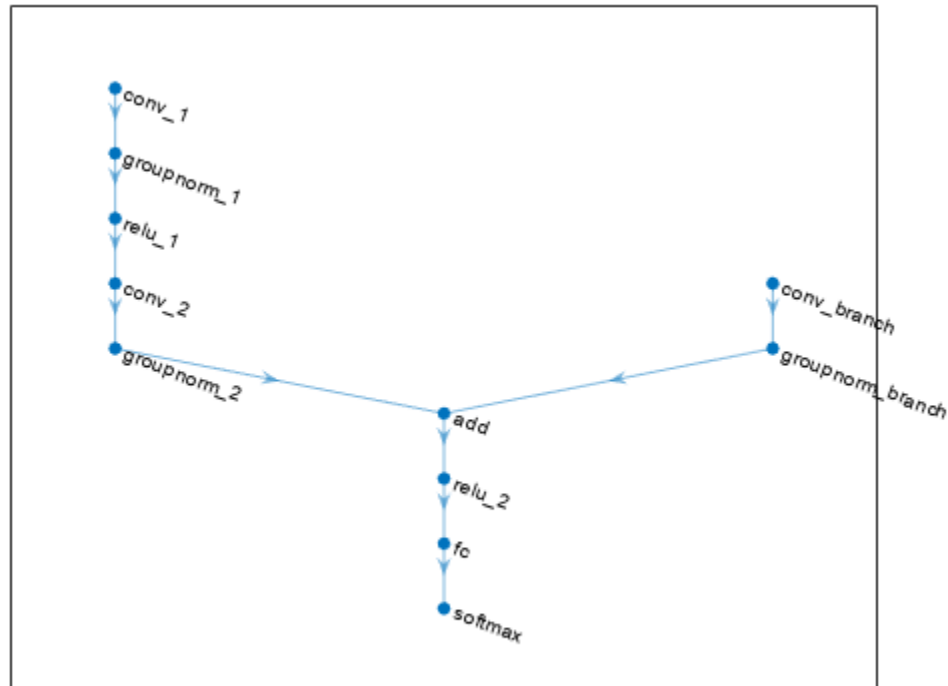
layersBranch2 = [
    convolution2dLayer(1,numFilters,Name="conv_branch")
    groupNormalizationLayer("all-channels",Name="groupnorm_branch")];

lgraph = layerGraph(layersBranch1);
```

```
lgraph = addLayers(lgraph, layersBranch2);
lgraph = connectLayers(lgraph, "groupnorm_branch", "add/in2");
```

Visualize the layer graph in a plot.

```
figure
plot(lgraph)
```



Convert the layer graph to a `dlnetwork` object. Create an uninitialized `dlnetwork` object by setting the `Initialize` option to `false`.

```
net = dlnetwork(lgraph, Initialize=false);
```

View the learnable parameters of the network. Because the network is not initialized, the values are empty.

```
net.Learnables
```

```
ans=14×3 table
```

Layer	Parameter	Value
"conv_1"	"Weights"	{0×0 double}
"conv_1"	"Bias"	{0×0 double}
"groupnorm_1"	"Offset"	{0×0 double}
"groupnorm_1"	"Scale"	{0×0 double}
"conv_2"	"Weights"	{0×0 double}
"conv_2"	"Bias"	{0×0 double}

```

"groupnorm_2"      "Offset"      {0x0 double}
"groupnorm_2"      "Scale"       {0x0 double}
"fc"               "Weights"     {0x0 double}
"fc"               "Bias"        {0x0 double}
"conv_branch"     "Weights"     {0x0 double}
"conv_branch"     "Bias"        {0x0 double}
"groupnorm_branch" "Offset"      {0x0 double}
"groupnorm_branch" "Scale"       {0x0 double}

```

View the names of the network inputs.

```
net.InputNames
```

```
ans = 1x2 cell
    {'conv_1'}    {'conv_branch'}
```

Create random `dlarray` objects representing inputs to the network. Use an example input of size 64-by-64 with 3 channels for the main branch of the network. Use an input of size 64-by-64 with 18 channels for the second branch.

```
inputSize = [64 64 3];
inputSizeBranch = [32 32 18];

X1 = dlarray(rand(inputSize), "SSCB");
X2 = dlarray(rand(inputSizeBranch), "SSCB");
```

Initialize the learnable parameters of the network using the `initialize` function and specify the example inputs. Specify the inputs with order corresponding to the `InputNames` property of the network.

```
net = initialize(net,X1,X2);
```

View the learnable parameters of the network. Because the network is now initialized, the values are nonempty with sizes inferred using the size of the input data.

```
net.Learnables
```

```
ans=14x3 table
    Layer      Parameter      Value
    _____
```

Layer	Parameter	Value
"conv_1"	"Weights"	{ 3x3x3x144 dlarray}
"conv_1"	"Bias"	{ 1x1x144 dlarray}
"groupnorm_1"	"Offset"	{ 1x1x144 dlarray}
"groupnorm_1"	"Scale"	{ 1x1x144 dlarray}
"conv_2"	"Weights"	{ 3x3x144x24 dlarray}
"conv_2"	"Bias"	{ 1x1x24 dlarray}
"groupnorm_2"	"Offset"	{ 1x1x24 dlarray}
"groupnorm_2"	"Scale"	{ 1x1x24 dlarray}
"fc"	"Weights"	{ 1x1x18x24 dlarray}
"fc"	"Bias"	{ 1x1x24 dlarray}
"conv_branch"	"Weights"	{ 1x1x24 dlarray}
"conv_branch"	"Bias"	{ 1x1x24 dlarray}
"groupnorm_branch"	"Offset"	{10x24576 dlarray}
"groupnorm_branch"	"Scale"	{10x1 dlarray}

## Initialize Network using Network Data Layout Objects

Create an uninitialized `dlnetwork` object that has two unconnected inputs.

```
layers = [
    convolution2dLayer(5,16,Name="conv")
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(50)
    flattenLayer
    concatenationLayer(1,2,Name="cat")
    fullyConnectedLayer(10)
    softmaxLayer];

net = dlnetwork(layers,Initialize=false);
```

View the input names of the network.

```
net.InputNames

ans = 1x2 cell
    {'conv'}    {'cat/in2'}
```

Create network data layout objects that represent input data for the inputs. For the first input, specify a batch of 28-by-28 grayscale images. For the second input specify a batch of single-channel feature data.

```
layout1 = networkDataLayout([28 28 1 NaN], "SSCB");
layout2 = networkDataLayout([1 NaN], "CB");
```

Initialize the network using the network data layout objects.

```
net = initialize(net,layout1,layout2)

net =
    dlnetwork with properties:

        Layers: [8x1 nnet.cnn.layer.Layer]
    Connections: [7x2 table]
    Learnables: [8x3 table]
        State: [2x3 table]
    InputNames: {'conv' 'cat/in2'}
    OutputNames: {'softmax'}
    Initialized: 1
```

View summary with `summary`.

## Input Arguments

### **net** — Uninitialized network

`dlnetwork` object

Uninitialized network, specified as a `dlnetwork` object.

**X1, . . . ,Xn — Example network inputs or data layouts**

formatted `dlarray` object | formatted `networkDataLayout` object

Example network inputs or data layouts, specified as formatted `dlarray` objects or formatted `networkDataLayout` objects. The software propagates  $X_1, \dots, X_n$  through the network to determine the appropriate sizes and formats of the learnable and state parameters of the `dlnetwork`.

Provide example inputs in the same order as the order specified by the `InputNames` property of the input network.

---

**Note** Automatic initialization uses only the size and format information of the input data. For initialization that depends on the values on the input data, you must initialize the learnable parameters manually.

---

## Output Arguments

**netUpdated — Initialized network**

`dlnetwork` object

Initialized network, returned as an initialized `dlnetwork` object.

The `initialize` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

**Introduced in R2021a**

### See Also

`dlarray` | `dlnetwork` | `summary` | `networkDataLayout`

### 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”



# instancenorm

Normalize across each channel for each observation independently

## Syntax

```
Y = instancenorm(X,offset,scaleFactor)
Y = instancenorm(X,offset,scaleFactor,'DataFormat',FMT)
Y = 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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .

The `instancenorm` function applies the layer normalization operation to `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** To apply instance normalization within a `layerGraph` object or `Layer` array, use `instanceNormalizationLayer`.

---

`Y = instancenorm(X,offset,scaleFactor)` applies the instance normalization operation to the input data `X` 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 `X` for each observation in the 'C' (channel) and 'B' (batch) dimensions, independently.

For unformatted input data, use the 'DataFormat' option.

`Y = instancenorm(X,offset,scaleFactor,'DataFormat',FMT)` applies the instance normalization operation to the unformatted `darray` object `X` with format specified by `FMT` using any of the previous syntaxes. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. For example, 'DataFormat', 'SSCB' specifies data for 2-D image input with format 'SSCB' (spatial, spatial, channel, batch).

`Y = 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

### X — Input data

dLarray | numeric array

Input data, specified as a formatted dLarray, an unformatted dLarray, or a numeric array.

If X is an unformatted dLarray or a numeric array, then you must specify the format using the `DataFormat` option. If X is a numeric array, then either `scaleFactor` or `offset` must be a dLarray object.

X must have a 'C' (channel) dimension.

### offset — Offset

dLarray | numeric array

Offset  $\beta$ , 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 X.

If `offset` is a formatted `darray` object, then the nonsingleton dimension must have label 'C' (channel).

### **scaleFactor — Scale factor**

`darray` | numeric array

Scale factor  $\gamma$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input  $X$ .

If `scaleFactor` is a formatted `darray` object, then the nonsingleton dimension must have label 'C' (channel).

### **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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

### **Epsilon — Constant to add to mini-batch variances**

1e-5 (default) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`Epsilon` must be greater than or equal to 1e-5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **Y — Normalized data**

`darray`

Normalized data, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, Y has the same dimension format as X. If the input data is not a formatted `darray`, Y is an unformatted `darray` 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  $\mu_I$  and variance  $\sigma_I^2$  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  $\epsilon$  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  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

## Version History

**Introduced in R2021a**

**R2023a: Epsilon supports values less than 1e-5**

The `Epsilon` option also supports positive values less than 1e-5.

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:

- `X`
- `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”

# 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  $\gamma$  and shifts it by a learnable offset  $\beta$ .

## 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 1-1133, “Learning Rate and Regularization” on page 1-1135, 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) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

`Epsilon` must be greater than or equal to 1e-5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**NumChannels — Number of input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. NumChannels and the number of channels in the layer input data must match. For example, if the input is an RGB image, then NumChannels must be 3. If the input is the output of a convolutional layer with 16 filters, then NumChannels must be 16.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | string

**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  $\gamma$ , specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Scale` is nonempty, then the software uses the `Scale` property as the initial value. If `Scale` is empty, then the software uses the initializer specified by `ScaleInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

Data Types: `single` | `double`

**Offset – Channel offsets**

`[]` (default) | numeric array

Channel offsets  $\beta$ , specified as a numeric vector.

The channel offsets are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Offset` is nonempty, then the software uses the `Offset` property as the initial value. If `Offset` is empty, then the software uses the initializer specified by `OffsetInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ScaleL2Factor — L<sub>2</sub> regularization factor for scale factors

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### OffsetL2Factor — L<sub>2</sub> regularization factor for offsets

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> 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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0]
3	''	Instance Normalization	Instance normalization
4	''	ReLU	ReLU
5	''	2-D 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

## Algorithms

### Instance Normalization Layer

The instance normalization operation normalizes the elements  $x_i$  of the input by first calculating the mean  $\mu_I$  and variance  $\sigma_I^2$  over the spatial and time dimensions for each channel in each observation independently. Then, it calculates the normalized activations as

$$\hat{x}_i = \frac{x_i - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}},$$

where  $\epsilon$  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  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

## Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `InstanceNormalizationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `InstanceNormalizationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2021a

### R2023a: Epsilon supports values less than 1e-5

The Epsilon option also supports positive values less than 1e-5.

### R2023a: Layer supports 1-D image sequence data

InstanceNormalizationLayer objects support normalizing 1-D image sequence data (data with one spatial and one time dimension).

## 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"

## isdlarray

Check if object 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

#### Check if Object is a dlarray

Create a 2-by-3-by-4 array of random values.

```
X = rand(2,3,4)
```

```
X =
```

```
X(:,:,1) =
```

```
    0.8147    0.1270    0.6324  
    0.9058    0.9134    0.0975
```

```
X(:,:,2) =
```

```
    0.2785    0.9575    0.1576  
    0.5469    0.9649    0.9706
```

```
X(:,:,3) =
```

```
    0.9572    0.8003    0.4218  
    0.4854    0.1419    0.9157
```

```
X(:,:,4) =
```

```
    0.7922    0.6557    0.8491  
    0.9595    0.0357    0.9340
```

Check if the array is a `dlarray` object.

```
isdlarray(X)
```

```
ans = logical  
     0
```

Convert the array to `dllarray`.

```
X = dllarray(X)
```

```
X =
    2x3x4 dllarray
```

```
(:,: ,1) =
```

```
    0.8147    0.1270    0.6324
    0.9058    0.9134    0.0975
```

```
(:,: ,2) =
```

```
    0.2785    0.9575    0.1576
    0.5469    0.9649    0.9706
```

```
(:,: ,3) =
```

```
    0.9572    0.8003    0.4218
    0.4854    0.1419    0.9157
```

```
(:,: ,4) =
```

```
    0.7922    0.6557    0.8491
    0.9595    0.0357    0.9340
```

Check if the array is a `dllarray` object.

```
isdllarray(X)
```

```
ans = logical
     1
```

## Input Arguments

### **X** — Input variable

workspace variable

Input variable, specified as a workspace variable. X can be any data type.

## Version History

**Introduced in R2020b**

## 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”



# 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).

The `isequal` function can operate on arrays of networks and layer graphs. In this case, the function performs element-wise comparison.

`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

### **net1, net2** — 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.

## Version History

**Introduced in R2021a**

### See Also

isequaln | trainNetwork | SeriesNetwork | DAGNetwork | analyzeNetwork | assembleNetwork | dlnetwork

### Topics

“Create Simple Deep Learning Neural 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”

# 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).

The `isequaln` function can operate on arrays of networks and layer graphs. In this case, the function performs element-wise comparison.

`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

### **net1, net2** — 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.

## Version History

**Introduced in R2021a**

### See Also

`isequal` | `trainNetwork` | `SeriesNetwork` | `DAGNetwork` | `analyzeNetwork` | `assembleNetwork` | `dlnetwork`

### Topics

“Create Simple Deep Learning Neural 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”

# isInNetworkDistribution

Determine whether data is within the distribution of the network

## Syntax

```
tf = isInNetworkDistribution(net,X)
tf = isInNetworkDistribution(net,X,Threshold=thr)
tf = isInNetworkDistribution(discriminator,X)
```

## Description

`tf = isInNetworkDistribution(net,X)` returns a logical array that indicates which observations in `X` are in-distribution (ID) and which observations are out-of-distribution (OOD). If an observation is ID, then the corresponding element of `tf` is 1 (true). Otherwise, the corresponding element of `tf` is 0 (false).

The function computes the distribution confidence score for each observation using the baseline method. For more information, see “Softmax-Based Methods” on page 1-1151. The function classifies any observation with a score less than or equal to the threshold as OOD. To use the default threshold value, use this syntax.

To set the threshold, use the `thr` name-value argument. Alternatively, use the `networkDistributionDiscriminator` function to create a discriminator object that automatically finds an optimal threshold and use that as the first input argument instead of `net`. You can also use the discriminator object to specify a different method to use to compute the distribution confidence scores.

`tf = isInNetworkDistribution(net,X,Threshold=thr)` also specifies a nondefault threshold value.

`tf = isInNetworkDistribution(discriminator,X)` determines which observations in `X` are ID and which observations are OOD using `discriminator`. To create a discriminator object, use the `networkDistributionDiscriminator` function. This syntax uses the threshold stored in the `Threshold` property of `discriminator`. Use this syntax to specify additional options for the software to use when it computes the distribution confidence scores and to automatically find a suitable threshold. For example, when creating a discriminator, you can specify whether to use a target true positive or false positive rate to pick the threshold. For more information, see `networkDistributionDiscriminator`.

## Examples

### Check if Data is In-Distribution

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat")
```

Load data. Convert the data to a `dlarray` object.

```
X = digitTrain4DArrayData;  
X = dlarray(X, "SSCB");
```

Determine if the data is ID.

```
tf = isInNetworkDistribution(net,X);
```

Find the proportion of observations that the function classifies as OOD.

```
oodProportion = (sum(1-tf)/numel(tf))
```

```
oodProportion = 0.0026
```

### Specify Threshold

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat")
```

Load data and convert the data to a `dlarray` object.

```
X = digitTrain4DArrayData;  
X = dlarray(X, "SSCB");
```

Determine if the data is ID using a threshold of 0.9.

```
tf = isInNetworkDistribution(net,X,Threshold=0.9);
```

Find the proportion of observations that the function classifies as OOD.

```
oodProportion = (sum(1-tf)/numel(tf))
```

```
oodProportion = 0
```

### Specify Additional Options Using Discriminator

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat")
```

Load ID data. Convert the data to a `dlarray` object.

```
X = digitTrain4DArrayData;  
X = dlarray(X, "SSCB");
```

Create a discriminator using the `networkDistributionDiscriminator` function. Set the method to "odin" and the true positive goal to 0.975. The software finds the threshold that satisfies the true positive goal.

```
method = "odin";  
discriminator = networkDistributionDiscriminator(net,X,[],method, ...  
    TruePositiveGoal=0.975);
```

Determine if data is ID.

```
tf = isInNetworkDistribution(discriminator,X);
```

Find the true positive rate.

```
truePositives = sum(tf);
falseNegatives = sum(1-tf);
truePositiveRate = truePositives/(truePositives + falseNegatives)

truePositiveRate = 0.9750
```

## Input Arguments

### **net** — Neural network

`dlnetwork` object

Neural network, specified as a `dlnetwork` object with a single softmax output layer.

The software uses the baseline method to compute the distribution confidence scores. To use another method, such as ODIN or energy, specify `discriminator` as the first input argument. For more information about methods for computing distribution confidence scores, see “Distribution Confidence Scores” on page 1-1150.

For networks without a single softmax layer, create a discriminator object using the `networkDistributionDiscriminator` function with `method` set to “`hbos`” and use this object as the first input argument instead.

### **discriminator** — Distribution discriminator

`BaselineDistributionDiscriminator` object | `ODINDistributionDiscriminator` object | `EnergyDistributionDiscriminator` object | `HBOSDistributionDiscriminator` object

Distribution discriminator, specified as a `BaselineDistributionDiscriminator`, `ODINDistributionDiscriminator`, `EnergyDistributionDiscriminator`, or `HBOSDistributionDiscriminator` object. To create this object, use the `networkDistributionDiscriminator` function.

### **X** — Input data

formatted `dlarray`

Input data, specified as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

### **thr** — Distribution threshold

0.95 (default) | scalar in the range [0, 1]

Distribution threshold, specified as a scalar in the range [0, 1]. The software uses this value to separate the ID and OOD data.

### Dependency

You can only specify this input when the first argument is `net`. If the first argument is `discriminator`, then the software uses the threshold stored in the `Threshold` property of `discriminator`. For more information, see `networkDistributionDiscriminator`.

## More About

### In-Distribution and Out-of-Distribution Data

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-1150.

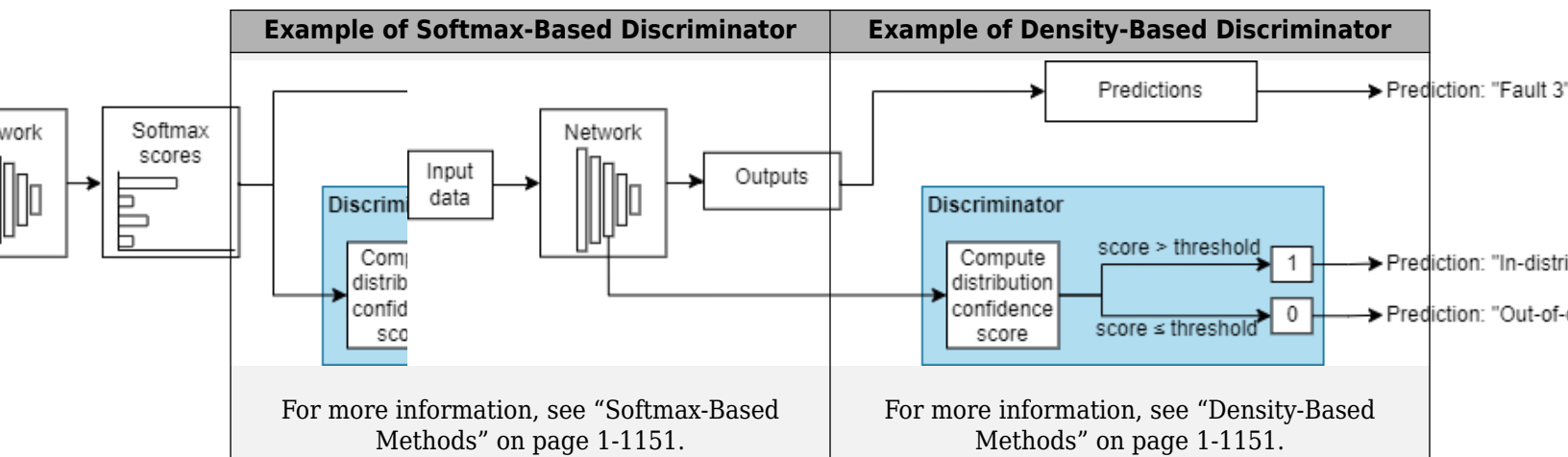
### OOD Data Detection

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-1150.

These images show how a discriminator acts as an additional output of a trained neural network.

### Example Data Discriminators



### Distribution Confidence Scores

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.



### Softmax-Based Methods

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}},$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is `-Inf`.

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. “A Baseline for Detecting Out-of-Distribution Examples in Image Captioning.” In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.

- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## 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 its inputs meet either or both of these conditions:
  - Any values of the network learnable parameters are `darray` objects with underlying data of type `gpuArray`. To see the learnables for a network `net`, call `net.Learnables.Value`.
  - Any of the data is a `darray` object with underlying data of type `gpuArray`.

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

### See Also

`networkDistributionDiscriminator` | `distributionScores`

### Topics

"Out-of-Distribution Detection for Deep Neural Networks"

"Out-of-Distribution Data Discriminator for YOLO v4 Object Detector"

"Verify Robustness of Deep Learning Neural Network"

## l1loss

$L_1$  loss for regression tasks

### Syntax

```
loss = l1loss(Y,targets)
loss = l1loss(Y,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

`loss = l1loss(Y,targets)` computes the MAE loss for the predictions `Y` and the target values `targets`. The input `Y` must be a formatted `darray`. The output `loss` is an unformatted `darray` scalar.

`loss = l1loss(Y,targets,weights)` computes the weighted  $L_1$  loss using the weight values `weights`. The output `loss` is an unformatted `darray` scalar.

`loss = l1loss( ____,DataFormat=FMT)` computes the loss for the unformatted `darray` object `Y` 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(Y,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 = darray(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 =
    1×1 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=12x1 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=12x1 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 `darray` with the format "CTB" (channel, time, batch). Because formatted `darray` objects automatically permute the dimensions of the underlying data, keep the order consistent by also converting the targets and mask to formatted `darray` objects with the format "CTB" (channel, batch, time).

```
dLY = darray(Y, "CTB");
targets = darray(targets, "CTB");
mask = darray(mask, "CTB");
```

View the sizes of the prediction scores, targets, and mask.

```
size(dLY)
```

```
ans = 1x3
```

```
    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 =
    1×1 darray
    32.6172
```

## Input Arguments

### Y — Predictions

`darray` | numeric array

Predictions, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. When `Y` is not a formatted `darray`, you must specify the dimension format using the `DataFormat` option.

If `Y` is a numeric array, `targets` must be a `darray`.

### targets — Target responses

`darray` | numeric array

Target responses, specified as a formatted or unformatted `darray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `Y`.

If `targets` is a formatted `darray`, then its format must be the same as the format of `Y`, or the same as `DataFormat` if `Y` is unformatted.

If `targets` is an unformatted `darray` or a numeric array, then the function applies the format of `Y` or the value of `DataFormat` to `targets`.

---

**Tip** Formatted `darray` 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 `Y` and `targets` are consistent, when `Y` is a formatted `darray`, also specify `targets` as a formatted `darray`.

---

### weights — Weights

`darray` | numeric array

Weights, specified as a formatted or unformatted `darray` or a numeric array.

If `weights` is a vector and `Y` has two or more nonsingleton dimensions, then `weights` must be a formatted `darray`, 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 `Y`.

If `weights` is a formatted `darray` with two or more nonsingleton dimensions, then its format must match the format of `Y`.

If `weights` is not a formatted `darray` and has two or more nonsingleton dimensions, then its size must match the size of `Y` and the function uses the same format as `Y`. 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 = llloss(Y, 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**

`darray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `darray` object, a logical array, or a numeric array with the same size as `Y`.

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.

If `Mask` is a formatted `darray` object, then its format must match that of `Y`. If `Mask` is not a formatted `darray` object, then the function uses the same format as `Y`.

If you specify the `DataFormat` option, then the function also uses the specified format for the mask.

The size of each dimension of `Mask` must match the size of the corresponding dimension in `Y`. The default value is a logical array of ones.

---

**Tip** Formatted `darray` 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, `darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

### **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 `darray` object with the same size as `Y`.

### **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 `Y`.
- "all-elements" — Normalize the loss by dividing it by the number of elements of `Y`.
- "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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `Y` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `Y` are vectors, then default value of `WeightsFormat` is the same as the format of `Y`.

Data Types: `char` | `string`

## Output Arguments

### loss — L<sub>1</sub> loss

`dlarray`

L<sub>1</sub> loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `Y`.

The size of `loss` depends on the `Reduction` option.

## Algorithms

### L1 Loss

The L<sub>1</sub> loss operation computes the L<sub>1</sub> 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

$$\text{loss}_j = |Y_j - T_j|,$$

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:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

## Version History

**Introduced in R2021b**

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `Y`
  - `targets`
  - `weights`
  - `Mask`

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

### See Also

`darray` | `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 `darray` Support”

## l2loss

$L_2$  loss for regression tasks

### Syntax

```
loss = l2loss(Y,targets)
loss = l2loss(Y,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

`loss = l2loss(Y,targets)` computes MSE loss for the predictions `Y` and the target values `targets`. The input `Y` must be a formatted `darray`. The output `loss` is an unformatted `darray` scalar.

`loss = l2loss(Y,targets,weights)` computes the weighted  $L_2$  loss using the weight values `weights`. The output `loss` is an unformatted `darray` scalar.

`loss = l2loss( ____,DataFormat=FMT)` computes the loss for the unformatted `darray` object `Y` 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(Y,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 = darray(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 =
    1×1 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=12x1 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=12x1 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 `darray` with the format "CTB" (channel, time, batch). Because formatted `darray` objects automatically permute the dimensions of the underlying data, keep the order consistent by also converting the targets and mask to formatted `darray` objects with the format "CTB" (channel, batch, time).

```
dLY = darray(Y, "CTB");
targets = darray(targets, "CTB");
mask = darray(mask, "CTB");
```

View the sizes of the prediction scores, targets, and mask.

```
size(dLY)
```

```
ans = 1x3
```

```
    10    12    15
```

```
size(targets)
```

```
ans = 1x3
```

```

    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 =
    1×1 dlarray

    16.3668

```

## Input Arguments

### Y — Predictions

dlarray | numeric array

Predictions, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array. When `Y` is not a formatted `dlarray`, you must specify the dimension format using the `DataFormat` option.

If `Y` 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 `Y`.

If `targets` is a formatted `dlarray`, then its format must be the same as the format of `Y`, or the same as `DataFormat` if `Y` is unformatted.

If `targets` is an unformatted `dlarray` or a numeric array, then the function applies the format of `Y` 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 `Y` and `targets` are consistent, when `Y` 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 `Y` has two or more nonsingleton dimensions, then `weights` must be a formatted `darray`, 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 `Y`.

If `weights` is a formatted `darray` with two or more nonsingleton dimensions, then its format must match the format of `Y`.

If `weights` is not a formatted `darray` and has two or more nonsingleton dimensions, then its size must match the size of `Y` and the function uses the same format as `Y`. 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(Y, 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**

`darray` | logical array | numeric array

Mask indicating which elements to include for loss computation, specified as a `darray` object, a logical array, or a numeric array with the same size as `Y`.

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.

If `Mask` is a formatted `darray` object, then its format must match that of `Y`. If `Mask` is not a formatted `darray` object, then the function uses the same format as `Y`.

If you specify the `DataFormat` option, then the function also uses the specified format for the mask.

The size of each dimension of `Mask` must match the size of the corresponding dimension in `Y`. The default value is a logical array of ones.

---

**Tip** Formatted `darray` 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, `darray` objects automatically permute the dimensions of data with format "TSCSBS" to have format "SSSCBT".

To ensure that the dimensions of `Y` and the mask are consistent, when `Y` is a formatted `darray`, also specify the mask as a formatted `darray`.

---

### **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 `darray` object with the same size as `Y`.



**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 `Y`.
- "all-elements" — Normalize the loss by dividing it by the number of elements of `Y`.
- "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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `Y` has two or more nonsingleton dimensions.

If `weights` is not a vector, or both `weights` and `Y` are vectors, then default value of `WeightsFormat` is the same as the format of `Y`.

Data Types: `char` | `string`

## Output Arguments

### loss — L<sub>2</sub> loss

`dlarray`

L<sub>2</sub> loss, returned as an unformatted `dlarray`. The output `loss` is an unformatted `dlarray` with the same underlying data type as the input `Y`.

The size of `loss` depends on the `Reduction` option.

## Algorithms

### L2 Loss

The L<sub>2</sub> loss operation computes the L<sub>2</sub> loss (based on the squared L<sub>2</sub> 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

$$\text{loss}_j = (Y_j - T_j)^2,$$

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:

$$\text{loss}_j^* = m_j w_j \text{loss}_j$$

## Version History

Introduced in R2021b

## 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 `darray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `Y`
  - `targets`
  - `weights`
  - `Mask`

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

### See Also

`darray` | `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 `darray` Support”

## 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	''	2-D Convolution	10 5x5 convolutions with stride [1 1] and padding [0 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	''	2-D Convolution	10 5x5 convolutions with stride [1 1] and padding [0 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]
        SplitComplexInputs: 0

    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)

```

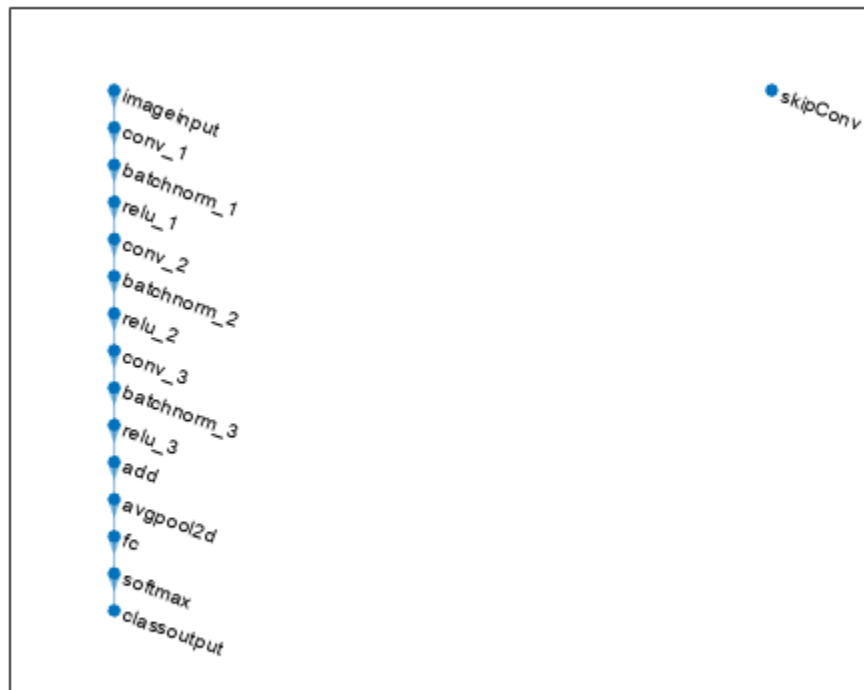


Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the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

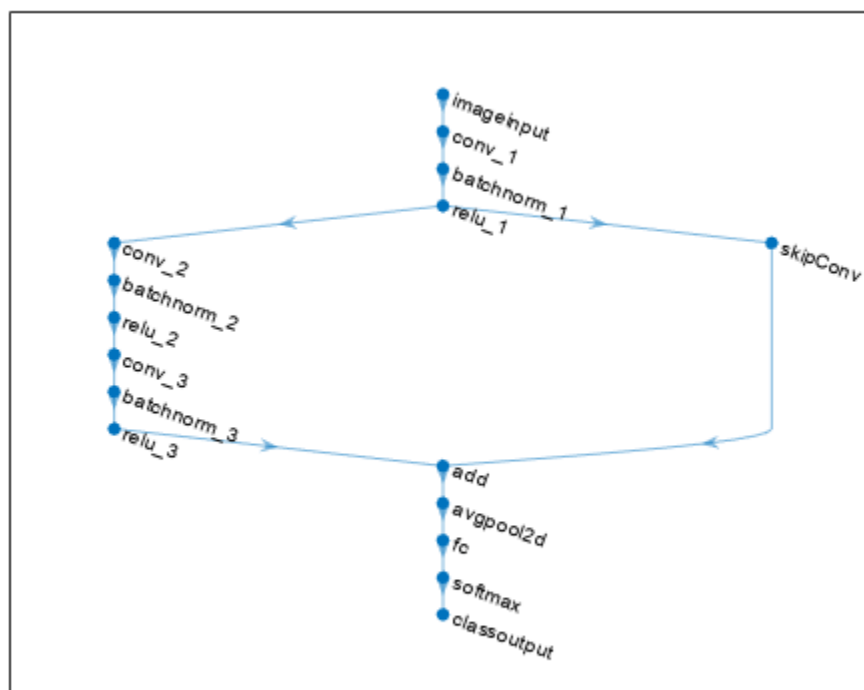
```



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);
```



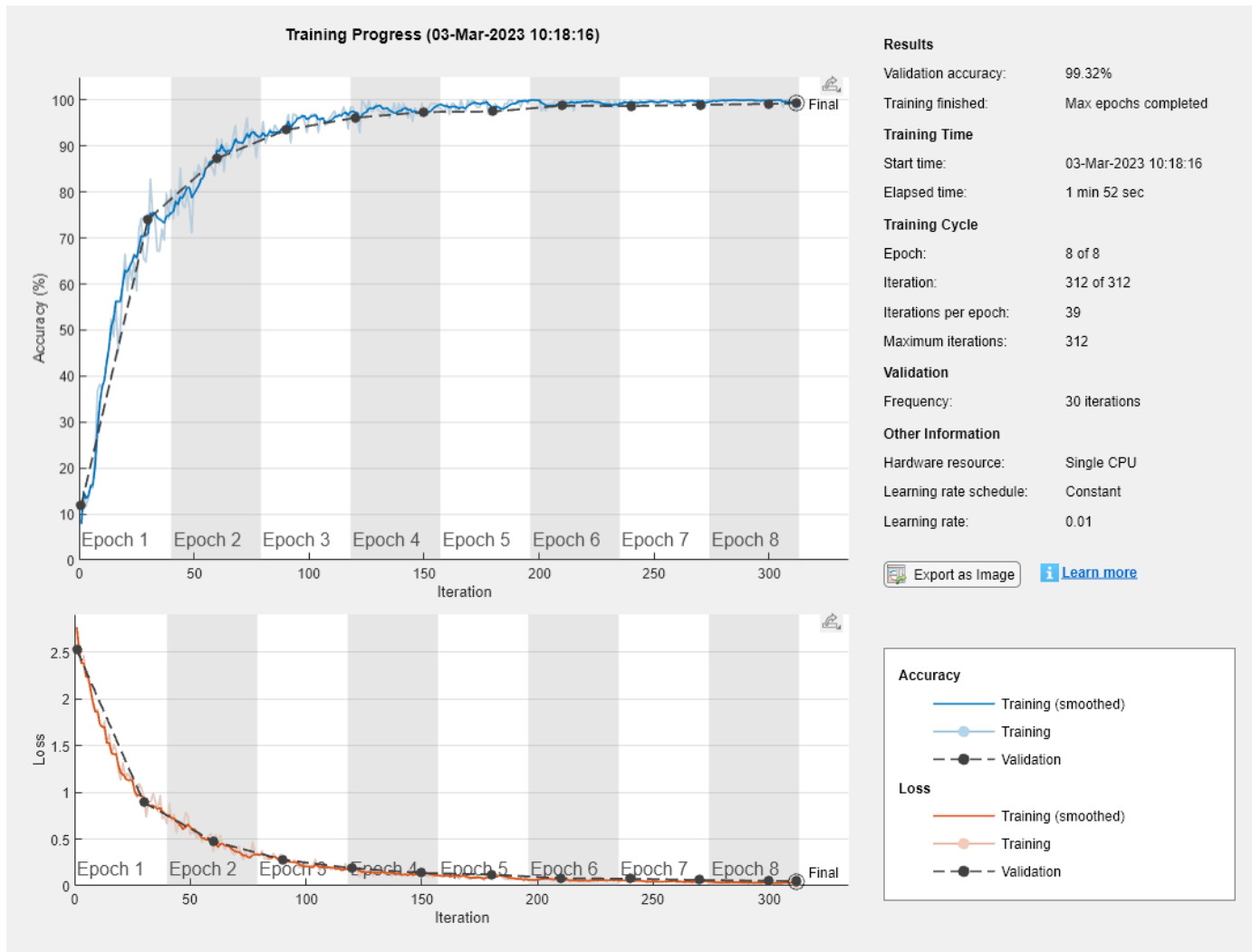


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[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 = 0.9932
```

## Version History

Introduced in R2016a

### See Also

[importCaffeLayers](#) | [trainNetwork](#) | [LayerGraph](#) | [importKerasLayers](#) | [assembleNetwork](#)

### Topics

["Create Simple Deep Learning Neural 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"](#)

# 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 or convert it to a `dlnetwork` and train it using a custom training loop.

## Creation

### Syntax

```
lgraph = layerGraph
lgraph = layerGraph(layers)
lgraph = layerGraph(net)
```

### 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`, `DAGNetwork`, or `dlnetwork` object. For example, you can extract the layer graph of a pretrained network to perform transfer learning.

### Input Arguments

#### **net** — Deep learning network

`SeriesNetwork` object | `DAGNetwork` object | `dlnetwork` object

Deep learning network, specified as a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object.

## 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**

<code>addLayers</code>	Add layers to layer graph or network
<code>removeLayers</code>	Remove layers from layer graph or network
<code>replaceLayer</code>	Replace layer in layer graph or network
<code>connectLayers</code>	Connect layers in layer graph or network
<code>disconnectLayers</code>	Disconnect layers in layer graph or network
<code>plot</code>	Plot neural network architecture

**Examples****Add Layers to Layer Graph**

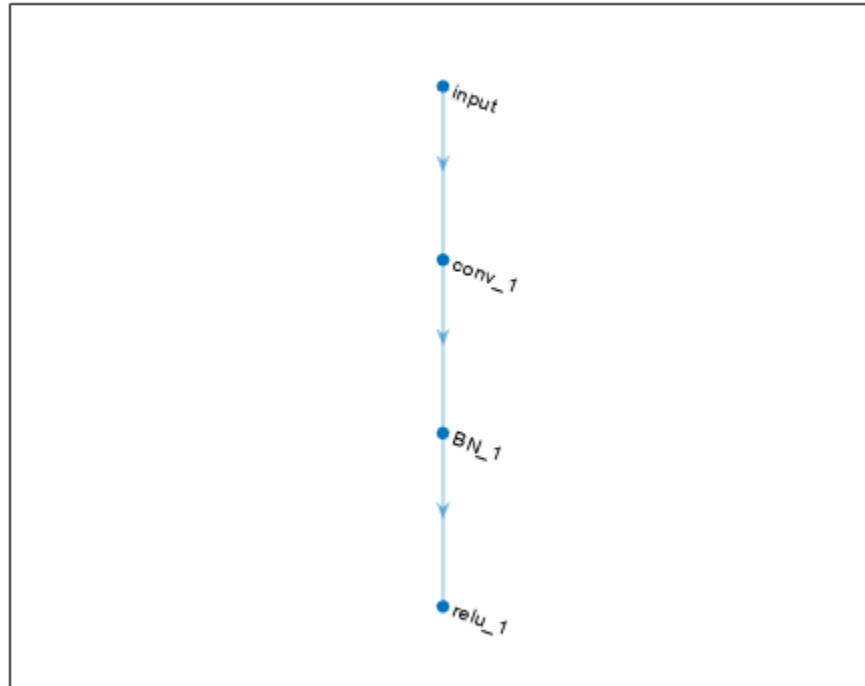
Create an empty layer graph and an array of layers. Add the layers to the layer graph and plot the graph. `addLayers` connects the layers sequentially.

```
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)
```



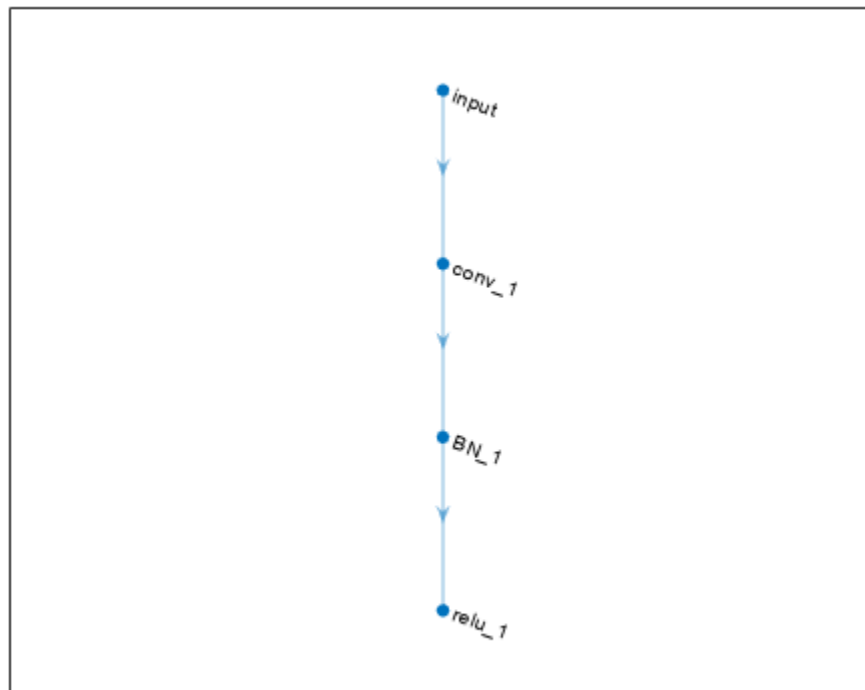
### Create Layer Graph from an Array of Layers

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:  
    InputNames: {'data'}  
    OutputNames: {'ClassificationLayer_predictions'}  
    Layers: [68x1 nnet.cnn.layer.Layer]  
    Connections: [75x2 table]
```

### 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.

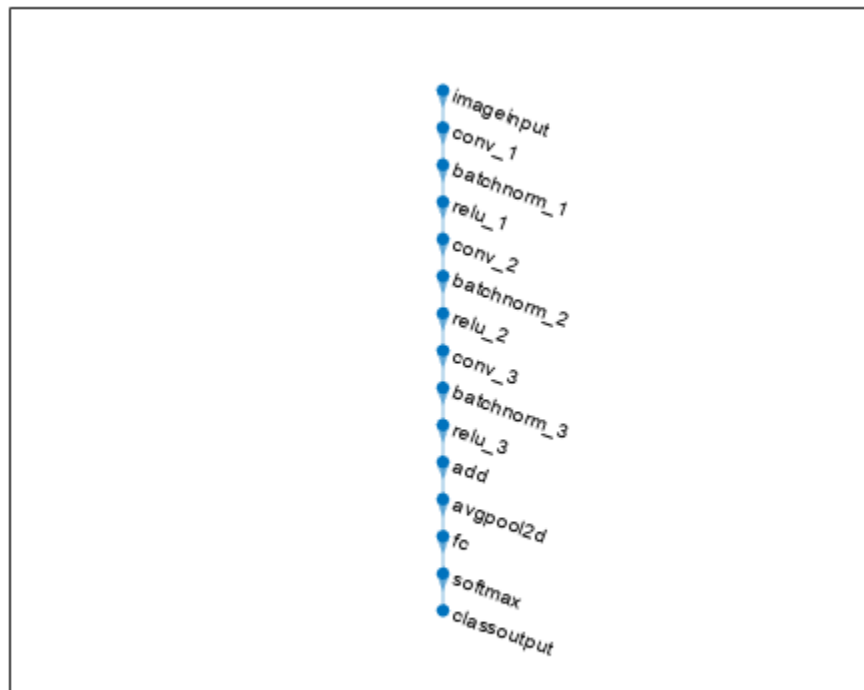
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. To easily add connections later, specify names for the first ReLU layer and the addition layer.

```
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.

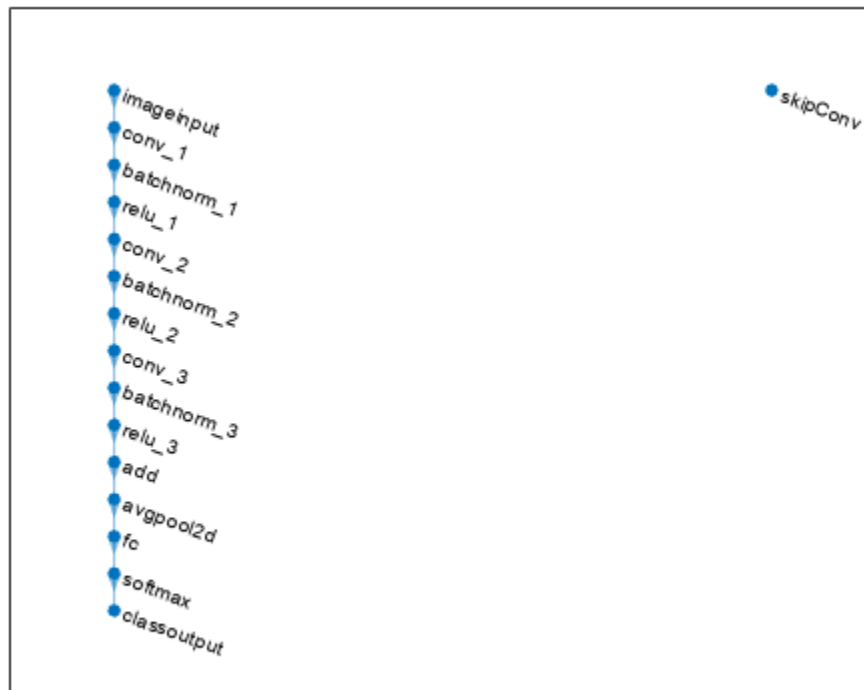
```
lgraph = layerGraph(layers);  
figure  
plot(lgraph)
```





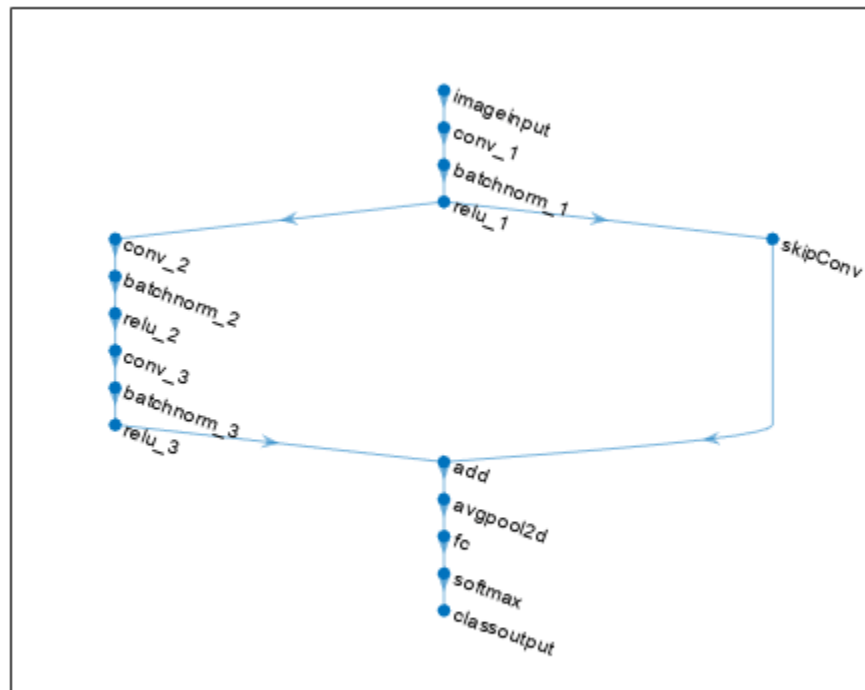
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the third ReLU layer. This arrangement enables the addition layer to add the outputs of the third ReLU layer and the 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);
```

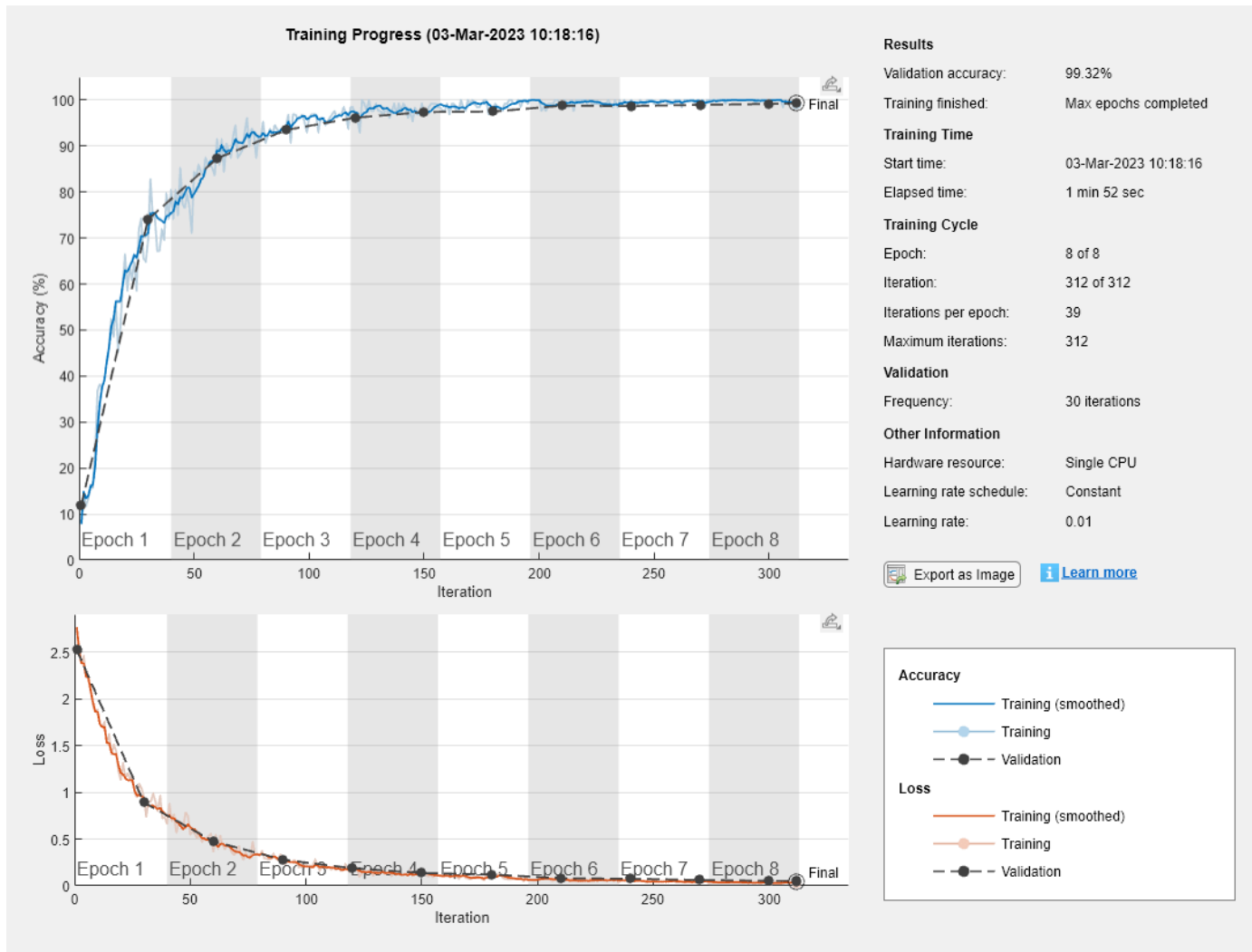


Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```
[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 = 0.9932
```

## Limitations

- Layer graph objects contain no quantization information. Extracting the layer graph from a quantized network and then reassembling the network using `assembleNetwork` or `dlnetwork` removes quantization information from the network.

## Version History

Introduced in R2017b

### 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 Neural 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”

## layernorm

Normalize data across all channels for each observation independently

### Syntax

```
Y = layernorm(X,offset,scaleFactor)
Y = layernorm(X,offset,scaleFactor,'DataFormat',FMT)
Y = 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  $\beta$  and scales it by a learnable scale factor  $\gamma$ .

The `layernorm` function applies the layer normalization operation to `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** To apply layer normalization within a `layerGraph` object or `Layer` array, use `layerNormalizationLayer`.

---

`Y = layernorm(X,offset,scaleFactor)` applies the layer normalization operation to the input data `X` 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 `X` for each observation in the 'B' (batch) dimension, independently.

For unformatted input data, use the 'DataFormat' option.

`Y = layernorm(X,offset,scaleFactor,'DataFormat',FMT)` applies the layer normalization operation to the unformatted `darray` object `X` with the format specified by `FMT`. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. 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.

`Y = 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 a single input observation.

Create a formatted `darray` object containing a batch of 128 224-by-224 images with 3 channels. Specify the format "SSCB" (spatial, spatial, channel, batch).

```
numChannels = 3;
miniBatchSize = 128;
H = 224;
W = 224;
X = rand(H,W,numChannels,miniBatchSize);
X = darray(X, "SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
      224   224     3   128
```

```
dims(X)
```

```
ans =
'SSCB'
```

For element-wise layer normalization, initialize the offset and scale with an array of zeros and ones, respectively.

```
offset = zeros(H,W,numChannels);
scaleFactor = ones(H,W,numChannels);
```

Apply the layer normalization operation using the `layernorm` function. Specify the offset and scale formats as "SSC" (spatial, spatial, channel) using the 'OffsetFormat' and 'ScaleFormat' options, respectively.

```
Y = layernorm(X,offset,scaleFactor, 'OffsetFormat', "SSC", 'ScaleFormat', "SSC");
```

View the size and the format of the output data.

```
size(Y)
```

```
ans = 1×4
      224   224     3   128
```

```
dims(Y)
```

```
ans =
'SSCB'
```



## Input Arguments

### X — Input data

`darray` | numeric array

Input data, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

If X is an unformatted `darray` or a numeric array, then you must specify the format using the `DataFormat` option. If X is a numeric array, then either `scaleFactor` or `offset` must be a `darray` object.

X must have a 'C' (channel) dimension.

### offset — Offset

`darray` | numeric array

Offset  $\beta$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

The size and format of the offset depends on the type of transformation.

Task	Description
Channel-wise transformation	<p>Array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input X.</p> <p>For channel-wise transformation, if <code>offset</code> is a formatted <code>darray</code> object, then the nonsingleton dimension must have label 'C' (channel).</p>
Element-wise transformation	<p>Array with a 'C' (channel) dimension with the same size as the 'C' (channel) dimension of the input X and zero or the same number of 'S' (spatial), 'T' (time), and 'U' (unspecified) dimensions of the input X.</p> <p>Each dimension must have size 1 or have sizes matching the corresponding dimensions in the input X. For any repeated dimensions, for example, multiple 'S' (spatial) dimensions, the sizes must match the corresponding dimensions in X or must all be singleton.</p> <p>The software automatically expands any singleton dimensions to match the size of a single observation in the input X.</p> <p>For element-wise transformation, if <code>offset</code> is a numeric array or an unformatted <code>darray</code>, then you must specify the offset format using the '<code>OffsetFormat</code>' option.</p>

### scaleFactor — Scale factor

`darray` | numeric array

Scale factor  $\gamma$ , specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

The size and format of the offset depends on the type of transformation.

Task	Description
Channel-wise transformation	<p>Array with one nonsingleton dimension with size matching the size of the 'C' (channel) dimension of the input X.</p> <p>For channel-wise transformation, if <code>scaleFactor</code> is a formatted <code>darray</code> object, then the nonsingleton dimension must have label 'C' (channel).</p>
Element-wise transformation	<p>Array with a 'C' (channel) dimension with the same size as the 'C' (channel) dimension of the input X and zero or the same number of 'S' (spatial), 'T' (time), and 'U' (unspecified) dimensions of the input X.</p> <p>Each dimension must have size 1 or have sizes matching the corresponding dimensions in the input X. For any repeated dimensions, for example, multiple 'S' (spatial) dimensions, the sizes must match the corresponding dimensions in X or must all be singleton.</p> <p>The software automatically expands any singleton dimensions to match the size of a single observation in the input X.</p> <p>For element-wise transformation, if <code>scaleFactor</code> is a numeric array or an unformatted <code>darray</code>, then you must specify the scale format using the 'ScaleFormat' option.</p>

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Data Types: `char` | `string`

### **Epsilon — Constant to add to mini-batch variances**

`1e-5` (default) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

Epsilon must be greater than or equal to `1e-5`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 unformatted `darray`.

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

Example: `'OffsetFormat', "SSCB"`

Data Types: `char | string`

### **OperationDimension — Dimension to normalize over**

`"auto" (default) | "channel-only" | "spatial-channel" | "batch-excluded"`

Dimension to normalize over, specified as one of these values:

- "auto" — For feature, sequence, 1-D image, or spatial-temporal input, normalize over the channel dimension. Otherwise, normalize over the spatial and channel dimensions.
- "channel-only" — Normalize over the channel dimension.
- "spatial-channel" — Normalize over the spatial and channel dimensions.
- "batch-excluded" — Normalize over all dimensions except for the batch dimension.

## **Output Arguments**

### **Y — Normalized data**

`darray`

Normalized data, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, Y has the same dimension labels as X. If the input data is not a formatted `darray`, Y is an unformatted `darray` 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  $\mu_L$  and variance  $\sigma_L^2$  over the spatial, time, and channel dimensions for each observation independently. Then, it calculates the normalized activations as

$$\hat{x}_i = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}}$$

where  $\epsilon$  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

$$y_i = \gamma \hat{x}_i + \beta,$$

where the offset  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

## Version History

Introduced in R2021a

### R2023a: Specify operation dimension

Specify which dimensions to normalize over using the `OperationDimension` option.

### R2023a: Epsilon supports values less than 1e-5

The `Epsilon` option also supports positive values less than 1e-5.

### R2023a: Operation normalizes over channel and spatial dimensions of 2-D and 3-D image sequence data

*Behavior changed in R2023a*

Starting in R2023a, by default, the operation normalizes 2-D and 3-D image sequence data over the channel and spatial dimensions. In previous versions, the software normalizes over all dimensions except for the batch dimension (the spatial, time, and channel dimensions). Normalization over the channel and spatial dimensions is usually better suited for these types of data. To reproduce the previous behavior, set `OperationDimension` to "batch-excluded".

### R2023a: Operation normalizes over channel dimension of 1-D image, vector sequence, and 1-D image sequence data

*Behavior changed in R2023a*

Starting in R2023a, by default, the operation normalizes 1-D image data (data with one spatial dimension and no time dimension), vector sequence (data with a time dimension and no spatial dimensions) and 1-D image sequence data (data with one spatial dimension and a time dimension) over the channel dimension. In previous versions, the software normalizes over all dimensions except for the batch dimension (the spatial, time, and channel dimensions). Normalization over the channel dimension is usually better suited for these types of data. To reproduce the previous behavior, set `OperationDimension` to "batch-excluded".

## 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:
  - `X`
  - `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”

# 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  $\gamma$  and shifts it by a learnable offset  $\beta$ .

## 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 1-1198, “Learning Rate and Regularization” on page 1-1200, 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) | positive scalar

Constant to add to the mini-batch variances, specified as a positive scalar.

The software adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

Epsilon must be greater than or equal to 1e-5.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### NumChannels — Number of input channels

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

#### **OperationDimension — Dimension to normalize over**

"auto" (default) | "channel-only" | "spatial-channel" | "batch-excluded"

Dimension to normalize over, specified as one of these values:

- "auto" — For feature, sequence, 1-D image, or spatial-temporal input, normalize over the channel dimension. Otherwise, normalize over the spatial and channel dimensions.
- "channel-only" — Normalize over the channel dimension.
- "spatial-channel" — Normalize over the spatial and channel dimensions.
- "batch-excluded" — Normalize over all dimensions except for the batch dimension.

#### **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  $\gamma$ , specified as a numeric array.

The channel scale factors are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Scale` is nonempty, then the software uses the `Scale` property as the initial value. If `Scale` is empty, then the software uses the initializer specified by `ScaleInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

Data Types: `single` | `double`

### Offset — Channel offsets

`[]` (default) | numeric array

Channel offsets  $\beta$ , specified as a numeric vector.

The channel offsets are learnable parameters. When you train a network using the `trainNetwork` function or initialize a `dlnetwork` object, if `Offset` is nonempty, then the software uses the `Offset` property as the initial value. If `Offset` is empty, then the software uses the initializer specified by `OffsetInitializer`.

Depending on the type of layer input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically reshape this property to have of the following sizes:

Layer Input	Property Size
feature input	NumChannels-by-1
vector sequence input	
1-D image input	1-by-NumChannels

Layer Input	Property Size
1-D image sequence input	
2-D image input	1-by-1-by-NumChannels
2-D image sequence input	
3-D image input	1-by-1-by-1-by-NumChannels
3-D image sequence input	

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### ScaleL2Factor — L<sub>2</sub> regularization factor for scale factors

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the scale factors in a layer. For example, if `ScaleL2Factor` is 2, then the L<sub>2</sub> regularization for the offsets in the layer is twice the global L<sub>2</sub> regularization factor. You can specify the global L<sub>2</sub> regularization factor using the `trainingOptions` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### OffsetL2Factor — L<sub>2</sub> regularization factor for offsets

1 (default) | nonnegative scalar

L<sub>2</sub> regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L<sub>2</sub> regularization factor to determine the learning rate for the offsets in a layer. For example, if `OffsetL2Factor` is 2, then the L<sub>2</sub> 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.

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 the name ' '.

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
OperationDimension: 'auto'
```

```
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	''	2-D Convolution	16 3x3 convolutions with stride [1 1] and padding [1 1]
3	''	Layer Normalization	Layer normalization
4	''	ReLU	ReLU
5	''	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6	''	2-D Convolution	32 3x3 convolutions with stride [1 1] and padding [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

### Layer Normalization Layer

The layer normalization operation normalizes the elements  $x_i$  of the input by first calculating the mean  $\mu_L$  and variance  $\sigma_L^2$  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  $\epsilon$  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  $\beta$  and scale factor  $\gamma$  are learnable parameters that are updated during network training.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `LayerNormalizationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `LayerNormalizationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2021a

### R2023a: Specify operation dimension

Specify which dimensions to normalize over using the `OperationDimension` option.

### R2023a: Epsilon supports values less than 1e-5

The `Epsilon` option also supports positive values less than 1e-5.

### R2023a: Layer supports 1-D image sequence data

`LayerNormalizationLayer` objects support normalizing 1-D image sequence data (data with one spatial and one time dimension).

### R2023a: Layer normalizes over channel and spatial dimensions of sequence data

*Behavior changed in R2023a*

Starting in R2023a, by default, the layer normalizes sequence data over the channel and spatial dimensions. In previous versions, the software normalizes over all dimensions except for the batch dimension (the spatial, time, and channel dimensions). Normalization over the channel and spatial dimensions is usually better suited for this type of data. To reproduce the previous behavior, set `OperationDimension` to "batch-excluded".

## 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 Neural Network for Classification"

"Train Convolutional Neural Network for Regression"

"Deep Learning in MATLAB"

"Specify Layers of Convolutional Neural Network"

"List of Deep Learning Layers"

## leakyrelu

Apply leaky rectified linear unit activation

### Syntax

```
Y = leakyrelu(X)
Y = leakyrelu(X, 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 `darray` data. If you want to apply leaky ReLU activation within a `layerGraph` object or `Layer` array, use the following layer:

- `leakyReluLayer`
- 

`Y = leakyrelu(X)` computes the leaky ReLU activation of the input `X` by applying a threshold operation. All values in `X` less than zero are multiplied by a default scale factor of `0.01`.

`Y = leakyrelu(X, scaleFactor)` specifies the scale factor for the leaky ReLU operation.

### Examples

#### Apply Leaky ReLU Operation

Create a formatted `darray` 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);
X = darray(X, "SSCB");
```

View the size and format of the input data.

```
size(X)
ans = 1x4
```



```
28 28 3 128
```

```
dims(X)
```

```
ans =  
'SSCB'
```

Apply the leaky ReLU operation using the `leakyrelu` function.

```
Y = leakyrelu(X);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
```

```
28 28 3 128
```

```
dims(Y)
```

```
ans =  
'SSCB'
```

### Specify Leaky ReLU Scale

Create a formatted `darray` 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);  
X = darray(X,"SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1×4
```

```
28 28 3 128
```

```
dims(X)
```

```
ans =  
'SSCB'
```

Apply the leaky ReLU operation using the `leakyrelu` function and specify a scale of 0.5.

```
Y = leakyrelu(X,0.5);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
      28    28     3   128
```

```
dims(Y)
```

```
ans =
'SSCB'
```

## Input Arguments

### **X — Input data**

`dlarray` object

Input data, specified as a formatted or unformatted `dlarray` object.

### **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

### **Y — Leaky ReLU activations**

`dlarray`

Leaky ReLU activations, returned as a `dlarray`. The output `Y` has the same underlying data type as the input `X`.

If the input data `X` is a formatted `dlarray`, `Y` has the same dimension format as `X`. If the input data is not a formatted `dlarray`, `Y` is an unformatted `dlarray` with the same dimension order as the input data.

## Version History

**Introduced in R2019b**

## 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 `X` 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](#) | [gelu](#) | [dlgradient](#) | [dlfeval](#)

**Topics**

["Define Custom Training Loops, Loss Functions, and Networks"](#)

["Train Network Using Model Function"](#)

["List of Functions with dlarray Support"](#)

# 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 the name ' '.

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	''	2-D Convolution	16 3x3 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	Batch Normalization	Batch normalization
4	''	Leaky ReLU	Leaky ReLU with scale 0.01
5	''	2-D Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6	''	2-D Convolution	32 3x3 convolutions with stride [1 1] and padding [0 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

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

LeakyReLU objects apply an element-wise operation and supports input data of any format. The layer does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

Introduced in R2017b

## 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 Neural 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"](#)

# lbfgsState

State of limited-memory BFGS (L-BFGS) solver

## Description

An `lbfgsState` object stores information about steps in the L-BFGS algorithm.

The L-BFGS algorithm [1] is a quasi-Newton method that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The L-BFGS algorithm is best suited for small networks and data sets that you can process in a single batch.

Use `lbfgsState` objects in conjunction with the `lbfgsupdate` function to train a neural network using the L-BFGS algorithm.

## Creation

### Syntax

```
solverState = lbfgsState  
solverState = lbfgsState(Name=Value)
```

### Description

`solverState = lbfgsState` creates an L-BFGS state object with a history size of 10 and an initial inverse Hessian factor of 1.

`solverState = lbfgsState(Name=Value)` sets the `HistorySize` and `InitialInverseHessianFactor` properties using one or more name-value arguments.

## Properties

### L-BFGS State

#### HistorySize — Number of state updates to store

10 (default) | positive integer

Number of state updates to store, specified as a positive integer. Values between 3 and 20 suit most tasks.

The L-BFGS algorithm uses a history of gradient calculations to approximate the Hessian matrix recursively. For more information, see “Limited-Memory BFGS” on page 1-1225.

After creating the `lbfgsState` object, this property is read-only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### InitialInverseHessianFactor — Initial value that characterizes approximate inverse Hessian matrix

1 (default) | positive scalar



This property is read-only.

Initial value that characterizes the approximate inverse Hessian matrix, specified as a positive scalar.

To save memory, the L-BFGS algorithm does not store and invert the dense Hessian matrix  $B$ . Instead, the algorithm uses the approximation  $B_{k-m}^{-1} \approx \lambda_k I$ , where  $m$  is the history size, the inverse Hessian factor  $\lambda_k$  is a scalar, and  $I$  is the identity matrix, and stores the scalar inverse Hessian factor only. The algorithm updates the inverse Hessian factor at each step.

The `InitialInverseHessianFactor` property is the value of  $\lambda_0$ .

For more information, see “Limited-Memory BFGS” on page 1-1225.

After creating the `lbfgsState` object, this property is read-only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **InverseHessianFactor** – Value that characterizes approximate inverse Hessian matrix

1 (default) | positive scalar

Value that characterizes the approximate inverse Hessian matrix, specified as a positive scalar.

To save memory, the L-BFGS algorithm does not store and invert the dense Hessian matrix  $B$ . Instead, the algorithm uses the approximation  $B_{k-m}^{-1} \approx \lambda_k I$ , where  $m$  is the history size, the inverse Hessian factor  $\lambda_k$  is a scalar, and  $I$  is the identity matrix, and stores the scalar inverse Hessian factor only. The algorithm updates the inverse Hessian factor at each step.

For more information, see “Limited-Memory BFGS” on page 1-1225.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **StepHistory** – Step history

{ } (default) | cell array

Step history, specified as a cell array.

The L-BFGS algorithm uses a history of gradient calculations to approximate the Hessian matrix recursively. For more information, see “Limited-Memory BFGS” on page 1-1225.

Data Types: `cell`

### **GradientsDifferenceHistory** – Gradients difference history

{ } (default) | cell array

Gradients difference history, specified as a cell array.

The L-BFGS algorithm uses a history of gradient calculations to approximate the Hessian matrix recursively. For more information, see “Limited-Memory BFGS” on page 1-1225.

Data Types: `cell`

### **HistoryIndices** – History indices

0-by-1 vector (default) | row vector

History indices, specified as a row vector.

HistoryIndices is a 1-by-HistorySize vector, where StepHistory(i) and GradientsDifferenceHistory(i) correspond to iteration HistoryIndices(i).

For more information, see “Limited-Memory BFGS” on page 1-1225.

Data Types: double

### **Iteration Information**

#### **Loss — Loss**

[] (default) | dLarray scalar | numeric scalar

This property is read-only.

Loss, specified as a dLarray scalar, a numeric scalar, or [].

If the state object is the output of the lbfgsupdate function, then Loss is the first output of the loss function that you pass to the lbfgsupdate function. Otherwise, Loss is [].

#### **Gradients — Gradients**

[] (default) | dLarray object | numeric array | cell array | structure | table

This property is read-only.

Gradients, specified as a dLarray object, a numeric array, a cell array, a structure, a table, or [].

If the state object is the output of the lbfgsupdate function, then Gradients is the second output of the loss function that you pass to the lbfgsupdate function. Otherwise, Gradients is [].

#### **AdditionalLossFunctionOutputs — Additional loss function outputs**

1-by-0 cell array (default) | cell array

This property is read-only.

Additional loss function outputs, specified as a cell array.

If the state object is the output of the lbfgsupdate function, then AdditionalLossFunctionOutputs is a cell array containing additional outputs of the loss function that you pass to the lbfgsupdate function. Otherwise, AdditionalLossFunctionOutputs is a 1-by-0 cell array.

Data Types: cell

#### **StepNorm — Norm of step**

[] (default) | dLarray scalar | numeric scalar

This property is read-only.

Norm of the step, specified as a dLarray scalar, numeric scalar, or [].

If the state object is the output of the lbfgsupdate function, then StepNorm is the norm of the step that the lbfgsupdate function calculates. Otherwise, StepNorm is [].

#### **GradientsNorm — Norm of gradients**

[] (default) | dLarray scalar | numeric scalar

This property is read-only.

Norm of the gradients, specified as a `darray` scalar, a numeric scalar, or `[]`.

If the state object is the output of the `lbfgsupdate` function, then `GradientsNorm` is the norm of the second output of the loss function that you pass to the `lbfgsupdate` function. Otherwise, `GradientsNorm` is `[]`.

### **LineSearchStatus — Status of line search algorithm**

`""` (default) | `"completed"` | `"failed"`

This property is read-only.

Status of the line search algorithm, specified as `""`, `"completed"`, or `"failed"`.

If the state object is the output of the `lbfgsupdate` function, then `LineSearchStatus` is one of these values:

- `"completed"` — The algorithm finds a learning rate that satisfies the `LineSearchMethod` and `MaxNumLineSearchIterations` options that the `lbfgsupdate` function uses.
- `"failed"` — The algorithm fails to find a learning rate that satisfies the `LineSearchMethod` and `MaxNumLineSearchIterations` options that the `lbfgsupdate` function uses.

Otherwise, `LineSearchStatus` is `""`.

### **LineSearchMethod — Method solver uses to find suitable learning rate**

`""` (default) | `"weak-wolfe"` | `"strong-wolfe"` | `"backtracking"`

This property is read-only.

Method solver uses to find a suitable learning rate, specified as `"weak-wolfe"`, `"strong-wolfe"`, `"backtracking"`, or `[]`.

If the state object is the output of the `lbfgsupdate` function, then `LineSearchMethod` is the line search method that the `lbfgsupdate` function uses. Otherwise, `LineSearchMethod` is `""`.

### **MaxNumLineSearchIterations — Maximum number of line search iterations**

`0` (default) | nonnegative integer

This property is read-only.

Maximum number of line search iterations, specified as a nonnegative integer.

If the state object is the output of the `lbfgsupdate` function, then `MaxNumLineSearchIterations` is the maximum number of line search iterations that the `lbfgsupdate` function uses. Otherwise, `MaxNumLineSearchIterations` is `0`.

Data Types: `double`

## **Examples**

### **Create L-BFGS Solver State Object**

Create an L-BFGS solver state object.

```
solverState = lbfgsState
```

```
solverState =
  LBFGSState with properties:

    InverseHessianFactor: 1
        StepHistory: {}
  GradientsDifferenceHistory: {}
        HistoryIndices: [1x0 double]

  Iteration Information
        Loss: []
        Gradients: []
  AdditionalLossFunctionOutputs: {1x0 cell}
        GradientsNorm: []
        StepNorm: []
  LineSearchStatus: ""

  Show all properties
```

### Update Learnable Parameters in Neural Network

Load the iris flower dataset.

```
[XTrain, TTrain] = iris_dataset;
```

Convert the predictors to a `darray` object with format "CB" (channel, batch).

```
XTrain = darray(XTrain, "CB");
```

Define the network architecture.

```
numInputFeatures = size(XTrain,1);
numClasses = size(TTrain,1);
numHiddenUnits = 32;
```

```
layers = [
  featureInputLayer(numInputFeatures)
  fullyConnectedLayer(numHiddenUnits)
  reluLayer
  fullyConnectedLayer(numHiddenUnits)
  reluLayer
  fullyConnectedLayer(numClasses)
  softmaxLayer];
```

```
net = dlnetwork(layers);
```

Define the `modelLoss` function, listed in the Model Loss Function on page 1-1219 section of the example. This function takes as input a neural network, input data, and targets. The function returns the loss and the gradients of the loss with respect to the network learnable parameters.

The `lbfgsupdate` function requires a loss function with the syntax `[loss,gradients] = f(net)`. Create a variable that parameterizes the evaluated `modelLoss` function to take a single input argument.

```
lossFcn = @(net) dlfeval(@modelLoss,net,XTrain,TTrain);
```

Initialize an L-BFGS solver state object with a maximum history size of 3 and an initial inverse Hessian approximation factor of 1.1.

```
solverState = lbfgsState( ...
    HistorySize=3, ...
    InitialInverseHessianFactor=1.1);
```

Train the network for 10 epochs.

```
numEpochs = 10;
for i = 1:numEpochs
    [net, solverState] = lbfgsupdate(net, lossFcn, solverState);
end
```

### Model Loss Function

The `modelLoss` function takes as input a neural network `net`, input data `X`, and targets `T`. The function returns the loss and the gradients of the loss with respect to the network learnable parameters.

```
function [loss, gradients] = modelLoss(net, X, T)

Y = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);

end
```

## Algorithms

### Limited-Memory BFGS

The L-BFGS algorithm [1] is a quasi-Newton method that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The L-BFGS algorithm is best suited for small networks and data sets that you can process in a single batch.

The algorithm updates learnable parameters  $W$  at iteration  $k+1$  using the update step given by

$$W_{k+1} = W_k - \eta_k B_k^{-1} \nabla J(W_k),$$

where  $W_k$  denotes the weights at iteration  $k$ ,  $\eta_k$  is the learning rate at iteration  $k$ ,  $B_k$  is an approximation of the Hessian matrix at iteration  $k$ , and  $\nabla J(W_k)$  denotes the gradients of the loss with respect to the learnable parameters at iteration  $k$ .

The L-BFGS algorithm computes the matrix-vector product  $B_k^{-1} \nabla J(W_k)$  directly. The algorithm does not require computing the inverse of  $B_k$ .

To save memory, the L-BFGS algorithm does not store and invert the dense Hessian matrix  $B$ . Instead, the algorithm uses the approximation  $B_k^{-1} \approx \lambda_k I$ , where  $m$  is the history size, the inverse Hessian factor  $\lambda_k$  is a scalar, and  $I$  is the identity matrix, and stores the scalar inverse Hessian factor only. The algorithm updates the inverse Hessian factor at each step.

To compute the matrix-vector product  $B_k^{-1} \nabla J(W_k)$  directly, the L-BFGS algorithm uses this recursive algorithm:

- 1** Set  $\mathbf{r} = B_{k-m}^{-1} \nabla J(W_k)$ , where  $m$  is the history size.
- 2** For  $i = m, \dots, 1$ :
  - a** Let  $\beta = \frac{1}{\mathbf{s}_{k-i}^\top \mathbf{y}_{k-i}} \mathbf{y}_{k-i}^\top \mathbf{r}$ , where  $\mathbf{s}_{k-i}$  and  $\mathbf{y}_{k-i}$  are the step and gradient differences for iteration  $k-i$ , respectively.
  - b** Set  $\mathbf{r} = \mathbf{r} + \mathbf{s}_{k-i}(\mathbf{a}_{k-i} - \beta)$ , where  $\mathbf{a}$  is derived from  $\mathbf{s}$ ,  $\mathbf{y}$ , and the gradients of the loss with respect to the loss function. For more information, see [1].
- 3** Return  $B_k^{-1} \nabla J(W_k) = \mathbf{r}$ .

## Version History

Introduced in R2023a

## References

- [1] Liu, Dong C., and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization." *Mathematical programming* 45, no. 1 (August 1989): 503-528. <https://doi.org/10.1007/BF01589116>.

## See Also

`lbfgsupdate` | `dlnetwork` | `dlarray` | `dlgradient`

## Topics

- "Solve Partial Differential Equation with L-BFGS Method and Deep Learning"
- "Define Custom Training Loops, Loss Functions, and Networks"
- "Specify Training Options in Custom Training Loop"
- "Train Network Using Custom Training Loop"

# lbfgsupdate

Update parameters using limited-memory BFGS (L-BFGS)

## Syntax

```
[netUpdated, solverStateUpdated] = lbfgsupdate(net, lossFcn, solverState)
[parametersUpdated, solverStateUpdated] = lbfgsupdate(parameters, solverState)
```

## Description

Update the network learnable parameters in a custom training loop using the limited-memory BFGS (L-BFGS) algorithm.

The L-BFGS algorithm [1] is a quasi-Newton method that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The L-BFGS algorithm is best suited for small networks and data sets that you can process in a single batch.

---

**Note** This function applies the L-BFGS 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 an SGDM, Adam, or RMSProp training options object using the `trainingOptions` function.
  - Use the options object with the `trainNetwork` function.
- 

`[netUpdated, solverStateUpdated] = lbfgsupdate(net, lossFcn, solverState)` updates the learnable parameters of the network `net` using the L-BFGS algorithm with the specified loss function and solver state. Use this syntax in a training loop to iteratively update a network defined as a `dlnetwork` object.

`[parametersUpdated, solverStateUpdated] = lbfgsupdate(parameters, solverState)` updates the learnable parameters in `parameters` using the L-BFGS algorithm with the specified loss function and solver state. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined as a function.

## Examples

### Update Learnable Parameters in Neural Network

Load the iris flower dataset.

```
[XTrain, TTrain] = iris_dataset;
```

Convert the predictors to a `dlarray` object with format "CB" (channel, batch).

```
XTrain = dlarray(XTrain, "CB");
```

Define the network architecture.

```
numInputFeatures = size(XTrain,1);
numClasses = size(TTrain,1);
numHiddenUnits = 32;

layers = [
    featureInputLayer(numInputFeatures)
    fullyConnectedLayer(numHiddenUnits)
    reluLayer
    fullyConnectedLayer(numHiddenUnits)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer];

net = dlnetwork(layers);
```

Define the `modelLoss` function, listed in the Model Loss Function on page 1-1222 section of the example. This function takes as input a neural network, input data, and targets. The function returns the loss and the gradients of the loss with respect to the network learnable parameters.

The `lbfgsupdate` function requires a loss function with the syntax `[loss,gradients] = f(net)`. Create a variable that parameterizes the evaluated `modelLoss` function to take a single input argument.

```
lossFcn = @(net) dlfeval(@modelLoss,net,XTrain,TTrain);
```

Initialize an L-BFGS solver state object with a maximum history size of 3 and an initial inverse Hessian approximation factor of 1.1.

```
solverState = lbfgsState( ...
    HistorySize=3, ...
    InitialInverseHessianFactor=1.1);
```

Train the network for 10 epochs.

```
numEpochs = 10;
for i = 1:numEpochs
    [net, solverState] = lbfgsupdate(net,lossFcn,solverState);
end
```

### **Model Loss Function**

The `modelLoss` function takes as input a neural network `net`, input data `X`, and targets `T`. The function returns the loss and the gradients of the loss with respect to the network learnable parameters.

```
function [loss, gradients] = modelLoss(net, X, T)

Y = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);
```



end

## Input Arguments

### net — Neural network

`dlnetwork` object

Neural network, specified as a `dlnetwork` object.

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

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

### parameters — Learnable parameters

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

Learnable parameters, specified as a `dlarray` object, a numeric array, a cell array, a structure, or a table.

If you specify `parameters` as a table, it must contain these variables:

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

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

If `parameters` is a numeric array, then `lossFcn` must not use the `dlgradient` function.

### lossFcn — Loss function

function handle | `AcceleratedFunction` object

Loss function, specified as a function handle or an `AcceleratedFunction` object with the syntax `[loss,gradients] = f(net)`, where `loss` and `gradients` correspond to the loss and gradients of the loss with respect to the learnable parameters, respectively.

To parametrize a model loss function that has a call to the `dlgradient` function, specify the loss function as `@(net) dlfeval(@modelLoss,net,arg1,...,argN)`, where `modelLoss` is a function with the syntax `[loss,gradients] = modelLoss(net,arg1,...,argN)` that returns the loss and gradients of the loss with respect to the learnable parameters in `net` given arguments `arg1,...,argN`.

If `parameters` is a numeric array, then the loss function must not use the `dlgradient` or `dlfeval` functions.

If the loss function has more than two outputs, also specify the `NumLossFunctionOutputs` argument.

Data Types: `function_handle`

**solverState — Solver state**

`lbfgsState` object | []

Solver state, specified as an `lbfgsState` object or [].

**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: `lbfgsupdate(net, lossFcn, solverState, LineSearchMethod="strong-wolfe")` updates the learnable parameters in `net` and searches for a learning rate that satisfies the strong Wolfe conditions.

**LineSearchMethod — Method to find suitable learning rate**

"weak-wolfe" (default) | "strong-wolfe" | "backtracking"

Method to find suitable learning rate, specified as one of these values:

- "weak-wolfe" — Search for a learning rate that satisfies the weak Wolfe conditions. This method maintains a positive definite approximation of the inverse Hessian matrix.
- "strong-wolfe" — Search for a learning rate that satisfies the strong Wolfe conditions. This method maintains a positive definite approximation of the inverse Hessian matrix.
- "backtracking" — Search for a learning rate that satisfies sufficient decrease conditions. This method does not maintain a positive definite approximation of the inverse Hessian matrix.

**MaxNumLineSearchIterations — Maximum number of line search iterations**

20 (default) | positive integer

Maximum number of line search iterations to determine learning rate, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**NumLossFunctionOutputs — Number of loss function outputs**

2 (default) | integer greater than or equal to two

Number of loss function outputs, specified as an integer greater than or equal to two. Set this option when `lossFcn` has more than two output arguments.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Output Arguments****netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as a `dlnetwork` object.

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

**parametersUpdated — Updated learnable parameters**

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

Updated learnable parameters, returned as an object with the same type as `parameters`.

### **solverStateUpdated** — Updated solver state

`lbfgsState`

Updated solver state, returned as an `lbfgsState` state object.

## Algorithms

### Limited-Memory BFGS

The L-BFGS algorithm [1] is a quasi-Newton method that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The L-BFGS algorithm is best suited for small networks and data sets that you can process in a single batch.

The algorithm updates learnable parameters  $W$  at iteration  $k+1$  using the update step given by

$$W_{k+1} = W_k - \eta_k B_k^{-1} \nabla J(W_k),$$

where  $W_k$  denotes the weights at iteration  $k$ ,  $\eta_k$  is the learning rate at iteration  $k$ ,  $B_k$  is an approximation of the Hessian matrix at iteration  $k$ , and  $\nabla J(W_k)$  denotes the gradients of the loss with respect to the learnable parameters at iteration  $k$ .

The L-BFGS algorithm computes the matrix-vector product  $B_k^{-1} \nabla J(W_k)$  directly. The algorithm does not require computing the inverse of  $B_k$ .

To save memory, the L-BFGS algorithm does not store and invert the dense Hessian matrix  $B$ . Instead, the algorithm uses the approximation  $B_k^{-1} \approx \lambda_k I$ , where  $m$  is the history size, the inverse Hessian factor  $\lambda_k$  is a scalar, and  $I$  is the identity matrix, and stores the scalar inverse Hessian factor only. The algorithm updates the inverse Hessian factor at each step.

To compute the matrix-vector product  $B_k^{-1} \nabla J(W_k)$  directly, the L-BFGS algorithm uses this recursive algorithm:

- 1 Set  $\mathbf{r} = B_{k-m}^{-1} \nabla J(W_k)$ , where  $m$  is the history size.
- 2 For  $i = m, \dots, 1$ :
  - a Let  $\beta = \frac{1}{\mathbf{s}_{k-i}^\top \mathbf{y}_{k-i}} \mathbf{y}_{k-i}^\top \mathbf{r}$ , where  $\mathbf{s}_{k-i}$  and  $\mathbf{y}_{k-i}$  are the step and gradient differences for iteration  $k-i$ , respectively.
  - b Set  $\mathbf{r} = \mathbf{r} + \mathbf{s}_{k-i}(\mathbf{a}_{k-i} - \beta)$ , where  $\mathbf{a}$  is derived from  $\mathbf{s}$ ,  $\mathbf{y}$ , and the gradients of the loss with respect to the loss function. For more information, see [1].
- 3 Return  $B_k^{-1} \nabla J(W_k) = \mathbf{r}$ .

## Version History

Introduced in R2023a

## References

- [1] Liu, Dong C., and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization." *Mathematical programming* 45, no. 1 (August 1989): 503-528. <https://doi.org/10.1007/BF01589116>.

## Extended Capabilities

### GPU Arrays

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

Usage notes and limitations:

- When `lossFcn` outputs data of type `gpuArray` or `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

`adamupdate` | `rmspropupdate` | `sgdmupdate` | `dlupdate` | `lbfgsState` | `dlnetwork` | `dlarray` | `dlgradient`

### Topics

"Define Custom Training Loops, Loss Functions, and Networks"

"Specify Training Options in Custom Training Loop"

"Train Network Using Custom Training Loop"

# loadTFLiteModel

Load TensorFlow Lite model

## Syntax

```
net = loadTFLiteModel(modelFileName)
```

## Description

`net = loadTFLiteModel(modelFileName)` loads a pretrained TensorFlow Lite model file `modelFileName` and returns a `TFLiteModel` object.

Use this `TFLiteModel` object with the `predict` function in your MATLAB code to perform inference in MATLAB execution, code generation, or MATLAB Function block in Simulink models. For more information, see “Prerequisites for Deep Learning with TensorFlow Lite Models”.

To use this function, you must install the Deep Learning Toolbox Interface for TensorFlow Lite support package.

## Examples

### Perform Inference with TensorFlow Lite Model

Suppose that your current working directory contains a TensorFlow Lite Model named `mobilenet_v1_0.5_224.tflite`.

Load the model by using the `loadTFLite` function. Inspect the object this function creates.

```
net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
disp(net)
```

```
TFLiteModel with properties:
    ModelName: 'mobilenet_v1_0.5_224.tflite'
    NumInputs: 1
    NumOutputs: 1
    InputSize: {[224 224 3]}
    OutputSize: {[1001 1]}
    NumThreads: 8
           Mean: 127.5000
    StandardDeviation: 127.5000
```

Create a MATLAB function that can perform inference using the object `net`. This function loads the Mobilenet-V1 model into a persistent network object. Then the function performs prediction by passing the network object to the `predict` function. Subsequent calls to this function reuse this the persistent object.

```
function out = tflite_predict(in)
persistent net;
if isempty(net)
    net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
```

```
end  
out = predict(net,in);  
end
```

For an example that shows how to generate code for this function and deploy on Raspberry Pi hardware, see “Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”.

---

**Note** By default, the `Mean` and `StandardDeviation` properties of a `TFLiteModel` object are both set to 127.5. To change these default values after you create the object, make assignments by using the dot notation. For example:

```
net.Mean = 0;  
net.StandardDeviation = 1;
```

If the input data is not normalized, you must set `Mean` to 0 and `StandardDeviation` to 1. Otherwise, set these properties based on how the input data is normalized.

---

## Input Arguments

**modelName** — TensorFlow Lite model file name

character vector | string scalar

Name of the TensorFlow Lite model file, specified as a character vector or a string scalar.

## Output Arguments

**net** — Object representing TensorFlow Lite model

`TFLiteModel` object

`TFLiteModel` object that represents the TensorFlow Lite model file.

## Version History

Introduced in R2022a

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

`TFLiteModel` | `predict`

## Topics

“Deploy Pose Estimation Application Using TensorFlow Lite Model (TFLite) Model on Host and Raspberry Pi”

“Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”

“Deploy Super Resolution Application That Uses TensorFlow Lite (TFLite) Model on Host and Raspberry Pi”

“Prerequisites for Deep Learning with TensorFlow Lite Models”

# lstm

Long short-term memory

## Syntax

```
Y = lstm(X,H0,C0,weights,recurrentWeights,bias)
[Y,hiddenState,cellState] = lstm(X,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 `darray` data. If you want to apply an LSTM operation within a `layerGraph` object or `Layer` array, use the following layer:

- `lstmLayer`
- 

`Y = lstm(X,H0,C0,weights,recurrentWeights,bias)` applies a long short-term memory (LSTM) calculation to input `X` using the initial hidden state `H0`, initial cell state `C0`, and parameters `weights`, `recurrentWeights`, and `bias`. The input `X` must be a formatted `darray`. The output `Y` is a formatted `darray` with the same dimension format as `X`, 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.

`[Y,hiddenState,cellState] = lstm(X,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 `X` is not a formatted `darray`. The output `Y` is an unformatted `darray` with the same dimension order as `X`, 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

### **X — Input data**

`darray` | numeric array

Input data, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. When `X` is not a formatted `darray`, you must specify the dimension label format using `'DataFormat', FMT`. If `X` is a numeric array, at least one of `H0`, `C0`, `weights`, `recurrentWeights`, or `bias` must be a `darray`.

`X` must contain a sequence dimension labeled "T". If `X` has any spatial dimensions labeled "S", they are flattened into the "C" channel dimension. If `X` does not have a channel dimension, then one is added. If `X` has any unspecified dimensions labeled "U", they must be singleton.

Data Types: `single` | `double`

### **H0 — Initial hidden state vector**

`darray` | numeric array

Initial hidden state vector, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

If `H0` is a formatted `darray`, it must contain a channel dimension labeled 'C' and optionally a batch dimension labeled 'B' with the same size as the 'B' dimension of `X`. If `H0` does not have a 'B' dimension, the function uses the same hidden state vector for each observation in `X`.

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 not a formatted `darray`, 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**

`darray` | numeric array

Initial cell state vector, specified as a formatted `darray`, an unformatted `darray`, or a numeric array.

If `C0` is a formatted `darray`, it must contain a channel dimension labeled 'C' and optionally a batch dimension labeled 'B' with the same size as the 'B' dimension of `X`. If `C0` does not have a 'B' dimension, the function uses the same cell state vector for each observation in `X`.

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 not a formatted `darray`, 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  $4 \times \text{NumHiddenUnits}$ -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 `X` multiplied by the size of each 'S' dimension of `X`, where present.

If `weights` is a formatted dlarray, it must contain a 'C' dimension of size  $4 \times \text{NumHiddenUnits}$  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  $4 \times \text{NumHiddenUnits}$ -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  $4 \times \text{NumHiddenUnits}$  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  $4 \times \text{NumHiddenUnits}$ , 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

### Y — LSTM output

darray

LSTM output, returned as a darray. The output Y has the same underlying data type as the input X.

If the input data X is a formatted darray, Y has the same dimension format as X, except for any 'S' dimensions. If the input data is not a formatted darray, Y is an unformatted darray with the same dimension order as the input data.

The size of the 'C' dimension of Y 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

darray | numeric array

Hidden state vector for each observation, returned as a darray or a numeric array with the same data type as H0.

If the input H0 is a formatted darray, then the output hiddenState is a formatted darray with the format "CB".

### cellState — Cell state vector

darray | numeric array

Cell state vector for each observation, returned as a darray or a numeric array. cellState is returned with the same data type as C0.

If the input C0 is a formatted darray, the output cellState is returned as a formatted darray with the format 'CB'.

## 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-Term Memory Layer” on page 1-1251 on the lstmLayer reference page.

## Version History

Introduced in R2019b

## 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:
  - `X`
  - `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` | `attention`

## 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”

# IstmLayer

Long short-term memory (LSTM) layer for recurrent neural network (RNN)

## Description

An LSTM layer is an RNN layer that 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 1-1237, “State” on page 1-1237, “Parameters and Initialization” on page 1-1238, “Learning Rate and Regularization” on page 1-1240, 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

This property is read-only.

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 that the layer remembers between time steps (the hidden state). The hidden state can contain information from all the 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.

The hidden state does not limit the number of time steps that the layer processes in an iteration. To split your sequences into smaller sequences for when you use the `trainNetwork` function, use the `SequenceLength` training option.

The layer outputs data with `NumHiddenUnits` channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputMode — Output mode**

`'sequence'` (default) | `'last'`

This property is read-only.

Output mode, specified as one of these values:

- `'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)

This property is read-only.

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 the 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 the 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)

This property is read-only.

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 the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (true), then the layer has three outputs with the 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 that it computes.

### **InputSize — Input size**

`'auto'` (default) | positive integer

This property is read-only.

Input size, specified as a positive integer or `'auto'`. If `InputSize` is `'auto'`, then the software automatically assigns the input size at training time.

Data Types: `double` | `char`

## Activations

### StateActivationFunction — Activation function to update cell and hidden state

'tanh' (default) | 'softsign'

This property is read-only.

Activation function to update the cell and hidden state, specified as one of these values:

- '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  $\sigma_c$  in the calculations to update the cell and hidden state. For more information on how an LSTM layer uses activation functions, see “Long Short-Term Memory Layer” on page 1-1251.

### GateActivationFunction — Activation function to apply to gates

'sigmoid' (default) | 'hard-sigmoid'

This property is read-only.

Activation function to apply to the gates, specified as one of these values:

- 'sigmoid' — Use the sigmoid function  $\sigma(x) = (1 + e^{-x})^{-1}$ .
- 'hard-sigmoid' — Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}$$

The layer uses this option as the function  $\sigma_g$  in the calculations for the layer gates.

## State

### CellState — Cell state

[] (default) | 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 you set this property manually, calls to the `resetState` function set the cell state to this value.

If `HasStateInputs` is 1 (true), then the `CellState` property must be empty.

Data Types: `single` | `double`

### HiddenState — Hidden state

[] (default) | 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 you set this property manually, calls to the `resetState` function set the hidden state to this value.

If `HasStateInputs` is 1 (true), then the `HiddenState` property must be empty.

Data Types: `single` | `double`

### 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 these values:

- '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 a standard deviation of 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 you train a neural network using the `trainNetwork` function, if `InputWeights` is nonempty, then the software uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then the software 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 you train an RNN using the `trainNetwork` function, if `RecurrentWeights` is nonempty, then the software uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then the software uses the initializer specified by `RecurrentWeightsInitializer`.

At training time `RecurrentWeights` is a  $4 \times \text{NumHiddenUnits}$ -by- $\text{NumHiddenUnits}$  matrix.

### **Bias – Layer biases**

`[]` (default) | numeric vector

Layer biases, specified as a numeric vector.

The bias vector is a concatenation of the four bias vectors for the components (gates) in the layer. The layer vertically concatenates the four vectors in this order:

- 1 Input gate
- 2 Forget gate
- 3 Cell candidate
- 4 Output gate

The layer biases are learnable parameters. When you train a neural 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  $4 \times \text{NumHiddenUnits}$ -by-1 numeric vector.

### **Learning Rate and Regularization**

#### **InputWeightsLearnRateFactor – Learning rate factor for input weights**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the input weights, 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 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 you specify 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 these components:

- 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) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the recurrent weights, 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 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 you specify using 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 these components:

- 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 these components:

- 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 —  $L_2$  regularization factor for input weights**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  regularization factor for the input weights, 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 factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the input weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual matrices in `InputWeights`, specify a 1-by-4 vector. The entries of `InputWeightsL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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 —  $L_2$  regularization factor for recurrent weights**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  regularization factor for the recurrent weights, 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 factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the recurrent weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual matrices in `RecurrentWeights`, specify a 1-by-4 vector. The entries of `RecurrentWeightsL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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 — $L_2$ regularization factor for biases**

0 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  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. The software determines the global  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual vectors in `Bias`, specify a 1-by-4 vector. The entries of `BiasL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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 the name ' '.

Data Types: char | string

### **NumInputs — Number of inputs**

1 | 3

This property is read-only.

Number of inputs to the layer.

If the `HasStateInputs` property is 0 (false), then the layer has one input with the 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 the 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'}
```

This property is read-only.

Input names of the layer.

If the `HasStateInputs` property is `0` (`false`), then the layer has one input with the 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 the 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
```

This property is read-only.

Number of outputs to the layer.

If the `HasStateOutputs` property is `0` (`false`), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (`true`), then the layer has three outputs with the 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 that it computes.

Data Types: `double`

**OutputNames — Output names**

```
{'out'} | {'out', 'hidden', 'cell'}
```

This property is read-only.

Output names of the layer.

If the `HasStateOutputs` property is `0` (`false`), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is `1` (`true`), then the layer has three outputs with the 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 that it computes.

**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

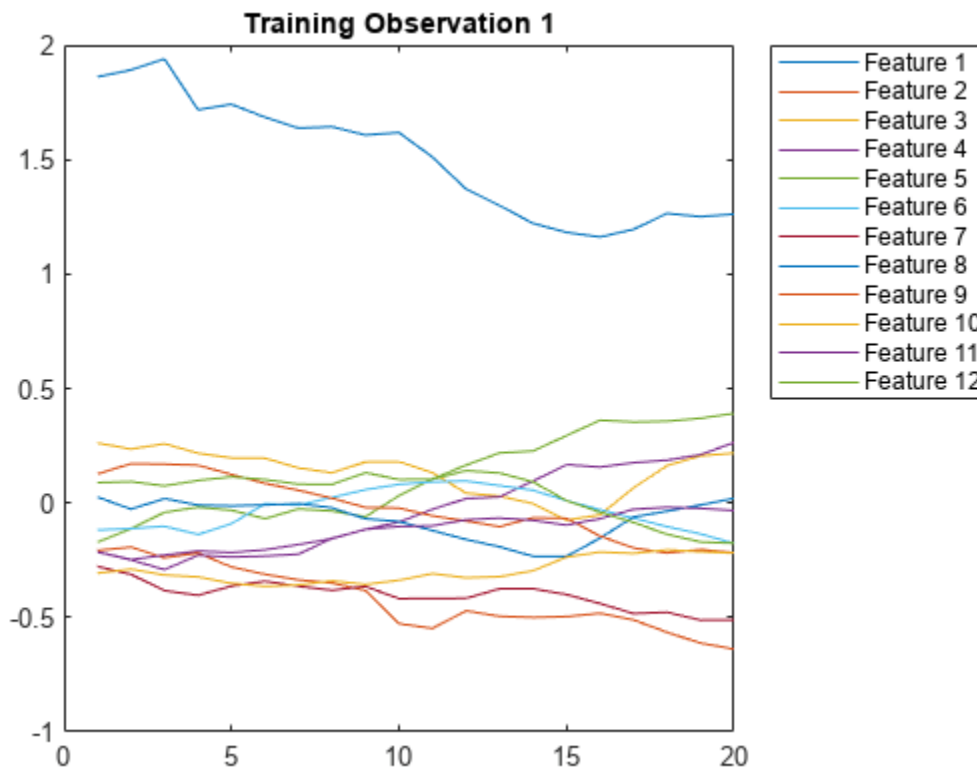
Train a deep learning LSTM network for sequence-to-label classification.

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).

```

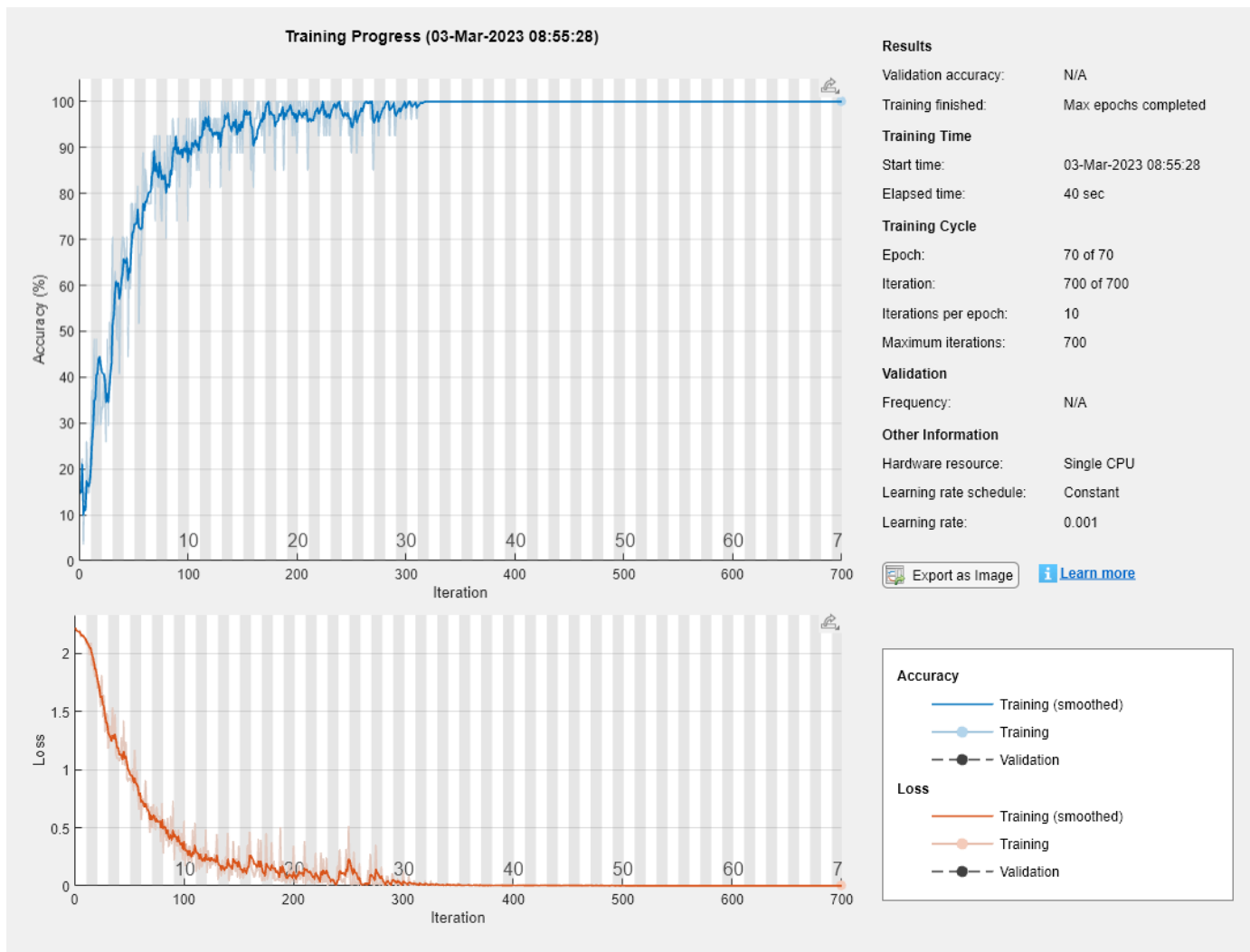
maxEpochs = 70;
miniBatchSize = 27;

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 = 0.9432
```

## 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];
```

```
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

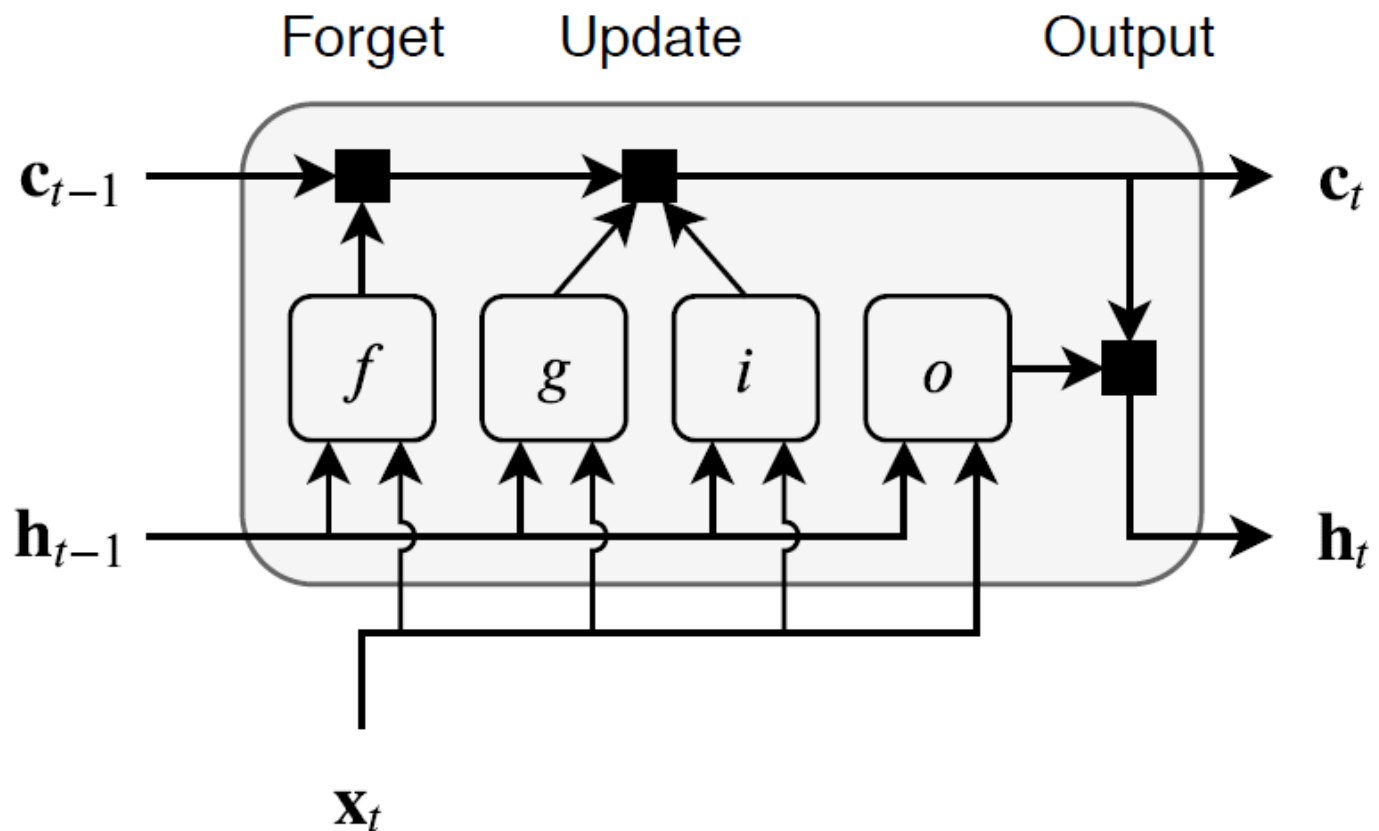
An LSTM layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data.

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*.

These 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  $t$ . This diagram shows 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. The layer concatenates the matrices according to these equations:

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

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  $\sigma_c$  denotes the state activation function. By default, the `lstmLayer` function uses the hyperbolic tangent function (`tanh`) to compute the state activation function.

These formulas describe the components at time step  $t$ .

Component	Formula
Input gate	$i_t = \sigma_g(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1} + b_i)$
Forget gate	$f_t = \sigma_g(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1} + b_f)$
Cell candidate	$g_t = \sigma_c(W_g \mathbf{x}_t + R_g \mathbf{h}_{t-1} + b_g)$
Output gate	$o_t = \sigma_g(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1} + b_o)$

In these calculations,  $\sigma_g$  denotes the gate activation function. By default, the `lstmLayer` function, 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 to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `LSTMLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	OutputMode	Output Format
"CB" (channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"CBT" (channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SB" (spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	

In `dlnetwork` objects, `LSTMLayer` objects also support these input and output format combinations.

Input Format	OutputMode	Output Format
"SCB" (spatial, channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"SSCB" (spatial, spatial, channel, batch)	"sequence"	
	"last"	
"SSSCB" (spatial, spatial, spatial, channel, batch)	"sequence"	
	"last"	
"SCBT" (spatial, channel, batch, time)	"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)
"SC" (spatial, channel)	"sequence"	"CU" (channel, unspecified)
	"last"	
"SSC" (spatial, spatial, channel)	"sequence"	
	"last"	
"SSSC" (spatial, spatial, spatial, channel)	"sequence"	
	"last"	
"CT" (channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)
	"last"	"CU" (channel, unspecified)
"SSB" (spatial, spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"SSSB" (spatial, spatial, spatial, batch)	"sequence"	
	"last"	
"BT" (batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SBT" (spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SSBT" (spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)



Input Format	OutputMode	Output Format
"SSSBT" (spatial, spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)

To use these input formats in `trainNetwork` workflows, convert the data to "CB" (channel, batch) or "CBT" (channel, batch, time) format using `flattenLayer`.

If the `HasStateInputs` property is 1 (true), then the layer has two additional inputs with the 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).

## Version History

### Introduced in R2017b

#### **R2019a: 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 neural 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'.

#### **R2019a: 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 neural 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>
- [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:

For code generation in general, the `HasStateInputs` and `HasStateOutputs` properties must be set to `0` (false).

When generating code with Intel MKL-DNN or ARM Compute Library:

- The `StateActivationFunction` property must be set to `'tanh'`.
- The `GateActivationFunction` property must be set to `'sigmoid'`.

### 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"  
"Sequence-to-One Regression Using Deep Learning"  
"Classify Videos Using Deep Learning"  
"Long Short-Term Memory Neural Networks"  
"List of Deep Learning Layers"

“Deep Learning Tips and Tricks”

# IstmProjectedLayer

Long short-term memory (LSTM) projected layer for recurrent neural network (RNN)

## Description

An LSTM projected layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data using projected learnable weights.

To compress a deep learning network, you can use *projected layers*. A projected layer is a type of deep learning layer that enables compression by reducing the number of stored learnable parameters. The layer introduces learnable projector matrices  $Q$ , replaces multiplications of the form  $Wx$ , where  $W$  is a learnable matrix, with the multiplication  $WQQ^T x$ , and stores  $Q$  and  $W' = WQ$  instead of storing  $W$ . Projecting  $x$  into a lower-dimensional space using  $Q$  typically requires less memory to store the learnable parameters and can have similarly strong prediction accuracy.

Reducing the number of learnable parameters by projecting an LSTM layer rather than reducing the number of hidden units of the LSTM layer maintains the output size of the layer and, in turn, the sizes of the downstream layers, which can result in better prediction accuracy.

## Creation

### Syntax

```
layer = lstmProjectedLayer(numHiddenUnits,outputProjectorSize,  
inputProjectorSize)  
layer = lstmProjectedLayer( __ ,Name=Value)
```

### Description

`layer = lstmProjectedLayer(numHiddenUnits,outputProjectorSize, inputProjectorSize)` creates an LSTM projected layer and sets the `NumHiddenUnits`, `OutputProjectorSize`, and `InputProjectorSize` properties.

`layer = lstmProjectedLayer( __ ,Name=Value)` sets the `OutputMode`, `HasStateInputs`, `HasStateOutputs`, “Activations” on page 1-1261, “State” on page 1-1261, “Parameters and Initialization” on page 1-1262, “Learning Rate and Regularization” on page 1-1266, and `Name` properties using one or more name-value arguments.

## Properties

### Projected LSTM

#### **NumHiddenUnits** – Number of hidden units

positive integer

This property is read-only.

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 that the layer remembers between time steps (the hidden state). The hidden state can contain information from all the 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.

The hidden state does not limit the number of time steps that the layer processes in an iteration. To split your sequences into smaller sequences for when you use the `trainNetwork` function, use the `SequenceLength` training option.

The layer outputs data with `NumHiddenUnits` channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputProjectorSize** — Output projector size

positive integer

This property is read-only.

Output projector size, specified as a positive integer.

The LSTM layer operation uses four matrix multiplications of the form  $Rh_{t-1}$ , where  $R$  denotes the recurrent weights and  $h_t$  denotes the hidden state (or, equivalently, the layer output) at time step  $t$ .

The LSTM projected layer operation instead uses multiplications of the form  $RQ_oQ_o^T h_{t-1}$ , where  $Q_o$  is a `NumHiddenUnits`-by-`OutputProjectorSize` matrix known as the *output projector*. The layer uses the same projector  $Q_o$  for each of the four multiplications.

To perform the four multiplications of the form  $Rh_{t-1}$ , an LSTM layer stores four recurrent weights  $R$ , which necessitates storing  $4 * \text{NumHiddenUnits}^2$  learnable parameters. By instead storing the  $4 * \text{NumHiddenUnits}$ -by-`OutputProjectorSize` matrix  $R' = RQ_o$  and  $Q_o$ , an LSTM projected layer can perform the multiplication  $RQ_oQ_o^T h_{t-1}$  and store only  $5 * \text{NumHiddenUnits} * \text{OutputProjectorSize}$  learnable parameters.

---

**Tip** To ensure that  $RQ_oQ_o^T h_{t-1}$  requires fewer learnable parameters, set the `OutputProjectorSize` property to a value less than  $(4/5) * \text{NumHiddenUnits}$ .

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **InputProjectorSize** — Input projector size

positive integer

This property is read-only.

Input projector size, specified as a positive integer.

The LSTM layer operation uses four matrix multiplications of the form  $Wx_t$ , where  $W$  denotes the input weights and  $x_t$  denotes the layer input at time step  $t$ .

The LSTM projected layer operation instead uses multiplications of the form  $WQ_iQ_i^T x_t$ , where  $Q_i$  is an `InputSize`-by-`InputProjectorSize` matrix known as the *input projector*. The layer uses the same projector  $Q_i$  for each of the four multiplications.

To perform the four multiplications of the form  $Wx_t$ , an LSTM layer stores four weight matrices  $W$ , which necessitates storing  $4*\text{NumHiddenUnits}*InputSize$  learnable parameters. By instead storing the  $4*\text{NumHiddenUnits}$ -by-`InputProjectorSize` matrix  $W' = WQ_i$  and  $Q_i$ , an LSTM projected layer can perform the multiplication  $WQ_iQ_i^T x_t$  and store only  $(4*\text{NumHiddenUnits} + InputSize)*InputProjectorSize$  learnable parameters.

---

**Tip** To ensure that  $WQ_iQ_i^T x_t$  requires fewer learnable parameters, set the `InputProjectorSize` property to a value less than  $(4*\text{numHiddenUnits}*inputSize)/(4*\text{numHiddenUnits} + inputSize)$ .

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputMode — Output mode**

`'sequence'` (default) | `'last'`

This property is read-only.

Output mode, specified as one of these values:

- `'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)

This property is read-only.

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 the 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 the 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)

This property is read-only.

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 the name `'out'`, which corresponds to the output data.

If the `HasStateOutputs` property is 1 (true), then the layer has three outputs with the 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 that it computes.

### InputSize — Input size

'auto' (default) | positive integer

This property is read-only.

Input size, specified as a positive integer or 'auto'. If `InputSize` is 'auto', then the software automatically assigns the input size at training time.

Data Types: double | char

### Activations

#### StateActivationFunction — Activation function to update cell and hidden state

'tanh' (default) | 'softsign'

This property is read-only.

Activation function to update the cell and hidden state, specified as one of these values:

- '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  $\sigma_c$  in the calculations to update the cell and hidden state. For more information on how an LSTM layer uses activation functions, see “Long Short-Term Memory Layer” on page 1-1251.

#### GateActivationFunction — Activation function to apply to gates

'sigmoid' (default) | 'hard-sigmoid'

This property is read-only.

Activation function to apply to the gates, specified as one of these values:

- 'sigmoid' — Use the sigmoid function  $\sigma(x) = (1 + e^{-x})^{-1}$ .
- 'hard-sigmoid' — Use the hard sigmoid function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\ 1 & \text{if } x > 2.5 \end{cases}$$

The layer uses this option as the function  $\sigma_g$  in the calculations for the layer gates.

### State

#### CellState — Cell state

[] (default) | 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 you set this property manually, calls to the `resetState` function set the cell state to this value.

If `HasStateInputs` is 1 (true), then the `CellState` property must be empty.

Data Types: `single` | `double`

### **HiddenState — Hidden state**

`[]` (default) | 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 you set this property manually, calls to the `resetState` function set the hidden state to this value.

If `HasStateInputs` is 1 (true), then the `HiddenState` property must be empty.

Data Types: `single` | `double`

### **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 these values:

- 'glorot' — Initialize the input weights with the Glorot initializer [1] (also known as the Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and a variance of  $2 / (\text{InputProjectorSize} + \text{numOut})$ , where  $\text{numOut} = 4 * \text{NumHiddenUnits}$ .
- 'he' — Initialize the input weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and a variance of  $2 / \text{InputProjectorSize}$ .
- 'orthogonal' — Initialize the input weights with  $Q$ , the orthogonal matrix in 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 a standard deviation of 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 in 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 the Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and a variance of  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{OutputProjectorSize}$  and  $\text{numOut} = 4 * \text{NumHiddenUnits}$ .
- `'he'` — Initialize the recurrent weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and a variance of  $2/\text{OutputProjectorSize}$ .
- `'narrow-normal'` — Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and a standard deviation of 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`

### **InputProjectorInitializer — Function to initialize input projector**

`'orthogonal'` (default) | `'glorot'` | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the input projector, specified as one of the following:

- `'orthogonal'` — Initialize the input projector with  $Q$ , the orthogonal matrix in the QR decomposition of  $Z = QR$  for a random matrix  $Z$  sampled from a unit normal distribution [3].
- `'glorot'` — Initialize the input projector with the Glorot initializer [1] (also known as the Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and a variance of  $2/(\text{InputSize} + \text{inputProjectorSize})$ .
- `'he'` — Initialize the input projector with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and a variance of  $2/\text{InputSize}$ .
- `'narrow-normal'` — Initialize the input projector by independently sampling from a normal distribution with zero mean and a standard deviation of 0.01.
- `'zeros'` — Initialize the input weights with zeros.
- `'ones'` — Initialize the input weights with ones.
- Function handle — Initialize the input projector 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 projector.

The layer only initializes the input projector when the `InputProjector` property is empty.

Data Types: `char` | `string` | `function_handle`

### **OutputProjectorInitializer — Function to initialize output projector**

`'orthogonal'` (default) | `'glorot'` | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the output projector, specified as one of the following:

- `'orthogonal'` — Initialize the output projector with  $Q$ , the orthogonal matrix in the QR decomposition of  $Z = QR$  for a random matrix  $Z$  sampled from a unit normal distribution [3].

- 'glorot' — Initialize the output projector with the Glorot initializer [1] (also known as the Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and a variance of  $2/(\text{NumHiddenUnits} + \text{OutputProjectorSize})$ .
- 'he' — Initialize the output projector with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and a variance of  $2/\text{NumHiddenUnits}$ .
- 'narrow-normal' — Initialize the output projector by independently sampling from a normal distribution with zero mean and a standard deviation of 0.01.
- 'zeros' — Initialize the output projector with zeros.
- 'ones' — Initialize the output projector with ones.
- Function handle — Initialize the output projector 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 output projector.

The layer only initializes the output projector when the `OutputProjector` 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 these values:

- '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 a standard deviation of 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 layer vertically concatenates the four matrices in this order:

- 1 Input gate
- 2 Forget gate
- 3 Cell candidate
- 4 Output gate

The input weights are learnable parameters. When you train a neural network using the `trainNetwork` function, if `InputWeights` is nonempty, then the software uses the `InputWeights` property as the initial value. If `InputWeights` is empty, then the software uses the initializer specified by `InputWeightsInitializer`.

At training time, `InputWeights` is a  $4 \times \text{NumHiddenUnits}$ -by-`InputProjectorSize` 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 layer vertically concatenates the four matrices in this order:

- 1 Input gate
- 2 Forget gate
- 3 Cell candidate
- 4 Output gate

The recurrent weights are learnable parameters. When you train an RNN using the `trainNetwork` function, if `RecurrentWeights` is nonempty, then the software uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then the software uses the initializer specified by `RecurrentWeightsInitializer`.

At training time, `RecurrentWeights` is a  $4 \times \text{NumHiddenUnits}$ -by-`OutputProjectorSize` matrix.

### **InputProjector — Input projector**

`[]` (default) | matrix

Input projector, specified as a matrix.

The input projector weights are learnable parameters. When you train a network using the `trainNetwork` function, if `InputProjector` is nonempty, then the software uses the `InputProjector` property as the initial value. If `InputProjector` is empty, then the software uses the initializer specified by `InputProjectorInitializer`.

At training time, `InputProjector` is a `InputSize`-by-`InputProjectorSize` matrix.

Data Types: `single` | `double`

### **OutputProjector — Output projector**

`[]` (default) | matrix

Output projector, specified as a matrix.

The output projector weights are learnable parameters. When you train a network using the `trainNetwork` function, if `OutputProjector` is nonempty, then the software uses the `OutputProjector` property as the initial value. If `OutputProjector` is empty, then the software uses the initializer specified by `OutputProjectorInitializer`.

At training time, `OutputProjector` is a `NumHiddenUnits`-by-`OutputProjectorSize` matrix.

Data Types: `single` | `double`

### **Bias — Layer biases**

`[]` (default) | numeric vector

Layer biases, specified as a numeric vector.

The bias vector is a concatenation of the four bias vectors for the components (gates) in the layer. The layer vertically concatenates the four vectors in this order:

- 1 Input gate
- 2 Forget gate
- 3 Cell candidate
- 4 Output gate

The layer biases are learnable parameters. When you train a neural 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  $4 \times \text{NumHiddenUnits}$ -by-1 numeric vector.

### Learning Rate and Regularization

#### **InputWeightsLearnRateFactor** — Learning rate factor for input weights

1 (default) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the input weights, 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 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 you specify 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 these components:

- 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) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the recurrent weights, 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 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 you specify using 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 these components:

- 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]

### **InputProjectorLearnRateFactor — Learning rate factor for input projector**

1 (default) | nonnegative scalar

Learning rate factor for the input projector, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input projector of the layer. For example, if `InputProjectorLearnRateFactor` is 2, then the learning rate factor for the input projector of 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.

### **OutputProjectorLearnRateFactor — Learning rate factor for output projector**

1 (default) | nonnegative scalar

Learning rate factor for the output projector, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the output projector of the layer. For example, if `OutputProjectorLearnRateFactor` is 2, then the learning rate factor for the output projector of 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.

### **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 these components:

- 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 —  $L_2$  regularization factor for input weights**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  regularization factor for the input weights, 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 factor for the input weights of the layer. For example, if `InputWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the input weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual matrices in `InputWeights`, specify a 1-by-4 vector. The entries of `InputWeightsL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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 —  $L_2$  regularization factor for recurrent weights**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  regularization factor for the recurrent weights, 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 factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the  $L_2$  regularization factor for the recurrent weights of the layer is twice the current global  $L_2$  regularization factor. The software determines the  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual matrices in `RecurrentWeights`, specify a 1-by-4 vector. The entries of `RecurrentWeightsL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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]

**InputProjectorL2Factor —  $L_2$  regularization factor for input projector**

1 (default) | nonnegative scalar

$L_2$  regularization factor for the input projector, specified as a nonnegative scalar.

The software multiplies this factor by the global  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the input projector of the layer. For example, if `InputProjectorL2Factor` is 2, then the  $L_2$  regularization factor for the input projector of the layer is twice the current global  $L_2$  regularization factor. The software determines the global  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

**OutputProjectorL2Factor —  $L_2$  regularization factor for output projector**

1 (default) | nonnegative scalar

$L_2$  regularization factor for the output projector, specified as a nonnegative scalar.

The software multiplies this factor by the global  $L_2$  regularization factor to determine the  $L_2$  regularization factor for the output projector of the layer. For example, if `OutputProjectorL2Factor` is 2, then the  $L_2$  regularization factor for the output projector of the layer is twice the current global  $L_2$  regularization factor. The software determines the global  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

**BiasL2Factor —  $L_2$  regularization factor for biases**

0 (default) | nonnegative scalar | 1-by-4 numeric vector

$L_2$  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. The software determines the global  $L_2$  regularization factor based on the settings you specify using the `trainingOptions` function.

To control the value of the  $L_2$  regularization factor for the four individual vectors in `Bias`, specify a 1-by-4 vector. The entries of `BiasL2Factor` correspond to the  $L_2$  regularization factor of these components:

- 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 the name `''`.

Data Types: `char` | `string`

### NumInputs — Number of inputs

1 | 3

This property is read-only.

Number of inputs to the layer.

If the `HasStateInputs` property is `0` (`false`), then the layer has one input with the 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 the 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'}`

This property is read-only.

Input names of the layer.

If the `HasStateInputs` property is `0` (`false`), then the layer has one input with the 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 the 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

This property is read-only.

Number of outputs to the layer.

If the `HasStateOutputs` property is `0` (`false`), then the layer has one output with the name `'out'`, which corresponds to the output data.



If the `HasStateOutputs` property is 1 (true), then the layer has three outputs with the 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 that it computes.

Data Types: double

### OutputNames — Output names

```
{'out'} | {'out', 'hidden', 'cell'}
```

This property is read-only.

Output names of the layer.

If the `HasStateOutputs` property is 0 (false), then the layer has one output with the name 'out', which corresponds to the output data.

If the `HasStateOutputs` property is 1 (true), then the layer has three outputs with the 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 that it computes.

## Examples

### Create LSTM Projected Layer

Create an LSTM projected layer with 100 hidden units, an output projector size of 30, an input projector size of 16, and the name "lstm".

```
layer = lstmProjectedLayer(100,30,16,Name="lstm")
```

```
layer =
```

```
  LSTMProjectedLayer with properties:
```

```

        Name: 'lstm'
    InputNames: {'in'}
    OutputNames: {'out'}
        NumInputs: 1
        NumOutputs: 1
    HasStateInputs: 0
    HasStateOutputs: 0
```

```
Hyperparameters
```

```

        InputSize: 'auto'
        NumHiddenUnits: 100
    InputProjectorSize: 16
    OutputProjectorSize: 30
        OutputMode: 'sequence'
    StateActivationFunction: 'tanh'
    GateActivationFunction: 'sigmoid'
```

```
Learnable Parameters
```

```

        InputWeights: []
    RecurrentWeights: []
            Bias: []
        InputProjector: []
    OutputProjector: []
```

```
State Parameters
  HiddenState: []
  CellState: []
```

Show all properties

Include an LSTM projected layer in a layer array.

```
inputSize = 12;
numHiddenUnits = 100;
outputProjectorSize = max(1,floor(0.75*numHiddenUnits));
inputProjectorSize = max(1,floor(0.25*inputSize));

layers = [
    sequenceInputLayer(inputSize)
    lstmProjectedLayer(numHiddenUnits,outputProjectorSize,inputProjectorSize)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

### Compare Network Projection Sizes

Compare the sizes of networks that do and do not contain projected layers.

Define an LSTM network architecture. Specify the input size as 12, which corresponds to the number of features of the input data. Configure an LSTM layer with 100 hidden units that outputs 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;

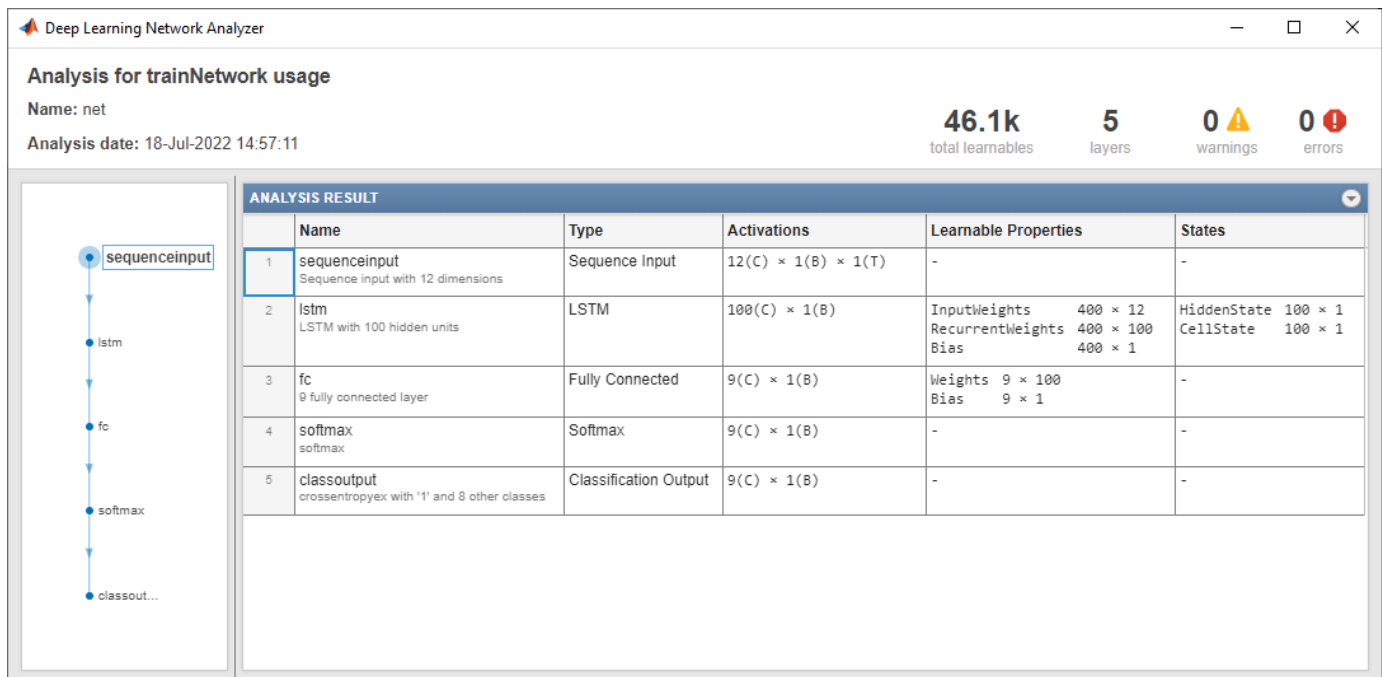
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,OutputMode="last")
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
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

Analyze the network using the `analyzeNetwork` function. The network has approximately 46,100 learnable parameters.

```
analyzeNetwork(layers)
```



Create an identical network with an LSTM projected layer in place of the LSTM layer.

For the LSTM projected layer:

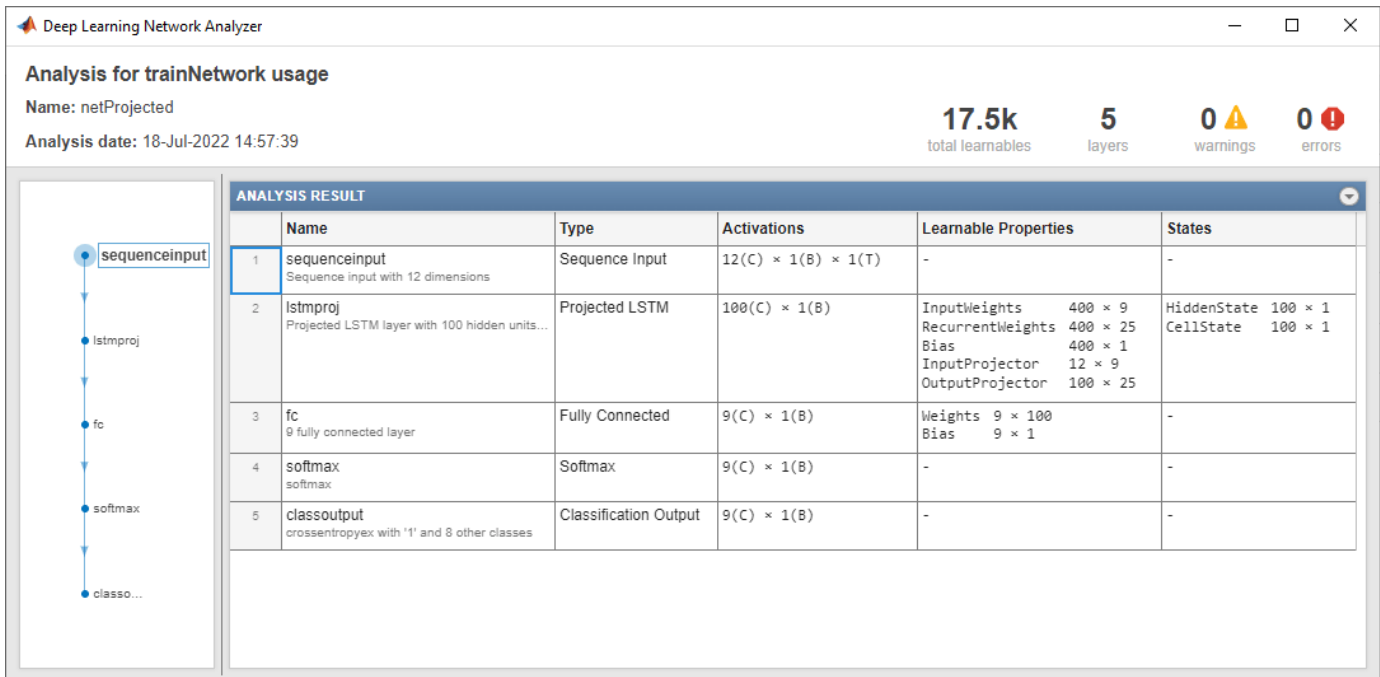
- Specify the same number of hidden units as the LSTM layer
- Specify an output projector size of 25% of the number of hidden units.
- Specify an input projector size of 75% of the input size.
- Ensure that the output and input projector sizes are positive by taking the maximum of the sizes and 1.

```
outputProjectorSize = max(1,floor(0.25*numHiddenUnits));
inputProjectorSize = max(1,floor(0.75*inputSize));
```

```
layersProjected = [ ...
    sequenceInputLayer(inputSize)
    lstmProjectedLayer(numHiddenUnits,outputProjectorSize,inputProjectorSize,OutputMode="last")
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Analyze the network using the `analyzeNetwork` function. The network has approximately 17,500 learnable parameters, which is a reduction of more than half. The sizes of the learnable parameters of the layers following the projected layer have the same sizes as the network without the LSTM projected layer. Reducing the number of learnable parameters by projecting an LSTM layer rather than reducing the number of hidden units of the LSTM layer maintains the output size of the layer and, in turn, the sizes of the downstream layers, which can result in better prediction accuracy.

```
analyzeNetwork(layersProjected)
```



## Algorithms

### Long Short-Term Memory Layer

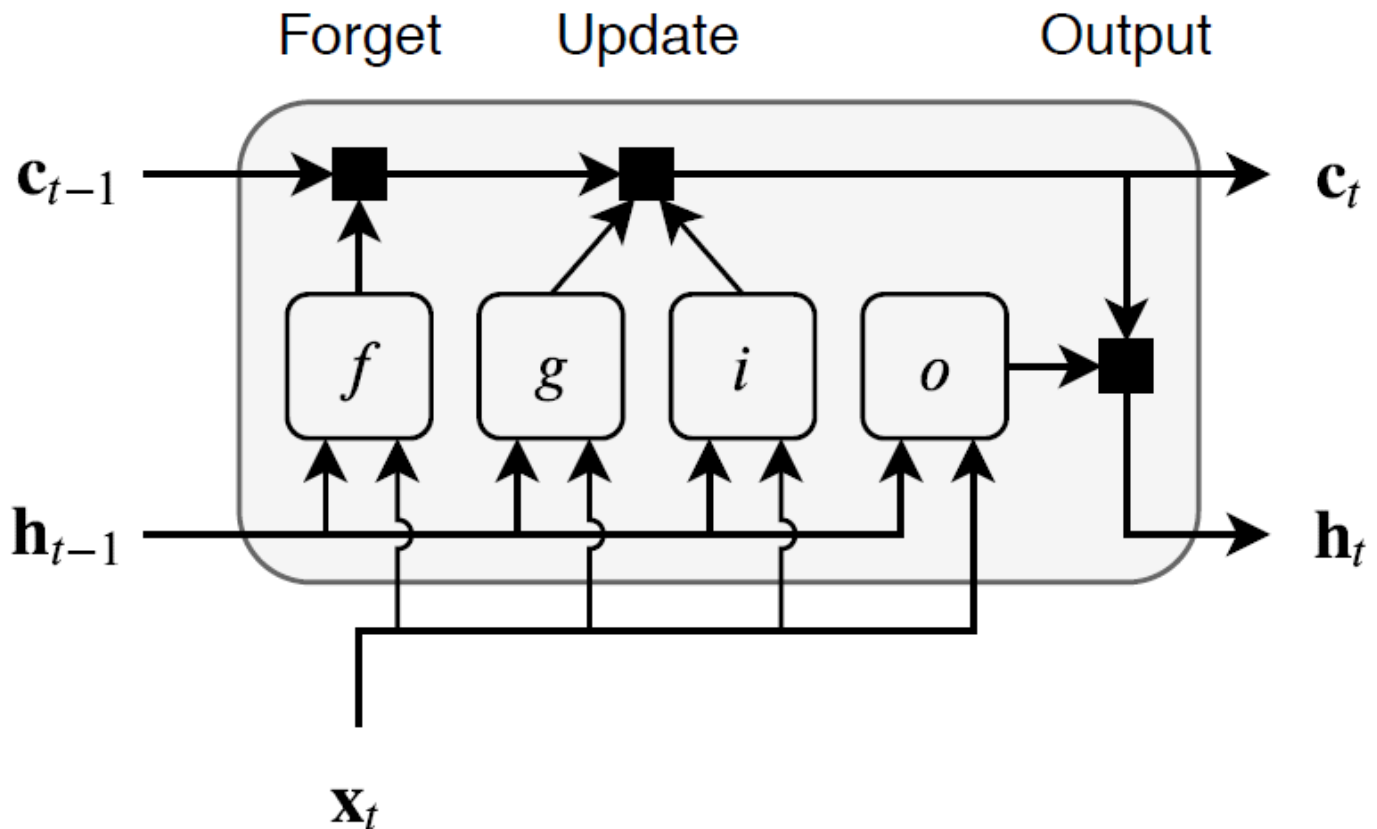
An LSTM layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data.

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*.

These 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  $t$ . This diagram shows 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. The layer concatenates the matrices according to these equations:

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

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  $\sigma_c$  denotes the state activation function. By default, the `LstmLayer` function uses the hyperbolic tangent function (`tanh`) to compute the state activation function.

These formulas describe the components at time step  $t$ .

Component	Formula
Input gate	$i_t = \sigma_g(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1} + b_i)$
Forget gate	$f_t = \sigma_g(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1} + b_f)$
Cell candidate	$g_t = \sigma_c(W_g \mathbf{x}_t + R_g \mathbf{h}_{t-1} + b_g)$
Output gate	$o_t = \sigma_g(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1} + b_o)$

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

### LSTM Projected Layer

An LSTM projected layer is an RNN layer that learns long-term dependencies between time steps in time series and sequence data using projected learnable weights.

To compress a deep learning network, you can use *projected layers*. A projected layer is a type of deep learning layer that enables compression by reducing the number of stored learnable parameters. The layer introduces learnable projector matrices  $Q$ , replaces multiplications of the form  $Wx$ , where  $W$  is a learnable matrix, with the multiplication  $WQQ^T x$ , and stores  $Q$  and  $W' = WQ$  instead of storing  $W$ . Projecting  $x$  into a lower-dimensional space using  $Q$  typically requires less memory to store the learnable parameters and can have similarly strong prediction accuracy.

Reducing the number of learnable parameters by projecting an LSTM layer rather than reducing the number of hidden units of the LSTM layer maintains the output size of the layer and, in turn, the sizes of the downstream layers, which can result in better prediction accuracy.

#### Output Projection

The LSTM layer operation uses four matrix multiplications of the form  $Rh_{t-1}$ , where  $R$  denotes the recurrent weights and  $h_t$  denotes the hidden state (or, equivalently, the layer output) at time step  $t$ .

The LSTM projected layer operation instead uses multiplications of the form  $RQ_oQ_o^T h_{t-1}$ , where  $Q_o$  is a `NumHiddenUnits-by-OutputProjectorSize` matrix known as the *output projector*. The layer uses the same projector  $Q_o$  for each of the four multiplications.

To perform the four multiplications of the form  $Rh_{t-1}$ , an LSTM layer stores four recurrent weights  $R$ , which necessitates storing  $4 * \text{NumHiddenUnits}^2$  learnable parameters. By instead storing the `4*NumHiddenUnits-by-OutputProjectorSize` matrix  $R' = RQ_o$  and  $Q_o$ , an LSTM projected layer can perform the multiplication  $RQ_oQ_o^T h_{t-1}$  and store only  $5 * \text{NumHiddenUnits} * \text{OutputProjectorSize}$  learnable parameters.

#### Input Projection

The LSTM layer operation uses four matrix multiplications of the form  $Wx_t$ , where  $W$  denotes the input weights and  $x_t$  denotes the layer input at time step  $t$ .

The LSTM projected layer operation instead uses multiplications of the form  $WQ_iQ_i^T x_t$ , where  $Q_i$  is an `InputSize-by-InputProjectorSize` matrix known as the *input projector*. The layer uses the same projector  $Q_i$  for each of the four multiplications.

To perform the four multiplications of the form  $Wx_t$ , an LSTM layer stores four weight matrices  $W$ , which necessitates storing  $4 * \text{NumHiddenUnits} * \text{InputSize}$  learnable parameters. By instead

storing the  $4 * \text{NumHiddenUnits} \text{-by-InputProjectorSize}$  matrix  $W' = WQ_i$  and  $Q_i$ , an LSTM projected layer can perform the multiplication  $WQ_i Q_i^T x_t$  and store only  $(4 * \text{NumHiddenUnits} + \text{InputSize}) * \text{InputProjectorSize}$  learnable parameters.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `LSTMProjectedLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	OutputMode	Output Format
"CB" (channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"CBT" (channel, batch, time)	"sequence"	"CBT" (channel, batch, time)
	"last"	"CB" (channel, batch)
"SB" (spatial, batch)	"sequence"	"CB" (channel, batch)
	"last"	

In `dlnetwork` objects, `LSTMProjectedLayer` objects also support these input and output format combinations.

Input Format	OutputMode	Output Format
"SCB" (spatial, channel, batch)	"sequence"	"CB" (channel, batch)
	"last"	
"SSCB" (spatial, spatial, channel, batch)	"sequence"	
	"last"	

Input Format	OutputMode	Output Format	
"SSSCB" (spatial, spatial, spatial, channel, batch)	"sequence"		
	"last"		
"SCBT" (spatial, channel, batch, time)	"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)	
"SC" (spatial, channel)	"sequence"	"CU" (channel, unspecified)	
	"last"		
"SSC" (spatial, spatial, channel)	"sequence"		
	"last"		
"SSSC" (spatial, spatial, spatial, channel)	"sequence"		
	"last"		
"CT" (channel, time)	"sequence"		"CT" (channel, time)
	"last"		"CU" (channel, unspecified)
"SCT" (spatial, channel, time)	"sequence"		"CT" (channel, time)
	"last"	"CU" (channel, unspecified)	
"SSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)	
	"last"	"CU" (channel, unspecified)	
"SSSCT" (spatial, spatial, channel, time)	"sequence"	"CT" (channel, time)	
	"last"	"CU" (channel, unspecified)	
"SSB" (spatial, spatial, batch)	"sequence"	"CB" (channel, batch)	
	"last"		
"SSSB" (spatial, spatial, spatial, batch)	"sequence"		
	"last"		
"BT" (batch, time)	"sequence"	"CBT" (channel, batch, time)	
	"last"	"CB" (channel, batch)	
"SBT" (spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)	
	"last"	"CB" (channel, batch)	
"SSBT" (spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)	
	"last"	"CB" (channel, batch)	
"SSSBT" (spatial, spatial, spatial, batch, time)	"sequence"	"CBT" (channel, batch, time)	
	"last"	"CB" (channel, batch)	

To use these input formats in `trainNetwork` workflows, convert the data to "CB" (channel, batch) or "CBT" (channel, batch, time) format using `flattenLayer`.



If the `HasStateInputs` property is 1 (true), then the layer has two additional inputs with the 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).

## Version History

Introduced in R2022b

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>
- [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:

LSTM projected layer objects support generic C and C++ code generation only.

## See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `convolution1dLayer` | `neuronPCA` | `compressNetworkUsingProjection`

## Topics

"Train Network with LSTM Projected Layer"  
 "Compress Neural Network Using Projection"  
 "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"  
 "Sequence-to-One Regression Using Deep Learning"  
 "Classify Videos Using Deep Learning"  
 "Long Short-Term Memory Neural Networks"

“List of Deep Learning Layers”  
“Deep Learning Tips and Tricks”

# maxpool

Pool data to maximum value

## Syntax

```
Y = maxpool(X,poolsize)
[Y,indx,inputSize] = maxpool(X,poolsize)
Y = maxpool(X,'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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` 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`
- 

`Y = maxpool(X,poolsize)` applies the maximum pooling operation to the formatted `darray` object `X`. 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 `Y` is a formatted `darray` with the same dimension format as `X`.

The function, by default, pools over up to three dimensions of `X` 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.

`[Y,indx,inputSize] = maxpool(X,poolsize)` also returns the linear indices of the maximum value within each pooled region and the size of the input feature map `X` for use with the `maxunpool` function.

`Y = maxpool(X,'global')` computes the global maximum over the spatial dimensions of the input `X`. This syntax is equivalent to setting `poolsize` in the previous syntaxes to the size of the 'S' dimensions of `X`.

`___ = maxpool( ___, 'DataFormat', FMT)` applies the maximum pooling operation to the unformatted `darray` object `X` with format specified by `FMT` using any of the previous syntaxes. The output `Y` is an unformatted `darray` object with dimensions in the same order as `X`. 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 `darray` 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);  
d1X = darray(X,'SSCB');
```

View the size and format of the input data.

```
size(d1X)
```

```
ans = 1×4
```

```
    28    28     3   128
```

```
dims(d1X)
```

```
ans =  
'SSCB'
```

Apply 2-D maximum pooling with 2-by-2 pooling windows using the `maxpool` function.

```
poolSize = [2 2];  
d1Y = maxpool(d1X,poolSize);
```

View the size and format of the output.

```
size(d1Y)
```

```
ans = 1×4
```

```
    27    27     3   128
```

```
dims(d1Y)
```

```
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

### X — Input data

`dlarray`

Input data, specified as a formatted or unformatted `dlarray` object.

If X 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 X 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 `darray` object, each character provides a label for each dimension of the data and must be one of these options:

- "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 `darray`.

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 X 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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

### Y — Pooled data

darray

Pooled data, returned as a darray with the same underlying data type as X.

If the input data X is a formatted darray, then Y has the same format as X. If the input data is not a formatted darray, then Y is an unformatted darray with the same dimension order as the input data.

### indx — Indices of maximum values

darray

Indices of maximum values in each pooled region, returned as a darray. Each value in indx represents the location of the corresponding maximum value in Y, given as a linear index of the values in X.

If X is a formatted darray, indx has the same size and format as the output Y.

If X is not a formatted darray, indx is an unformatted darray. 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 Y when Y 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 “2-D Max Pooling Layer” on page 1-1303 section of the maxPooling2dLayer reference page.

## Version History

Introduced in R2019b

**R2020a: 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 `darray` 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 `darray` 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:

```
[Y,indx,inputSize] = maxpool(Y,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 `X` is a `darray` 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` | `darray`

**Topics**

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Network Using Model Function”

“List of Functions with `darray` Support”

# 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (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(3,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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $\text{ceil}(\text{inputSize}/\text{Stride})$ , 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 the name `' '`.

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,MinLength=40)
    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	''	1-D 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	''	1-D 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 "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "T" (time) dimension.
- For 1-D image input (data with three dimensions corresponding to the "S" (spatial), "C" (channel), and "B" (batch) dimensions), the layer pools over the "S" (spatial) dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the "S" (spatial), "C" (channel), "B" (batch), and "T" (time) dimensions), the layer pools over the "S" (spatial) dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel



- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `MaxPooling1dLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)
"SB" (spatial, batch)	"SB" (spatial, batch)

In `dlnetwork` objects, `MaxPooling1dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"BT" (batch, time)	"BT" (batch, time)
"SBT" (spatial, batch, time)	"SBT" (spatial, batch, time)

## Version History

Introduced in R2021b

### 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 Neural Networks”  
“List of Deep Learning Layers”  
“Deep Learning Tips and Tricks”

# 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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 the name ' '.

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 '' 2-D Convolution      20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
   3 '' ReLU                  ReLU
   4 '' 2-D 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 '' 2-D Convolution      20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
   3 '' ReLU                  ReLU

```



4	''	2-D 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

## Algorithms

### 2-D 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.

The dimensions that the layer pools over depends on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `MaxPooling2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSB" (spatial, spatial, batch)	"SSB" (spatial, spatial, batch)

In `dlnetwork` objects, `MaxPooling2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SBT" (spatial, batch, time)	"SBT" (spatial, batch, time)
"SSBT" (spatial, spatial, batch, time)	"SSBT" (spatial, spatial, batch, time)

## Version History

Introduced in R2016a

## 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

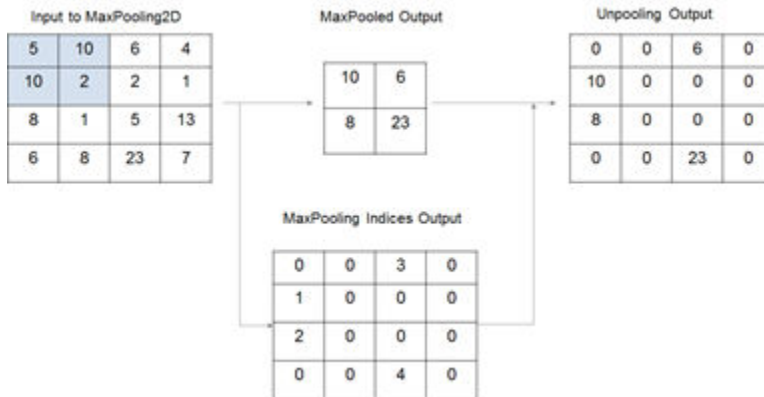
### C/C++ Code Generation

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

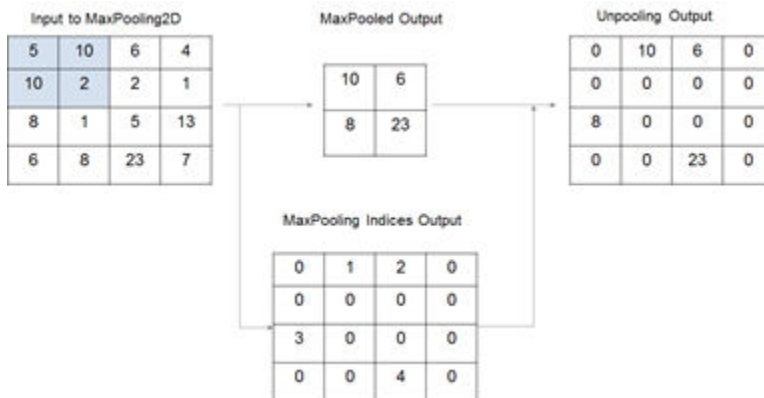
Usage notes and limitations:

If equal max values exists along the off-diagonal in a kernel window, implementation differences for the `maxPooling2dLayer` might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region.

For example, consider the following input to `maxPooling2dLayer`. The output from MATLAB is shown.



For the same input, the output from the generated code is shown.



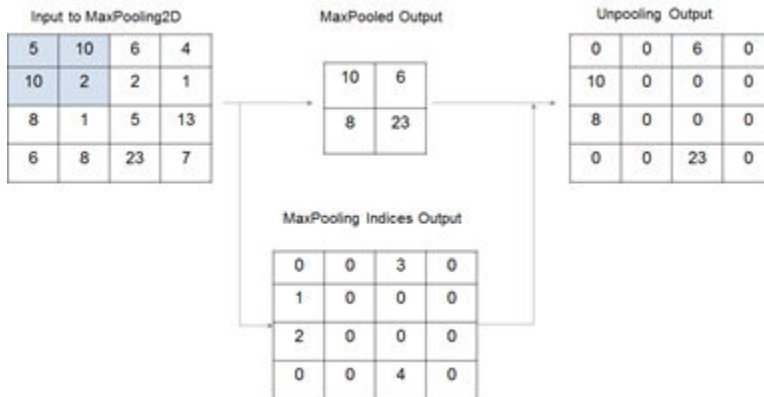
### GPU Code Generation

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

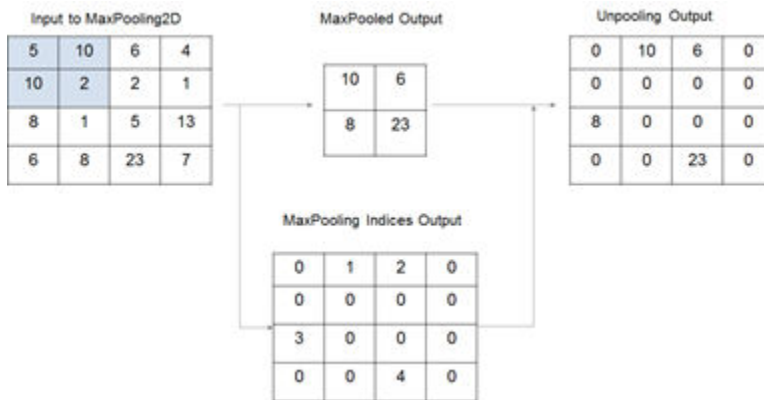
Usage notes and limitations:

If equal max values exists along the off-diagonal in a kernel window, implementation differences for the `maxPooling2dLayer` might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region.

For example, consider the following input to `maxPooling2dLayer`. The output from MATLAB is shown.



For the same input, the output from the generated code is shown.



### See Also

[averagePooling2dLayer](#) | [globalAveragePooling2dLayer](#) | [convolution2dLayer](#) | [maxUnpooling2dLayer](#)

### Topics

- “Create Simple Deep Learning Neural Network for Classification”
- “Train Convolutional Neural Network for Regression”
- “Deep Learning in MATLAB”
- “Specify Layers of Convolutional Neural Network”
- “List of Deep Learning Layers”

# 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.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

## 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

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.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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  $\text{ceil}(\text{inputSize}/\text{stride})$ , 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 the name ' '.

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  ''  3-D 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`.

## Algorithms

### 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 “2-D Max Pooling Layer” on page 1-1303 section of the `maxPooling2dLayer` reference page.

The dimensions that the layer pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer pools over the spatial and time dimensions.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `MaxPooling3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSB" (spatial, spatial, spatial, batch)	"SSSB" (spatial, spatial, spatial, batch)

In `dl` network objects, `MaxPooling3dLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)
"SSBT" (spatial, spatial, batch, time)	"SSBT" (spatial, spatial, batch, time)
"SSSBT" (spatial, spatial, spatial, batch, time)	"SSSBT" (spatial, spatial, spatial, batch, time)

## Version History

Introduced in R2019a

### 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"

## maxunpool

Unpool the output of a maximum pooling operation

### Syntax

```
Y = maxunpool(X,indx,outputSize)
Y = maxunpool(X,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 `darray` data. Using `darray` 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 `darray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `darray` object directly, or by using the `DataFormat` option.

---

**Note** To apply maximum unpooling within a `layerGraph` object or `Layer` array, use `maxUnpooling2dLayer`.

---

`Y = maxunpool(X,indx,outputSize)` upsamples the spatial or time dimensions of input data `X` to match the size `outputSize`. The data is padded with zeros between the locations of maximum values specified by `indx`. The input `X` is a formatted `darray` with dimension labels. The output `Y` is a formatted `darray` with the same dimension format as `X`.

`Y = maxunpool(X,indx,outputSize,'DataFormat',FMT)` also specifies the dimension format `FMT` when `X` is not a formatted `darray`. The output `Y` is an unformatted `darray` with the same dimension order as `X`.

### Examples

#### Unpool 2-D Maximum Pooled Data

Create a formatted `darray` 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);
dIX = darray(X,'SSCB');
```

View the size and format of the input data.

```
size(dIX)
```

```
ans = 1×4
      28    28     3   128
```

```
dims(d1X)
```

```
ans =
'SSCB'
```

Pool the data to maximum values over pooling regions of size 2 using a stride of 2.

```
[d1Y,indx,dataSize] = maxpool(d1X,2,'Stride',2);
```

View the size and format of the pooled data.

```
size(d1Y)
```

```
ans = 1×4
      14    14     3   128
```

```
dims(d1Y)
```

```
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.

```
d1Y = maxunpool(d1Y,indx,dataSize);
```

View the size and format of the unpooled data.

```
size(d1Y)
```

```
ans = 1×4
      28    28     3   128
```

```
dims(d1Y)
```

```
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

### X — Input data

`darray`

Input data, specified as a formatted or unformatted `darray` object.

If X is an unformatted `darray`, 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

`darray`

Indices of maximum values in each pooled region, specified as a `darray`.

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 `darray`.

Example: `'DataFormat', 'SSCB'`

Data Types: char | string

## Output Arguments

### **Y — Unpooled data**

`darray`

Unpooled data, returned as a `darray`. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, then Y has the same dimension format as X. If the input data is not a formatted `darray`, then Y is an unformatted `darray` with the same dimension order as the input data.

## Version History

**Introduced in R2019b**

## 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 X is a `darray` 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` | `darray` | `dlgradient` | `dlfeval`

### **Topics**

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Network Using Model Function”

“List of Functions with `darray` Support”



# 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 the name ' '.

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'    2-D Max Pooling    2x2 max pooling with stride [2  2] and padding [0  0  0
     2  'unpool'  2-D Max Unpooling    2-D 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:

        InputNames: {1x0 cell}
        OutputNames: {1x0 cell}
        Layers: [2x1 nnet.cnn.layer.Layer]
        Connections: [1x2 table]
```

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');
```

## Version History

**Introduced in R2017b**

### Extended Capabilities

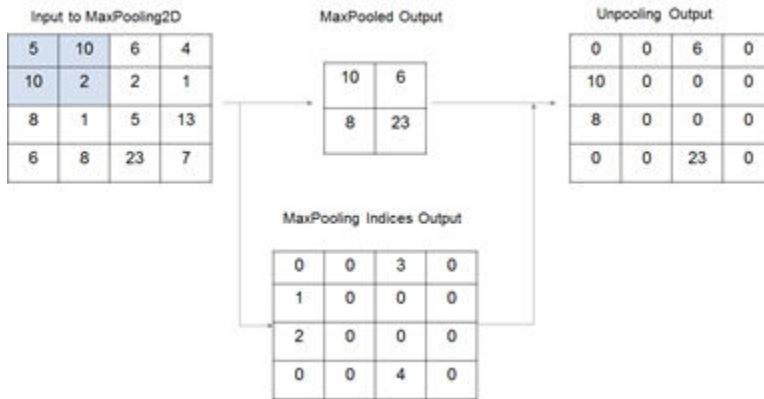
#### C/C++ Code Generation

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

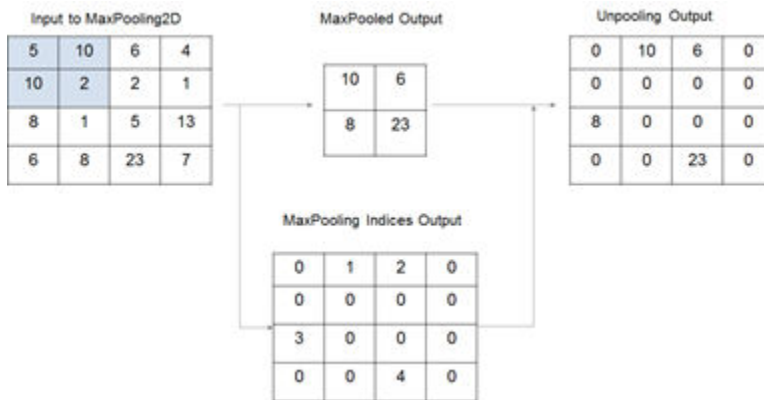
Usage notes and limitations:

If equal max values exists along the off-diagonal in a kernel window, implementation differences for the maxPooling2dLayer might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region.

For example, consider the following input to maxPooling2dLayer. The output from MATLAB is shown.



For the same input, the output from the generated code is shown.



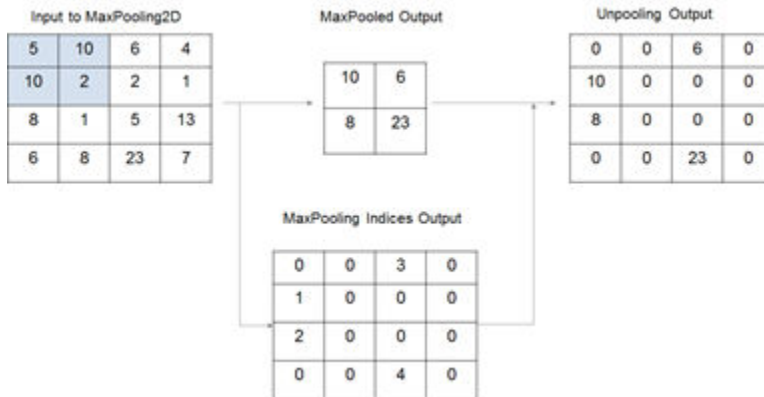
**GPU Code Generation**

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

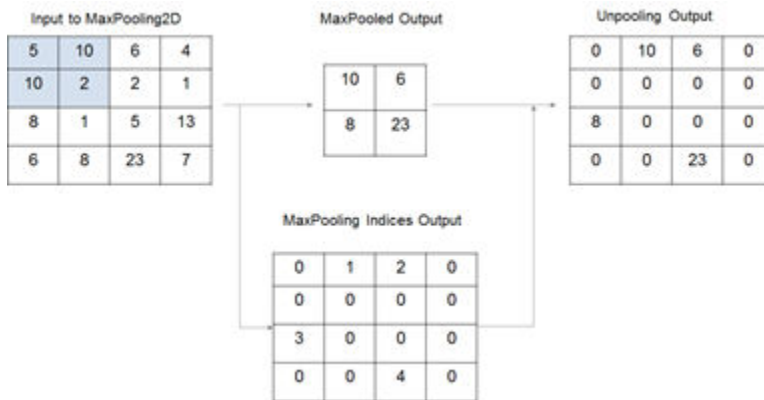
Usage notes and limitations:

If equal max values exists along the off-diagonal in a kernel window, implementation differences for the `maxPooling2dLayer` might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region.

For example, consider the following input to `maxPooling2dLayer`. The output from MATLAB is shown.



For the same input, the output from the generated code is shown.



## 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”

# 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.	<code>OutputCast</code>
Move data to GPU.	<code>OutputEnvironment</code>
Convert data to <code>darray</code> .	<code>OutputAsDarray</code>
Apply data format to <code>darray</code> variable.	<code>MiniBatchFormat</code>

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 subsettable or partitionable. To use this option, custom datastores should implement the `matlab.io.datastore.Subsettable` 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`, `OutputAsDarray`, 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 `darray` variables only. Non-`darray` mini-batch variables must have a `MiniBatchFormat` of `''`.

To avoid an error when you have a mix of `darray` and non-`darray` 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 Computing Requirements” (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**

<code>hasdata</code>	Determine if <code>minibatchqueue</code> can return mini-batch
<code>next</code>	Obtain next mini-batch of data from <code>minibatchqueue</code>
<code>partition</code>	Partition <code>minibatchqueue</code>
<code>reset</code>	Reset <code>minibatchqueue</code> to start of data
<code>shuffle</code>	Shuffle data in <code>minibatchqueue</code>

### **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
-----
{100x100 uint8}      0
{100x100 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 `darray` 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-`darray` 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 and Background Dispatch

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`.

- Set the number of outputs to 2, to match the number of outputs of the function.
- Set the mini-batch size.

- Preprocesses the 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.
- Apply the preprocessing function in the background using a parallel pool by setting the `DispatchInBackground` property to `true`. Setting `DispatchInBackground` to `true` requires Parallel Computing Toolbox™.
- Do not convert the mini-batch output variables to a `dlarray`.

```
mbq = minibatchqueue(imds,2,...  
    MiniBatchSize=16,...  
    MiniBatchFcn=@preprocessMiniBatch,...  
    DispatchInBackground=true,...  
    OutputAsDlarray=false)
```

```
mbq =  
minibatchqueue with 2 outputs and properties:
```

```
Mini-batch creation:  
    MiniBatchSize: 16  
    PartialMiniBatch: 'return'  
    MiniBatchFcn: @preprocessMiniBatch  
    DispatchInBackground: 1
```

```
Outputs:  
    OutputCast: {'single' 'single'}  
    OutputAsDlarray: [0 0]  
    MiniBatchFormat: {'' ''}  
    OutputEnvironment: {'auto' 'auto'}
```

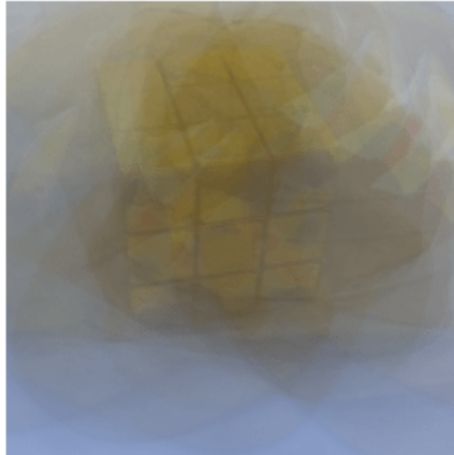
If you are using `DispatchInBackground` and a parallel pool is not already open, a local parallel pool will automatically open when data is read from the mini-batch queue. If your preprocessing function is supported for a thread-based environment, open a thread-based parallel pool for reduced memory usage, faster scheduling, and lower data transfer costs. For more information, see “Choose Between Thread-Based and Process-Based Environments” (Parallel Computing Toolbox).

```
parpool("Threads");
```

```
Starting parallel pool (parpool) using the 'Threads' profile ...  
Connected to parallel pool with 20 workers.
```

Obtain a mini-batch and display the average of the images in the mini-batch. A thread worker applies the preprocessing function.

```
[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],Mean=mean(XTrain,4))  
    convolution2dLayer(5,20)  
    reluLayer  
    convolution2dLayer(3,20,Padding=1)  
    reluLayer  
    convolution2dLayer(3,20,Padding=1)  
    reluLayer  
    fullyConnectedLayer(numClasses)  
    softmaxLayer];  
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
net = dlnetwork(lgraph);
```

### Define Model Loss Function

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

### 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 Computing Requirements” (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 average gradients and squared average gradients.

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

Calculate the total number of iterations for the training progress monitor.

```
numObservationsTrain = numel(YTrain);
numIterationsPerEpoch = ceil(numObservationsTrain / miniBatchSize);
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the TrainingProgressMonitor object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
if plots == "training-progress"
    monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
end
```

Train the network.

```
iteration = 0;
epoch = 0;
```

```
while epoch < numEpochs && ~monitor.Stop
    epoch = epoch + 1;

    % Shuffle data.
    shuffle (mbq);

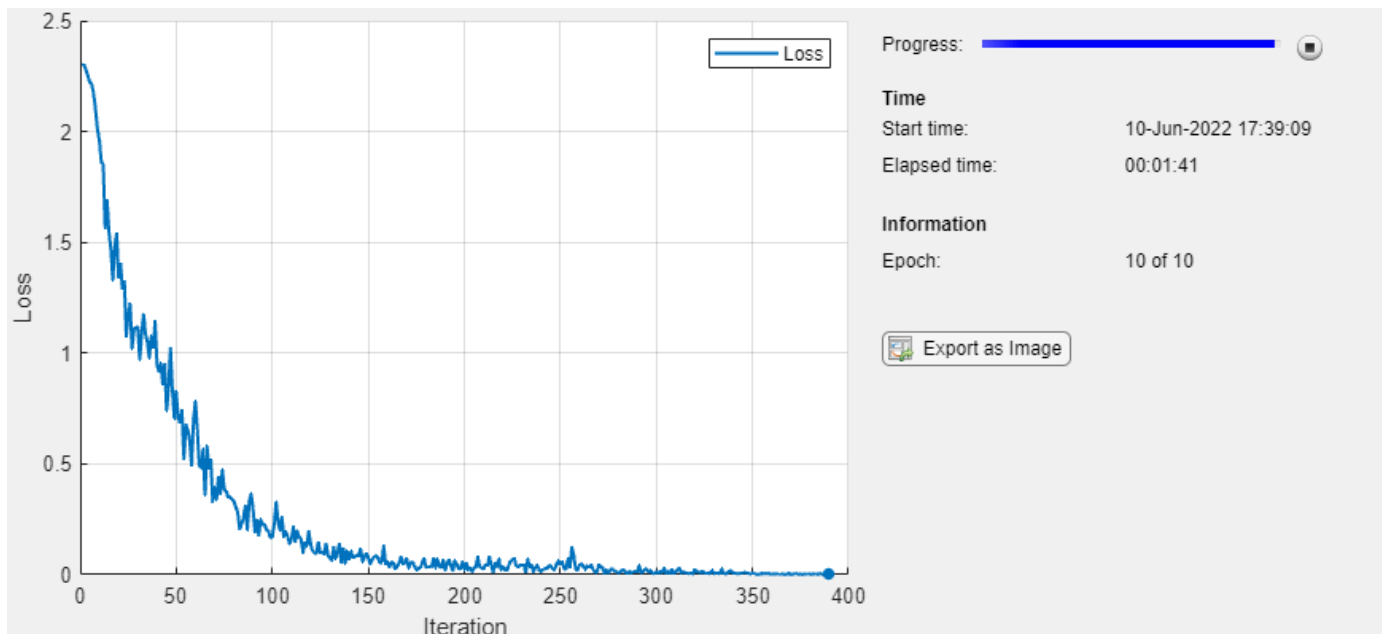
    while hasdata(mbq) && ~monitor.Stop
        iteration = iteration + 1;

        % Read mini-batch of data.
        [X,Y] = next(mbq);

        % Evaluate the model loss and gradients using dlfeval and the
        % modelLoss helper function.
        [loss,grad] = dlfeval(@modelLoss,net,X,Y);

        % Update the network parameters using the Adam optimizer.
        [net,averageGrad,averageSqGrad] = adamupdate(net,grad,averageGrad,averageSqGrad,iteration);

        % Update the training progress monitor.
        if plots == "training-progress"
            recordMetrics(monitor,iteration,Loss=loss);
            updateInfo(monitor,Epoch=epoch + " of " + numEpochs);
            monitor.Progress = 100 * iteration/numIterations;
        end
    end
end
end
```



### Model Loss Function

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

```
function [loss,gradients] = modelLoss(net,X,Y)
    YPred = forward(net,X);
    loss = crossentropy(YPred,Y);
    gradients = dlgradient(loss,net.Learnables);
```

```
end
```

### Mini-Batch Preprocessing Function

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

- 1 Extract the image data from the incoming cell array and concatenate 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
```

## Version History

Introduced in R2020b

### 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”

## 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 =
```

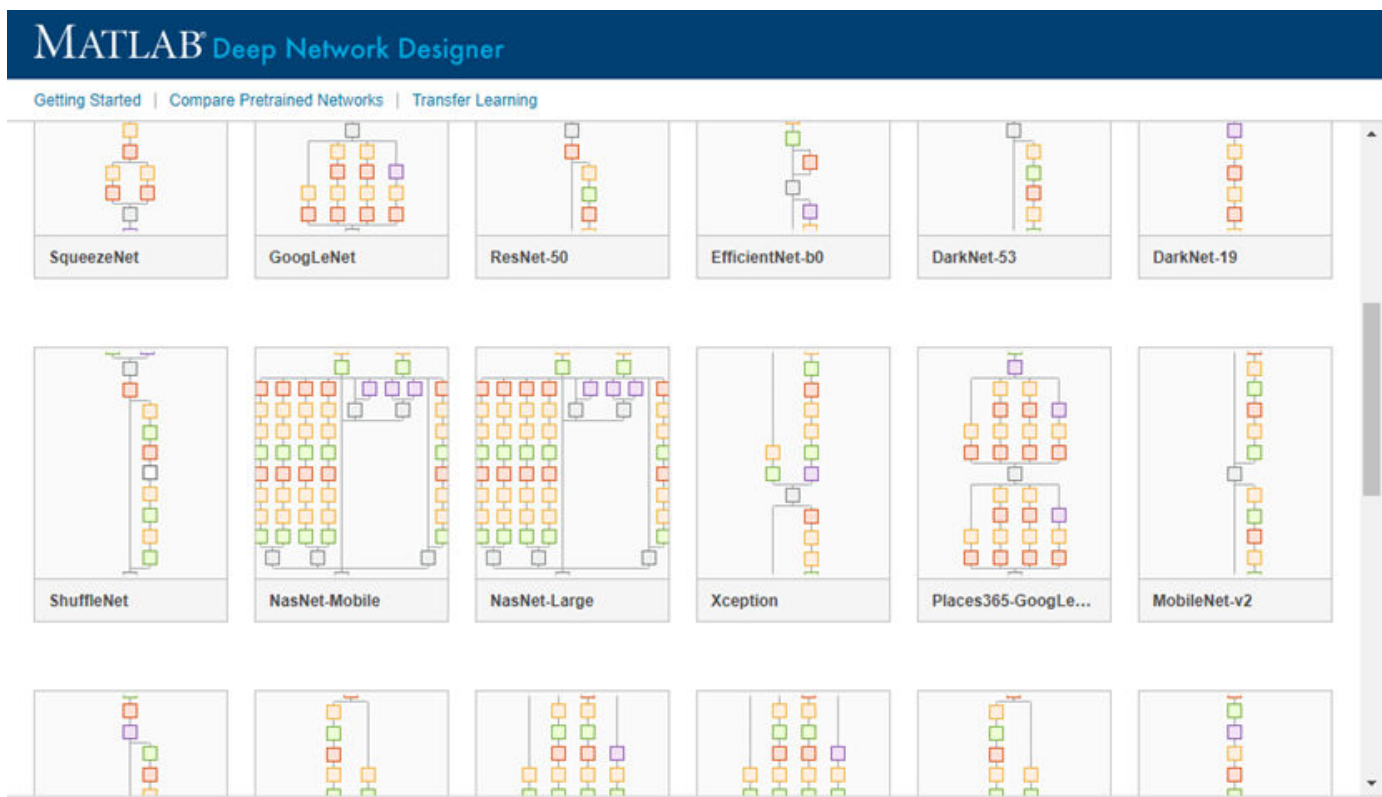
```
DAGNetwork with properties:
```

```
Layers: [155x1 nnet.cnn.layer.Layer]
Connections: [164x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(mobilenetv2)
```

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



If you need to download a neural network, pause on the desired neural 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.

## **Version History**

**Introduced in R2019a**

## **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”

## mse

Half mean squared error

### Syntax

```
loss = mse(Y,targets)
loss = mse(Y,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(Y,targets)` computes the half mean squared error loss between the predictions `Y` and the target values `targets` for regression problems. The input `Y` must be a formatted `dlarray`. The output `loss` is an unformatted `dlarray` scalar.

`loss = mse(Y,targets,'DataFormat',FMT)` also specifies the dimension format `FMT` when `Y` 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);
Y = darray(Y, 'SSCB')

```

Create the target values as a numeric array with the same dimension order as the input data `Y`.

```
targets = ones(height,width,channels,observations);
```

Compute the half mean squared error between the predictions and the targets.

```
loss = mse(Y,targets)
```

```
loss =
```

```
    1x1 darray
```

```
    5.2061
```

## Input Arguments

### Y — Predictions

`darray` | numeric array

Predictions, specified as a formatted `darray`, an unformatted `darray`, or a numeric array. When `Y` is not a formatted `darray`, you must specify the dimension format using the `DataFormat` option.

If `Y` is a numeric array, `targets` must be a `darray`.

### targets — Target responses

`darray` | numeric array

Target responses, specified as a formatted or unformatted `darray` or a numeric array.

The size of each dimension of `targets` must match the size of the corresponding dimension of `Y`.

If `targets` is a formatted `darray`, then its format must be the same as the format of `Y`, or the same as `DataFormat` if `Y` is unformatted.

If `targets` is an unformatted `darray` or a numeric array, then the function applies the format of `Y` or the value of `DataFormat` to `targets`.

---

**Tip** Formatted `darray` 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 `Y` and `targets` are consistent, when `Y` is a formatted `darray`, also specify `targets` as a formatted `darray`.

---

### 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 d`larray`.

Example: 'DataFormat', 'SSCB'

Data Types: char | string

## Output Arguments

### Loss — Half mean squared error loss

d`larray` scalar

Half mean squared error loss, returned as an unformatted d`larray` scalar. The output `loss` has the same underlying data type as the input `Y`.

## 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 1-1503 on the `RegressionOutputLayer` reference page.

## Version History

Introduced in R2019b

## 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 d`larray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `Y`
  - `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"](#)

# 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 the name ' '.

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.
```

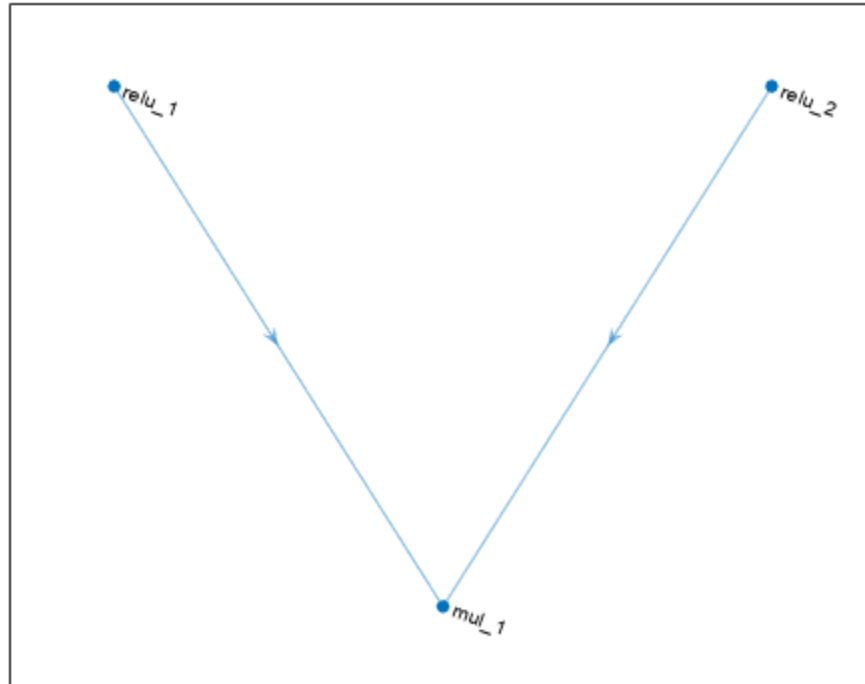
```
  Show all properties
```

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);
```



## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the

images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `MultiplicationLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Each input must have data of the same format.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)
"BU" (batch, unspecified)	"BU" (batch, unspecified)

In `dlnetwork` objects, `MultiplicationLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2020b

## **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”

# 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 =
```

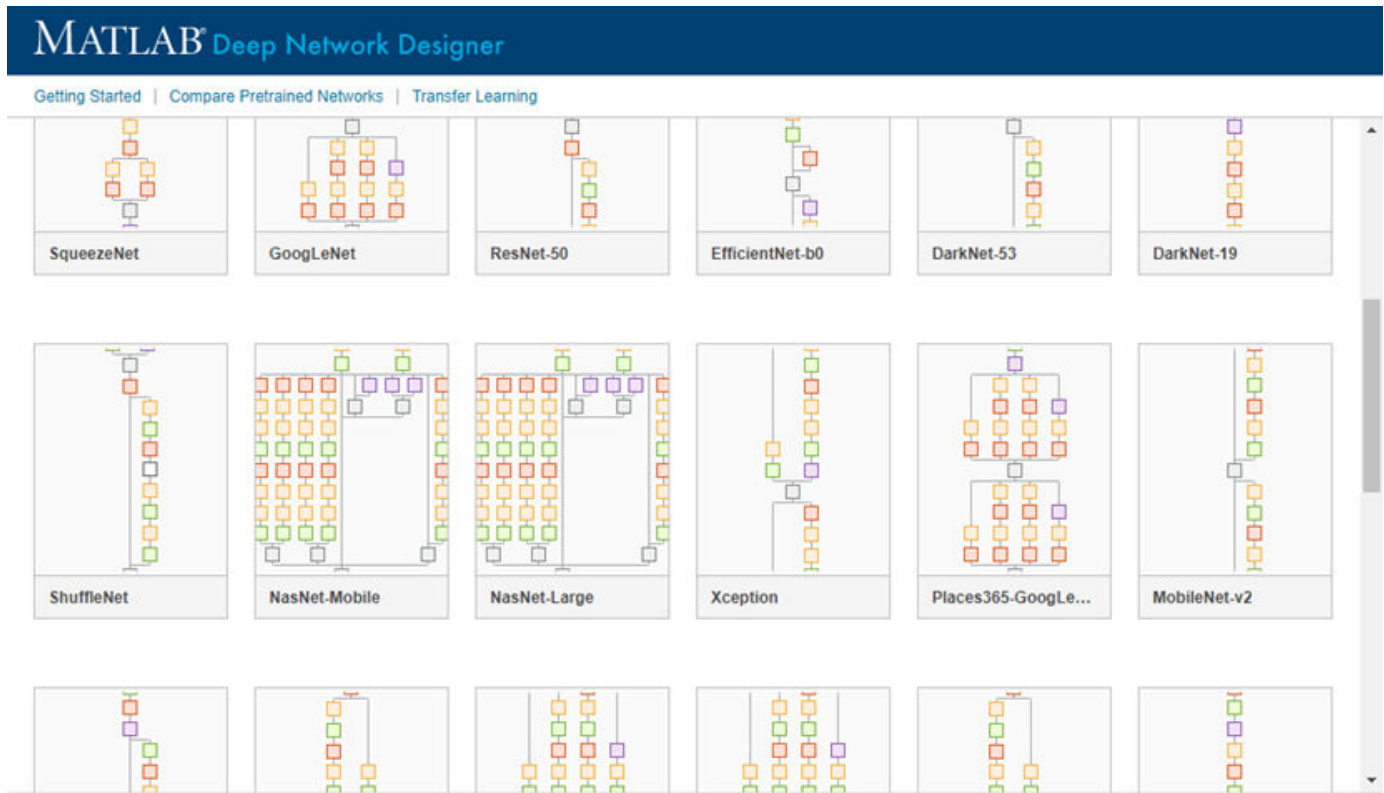
```
DAGNetwork with properties:
```

```
    Layers: [1244x1 nnet.cnn.layer.Layer]
Connections: [1463x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(nasnetLarge)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2019a

## 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"

# 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 =
```

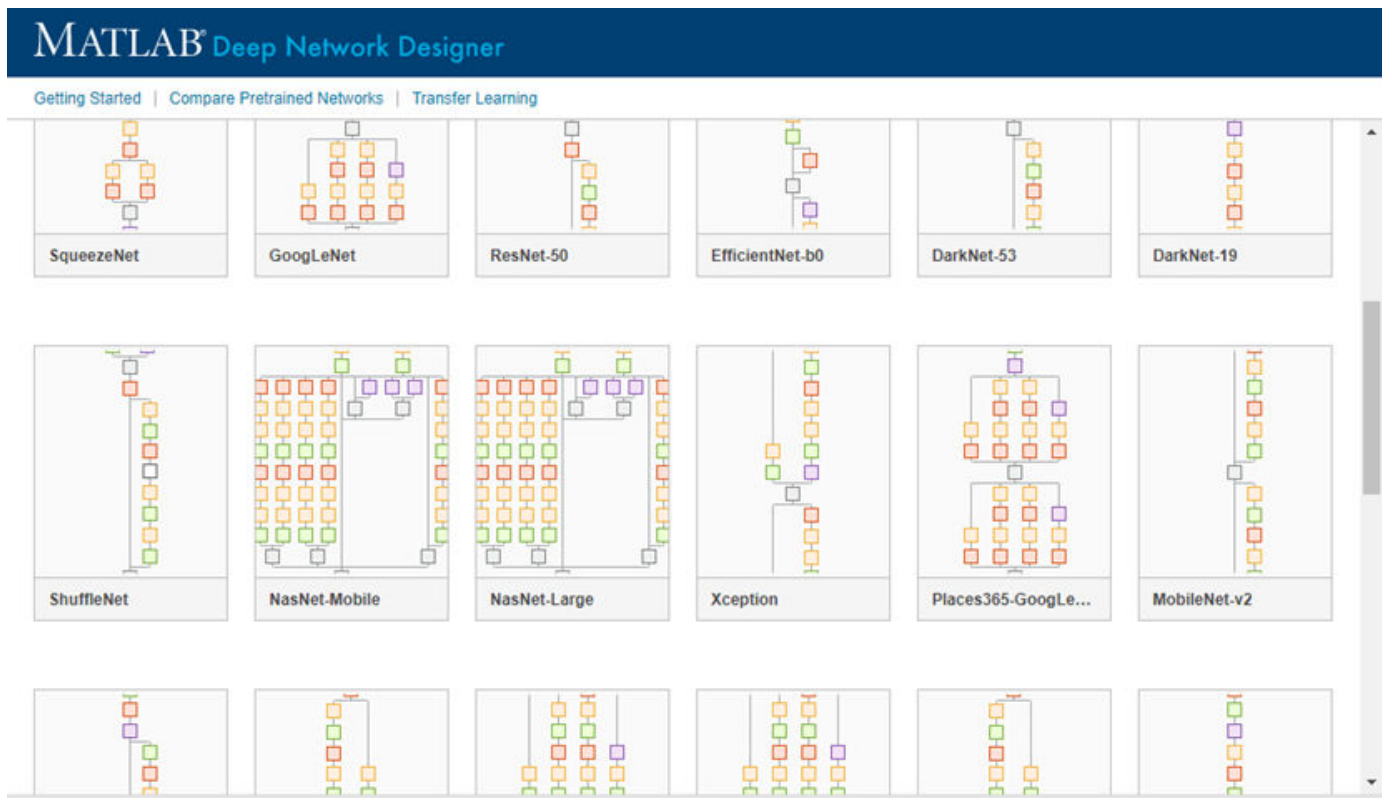
```
DAGNetwork with properties:
```

```
Layers: [914x1 nnet.cnn.layer.Layer]  
Connections: [1073x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(nasnetmobile)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2019a

## 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"

# networkDataLayout

Deep learning network data layout for learnable parameter initialization

## Description

Network data layout objects represent the data size and `dLarray` format information of input data for parameter initialization.

You can use `networkDataLayout` objects to initialize `dLnetwork` objects and custom layer learnable parameters using its size and format information as an alternative to using example data.

## Creation

### Syntax

```
layout = networkDataLayout(sz)
layout = networkDataLayout(sz,fmt)
```

### Description

`layout = networkDataLayout(sz)` creates an unformatted network data layout object and sets the `Size` property.

`layout = networkDataLayout(sz,fmt)` creates a formatted network data layout object and sets the `Size` and `Format` properties.

## Properties

### Size — Size

row vector

Size, specified as a row vector of two or more nonnegative integers or NaN values, where `sz(i)` denotes the size of dimension `i` and NaN values correspond to unknown dimension sizes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Format — Data format

' ' (default) | string scalar | character vector

Data format, specified as a string scalar or a character vector. Each character in the string must be one of the following dimension labels:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time

- "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.

Data Types: string | char

## Object Functions

`finddim` Find dimensions with specified label

## Examples

### Create Unformatted Network Data Layout Object

Create an unformatted `networkDataLayout` object representing data of size [28 28 1].

```
layout = networkDataLayout([28 28 1])  
  
layout =  
    networkDataLayout with properties:  
  
        Size: [28 28 1]  
        Format: ''
```

### Create Formatted Network Data Layout Object

Create a formatted `networkDataLayout` object representing a batch of 2-D RGB images of size [227 227], where the batch size is unknown.

```
layout = networkDataLayout([227 227 3 NaN], "SSCB")  
  
layout =  
    networkDataLayout with properties:  
  
        Size: [227 227 3 NaN]  
        Format: 'SSCB'
```

### Initialize Network using Network Data Layout Objects

Create an uninitialized `dlnetwork` object that has two unconnected inputs.

```
layers = [  
    convolution2dLayer(5,16,Name="conv")  
    batchNormalizationLayer  
    reluLayer  
    fullyConnectedLayer(50)  
    flattenLayer  
    concatenationLayer(1,2,Name="cat")
```



```
    fullyConnectedLayer(10)
    softmaxLayer];
```

```
net = dlnetwork(layers,Initialize=false);
```

View the input names of the network.

```
net.InputNames
```

```
ans = 1x2 cell
    {'conv'}    {'cat/in2'}
```

Create network data layout objects that represent input data for the inputs. For the first input, specify a batch of 28-by-28 grayscale images. For the second input specify a batch of single-channel feature data.

```
layout1 = networkDataLayout([28 28 1 NaN], "SSCB");
layout2 = networkDataLayout([1 NaN], "CB");
```

Initialize the network using the network data layout objects.

```
net = initialize(net,layout1,layout2)
```

```
net =
  dlnetwork with properties:
    Layers: [8x1 nnet.cnn.layer.Layer]
    Connections: [7x2 table]
    Learnables: [8x3 table]
    State: [2x3 table]
    InputNames: {'conv' 'cat/in2'}
    OutputNames: {'softmax'}
    Initialized: 1
```

View summary with summary.

## Find Dimensions of Network Data Layout Object

Create a formatted network data layout object representing 2-D image sequences. Specify the format "SSCBT" (spatial, spatial, channel, batch, time).

```
layout = networkDataLayout([227 227 3 NaN 100], "SSCBT");
```

Find the dimensions with the label "S".

```
dim = finddim(layout, "S")
```

```
dim = 1x2
    1    2
```

## **Version History**

**Introduced in R2022b**

### **See Also**

`dlarray` | `dlnetwork` | `finddim` | `initialize`

### **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”

# networkDistributionDiscriminator

Deep learning distribution discriminator

## Syntax

```
discriminator = networkDistributionDiscriminator(net,XID,XOOD,method)
discriminator = networkDistributionDiscriminator(net,XID,[],method)
discriminator = networkDistributionDiscriminator(net,[],XOOD,method)
discriminator = networkDistributionDiscriminator( ____,Name=Value)
```

## Description

`discriminator = networkDistributionDiscriminator(net,XID,XOOD,method)` returns a distribution discriminator using the method specified by `method`.

You can use the discriminator to classify observations as in-distribution (ID) and out-of-distribution (OOD). OOD data refers to data that is sufficiently different from the data you use to train the network, which can cause the network to behave unexpectedly. For more information, see “In-Distribution and Out-of-Distribution Data” on page 1-1369.

The `networkDistributionDiscriminator` function first finds distribution confidence scores using the method you specify in `method`. The function then finds a threshold that best separates the ID and OOD distribution confidence scores. You can classify any observation with a score below the threshold as OOD. For more information about how the function computes the threshold, see “Algorithms” on page 1-1372. You can find the threshold using a set of ID data, a set of OOD data, or both.

To determine whether new data is ID or OOD, pass `discriminator` as an input to the `isInNetworkDistribution` function.

To find the distribution confidence scores, pass `discriminator` as an input to the `distributionScores` function. For more information about distribution confidence scores, see “Distribution Confidence Scores” on page 1-1370.

`discriminator = networkDistributionDiscriminator(net,XID,[],method)` returns a distribution discriminator using only the ID data. The `Threshold` property of `discriminator` contains the threshold such that the discriminator attains a true positive rate greater than the value of the `TruePositiveGoal` name-value argument. For more information, see “Algorithms” on page 1-1372.

`discriminator = networkDistributionDiscriminator(net,[],XOOD,method)` returns a distribution discriminator using only the OOD data. The `Threshold` property of `discriminator` contains the threshold such that the discriminator attains a false positive rate less than the value of the `FalsePositiveGoal` name-value argument. For more information, see “Algorithms” on page 1-1372.

`discriminator = networkDistributionDiscriminator( ____,Name=Value)` returns a discriminator with additional options specified by one or more name-value arguments.

## Examples

### Create Distribution Discriminator Using In- and Out-of-Distribution Data

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data. Convert the data to a `dLarray` object.

```
XID = digitTrain4DArrayData;  
XID = dLarray(XID, "SSCB");
```

Modify the ID training data to create an OOD set.

```
XOOD = XID*0.3 + 0.1;
```

Create a discriminator. The function finds the threshold that best separates the two distributions of data by maximizing the true positive rate and minimizing the false positive rate.

```
method = "baseline";  
discriminator = networkDistributionDiscriminator(net,XID,XOOD,method)
```

```
discriminator =  
  BaselineDistributionDiscriminator with properties:  
  
    Method: "baseline"  
   Network: [1x1 dlnetwork]  
  Threshold: 0.9743
```

### Create Distribution Discriminator Using Only In-Distribution Data

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data. Convert the data to a `dLarray` object.

```
XID = digitTrain4DArrayData;  
XID = dLarray(XID, "SSCB");
```

Create a discriminator using only the ID data. Set the method to "energy" with a temperature of 10. Specify the true positive goal as 0.975. To specify the true positive goal, you must specify ID data and not specify the false positive goal.

```
method = "energy";  
discriminator = networkDistributionDiscriminator(net,XID,[],method, ...  
    Temperature=10, ...  
    TruePositiveGoal=0.975)
```

```
discriminator =  
  EnergyDistributionDiscriminator with properties:  
  
    Method: "energy"
```

```

    Network: [1x1 dlnetwork]
    Temperature: 10
    Threshold: 23.5541

```

### Create Distribution Discriminator Using Only Out-of-Distribution Data

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Create OOD data. In this example, the OOD data is 1000 images of random noise. Each image is 28-by-28 pixels, the same size as the input to the network. Convert the data to a `darray` object.

```
X00D = rand([28 28 1 1000]);
X00D = darray(X00D,"SSCB");
```

Create a discriminator. Specify the false positive goal as 0.025. To specify the false positive goal, you must specify OOD data and not specify the true positive goal.

```
method = "baseline";
discriminator = networkDistributionDiscriminator(net,[],X00D,method, ...
    FalsePositiveGoal=0.025)
```

```
discriminator =
    BaselineDistributionDiscriminator with properties:

        Method: "baseline"
        Network: [1x1 dlnetwork]
        Threshold: 0.9998
```

### Create Distribution Discriminator for Regression Network

Load a pretrained regression network.

```
load("digitsRegressionMLPNet.mat")
```

Load ID data. Convert the data to a `darray` object.

```
XID = digitTrain4DArrayData;
XID = darray(XID,"SSCB");
```

Modify the ID training data to create an OOD set.

```
X00D = XID*0.3 + 0.1;
```

Create a discriminator. For regression tasks, set the method to "hbos". When using the HBOS method, you can specify additional options. Set the variance cutoff to 0.0001 and use the penultimate layer to compute the HBOS distribution scores.

```
method = "hbos";
discriminator = networkDistributionDiscriminator(net,XID,X00D,method, ...
```

```
VarianceCutoff=0.0001, ...
LayerNames="relu_2")

discriminator =
  HBOSDistributionDiscriminator with properties:

    Method: "hbos"
    Network: [1x1 dlnetwork]
    LayerNames: "relu_2"
    VarianceCutoff: 1.0000e-04
    Threshold: -54.0601
```

### Assess Performance of Distribution Discriminator

Load a pretrained classification network.

```
load('digitsClassificationMLPNet.mat');
```

Load ID data. Convert the data to a `darray` object and count the number of observations.

```
XID = digitTrain4DArrayData;
XID = darray(XID,"SSCB");
numObservations = size(XID,4);
```

Modify the ID training data to create an OOD set.

```
X00D = XID*0.3 + 0.1;
```

Create a discriminator. The function finds the threshold that best separates the two distributions of data.

```
method = "baseline";
discriminator = networkDistributionDiscriminator(net,XID,X00D,method);
```

Test the discriminator on the ID and OOD data using the `isInNetworkDistribution` function. The `isInNetworkDistribution` function returns a logical array indicating which observations are ID and which observations are OOD.

```
tfID = isInNetworkDistribution(discriminator,XID);
tf00D = isInNetworkDistribution(discriminator,X00D);
```

Calculate the accuracy for the ID and OOD observations.

```
accuracyID = sum(tfID == 1)/numel(tfID)
accuracyID = 0.9856
accuracy00D = sum(tf00D == 0)/numel(tf00D)
accuracy00D = 0.9402
```

Plot the confusion chart for the ID and OOD data.

```
trueDataLabels = [
    repmat("In-Distribution",numObservations), ...
```

```

    repelem("Out-of-Distribution",numObservations)];

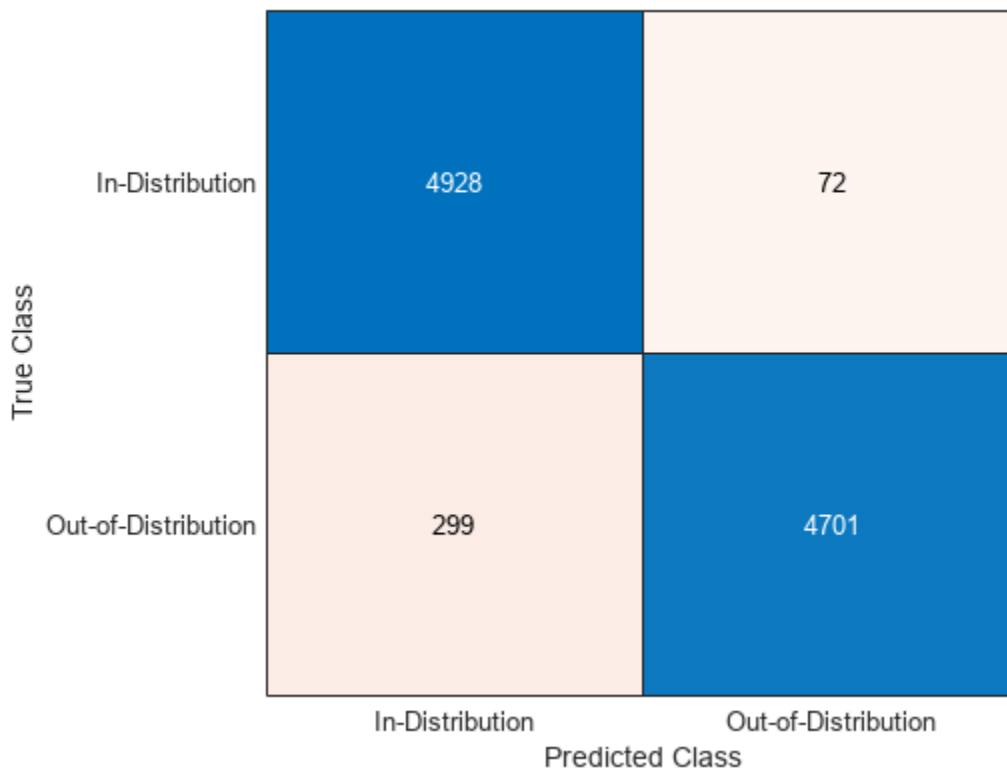
predDataLabelsID = repelem("Out-of-Distribution",numObservations);
predDataLabelsID(tfID == 1) = "In-Distribution";

predDataLabelsOOD = repelem("Out-of-Distribution",numObservations);
predDataLabelsOOD(tfOOD == 1) = "In-Distribution";

predDataLabels = [predDataLabelsID,predDataLabelsOOD];

confusionchart(trueDataLabels,predDataLabels)

```



Use the `distributionScores` function to find the distribution scores for the ID and OOD data.

```

scoresID = distributionScores(discriminator,XID);
scoresOOD = distributionScores(discriminator,XOOD);
scores = [scoresID',scoresOOD'];

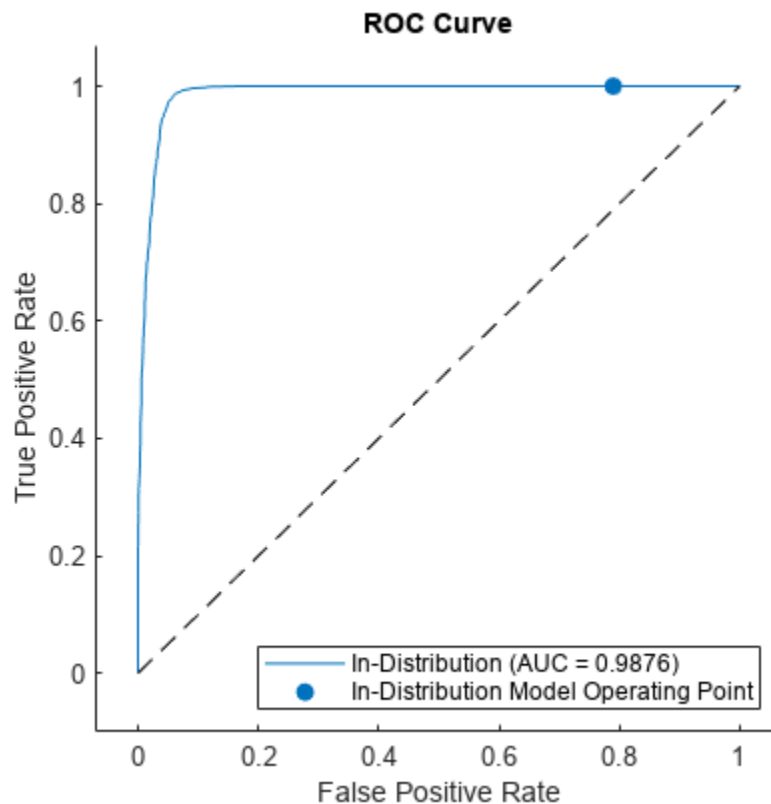
```

Use `rocmetrics` to plot a ROC curve to show how well the model is at separating the data into ID and OOD.

```

rocObj = rocmetrics(trueDataLabels,scores,"In-Distribution");
plot(rocObj)

```



## Input Arguments

### **net** — Neural network

`dlnetwork` object

Neural network, specified as a `dlnetwork` object.

### **XID** — In-distribution data

formatted `dlarray` object | []

In-distribution (ID) data, specified as a formatted `dlarray` object or []. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

At least one of the `XID` and `XOOD` input arguments must be nonempty.

For more information about ID and OOD data, see “In-Distribution and Out-of-Distribution Data” on page 1-1369.

### **XOOD** — Out-of-distribution data

formatted `dlarray` object | []

Out-of-distribution (OOD) data, specified as a formatted `dlarray` object or []. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

At least one of the `XID` and `XOOD` input arguments must be nonempty.



For more information about ID and OOD data, see “In-Distribution and Out-of-Distribution Data” on page 1-1369.

### method — Method for computing distribution confidence scores

"baseline" | "odin" | "energy" | "hbos"

Method for computing the distribution confidence scores, specified as one of these values:

- "baseline" — Use maximum softmax activations as the distribution confidence scores [1]. This method creates a `BaselineDistributionDiscriminator` object.
- "odin" — Use rescaled softmax activations as distribution confidence scores (also known as the ODIN method) [2]. Set the scale by specifying the `Temperature` name-value argument. This method creates an `ODINDistributionDiscriminator` object.
- "energy" — Use scaled, energy-based distribution confidence scores [3]. Set the scale by specifying the `Temperature` name-value argument. This method creates an `EnergyDistributionDiscriminator` object.
- "hbos" — Use histogram-based outlier scores (also known as the HBOS method) [4] as distribution confidence scores. The function computes the scores by constructing histograms for the principal component features for each layer that you specify in the `LayerNames` name-value argument. Use the `VarianceCutoff` name-value argument to control the number of principal component features the software uses. To use this method, `XID` must be nonempty. This method creates a `HBOSDistributionDiscriminator` object.

For more information about each method, see “Distribution Confidence Scores” on page 1-1370.

---

**Note** Specifying method as "baseline", "odin", or "energy" is valid only for single-output networks with a softmax layer as the final layer. For example, specifying `net` as a single-output classification network. You can specify method as "hbos" for any network.

---

Data Types: `char` | `string`

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `TruePositiveGoal=0.99, Temperature=10`

### TruePositiveGoal — True positive goal

0.95 (default) | scalar in the range [0, 1]

True positive goal, specified as a scalar in the range [0, 1]. The software returns the threshold that correctly classifies at least this proportion of the ID data as ID.

### Dependency

If you specify a true positive goal, then `XID` must be nonempty and you cannot specify `FalsePositiveGoal`. If you specify a nonempty `XOOD` value, then the software uses only `XID` to compute the threshold.

If you do not specify the `TruePositiveGoal` and `FalsePositiveGoal` name-value arguments and provide nonempty `XID` and `XOOD` values, then the function does not use the default values and instead optimizes over the true positive rate and false positive rate to find the optimal threshold. For more information, see “Algorithms” on page 1-1372.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **FalsePositiveGoal — False positive goal**

`0.05` (default) | scalar in the range `[0, 1]`

False positive goal, specified as a scalar in the range `[0, 1]`. The software returns the threshold that incorrectly classifies at most this proportion of the OOD data as ID.

#### **Dependency**

If you specify a false positive goal, then `XOOD` must be nonempty and you must not specify `TruePositiveGoal`. If you specify a nonempty `XID` value, then the software uses only `XOOD` to compute the threshold.

If you do not specify the `TruePositiveGoal` and `FalsePositiveGoal` name-value arguments and provide nonempty `XID` and `XOOD` values, then the function does not use the default values and instead optimizes over the true positive rate and false positive rate to find the optimal threshold. For more information, see “Algorithms” on page 1-1372.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Temperature — Temperature scaling**

`1` (default) | positive scalar

Temperature scaling for the ODIN and energy methods, specified as a positive scalar. The temperature controls the scaling of the softmax scores when the function computes the distribution confidence scores. For more information, see “Distribution Confidence Scores” on page 1-1370.

#### **Dependency**

To enable this input, specify `method` as “`odin`” or “`energy`”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **LayerNames — Layer names**

string scalar | character vector | string array | cell array of character vectors

Layer names for the HBOS method, specified as a string scalar, character vector, string array, or cell array of character vectors. The default value is the first layer specified in the `OutputNames` property of `net`.

The software computes the distribution confidence scores using the principal components from all of the specified layers. For more information, see “Density-Based Methods” on page 1-1371.

---

**Note** To compute the histograms, the software must first compute the principal component features for each layer. Computing the principal components can be slow for layers with a large output size and can cause you to run out of memory. If you run out of memory, try specifying layers with a smaller output size. For more information, see “Density-Based Methods” on page 1-1371.

---

**Dependency**

To enable this input, specify method as "hbos".

Data Types: char | string | cell

**VarianceCutoff — Variance cutoff**

1e-3 (default) | scalar in the range [0, 1]

Variance cutoff for the HBOS method, specified as a scalar in the range [0, 1].

The variance cutoff controls the number of principal component features that the software uses to compute the distribution confidence scores. Using a greater number of features takes more computation time. The closer the VarianceCutoff value is to 1, the fewer principal components the software uses to compute the scores. For more information, see “Density-Based Methods” on page 1-1371.

**Dependency**

To enable this input, specify method as "hbos".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Output Arguments****discriminator — Distribution discriminator**

BaselineDistributionDiscriminator object | ODINDistributionDiscriminator object | EnergyDistributionDiscriminator object | HBOSDistributionDiscriminator object

Distribution discriminator, returned as a BaselineDistributionDiscriminator, ODINDistributionDiscriminator, EnergyDistributionDiscriminator, or HBOSDistributionDiscriminator object.

The function returns this output as a BaselineDistributionDiscriminator, ODINDistributionDiscriminator, EnergyDistributionDiscriminator, or HBOSDistributionDiscriminator object when you specify method as "baseline", "odin", "energy", or "hbos", respectively.

The Threshold property of discriminator contains the threshold for separating the ID and OOD data. To access the threshold value, call discriminator.Threshold.

**More About****In-Distribution and Out-of-Distribution Data**

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-1370.

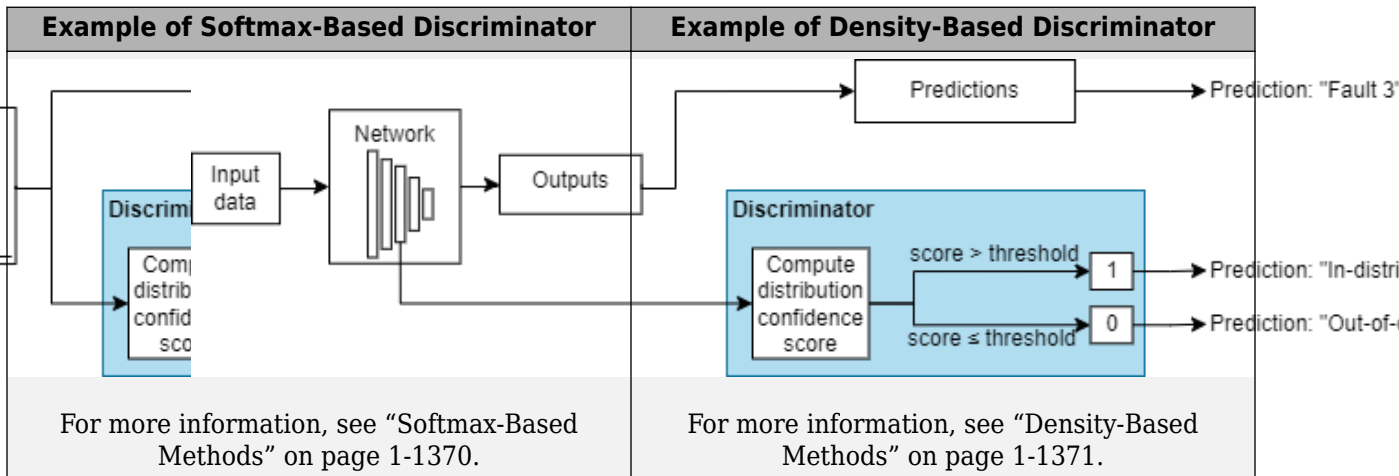
**OOD Data Detection**

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-1370.

These images show how a discriminator acts as an additional output of a trained neural network.

**Example Data Discriminators**



**Distribution Confidence Scores**

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.

**Softmax-Based Methods**

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}}$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is `-Inf`.

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Algorithms

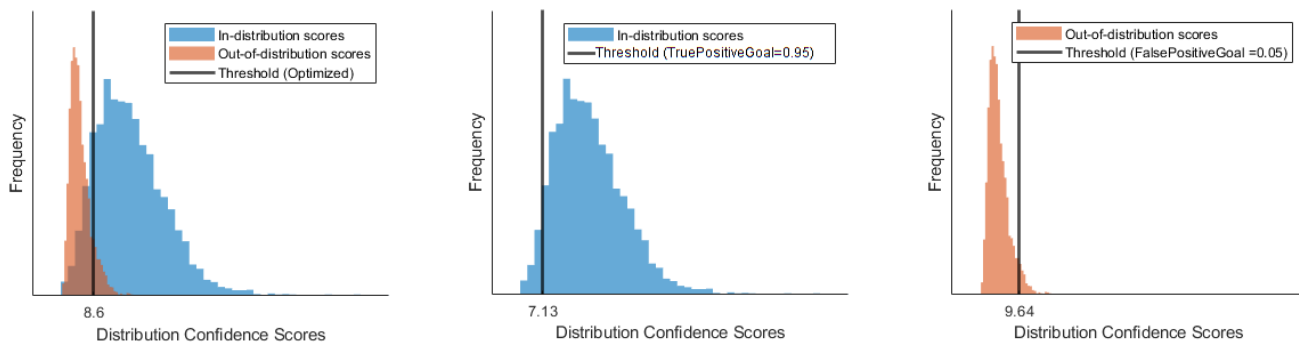
The function creates a discriminator using the trained network. The discriminator behaves as an additional output of the network and classifies an observation as ID or OOD using a threshold. For more information, see “OOD Data Detection” on page 1-1370.

To compute the distribution threshold, the function first computes the distribution confidence scores using the method that you specify in the `method` input argument. For more information, see “Distribution Confidence Scores” on page 1-1370. The software then finds the threshold that best separates the scores of the ID and OOD data. To find the threshold, the software optimizes over these values:

- True positive goal — Number of ID observations that the discriminator correctly classifies as ID. To optimize for this value, the ID data `XID` must be nonempty and you must specify `TruePositiveGoal`. If you specify `TruePositiveGoal` as `p`, then the software finds the threshold above which the proportion of ID confidence scores is `p`. This process is equivalent to finding the 100(1-`p`)-th percentile for the ID confidence scores.
- False positive goal — Number of OOD observations that the discriminator incorrectly classifies as ID. To optimize for this value, the OOD data `XOOD` must be nonempty and you must specify `FalsePositiveGoal`. If you specify `FalsePositiveGoal` as `p`, then the software finds the threshold above which the proportion of OOD confidence scores is `p`. This process is equivalent to finding the 100`p`-th percentile for the OOD confidence scores.

If you provide ID and OOD data and do not specify `TruePositiveGoal` or `FalsePositiveGoal`, then the software finds the threshold that maximizes the balanced accuracy  $\frac{1}{2}(TPR + (1 - FPR))$ . If you provide only ID data, then the software optimizes using only `TruePositiveGoal`, whose default is  $0.95$ . If you provide only OOD data, then the software optimizes using only `FalsePositiveGoal`, whose default is  $0.05$ .

This figure illustrates the different thresholds that the software chooses if you optimize over both the true positive rate and false positive rate, just the true positive rate, or just the false positive rate.



## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. "A Baseline for Detecting Out-of-Distribution Examples in Image Captioning." In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.
- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## 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 its inputs meet either or both of these conditions:
  - Any values of the network learnable parameters are `darray` objects with underlying data of type `gpuArray`. To see the learnables for a network `net`, call `net.Learnables.Value`.
  - Any of the data is a `darray` object with underlying data of type `gpuArray`.

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

### See Also

`isInNetworkDistribution` | `distributionScores`

### Topics

“Out-of-Distribution Detection for Deep Neural Networks”

“Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”

“Verify Robustness of Deep Learning Neural Network”



# neuronPCA

Principal component analysis of neuron activations

## Description

The `compressNetworkUsingProjection` function uses principal component analysis (PCA) to identify the subspace of learnable parameters that result in the highest variance in neuron activations by analyzing the network activations using a data set representative of the training data. The PCA step can be computationally intensive. If you expect to compress the same network multiple times (for example, when exploring different levels of compression), then you can perform the PCA step up front using a `neuronPCA` object.

Forward passes of a projected deep neural network are typically faster when you deploy the network to embedded hardware using library-free C or C++ code generation.

If you prune or quantize your network, then use compression using projection after pruning and before quantization. Network compression using projection supports projecting LSTM layers only.

## Creation

### Syntax

```
npca = neuronPCA(net,mbq)
npca = neuronPCA(net,X1,...,XN)
npca = neuronPCA( ___,LayerNames=names)
npca = neuronPCA( ___,VerbosityLevel=level)
```

### Description

`npca = neuronPCA(net,mbq)` computes the principal component analysis of the neuron activations in `net` using the data in the mini-batch queue `mbq`. The software analyzes the input and output activations of the projectable layers in `net`. This function requires the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

`npca = neuronPCA(net,X1,...,XN)` computes the principal component analysis of the neuron activations using the data in the `darray` objects `X1,...,XN`, where `N` is the number of network inputs.

`npca = neuronPCA( ___,LayerNames=names)` specifies which layers to analyze and sets the `LayerNames` property.

`npca = neuronPCA( ___,VerbosityLevel=level)` specifies the amount of information displayed in the command window.

## Input Arguments

### **net** — Neural network

initialized `dlnetwork` object

Neural network, specified as an initialized `dlnetwork` object.

### **mbq** — Mini-batch queue

`minibatchqueue` object

Mini-batch queue that outputs data for each input of the network, specified as a `minibatchqueue` object.

The PCA step typically works best when using the full training set. However, any dataset that is representative of the training data distribution suffices. The input data must contain two or more observations and sequences must contain two or more time steps.

---

**Note** Do not pad sequence as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

---

### **X1, . . . , XN** — Input data

formatted `dlarray`

Input data, specified as a formatted `dlarray`.

For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.

The PCA step typically works best when using the full training set. However, any dataset that is representative of the training data distribution suffices. The input data must contain two or more observations and sequences must contain two or more time steps.

---

**Note** Do not pad sequence as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

---

### **level** — Verbosity level

"summary" (default) | "steps" | "iterations" | "off"

Verbosity level, specified as one of these values:

- "summary" — Display a summary of the neuron PCA algorithm.
- "steps" — Display information about the steps of the neuron PCA algorithm.
- "iterations" — Display information about the iterations of the neuron PCA algorithm.
- "off" — Do not display information.

## Properties

### **LayerNames** — Names of layers to analyze

string array | cell array of character vectors | character vector

Names of layers to analyze, specified as a string array, cell array of character vectors, or a character vector containing a single layer name.

The software, by default, analyzes all the layers in the network that support projection. Network compression using projection supports projecting LSTM layers only.

Data Types: `string` | `cell`

### **InputEigenvalues — Eigenvalues of activations input to projectable layers**

cell array

Eigenvalues of activations input to projectable layers, specified as a cell array. Each element contains the input eigenvalues for the corresponding projectable layer in `LayerNames`.

Data Types: `cell`

### **InputEigenvectors — Eigenvectors of activations input to projectable layers**

cell array

Eigenvectors of activations input to projectable layers, specified as a cell array. Each element contains the input eigenvectors for the corresponding projectable layer in `LayerNames`.

Data Types: `cell`

### **OutputEigenvalues — Eigenvalues of activations output by projectable layers**

cell array

Eigenvalues of activations output by projectable layers, specified as a cell array. Each element contains the output eigenvalues for the corresponding projectable layer in `LayerNames`.

Data Types: `cell`

### **OutputEigenvectors — Eigenvectors of activations output by projectable layers**

cell array

Eigenvectors of activations output by projectable layers, specified as a cell array. Each element contains the output eigenvectors for the corresponding projectable layer in `LayerNames`.

Data Types: `cell`

## **Examples**

### **Analyze Neuron Activations for Compression Using Projection**

Load the pretrained network in `dlnetJapaneseVowels` and the training data.

```
load dlnetJapaneseVowels
XTrain = japaneseVowelsTrainData;
```

Create a mini-batch queue containing the training data. To create a mini-batch queue from in-memory data, convert the sequences to an array datastore.

```
adsXTrain = arrayDatastore(XTrain,OutputType="same");
```

Create the `minibatchqueue` object.

- Specify a mini-batch size of 16.
- Preprocess the mini-batches using the `preprocessMiniBatchPredictors` function, listed in the Mini-Batch Predictors Preprocessing Function section of the example.
- Specify that the output data has format "CTB" (channel, time, batch).

```
miniBatchSize = 16;
```

```
mbqTrain = minibatchqueue(adsXTrain, ...  
    MiniBatchSize=miniBatchSize, ...  
    MiniBatchFcn=@preprocessMiniBatchPredictors, ...  
    MiniBatchFormat="CTB");
```

Create the `neuronPCA` object.

```
npca = neuronPCA(net,mbqTrain);
```

```
neuronPCA analyzed 1 layers: "lstm"
```

View the properties of the `neuronPCA` object.

```
npca
```

```
npca =  
    neuronPCA with properties:  
  
        LayerNames: "lstm"  
    InputEigenvalues: {[12×1 double]}  
    InputEigenvectors: {[12×12 double]}  
    OutputEigenvalues: {[100×1 double]}  
    OutputEigenvectors: {[100×100 double]}
```

### Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the sequence data from the input cell array and truncating them along the second dimension so that they have the same length.

**Note:** Do not pad sequence data when doing the PCA step for projection as this can negatively impact the analysis. Instead, truncate mini-batches of data to have the same length or use mini-batches of size 1.

```
function X = preprocessMiniBatchPredictors(dataX)
```

```
X = padsequences(dataX,2,Length="shortest");
```

```
end
```

## Version History

Introduced in R2022b

### 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 any of these conditions are met:
  - Any of the values of the network learnable parameters inside `net.Learnables.Value` are `dLarray` objects with underlying data of type `gpuArray`
  - The input argument `mbq` outputs `dLarray` objects with underlying data of type `gpuArray`
  - The input arguments `X1, . . . , XN` are `dLarray` objects with underlying data of type `gpuArray`

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

## See Also

`minibatchqueue` | `dLarray` | `dLnetwork` | `compressNetworkUsingProjection` | `lstmProjectedLayer`

## Topics

“Compress Neural Network Using Projection”

“Parameter Pruning and Quantization of Image Classification Network”

“Quantization of Deep Neural Networks”

## 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 = 1x4  
    100    100     1     2
```

```
ans = 'SSCB'  
ans = 1×2  
      1      2  
ans = 'CB'
```

## Input Arguments

### **mbq** — Mini-batch queue

minibatchqueue object

Mini-batch queue, 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`.

## Version History

**Introduced in R2020b**

## 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”

## 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 sensitivity 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. The total activation fulfills the role of the class score for classification tasks and generalizes the occlusion sensitivity technique to nonclassification tasks.

Use this syntax to compute the occlusion sensitivity map for nonclassification tasks, such as regression, or for use with a network containing a custom classification layer.

`___ = 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)
```

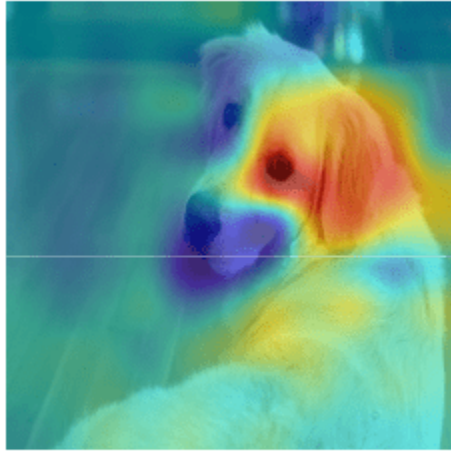
```
label = categorical  
       golden retriever
```

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 sensitivity 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 sensitivity 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 sensitivity 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 sensitivity 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 Computing Requirements” (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 sensitivity 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 sensitivity 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 sensitivity map for the `i`th element in `channel`.

## **Version History**

**Introduced in R2019b**

### **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”

# ODINDistributionDiscriminator

ODIN distribution discriminator

## Description

The `ODINDistributionDiscriminator` is a distribution discriminator that uses the out-of-distribution detector for neural networks (ODIN) method to compute distribution confidence scores. The object contains a threshold that you can use to separate observations into in-distribution (ID) and out-of-distribution (OOD) data sets.

## Creation

Create an `ODINDistributionDiscriminator` object using the `networkDistributionDiscriminator` function and setting the method input argument to "odin".

---

**Note** The object type depends on the method you specify when you use the `networkDistributionDiscriminator` function. The method determines how the software computes the distribution confidence scores. You can set the method argument to either "baseline", "odin", "energy", or "hbos". For more information, see “Distribution Confidence Scores” on page 1-1391.

---

## Properties

### Method — Method for computing distribution confidence scores

"odin"

This property is read-only.

Method for computing the distribution confidence scores, returned as "odin". For more information, see “Distribution Confidence Scores” on page 1-1391.

### Network — Deep learning network

`dlnetwork` object

This property is read-only.

Deep learning network, returned as a `dlnetwork` object.

### Temperature — Temperature scaling

positive scalar

This property is read-only.

Temperature scaling, returned as a positive scalar. The temperature controls the scaling of the softmax scores when the software computes the distribution confidence scores. For more information, see “Distribution Confidence Scores” on page 1-1391.

Example: 1000

**Threshold – Distribution threshold**

positive scalar

This property is read-only.

Distribution threshold, returned as a positive scalar. The threshold separates the ID and OOD data using the distribution confidence scores.

Example: 1.2

**Object Functions**

`isInNetworkDistribution` Determine whether data is within the distribution of the network  
`distributionScores` Distribution confidence scores

**Examples****Create Distribution Discriminator Using ODIN Method**

Load a pretrained classification network.

```
load("digitsClassificationMLPNet.mat");
```

Load ID data. Convert the data to a `darray` object.

```
XID = digitTrain4DArrayData;  
XID = darray(XID, "SSCB");
```

Modify the ID training data to create an OOD set.

```
X00D = XID.*0.3 + 0.1;
```

Create a discriminator using the ODIN method. Set the temperature scaling to 1000.

```
method="odin";  
discriminator = networkDistributionDiscriminator(net,XID,X00D,method, ...  
    Temperature=1000)
```

```
discriminator =  
    ODINDistributionDiscriminator with properties:
```

```
    Method: "odin"  
    Network: [1x1 dlnetwork]  
    Temperature: 1000  
    Threshold: 0.1009
```

Assess the performance of the discriminator on the OOD data.

```
tf = isInNetworkDistribution(discriminator,X00D);  
accuracy = sum(tf == 0)/numel(tf)
```

```
accuracy = 0.9074
```



## More About

### In-Distribution and Out-of-Distribution Data

In-distribution (ID) data refers to any data that you use to construct and train your model. Additionally, any data that is sufficiently similar to the training data is also said to be ID.

Out-of-distribution (OOD) data refers to data that is sufficiently different to the training data. For example, data collected in a different way, at a different time, under different conditions, or for a different task than the data on which the model was originally trained. Models can receive OOD data when you deploy them in an environment other than the one in which you train them. For example, suppose you train a model on clear X-ray images but then deploy the model on images taken with a lower-quality camera.

OOD data detection is important for assigning confidence to the predictions of a network. For more information, see “OOD Data Detection” on page 1-1391.

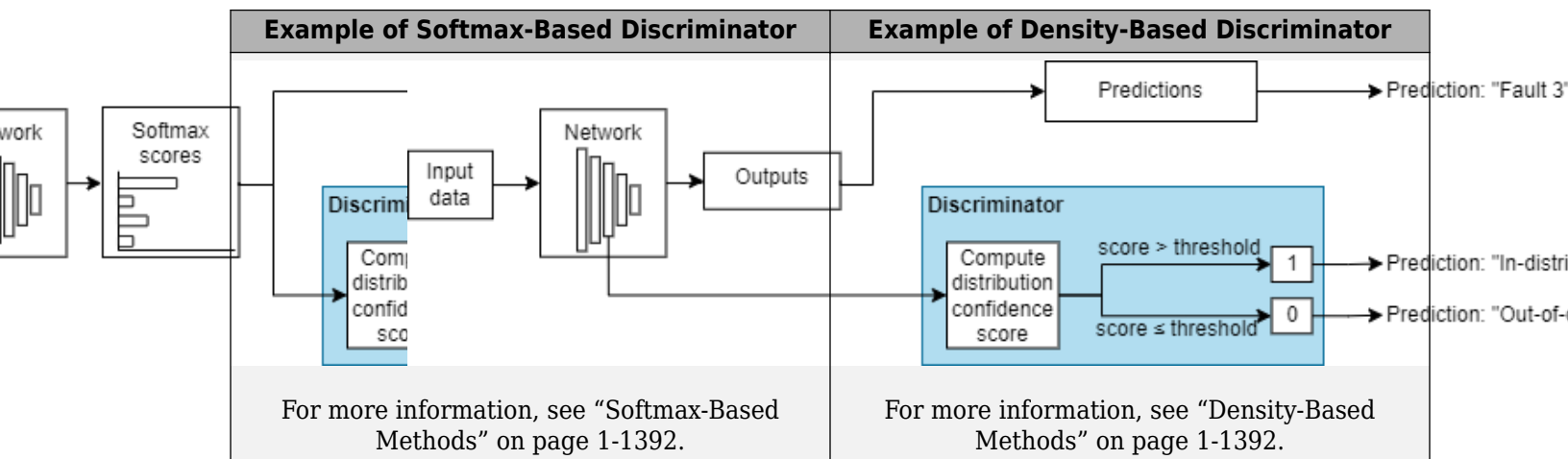
### OOD Data Detection

OOD data detection is a technique for assessing whether the inputs to a network are OOD. For methods that you apply after training, you can construct a discriminator which acts as an additional output of the trained network that classifies an observation as ID or OOD.

The discriminator works by finding a distribution confidence score for an input. You can then specify a threshold. If the score is less than or equal to that threshold, then the input is OOD. Two groups of metrics for computing distribution confidence scores are softmax-based and density-based methods. Softmax-based methods use the softmax layer to compute the scores. Density-based methods use the outputs of layers that you specify to compute the scores. For more information about how to compute distribution confidence scores, see “Distribution Confidence Scores” on page 1-1391.

These images show how a discriminator acts as an additional output of a trained neural network.

### Example Data Discriminators



### Distribution Confidence Scores

Distribution confidence scores are metrics for classifying data as ID or OOD. If an input has a score less than or equal to a threshold value, then you can classify that input as OOD. You can use different techniques for finding the distribution confidence scores.

### Softmax-Based Methods

ID data usually corresponds to a higher softmax output than OOD data[1]. Therefore, a method of defining distribution confidence scores is as a function of the softmax scores. These methods are called *softmax-based* methods. These methods only work for classification networks with a single softmax output.

Let  $a_i(X)$  be the input to the softmax layer for class  $i$ . The output of the softmax layer for class  $i$  is given by this equation:

$$P_i(X; T) = \frac{e^{a_i(X)/T}}{\sum_{j=1}^C e^{a_j(X)/T}},$$

where  $C$  is the number of classes and  $T$  is a temperature scaling. When the network predicts the class label of  $X$ , the temperature  $T$  is set to 1.

The baseline, ODIN, and energy methods each define distribution confidence scores as functions of the softmax input.

- The *baseline* method [1] uses the maximum of the unscaled output values of the softmax scores:

$$\text{confidence}(X) = \max_i P_i(X; 1).$$

- The out-of-distribution detector for neural networks (ODIN) method [2] uses the maximum of the scaled output of the softmax scores:

$$\text{confidence}(X; T) = \max_i P_i(X; T), \quad T > 0.$$

- The *energy* method [3] uses the scaled denominator of the softmax output:

$$\text{energy}(X; T) = -T \log \left( \sum_{j=1}^C e^{a_j(X)/T} \right), \quad T > 0$$

$$\text{confidence}(X; T) = -\text{energy}(X; T), \quad T > 0.$$

### Density-Based Methods

Density-based methods compute the distribution scores by describing the underlying features learned by the network as probabilistic models. Observations falling into areas of low density correspond to OOD observations.

To model the distributions of the features, you can describe the density function for each feature using a histogram. This technique is based on the histogram-based outlier score (HBOS) method [4]. This method uses a data set of ID data, such as training data, to construct histograms representing the density distributions of the ID features. This method has three stages:

- 1 Find the principal component features for which to compute the distribution confidence scores:
  - a For each specified layer, find the activations using the  $n$  data set observations. Flatten the activations across all dimensions except the batch dimension.
  - b Compute the principal components of the flattened activations matrix. Normalize the eigenvalues such that the largest eigenvalue is 1 and corresponds to the principal component that carries the greatest variance through the layer. Denote the matrix of principal components for layer  $l$  by  $Q^{(l)}$ .

The principal components are linear combinations of the activations and represent the features that the software uses to compute the distribution scores. To compute the score, the software uses only the principal components whose eigenvalues are greater than the variance cutoff value  $\sigma$ .

---

**Note** The HBOS algorithm assumes that the features are statistically independent. The principal component features are pairwise linearly independent but they can have nonlinear dependencies. To investigate feature dependencies, you can use functions such as `corr`. For an example showing how to investigate feature dependence, see “Out-of-Distribution Data Discriminator for YOLO v4 Object Detector”. If the features are not statistically independent, then the algorithm can return poor results. Using multiple layers to compute the distribution scores can increase the number of statistically dependent features.

---

- 2 For each of the principal component features with an eigenvalue greater than  $\sigma$ , construct a histogram. For each histogram:
  - a Dynamically adjusts the width of the bins to create  $\sqrt{n}$  bins of approximately equal area.
  - b Normalize the bins heights such that the largest height is 1.
- 3 Find the distribution score for an observation by summing the logarithmic height of the bin containing the observation for each of the feature histograms, over each of the layers.

Let  $f^{(l)}(X)$  denote the output of layer  $l$  for input  $X$ . Use the principal components to project the output into a lower dimensional feature space using this equation:  $\hat{f}^{(l)}(X) = (Q^{(l)})^T f^{(l)}(X)$ .

Compute the confidence score using this equation:

$$\text{HBOS}(X; \sigma) = - \sum_{l=1}^L \left( \sum_{k=1}^{N^{(l)}(\sigma)} \log(\text{hist}_k(\hat{f}_k^{(l)}(X))) \right),$$

$$\text{confidence}(X; \sigma) = - \text{HBOS}(X; \sigma),$$

where  $N^{(l)}(\sigma)$  is the number of number of principal component with an eigenvalue less than  $\sigma$  and  $L$  is the number of layers. A larger score corresponds to an observation that lies in the areas of higher density. If the observation lies outside of the range of any of the histograms, then the bin height for those histograms is 0 and the confidence score is `-Inf`.

---

**Note** The distribution scores depend on the properties of the data set used to construct the histograms [6].

---

## Version History

Introduced in R2023a

## References

- [1] Shalev, Gal, Gabi Shalev, and Joseph Keshet. “A Baseline for Detecting Out-of-Distribution Examples in Image Captioning.” In *Proceedings of the 30th ACM International Conference on Multimedia*, 4175–84. Lisboa Portugal: ACM, 2022. <https://doi.org/10.1145/3503161.3548340>.

- [2] Shiyu Liang, Yixuan Li, and R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks" arXiv:1706.02690 [cs.LG], August 30, 2020, <http://arxiv.org/abs/1706.02690>.
- [3] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li, "Energy-based Out-of-distribution Detection" arXiv:2010.03759 [cs.LG], April 26, 2021, <http://arxiv.org/abs/2010.03759>.
- [4] Markus Goldstein and Andreas Dengel. "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm." KI-2012: poster and demo track 9 (2012).
- [5] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu, "Generalized Out-of-Distribution Detection: A Survey" August 3, 2022, <http://arxiv.org/abs/2110.11334>.
- [6] Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks." arXiv, October 27, 2018. <http://arxiv.org/abs/1807.03888>.

## **See Also**

`networkDistributionDiscriminator` | `isInNetworkDistribution` | `distributionScores`

## **Topics**

"Out-of-Distribution Detection for Deep Neural Networks"

"Verify Robustness of Deep Learning Neural Network"

# 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 = 4x6
    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.

## Version History

**Introduced in R2020b**

**See Also**

onehotencode | categories

**Topics**

“Train Network Using Custom Training Loop”

“Sequence-to-Sequence Classification Using 1-D Convolutions”



# 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 `tblA`, 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 = 6x4
```

```
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=6x4 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=6x3 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=6x11 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 `tblA`, and a 0 in all other variables.

## **Version History**

**Introduced in R2020b**

### **See Also**

`categorical` | `onehotdecode` | `minibatchqueue`

### **Topics**

“Train Network Using Custom Training Loop”

“Sequence-to-Sequence Classification Using 1-D Convolutions”

# 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 -0.0179 -0.1081 0.1081 0.0179 0.1942
```

```
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    1.5112
sPad(1,:,28)
ans = 1×14
    1.0772    1.2365    1.4858    1.6191    1.4893    1.2791    1.4692    1.5592    1.5112    1.5112
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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 padding 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.

## Version History

**Introduced in R2021a**

### **See Also**

`crossentropy` | `onehotencode` | `minibatchqueue`

### **Topics**

“Sequence-to-Sequence Classification Using 1-D Convolutions”

“Long Short-Term Memory Neural Networks”

# 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("Processes",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** — Mini-batch queue

`minibatchqueue` object

Mini-batch queue, 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`.

## Version History

**Introduced in R2020b**

## See Also

`shuffle` | `reset` | `minibatchqueue` | `next`

## Topics

“Train Deep Learning Model in MATLAB”

“Define Custom Training Loops, Loss Functions, and Networks”

# 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 1-1422.

### **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 1-1422.

### **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 1-1422.

### **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**

<code>addParameter</code>	Add parameter to ONNXParameters object
<code>freezeParameters</code>	Convert learnable network parameters in ONNXParameters to nonlearnable
<code>removeParameter</code>	Remove parameter from ONNXParameters object
<code>unfreezeParameters</code>	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 1-1420.

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")
```

Function containing the imported ONNX network architecture was saved to the file `squeezenetFcn.m`. To learn how to use this function, type: `help squeezenetFcn`.

```
params =
  ONNXParameters with properties:

      Learnables: [1x1 struct]
    Nonlearnables: [1x1 struct]
           State: [1x1 struct]
    NumDimensions: [1x1 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: [1x1x512x1000 dlarray]
```

```
conv10_B: [1000x1 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");
```

The network is ready for training. Specify the training options.

```
velocity = [];  
numEpochs = 5;  
miniBatchSize = 16;  
initialLearnRate = 0.01;  
momentum = 0.9;  
decay = 0.01;
```

Calculate the total number of iterations for the training progress monitor.

```
numObservations = size(YTrain,2);  
numIterationsPerEpoch = floor(numObservations./miniBatchSize);  
numIterations = numEpochs*numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object immediately after the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

Train the network.

```
epoch = 0;  
iteration = 0;  
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.
```

```
% Loop over epochs.
```

```
while epoch < numEpochs && ~monitor.Stop
```

```
    epoch = epoch + 1;
```

```
    % Shuffle data.
```

```
    idx = randperm(numObservations);  
    XTrain = XTrain(:,:,,idx);  
    YTrain = YTrain(:,idx);
```

```
    % Loop over mini-batches.
```

```
    i = 0;  
    while i < numIterationsPerEpoch && ~monitor.Stop  
        i = i + 1;  
        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

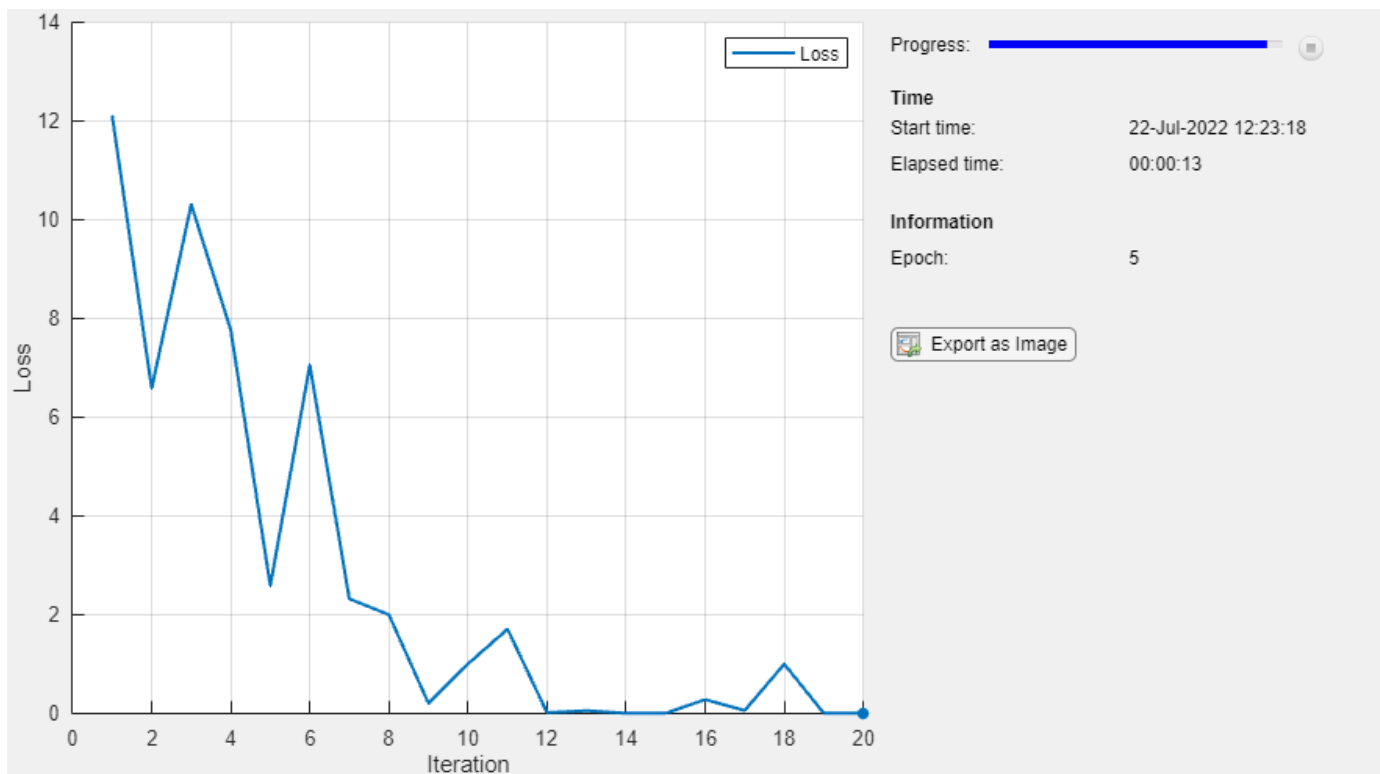
% Evaluate the model gradients and loss using dlfeval and the
% modelGradients function.
[gradients,loss,state] = dlfeval(@modelGradients,X,Y,params);
params.State = state;

% 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);

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch);
monitor.Progress = 100 * iteration/numIterations;
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: [1x1 dlarray]
        conv_W: [5x5x1x20 dlarray]
        conv_B: [20x1 dlarray]
    batchnorm_scale: [20x1 dlarray]
        batchnorm_B: [20x1 dlarray]
        fc_W: [24x24x20x10 dlarray]
        fc_B: [10x1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
    ConvStride1004: [2x1 dlarray]
    ConvDilationFactor1005: [2x1 dlarray]
    ConvPadding1006: [4x1 dlarray]
    ConvStride1007: [2x1 dlarray]
    ConvDilationFactor1008: [2x1 dlarray]
    ConvPadding1009: [4x1 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 `params.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: [5x5x1x20 dlarray]
    conv_B: [20x1 dlarray]
    batchnorm_scale: [20x1 dlarray]
    batchnorm_B: [20x1 dlarray]
    fc_W: [24x24x20x10 dlarray]
    fc_B: [10x1 dlarray]
```

```
params.Nonlearnables
```

```
ans = struct with fields:
    ConvStride1004: [2x1 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.

## Version History

**Introduced in R2020b**

### See Also

`importONNXFunction`

### Topics

“Make Predictions Using Model Function”

“Train Network Using Custom Training Loop”



# 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`.

## Version History

Introduced in R2018a

## See Also

`read` | `readall` | `readByIndex`

# PlaceholderLayer

Layer replacing an unsupported Keras or ONNX layer

## 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.

## Creation

Importing layers from a Keras or ONNX network that has layers that are not supported by Deep Learning Toolbox creates 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:
    InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
    Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
```

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'         2-D Convolution      20 7x7 convolutions with s
    3 'conv2d_1_relu'    ReLU                  ReLU
    4 'conv2d_2'         2-D Convolution      20 3x3 convolutions with s
    5 'conv2d_2_relu'    ReLU                  ReLU
    6 'gaussian_noise_1' PLACEHOLDER LAYER    Placeholder for 'GaussianN
    7 'gaussian_noise_2' PLACEHOLDER LAYER    Placeholder for 'GaussianN
```

```

8 'max_pooling2d_1'      2-D Max Pooling      2x2 max pooling with stride
9 'max_pooling2d_2'      2-D 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.

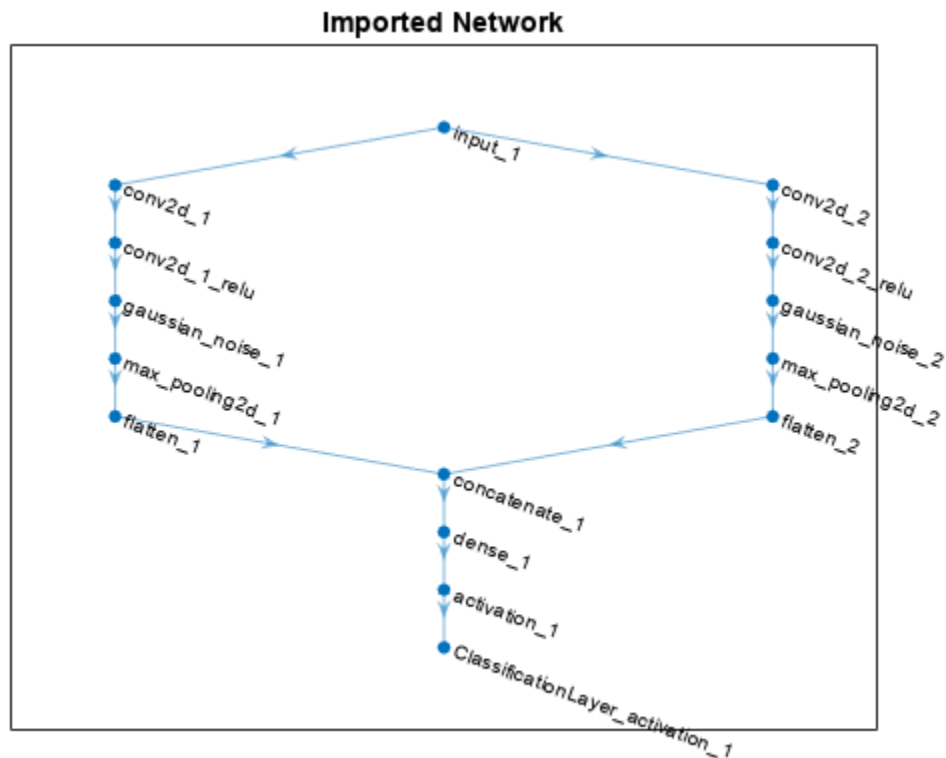
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename, 'ImportWeights', true);
```

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
```

Create two Gaussian noise layers with the same configurations as the imported Keras layers using the helper `gaussianNoiseLayer` function.

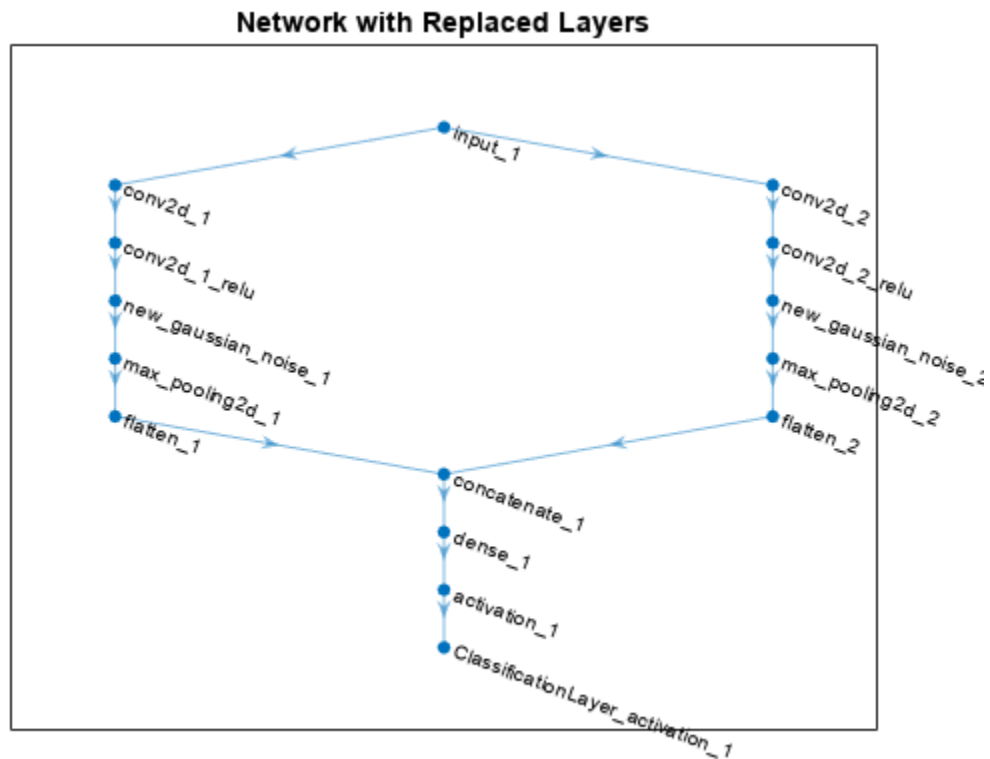
```
gnLayer1 = gaussianNoiseLayer(1.5, 'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7, 'new_gaussian_noise_2');
```

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'         2-D Convolution      20 7x7x1 convolutions with
     3  'conv2d_1_relu'    ReLU                  ReLU
     4  'conv2d_2'         2-D Convolution      20 3x3x1 convolutions with
     5  'conv2d_2_relu'    ReLU                  ReLU
     6  'new_gaussian_noise_1' Gaussian Noise        Gaussian noise with standa
     7  'new_gaussian_noise_2' Gaussian Noise        Gaussian noise with standa
     8  'max_pooling2d_1'  2-D Max Pooling      2x2 max pooling with stride
     9  'max_pooling2d_2'  2-D 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'}
```

## Version History

**Introduced in R2017b**

## See Also

`importKerasLayers` | `importONNXLayers` | `findPlaceholderLayers` | `assembleNetwork` | `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”



# plot

Plot neural network architecture

## 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.

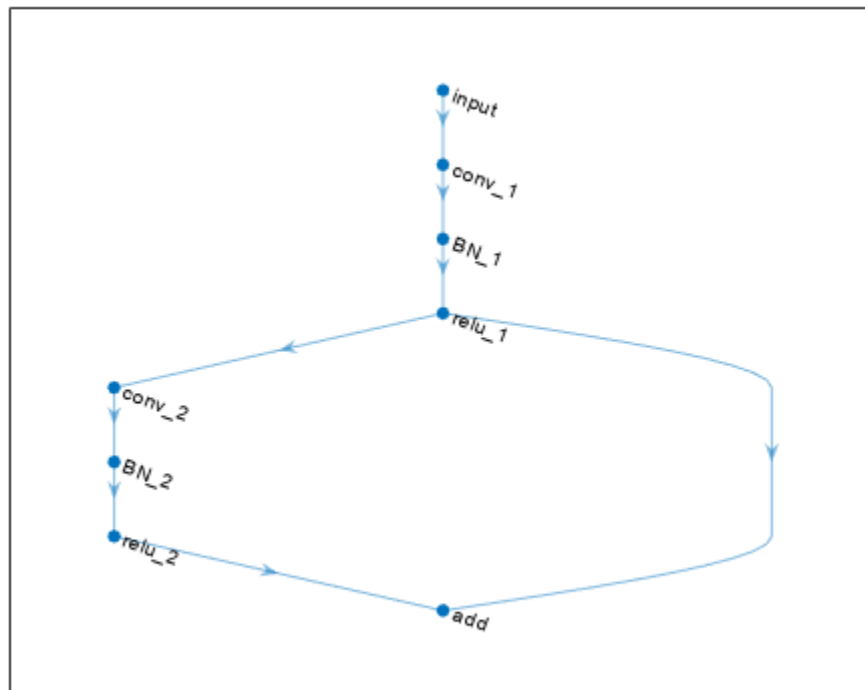
```
layers = [
    imageInputLayer([32 32 3], 'Name', 'input')
    convolution2dLayer(3,16, 'Padding', 'same', 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'BN_1')
    reluLayer('Name', 'relu_1')

    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 =
```

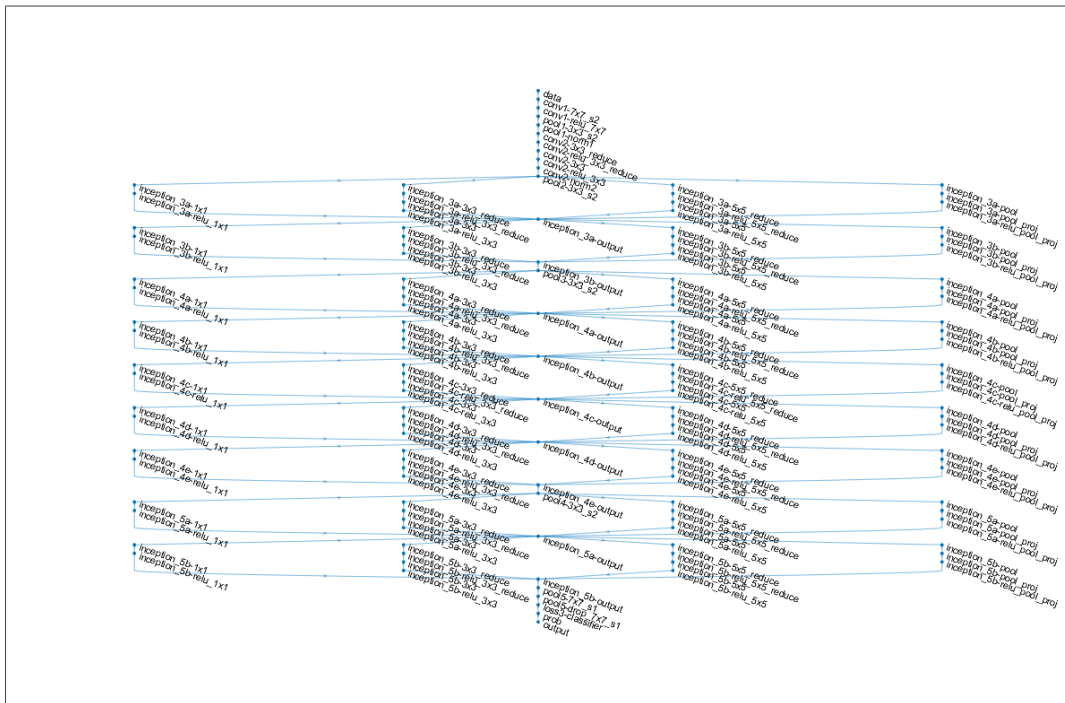
```
DAGNetwork with properties:
```

```
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.

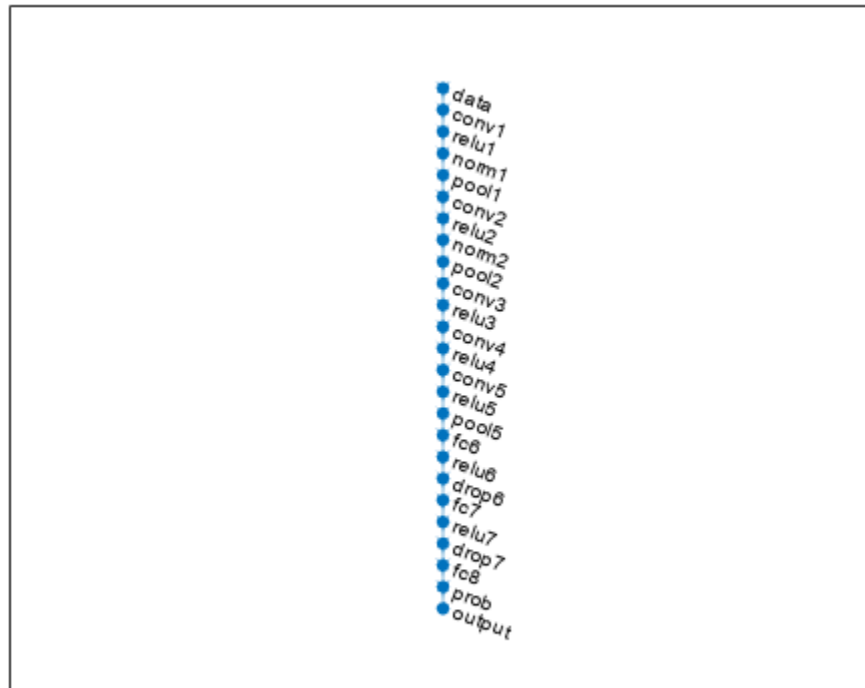
```
net = alexnet
```

```
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` — Deep learning network

SeriesNetwork object | DAGNetwork object | `dlnetwork` object

Deep learning network, specified as a SeriesNetwork, DAGNetwork, or `dlnetwork` object.

## Version History

Introduced in R2017b

### See Also

**Deep Network Designer** | `layerGraph` | `addLayers` | `removeLayers` | `replaceLayer` | `connectLayers` | `disconnectLayers` | `analyzeNetwork` | `dlnetwork` | `summary`

### Topics

“Train Residual Network for Image Classification”

“Train Deep Learning Network to Classify New Images”

## plot

Plot receiver operating characteristic (ROC) curves and other performance curves

### Syntax

```
plot(rocObj)
plot(ax, rocObj)
plot( ____, Name=Value)
curveObj = plot( ____)
[curveObj, graphicsObjs] = plot( ____)
```

### Description

`plot(rocObj)` creates a receiver operating characteristic (ROC) curve on page 1-1446, which is a plot of the true positive rate (TPR) versus the false positive rate (FPR), for each class in the `ClassNames` property of the `rocmetrics` object `rocObj`. The function marks the model operating point on page 1-1624 for each curve, and displays the value of the area under the ROC curve (AUC on page 1-1446) and the class name for the curve in the legend.

`plot(ax, rocObj)` creates the plot on the axes specified by `ax` instead of the current axes.

`plot( ____, Name=Value)` specifies additional options using one or more name-value arguments in addition to any of the input argument combinations in the previous syntaxes. For example, `AverageROCType="macro", ClassNames=[]` computes the average performance metrics using the macro-averaging method and plots the average ROC curve only.

`curveObj = plot( ____)` returns a `ROCCurve` object for each performance curve.

`[curveObj, graphicsObjs] = plot( ____)` also returns graphics objects for the model operating points and diagonal line.

### Examples

#### Plot ROC Curve

Load a sample of predicted classification scores and true labels for a classification problem.

```
load('flowersDataResponses.mat')
```

`trueLabels` is the true labels for an image classification problem and `scores` is the softmax prediction scores. `scores` is an N-by-K array where N is the number of observations and K is the number of classes.

```
trueLabels = flowersData.trueLabels;
scores = flowersData.scores;
```

Load the class names. The column order of `scores` follows the class order stored in `classNames`.

```
classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels in `trueLabels` and the classification scores in `scores`. Specify the column order of scores using `classNames`.

```
rocObj = rocmetrics(trueLabels,scores,classNames);
```

`rocObj` is a `rocmetrics` object that stores the AUC values and performance metrics for each class in the `AUC` and `Metrics` properties. Display the `AUC` property.

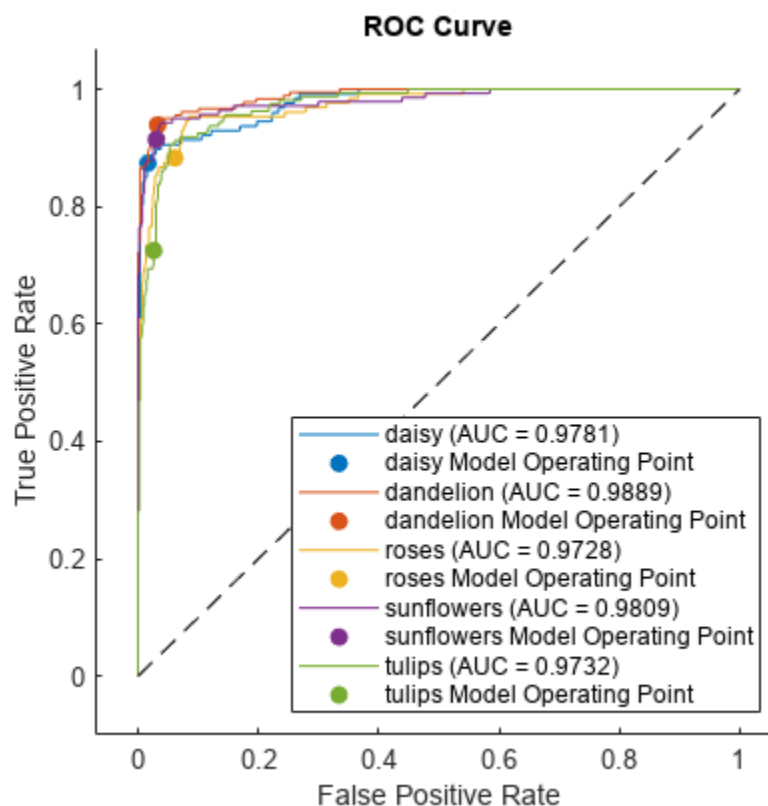
```
rocObj.AUC
```

```
ans = 1x5 single row vector
```

```
    0.9781    0.9889    0.9728    0.9809    0.9732
```

Plot the ROC curve for each class.

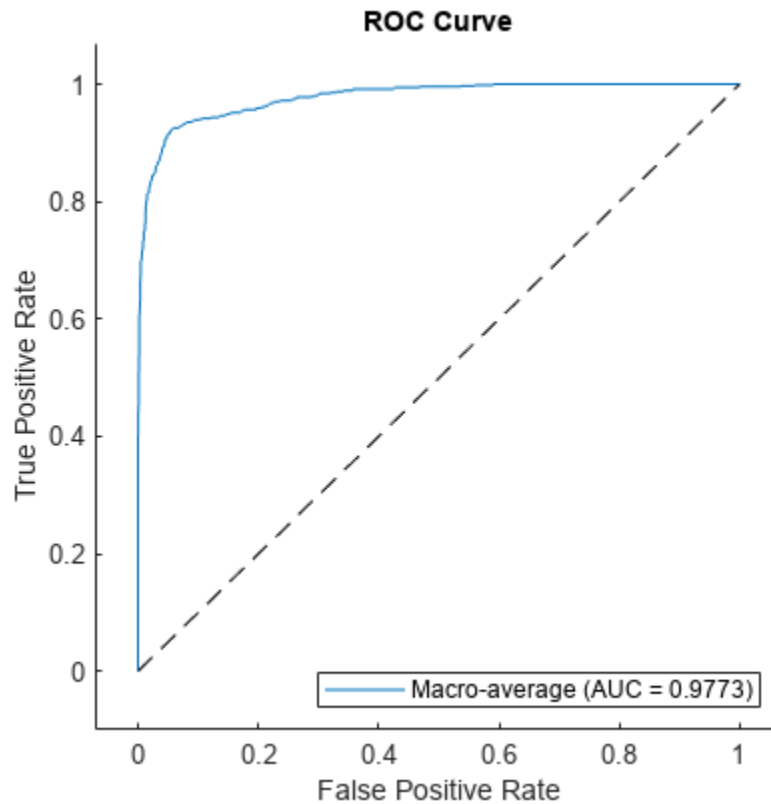
```
plot(rocObj)
```



The filled circle markers indicate the model operating points. The legend displays the class name and AUC value for each curve.

Plot the macro average ROC curve.

```
plot(rocObj,AverageROCType=["macro"],ClassNames=[])
```



### Plot Precision-Recall Curve and Detection Error Tradeoff (DET) Graph

Create a `rocmetrics` object and plot performance curves by using the `plot` function. Specify the `XAxisMetric` and `YAxisMetric` name-value arguments of the `plot` function to plot different types of performance curves other than the ROC curve. If you specify new metrics when you call the `plot` function, the function computes the new metrics and then uses them to plot the curve.

Load a sample of true labels and the prediction scores for a classification problem. For this example, there are five classes: daisy, dandelion, roses, sunflowers, and tulips. The class names are stored in `classNames`. The scores are the softmax prediction scores generated using the `predict` function. `scores` is an N-by-K array where N is the number of observations and K is the number of classes. The column order of `scores` follows the class order stored in `classNames`.

```
load('flowersDataResponses.mat')

scores = flowersData.scores;
trueLabels = flowersData.trueLabels;

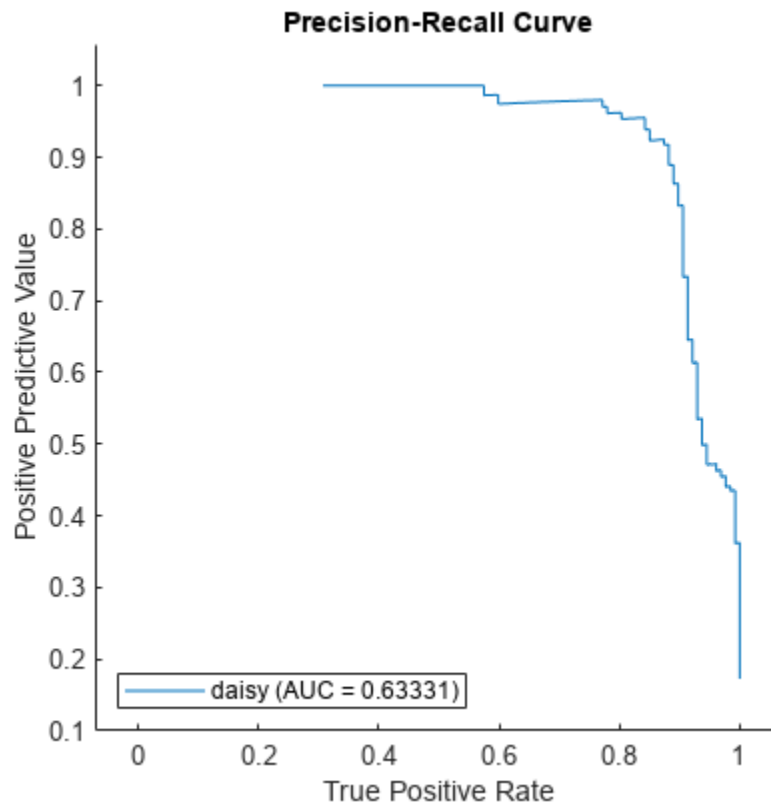
classNames = flowersData.classNames;
```

Create a `rocmetrics` object. The `rocmetrics` function computes the FPR and TPR at different thresholds.

```
rocObj = rocmetrics(trueLabels,scores,classNames);
```

Plot the precision-recall curve for the first class. Specify the y-axis metric as precision (or positive predictive value) and the x-axis metric as recall (or true positive rate). The `plot` function computes the new metric values and plots the curve. Compute the area under the precision-recall curve using the trapezoidal method of the `trapz` function, and display the value in the legend.

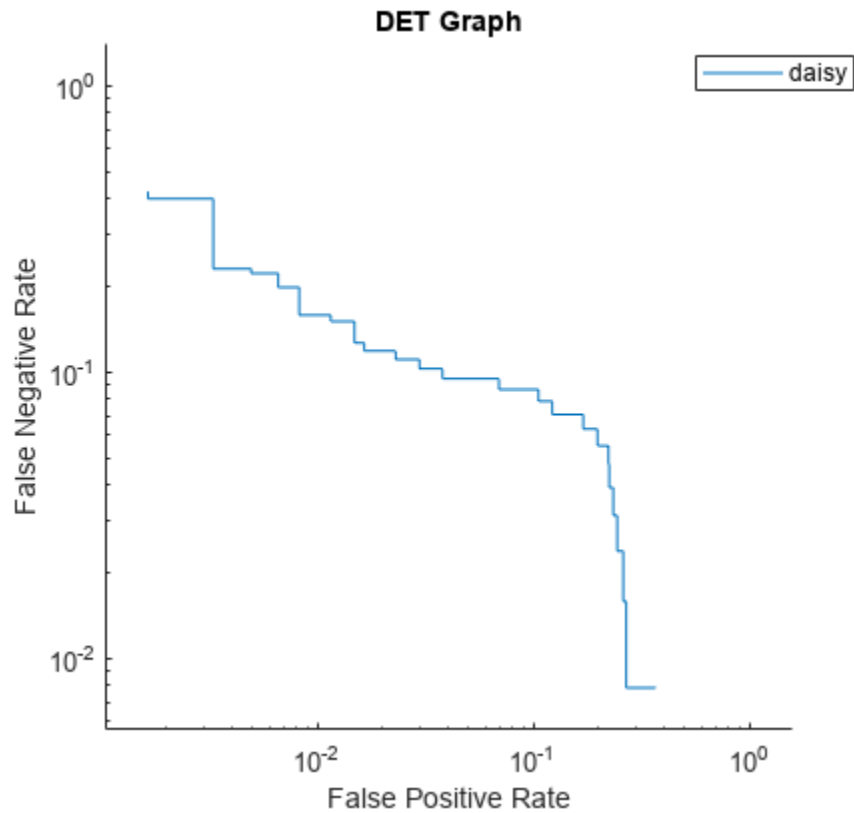
```
curveObj = plot(rocObj,ClassNames=classNames(1), ...
    YAxisMetric="PositivePredictiveValue",XAxisMetric="TruePositiveRate");
xyData = rmmissing([curveObj.XData curveObj.YData]);
auc = trapz(xyData(:,1),xyData(:,2));
legend(join([string(classNames(1)) " (AUC = " string(auc) ")"],""), ...
    Location="southwest")
title("Precision-Recall Curve")
```



Plot the detection error tradeoff (DET) graph for the first class. Specify the y-axis metric as the false negative rate and the x-axis metric as the false positive rate. Use a log scale for the x-axis and y-axis.

```
f = figure;
plot(rocObj,ClassNames=classNames(1), ...
    YAxisMetric="FalseNegativeRate",XAxisMetric="FalsePositiveRate")
f.CurrentAxes.XScale = "log";
f.CurrentAxes.YScale = "log";
title("DET Graph")
```





### Plot Confidence Intervals

Compute the confidence intervals for FPR and TPR for fixed threshold values by using bootstrap samples, and plot the confidence intervals for TPR on the ROC curve by using the `plot` function. This examples requires Statistics and Machine Learning Toolbox™.

Load a sample of true labels and the prediction scores for a classification problem. For this example, there are five classes: daisy, dandelion, roses, sunflowers, and tulips. The class names are stored in `classNames`. The scores are the softmax prediction scores generated using the `predict` function. `scores` is an N-by-K array where N is the number of observations and K is the number of classes. The column order of `scores` follows the class order stored in `classNames`.

```
load('flowersDataResponses.mat')

scores = flowersData.scores;
trueLabels = flowersData.trueLabels;

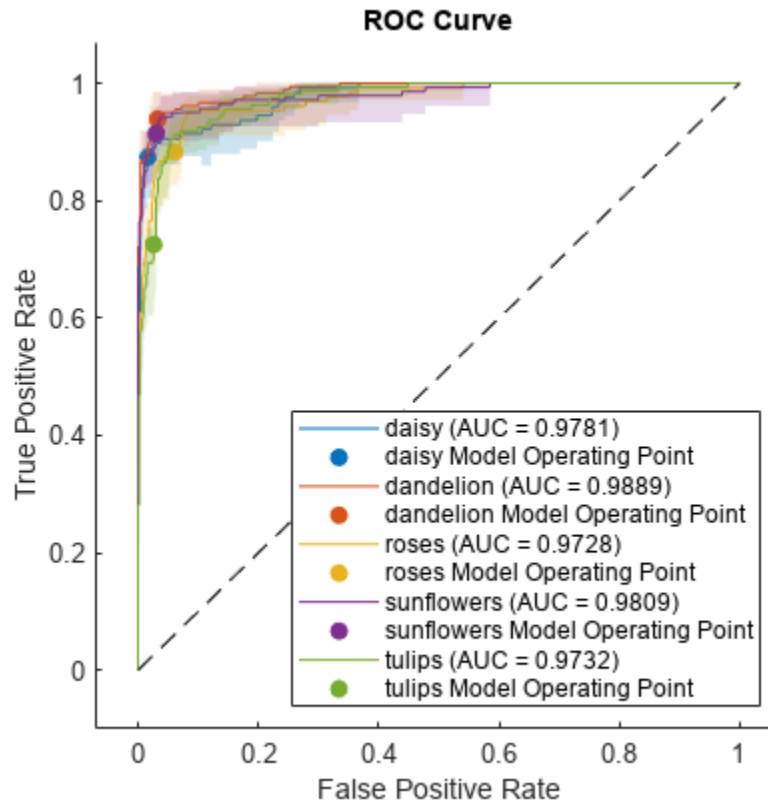
classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels in `trueLabels` and the classification scores in `scores`. Specify the column order of scores using `classNames`. Specify `NumBootstraps` as 100 to use 100 bootstrap samples to compute the confidence intervals.

```
rocObj = rocmetrics(trueLabels,scores,classNames,NumBootstraps=100);
```

Plot the ROC curve and the confidence intervals for TPR. Specify `ShowConfidenceIntervals=true` to show the confidence intervals.

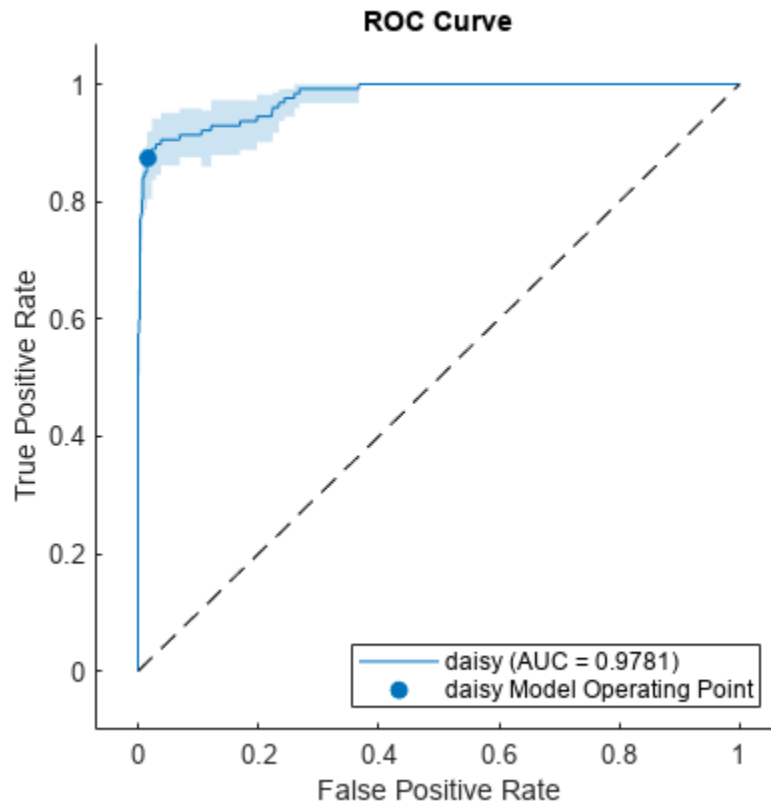
```
plot(rocObj,ShowConfidenceIntervals=true)
```



The shaded area around each curve indicates the confidence intervals. `rocmetrics` computes the ROC curves using the scores. The confidence intervals represent the estimates of uncertainty for the curve.

Specify one class to plot by using the `ClassNames` name-value argument.

```
plot(rocObj,ShowConfidenceIntervals=true,ClassNames="daisy")
```



## Input Arguments

### **rocObj** — Object evaluating classification performance

rocmetrics object

Object evaluating classification performance, specified as a rocmetrics object.

### **ax** — Target axes

Axes object

Target axes, specified as an Axes object.

If you do not specify the axes and the current axes are Cartesian, then `plot` uses the current axes (`gca`). For more information on creating an Axes object, see `axes` and `gca`.

## 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: `YAxisMetric="PositivePredictiveValue", XAxisMetric="TruePositiveRate"` plots the precision (positive predictive value) versus the recall (true positive rate), which represents a precision-recall curve.

**AverageROCType — Method for averaging ROC curves**

"none" (default) | "micro" | "macro" | "weighted" | string array | cell array of character vectors

Method for averaging ROC curves, specified as "none", "micro", "macro", "weighted", a string array of method names, or a cell array of method names.

- If you specify "none" (default), the `plot` function does not create the average ROC curve. The `AverageROCType` value must be "none" if `plot` creates performance curves other than a ROC curve.
- If you specify multiple methods as a string array or a cell array of character vectors, then the `plot` function plots multiple average ROC curves using the specified methods.
- If you specify one or more averaging methods and specify `ClassNames=[]`, then the `plot` function plots only the average ROC curves.

`plot` computes the averages of performance metrics for a multiclass classification problem, and plots the average ROC curves using these methods:

- "micro" (micro-averaging) — `plot` finds the average performance metrics by treating all one-versus-all on page 1-1446 binary classification problems as one binary classification problem. The function computes the confusion matrix components for the combined binary classification problem, and then computes the average FPR and TPR using the values of the confusion matrix.
- "macro" (macro-averaging) — `plot` computes the average values for FPR and TPR by averaging the values of all one-versus-all binary classification problems.
- "weighted" (weighted macro-averaging) — `plot` computes the weighted average values for FPR and TPR using the macro-averaging method and using the prior class probabilities (the `Prior` property of `rocObj`) as weights.

The algorithm type determines the length of the vectors in the `XData`, `YData`, and `Thresholds` properties of a `ROCCurve` object, returned by `plot`, for the average ROC curve. For more details, see "Average of Performance Metrics".

Example: `AverageROCType="macro"`

Example: `AverageROCType=["micro","macro"]`

Data Types: `char` | `string`

**ClassNames — Class labels to plot**

`rocObj.ClassNames` (default) | categorical array | character array | string array | logical vector | numeric vector | cell array of character vectors

Class labels to plot, specified as a categorical, character, or string array, logical or numeric vector, or cell array of character vectors. The values and data types in `ClassNames` must match those of the class names in the `ClassNames` property of `rocObj`. (The software treats character or string arrays as cell arrays of character vectors.)

- If you specify multiple class labels, the `plot` function plots a ROC curve for each class.
- If you specify `ClassNames=[]` and specify one or more averaging methods using `AverageROCType`, then the `plot` function plots only the average ROC curves.

Example: `ClassNames=["red","blue"]`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell` | `categorical`

**ShowConfidenceIntervals — Flag to show confidence intervals of y-axis metric**

false or 0 (default) | true or 1

Flag to show the confidence intervals of the y-axis metric (`YAxisMetric`), specified as logical 0 (false) or 1 (true).

The `ShowConfidenceIntervals` value can be true only if the `Metrics` property of `rocObj` contains the confidence intervals for the y-axis metric.

Example: `ShowConfidenceIntervals=true`

Using confidence intervals requires Statistics and Machine Learning Toolbox.

Data Types: logical

**ShowDiagonalLine — Flag to show diagonal line**

true or 1 | false or 0

Flag to show the diagonal line that extends from  $[0, 0]$  to  $[1, 1]$ , specified as logical 1 (true) or 0 (false).

The default value is true if you plot a ROC curve or an average ROC curve, and false otherwise.

In the ROC curve plot, the diagonal line represents a random classifier, and the line passing through  $[0, 0]$ ,  $[0, 1]$ , and  $[1, 1]$  represents a perfect classifier.

Example: `ShowDiagonalLine=false`

Data Types: logical

**ShowModelOperatingPoint — Flag to show model operating point**

true or 1 | false or 0

Flag to show the model operating point on page 1-1624, specified as logical 1 (true) or 0 (false).

The default value is true for a ROC curve, and false for an average ROC curve. The `ShowModelOperatingPoint` value must be false for performance curves other than ROC.

Example: `ShowModelOperatingPoint=false`

Data Types: logical

**XAxisMetric — Metric for x-axis**

"FalsePositiveRate" (default) | name of performance metric | function handle

Metric for the x-axis, specified as a character vector or string scalar of the built-in metric name or a custom metric name, or a function handle (`@metricName`).

- Built-in metrics — Specify one of the following built-in metric names by using a character vector or string scalar.

Name	Description
"TruePositives" or "tp"	Number of true positives (TP)
"FalseNegatives" or "fn"	Number of false negatives (FN)
"FalsePositives" or "fp"	Number of false positives (FP)

Name	Description
"TrueNegatives" or "tn"	Number of true negatives (TN)
"SumOfTrueAndFalsePositives" or "tp+fp"	Sum of TP and FP
"RateOfPositivePredictions" or "rpp"	Rate of positive predictions (RPP), $(TP+FP)/(TP+FN+FP+TN)$
"RateOfNegativePredictions" or "rnp"	Rate of negative predictions (RNP), $(TN+FN)/(TP+FN+FP+TN)$
"Accuracy" or "accu"	Accuracy, $(TP+TN)/(TP+FN+FP+TN)$
"TruePositiveRate" or "tpr"	True positive rate (TPR), also known as recall or sensitivity, $TP/(TP+FN)$
"FalseNegativeRate", "fnr", or "miss"	False negative rate (FNR), or miss rate, $FN/(TP+FN)$
"FalsePositiveRate" or "fpr"	False positive rate (FPR), also known as fallout or 1-specificity, $FP/(TN+FP)$
"TrueNegativeRate", "tnr", or "spec"	True negative rate (TNR), or specificity, $TN/(TN+FP)$
"PositivePredictiveValue", "ppv", or "prec"	Positive predictive value (PPV), or precision, $TP/(TP+FP)$
"NegativePredictiveValue" or "npv"	Negative predictive value (NPV), $TN/(TN+FN)$
"ExpectedCost" or "ecost"	<p>Expected cost, <math>(TP*cost(P P)+FN*cost(N P)+FP*cost(P N)+TN*cost(N N))/(TP+FN+FP+TN)</math>, where <math>cost</math> is a 2-by-2 misclassification cost matrix containing <math>[0, cost(N P); cost(P N), 0]</math>. <math>cost(N P)</math> is the cost of misclassifying a positive class (P) as a negative class (N), and <math>cost(P N)</math> is the cost of misclassifying a negative class as a positive class.</p> <p>The software converts the K-by-K matrix specified by the <code>Cost</code> name-value argument of <code>rocmetrics</code> to a 2-by-2 matrix for each one-versus-all binary problem. For details, see "Misclassification Cost Matrix".</p>

The software computes the scale vector using the prior class probabilities (`Prior`) and the number of classes in `Labels`, and then scales the performance metrics according to this scale vector. For details, see "Performance Metrics".

- Custom metric stored in the `Metrics` property — Specify the name of a custom metric stored in the `Metrics` property of the input object `rocObj`. The `rocmetrics` function names a custom metric "CustomMetricN", where N is the number that refers to the custom metric. For example, specify `XAxisMetric="CustomMetric1"` to use the first custom metric in `Metrics` as a metric for the x-axis.
- Custom metric — Specify a new custom metric by using a function handle. A custom function that returns a performance metric must have this form:

```
metric = customMetric(C,scale,cost)
```

- The output argument `metric` is a scalar value.
- A custom metric is a function of the confusion matrix (`C`), scale vector (`scale`), and cost matrix (`cost`). The software finds these input values for each one-versus-all binary problem. For details, see “Performance Metrics”.
  - `C` is a 2-by-2 confusion matrix consisting of [TP, FN; FP, TN].
  - `scale` is a 2-by-1 scale vector.
  - `cost` is a 2-by-2 misclassification cost matrix.

The `plot` function names a custom metric “Custom Metric” for the axis label.

The software does not support cross-validation for a custom metric. Instead, you can specify to use bootstrap when you create a `rocmetrics` object.

If you specify a new metric instead of one in the `Metrics` property of the input object `rocObj`, the `plot` function computes and plots the metric values. If you compute confidence intervals when you create `rocObj`, the `plot` function also computes confidence intervals for the new metric.

The `plot` function ignores NaNs in the performance metric values. Note that the positive predictive value (PPV) is NaN for the reject-all threshold for which  $TP = FP = 0$ , and the negative predictive value (NPV) is NaN for the accept-all threshold for which  $TN = FN = 0$ . For more details, see “Thresholds, Fixed Metric, and Fixed Metric Values”.

Example: `XAxisMetric="FalseNegativeRate"`

Data Types: `char` | `string` | `function_handle`

### **YAxisMetric — Metric for y-axis**

`"TruePositiveRate"` (default) | name of performance metric | function handle

Metric for the y-axis, specified as a character vector or string scalar of the built-in metric name or custom metric name, or a function handle (`@metricName`). For details, see `XAxisMetric`.

Example: `YAxisMetric="FalseNegativeRate"`

Data Types: `char` | `string` | `function_handle`

## **Output Arguments**

### **curveObj — Object for performance curve**

`ROCCurve` object | array of `ROCCurve` objects

Object for the performance curve, returned as a `ROCCurve` object or an array of `ROCCurve` objects. `plot` returns a `ROCCurve` object for each performance curve.

Use `curveObj` to query and modify properties of the plot after creating it. For a list of properties, see `ROCCurve` Properties.

### **graphicsObjs — Graphics objects**

graphics array

Graphics objects for the model operating points and diagonal line, returned as a graphics array containing `Scatter` and `Line` objects.

`graphicsObjs` contains a `Scatter` object for each model operating point (if `ShowModelOperatingPoint=true`) and a `Line` object for the diagonal line (if

ShowDiagonalLine=true). Use `graphicsObjs` to query and modify properties of the model operating points and diagonal line after creating the plot. For a list of properties, see Scatter Properties and Line Properties.

## More About

### Receiver Operating Characteristic (ROC) Curve

A ROC curve shows the true positive rate versus the false positive rate for different thresholds of classification scores.

The true positive rate and the false positive rate are defined as follows:

- True positive rate (TPR), also known as recall or sensitivity —  $TP / (TP + FN)$ , where TP is the number of true positives and FN is the number of false negatives
- False positive rate (FPR), also known as fallout or 1-specificity —  $FP / (TN + FP)$ , where FP is the number of false positives and TN is the number of true negatives

Each point on a ROC curve corresponds to a pair of TPR and FPR values for a specific threshold value. You can find different pairs of TPR and FPR values by varying the threshold value, and then create a ROC curve using the pairs. For each class, `rocmetrics` uses all distinct adjusted score on page 1-1447 values as threshold values to create a ROC curve.

For a multiclass classification problem, `rocmetrics` formulates a set of one-versus-all on page 1-1446 binary classification problems to have one binary problem for each class, and finds a ROC curve for each class using the corresponding binary problem. Each binary problem assumes one class as positive and the rest as negative.

For a binary classification problem, if you specify the classification scores as a matrix, `rocmetrics` formulates two one-versus-all binary classification problems. Each of these problems treats one class as a positive class and the other class as a negative class, and `rocmetrics` finds two ROC curves. Use one of the curves to evaluate the binary classification problem.

For more details, see “ROC Curve and Performance Metrics”.

### Area Under ROC Curve (AUC)

The area under a ROC curve (AUC) corresponds to the integral of a ROC curve (TPR values) with respect to FPR from  $FPR = 0$  to  $FPR = 1$ .

The AUC provides an aggregate performance measure across all possible thresholds. The AUC values are in the range 0 to 1, and larger AUC values indicate better classifier performance.

### One-Versus-All (OVA) Coding Design

The one-versus-all (OVA) coding design reduces a multiclass classification problem to a set of binary classification problems. In this coding design, each binary classification treats one class as positive and the rest of the classes as negative. `rocmetrics` uses the OVA coding design for multiclass classification and evaluates the performance on each class by using the binary classification that the class is positive.

For example, the OVA coding design for three classes formulates three binary classifications:



	Binary 1	Binary 2	Binary 3
Class 1	1	-1	-1
Class 2	-1	1	-1
Class 3	-1	-1	1

Each row corresponds to a class, and each column corresponds to a binary classification problem. The first binary classification assumes that class 1 is a positive class and the rest of the classes are negative. `rocmetrics` evaluates the performance on the first class by using the first binary classification problem.

## Algorithms

### Adjusted Scores for Multiclass Classification Problem

For each class, `rocmetrics` adjusts the classification scores (input argument `Scores` of `rocmetrics`) relative to the scores for the rest of the classes if you specify `Scores` as a matrix. Specifically, the adjusted score for a class given an observation is the difference between the score for the class and the maximum value of the scores for the rest of the classes.

For example, if you have  $[s_1, s_2, s_3]$  in a row of `Scores` for a classification problem with three classes, the adjusted score values are  $[s_1 - \max(s_2, s_3), s_2 - \max(s_1, s_3), s_3 - \max(s_1, s_2)]$ .

`rocmetrics` computes the performance metrics using the adjusted score values for each class.

For a binary classification problem, you can specify `Scores` as a two-column matrix or a column vector. Using a two-column matrix is a simpler option because the `predict` function of a classification object returns classification scores as a matrix, which you can pass to `rocmetrics`. If you pass scores in a two-column matrix, `rocmetrics` adjusts scores in the same way that it adjusts scores for multiclass classification, and it computes performance metrics for both classes. You can use the metric values for one of the two classes to evaluate the binary classification problem. The metric values for a class returned by `rocmetrics` when you pass a two-column matrix are equivalent to the metric values returned by `rocmetrics` when you specify classification scores for the class as a column vector.

## Version History

Introduced in R2022b

## References

- [1] Sebastiani, Fabrizio. "Machine Learning in Automated Text Categorization." *ACM Computing Surveys* 34, no. 1 (March 2002): 1-47.

## See Also

`rocmetrics` | `ROCCurve` Properties | `addMetrics` | `average`

## Topics

"ROC Curve and Performance Metrics"  
 "Compare Deep Learning Models Using ROC Curves"

## predict

Predict responses using trained deep learning neural network

### Syntax

```
Y = predict(net,images)
Y = predict(net,sequences)
Y = predict(net,features)

Y = predict(net,X1,...,XN)
Y = predict(net,mixed)

[Y1,...,YM] = predict( ___ )

___ = 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 a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value argument.

Use this function to predict responses using a trained `SeriesNetwork` or `DAGNetwork` object. For information on predicting responses using a `dlnetwork` object, see `predict`.

`Y = predict(net,images)` predicts the responses of the specified images using the trained network `net`.

`Y = predict(net,sequences)` predicts the responses of the specified sequences using the trained network `net`.

`Y = predict(net,features)` predicts the responses of the specified feature data using the trained network `net`.

`Y = predict(net,X1,...,XN)` predicts the responses for the data in the numeric or cell arrays `X1, ..., XN` for the multi-input network `net`. The input `Xi` corresponds to the network input `net.InputNames(i)`.

`Y = predict(net,mixed)` predicts the responses using the trained network `net` with multiple inputs of mixed data types.

`[Y1,...,YM] = predict( ___ )` predicts responses for the `M` outputs of a multi-output network using any of the previous input arguments. The output `Yj` corresponds to the network output `net.OutputNames(j)`. To return categorical outputs for the classification output layers, set the `ReturnCategorical` option to `1` (true).

`___ = predict( ___ ,Name=Value)` predicts the responses with additional options specified by one or more name-value arguments.

**Tip**

- Use the `predict` function to predict responses using a regression network or to classify data using a multi-output network. To classify data using a single-output classification network, use the `classify` function.
- When you make 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.
- For predicting responses using `dlnetwork` objects, see `predict`.

**Examples****Predict Numeric Responses Using Trained Convolutional Neural Network**

Load the pretrained network `digitsRegressionNet`. This network is a regression convolutional neural network that predicts the angle of rotation of handwritten digits.

```
load digitsRegressionNet
```

View the network layers. The output layer of the network is a regression layer.

```
layers = net.Layers
```

```
layers =
```

```
18x1 Layer array with layers:
```

1	'imageinput'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv_1'	2-D Convolution	8 3x3x1 convolutions with stride [1 1] and padding [1 1]
3	'batchnorm_1'	Batch Normalization	Batch normalization with 8 channels
4	'relu_1'	ReLU	ReLU
5	'avgpool2d_1'	2-D Average Pooling	2x2 average pooling with stride [2 2] and padding [1 1]
6	'conv_2'	2-D Convolution	16 3x3x8 convolutions with stride [1 1] and padding [1 1]
7	'batchnorm_2'	Batch Normalization	Batch normalization with 16 channels
8	'relu_2'	ReLU	ReLU
9	'avgpool2d_2'	2-D Average Pooling	2x2 average pooling with stride [2 2] and padding [1 1]
10	'conv_3'	2-D Convolution	32 3x3x16 convolutions with stride [1 1] and padding [1 1]
11	'batchnorm_3'	Batch Normalization	Batch normalization with 32 channels
12	'relu_3'	ReLU	ReLU
13	'conv_4'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding [1 1]
14	'batchnorm_4'	Batch Normalization	Batch normalization with 32 channels
15	'relu_4'	ReLU	ReLU
16	'dropout'	Dropout	20% dropout
17	'fc'	Fully Connected	1 fully connected layer
18	'regressionoutput'	Regression Output	mean-squared-error with response 'Response'

Load the test images.

```
XTest = digitTest4DArrayData;
```

Predict the responses of the input data using the `predict` function.

```
YTest = predict(net,XTest);
```

View some of the test images at random with their predictions.

```

numPlots = 9;
idx = randperm(size(XTest,4),numPlots);

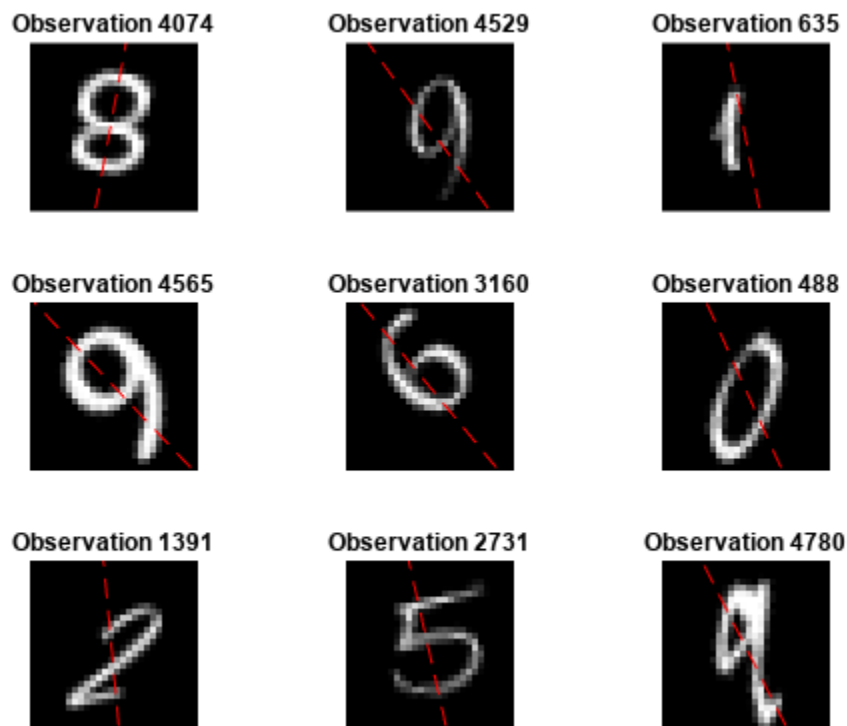
sz = size(XTest,1);
offset = sz/2;

figure
tiledlayout("flow")

for i = 1:numPlots
    nexttile
    imshow(XTest(:,:,,idx(i)))
    title("Observation " + idx(i))

    hold on
    plot(offset*[1-tand(YTest(idx(i))) 1+tand(YTest(idx(i)))],[sz 0],"r--")
    hold off
end

```



### Predict Numeric Responses of Sequences Using Trained LSTM Network

Load the pretrained network `freqNet`. This network is an LSTM regression neural network that predicts the frequency of waveforms.

```
load freqNet
```

View the network layers. The output layer of the network is a regression layer.

```
net.Layers
```

```
ans =
  4x1 Layer array with layers:

    1 'sequenceinput'    Sequence Input    Sequence input with 3 dimensions
    2 'lstm'             LSTM             LSTM with 100 hidden units
    3 'fc'              Fully Connected  1 fully connected layer
    4 'regressionoutput' Regression Output mean-squared-error with response 'Response'
```

Load the test sequences.

```
load WaveformData
```

```
X = data;
```

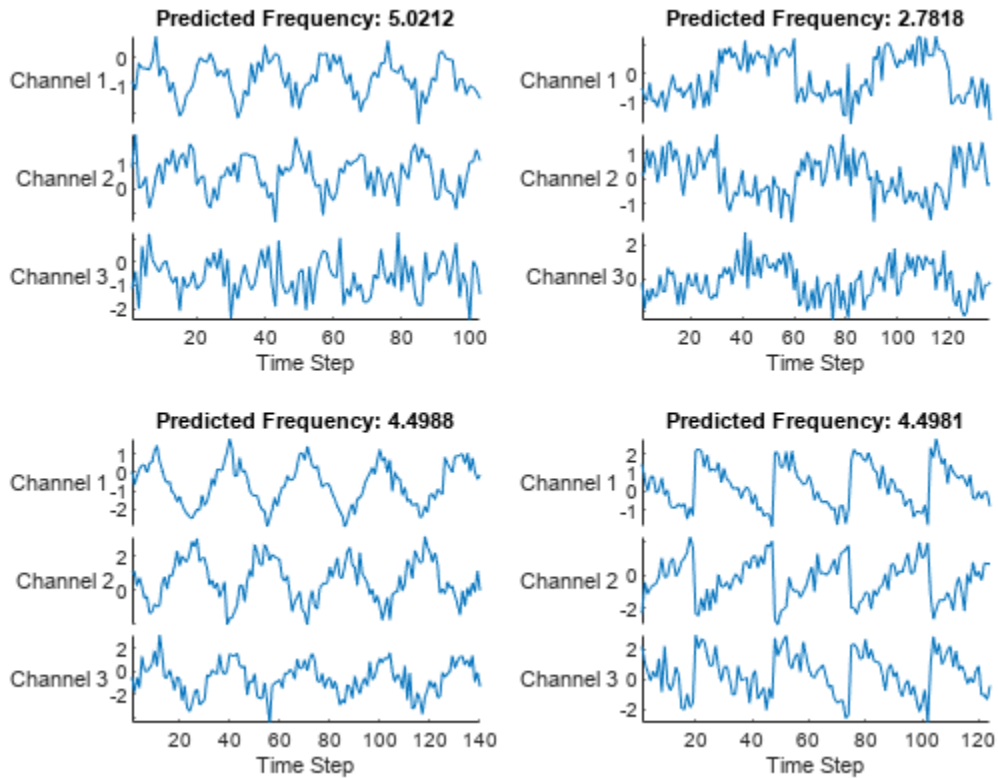
Predict the responses of the input data using the `predict` function. Because the network was trained using sequences truncated to the shortest sequence length of each mini-batch, also truncate the test sequences by setting the `SequenceLength` option to "shortest".

```
Y = predict(net,X,SequenceLength="shortest");
```

Visualize the first few predictions in a plot.

```
figure
tiledlayout(2,2)
for i = 1:4
    nexttile
    stackedplot(X{i}',DisplayLabels="Channel " + (1:3))

    xlabel("Time Step")
    title("Predicted Frequency: " + string(Y(i)))
end
```



## 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`.

For information on predicting responses using `dlnetwork` objects, see `predict`.

### **images** — Image data

datastore | numeric array | table

Image data, specified as one of the following.

Data Type		Description	Example Usage
Datastore	ImageDatastore	Datastore of images saved on disk	Make predictions with images saved on disk, where the images are the same size.  When the images are different sizes, use an <b>AugmentedImageDatastore</b> object.
	AugmentedImageDatastore	Datastore that applies random affine geometric transformations, including resizing, rotation, reflection, shear, and translation	Make predictions with images saved on disk, where the images are different sizes.
	TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by <b>predict</b>.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>Make predictions using networks with multiple inputs.</li> <li>Combine predictors from different data sources.</li> </ul>
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions using data in a format that other datastores do not support.  For details, see “Develop Custom Mini-Batch Datastore”.
Numeric array		Images specified as a numeric array	Make predictions using data that fits in memory and does not require additional processing like resizing.
Table		Images specified as a table	Make predictions using data stored in a table.

When you use a datastore with 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.

**Datstore**

Datstores read mini-batches of images and responses. Use datstores when you have data that does not fit in memory or when you want to resize the input data.

These datstores are directly compatible with `predict` for image data.:

- `ImageDatastore`
- `AugmentedImageDatastore`
- `CombinedDatstore`
- `TransformedDatstore`
- Custom mini-batch datstore. For details, see “Develop Custom Mini-Batch Datstore”.

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 the `imageDatstore` function for preprocessing or resizing, as this option is usually significantly slower.

You can use other built-in datstores for making predictions by using the `transform` and `combine` functions. These functions can convert the data read from datstores to the format required by `classify`.

The required format of the datstore output depends on the network architecture.

Network Architecture	Datstore Output	Example Output
Single input	<p>Table or cell array, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom datstores must output tables.</p>	<pre>data = read(ds) data =     4x1 table            Predictors     _____     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}     {224x224x3 double}</pre>



Network Architecture	Datastore Output	Example Output
		<pre>data = read(ds)  data =      4×1 cell array      {224×224×3 double}     {224×224×3 double}     {224×224×3 double}     {224×224×3 double}</pre>
Multiple input	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>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</pre>

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

Data	Format
2-D images	<i>h-by-w-by-c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively
3-D images	<i>h-by-w-by-d-by-c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively

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.

The size and shape of the numeric array depends on the type of image data.

Data	Format
2-D images	<i>h-by-w-by-c-by-N</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively, and <i>N</i> is the number of images

Data	Format
3-D images	<i>h-by-w-by-d-by-c-by-N</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively, and <i>N</i> is the number of images

**Table**

As an alternative to datastores or numeric arrays, you can also specify images in a table.

When you specify images 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

---

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`

Complex Number Support: Yes

**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	<code>TransformedDatastore</code>	<ul style="list-style-type: none"> <li>• Transform datastores with outputs not supported by <code>predict</code>.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
	<code>CombinedDatastore</code>	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>

Data Type		Description	Example Usage
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions 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	Make predictions using data that fits in memory and does not require additional processing like custom transformations.

### Datastore

Datastores read mini-batches of sequences and responses. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with `predict` for sequence data:

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `predict`. For example, you can transform and combine data read from in-memory arrays and CSV files using an `ArrayDatastore` and an `TabularTextDatastore` object, respectively.

The datastore must return data in a table or cell array. Custom mini-batch datastores must output tables.

Datastore Output	Example Output
Table	<pre>data = read(ds)  data =      4x2 table      Predictors     _____     {12x50 double}     {12x50 double}     {12x50 double}     {12x50 double}</pre>

Datastore Output	Example Output
Cell array	<pre>data = read(ds)  data =      4×2 cell array      {12×50 double}     {12×50 double}     {12×50 double}     {12×50 double}</pre>

The format of the predictors depends 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 $h$ 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.

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.

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.

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.

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** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

### features — Feature data

`datastore` | numeric array | table

Feature data, specified as one of the following.

Data Type	Description	Example Usage
Datastore	<code>TransformedDatastore</code>	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by <code>predict</code>.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	<code>CombinedDatastore</code>	<ul style="list-style-type: none"> <li>Make predictions using networks with multiple inputs.</li> <li>Combine predictors from different data sources.</li> </ul>

Data Type		Description	Example Usage
	Custom mini-batch datastore	Custom datastore that returns mini-batches of data	Make predictions 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	Make predictions using data stored in a table.
Numeric array		Feature data specified as numeric array	Make predictions 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. Use datastores when you have data that does not fit in memory or when you want to apply transformations to the data.

These datastores are directly compatible with `predict` for feature data:

- `CombinedDatastore`
- `TransformedDatastore`
- Custom mini-batch datastore. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for making predictions 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 `predict`. 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	<p>Table or cell array with at least one column, where the first column specifies the predictors.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom mini-batch datastores must output tables.</p>	<p>Table for network with one input:</p> <pre>data = read(ds) data =     4x2 table         Predictors     _____     {24x1 double}     {24x1 double}     {24x1 double}     {24x1 double}</pre> <p>Cell array for network with one input:</p> <pre>data = read(ds) data =     4x1 cell array         {24x1 double}         {24x1 double}         {24x1 double}         {24x1 double}</pre>
Multiple input layers	<p>Cell array with at least <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<p>Cell array for network with two inputs:</p> <pre>data = read(ds) data =     4x3 cell array         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}         {24x1 double}  {28x1 double}</pre>

The predictors must be  $c$ -by-1 column vectors, where  $c$  is the number of features.

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 in the table columns depends on the type of task.

Task	Predictors
Feature classification	Features specified in one or more columns as scalars.

**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.

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** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`  
 Complex Number Support: Yes

**X1, . . . , XN — Numeric or cell arrays for networks with multiple inputs**

`numeric array` | `cell array`

Numeric or cell arrays for networks with multiple inputs.

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`  
 Complex Number Support: Yes

**mixed — Mixed data**

`TransformedDatastore` | `CombinedDatastore` | `custom mini-batch datastore`

Mixed data, specified as one of the following.



Data Type	Description	Example Usage
TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Transform outputs of datastores not supported by <code>predict</code> so they have the required format.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
CombinedDatastore	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>

You can use other built-in datastores for making predictions 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 `predict`. For more information, see “Datastores for Deep Learning”.

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.

Datastore Output	Example Output
<p>Cell array with <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>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}</pre>

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To convert a numeric array to a datastore, use `arrayDatastore`.

## 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in 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 you make 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Acceleration — Performance optimization

"auto" (default) | "mex" | "none"

Performance optimization, specified as 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 only when you use a GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- "none" — Disable all acceleration.

If `Acceleration` is "auto", then MATLAB applies a number of compatible optimizations and does not generate a MEX function.

The "auto" and "mex" options can offer performance benefits 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 supports networks that contain the layers listed on the "Supported Layers" (GPU Coder) page, except for `sequenceInputLayer` objects.

The "mex" option is available when you use a single GPU.

To use the "mex" option, you must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning 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.

For quantized networks, the "mex" option requires a CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3, or higher.

MATLAB Compiler does not support deploying networks when you use the "mex" option.

### **ExecutionEnvironment — Hardware resource**

"auto" (default) | "gpu" | "cpu" | "multi-gpu" | "parallel"

Hardware resource, specified as one of the following:

- "auto" — Use a GPU if one is available; otherwise, use the CPU.
- "gpu" — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see "GPU Computing Requirements" (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".

The "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 Computing Requirements" (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 make predictions in parallel with networks with recurrent layers (by setting ExecutionEnvironment to either "multi-gpu" or "parallel"), the SequenceLength option must be "shortest" or "longest".

Networks with custom layers that contain State parameters do not support making predictions in parallel.

### **ReturnCategorical — Option to return categorical labels**

0 (false) (default) | 1 (true)

Option to return categorical labels, specified as 0 (false) or 1 (true).

If ReturnCategorical is 1 (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 or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is 'sequence' for each recurrent 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**

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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### Y — Predicted responses

numeric array | categorical array | cell array

Predicted responses, returned as a numeric array, a categorical array, or a cell array. The format of Y depends on the type of problem.

The following table describes the format for regression problems.

Task	Format
2-D image regression	<ul style="list-style-type: none"> <li><math>N</math>-by-<math>R</math> matrix, where <math>N</math> is the number of images and <math>R</math> is the number of responses</li> <li><math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>N</math> numeric array, where <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and number of channels of the images, respectively, and <math>N</math> is the number of images</li> </ul>
3-D image regression	<ul style="list-style-type: none"> <li><math>N</math>-by-<math>R</math> matrix, where <math>N</math> is the number of images and <math>R</math> is the number of responses</li> <li><math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math>-by-<math>N</math> numeric array, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and number of channels of the images, respectively, and <math>N</math> is the number of images</li> </ul>
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	<p><math>N</math>-by-1 cell array of numeric sequences, where <math>N</math> is the number of sequences. The sequences are matrices with <math>R</math> rows, where <math>R</math> is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after the <code>SequenceLength</code> option is applied to each mini-batch independently.</p> <p>For sequence-to-sequence regression tasks with one observation, sequences can be a matrix. In this case, Y is a matrix of responses.</p>
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, Y is a matrix of responses.

If `ReturnCategorical` is 0 (false) and the output layer of the network is a classification layer, then Y is the predicted classification scores. This table describes the format of the scores for classification tasks.

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	

Task	Format
Feature classification	
Sequence-to-sequence classification	<i>N</i> -by-1 cell array of matrices, where <i>N</i> is the number of observations. The sequences are matrices with <i>K</i> rows, where <i>K</i> is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after the <code>SequenceLength</code> option is applied to each mini-batch independently.

If `ReturnCategorical` is 1 (true), and the output layer of the network is a classification layer, then *Y* is a categorical vector or a cell array of categorical vectors. This table describes the format of the labels for classification tasks.

Task	Format
Image or feature classification	<i>N</i> -by-1 categorical vector of labels, where <i>N</i> is the number of observations
Sequence-to-label classification	
Sequence-to-sequence classification	<p><i>N</i>-by-1 cell array of categorical sequences of labels, where <i>N</i> is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after the <code>SequenceLength</code> option is applied to each mini-batch independently.</p> <p>For sequence-to-sequence classification tasks with one observation, sequences can be a matrix. In this case, <i>Y</i> is a categorical sequence of labels.</p>

***Y*<sub>1</sub>, . . . , *Y*<sub>*M*</sub> — Predicted scores or responses of networks with multiple outputs**

numeric array | categorical array | cell array

Predicted scores or responses of networks with multiple outputs, returned as numeric arrays, categorical arrays, or cell arrays.

Each output *Y*<sub>*j*</sub> corresponds to the network output `net.OutputNames(j)` and has format as described in the *Y* output argument.

## Algorithms

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get

different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Alternatives

For networks with a single classification layer only, you can compute the predicted classes and the predicted scores from a trained network using the `classify` function.

To compute the activations from a network layer, use the `activations` function.

For recurrent networks such as LSTM networks, you can make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.

## Version History

### Introduced in R2016a

### **R2022b: Prediction functions pad mini-batches to length of longest sequence before splitting when you specify `SequenceLength` option as an integer**

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

## References

- [1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Multidimensional Curve Classification Using Passing-through Regions." *Pattern Recognition Letters* 20, no. 11-13 (November 1999): 1103-11. [https://doi.org/10.1016/S0167-8655\(99\)00077-X](https://doi.org/10.1016/S0167-8655(99)00077-X).
- [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:
  - `Y = predict(net, images)`, where `images` is a numeric array
  - `Y = predict(net, sequences)`, where `sequences` is a cell array
  - `Y = predict(net, features)`, where `features` is a numeric array
  - `[Y1, ..., YM] = predict(__)` using any of the previous syntaxes
  - `__ = predict(__, Name=Value)` using any of the previous syntaxes
- For numeric inputs, the input 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" options of the `SequenceLength` name-value pair is supported for code generation.
- If `ReturnCategorical` is 1 (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:
  - `Y = predict(net, images)`, where `images` is a numeric array
  - `Y = predict(net, sequences)`, where `sequences` is a cell array or numeric array
  - `Y = predict(net, features)`, where `features` is a numeric array
  - `[Y1, ..., YM] = predict(__)` using any of the previous syntaxes
  - `__ = predict(__, Name=Value)` using any of the previous syntaxes
- For numeric inputs, the input 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 1 (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™.

- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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

### See Also

`activations` | `classify` | `classifyAndUpdateState` | `predictAndUpdateState`

### Topics

"Train Convolutional Neural Network for Regression"  
 "Sequence-to-Sequence Regression Using Deep Learning"  
 "Sequence-to-One Regression Using Deep Learning"  
 "Time Series Forecasting Using Deep Learning"  
 "Convert Classification Network into Regression Network"  
 "Deep Learning in MATLAB"  
 "Deep Learning Tips and Tricks"

## predictAndUpdateState

Predict responses using a trained recurrent neural network and update the network state

### Syntax

```
[updatedNet,Y] = predictAndUpdateState(recNet,sequences)
[updatedNet,Y] = predictAndUpdateState(recNet,X1,...,XN)
[updatedNet,Y] = predictAndUpdateState(recNet,mixed)
[updatedNet,Y1,...,YM] = predictAndUpdateState( ___ )
[ ___ ] = predictAndUpdateState( ___ ,Name=Value)
```

### Description

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). Specify the hardware requirements using the `ExecutionEnvironment` name-value argument.

`[updatedNet,Y] = 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 such as an LSTM layer or a custom layer with state parameters.

`[updatedNet,Y] = predictAndUpdateState(recNet,X1,...,XN)` predicts the responses for the data in the numeric or cell arrays `X1, ..., XN` for the multi-input network `recNet`. The input `Xi` corresponds to the network input `recNet.InputNames(i)`.

`[updatedNet,Y] = predictAndUpdateState(recNet,mixed)` makes predictions using the multi-input network `recNet` with data of mixed data types.

`[updatedNet,Y1,...,YM] = predictAndUpdateState( ___ )` predicts responses for the `M` outputs of a multi-output network using any of the previous input arguments. The output `Yj` corresponds to the network output `recNet.OutputNames(j)`. To return categorical outputs for the classification output layers, set the `ReturnCategorical` option to `1` (true).

`[ ___ ] = predictAndUpdateState( ___ ,Name=Value)` makes predictions with additional options specified by one or more name-value arguments using any of the previous syntaxes. For example, `MiniBatchSize=27` makes predictions using mini-batches of size 27.

---

**Tip** When you make 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,TTest] = 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.

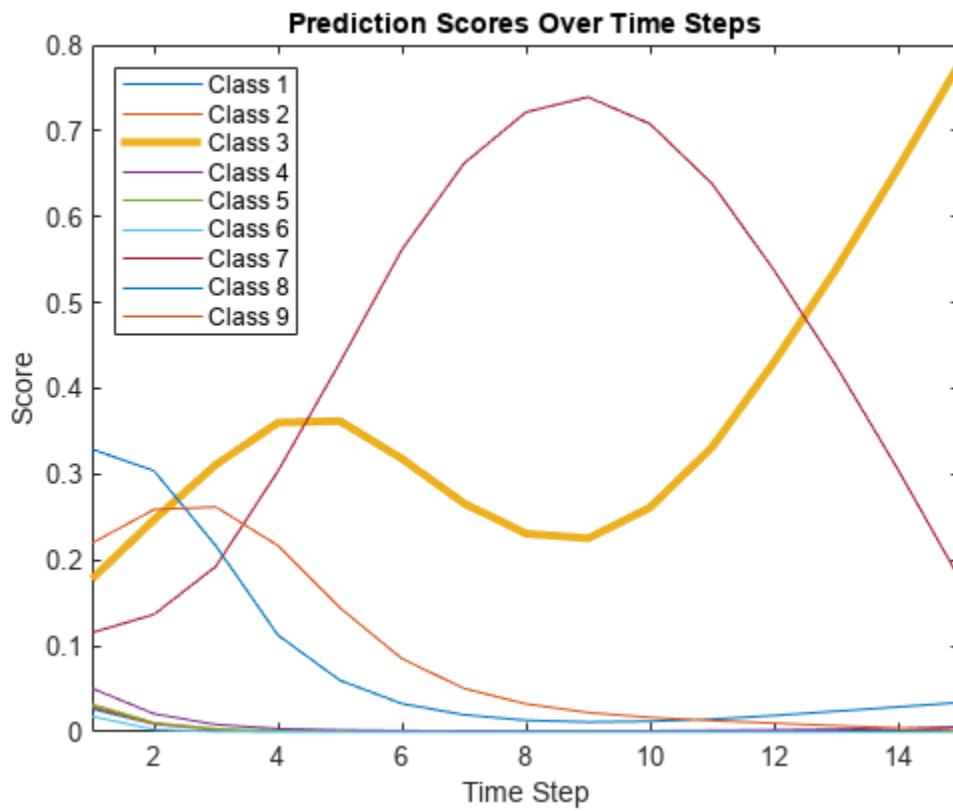
```
classNames = string(net.Layers(end).Classes);
figure
lines = plot(scores');
xlim([1 numTimeSteps])
legend("Class " + classNames,Location="northwest")
xlabel("Time Step")
ylabel("Score")
title("Prediction Scores Over Time Steps")
```

Highlight the prediction scores over time steps for the correct class.

```
trueLabel = TTest(94)

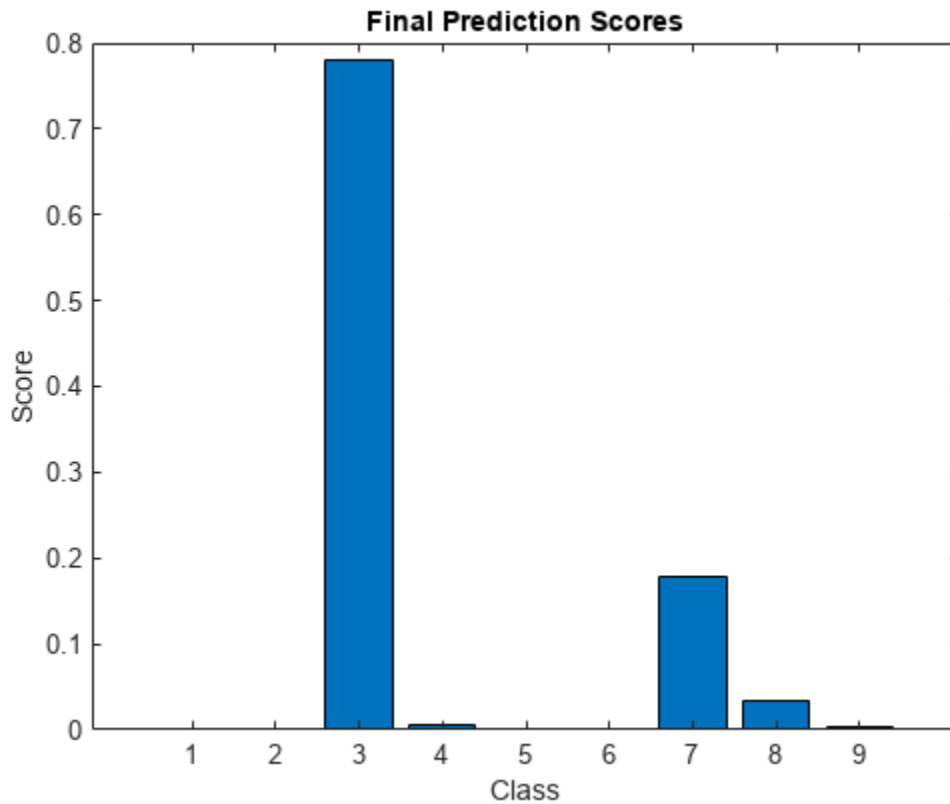
trueLabel = categorical
           3

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.

**Tip** To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

**X1, . . . , XN — Numeric or cell arrays for networks with multiple inputs**

numeric array | cell array

Numeric or cell arrays for networks with multiple inputs.

For sequence predictor input, the input must be a numeric array representing a single sequence or a cell array of sequences, where the format of the predictors match the formats described in the `sequences` argument description. For image and feature predictor input, the input must be a numeric array and the format of the predictors must match the one of the following:

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.
Feature data	$c$ -by-1 column vectors, where $c$ is the number of features.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To input complex-valued data into a network, the `SplitComplexInputs` option of the input layer must be 1.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

### **mixed** – Mixed data

`TransformedDatastore` | `CombinedDatastore` | custom mini-batch datastore

Mixed data, specified as one of the following.

Data Type	Description	Example Usage
<code>TransformedDatastore</code>	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Transform outputs of datastores not supported by <code>predictAndUpdateState</code> so they have the required format.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
<code>CombinedDatastore</code>	Datastore that reads from two or more underlying datastores	<ul style="list-style-type: none"> <li>• Make predictions using networks with multiple inputs.</li> <li>• Combine predictors from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data	<p>Make predictions using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>

You can use other built-in datastores for making predictions 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 `predictAndUpdateState`. For more information, see “Datastores for Deep Learning”.

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.

Datastore Output	Example Output
<p>Cell array with <code>numInputs</code> columns, where <code>numInputs</code> is the number of network inputs.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the network.</p>	<pre>data = read(ds) data =     4x3 cell array     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}     {12x50 double}    {28x1 double}</pre>

For sequence predictor input, the input must be a numeric array representing a single sequence or a cell array of sequences, where the format of the predictors match the formats described in the `sequences` argument description. For image and feature predictor input, the input must be a numeric array and the format of the predictors must match the one of the following:

Data	Format
2-D images	<i>h</i> -by- <i>w</i> -by- <i>c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively.
3-D images	<i>h</i> -by- <i>w</i> -by- <i>d</i> -by- <i>c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> are the height, width, depth, and number of channels of the images, respectively.
Feature data	<i>c</i> -by-1 column vectors, where <i>c</i> is the number of features.

For an example showing how to train a network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip** To convert a numeric array to a datastore, use `ArrayDatastore`.

---

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `[updatedNet,Y] = 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 you make 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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.

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.

### **ExecutionEnvironment — Hardware resource**

`"auto"` (default) | `"gpu"` | `"cpu"`

Hardware resource, specified as one of the following:

- `"auto"` — Use a GPU if one is available; otherwise, use the CPU.
- `"gpu"` — Use the GPU. Using a GPU requires a Parallel Computing Toolbox license and a supported GPU device. For information about supported devices, see “GPU Computing Requirements” (Parallel Computing Toolbox). If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- `"cpu"` — Use the CPU.

### **ReturnCategorical — Option to return categorical labels**

`0` (false) (default) | `1` (true)

Option to return categorical labels, specified as `0` (false) or `1` (true).

If `ReturnCategorical` is `1` (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 or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is `'sequence'` for each recurrent 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**

`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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Output Arguments**

### **updatedNet — Updated network**

`SeriesNetwork` object | `DAGNetwork` object

Updated network. `updatedNet` is the same type of network as the input network.

### **Y — Predicted scores or responses**

numeric array | categorical array | cell array

Predicted scores or responses, returned as a numeric array, categorical array, or a cell array. The format of Y depends on the type of problem.

The following table describes the format for classification tasks.

<b>Task</b>	<b>ReturnCategorical</b>	<b>Format</b>
Sequence-to-label classification	0 (false)	$N$ -by- $K$ matrix, where $N$ is the number of observations, and $K$ is the number of classes
	1 (true)	$N$ -by-1 categorical vector, where $N$ is the number of observations
Sequence-to-sequence classification	0 (false)	$N$ -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 the <code>SequenceLength</code> option is applied to each mini-batch independently.
	1 (true)	$N$ -by-1 cell array of categorical sequences, where $N$ is the number of observations. The categorical sequences are categorical vectors with the same number of time steps as the corresponding input sequence after the <code>SequenceLength</code> option is applied to each mini-batch independently.

For sequence-to-sequence classification tasks with one observation, `sequences` can be a matrix. In this case, the format of `Y` depends on the `ReturnCategorical` option:

- If `ReturnCategorical` is 0 (false), then `Y` is a  $K$ -by- $S$  matrix scores, where  $K$  is the number of classes, and  $S$  is the total number of time steps in the corresponding input sequence.
- If `ReturnCategorical` is 1 (true), then `Y` is a  $1$ -by- $S$  categorical vector, where  $S$  is the total number of time steps in the corresponding input sequence.

The following table describes the format for regression tasks.

<b>Task</b>	<b>Format</b>
Sequence-to-one regression	$N$ -by- $R$ matrix, where $N$ is the number of observations and $R$ is the number of responses.

Task	Format
Sequence-to-sequence regression	<p><math>N</math>-by-1 cell array of numeric sequences, where <math>N</math> is the number of observations. The sequences are matrices with <math>R</math> rows, where <math>R</math> is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the <code>SequenceLength</code> option to each mini-batch independently.</p> <p>For sequence-to-sequence problems with one observation, <code>sequences</code> can be a matrix. In this case, <math>Y</math> is a matrix of responses.</p>

### $Y_1, \dots, Y_M$ — Predicted scores or responses of networks with multiple outputs

numeric array | categorical array | cell array

Predicted scores or responses of networks with multiple outputs, returned as numeric arrays, categorical arrays, or cell arrays.

Each output  $Y_j$  corresponds to the network output `recNet.OutputNames(j)` and has format as described in the  $Y$  output argument.

## Algorithms

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

### Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Alternatives

For recurrent neural networks with a single classification layer only, you can compute the predicted classes and scores and update the network state using the `classifyAndUpdateState` function.

To compute the activations of a network layer, use the `activations` function. The `activations` function does not update the network state.

To make predictions without updating the network state, use the `classify` function or the `predict` function.

## Version History

Introduced in R2017b

### R2022b: Prediction functions pad mini-batches to length of longest sequence before splitting when you specify SequenceLength option as an integer

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

## 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,Y] = predictAndUpdateState(recNet,sequences)`, where `sequences` is cell array.
  - `[updatedNet,Y] = predictAndUpdateState(recNet,sequences,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 arguments are supported for code generation. All name-value arguments must be compile-time constants.
- Only the "longest" and "shortest" options of the `SequenceLength` name-value argument 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,Y] = predictAndUpdateState(recNet,sequences)`, where `sequences` is cell array or numeric array.
  - `[updatedNet,Y] = predictAndUpdateState(recNet,sequences,Name=Value)`
- GPU code generation for the `predictAndUpdateState` function is only supported for recurrent neural networks targeting cuDNN and TensorRT libraries.
- GPU code generation does not support `gpuArray` inputs to the `predictAndUpdateState` function.
- 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 arguments are supported for code generation. All name-value pairs must be compile-time constants.
- Only the "longest" and "shortest" options of the `SequenceLength` name-value argument is supported for code generation.

### GPU Arrays

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

- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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 Neural Networks”  
“Deep Learning in MATLAB”

## 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. When the datastore `auimds` reads a full batch of data, the table has `MiniBatchSize` rows. For the last batch of data in the datastore, if `numObservations` is not cleanly divisible by `MiniBatchSize`, then `read` returns a partial batch containing all of the remaining observations.

The first column of the table, "input", returns the data for each image. The table has a second column, "response", when `auimds` reads data of these types:

- Image files containing labeled image data, when the file names are specified by an image datastore
- A numeric array containing categorical labels or numeric responses
- A table that includes one or more responses

#### **info** — Information about read data

structure array

Information about read data, returned as a structure array. The structure array can contain these fields.



Field Name	Description
Filename	Cell array of character vectors. Each element is a fully resolved path containing the path string, name of the file, and file extension for the corresponding image. This field is present when <code>auimds</code> reads data from files specified by an image datastore.
FileSize	Numeric vector. Each element is the total file size, in bytes, of the corresponding image. For MAT-files, <code>FileSize</code> is the total number of key-value pairs in the file. This field is present when <code>auimds</code> reads data from files specified by an image datastore.
Label	Categorical vector. Each element is the category label of the corresponding image. This field is present when <code>auimds</code> reads labeled data from files specified by an image datastore.
MiniBatchIndices	Numeric vector. Each element is the index of the input image in the array. This field is present when <code>auimds</code> reads data from numeric arrays.

## Version History

Introduced in R2018a

### See Also

`read (Datastore)` | `readByIndex` | `readall`

## 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.

## Version History

Introduced in R2018a

## See Also

read | readall | partitionByIndex

## recordMetrics

**Package:** experiments

Record metric values in experiment results table and training plot

### Syntax

```
recordMetrics(monitor, step, metricName=metricValue)
recordMetrics(monitor,
step, metricName1=metricValue1, ..., metricNameN=metricValueN)
recordMetrics(monitor, step, metricStructure)
```

### Description

`recordMetrics(monitor, step, metricName=metricValue)` records the specified metric value for a trial in the **Experiment Manager** results table and training plot.

`recordMetrics(monitor, step, metricName1=metricValue1, ..., metricNameN=metricValueN)` records multiple metric values for a trial.

`recordMetrics(monitor, step, metricStructure)` records the metric values specified by the structure `metricStructure`.

### 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 = 100 * (iteration/numIterations);
```

### 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.

### step — Custom training loop step

numeric scalar | `dlarray`

Custom training loop step, such as the iteration or epoch number, specified as a numeric scalar or `dlarray` object. Experiment Manager uses this value as the x-coordinate in the training plot.

### 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`

### metricValue — Metric value

numeric scalar | `dlarray`

Metric value, specified as a numeric scalar or `dlarray` object. Experiment Manager uses this value as the y-coordinate in the training plot.

### metricStructure — 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`.

## Version History

Introduced in R2021a

## See Also

### Apps

**Experiment Manager**

### Objects

`experiments.Monitor`

### Functions

`groupSubPlot` | `struct` | `updateInfo`

### Topics

“Use Bayesian Optimization in Custom Training Experiments”

“Run a Custom Training Experiment for Image Comparison”

“Use Experiment Manager to Train Generative Adversarial Networks (GANs)”

“Custom Training with Multiple GPUs in Experiment Manager”

# recordMetrics

**Package:** deep

Record metric values for custom training loops

## Syntax

```
recordMetrics(monitor, step, metricName=metricValue)
recordMetrics(monitor,
step, metricName1=metricValue1, ..., metricNameN=metricValueN)
recordMetrics(monitor, step, metricStructure)
```

## Description

`recordMetrics(monitor, step, metricName=metricValue)` records the specified metric value in the training plot and the `MetricData` property of the `TrainingProgressMonitor` object `monitor`.

`recordMetrics(monitor, step, metricName1=metricValue1, ..., metricNameN=metricValueN)` records multiple metric values.

`recordMetrics(monitor, step, metricStructure)` records the metric values specified by the structure `metricStructure`.

## Examples

### Track Progress and Produce Training Plots

Use a `TrainingProgressMonitor` object to track training progress and produce training plots for custom training loops.

Create a `TrainingProgressMonitor` object. The monitor automatically tracks the start time and the elapsed time. The timer starts when you create the object.

---

**Tip** To ensure that the elapsed time accurately reflects the training time, make sure you create the `TrainingProgressMonitor` object close to the start of your custom training loop.

---

```
monitor = trainingProgressMonitor;
```

Before you start the training, specify names for the information and metric values.

```
monitor.Info = ["LearningRate", "Epoch", "Iteration"];
monitor.Metrics = ["TrainingLoss", "ValidationLoss", "TrainingAccuracy", "ValidationAccuracy"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot, and group the training and validation accuracy in the same plot.

```
monitor.XLabel = "Iteration";  
groupSubPlot(monitor, "Loss", ["TrainingLoss", "ValidationLoss"]);  
groupSubPlot(monitor, "Accuracy", ["TrainingAccuracy", "ValidationAccuracy"]);
```

During training:

- Evaluate the **Stop** property at the start of each step in your custom training loop. When you click the **Stop** button in the Training Progress window, the **Stop** property changes to 1. Training stops if your training loop exits when the **Stop** property is 1.
- Update the information values. The updated values appear in the Training Progress window.
- Record the metric values. The recorded values appear in the training plot.
- Update the training progress percentage based on the fraction of iterations completed.

---

**Note** The following example code is a template. You must edit this training loop to compute your metric and information values. For a complete example that you can run in MATLAB, see “Monitor Custom Training Loop Progress During Training”.

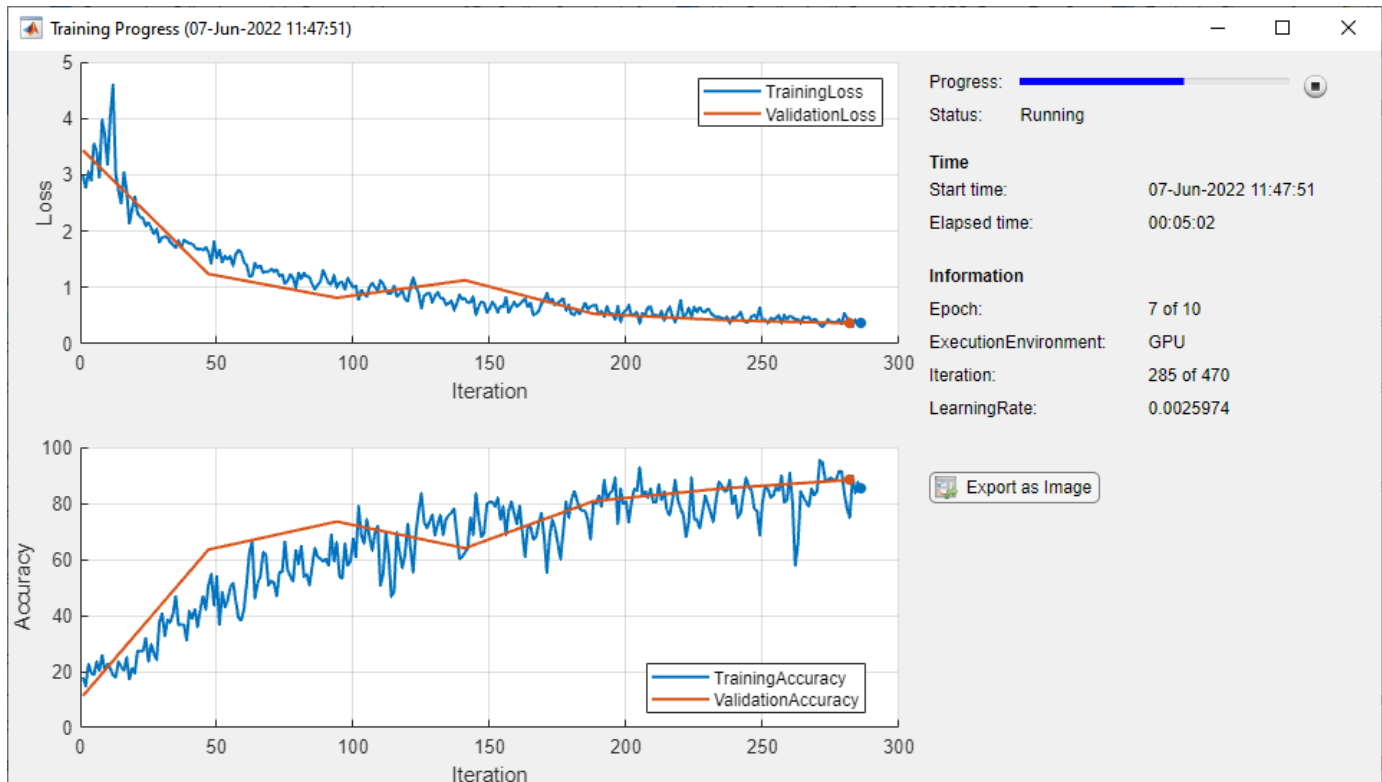
---

```
epoch = 0;  
iteration = 0;  
  
monitor.Status = "Running";  
  
while epoch < maxEpochs && ~monitor.Stop  
    epoch = epoch + 1;  
  
    while hasData(mbq) && ~monitor.Stop  
        iteration = iteration + 1;  
  
        % Add code to calculate metric and information values.  
        % lossTrain = ...  
  
        updateInfo(monitor, ...  
            LearningRate=learnRate, ...  
            Epoch=string(epoch) + " of " + string(maxEpochs), ...  
            Iteration=string(iteration) + " of " + string(numIterations));  
  
        recordMetrics(monitor, iteration, ...  
            TrainingLoss=lossTrain, ...  
            TrainingAccuracy=accuracyTrain, ...  
            ValidationLoss=lossValidation, ...  
            ValidationAccuracy=accuracyValidation);  
  
        monitor.Progress = 100*iteration/numIterations;  
    end  
end
```

The Training Progress window shows animated plots of the metrics, and the information values, training progress bar, and elapsed time.

- The training plots update each time you call `recordMetrics`.
- The values under **Information** update each time you call `updateInfo`.
- The elapsed time updates each time you call `recordMetrics` or `updateInfo`, and when you update the property.





## Specify Metric Values by Using Structure

Use a structure to record metric values.

```
structure.TrainingLoss = trainingLoss;
structure.ValidationLoss = validationLoss;
recordMetrics(monitor, iteration, structure);
```

## Input Arguments

### monitor — Training progress monitor

TrainingProgressMonitor object

Training progress monitor, specified as a TrainingProgressMonitor object.

### step — Custom training loop step

numeric scalar | dlarray object

Custom training loop step, such as the iteration or epoch number, specified as a numeric scalar or dlarray object. The software uses this value as the x-coordinate in the training plot.

### metricName — Metric name

string scalar | character vector

Metric name, specified as a string scalar or character vector. This input must be an element of the Metrics property of monitor.

Data Types: char | string | cell

**metricValue — Metric value**

numeric scalar | dIarray object

Metric value, specified as a numeric scalar or dIarray object. The software uses this value as the y-coordinate in the training plot.

**metricStructure — Metric names and values**

structure

Metric names and values, specified as a structure. Names must be elements of the `Metrics` property of `monitor` and can appear in any order in the structure.

Example: `struct(TrainingLoss=trainingLoss,ValidationLoss=validationLoss)`

Data Types: struct

**Tips**

- The information values appear in the Training Progress window and the training plot shows a record of the metric values. Use information values for text and for numerical values that you want to display in the training window but not in the training plot.
- Use the `groupSubPlot` function to group multiple metrics in a single training subplot.

**Version History**

Introduced in R2022b

**See Also**

`trainingProgressMonitor` | `updateInfo` | `groupSubPlot`

**Topics**

“Monitor Custom Training Loop Progress”

“Train Network Using Custom Training Loop”

“Detect Issues During Deep Neural Network Training”

# regressionLayer

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 'rouput'.

```
layer = regressionLayer('Name','rouput')
```

```
layer =
  RegressionOutputLayer with properties:
```

```
    Name: 'rouput'
  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 '' 2-D 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 the name ''.

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.

## Version History

Introduced in R2017a

### 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”

# 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 the name `''`.

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 '' 2-D 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.

## Version History

Introduced in R2017a

### See Also

`trainNetwork` | `regressionLayer` | `classificationLayer` | `fullyConnectedLayer`

**Topics**

“Create Simple Deep Learning Neural Network for Classification”

“Train Convolutional Neural Network for Regression”

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“List of Deep Learning Layers”

## 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 generating data.
```

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** — Mini-batch queue

minibatchqueue object

Mini-batch queue, specified as a minibatchqueue object.

## Version History

**Introduced in R2020b**

## See Also

`hasdata` | `next` | `shuffle` | `minibatchqueue`

## Topics

“Train Deep Learning Model in MATLAB”

“Define Custom Training Loops, Loss Functions, and Networks”

# quantizationDetails

Display quantization details for a neural network

## Syntax

```
qDetails = quantizationDetails(net)
```

## Description

`qDetails = quantizationDetails(net)` returns a 1-by-1 structure array containing the quantization details for your neural network. The data is returned as a structure with the fields:

- `IsQuantized` — Returns 1 (true) if the network is quantized; otherwise, returns 0 (false)
- `TargetLibrary` — Target library for code generation
- `QuantizedLayerNames` — List of quantized layers
- `QuantizedLearnables` — Quantized network learnable parameters

## Examples

### Show Quantization Details

This example shows how to display the quantization details for a neural network.

Load the pretrained network. `net` is a SqueezeNet convolutional neural network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
  Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Use the `quantizationDetails` function to see that the network is not quantized.

```
qDetails_original = quantizationDetails(net)

qDetails_original = struct with fields:
    IsQuantized: 0
    TargetLibrary: ""
  QuantizedLayerNames: [0x0 string]
  QuantizedLearnables: [0x3 table]
```

The `IsQuantized` field returns 0 (false) because the original network uses the single-precision floating-point data type.

Unzip and load the `MerchData` images as an image datastore. Define an `augmentedImageDatastore` object to resize the data for the network, and split the data into calibration and validation data sets to use for quantization.

```
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. Set the execution environment to MATLAB.

```
quantObj = dlquantizer(net,'ExecutionEnvironment','MATLAB');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information.

```
calResults = calibrate(quantObj,aug_calData);
```

Use the `quantize` method to quantize the network object and return a simulatable quantized network.

```
qNet = quantize(quantObj)
```

```
qNet =
Quantized DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
  InputNames: {'data'}
OutputNames: {'new_classoutput'}
```

Use the `quantizationDetails` method to extract quantization details.

Use the `quantizationDetails` method to extract the quantization details.

```
qDetails = quantizationDetails(qNet)

qDetails = struct with fields:
    IsQuantized: 1
    TargetLibrary: "none"
    QuantizedLayerNames: [26x1 string]
    QuantizedLearnables: [52x3 table]
```

Inspect the `QuantizedLayerNames` field to see a list of the quantized layers.

```
qDetails.QuantizedLayerNames
```

```
ans = 26x1 string
    "conv1"
    "fire2-squeeze1x1"
    "fire2-expand1x1"
```

```

"fire2-expand3x3"
"fire3-squeeze1x1"
"fire3-expand1x1"
"fire3-expand3x3"
"fire4-squeeze1x1"
"fire4-expand1x1"
"fire4-expand3x3"
"fire5-squeeze1x1"
"fire5-expand1x1"
"fire5-expand3x3"
"fire6-squeeze1x1"
"fire6-expand1x1"
"fire6-expand3x3"
"fire7-squeeze1x1"
"fire7-expand1x1"
"fire7-expand3x3"
"fire8-squeeze1x1"
"fire8-expand1x1"
"fire8-expand3x3"
"fire9-squeeze1x1"
"fire9-expand1x1"
"fire9-expand3x3"
"new_conv"

```

Inspect the `QuantizedLearnables` field to see the quantized values for learnable parameters in the network.

`qDetails.QuantizedLearnables`

ans=52x3 table

Layer	Parameter	Value
"conv1"	"Weights"	{3x3x3x64 int8 }
"conv1"	"Bias"	{1x1x64 int32}
"fire2-squeeze1x1"	"Weights"	{1x1x64x16 int8 }
"fire2-squeeze1x1"	"Bias"	{1x1x16 int32}
"fire2-expand1x1"	"Weights"	{1x1x16x64 int8 }
"fire2-expand1x1"	"Bias"	{1x1x64 int32}
"fire2-expand3x3"	"Weights"	{3x3x16x64 int8 }
"fire2-expand3x3"	"Bias"	{1x1x64 int32}
"fire3-squeeze1x1"	"Weights"	{1x1x128x16 int8 }
"fire3-squeeze1x1"	"Bias"	{1x1x16 int32}
"fire3-expand1x1"	"Weights"	{1x1x16x64 int8 }
"fire3-expand1x1"	"Bias"	{1x1x64 int32}
"fire3-expand3x3"	"Weights"	{3x3x16x64 int8 }
"fire3-expand3x3"	"Bias"	{1x1x64 int32}
"fire4-squeeze1x1"	"Weights"	{1x1x128x32 int8 }
"fire4-squeeze1x1"	"Bias"	{1x1x32 int32}
:	:	:

## Input Arguments

**net** — Neural network

dlnetwork object | DAGNetwork object | SeriesNetwork object

Quantized neural network specified as a `dlnetwork`, `SeriesNetwork`, or a `DAGNetwork` object.

## Version History

Introduced in R2022a

### R2023a: Display quantization details of quantized `dlnetwork` objects

The `quantizationDetails` function now supports quantized `dlnetwork` objects.

## See Also

### Apps

**Deep Network Quantizer**

### Functions

`dlquantizer` | `dlquantizationOptions` | `calibrate` | `quantize` | `validate` | `estimateNetworkMetrics`

### Topics

“Quantization of Deep Neural Networks”

“Quantize Layers in Object Detectors and Generate CUDA Code”

“Classify Images on FPGA Using Quantized Neural Network” (Deep Learning HDL Toolbox)

“Generate INT8 Code for Deep Learning Network on Raspberry Pi” (MATLAB Coder)



## resetState

Reset state parameters of neural network

### Syntax

```
netUpdated = resetState(net)
```

### Description

`netUpdated = resetState(net)` resets the state parameters of a neural network. Use this function to reset the state of a recurrent neural network such as an LSTM network.

### 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

### **net** — Neural network

SeriesNetwork object | DAGNetwork object | dlnetwork object

Neural network, specified as a SeriesNetwork, DAGNetwork, or dlnetwork object.

The `resetState` function only has an effect if `net` has state parameters (for example, a network with at least one recurrent layer such as an LSTM layer). If the input network does not have state parameters, then the function has no effect and returns the input network.

## Output Arguments

### **netUpdated** — Updated network

SeriesNetwork object | DAGNetwork object | dlnetwork object

Updated network, returned as a network of the same type as the input network.

The `resetState` function only has an effect if `net` has state parameters (for example, a network with at least one recurrent layer such as an LSTM layer). If the input network does not have state parameters, then the function has no effect and returns the input network.

## Version History

**Introduced in R2017b**

## 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 targeting cuDNN and TensorRT libraries.

## See Also

`sequenceInputLayer` | `lstmLayer` | `biLstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState`

### Topics

"Sequence Classification Using Deep Learning"  
"Visualize Activations of LSTM Network"  
"Long Short-Term Memory Neural Networks"  
"Specify Layers of Convolutional Neural Network"  
"Set Up Parameters and Train Convolutional Neural Network"  
"Deep Learning in MATLAB"

## rmspropupdate

Update parameters using root mean squared propagation (RMSProp)

### Syntax

```
[netUpdated, averageSqGrad] = rmspropupdate(net, 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.
- 

`[netUpdated, averageSqGrad] = rmspropupdate(net, grad, averageSqGrad)` updates the learnable parameters of the network `net` 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,TTrain] = digitTrain4DArrayData;
classes = categories(TTrain);
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], 'Mean', mean(XTrain,4))
    convolution2dLayer(5,20)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers);
```

### Define Model Loss Function

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

### Specify Training Options

Specify the options to use during training.

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

## Train Network

Initialize the squared average gradients.

```
averageSqGrad = [];
```

Calculate the total number of iterations for the training progress monitor.

```
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");
```

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 iteration, display the training progress.

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

Train the network.

```
iteration = 0;  
epoch = 0;
```

```
while epoch < numEpochs && ~monitor.Stop  
    epoch = epoch + 1;  
  
    % Shuffle data.  
    idx = randperm(numel(TTrain));  
    XTrain = XTrain(:,:,,idx);  
    TTrain = TTrain(idx);  
  
    i = 0;  
    while i < numIterationsPerEpoch && ~monitor.Stop  
        i = i + 1;  
        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);  
  
        T = zeros(numClasses,miniBatchSize,"single");  
        for c = 1:numClasses  
            T(c,TTrain(idx)==classes(c)) = 1;  
        end  
  
        % Convert mini-batch of data to a dlarray.  
        X = dlarray(single(X),"SSCB");  
  
        % If training on a GPU, then convert data to a gpuArray.  
        if canUseGPU  
            X = gpuArray(X);  
        end
```

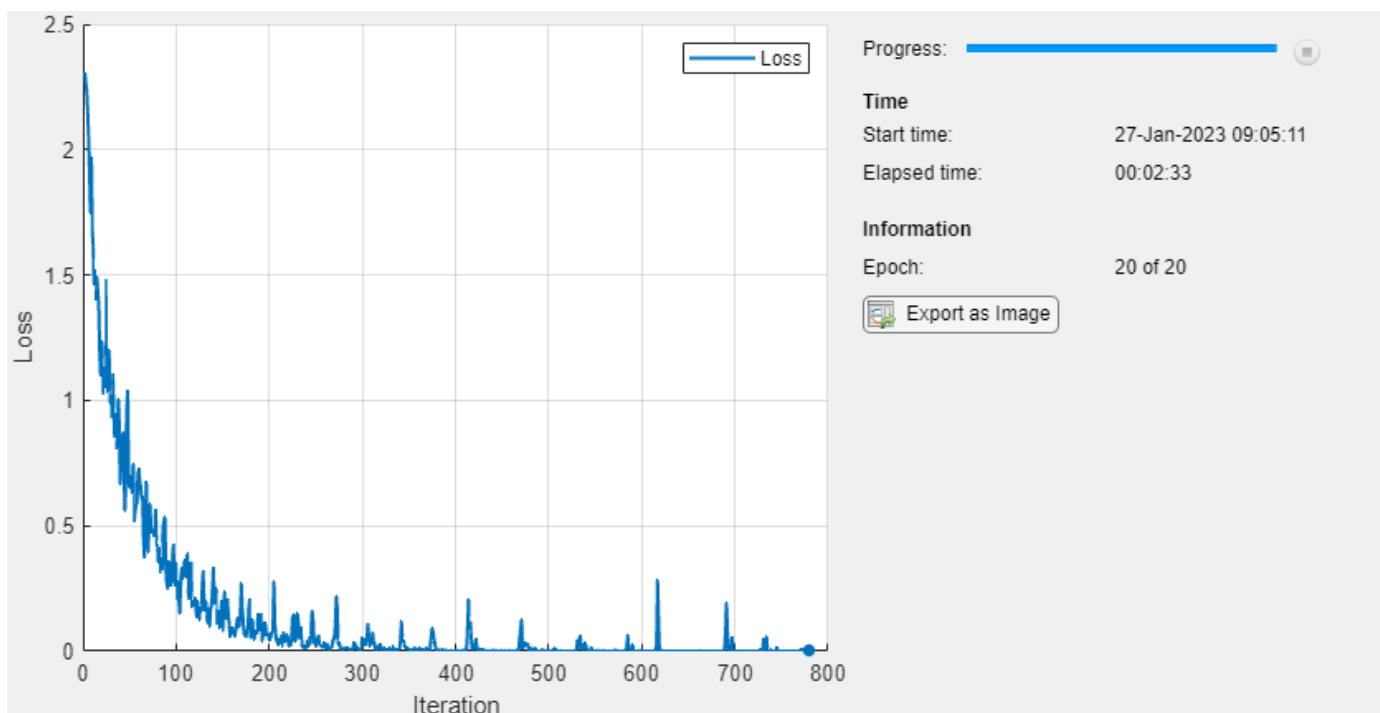
```

% Evaluate the model loss and gradients using dlfeval and the
% modelLoss function.
[loss,gradients] = dlfeval(@modelLoss,net,X,T);

% Update the network parameters using the RMSProp optimizer.
[net,averageSqGrad] = rmspropupdate(net,gradients,averageSqGrad);

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch + " of " + numEpochs);
monitor.Progress = 100 * iteration/numIterations;
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,TTest] = digitTest4DArrayData;
```

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

```

XTest = dlarray(XTest,"SSCB");
if canUseGPU
    XTest = gpuArray(XTest);
end

```

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

```
YTest = predict(net,XTest);  
[~,idx] = max(extractdata(YTest),[],1);  
YTest = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YTest==TTest)
```

```
accuracy = 0.9926
```

### Model Loss Function

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

```
function [loss,gradients] = modelLoss(net,X,T)  
  
Y = forward(net,X);  
  
loss = crossentropy(Y,T);  
  
gradients = dlgradient(loss,net.Learnables);  
  
end
```

## Input Arguments

### **net** — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `Learnables` property of the `dlnetwork` object. `net.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 `net.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 `darray` 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**

`darray` | numeric array | cell array | structure | table

Gradients of the loss, specified as a `darray`, 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`.

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

`[]` | `darray` | numeric array | cell array | structure | table

Moving average of squared parameter gradients, specified as an empty array, a `darray`, 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
<code>net</code>	Table <code>net.Learnables</code> containing <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data type, variables, and ordering as <code>net.Learnables</code> . <code>averageSqGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.
<code>params</code>	<code>dlarray</code>	<code>dlarray</code> with the same data type and ordering as <code>params</code>
	Numeric array	Numeric array with the same data type and ordering as <code>params</code>
	Cell array	Cell array with the same data types, structure, and ordering as <code>params</code>
	Structure	Structure with the same data types, fields, and ordering as <code>params</code>
	Table with <code>Layer</code> , <code>Parameter</code> , and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code> .	Table with the same data types, variables, and ordering as <code>params</code> . <code>averageSqGrad</code> must have a <code>Value</code> 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**

### **netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as a `dlnetwork` object.

The function updates the `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 1-1782 on the `trainingOptions` reference page.

## **Version History**

### **Introduced in R2019b**

#### **R2020a: 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:

```
[net,averageSqGrad] = rmspropupdate(net,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”

# relu

Apply rectified linear unit activation

## Syntax

$Y = \text{relu}(X)$

## 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 `darray` data. If you want to apply the ReLU activation within a `layerGraph` object or `Layer` array, use the following layer:

- `reluLayer`
- 

$Y = \text{relu}(X)$  computes the ReLU activation of the input  $X$  by applying a threshold operation. All values in  $X$  that are less than zero are set to zero.

## Examples

### Apply ReLU Operation

Create a formatted `darray` 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);
X = darray(X,"SSCB");
```

View the size and format of the input data.

```
size(X)
```

```
ans = 1x4
```

```
    28    28     3   128
```

```
dims(X)
```

```
ans =  
'SSCB'
```

Apply the ReLU operation using the `relu` function.

```
Y = relu(X);
```

View the size and format of the output.

```
size(Y)
```

```
ans = 1×4
```

```
    28    28     3   128
```

```
dims(Y)
```

```
ans =  
'SSCB'
```

## Input Arguments

### **X** — Input data

`darray` object

Input data, specified as a formatted or unformatted `darray` object.

## Output Arguments

### **Y** — ReLU activations

`darray`

ReLU activations, returned as a `darray`. The output `Y` has the same underlying data type as the input `X`.

If the input data `X` is a formatted `darray`, `Y` has the same dimension format as `X`. If the input data is not a formatted `darray`, `Y` is an unformatted `darray` with the same dimension order as the input data.

## Version History

**Introduced in R2019b**

## 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 `X` is a `darray` 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` | `gelu` | `dlgradient` | `dlfeval`

## **Topics**

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Network Using Model Function”

“List of Functions with `dlarray` Support”

# 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 the name ' '.

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 '' 2-D Convolution     20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
```

```
3 '' ReLU ReLU
4 '' 2-D 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”.

## Version History

**Introduced in R2016a**

### 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 Neural 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”

## removeLayers

**Package:** `nnet.cnn`

Remove layers from layer graph or network

### Syntax

```
lgraphUpdated = removeLayers(lgraph, layerNames)
netUpdated = removeLayers(net, layerNames)
```

### Description

`lgraphUpdated = removeLayers(lgraph, layerNames)` removes the layers specified by `layerNames` from the layer graph `lgraph`. The function also removes any connections to the removed layers.

`netUpdated = removeLayers(net, layerNames)` removes the layers specified by `layerNames` from the `dlnetwork` object `net`. The function also removes any connections to the removed layers.

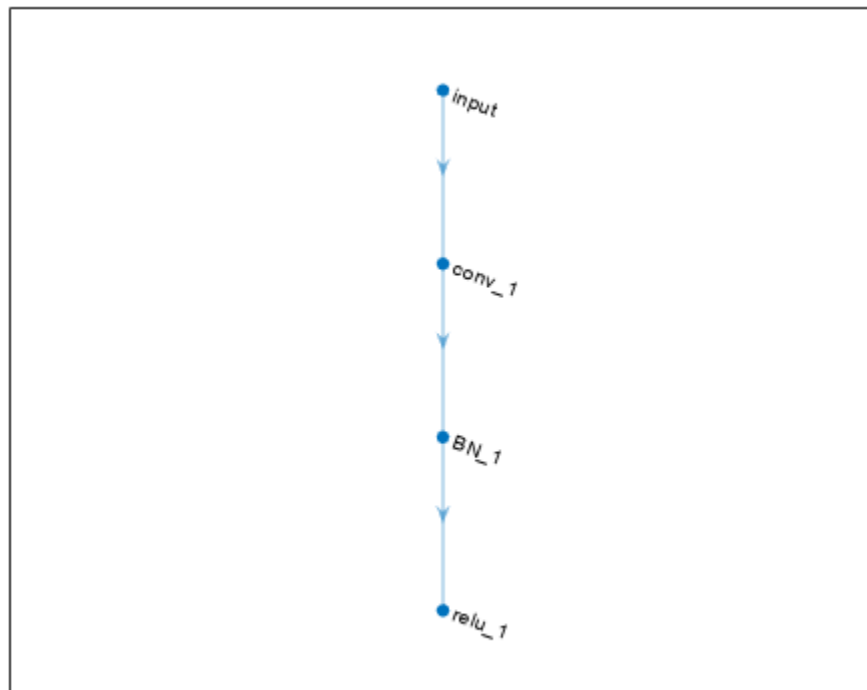
### Examples

#### Remove Layer from Layer Graph

Create a layer graph from an array of layers.

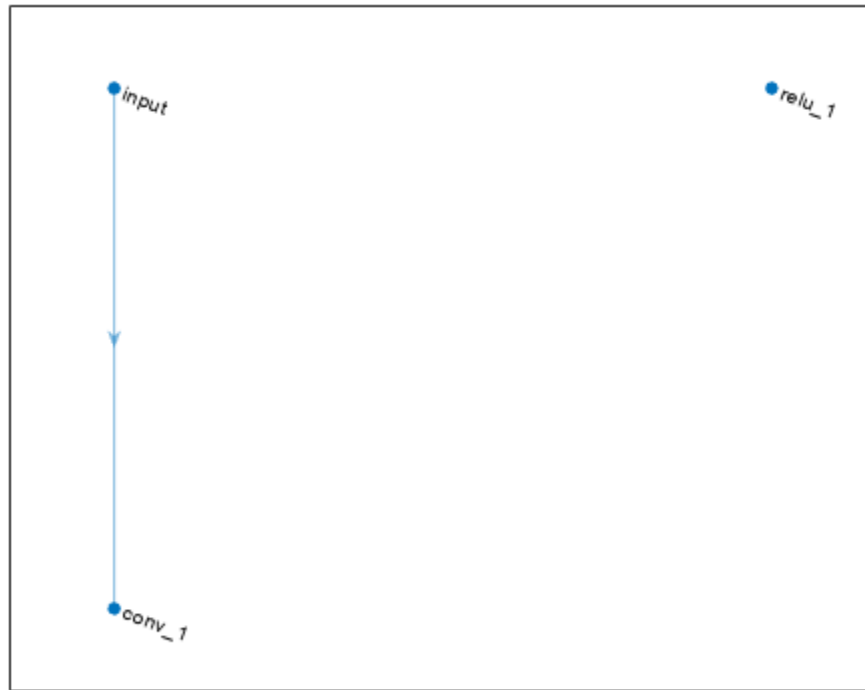
```
layers = [
    imageInputLayer([28 28 1], 'Name', 'input')
    convolution2dLayer(3, 16, 'Padding', 'same', 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'BN_1')
    reluLayer('Name', 'relu_1')];

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`.

### **net** — Neural network

dlnetwork object

Neural network, specified as a dlnetwork object.

### **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

### **lgraphUpdated** — Updated layer graph

LayerGraph object

Updated layer graph, returned as a LayerGraph object.

### **netUpdated** — Updated network

dlnetwork object

Updated network, returned as an uninitialized dlnetwork object.

To initialize the learnable parameters of a dlnetwork object, use the `initialize` function.

The `removeLayers` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

**Introduced in R2017b**

### See Also

`layerGraph` | `addLayers` | `replaceLayer` | `connectLayers` | `disconnectLayers` | `plot` | `assembleNetwork` | `dlnetwork`

### Topics

“Train Deep Learning Network to Classify New Images”

“Classify Videos Using Deep Learning”

## 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(Var...
Vars.fc_1 = dlconv(Vars.relu1001, weights, bias, 'Stride', stride, 'DilationFactor', dilationFactor...

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

% Conv:
[weights, bias, stride, dilationFactor, padding, dataFormat, NumDims.fc_3] = prepareConvArgs(Var...
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 darray]
batchnorm_B: [20×1 darray]
fc_1_W: [24×24×20×20 darray]
fc_1_B: [20×1 darray]
fc_3_W: [1×1×20×10 darray]
fc_3_B: [10×1 darray]
```

`params.Nonlearnables`

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

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', dilationFactor);

% 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.

## Version History

**Introduced in R2020b**

### **See Also**

[importONNXFunction](#) | [ONNXParameters](#) | [addParameter](#)

# replaceLayer

**Package:** `nnet.cnn`

Replace layer in layer graph or network

## Syntax

```
lgraphUpdated = replaceLayer(lgraph, layerName, larray)
netUpdated = replaceLayer(net, layerName, larray)
___ = replaceLayer( ___, 'ReconnectBy', mode)
```

## Description

`lgraphUpdated = 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.

`netUpdated = replaceLayer(net, layerName, larray)` replaces the layer `layerName` in the `dlnetwork` object `net` with the layers in `larray`.

`replaceLayer` connects the layers in `larray` sequentially and connects `larray` into the network.

`___ = replaceLayer( ___, 'ReconnectBy', mode)` additionally specifies the method of reconnecting layers.

## Examples

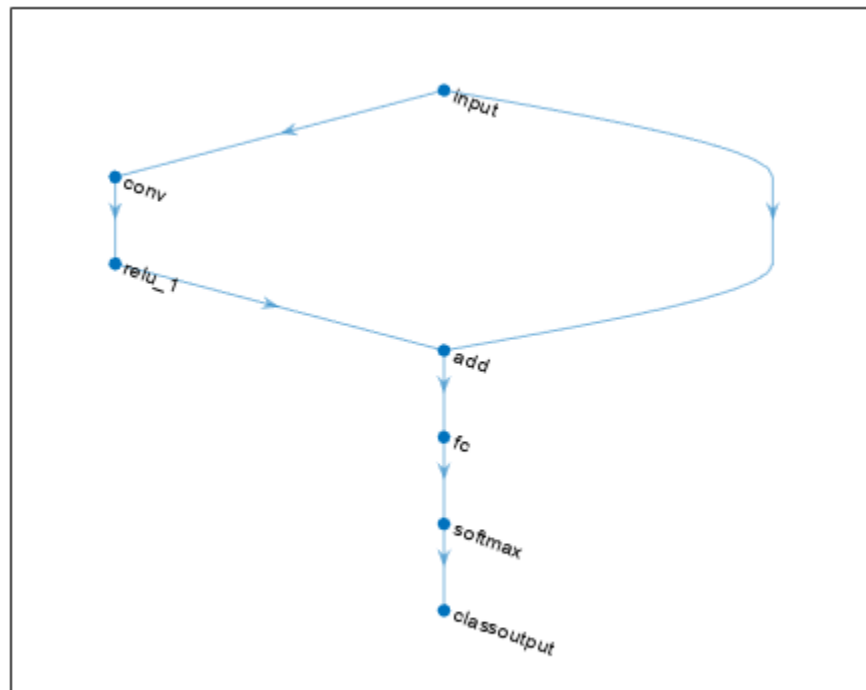
### Replace Layer in Layer Graph

Define a simple network architecture and plot it.

```
layers = [
    imageInputLayer([28 28 1], 'Name', 'input')
    convolution2dLayer(3, 16, 'Padding', 'same')
    reluLayer('Name', 'relu_1')
    additionLayer(2, 'Name', 'add')
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

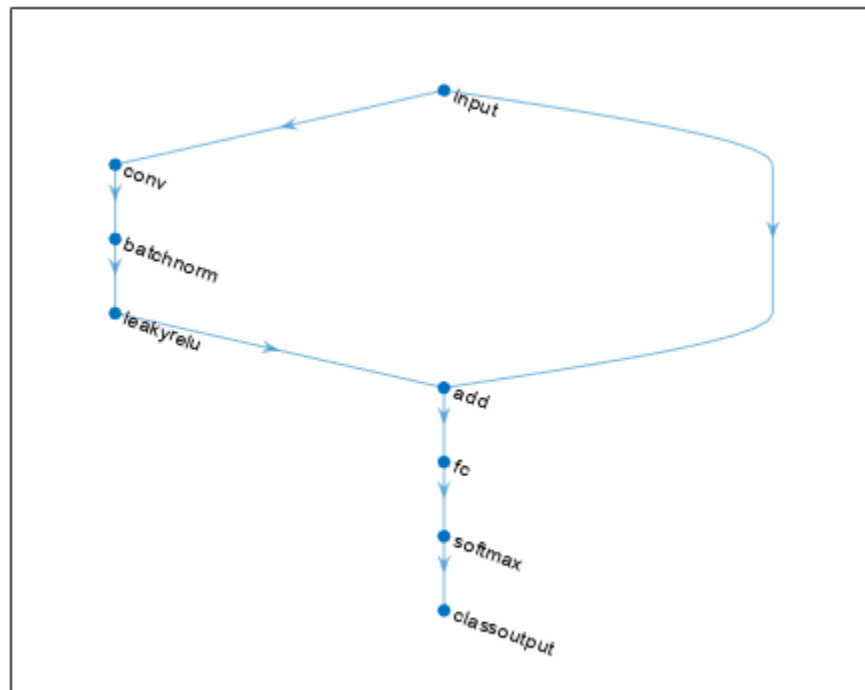
lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph, 'input', 'add/in2');

figure
plot(lgraph)
```



Replace the ReLU layer in the network with a batch normalization layer followed by a leaky ReLU layer.

```
layers = [  
    batchNormalizationLayer  
    leakyReluLayer(0.1)];  
lgraph = replaceLayer(lgraph, 'relu_1', layers);  
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.

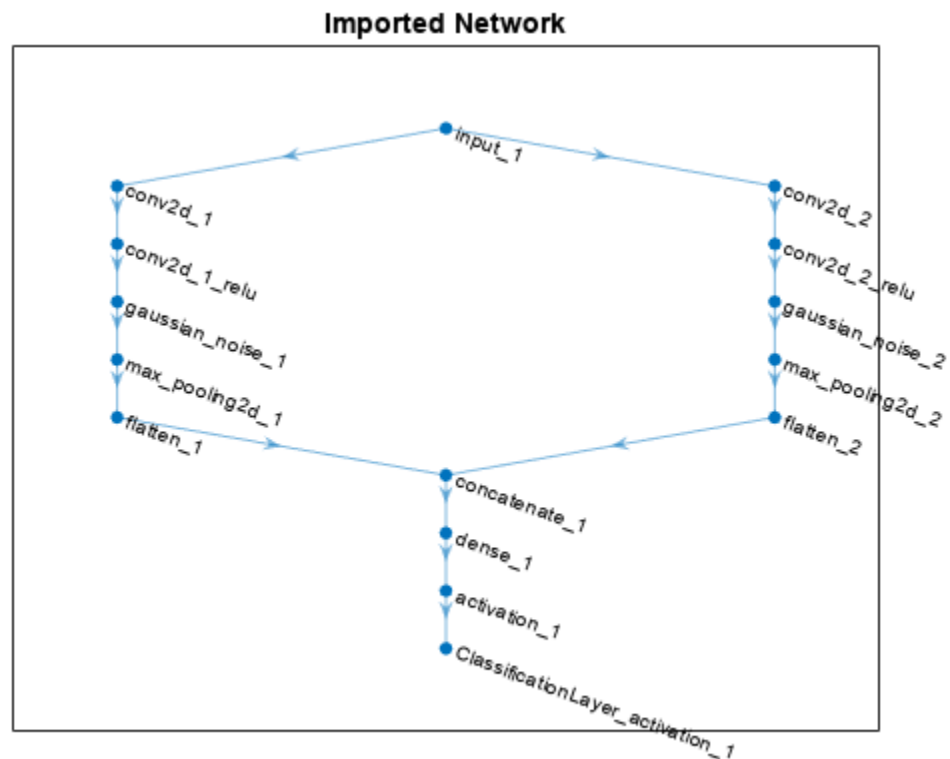
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename, 'ImportWeights', true);
```

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
```

Create two Gaussian noise layers with the same configurations as the imported Keras layers using the helper `gaussianNoiseLayer` function.

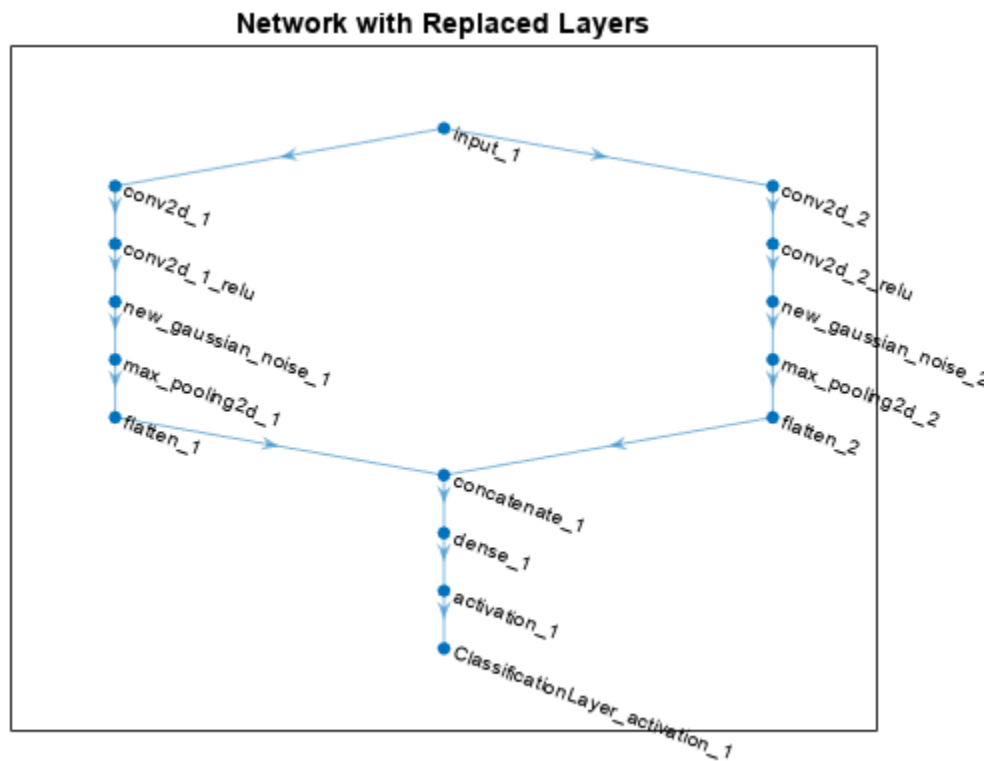
```
gnLayer1 = gaussianNoiseLayer(1.5, 'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7, 'new_gaussian_noise_2');
```

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'	2-D Convolution	20 7x7x1 convolutions with
3	'conv2d_1_relu'	ReLU	ReLU
4	'conv2d_2'	2-D Convolution	20 3x3x1 convolutions with
5	'conv2d_2_relu'	ReLU	ReLU
6	'new_gaussian_noise_1'	Gaussian Noise	Gaussian noise with standa
7	'new_gaussian_noise_2'	Gaussian Noise	Gaussian noise with standa
8	'max_pooling2d_1'	2-D Max Pooling	2x2 max pooling with stride
9	'max_pooling2d_2'	2-D 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`.

### **net** — Neural network

dlnetwork object

Neural network, specified as a dlnetwork object.

### **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

### **lgraphUpdated** — Updated layer graph

LayerGraph object

Updated layer graph, returned as a LayerGraph object.

**netUpdated — Updated network**`dlnetwork` object

Updated network, returned as an uninitialized `dlnetwork` object.

To initialize the learnable parameters of a `dlnetwork` object, use the `initialize` function.

The `replaceLayer` function does not preserve quantization information. If the input network is a quantized network, then the output network does not contain quantization information.

## Version History

**Introduced in R2018b**

### See Also

`layerGraph` | `findPlaceholderLayers` | `PlaceholderLayer` | `connectLayers` | `disconnectLayers` | `addLayers` | `removeLayers` | `assembleNetwork` | `functionLayer` | `dlnetwork`

### Topics

“Deep Learning in MATLAB”

“Pretrained Deep Neural Networks”

“Train Deep Learning Network to Classify New Images”

“Transfer Learning Using Pretrained Network”

## 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 =
```

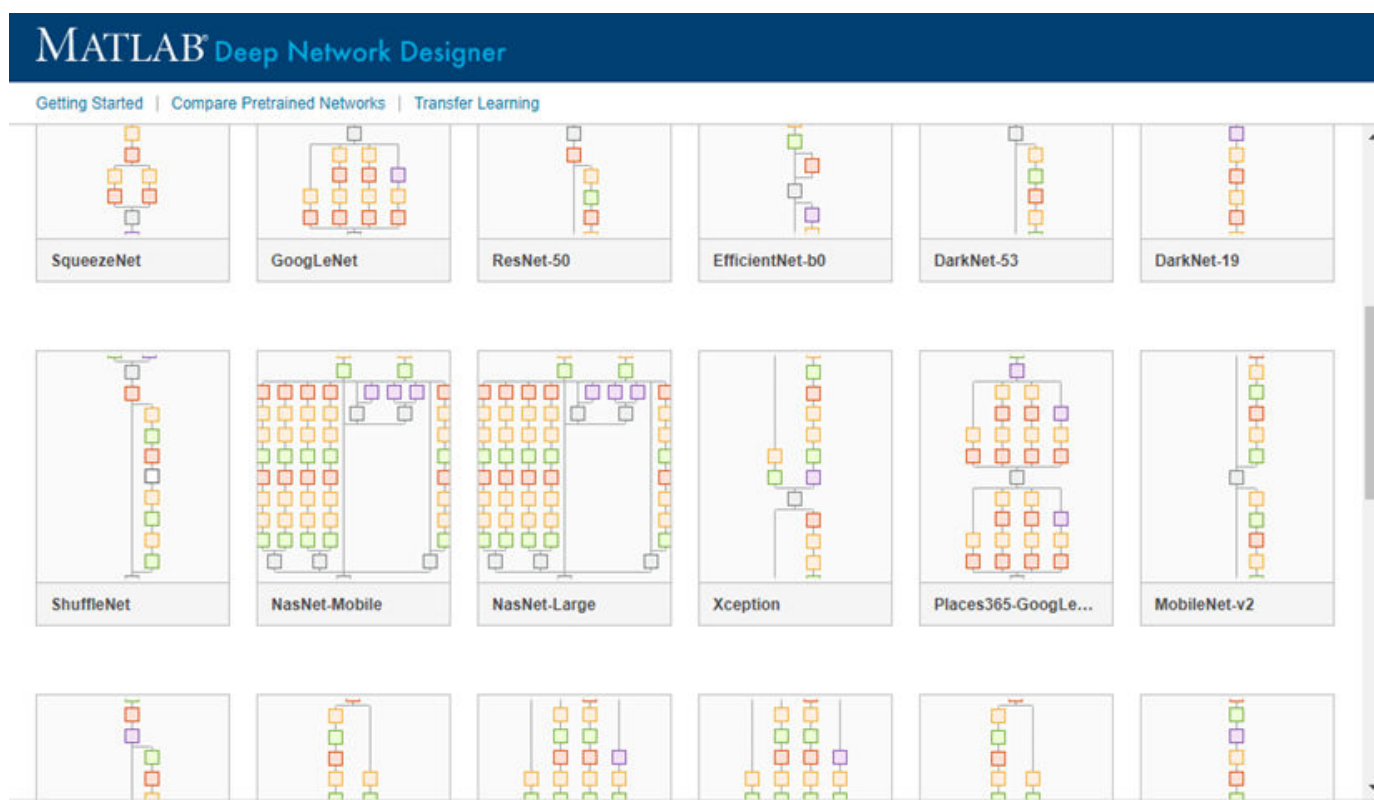
```
DAGNetwork with properties:
```

```
    Layers: [72x1 nnet.cnn.layer.Layer]  
    Connections: [79x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(resnet18)
```

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



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

### Classify Image Using ResNet-18

Load a pretrained ResNet-18 model.

```
net = resnet18;
```

Read a test image using `imread`.

```
I = imread("peppers.png");  
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
```

```
    224    224     3
```

Resize the image to the input size of the network.

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

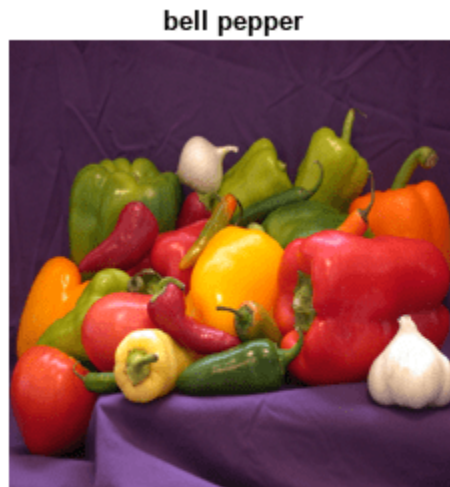
Classify the image using `classify`.

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

```
label = categorical  
    bell pepper
```

Show the image and classification result together.

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



### **Interactive Transfer Learning Using ResNet-18**

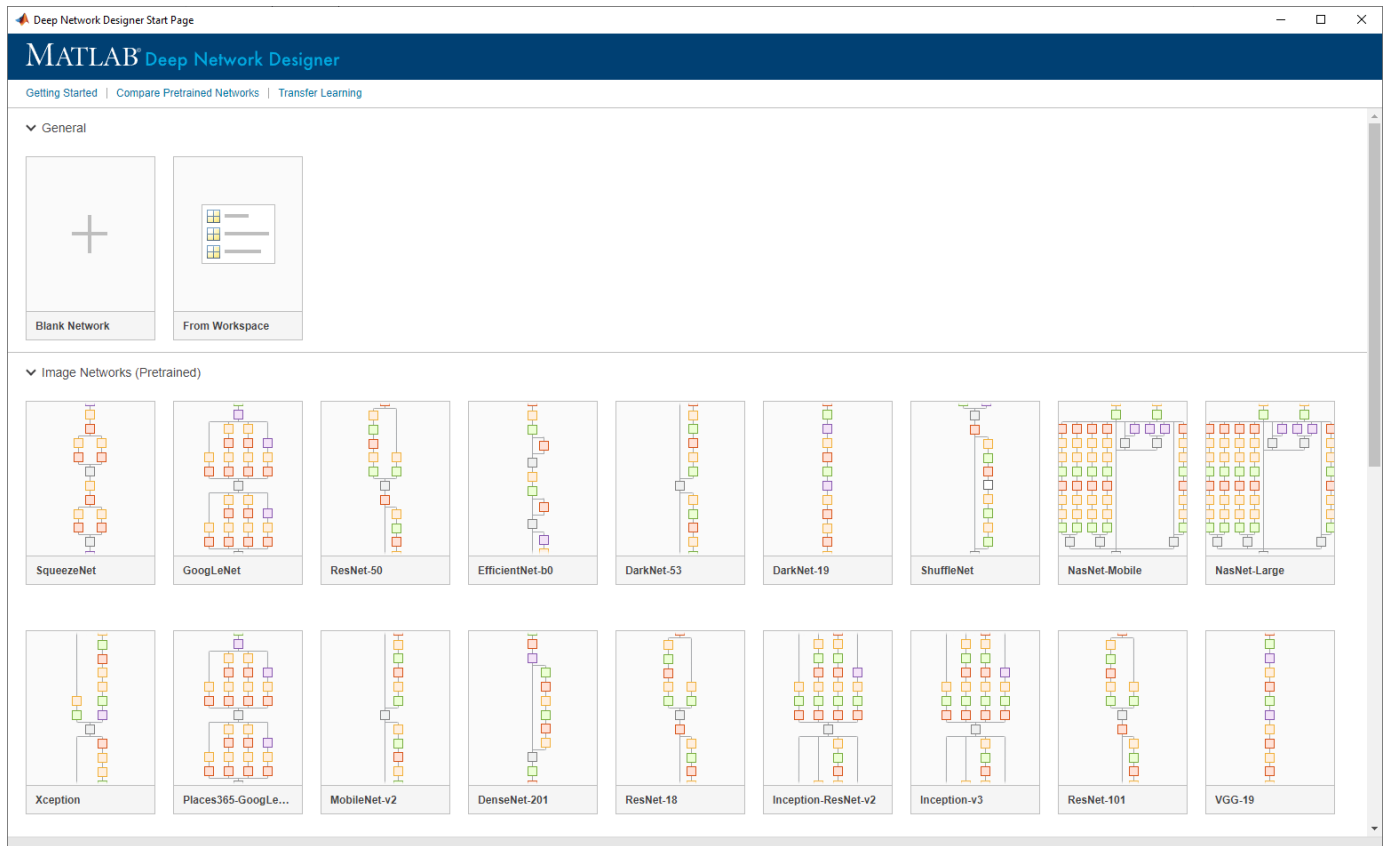
In the workspace, extract the MathWorks Merch data set.

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

Open Deep Network Designer to see available pretrained networks.

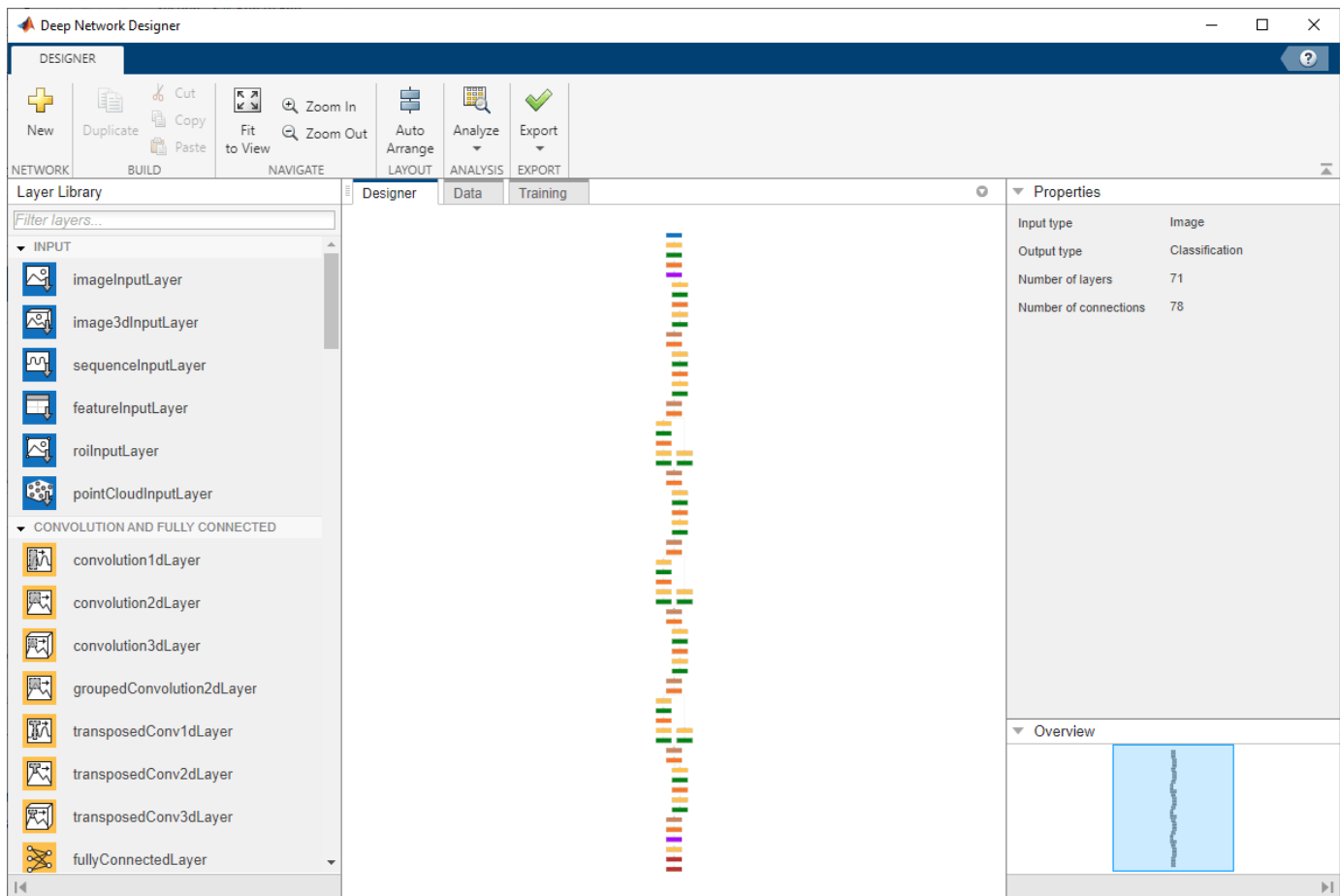
```
deepNetworkDesigner
```

# 1 Deep Learning Functions



Select ResNet-18 and click **Open**. If Deep Learning Toolbox™ Model for ResNet-18 Network is not installed, then the software opens the **Add-On Explorer**.

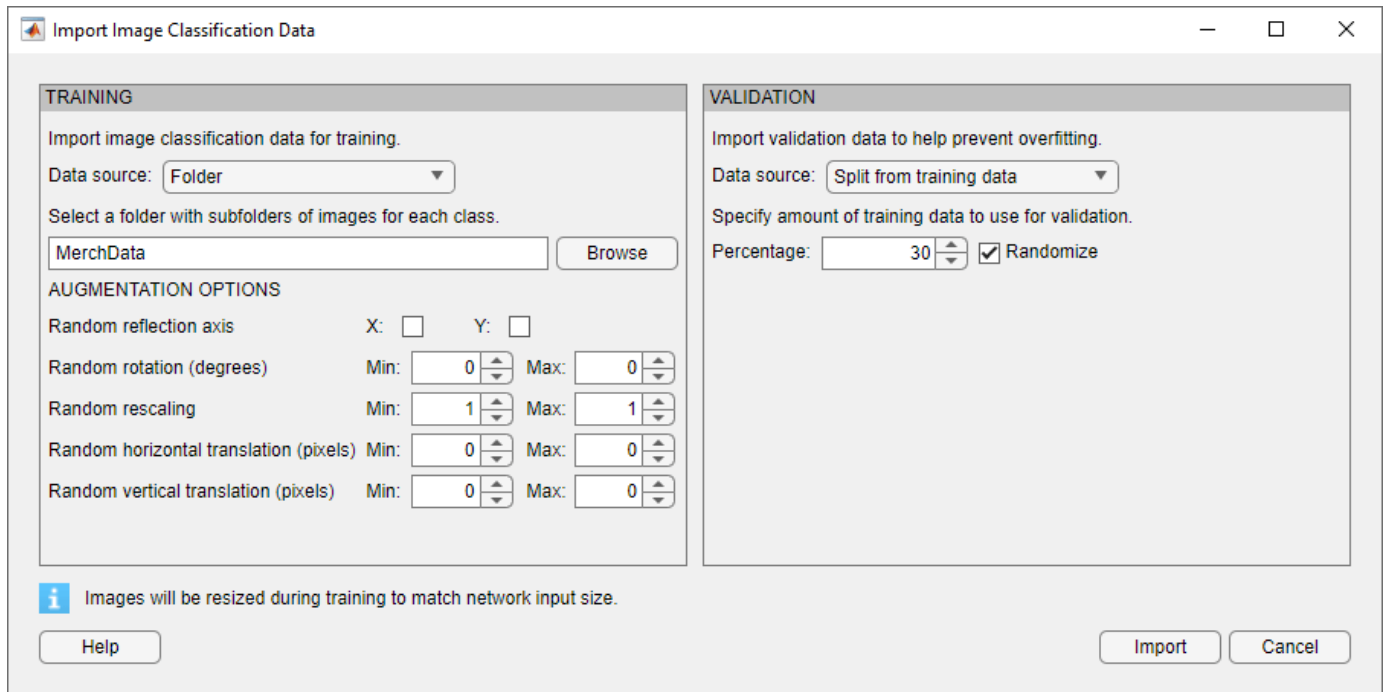




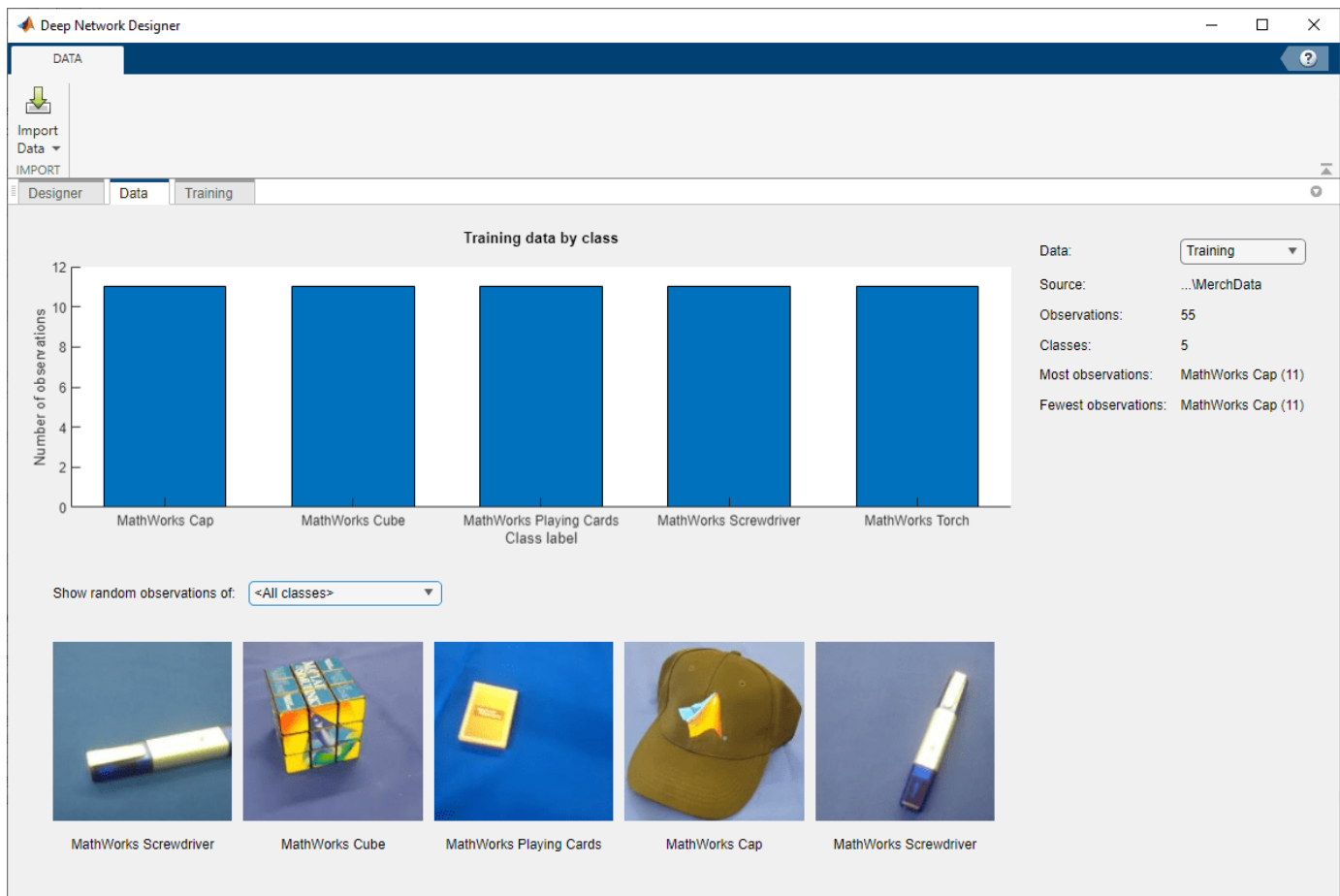
To load the data into Deep Network Designer, on the **Data** tab, click **Import Data > Import Image Classification Data**. In the **Data source** list, select **Folder**. Click **Browse** and select the extracted MerchData folder.

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. Select **Randomize** to randomly assign the specified proportion of images to the validation and training sets.

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



The app shows a summary of the imported data.



To retrain the pretrained network to classify new images, replace the last fully connected layer and the final classification layer with new layers adapted to the new data set.

In the **Designer** pane, scroll to the end of the network and then zoom in. Drag a new `fullyConnectedLayer` onto the canvas. Change **OutputSize** to the number of classes in the new data, in this example, 5.

To make learning faster in the new layers than in the transferred layers, set `WeightLearnRateFactor` and `BiasLearnRateFactor` to 10. Delete the last fully connected layer ('`fc1000`') and connect your new layer instead.

The diagram shows a vertical flow of three layers: a fully connected layer (fc), a softmax layer (prob), and a classification layer (classoutput). The properties panel for the selected 'fullyConnectedLayer' is as follows:

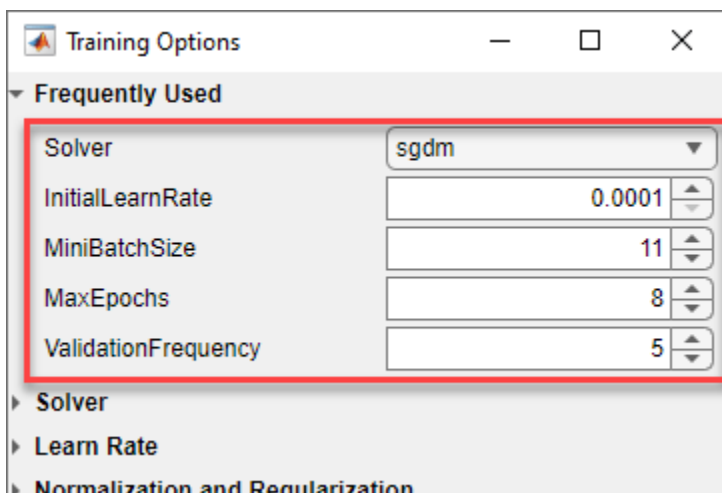
Property	Value
Name	fc
InputSize	auto
OutputSize	5
Weights	[]
Bias	[]
WeightLearnRateFactor	10
WeightL2Factor	1
BiasLearnRateFactor	10

Replace the output layer. Drag a new classificationLayer onto the canvas. Delete the original output layer ('ClassificationLayer\_predictions') and connect your new layer instead.

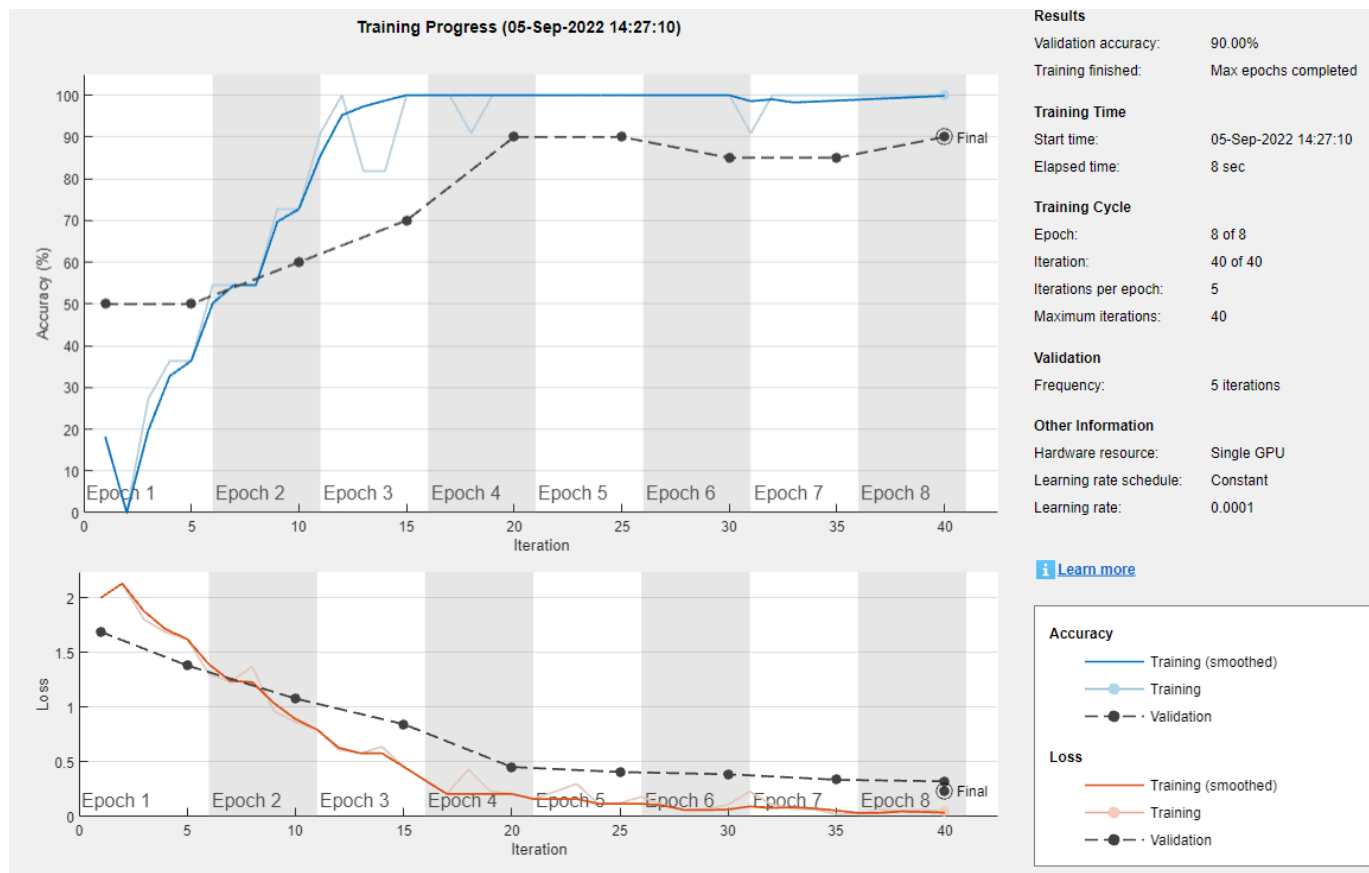
The diagram shows the same vertical flow of layers, but the classification layer (classoutput) is now highlighted in blue. The properties panel for the selected 'classificationLayer' is as follows:

Property	Value
Name	classoutput
Classes	auto
ClassWeights	none
OutputSize	auto
LossFunction	crossentropyex

Specify training options. Select the **Training** tab and click **Training Options**. Set **Solver** to `sgdm`, **InitialLearnRate** to `0.0001`, **MiniBatchSize** to `11`, **MaxEpochs** to `8`, and **ValidationFrequency** to `5`.



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

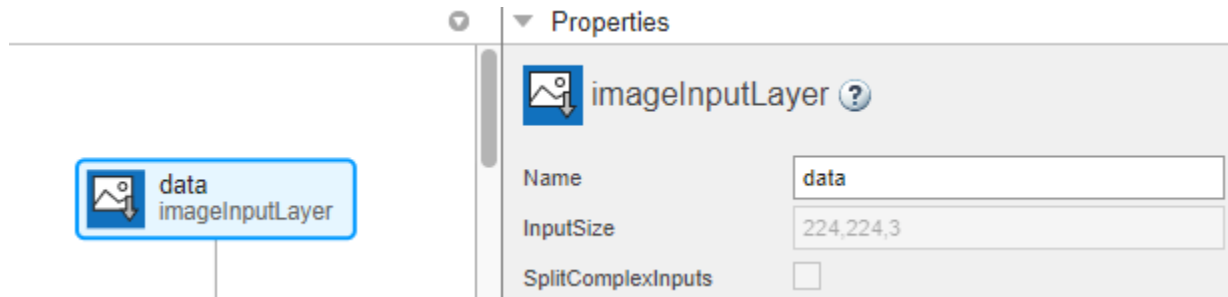


To export the network architecture with the trained weights, on the **Training** tab, select **Export > Export Trained Network and Results**. Deep Network Designer exports the trained network as the variable `trainedNetwork_1`.

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 224-by-224.



Resize the test image to match the network input size.

```
I = imresize(I,[224 224]);
```

Classify the test image using the trained network.

```
YPred = classify(trainedNetwork_1,I);  
imshow(I)  
title("Predicted Class: " + string(YPred));
```



## Programmatic Transfer Learning Using ResNet-18

Load the pretrained ResNet-18 neural network and find the input size.

```
net = resnet18;  
inputSize = net.Layers(1).InputSize;
```

Extract the MathWorks Merch data set. Load the new images as an image datastore.

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

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

Divide the data into training and validation data sets. Specify "randomized" to randomly assign the specified proportion of images to the validation and training sets.

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

Use an augmented image datastore to automatically resize the training images to match the input size of the network.

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

To retrain the pretrained network to classify new images, replace the last fully connected layer and the final classification layer with new layers adapted to the new data set. First, extract the layer graph from the trained network.

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

Replace the fully connected layer with a new fully connected layer that has number of outputs equal to the number of classes. To make learning faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
    WeightLearnRateFactor=10, ...
    BiasLearnRateFactor=10);
```

```
lgraph = replaceLayer(lgraph,"fc1000",newLearnableLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new layer without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

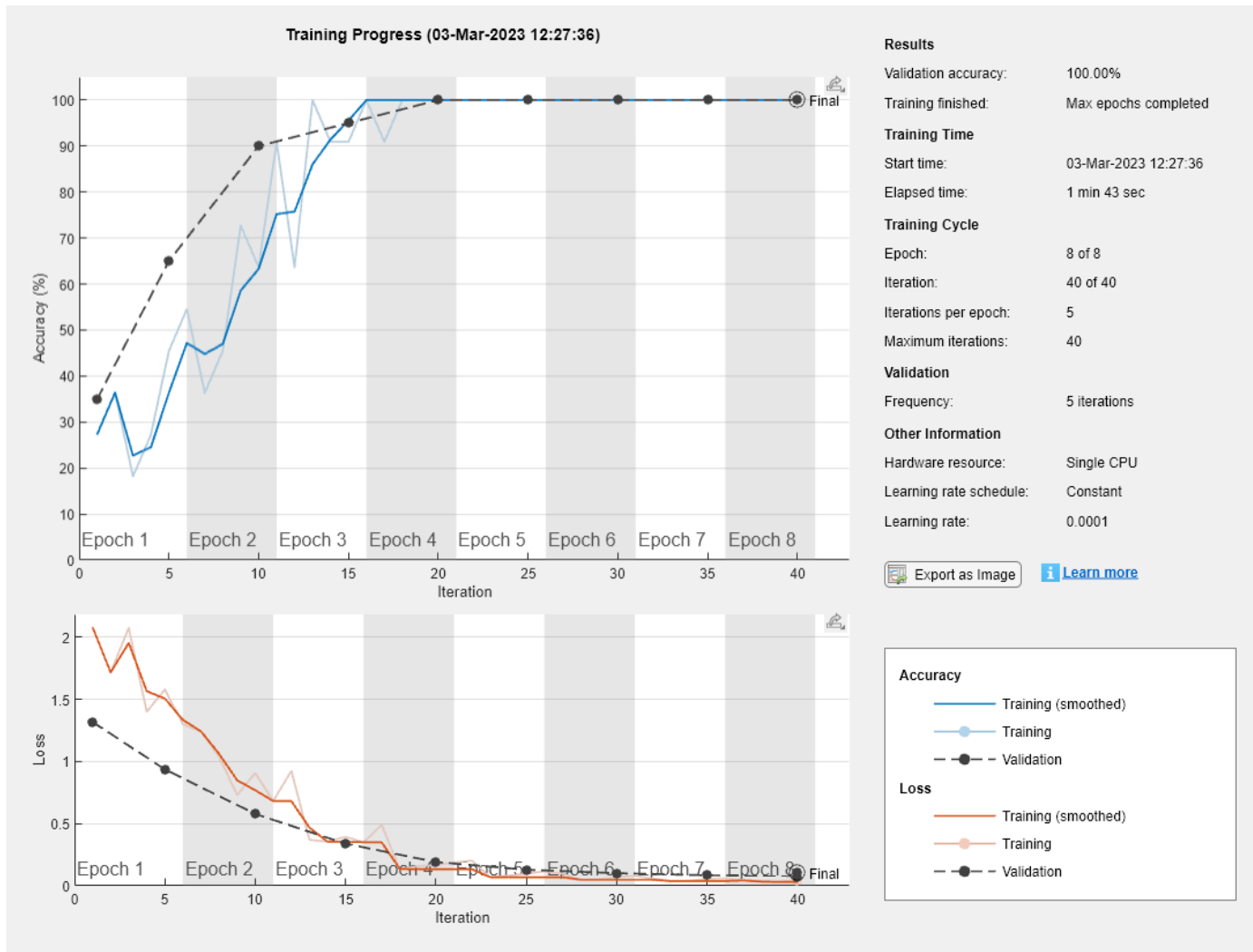
```
newClassLayer = classificationLayer;
lgraph = replaceLayer(lgraph,"ClassificationLayer_predictions",newClassLayer);
```

Specify the training options. To slow down learning in the transferred layers, set the initial learning rate to a small value.

```
options = trainingOptions("sgdm", ...
    InitialLearnRate=0.0001, ...
    MiniBatchSize=11, ...
    MaxEpochs=8, ...
    ValidationData=augimdsValidation, ...
    ValidationFrequency=5, ...
    Verbose=false, ...
    Plots="training-progress");
```

Train the network.

```
trainedNetwork = trainNetwork(augimdsTrain,lgraph,options);
```



Load a new image to classify using the trained network.

```
I = imread("MerchDataTest.jpg");
I = imresize(I,inputSize(1:2));
```

Classify the test image.

```
YPred = classify(trainedNetwork,I);
imshow(I)
title("Predicted class: " + string(YPred));
```



Predicted class: MathWorks Cube



## 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.

## Version History

Introduced in R2018a

## 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”

# 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 neural network trained on more than a million images from the ImageNet database [1]. The pretrained neural network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the neural network has learned rich feature representations for a wide range of images. The neural network has an image input size of 224-by-224. For more pretrained neural 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 neural 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 neural network suitable for image classification tasks, use `resnetLayers`.

---

`net = resnet50` returns a ResNet-50 neural 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 neural network trained on the ImageNet data set. This syntax is equivalent to `net = resnet50`.

`lgraph = resnet50('Weights','none')` returns the untrained ResNet-50 neural 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 =
```

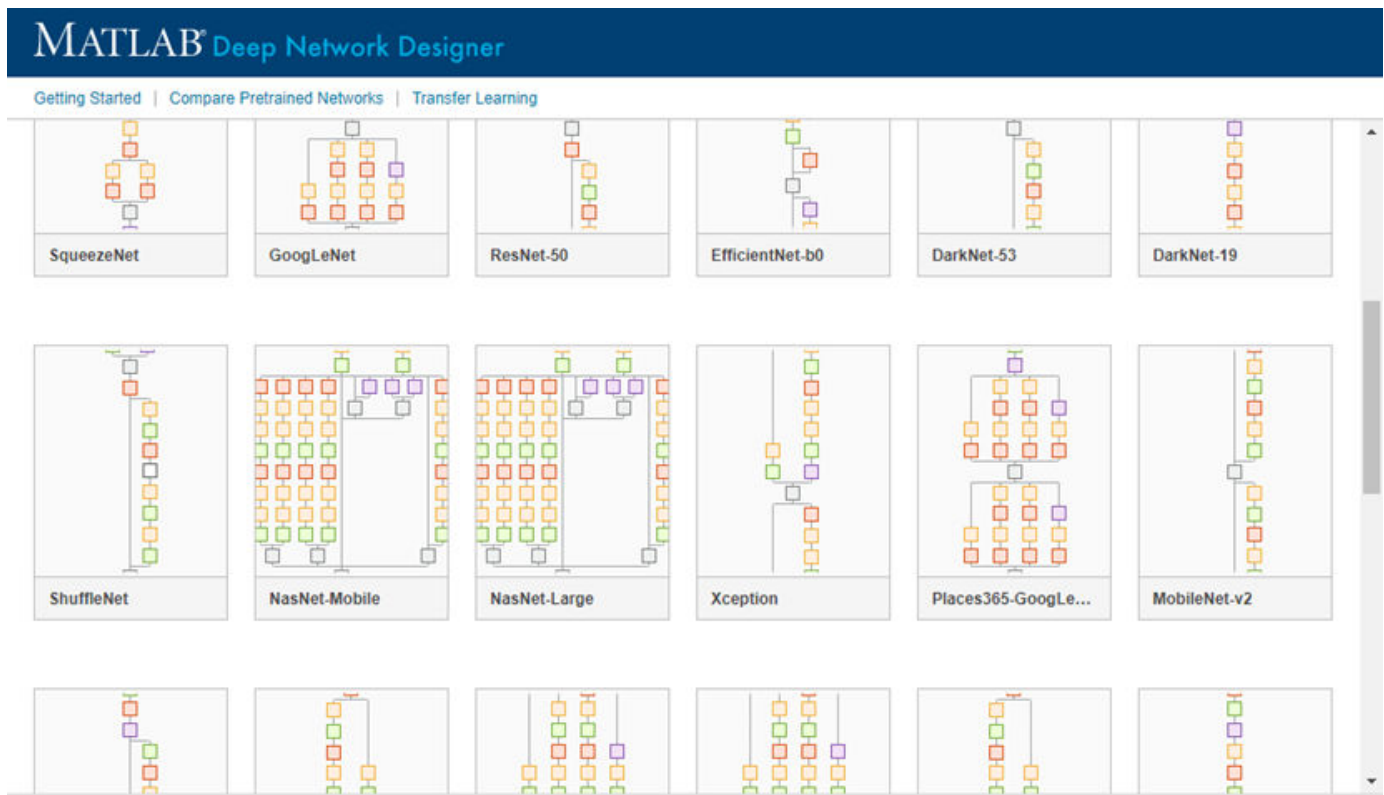
```
DAGNetwork with properties:
```

```
Layers: [177x1 nnet.cnn.layer.Layer]
Connections: [192x2 table]
```

Visualize the neural network using Deep Network Designer.

```
deepNetworkDesigner(resnet50)
```

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



If you need to download a neural network, pause on the desired neural 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.

## **Version History**

**Introduced in R2017b**

## **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 neural 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 neural 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”

# 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 =

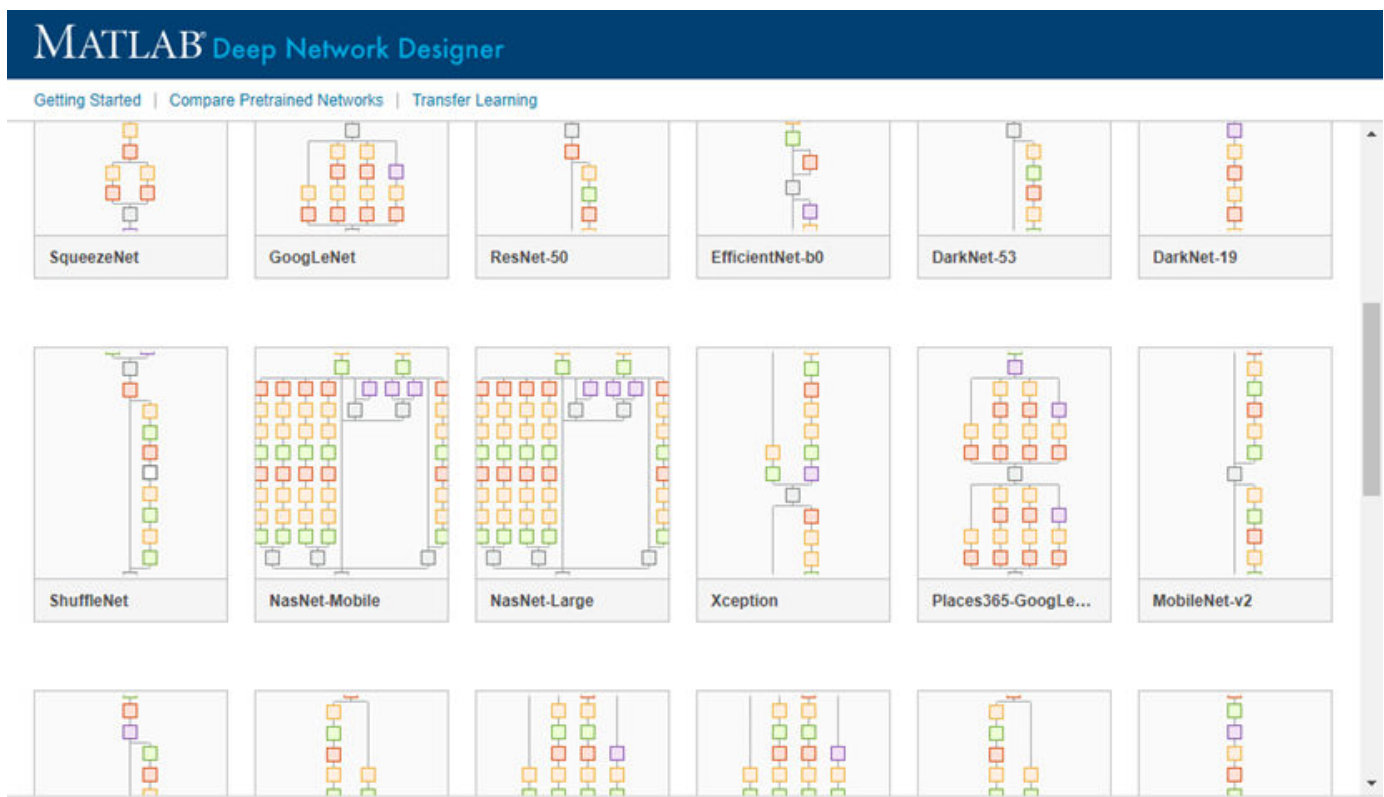
    DAGNetwork with properties:

        Layers: [347x1 nnet.cnn.layer.Layer]
    Connections: [379x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(resnet101)
```

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



If you need to download a neural network, pause on the desired neural 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.

## **Version History**

**Introduced in R2017b**

## **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”

# 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:
    InputNames: {'input'}
    OutputNames: {'output'}
    Layers: [177x1 nnet.cnn.layer.Layer]
    Connections: [192x2 table]
```

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:

        InputNames: {'input'}
        OutputNames: {'output'}
        Layers: [347x1 nnet.cnn.layer.Layer]
        Connections: [379x2 table]
```

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]);
```

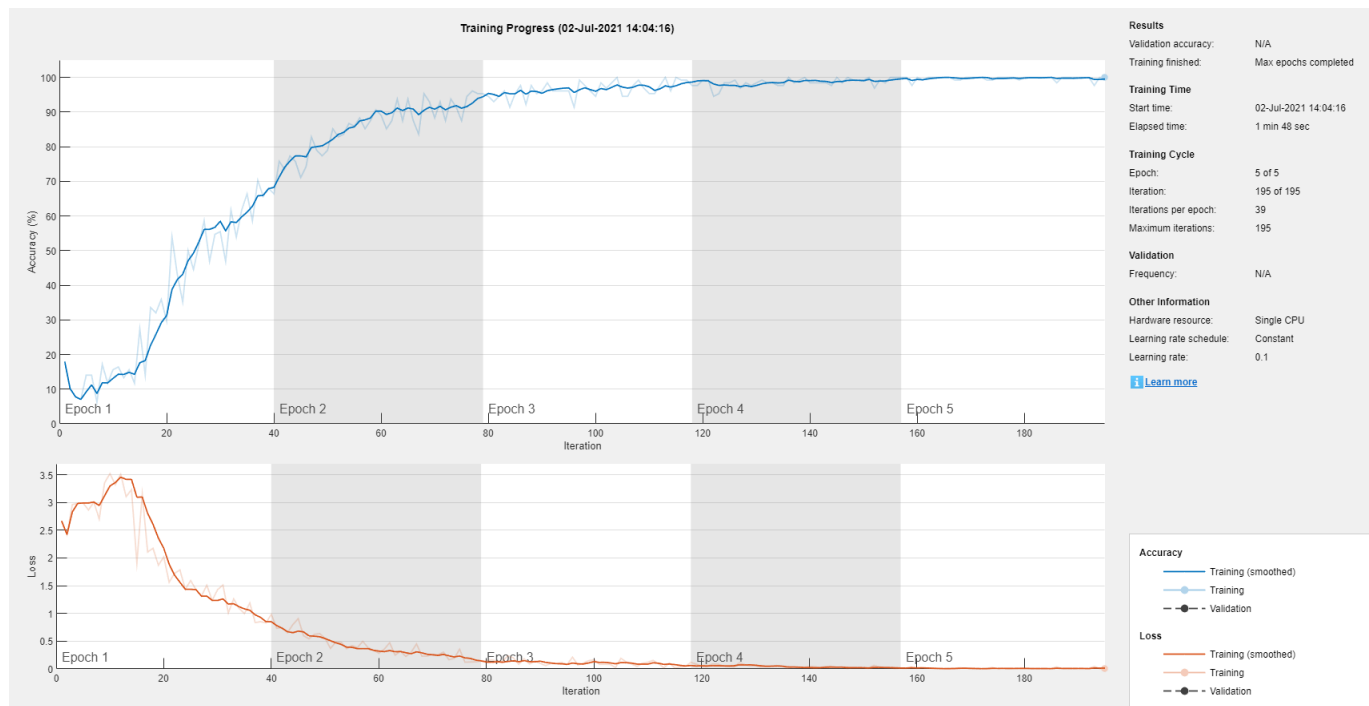
Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 5, and start the training with an initial learning rate of 0.1.

```
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
```

```
View summary with summary.
```

## 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 1-1576.

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 1-1576.

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 1-1576.

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

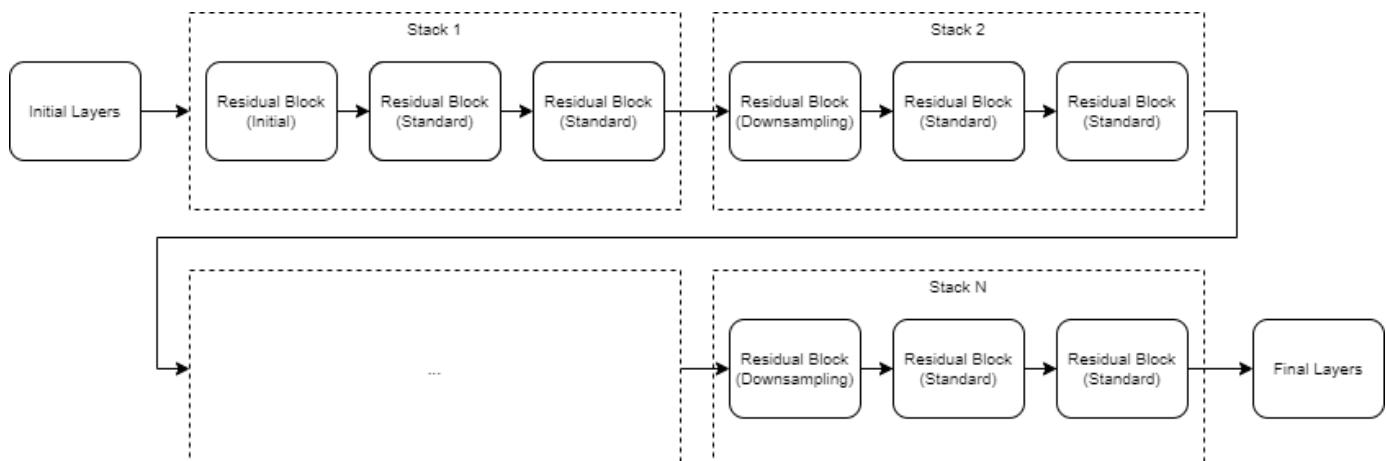
### Residual Network

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.

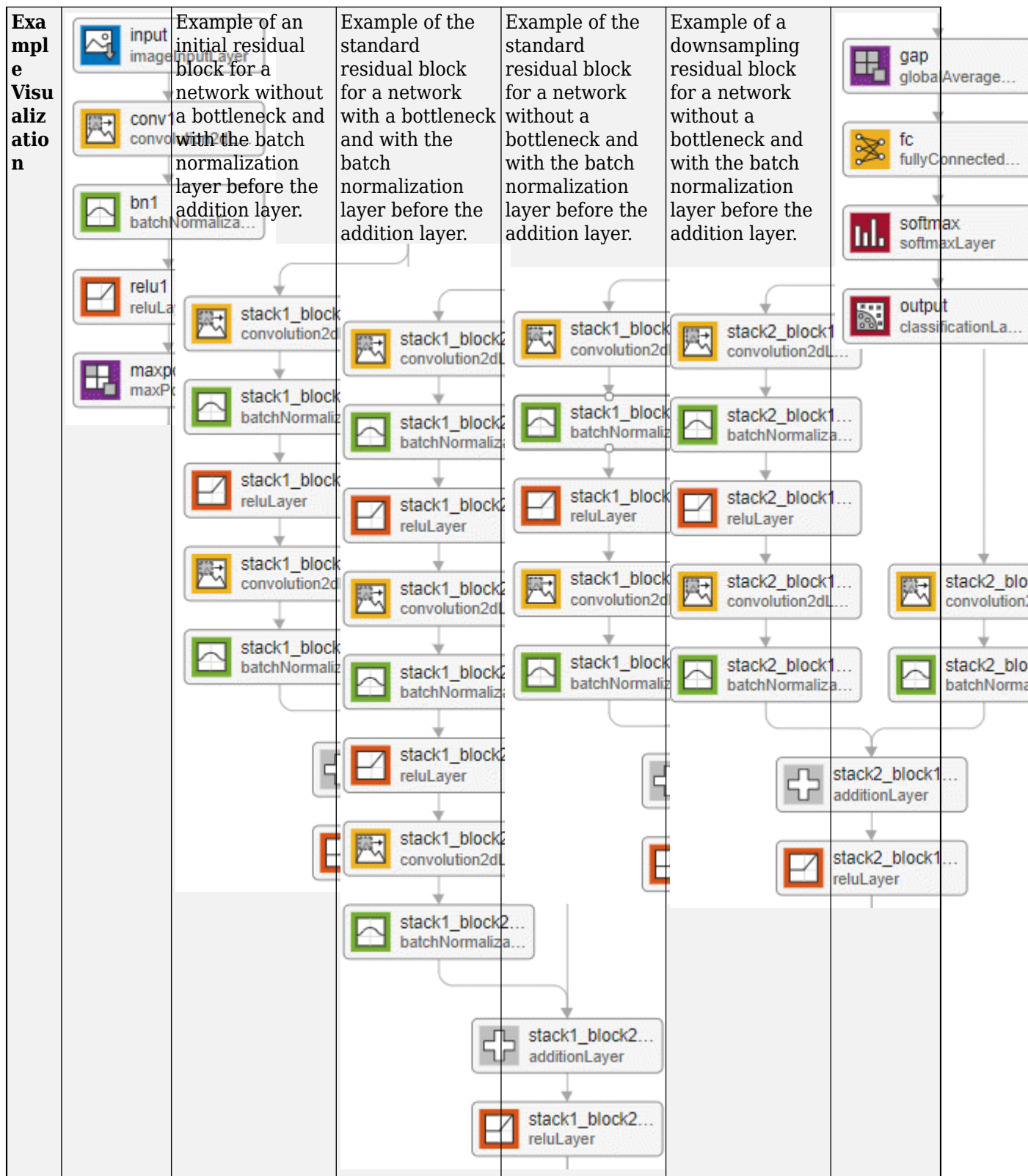
The layers inside each block are determined by the type of block and the options you set.

**Block Layers**

<b>Name</b>	<b>Initial Layers</b>	<b>Initial Residual Block</b>	<b>Standard Residual Block (BottleneckType="downsample-first-conv")</b>	<b>Standard Residual Block (BottleneckType="none")</b>	<b>Downsampling Residual Block</b>	<b>Final Layers</b>
-------------	-----------------------	-------------------------------	---	--	------------------------------------	---------------------

<b>Description</b>	<p>A residual network starts with the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>imageInputLayer</code></li> <li>• <code>convolution2dLayer</code></li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• (Optional) Pooling layer (either max, average, or none)</li> </ul> <p>Set the optional pooling layer using the <code>InitialPoolingLayer</code> argument.</p>	<p>The main branch of the initial residual block has the same layers as a standard residual block.</p> <p>The <code>InitialNumFilters</code> and <code>NumFilters</code> 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:</p> <ul style="list-style-type: none"> <li>• <code>BottleneckType="downsample-first-conv"</code> (default) and <code>InitialNumFilters</code> is not equal to four times the first element of <code>NumFilters</code>.</li> <li>• <code>BottleneckType="none"</code> and <code>InitialNumFilters</code> is not equal to the first element of <code>NumFilters</code>.</li> </ul> <p>If <code>ResidualBlockType</code> is set</p>	<p>The standard residual block with bottleneck units has the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>convolution2dLayer</code> with [1, 1] filter and [1, 1] stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution2dLayer</code> with [3, 3] filter and [1, 1] stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution2dLayer</code> with [1, 1] filter and [1, 1] stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>additionLayer</code></li> <li>• <code>reluLayer</code></li> </ul> <p>The standard block has a residual connection from the output of the previous block to the addition layer.</p> <p>Set the position of the addition layer using the</p>	<p>The standard residual block without bottleneck units has the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>convolution2dLayer</code> with [3, 3] filter and [1, 1] stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution2dLayer</code> with [3, 3] filter and [1, 1] stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>additionLayer</code></li> <li>• <code>reluLayer</code></li> </ul> <p>The standard block has a residual connection from the output of the previous block to the addition layer.</p> <p>Set the position of the addition layer using the <code>ResidualBlockType</code> argument.</p>	<p>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.</p> <p>The layers on the residual connection depend on the <code>ResidualBlockType</code> value.</p> <ul style="list-style-type: none"> <li>• When <code>ResidualBlockType</code> is set to <code>"batchnorm-before-add"</code>, the second branch contains a <code>convolution2dLayer</code> with [1, 1] filter and [2, 2] stride, and a <code>batchNormalizationLayer</code>.</li> <li>• When <code>ResidualBlockType</code> is set to <code>"batchnorm-after-add"</code>, the second branch</li> </ul>	<p>A residual network ends with the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>globalAveragePooling2dLayer</code></li> <li>• <code>fullyConnectedLayer</code></li> <li>• <code>softmaxLayer</code></li> <li>• <code>classificationLayer</code></li> </ul>
--------------------	---	---	--	--	---	---

		to "batchnorm-before-add", the residual connection will also have a batch normalization layer.	ResidualBlock Type 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.	
--	--	--	---------------------------------	--	---	--



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");
```

## Version History

Introduced in R2021b

## 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"

## 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:
        InputNames: {'input'}
        OutputNames: {'output'}
        Layers: [177x1 nnet.cnn.layer.Layer]
        Connections: [192x2 table]
```

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:

        InputNames: {'input'}
        OutputNames: {'output'}
        Layers: [347x1 nnet.cnn.layer.Layer]
        Connections: [379x2 table]
```

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 1-1589.

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 1-1589.

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 1-1589.

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

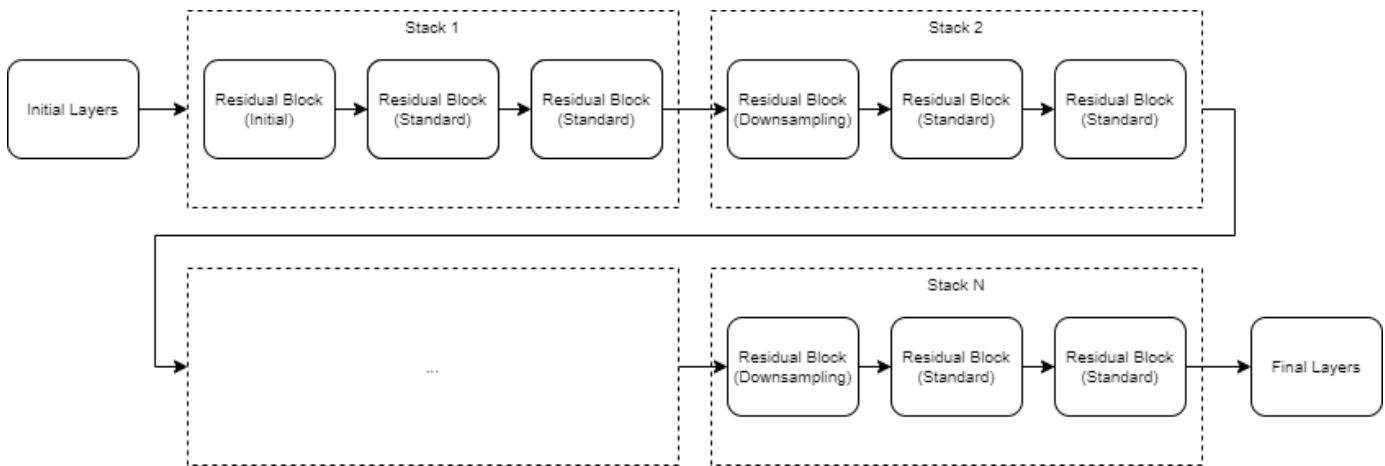
### Residual Network

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.

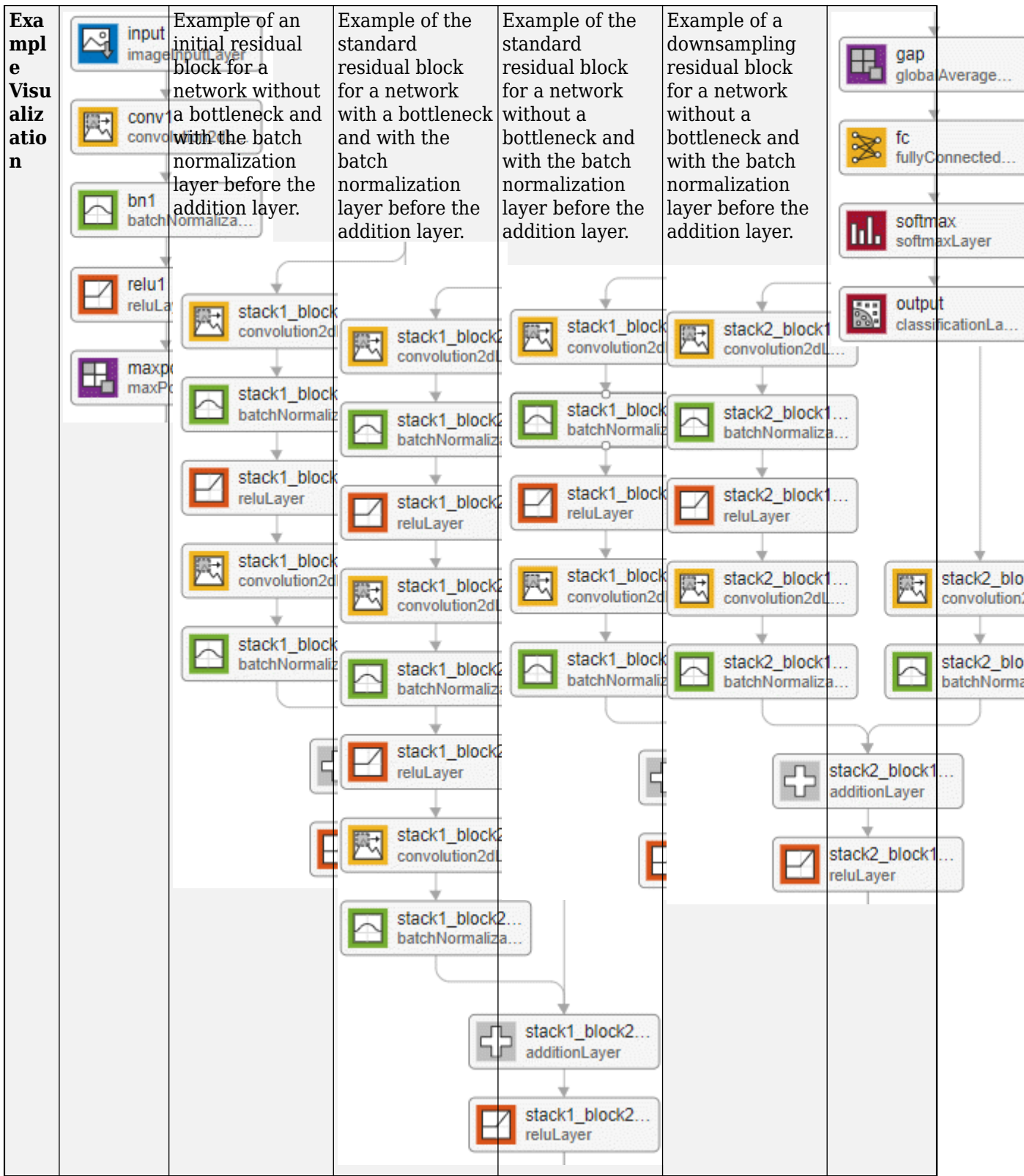


**Block Layers**

<b>Name</b>	<b>Initial Layers</b>	<b>Initial Residual Block</b>	<b>Standard Residual Block (BottleneckType="downsample-first-conv")</b>	<b>Standard Residual Block (BottleneckType="none")</b>	<b>Downsampling Residual Block</b>	<b>Final Layers</b>
-------------	-----------------------	-------------------------------	---	--	------------------------------------	---------------------

<p><b>Des crip tion</b></p>	<p>A residual network starts with the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>image3dInputLayer</code></li> <li>• <code>convolution3dLayer</code></li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• (Optional) Pooling layer (either max, average, or none)</li> </ul> <p>Set the optional pooling layer using the <code>InitialPoolingLayer</code> argument.</p>	<p>The main branch of the initial residual block has the same layers as a standard residual block.</p> <p>The <code>InitialNumFilters</code> and <code>NumFilters</code> values determine the layers on the residual connection. The residual connection has a convolutional layer with <code>[1,1,1]</code> filter and <code>[1,1,1]</code> stride if one of the following conditions is met:</p> <ul style="list-style-type: none"> <li>• <code>BottleneckType="downsample-first-conv"</code> (default) and <code>InitialNumFilters</code> is not equal to four times the first element of <code>NumFilters</code>.</li> <li>• <code>BottleneckType="none"</code> and <code>InitialNumFilters</code> is not equal to the first element of <code>NumFilters</code>.</li> </ul> <p>If <code>ResidualBlock</code></p>	<p>The standard residual block with bottleneck units has the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>convolution3dLayer</code> with <code>[1,1,1]</code> filter and <code>[1,1,1]</code> stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution3dLayer</code> with <code>[3,3,3]</code> filter and <code>[1,1,1]</code> stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution3dLayer</code> with <code>[1,1,1]</code> filter and <code>[1,1,1]</code> stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> </ul> <p>The standard block has a residual connection from the output of the previous block to</p>	<p>The standard residual block without bottleneck units has the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>convolution3dLayer</code> with <code>[3,3,3]</code> filter and <code>[1,1,1]</code> stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>reluLayer</code></li> <li>• <code>convolution3dLayer</code> with <code>[3,3,3]</code> filter and <code>[1,1,1]</code> stride</li> <li>• <code>batchNormalizationLayer</code></li> <li>• <code>additionLayer</code></li> <li>• <code>reluLayer</code></li> </ul> <p>The standard block has a residual connection from the output of the previous block to the addition layer.</p> <p>Set the position of the addition layer using the <code>ResidualBlockType</code> argument.</p>	<p>The downsampling residual block is the same as the standard block (either with or without the bottleneck) but with a stride of <code>[2,2,2]</code> in the first convolutional layer and additional layers on the residual connection.</p> <p>The layers on the residual connection depend on the <code>ResidualBlockType</code> value.</p> <ul style="list-style-type: none"> <li>• When <code>ResidualBlockType</code> is set to <code>"batchnorm-before-add"</code>, the second branch contains a <code>convolution3dLayer</code> with <code>[1,1,1]</code> filter and <code>[2,2,2]</code> stride, and a <code>batchNormalizationLayer</code>.</li> <li>• When <code>ResidualBlockType</code> is set to <code>"batchnorm-after-add"</code>, the second</li> </ul>	<p>A residual network ends with the following layers, in order:</p> <ul style="list-style-type: none"> <li>• <code>globalAveragePooling3dLayer</code></li> <li>• <code>fullyConnectedLayer</code></li> <li>• <code>softmaxLayer</code></li> <li>• <code>classificationLayer</code></li> </ul>
-------------------------------------	---	--	--	--	--	---

		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.	
--	--	--	--	--	--	--



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");
```

## Version History

Introduced in R2021b

## 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"

# ROCCurve Properties

Receiver operating characteristic (ROC) curve appearance and behavior

## Description

ROCCurve properties control the appearance and behavior of a ROCCurve object, which the `plot` function of a `rocmetrics` object returns for each receiver operating characteristic (ROC) curve or other performance curve.

By changing property values, you can modify certain aspects of the ROC curve. Use dot notation to query and set properties; for example:

```
rocObj = rocmetrics(Lables, Scores, ClassNames);
curveObj = plot(rocObj);
c = curveObj(1).Color;
curveObj(1).Color = "red";
```

## Properties

### ROC Curve Data

#### XAxisMetric — Performance metric for x-axis

name of performance metric

This property is read-only.

Performance metric for the x-axis, stored in `XData`, specified as a string scalar of the performance metric name.

The `XAxisMetric` name-value argument of the `plot` function sets this property. The default value of this argument is `"FalsePositiveRate"`.

For details about built-in and custom metric names, see the `XAxisMetric` name-value argument of the `plot` function.

#### XData — x-axis values

numeric vector

This property is read-only.

x-axis values for `XAxisMetric`, specified as a numeric vector.

Data Types: `double`

#### YAxisMetric — Performance metric for y-axis

name of performance metric

This property is read-only.

Performance metric for the y-axis, stored in `YData`, specified as a string scalar of the performance metric name.

The `YAxisMetric` name-value argument of the `plot` function sets this property. The default value of this argument is `"TruePositiveRate"`.

For details about built-in and custom metric names, see the `XAxisMetric` name-value argument of the `plot` function.

### **YData — y-axis values**

numeric vector

This property is read-only.

y-axis values for `YAxisMetric`, specified as a numeric vector.

Data Types: `double`

### **Thresholds — Thresholds on classification scores**

numeric vector

This property is read-only.

Thresholds on classification scores at which the software finds each of the performance metric values (`XData` and `YData`), specified as a numeric vector.

Data Types: `double`

### **ROC Curve Options**

#### **ShowConfidenceIntervals — Flag to show confidence intervals**

`true` or `1` | `false` or `0`

Flag to show the confidence intervals for the y-axis metric (`YAxisMetric`), specified as logical `1` (`true`) or `0` (`false`).

The `ShowConfidenceIntervals` name-value argument of the `plot` function sets this property. The default value of the argument is `false`.

The `ShowConfidenceIntervals` value can be `true` only if the `Metrics` property of the `rocmetrics` object contains the confidence intervals for the y-axis metric.

Using confidence intervals requires Statistics and Machine Learning Toolbox.

### **Color**

#### **Color — Color of line and confidence interval shading**

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

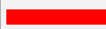







Color of the line and confidence interval shading, specified as an RGB triplet, hexadecimal color code, color name, or short name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

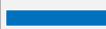






- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0, 1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with 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"	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

### ColorMode — Mode for setting Color

'auto' (default) | 'manual'

Mode for setting the Color property, specified as one of these values:

- 'auto' — The software controls the value of the Color property by selecting a color from the ColorOrder property of the axes.
- 'manual' — You control the value of the Color property manually by setting the value of the Color property directly on the object.


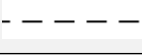
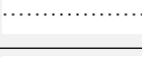
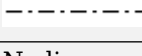
If you change the value of the Color property manually, the software changes the value of the ColorMode property to 'manual'.

### Line

#### LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the options in this table.

Line Style	Description	Resulting Line
"_"	Solid line	
"_ _"	Dashed line	
":"	Dotted line	
"- ."	Dash-dotted line	
"none"	No line	No line

### LineStyleMode — Mode for setting LineStyle

'auto' (default) | 'manual'

Mode for setting the LineStyle property, specified as one of these values:

- 'auto' — The software controls the value of the LineStyle property by selecting a line style from the LineStyleOrder property of the axes.
- 'manual' — You control the value of the LineStyle property manually by setting the value of the LineStyle property directly on the object.

If you change the value of the LineStyle property manually, the software changes the value of the LineStyleMode property to 'manual'.

### LineWidth — Line width

0.5 (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

The line width cannot be thinner than the width of a pixel. If you set the line width to a value that is less than the width of a pixel on your system, the line is displayed one pixel wide.

### SeriesIndex — Series index

whole number

Series index, specified as a whole number greater than or equal to 0. This property is useful for reassigning the colors, line styles, or markers of several ROCCurve objects so that they match each other. By default, the SeriesIndex property of a ROCCurve object is a number that corresponds to its order of creation, starting at 1.

The software uses the number to calculate indices for assigning color, line style, or markers when you call plotting functions. The indices refer to the rows of the arrays stored in the ColorOrder and LineStyleOrder properties of the axes.

The software automatically updates the color, line style, or markers of the ROCCurve object when you change its SeriesIndex, or when you change the ColorOrder or LineStyleOrder properties of the axes. However, the following conditions must be true for the changes to have any effect:

- At least one of these properties of the ROCCurve object is set to 'auto': ColorMode, LineStyleMode, or MarkerMode.

- The SeriesIndex property of the ROCCurve object is greater than 0.
- The NextSeriesIndex property of the axes object is greater than 0.

### Markers

#### Marker — Marker symbol

'none' | 'o' | '+' | '\*' | '.' | ...

Marker symbol, specified as one of the values in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Marker	Description	Resulting Marker
"o"	Circle	○
"+"	Plus sign	+
"*"	Asterisk	*
"."	Point	•
"x"	Cross	×
"_"	Horizontal line	—
" "	Vertical line	
"square"	Square	□
"diamond"	Diamond	◇
"^"	Upward-pointing triangle	△
"v"	Downward-pointing triangle	▽
">"	Right-pointing triangle	▷
"<"	Left-pointing triangle	◁
"pentagram"	Pentagram	☆
"hexagram"	Hexagram	☆
"none"	No markers	Not applicable

#### MarkerMode — Mode for setting Marker

'auto' (default) | 'manual'

Mode for setting the Marker property, specified as one of these values:

- 'auto' — The software controls the value of the object's Marker property.
- 'manual' — You control the value of the Marker property manually by setting the value of the Marker property directly on the object.

If you change the value of the `Marker` property manually, the software changes the value of the `MarkerMode` property to `'manual'`.

### Legend

#### **DisplayName** — Text used in legend

character vector

Text used in the legend, specified as a character vector.

The default value of `DisplayName` is the name of the class for which the curve describes the performance. If the curve is a ROC curve, then `DisplayName` also contains the area under the ROC curve (AUC) value.

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property.

If you interactively edit the character vector in an existing legend, then the software updates the `DisplayName` property to the edited character vector.

#### **Annotation** — Control for legend

Annotation object

Control for including the `ROCCurve` object in the legend or excluding the object from the legend, specified as an Annotation object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the `ROCCurve` object in the legend (default).
- `'off'` — Do not include the `ROCCurve` object in the legend.

For example, to exclude the `ROCCurve` object `curveObj` from the legend, set the `IconDisplayStyle` property to `'off'`.

```
curveObj.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

### Interactivity

#### **Visible** — State of visibility for object

`'on'` (default) | on/off logical value

State of visibility for an object, 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 object.
- 'off' — Hide the object without deleting it. You still can access the properties of an invisible object.

### DataTipTemplate — Data tip content

DataTipTemplate object

Data tip content, specified as a DataTipTemplate object. You can control the content that appears in a data tip by modifying the properties of the underlying DataTipTemplate object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

---

**Note** The DataTipTemplate object is not returned by findobj or findall, and it is not copied by copyobj.

---

### Selected — Selection state of object

'off' (default) | on/off logical value

Selection state of an object, 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' — The object is selected. If you click the object when editing the plot, the software sets the object's Selected property to 'on'. If the SelectionHighlight property is also set to 'on', the software displays selection handles around the object.
- 'off' — The object is not selected.

### SelectionHighlight — Display of selection handles

'on' (default) | on/off logical value

Display of selection handles when an object is selected, 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 selection handles around the object when the Selected property is set to 'on'.
- 'off' — Do not display selection handles around the object, even when the Selected property is set to 'on'.

### Callback Execution Control

#### PickableParts — Ability to capture mouse clicks

'visible' (default) | 'all' | 'none'

Ability to capture mouse clicks, specified as one of these values:

- 'visible' — Capture mouse clicks when they are visible. The Visible property must be set to 'on' and you must click a part of the ROCCurve object that has a defined color. You cannot click a part that has an associated color property set to 'none'. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the ROCCurve object responds to the click or if an ancestor does.

- 'all' — Capture mouse clicks regardless of visibility. The `Visible` property can be set to 'on' or 'off' and you can click a part of the `ROCCurve` object that has no color. The `HitTest` property determines if the `ROCCurve` object responds to the click or if an ancestor does.
- 'none' — Mouse clicks cannot be captured. Clicking the `ROCCurve` object passes the click to the object below it in the current view of the figure window. The `HitTest` property has no effect.

**HitTest — Response to captured mouse clicks**

'on' (default) | on/off logical value

Response to captured mouse clicks, 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' — Create a data tip at a data point of the `ROCCurve` object.
- 'off' — Trigger the callbacks for the nearest ancestor of the `ROCCurve` object that has one of these:
  - `HitTest` property set to 'on'
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

**Note** The `PickableParts` property determines if the `ROCCurve` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

---

**Parent/Child****Parent — Parent**

Axes object

Parent, specified as an Axes object.

**Children — Children**

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, specified as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips plotted on the performance curve.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the `ROCCurve` object.

**HandleVisibility — Visibility of object handle**

'on' (default) | 'off' | 'callback'

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- 'on' — The object handle is always visible.
- 'off' — The object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set `HandleVisibility` to 'off' to temporarily hide the handle during the execution of that function.

- `'callback'` — The object handle is visible from callbacks or functions invoked by callbacks, but not from functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access the object.

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 the object. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

### Identifier

#### Type — Type of graphics object

`'roccurve'`

This property is read-only.

Type of graphics object, specified as `'roccurve'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

#### Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need to access the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

#### UserData — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

## Version History

Introduced in R2022b

### See Also

`rocmetrics` | `addMetrics` | `average` | `plot`

### Topics

“ROC Curve and Performance Metrics”

“Compare Deep Learning Models Using ROC Curves”

## rocmetrics

Receiver operating characteristic (ROC) curve and performance metrics for binary and multiclass classifiers

### Description

Create a `rocmetrics` object to evaluate the performance of a classification model using receiver operating characteristic (ROC) curves on page 1-1623 or other performance metrics. `rocmetrics` supports both binary and multiclass problems.

For each class, `rocmetrics` computes performance metrics for a one-versus-all on page 1-1624 ROC curve. You can compute metrics for an average ROC curve by using the `average` function. After computing metrics for ROC curves, you can plot them by using the `plot` function.

By default, `rocmetrics` computes the false positive rates (FPR) and the true positive rates (TPR) to obtain a ROC curve and the area under the ROC curve (AUC) on page 1-1624. You can compute additional metrics by specifying the `AdditionalMetrics` name-value argument when you create an object or by calling the `addMetrics` function after you create an object. A `rocmetrics` object stores the computed metrics and AUC values in the `Metrics` and `AUC` properties, respectively.

`rocmetrics` computes pointwise confidence intervals for the performance metrics when you set the `NumBootstraps` value to a positive integer or when you specify cross-validated data for the true class labels (`Labels`), classification scores (`Scores`), and observation weights (`Weights`). For details, see “Pointwise Confidence Intervals”. Using confidence intervals requires Statistics and Machine Learning Toolbox.

### Creation

#### Syntax

```
rocObj = rocmetrics(Labels,Scores,ClassNames)
```

```
rocObj = rocmetrics(Labels,Scores,ClassNames,Name=Value)
```

#### Description

`rocObj = rocmetrics(Labels,Scores,ClassNames)` creates a `rocmetrics` object using the true class labels in `Labels` and the classification scores in `Scores`. Specify `Labels` as a vector of length `n`, and specify `Scores` as a matrix of size `n`-by-`K`, where `n` is the number of observations, and `K` is the number of classes. `ClassNames` specifies the column order in `Scores`.

The `Metrics` and `AUC` properties contain the performance metrics and AUC value for each class for which you specify `Scores` and `ClassNames`.

If you specify cross-validated data in `Labels` and `Scores` as cell arrays, then `rocmetrics` computes confidence intervals for the performance metrics. Using cross-validated data requires Statistics and Machine Learning Toolbox.



`rocObj = rocmetrics(Labels, Scores, ClassNames, Name=Value)` specifies additional options using one or more name-value arguments. For example, `Prior="uniform"` sets all class probabilities to be equal.

## Input Arguments

### Labels — True class labels

numeric vector | logical vector | categorical vector | character array | string array | cell array

True class labels, specified as a numeric vector, logical vector, categorical vector, character array, string array, or cell array of character vectors. You can also specify `Labels` as a cell array of one of these types for cross-validated data.

- For data that is not cross-validated, the length of `Labels` and the number of rows in `Scores` must be equal.
- For cross-validated data, you must specify `Labels`, `Scores`, and `Weights` as cell arrays with the same number of elements. `rocmetrics` treats an element in the cell arrays as data from one cross-validation fold and computes pointwise confidence intervals for the performance metrics. The length of `Labels{i}` and the number of rows in `Scores{i}` must be equal. Using cross-validated data requires Statistics and Machine Learning Toolbox.

Each row of `Labels` or `Labels{i}` represents the true label of one observation.

This argument sets the `Labels` property.

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

### Scores — Classification scores

numeric matrix | cell array of numeric matrices

Classification scores, specified as a numeric matrix or a cell array of numeric matrices.

Each row of the matrix in `Scores` contains the classification scores of one observation for all classes specified in `ClassNames`. The column order of `Scores` must match the class order in `ClassNames`.

- For a matrix input, `Score(j,k)` is the classification score of observation `j` for class `ClassNames(k)`. For example, `predict` returns predicted class scores as an  $n$ -by- $K$  matrix, where  $n$  is the number of observations and  $K$  is the number classes. Pass the output to `rocmetrics`.

The number of rows in `Scores` and the length of `Labels` must be equal. `rocmetrics` adjusts scores for each class relative to the scores for the rest of the classes. For details, see “Adjusted Scores for Multiclass Classification Problem” on page 1-1625.

- For a vector input, `Score(j)` is the classification score of observation `j` for the class specified in `ClassNames`.
  - `ClassNames` must contain only one class.
  - `Prior` must be a two-element vector with `Prior(1)` representing the prior probability for the specified class.
  - `Cost` must be a 2-by-2 matrix containing  $[\text{Cost}(P|P), \text{Cost}(N|P); \text{Cost}(P|N), \text{Cost}(N|N)]$ , where `P` is a positive class (the class for which you specify classification scores), and `N` is a negative class.
  - The length of `Scores` and the length of `Labels` must be equal.

If you want to display the model operating point when you plot the ROC curve using the `plot` function, the values in `Score(j)` must be the posterior probability. This restriction applies only to a vector input.

- For cross-validated data, you must specify `Labels`, `Scores`, and `Weights` as cell arrays with the same number of elements. `rocmetrics` treats an element in the cell arrays as data from one cross-validation fold and computes pointwise confidence intervals for the performance metrics. `Score{i}(j,k)` is the classification score of observation `j` in element `i` for class `ClassNames(k)`. The number of rows in `Scores{i}` and the length of `Labels{i}` must be equal. Using cross-validated data requires Statistics and Machine Learning Toolbox.

For more information, see “Classification Score Input for `rocmetrics`”.

This argument sets the `Scores` property.

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

### **ClassNames — Class names**

numeric vector | logical vector | categorical vector | character array | string array | cell array of character vectors

Class names, specified as a numeric vector, logical vector, categorical vector, character array, string array, or cell array of character vectors. `ClassNames` must have the same data type as the true labels in `Labels`. The values in `ClassNames` must appear in `Labels`.

- If you specify classification scores for only one class in `Scores`, `ClassNames` specifies only the name of this class.
- Otherwise, `ClassNames` specifies the order of the classes in `Scores`, `Cost`, and `Prior`.

This argument sets the `ClassNames` property.

Data Types: `single` | `double` | `logical` | `cell` | `categorical`

### **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: `FixedMetric="FalsePositiveRate", FixedMetricValues=0:0.01:1` holds the FPR values fixed at `0:0.01:1`.

### **Performance Metrics**

#### **AdditionalMetrics — Additional model performance metrics**

`[]` (default) | character vector | string array | function handle | cell array

Additional model performance metrics to compute, specified as a character vector or string scalar of the built-in metric name, string array of names, function handle (`@metricName`), or cell array of names or function handles. A `rocmetrics` object always computes the false positive rates (FPR) and the true positive rates (TPR) to obtain a ROC curve. Therefore, you do not have to specify to compute FPR and TPR.

- Built-in metrics — Specify one of the following built-in metric names by using a character vector or string scalar. You can specify more than one by using a string array.

Name	Description
"TruePositives" or "tp"	Number of true positives (TP)
"FalseNegatives" or "fn"	Number of false negatives (FN)
"FalsePositives" or "fp"	Number of false positives (FP)
"TrueNegatives" or "tn"	Number of true negatives (TN)
"SumOfTrueAndFalsePositives" or "tp+fp"	Sum of TP and FP
"RateOfPositivePredictions" or "rpp"	Rate of positive predictions (RPP), $(TP+FP)/(TP+FN+FP+TN)$
"RateOfNegativePredictions" or "rnp"	Rate of negative predictions (RNP), $(TN+FN)/(TP+FN+FP+TN)$
"Accuracy" or "accu"	Accuracy, $(TP+TN)/(TP+FN+FP+TN)$
"FalseNegativeRate", "fnr", or "miss"	False negative rate (FNR), or miss rate, $FN/(TP+FN)$
"TrueNegativeRate", "tnr", or "spec"	True negative rate (TNR), or specificity, $TN/(TN+FP)$
"PositivePredictiveValue", "ppv", or "prec"	Positive predictive value (PPV), or precision, $TP/(TP+FP)$
"NegativePredictiveValue" or "npv"	Negative predictive value (NPV), $TN/(TN+FN)$
"ExpectedCost" or "ecost"	<p>Expected cost, <math>(TP*cost(P P)+FN*cost(N P)+FP*cost(P N)+TN*cost(N N))/(TP+FN+FP+TN)</math>, where <i>cost</i> is a 2-by-2 misclassification cost matrix containing <math>[0, cost(N P); cost(P N), 0]</math>. <i>cost(N P)</i> is the cost of misclassifying a positive class (P) as a negative class (N), and <i>cost(P N)</i> is the cost of misclassifying a negative class as a positive class.</p> <p>The software converts the K-by-K matrix specified by the <i>Cost</i> name-value argument of <i>rocmetrics</i> to a 2-by-2 matrix for each one-versus-all binary problem. For details, see "Misclassification Cost Matrix".</p>

The software computes the scale vector using the prior class probabilities (*Prior*) and the number of classes in *Labels*, and then scales the performance metrics according to this scale vector. For details, see "Performance Metrics".

- Custom metric — Specify a custom metric by using a function handle. A custom function that returns a performance metric must have this form:

```
metric = customMetric(C, scale, cost)
```

- The output argument *metric* is a scalar value.
- A custom metric is a function of the confusion matrix (*C*), scale vector (*scale*), and cost matrix (*cost*). The software finds these input values for each one-versus-all binary problem. For details, see "Performance Metrics".
- *C* is a 2-by-2 confusion matrix consisting of  $[TP, FN; FP, TN]$ .

- `scale` is a 2-by-1 scale vector.
- `cost` is a 2-by-2 misclassification cost matrix.

The software does not support cross-validation for a custom metric. Instead, you can specify to use bootstrap when you create a `rocmetrics` object.

Note that the positive predictive value (PPV) is NaN for the reject-all threshold for which  $TP = FP = 0$ , and the negative predictive value (NPV) is NaN for the accept-all threshold for which  $TN = FN = 0$ . For more details, see “Thresholds, Fixed Metric, and Fixed Metric Values”.

Example: `AdditionalMetrics=["Accuracy","PositivePredictiveValue"]`

Example: `AdditionalMetrics={"Accuracy",@m1,@m2}` specifies the accuracy metric and the custom metrics `m1` and `m2` as additional metrics. `rocmetrics` stores the custom metric values as variables named `CustomMetric1` and `CustomMetric2` in the `Metrics` property.

Data Types: `char` | `string` | `cell` | `function_handle`

### **FixedMetric — Fixed metric**

"Thresholds" (default) | "FalsePositiveRate" | "TruePositiveRate" | metric specified by `AdditionalMetrics`

Fixed metric, specified as "Thresholds", "FalsePositiveRate" (or "fpr"), "TruePositiveRate" (or "tpr"), or a metric specified by the `AdditionalMetrics` name-value argument. To hold a custom metric fixed, specify `FixedMetric` as "CustomMetricN", where N is the number that refers to the custom metric. For example, specify "CustomMetric1" to use the first custom metric specified by `AdditionalMetrics` as the fixed metric.

`rocmetrics` finds the ROC curves and other metric values that correspond to the fixed values (`FixedMetricValues`) of the fixed metric (`FixedMetric`), and stores the values in the `Metrics` property as a table. For more details, see “Thresholds, Fixed Metric, and Fixed Metric Values”.

If `rocmetrics` computes confidence intervals, it uses one of two methods for the computation, depending on the `FixedMetric` value:

- If `FixedMetric` is "Thresholds" (default), `rocmetrics` uses threshold averaging.
- If `FixedMetric` is a nondefault value, `rocmetrics` uses vertical averaging.

For details, see “Pointwise Confidence Intervals”.

Using confidence intervals requires Statistics and Machine Learning Toolbox.

Example: `FixedMetric="TruePositiveRate"`

Data Types: `char` | `string`

### **FixedMetricValues — Values for fixed metric**

"all" (default) | numeric vector

Values for the fixed metric (`FixedMetric`), specified as "all" or a numeric vector.

`rocmetrics` finds the ROC curves and other metric values that correspond to the fixed values (`FixedMetricValues`) of the fixed metric (`FixedMetric`), and stores the values in the `Metrics` property as a table.

The default `FixedMetric` value is "Thresholds", and the default `FixedMetricValues` value is "all". For each class, `rocmetrics` uses all distinct adjusted score values on page 1-1625 as

threshold values and computes the performance metrics using the threshold values. Depending on the `UseNearestNeighbor` setting, `rocmetrics` uses the exact threshold values corresponding to the fixed values or the nearest threshold values. For more details, see “Thresholds, Fixed Metric, and Fixed Metric Values”.

If `rocmetrics` computes confidence intervals, it holds `FixedMetric` fixed at `FixedMetricValues`.

- `FixedMetric` value is "Thresholds", and `FixedMetricValues` is "all" — `rocmetrics` computes confidence intervals at the values corresponding to all distinct threshold values.
- `FixedMetric` value is a performance metric, and `FixedMetricValues` is "all" — `rocmetrics` finds the metric values corresponding to all distinct threshold values, and computes confidence intervals at the values corresponding to the metric values.

For details, see “Pointwise Confidence Intervals”. Using confidence intervals requires Statistics and Machine Learning Toolbox.

Example: `FixedMetricValues=0:0.01:1`

Data Types: `single` | `double`

### **NaNFlag — NaN condition**

"omitnan" (default) | "includenan"

NaN condition, specified as "omitnan" or "includenan".

- "omitnan" — `rocmetrics` ignores all NaN score values in the input `Scores` and the corresponding values in `Labels` and `Weights`.
- "includenan" — `rocmetrics` uses the NaN score values in the input `Scores` for the calculation. The function adds the observations with NaN scores to false classification counts in the respective class. That is, the function counts observations with NaN scores from the positive class as false negative (FN), and counts observations with NaN scores from the negative class as false positive (FP).

For more details, see “NaN Score Values”.

Example: `NaNFlag="includenan"`

Data Types: `char` | `string`

### **UseNearestNeighbor — Indicator to use nearest metric values**

false or 0 | true or 1

Indicator to use the nearest metric values, specified as logical 0 (false) or 1 (true).

- logical 0 (false) — `rocmetrics` uses the exact threshold values corresponding to the specified fixed metric values in `FixedMetricValues` for `FixedMetric`.
- logical 1 (true) — Among the adjusted input scores on page 1-1625, `rocmetrics` finds a value that is the nearest to the threshold value corresponding to each specified fixed metric value.

For more details, see “Thresholds, Fixed Metric, and Fixed Metric Values”.

The `UseNearestNeighbor` value must be false if `rocmetrics` computes confidence intervals. Otherwise, the default value is true.

Using confidence intervals requires Statistics and Machine Learning Toolbox.

Example: `UseNearestNeighbor=false`

Data Types: `logical`

### **Options for Classification Model**

#### **Cost — Misclassification cost**

square matrix

Misclassification cost, specified as a  $K$ -by- $K$  square matrix  $C$ , where  $K$  is the number of unique classes in `Labels`.  $C(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (that is, the rows correspond to the true class and the columns correspond to the predicted class). `ClassNames` specifies the order of the classes.

`rocmetrics` converts the  $K$ -by- $K$  matrix to a 2-by-2 matrix for each one-versus-all binary problem. For details, see “Misclassification Cost Matrix”.

If you specify classification scores for only one class in `Scores`, the `Cost` value must be a 2-by-2 matrix containing  $[0, \text{cost}(N|P); \text{cost}(P|N), 0]$ , where  $P$  is a positive class (the class for which you specify classification scores), and  $N$  is a negative class.  $\text{cost}(N|P)$  is the cost of misclassifying a positive class as a negative class, and  $\text{cost}(P|N)$  is the cost of misclassifying a negative class as a positive class.

The default value is  $C(i, j)=1$  if  $i \neq j$ , and  $C(i, j)=0$  if  $i=j$ . The diagonal entries of a cost matrix must be zero.

This argument sets the `Cost` property.

Example: `Cost=[0 2;1 0]`

Data Types: `single` | `double`

#### **Prior — Prior class probabilities**

"empirical" (default) | "uniform" | vector of scalar values

Prior class probabilities, specified as one of the following:

- "empirical" determines class probabilities from class frequencies in the true class labels `Labels`. If you pass observation weights (`Weights`), `rocmetrics` also uses the weights to compute the class probabilities.
- "uniform" sets all class probabilities to be equal.
- Vector of scalar values, with one scalar value for each class. `ClassNames` specifies the order of the classes.

If you specify classification scores for only one class in `Scores`, the `Prior` value must be a two-element vector with `Prior(1)` representing the prior probability for the specified class.

This argument sets the `Prior` property.

Example: `Prior="uniform"`

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

#### **Weights — Observation weights**

numeric vector of positive values | cell array containing numeric vectors of positive values

Observation weights, specified as a numeric vector of positive values or a cell array containing numeric vectors of positive values.

- For data that is not cross-validated, specify `Weights` as a numeric vector that has the same length as `Labels`.
- For cross-validated data, you must specify `Labels`, `Scores`, and `Weights` as cell arrays with the same number of elements. `rocmetrics` treats an element in the cell arrays as data from one cross-validation fold and computes pointwise confidence intervals for the performance metrics. The length of `Weights{i}` and the length of `Labels{i}` must be equal. Using cross-validated data requires Statistics and Machine Learning Toolbox.

`rocmetrics` weighs the observations in `Labels` and `Scores` with the corresponding values in `Weights`. If you set the `NumBootstraps` value to a positive integer, `rocmetrics` draws samples with replacement, using the weights as multinomial sampling probabilities. Using the `NumBootstraps` name-value argument requires Statistics and Machine Learning Toolbox.

By default, `Weights` is a vector of ones or a cell array containing vectors of ones.

This argument sets the `Weights` property.

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

### Options for Confidence Intervals

#### Alpha — Significance level

0.05 (default) | scalar in the range (0,1)

Significance level for the pointwise confidence intervals, specified as a scalar in the range (0,1).

If you specify `Alpha` as  $\alpha$ , then `rocmetrics` computes  $100 \times (1 - \alpha)\%$  pointwise confidence intervals for the performance metrics.

This argument is related to computing confidence intervals. Therefore, it is valid only when you specify cross-validated data for `Labels`, `Scores`, and `Weights`, or when you set the `NumBootstraps` value to a positive integer.

This option requires Statistics and Machine Learning Toolbox.

Example: `Alpha=0.01` specifies 99% confidence intervals.

Data Types: `single` | `double`

#### BootstrapOptions — Bootstrap options for parallel computation

`statset("rocmetrics")` (default) | structure

Bootstrap options for parallel computation, specified as a structure.

You can specify options for computing bootstrap iterations in parallel and setting random numbers during the bootstrap sampling. Create the `BootstrapOptions` structure with `statset`. This table lists the option fields and their values.

Field Name	Field Value	Default
<code>UseParallel</code>	Set this value to <code>true</code> to compute bootstrap iterations in parallel.	<code>false</code>

Field Name	Field Value	Default
UseSubstreams	Set this value to true to run computations in parallel in a reproducible manner.  To compute reproducibly, set Streams to a type that allows substreams: "mlfg6331_64" or "mrg32k3a".	false
Streams	Specify this value as a RandStream object or cell array of such objects. Use a single object except when the UseParallel value is true and the UseSubstreams value is false. In that case, use a cell array that has the same size as the parallel pool.	If you do not specify Streams, then rocmetrics uses the default stream or streams.

This argument is valid only when you specify NumBootstraps as a positive integer to compute confidence intervals using bootstrapping.

This option requires Statistics and Machine Learning Toolbox, and parallel computation requires Parallel Computing Toolbox.

Example: `BootstrapOptions=statset(UseParallel=true)`

Data Types: struct

**BootstrapType – Bootstrap confidence interval type**

"bca" (default) | "corrected percentile" | "normal" | "percentile" | "student"

Bootstrap confidence interval type, specified as one of the values in this table.

Value	Description
"bca"	Bias corrected and accelerated percentile method [8][9]. This method involves a $z_0$ factor computed using the proportion of bootstrap values that are less than the original sample value. To produce reasonable results when the sample is lumpy, the software computes $z_0$ by including half of the bootstrap values that are the same as the original sample value.
"corrected percentile" or "cper"	Bias corrected percentile method [10]
"normal" or "norm"	Normal approximated interval with bootstrapped bias and standard error [11]
"percentile" or "per"	Basic percentile method
"student" or "stud"	Studentized confidence interval [8]

This argument is valid only when you specify NumBootstraps as a positive integer to compute confidence intervals using bootstrapping.



This option requires Statistics and Machine Learning Toolbox.

Example: `BootstrapType="student"`

Data Types: `char` | `string`

### **NumBootstraps — Number of bootstrap samples to draw**

0 (default) | nonnegative integer scalar

Number of bootstrap samples to draw for computing pointwise confidence intervals, specified as a nonnegative integer scalar.

If you specify `NumBootstraps` as a positive integer, then `rocmetrics` uses `NumBootstraps` bootstrap samples. To create each bootstrap sample, the function randomly selects `n` out of the `n` rows of input data with replacement. The default value 0 implies that `rocmetrics` does not use bootstrapping.

`rocmetrics` computes confidence intervals by using either cross-validated data or bootstrap samples. Therefore, if you specify cross-validated data for `Labels`, `Scores`, and `Weights`, then `NumBootstraps` must be 0.

For details, see “Pointwise Confidence Intervals”.

This option requires Statistics and Machine Learning Toolbox.

Example: `NumBootstraps=500`

Data Types: `single` | `double`

### **NumBootstrapsStudentizedSE — Number of bootstrap samples to draw for studentized standard error estimate**

100 (default) | positive integer scalar

Number of bootstrap samples to draw for the studentized standard error estimate, specified as a positive integer scalar.

This argument is valid only when you specify `NumBootstraps` as a positive integer and `BootstrapType` as `"student"` to compute studentized bootstrap confidence intervals. `rocmetrics` estimates the studentized standard error estimate by using `NumBootstrapsStudentizedSE` bootstrap data samples.

This option requires Statistics and Machine Learning Toolbox.

Example: `NumBootstrapsStudentizedSE=500`

Data Types: `single` | `double`

## **Properties**

### **Performance Metrics**

#### **AUC — Area under ROC curve**

numeric vector | numeric matrix

This property is read-only.

Area under the ROC curve (AUC) on page 1-1624, specified as a numeric vector or matrix.

`rocmetrics` computes the AUC for each one-versus-all ROC curve (that is, for each class). The column order of the AUC property value matches the class order in `ClassNames`.

For a binary problem where you specify `Scores` as a two-column matrix, this property is a 1-by-2 vector containing identical AUC values. The AUC values are identical because the overall model performance on one class is identical to the performance on the other class for a binary problem.

If `rocmetrics` computes confidence intervals for AUC, the AUC property value is a matrix in which the first row corresponds to the AUC values, and the second and third rows correspond to the lower and upper bounds, respectively. `rocmetrics` computes confidence intervals for AUC if the function also computes confidence intervals for the performance metrics and you set `FixedMetric` to "Thresholds" (default), "FalsePositiveRate", or "TruePositiveRate". Using confidence intervals requires Statistics and Machine Learning Toolbox.

Data Types: `single` | `double`

### **Metrics — Performance metrics**

table

This property is read-only.

Performance metrics, specified as a table.

The table contains performance metric values for all classes, vertically concatenated according to the class order in `ClassNames`. The table has a row for each unique threshold value for each class. `rocmetrics` determines the threshold values to use based on the value of `FixedMetric`, `FixedMetricValues`, and `UseNearestNeighbor`. For details, see "Thresholds, Fixed Metric, and Fixed Metric Values".

The number of rows for each class in the table is the number of unique threshold values.

Each row of the table contains these variables: `ClassName`, `Threshold`, `FalsePositiveRate`, and `TruePositiveRate`, as well as a variable for each additional metric specified in `AdditionalMetrics`. If you specify a custom metric, `rocmetrics` names the metric "CustomMetricN", where N is the number that refers to the custom metric. For example, "CustomMetric1" corresponds to the first custom metric specified by `AdditionalMetrics`.

Each variable in the `Metrics` table contains a vector or a three-column matrix.

- If `rocmetrics` does not compute confidence intervals, each variable contains a vector.
- If `rocmetrics` computes confidence intervals, both `ClassName` and the variable for `FixedMetric` (`Threshold`, `FalsePositiveRate`, `TruePositiveRate`, or an additional metric) contain a vector, and the other variables contain a three-column matrix. The first column of the matrix corresponds to the metric values, and the second and third columns correspond to the lower and upper bounds, respectively.

Using confidence intervals requires Statistics and Machine Learning Toolbox.

Data Types: `table`

### **Classification Model Properties**

You can specify the following properties when creating a `rocmetrics` object.

#### **ClassNames — Class names**

numeric vector | logical vector | categorical vector | cell array of character vectors

This property is read-only.

Class names, specified as a numeric vector, logical vector, categorical vector, or cell array of character vectors.

For details, see the input argument `ClassNames`, which sets this property. (The software treats character or string arrays as cell arrays of character vectors.)

Data Types: `single` | `double` | `logical` | `cell` | `categorical`

### **Cost — Misclassification cost**

square matrix

This property is read-only.

Misclassification cost, specified as a square matrix.

For details, see the `Cost` name-value argument, which sets this property.

Data Types: `single` | `double`

### **Labels — True class labels**

numeric vector | logical vector | categorical vector | cell array

This property is read-only.

True class labels, specified as a numeric vector, logical vector, categorical vector, cell array of character vectors, or cell array of one of these types for cross-validated data.

For details, see the input argument `Labels`, which sets this property. (The software treats character or string arrays as cell arrays of character vectors.)

Data Types: `single` | `double` | `logical` | `cell` | `categorical`

### **Prior — Prior class probabilities**

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector.

For details, see the `Prior` name-value argument, which sets this property. If you specify this argument as a character vector or string scalar ("`empirical`" or "`uniform`"), `rocmetrics` computes the prior probabilities and stores the `Prior` property as a numeric vector.

Data Types: `single` | `double`

### **Scores — Classification scores**

numeric matrix | cell array of numeric matrices

This property is read-only.

Classification scores, specified as a numeric matrix or a cell array of numeric matrices.

For details, see the input argument `Scores`, which sets this property.

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

**Weights – Observation weights**

numeric vector of positive values | cell array containing numeric vectors of positive values

This property is read-only.

Observation weights, specified as a numeric vector of positive values or a cell array containing numeric vectors of positive values.

For details, see the `Weights` name-value argument, which sets this property.

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

**Object Functions**

`addMetrics` Compute additional classification performance metrics  
`average` Compute performance metrics for average receiver operating characteristic (ROC) curve in multiclass problem  
`plot` Plot receiver operating characteristic (ROC) curves and other performance curves

**Examples****Plot ROC Curve**

Load a sample of predicted classification scores and true labels for a classification problem.

```
load('flowersDataResponses.mat')
```

`trueLabels` is the true labels for an image classification problem and `scores` is the softmax prediction scores. `scores` is an N-by-K array where N is the number of observations and K is the number of classes.

```
trueLabels = flowersData.trueLabels;  
scores = flowersData.scores;
```

Load the class names. The column order of `scores` follows the class order stored in `classNames`.

```
classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels in `trueLabels` and the classification scores in `scores`. Specify the column order of `scores` using `classNames`.

```
rocObj = rocmetrics(trueLabels,scores,classNames);
```

`rocObj` is a `rocmetrics` object that stores the AUC values and performance metrics for each class in the `AUC` and `Metrics` properties. Display the `AUC` property.

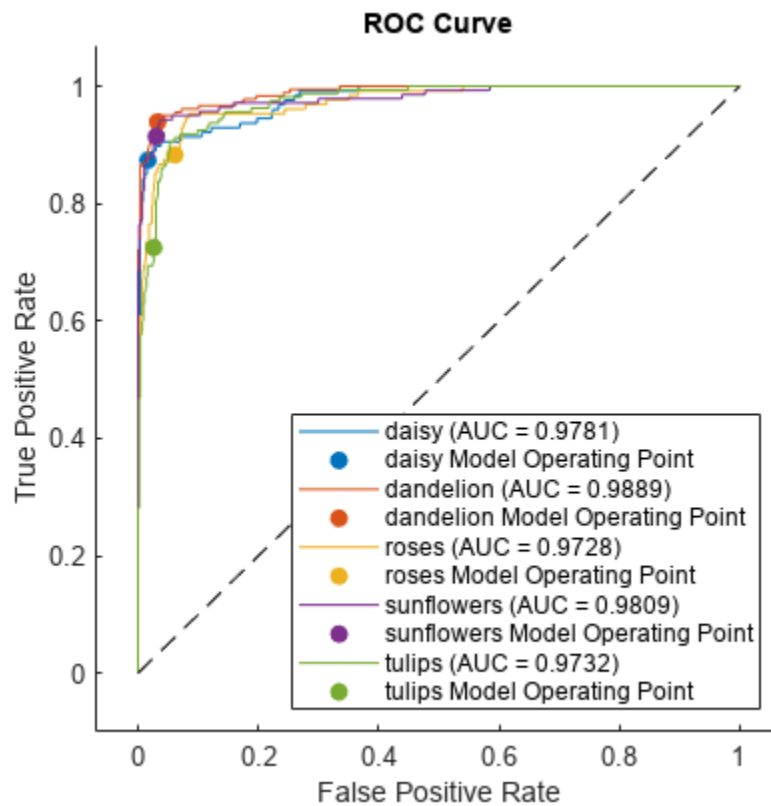
```
rocObj.AUC
```

```
ans = 1x5 single row vector
```

```
    0.9781    0.9889    0.9728    0.9809    0.9732
```

Plot the ROC curve for each class.

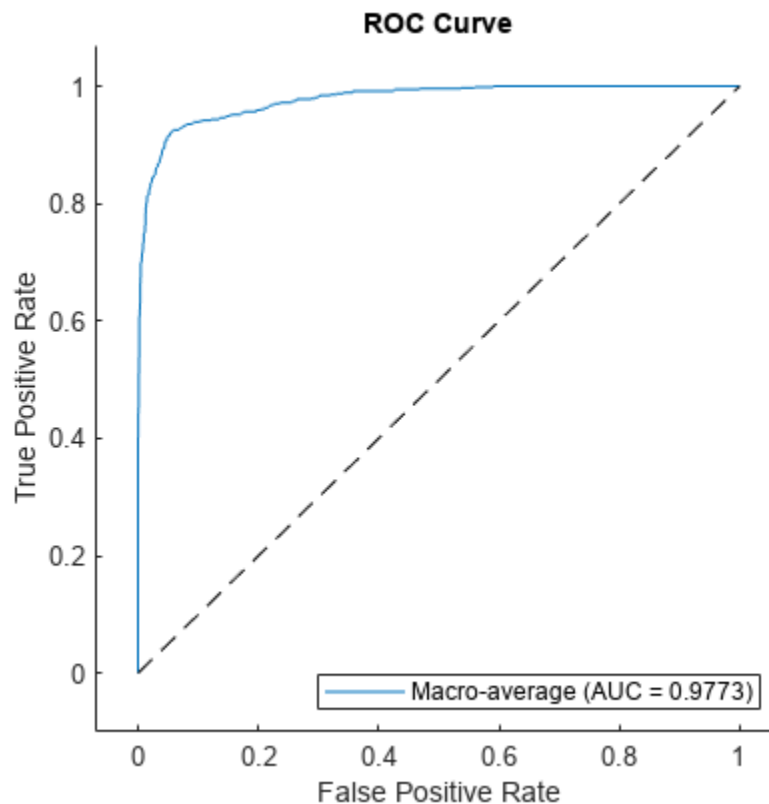
```
plot(rocObj)
```



The filled circle markers indicate the model operating points. The legend displays the class name and AUC value for each curve.

Plot the macro average ROC curve.

```
plot(rocObj, AverageROCType=["macro"], ClassNames=[])
```



### Compute Confidence Intervals Using Bootstrapping

Compute the confidence intervals for FPR and TPR for fixed threshold values by using bootstrap samples, and plot the confidence intervals for TPR on the ROC curve by using the `plot` function. This examples requires Statistics and Machine Learning Toolbox™.

Load a sample of true labels and the prediction scores for a classification problem. For this example, there are five classes: daisy, dandelion, roses, sunflowers, and tulips. The class names are stored in `classNames`. The scores are the softmax prediction scores generated using the `predict` function. `scores` is an N-by-K array where N is the number of observations and K is the number of classes. The column order of `scores` follows the class order stored in `classNames`.

```
load('flowersDataResponses.mat')

scores = flowersData.scores;
trueLabels = flowersData.trueLabels;
predLabels = flowersData.predictedLabels;

classNames = flowersData.classNames;
```

Create a `rocmetrics` object by using the true labels in `trueLabels` and the classification scores in `scores`. Specify the column order of `scores` using `classNames`. Specify `NumBootstraps` as 100 to use 100 bootstrap samples to compute the confidence intervals.

```
rocObj = rocmetrics(trueLabels,scores,classNames,NumBootstraps=100);
```

Find the rows for the second class in the table of the `Metrics` property, and display the first eight rows.

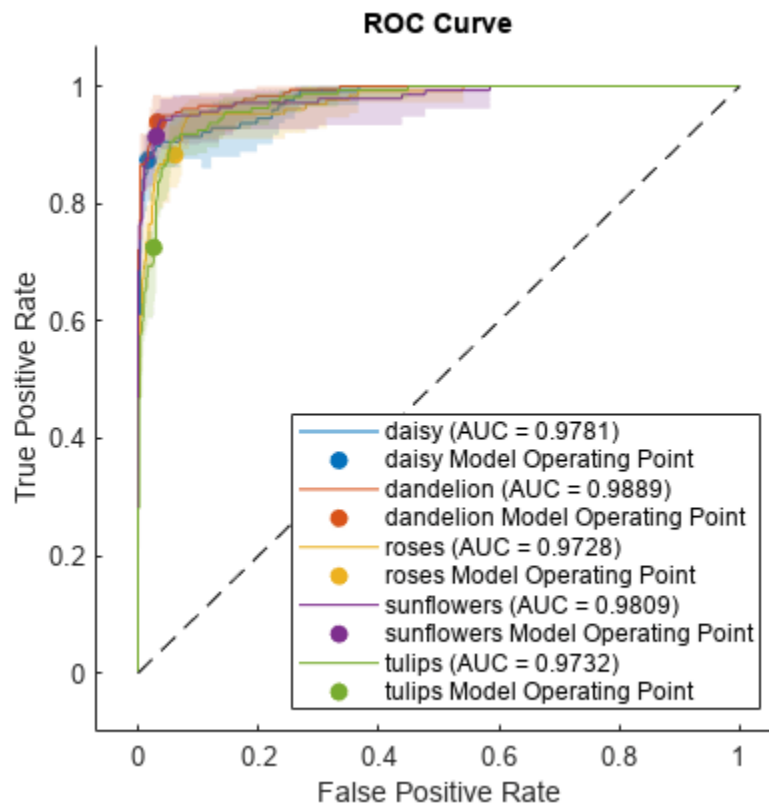
```
idx = rocObj.Metrics.ClassName ==classNames(2);
head(rocObj.Metrics(idx,:))
```

ClassName	Threshold	FalsePositiveRate			TruePositiveRate		
dandelion	1	0	0	0	0	0	0
dandelion	1	0	0	0	0.23889	0.17858	0.31326
dandelion	1	0	0	0	0.26111	0.20107	0.34007
dandelion	1	0	0	0	0.27222	0.21829	0.35778
dandelion	1	0	0	0	0.28889	0.22739	0.36583
dandelion	1	0	0	0	0.29444	0.23682	0.41685
dandelion	1	0	0	0	0.3	0.24296	0.42567
dandelion	1	0	0	0	0.31111	0.24964	0.42614

Each row of the table contains the metric value and its confidence intervals for FPR and TPR for a fixed threshold value. The `Threshold` variable is a column vector, and the `FalsePositiveRate` and `TruePositiveRate` variables are three-column matrices. The first column of the matrices corresponds to the metric values, and the second and third columns correspond to the lower and upper bounds, respectively.

Plot the ROC curve and the confidence intervals for TPR. Specify `ShowConfidenceIntervals=true` to show the confidence intervals.

```
plot(rocObj,ShowConfidenceIntervals=true)
```

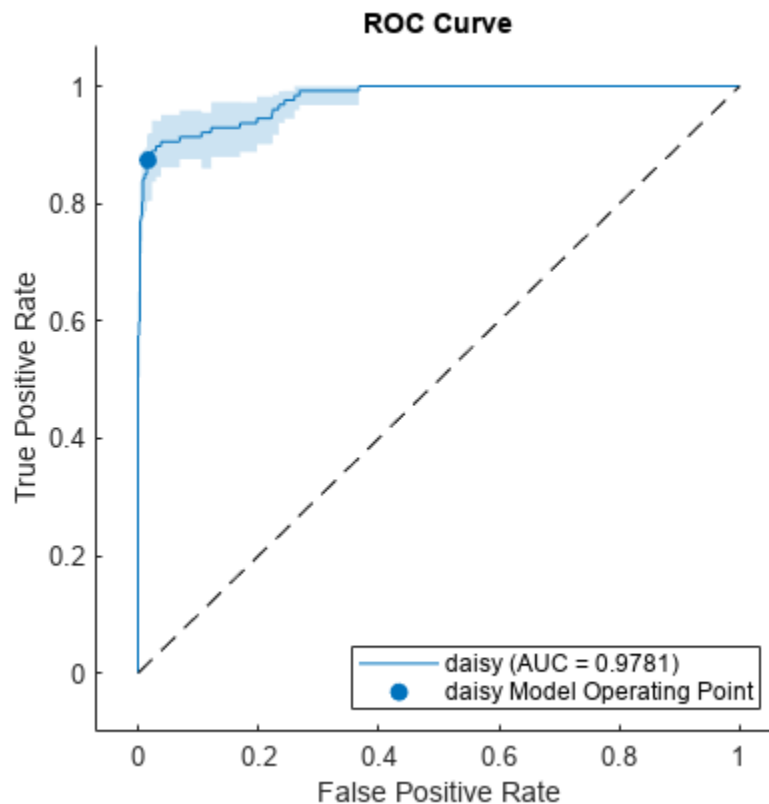


The shaded area around the ROC curve indicates the confidence intervals. The confidence intervals represent the uncertainty of the curve due to the variance in the data set for the trained model.

Specify one class to plot by using the `ClassNames` name-value argument.

```
plot(rocObj, ShowConfidenceIntervals=true, ClassNames="daisy")
```





## More About

### Receiver Operating Characteristic (ROC) Curve

A ROC curve shows the true positive rate versus the false positive rate for different thresholds of classification scores.

The true positive rate and the false positive rate are defined as follows:

- True positive rate (TPR), also known as recall or sensitivity —  $TP / (TP + FN)$ , where TP is the number of true positives and FN is the number of false negatives
- False positive rate (FPR), also known as fallout or 1-specificity —  $FP / (TN + FP)$ , where FP is the number of false positives and TN is the number of true negatives

Each point on a ROC curve corresponds to a pair of TPR and FPR values for a specific threshold value. You can find different pairs of TPR and FPR values by varying the threshold value, and then create a ROC curve using the pairs. For each class, `rocmetrics` uses all distinct adjusted score on page 1-1625 values as threshold values to create a ROC curve.

For a multiclass classification problem, `rocmetrics` formulates a set of one-versus-all on page 1-1624 binary classification problems to have one binary problem for each class, and finds a ROC curve for each class using the corresponding binary problem. Each binary problem assumes one class as positive and the rest as negative.

For a binary classification problem, if you specify the classification scores as a matrix, `rocmetrics` formulates two one-versus-all binary classification problems. Each of these problems treats one class as a positive class and the other class as a negative class, and `rocmetrics` finds two ROC curves. Use one of the curves to evaluate the binary classification problem.

For more details, see “ROC Curve and Performance Metrics”.

### Area Under ROC Curve (AUC)

The area under a ROC curve (AUC) corresponds to the integral of a ROC curve (TPR values) with respect to FPR from  $FPR = 0$  to  $FPR = 1$ .

The AUC provides an aggregate performance measure across all possible thresholds. The AUC values are in the range 0 to 1, and larger AUC values indicate better classifier performance.

### One-Versus-All (OVA) Coding Design

The one-versus-all (OVA) coding design reduces a multiclass classification problem to a set of binary classification problems. In this coding design, each binary classification treats one class as positive and the rest of the classes as negative. `rocmetrics` uses the OVA coding design for multiclass classification and evaluates the performance on each class by using the binary classification that the class is positive.

For example, the OVA coding design for three classes formulates three binary classifications:

	Binary 1	Binary 2	Binary 3
Class 1	1	-1	-1
Class 2	-1	1	-1
Class 3	-1	-1	1

Each row corresponds to a class, and each column corresponds to a binary classification problem. The first binary classification assumes that class 1 is a positive class and the rest of the classes are negative. `rocmetrics` evaluates the performance on the first class by using the first binary classification problem.

### Model Operating Point

The model operating point represents the FPR and TPR corresponding to the typical threshold value.

The typical threshold value depends on the input format of the `Scores` argument (classification scores) specified when you create a `rocmetrics` object:

- If you specify `Scores` as a matrix, `rocmetrics` assumes that the values in `Scores` are the scores for a multiclass classification problem and uses adjusted score on page 1-1625 values. A multiclass classification model classifies an observation into a class that yields the largest score, which corresponds to a nonnegative score in the adjusted scores. Therefore, the threshold value is 0.
- If you specify `Scores` as a column vector, `rocmetrics` assumes that the values in `Scores` are posterior probabilities of the class specified in `ClassNames`. A binary classification model classifies an observation into a class that yields a higher posterior probability, that is, a posterior probability greater than 0.5. Therefore, the threshold value is 0.5.

For a binary classification problem, you can specify `Scores` as a two-column matrix or a column vector. However, if the classification scores are not posterior probabilities, you must specify `Scores` as a matrix. A binary classifier classifies an observation into a class that yields a larger score, which

is equivalent to a class that yields a nonnegative adjusted score. Therefore, if you specify `Scores` as a matrix for a binary classifier, `rocmetrics` can find a correct model operating point using the same scheme that it applies to a multiclass classifier. If you specify classification scores that are not posterior probabilities as a vector, `rocmetrics` cannot identify a correct model operating point because it always uses 0.5 as a threshold for the model operating point.

The `plot` function displays a filled circle marker at the model operating point for each ROC curve (see `ShowModelOperatingPoint`). The function chooses a point corresponding to the typical threshold value. If the curve does not have a data point for the typical threshold value, the function finds a point that has the smallest threshold value greater than the typical threshold. The point on the curve indicates identical performance to the performance of the typical threshold value.

## Algorithms

### Adjusted Scores for Multiclass Classification Problem

For each class, `rocmetrics` adjusts the classification scores (input argument `Scores` of `rocmetrics`) relative to the scores for the rest of the classes if you specify `Scores` as a matrix. Specifically, the adjusted score for a class given an observation is the difference between the score for the class and the maximum value of the scores for the rest of the classes.

For example, if you have  $[s_1, s_2, s_3]$  in a row of `Scores` for a classification problem with three classes, the adjusted score values are  $[s_1 - \max(s_2, s_3), s_2 - \max(s_1, s_3), s_3 - \max(s_1, s_2)]$ .

`rocmetrics` computes the performance metrics using the adjusted score values for each class.

For a binary classification problem, you can specify `Scores` as a two-column matrix or a column vector. Using a two-column matrix is a simpler option because the `predict` function of a classification object returns classification scores as a matrix, which you can pass to `rocmetrics`. If you pass scores in a two-column matrix, `rocmetrics` adjusts scores in the same way that it adjusts scores for multiclass classification, and it computes performance metrics for both classes. You can use the metric values for one of the two classes to evaluate the binary classification problem. The metric values for a class returned by `rocmetrics` when you pass a two-column matrix are equivalent to the metric values returned by `rocmetrics` when you specify classification scores for the class as a column vector.

## Version History

Introduced in R2022b

## References

- [1] Fawcett, T. "ROC Graphs: Notes and Practical Considerations for Researchers", *Machine Learning* 31, no. 1 (2004): 1-38.
- [2] Zweig, M., and G. Campbell. "Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine." *Clinical Chemistry* 39, no. 4 (1993): 561-577 .
- [3] Davis, J., and M. Goadrich. "The Relationship Between Precision-Recall and ROC Curves." *Proceedings of ICML '06*, 2006, pp. 233-240.
- [4] Moskowitz, C. S., and M. S. Pepe. "Quantifying and Comparing the Predictive Accuracy of Continuous Prognostic Factors for Binary Outcomes." *Biostatistics* 5, no. 1 (2004): 113-27.

- [5] Huang, Y., M. S. Pepe, and Z. Feng. "Evaluating the Predictiveness of a Continuous Marker." *U. Washington Biostatistics Paper Series*, 2006, 250-61.
- [6] Briggs, W. M., and R. Zaretzki. "The Skill Plot: A Graphical Technique for Evaluating Continuous Diagnostic Tests." *Biometrics* 64, no. 1 (2008): 250-256.
- [7] Bettinger, R. "Cost-Sensitive Classifier Selection Using the ROC Convex Hull Method." *SAS Institute*, 2003.
- [8] DiCiccio, Thomas J., and Bradley Efron. "Bootstrap Confidence Intervals." *Statistical Science* 11, no. 3 (1996): 189-228.
- [9] Efron, Bradley, and Robert J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.
- [10] Efron, Bradley. *The Jackknife, the Bootstrap and Other Resampling Plans*. Philadelphia: The Society for Industrial and Applied Mathematics, 1982.
- [11] Davison, A. C., and D. V. Hinkley. *Bootstrap Methods and Their Applications*. Cambridge University Press, 1997.

## **See Also**

ROCCurve Properties | confusionchart

## **Topics**

"ROC Curve and Performance Metrics"

"Compare Deep Learning Models Using ROC Curves"

"Out-of-Distribution Detection for Deep Neural Networks"

# selfAttentionLayer

Self-attention layer

## Description

A self-attention layer computes single-head or multihead self-attention of its input.

The layer:

- 1 Computes the queries, keys, and values from the input
- 2 Computes the scaled dot-product attention across heads using the queries, keys, and values
- 3 Merges the results from the heads
- 4 Performs a linear transformation on the merged result

## Creation

### Syntax

```
layer = selfAttentionLayer(numHeads, numKeyChannels)
layer = selfAttentionLayer(numHeads, numKeyChannels, Name=Value)
```

### Description

`layer = selfAttentionLayer(numHeads, numKeyChannels)` creates a self-attention layer and sets the `NumHeads` and `NumKeyChannels` properties.

`layer = selfAttentionLayer(numHeads, numKeyChannels, Name=Value)` sets the optional `NumValueChannels`, `OutputSize`, `HasPaddingMaskInput`, `AttentionMask`, `DropoutProbability`, `HasScoresOutput`, “Parameters and Initialization” on page 1-1629, “Learning Rate and Regularization” on page 1-1631, and `Name` properties.

## Properties

### Self-Attention

#### **NumHeads** — Number of attention heads

positive integer

This property is read-only.

Number of attention heads, specified as a positive integer that evenly divides `NumKeyChannels`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **NumKeyChannels** — Number of channels for keys and queries

positive integer

This property is read-only.

Number of channels for the keys and queries, specified as a positive integer that is divisible by `NumHeads`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **NumValueChannels — Number of channels for values**

`"auto"` (default) | positive integer

Number of channels for the values, specified as one of these values:

- `"auto"` — Use `NumKeyChannels`.
- Positive integer — Use the specified number of channels. This value must be divisible by `NumHeads`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **OutputSize — Number of channels of layer output**

`"auto"` (default) | positive integer

Number of channels of the layer output, specified as one of these values:

- `"auto"` — Use the number of channels in the layer input.
- Positive integer — Use the specified number of channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **HasPaddingMaskInput — Flag indicating whether layer has mask input**

`0` (`false`) (default) | `1` (`true`)

Flag indicating whether the layer has an input that represents the padding mask, specified as `0` (`false`) or `1` (`true`).

If the `HasPaddingMaskInput` property is `0` (`false`), then the layer has one input with the name `'in'`, which corresponds to the input data. In this case, the layer treats all elements as data.

If the `HasPaddingMaskInput` property is `1` (`true`), then the layer has two inputs with the names `'in'` and `'mask'`, which correspond to the input data and the mask, respectively. In this case, the padding mask is an array of ones and zeros. The layer prevents or allows attention to elements in the key-value pairs when the corresponding element in the mask is zero or one, respectively.

### **AttentionMask — Mask preventing attention to elements in key-value pairs**

`"none"` (default) | `"causal"`

Mask preventing attention to elements in key-value pairs, specified as one of these values:

- `"none"` — Do not prevent attention to elements based on their positions. If `HasPaddingMaskInput` is `1` (`true`), then the layer prevents attention to padding elements only.
- `"causal"` — Prevent elements in position `M` from attending to elements in position `N`, where `N` is greater than `M`. Use this option for auto-regressive models.

### **DropoutProbability — Dropout probability for attention scores**

`0` (default) | scalar in the range `[0, 1)`

Dropout probability for attention scores, specified as a scalar in the range [0, 1).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### HasScoresOutput — Flag indicating whether layer has scores output

0 (false) (default) | 1 (true)

Flag indicating whether the layer has an output that represents the attention scores, specified as one of these values:

- 0 (false) — The layer does not have an output that represents the attention scores.
- 1 (true) — The layer has an output with the name "scores" that represents the attention scores.

### InputSize — Number of input channels

"auto" (default) | positive integer

This property is read-only.

Number of input channels, specified as one of these values:

- "auto" — Automatically determine the number of input channels when you initialize the network
- Positive integer — Configure the layer for the specified number of input channels. `InputSize` and the number of channels in the layer input data must match.

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 query, key, value, and output weights, specified as one of these values:

- "glorot" - Initialize the weights with the Glorot initializer (also known as Xavier initializer) [2]. The Glorot initializer independently samples from a uniform distribution with zero mean and a variance of  $2 / (\text{numIn} + \text{numOut})$ . The values of `numIn` and `numOut` depend on the weight matrix:

Weight	numIn	numOut
Query	InputSize	NumKeyChannels
Key	InputSize	NumKeyChannels
Value	InputSize	NumValueChannels
Output	NumValueChannels	OutputSize

- "he" - Initialize the weights with the He initializer [3]. The He initializer samples from a normal distribution with zero mean and a variance of  $2 / \text{numIn}$ . The values of `numIn` and `numOut` depend on the weight matrix:

Weight	numIn	numOut
Query	InputSize	NumKeyChannels
Key	InputSize	NumKeyChannels

Weight	numIn	numOut
Value	InputSize	NumValueChannels
Output	NumValueChannels	OutputSize

- "narrow-normal" — Initialize the weights by independently sampling from a normal distribution with zero mean 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 have 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 corresponding weights property is empty.

Data Types: `char` | `string` | `function_handle`

### **BiasInitializer** — Function to initialize biases

"zeros" (default) | "narrow-normal" | "ones" | function handle

Function to initialize the query, key, value, and output biases, specified as one of these values:

- "zeros" — Initialize the biases with zeros.
- "ones" — Initialize the biases with ones.
- "narrow-normal" — Initialize the biases by independently sampling from a normal distribution with zero mean and a standard deviation of 0.01.
- Function handle — Initialize the biases with a custom function. If you specify a function handle, then the function must have the form `bias = func(sz)`, where `sz` is the size of the biases.

The layer only initializes the biases when the corresponding bias property is empty.

Data Types: `char` | `string` | `function_handle`

### **QueryWeights** — Query weights

[] (default) | matrix

Query weights, specified as a `NumKeyChannels`-by-`numInputChannels` matrix or [], where `numInputChannels` is the number of channels in the layer input.

Data Types: `single` | `double`

### **KeyWeights** — Key weights

[] (default) | matrix

Key weights, specified as a `NumKeyChannels`-by-`numInputChannels` matrix or [], where `numInputChannels` is the number of channels in the layer input.

Data Types: `single` | `double`

### **ValueWeights** — Value weights

[] (default) | matrix

Value weights, specified as a `NumValueChannels`-by-`numInputChannels` matrix or [], where `numInputChannels` is the number of channels in the layer input.



Data Types: `single` | `double`

### **OutputWeights — Output weights**

`[]` (default) | matrix

Output weights, specified as an `OutputSize`-by-`NumValueChannels` matrix or `[]`.

Data Types: `single` | `double`

### **QueryBias — Query biases**

`[]` (default) | vector

Query biases, specified as a `NumKeyChannels`-by-1 vector or `[]`.

Data Types: `single` | `double`

### **KeyBias — Key biases**

`[]` (default) | vector

Key biases, specified as a `NumKeyChannels`-by-1 vector or `[]`.

Data Types: `single` | `double`

### **ValueBias — Value biases**

`[]` (default) | vector

Value biases, specified as a `NumValueChannels`-by-1 vector or `[]`.

Data Types: `single` | `double`

### **OutputBias — Output biases**

`[]` (default) | vector

Output biases, specified as an `OutputSize`-by-1 vector or `[]`.

Data Types: `single` | `double`

## **Learning Rate and Regularization**

### **WeightLearnRateFactor — Learning rate factor for weights**

1 (default) | nonnegative scalar

Learning rate factor for the query, key, value, and output 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 query, key, value, and output 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 query, key, value, and output 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 query, key, value, and output 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

Data Types: `char` | `string`

### **NumInputs — Number of inputs**

1 | 2

This property is read-only.

Number of inputs to the layer.

If the `HasPaddingMaskInput` property is 0 (`false`), then the layer has one input with the name 'in', which corresponds to the input data. In this case, the layer treats all elements as data.

If the `HasPaddingMaskInput` property is 1 (`true`), then the layer has two inputs with the names 'in' and 'mask', which correspond to the input data and the mask, respectively. In this case, the padding mask is an array of ones and zeros. The layer prevents or allows attention to elements in the key-value pairs when the corresponding element in the mask is zero or one, respectively.

Data Types: double

### InputNames — Input names

`{'in'} | {'in', 'mask'}`

This property is read-only.

Input names of the layer.

If the `HasPaddingMaskInput` property is `0` (`false`), then the layer has one input with the name `'in'`, which corresponds to the input data. In this case, the layer treats all elements as data.

If the `HasPaddingMaskInput` property is `1` (`true`), then the layer has two inputs with the names `'in'` and `'mask'`, which correspond to the input data and the mask, respectively. In this case, the padding mask is an array of ones and zeros. The layer prevents or allows attention to elements in the key-value pairs when the corresponding element in the mask is zero or one, respectively.

### NumOutputs — Number of outputs

`1 | 2`

This property is read-only.

Number of outputs of the layer.

If the `HasScoresOutput` property is `0` (`false`), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasScoresOutput` property is `1` (`true`), then the layer has two outputs with the names `'out'` and `'scores'`, which correspond to the output data and the attention scores, respectively.

Data Types: double

### OutputNames — Output names

`{'out'} | {'out', 'scores'}`

This property is read-only.

Output names of the layer.

If the `HasScoresOutput` property is `0` (`false`), then the layer has one output with the name `'out'`, which corresponds to the output data.

If the `HasScoresOutput` property is `1` (`true`), then the layer has two outputs with the names `'out'` and `'scores'`, which correspond to the output data and the attention scores, respectively.

## Examples

### Create Self-Attention Layer

Create a self-attention layer with eight heads and 256 key and query channels.

```
layer = selfAttentionLayer(8,256)
layer =
  SelfAttentionLayer with properties:
```

```
        Name: ''
        AttentionMask: 'none'
HasPaddingMaskInput: 0
HasScoresOutput: 0

Hyperparameters
    InputSize: 'auto'
    NumHeads: 8
    NumKeyChannels: 256
    NumValueChannels: 'auto'
    OutputSize: 'auto'
    DropoutProbability: 0

Learnable Parameters
    QueryWeights: []
    KeyWeights: []
    ValueWeights: []
    OutputWeights: []
    QueryBias: []
    KeyBias: []
    ValueBias: []
    OutputBias: []

Show all properties
```

Include a self-attention layer in a layer array.

```
layers = [
    sequenceInputLayer(12)
    selfAttentionLayer(4,12)
    layerNormalization
    fullyConnectedLayer(9)
    softmaxLayer];
```

## Version History

**Introduced in R2023a**

## References

- [1] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (December 2017): 6000-6010. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>.
- [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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## **See Also**

`trainingOptions` | `sequenceInputLayer` | `lstmLayer` | `fullyConnectedLayer`

## **Topics**

“Long Short-Term Memory Neural Networks”  
“Time Series Forecasting Using Deep Learning”  
“List of Deep Learning Layers”  
“Deep Learning in MATLAB”  
“Deep Learning Tips and Tricks”  
“Data Sets for Deep Learning”

# 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 1-1637.

## 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 the name ''.

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')
```



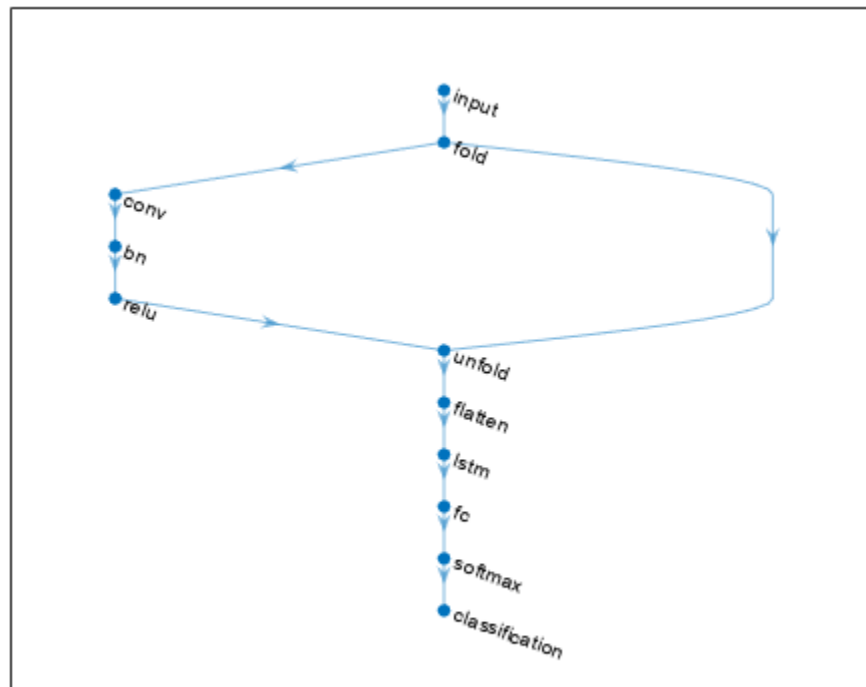
```
softmaxLayer('Name','softmax')
classificationLayer('Name','classification')];
```

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)
```



## Version History

Introduced in R2019a

## 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**

[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 Neural Networks”](#)  
[“Specify Layers of Convolutional Neural Network”](#)  
[“Set Up Parameters and Train Convolutional Neural Network”](#)  
[“Deep Learning in MATLAB”](#)  
[“List of Deep Learning Layers”](#)

# sequenceInputLayer

Sequence input layer

## Description

A sequence input layer inputs sequence data to a neural 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  $Y = \text{func}(X)$ , where  $X$  is the input data and the output  $Y$  is the normalized data.

---

**Tip** The software, by default, automatically calculates the normalization statistics when using the `trainNetwork` function. To save time when training, specify the required statistics for normalization and set the `ResetInputNormalization` option in `trainingOptions` to 0 (false).

---

The software applies normalization to all input elements, including padding values.

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 `trainNetwork` function calculates the mean and ignores padding values. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Mean` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the mean and ignores padding values. To train a `dlnetwork` object using a custom training loop or assemble a network

without training it using the `assembleNetwork` function, you must set the `StandardDeviation` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the minima and ignores padding values. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Min` property to a numeric scalar or a numeric array.

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 `trainNetwork` function calculates the maxima and ignores padding values. To train a `dlnetwork` object using a custom training loop or assemble a network without training it using the `assembleNetwork` function, you must set the `Max` property to a numeric scalar or a numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SplitComplexInputs — Flag to split input data into real and imaginary components**

`0` (`false`) (default) | `1` (`true`)

This property is read-only.

Flag to split input data into real and imaginary components specified as one of these values:

- `0` (`false`) - Do not split input data.
- `1` (`true`) - Split data into real and imaginary components.

When `SplitComplexInputs` is 1, then the layer outputs twice as many channels as the input data. For example, if the input data is complex-values with `numChannels` channels, then the layer outputs data with  $2 * \text{numChannels}$  channels, where channels 1 through `numChannels` contain the real components of the input data and `numChannels+1` through  $2 * \text{numChannels}$  contain the imaginary components of the input data. If the input data is real, then channels `numChannels+1` through  $2 * \text{numChannels}$  are all zero.

To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

For an example showing how to train a network with complex-valued data, see “Train Network with Complex-Valued Data”.

## 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 the name ' '.

Data Types: `char` | `string`

### NumInputs — Number of inputs

0 (default)

This property is read-only.

Number of inputs of the layer. The layer has no inputs.

Data Types: `double`

### InputNames — Input names

{ } (default)

This property is read-only.

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  
      SplitComplexInputs: 0  
  
  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  
        SplitComplexInputs: 0  
  
Hyperparameters  
    Normalization: 'none'  
    NormalizationDimension: 'auto'
```

## Train Network for Sequence Classification

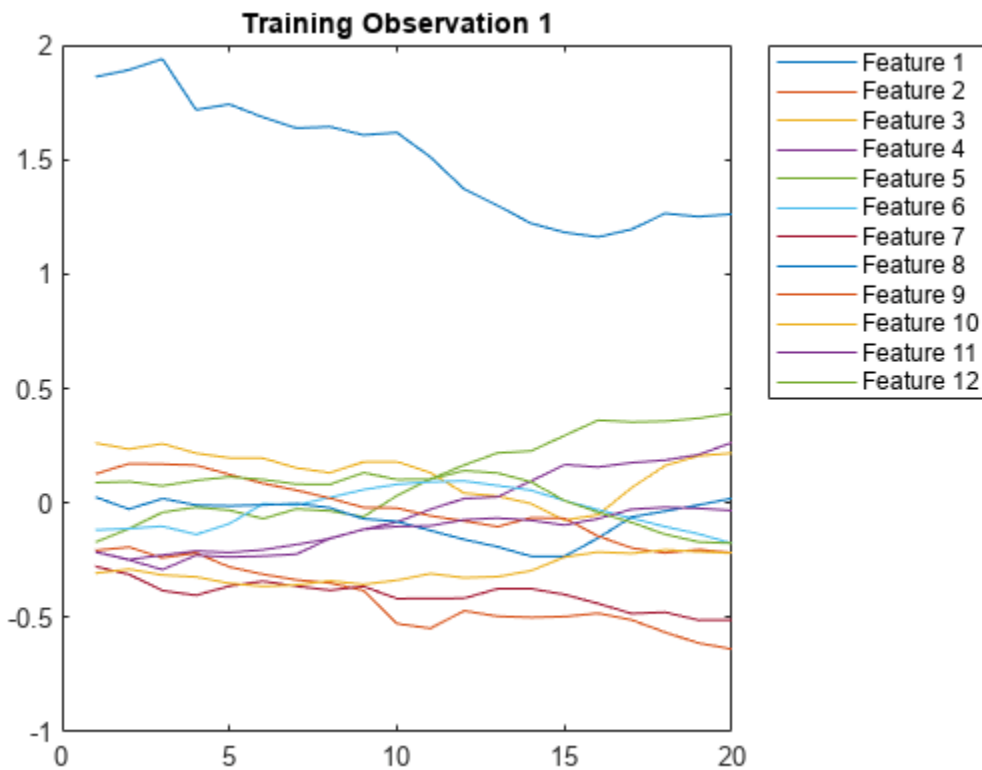
Train a deep learning LSTM network for sequence-to-label classification.

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.

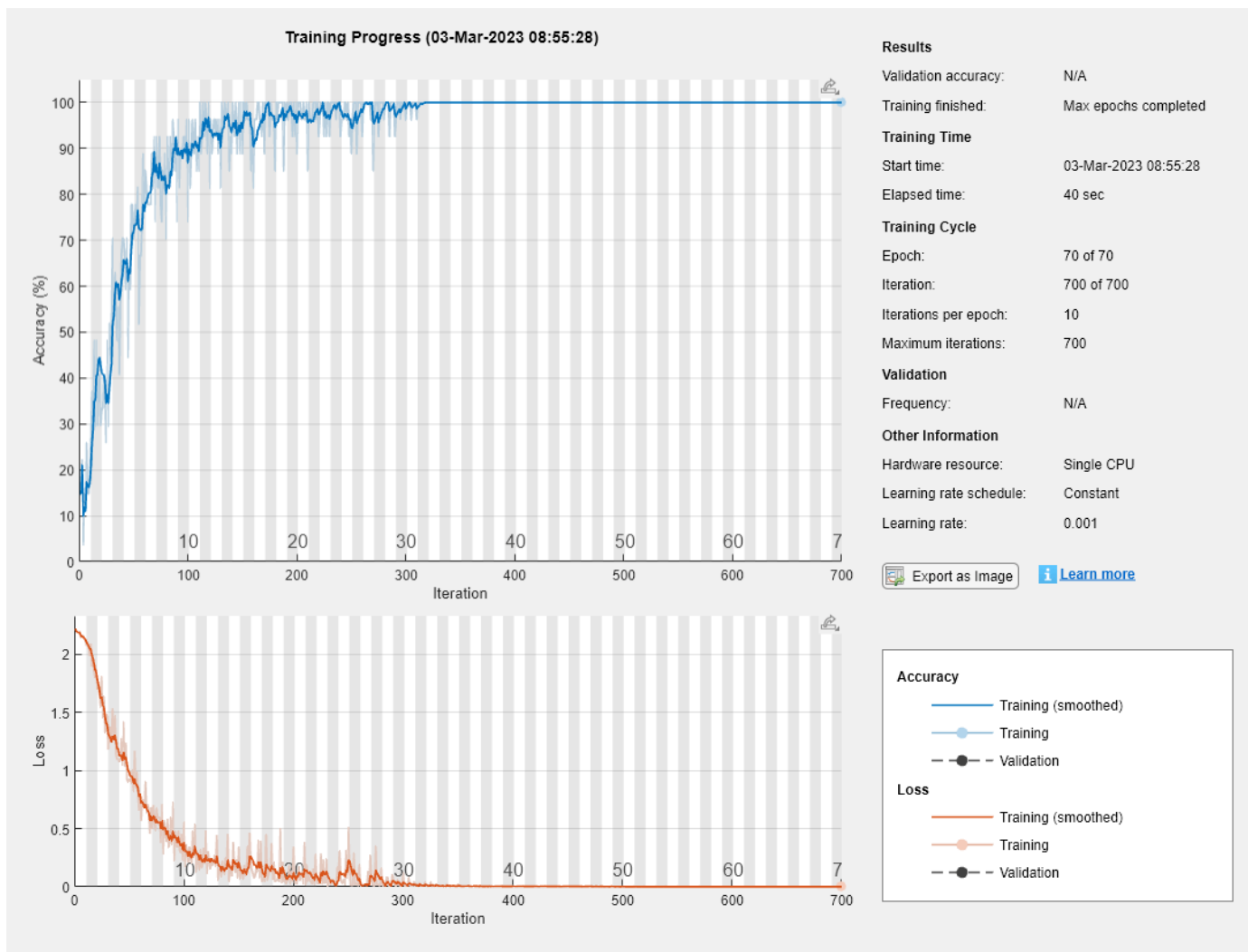
Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

```
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 = 0.9432
```

### **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')

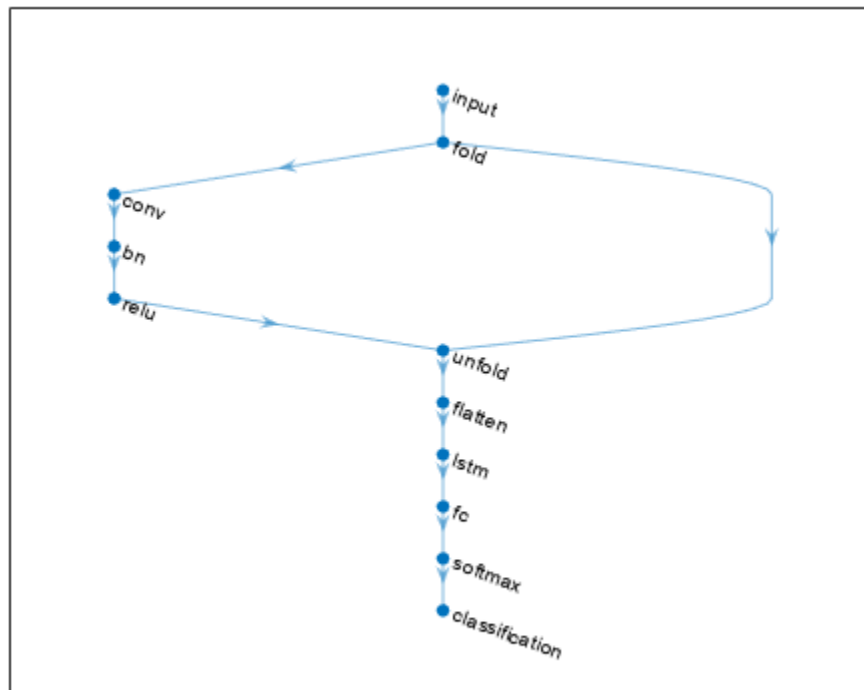
fullyConnectedLayer(numClasses, 'Name', 'fc')
softmaxLayer('Name', 'softmax')
classificationLayer('Name', 'classification')];
```

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)
```



## Version History

Introduced in R2017b

### **R2019b: 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'.

### **R2020a: trainNetwork ignores padding values when calculating normalization statistics**

*Behavior changed in R2020a*

Starting in R2020a, `trainNetwork` ignores padding values when calculating normalization statistics. 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.
- For code generation, the input data must contain either zero or two spatial dimensions.
- Code generation does not support 'Normalization' specified using a function handle.
- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

### 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.
- Code generation does not support complex input and does not support 'SplitComplexInputs' option.

## See Also

`trainNetwork` | `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 Neural Networks”  
“Specify Layers of Convolutional Neural Network”  
“Set Up Parameters and Train Convolutional Neural Network”  
“Deep Learning in MATLAB”  
“List of Deep Learning Layers”

# 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 1-1658.

## 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 the name `''`.

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')
```

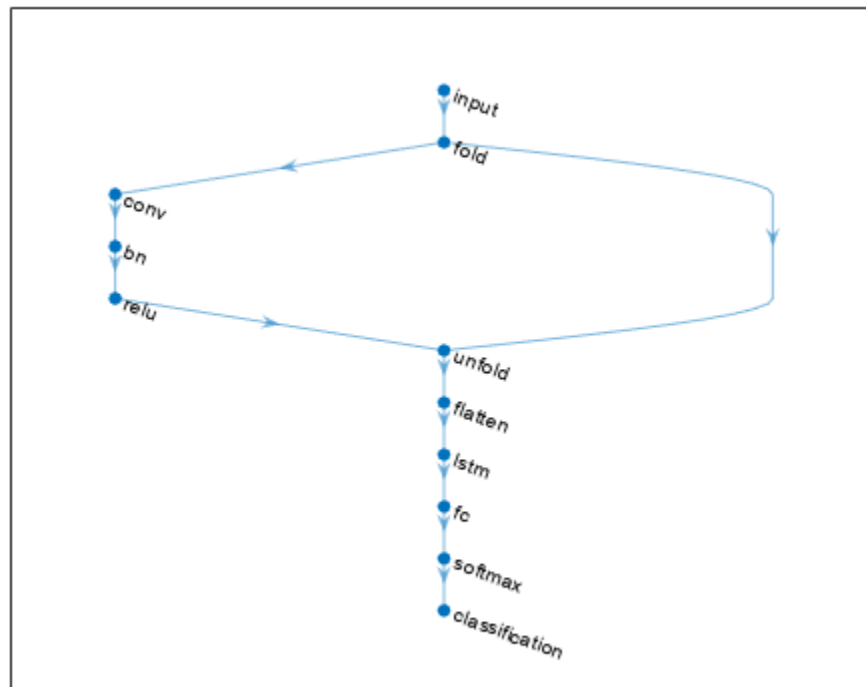
```
softmaxLayer('Name','softmax')  
classificationLayer('Name','classification')];
```

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)
```



## Version History

Introduced in R2019a

### 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**

[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 Neural 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"](#)

# 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 1-1663.
- Train or fine-tune a network using `trainNetwork`. For an example, see “Train Network for Image Classification” on page 1-1664.
- 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 1-983.
  - For a Caffe model, use `importCaffeNetwork`. For an example, see “Import Caffe Network” on page 1-960.
  - For an ONNX model, use `importONNXNetwork`. For an example, see “Import ONNX Network as DAGNetwork” on page 1-1061.
- 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 trained deep learning neural network
predict	Predict responses using 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 state parameters of neural network
plot	Plot neural network architecture

## 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: [25x1 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' normalizati
   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 pe
   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 pe
   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 stri
13 'relu4'    ReLU                      ReLU
14 'conv5'    Grouped Convolution      2 groups of 128 3x3x192 convolutions with stri
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 clas
```

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 = 10x1 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

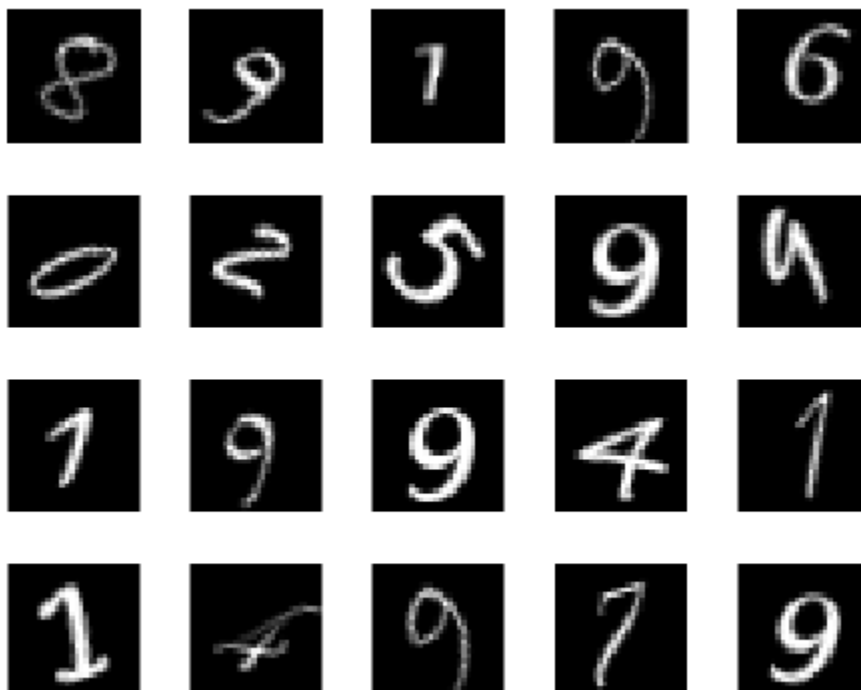
Load the data as an ImageDatastore object.

```
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.

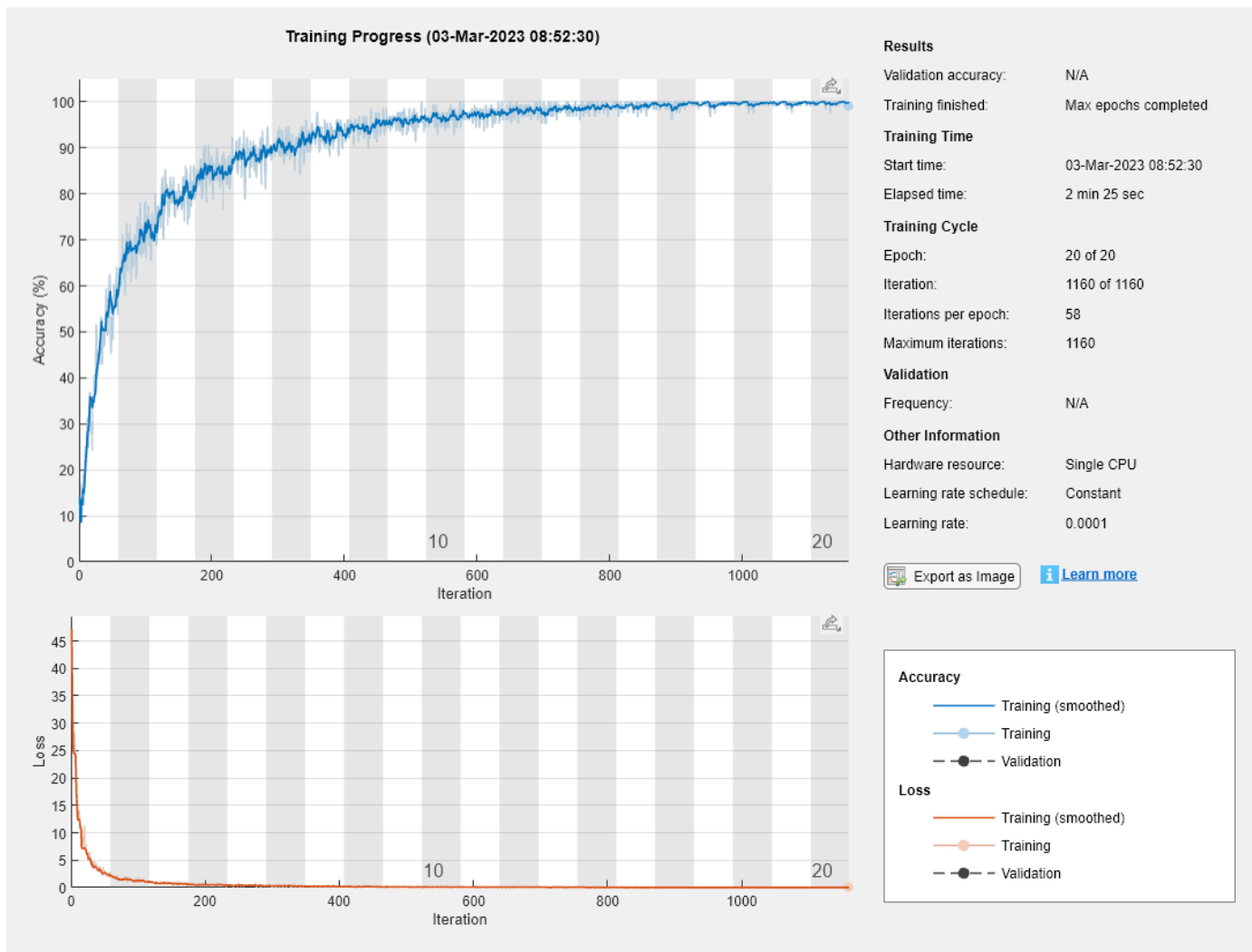
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

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 = 0.9416
```

## Version History

### Introduced in R2016a

## 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 Neural 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 Neural Networks”

# setL2Factor

**Package:** `nnet.cnn.layer`

Set L2 regularization factor of layer learnable parameter

## Syntax

```
layerUpdated = setL2Factor(layer, parameterName, factor)
layerUpdated = setL2Factor(layer, parameterPath, factor)

netUpdated = setL2Factor(net, layerName, parameterName, factor)
netUpdated = setL2Factor(net, parameterPath, factor)
```

## Description

`layerUpdated = 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.

`netUpdated = setL2Factor(net, 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.

`netUpdated = setL2Factor(net, 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
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
      Network: [1x1 dlnetwork]
```

```
Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:
```

1	'conv_1'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
2	'batchnorm_1'	Batch Normalization	Batch normalization
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
5	'batchnorm_2'	Batch Normalization	Batch normalization
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the L2 regularization factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the setL2Factor function.



```
factor = 2;
layer = setL2Factor(layer, 'Network/conv_1/Weights', factor);
```

Get the updated L2 regularization factor using the `getL2Factor` function.

```
factor = getL2Factor(layer, 'Network/conv_1/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/conv_1/Weights"	{3x3x32x32 dlarray}
"res1"	"Network/conv_1/Bias"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_1/Offset"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_1/Scale"	{1x1x32 dlarray}
"res1"	"Network/conv_2/Weights"	{3x3x32x32 dlarray}
"res1"	"Network/conv_2/Bias"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_2/Offset"	{1x1x32 dlarray}
"res1"	"Network/batchnorm_2/Scale"	{1x1x32 dlarray}

For the layer "res1", set the L2 regularization factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the `setL2Factor` function.

```

factor = 2;
dlnet = setL2Factor(dlnet,'res1/Network/conv_1/Weights',factor);

```

Get the updated L2 regularization factor using the `getL2Factor` function.

```

factor = getL2Factor(dlnet,'res1/Network/conv_1/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 `net`.

Data Types: `char` | `string`

**net — Neural network**

`dlnetwork` object

Neural network, 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`.

**netUpdated — Updated network**

`dlnetwork` object

Updated network, returned as a `dlnetwork`.

**Version History**

**Introduced in R2017b**

**See Also**

`setLearnRateFactor` | `getLearnRateFactor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

**Topics**

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“Define Custom Deep Learning Layers”

# setLearnRateFactor

**Package:** `nnet.cnn.layer`

Set learn rate factor of layer learnable parameter

## Syntax

```
layerUpdated = setLearnRateFactor(layer, parameterName, factor)
layerUpdated = setLearnRateFactor(layer, parameterPath, factor)

netUpdated = setLearnRateFactor(net, layerName, parameterName, factor)
netUpdated = setLearnRateFactor(net, 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.

`netUpdated = setLearnRateFactor(net, 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.

`netUpdated = setLearnRateFactor(net, parameterPath, factor)` sets 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
```

```
fullyConnectedLayer(10)
softmaxLayer
classificationLayer];
```

Set the learn rate factor of the Alpha learnable parameter of the preLU layer 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
    Network: [1x1 dlnetwork]
```

```
  Show all properties
```

View the layers of the nested network.

```
layer.Network.Layers
```

```
ans =
  7x1 Layer array with layers:
```

1	'conv_1'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
2	'batchnorm_1'	Batch Normalization	Batch normalization
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	64 3x3 convolutions with stride [1 1] and padding
5	'batchnorm_2'	Batch Normalization	Batch normalization
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the learning rate factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the `setLearnRateFactor` function.

```
factor = 2;
layer = setLearnRateFactor(layer, 'Network/conv_1/Weights', factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(layer, 'Network/conv_1/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	'conv_1'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding
2	'batchnorm_1'	Batch Normalization	Batch normalization with 32 channels
3	'relu_1'	ReLU	ReLU
4	'conv_2'	2-D Convolution	32 3x3x32 convolutions with stride [1 1] and padding
5	'batchnorm_2'	Batch Normalization	Batch normalization with 32 channels
6	'add'	Addition	Element-wise addition of 2 inputs
7	'relu_2'	ReLU	ReLU

Set the learning rate factor of the learnable parameter 'Weights' of the layer 'conv\_1' to 2 using the `setLearnRateFactor` function.

```
factor = 2;
dlnet = setLearnRateFactor(dlnet,'res1/Network/conv_1/Weights',factor);
```

Get the updated learning rate factor using the `getLearnRateFactor` function.

```
factor = getLearnRateFactor(dlnet,'res1/Network/conv_1/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)
```

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] = sgdupdate(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 `net`.

Data Types: char | string

### **net** — Neural network

`dlnetwork` object

Neural network, 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.

### **netUpdated** — Updated network

`dlnetwork` object

Updated network, returned as a `dlnetwork`.

## Version History

**Introduced in R2017b**

## See Also

`setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `trainNetwork` | `trainingOptions`

## Topics

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“Define Custom Deep Learning Layers”

## sgdmupdate

Update parameters using stochastic gradient descent with momentum (SGDM)

### Syntax

```
[netUpdated,vel] = sgdmupdate(net,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.
- 

`[netUpdated,vel] = sgdmupdate(net,grad,vel)` updates the learnable parameters of the network `net` 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,TTrain] = digitTrain4DArrayData;
classes = categories(TTrain);
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], 'Mean', mean(XTrain,4))
    convolution2dLayer(5,20)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    convolution2dLayer(3,20, 'Padding',1)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer];
```

Create a `dlnetwork` object from the layer array.

```
net = dlnetwork(layers);
```

### Define Model Loss Function

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

### Specify Training Options

Specify the options to use during training.

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

## Train Network

Initialize the velocity parameter.

```
vel = [];
```

Calculate the total number of iterations for the training progress monitor.

```
numIterations = numEpochs * numIterationsPerEpoch;
```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object close to the training loop.

```
monitor = trainingProgressMonitor(Metrics="Loss", Info="Epoch", XLabel="Iteration");
```

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 iteration, display the training progress.

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

```
iteration = 0;
```

```
epoch = 0;
```

```
while epoch < numEpochs && ~monitor.Stop
    epoch = epoch + 1;

    % Shuffle data.
    idx = randperm(numel(TTrain));
    XTrain = XTrain(:,:,,idx);
    TTrain = TTrain(idx);

    i = 0;
    while i < numIterationsPerEpoch && ~monitor.Stop
        i = i + 1;
        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);

        T = zeros(numClasses, miniBatchSize,"single");
        for c = 1:numClasses
            T(c,TTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to a darray.
        X = darray(single(X),"SSCB");

        % If training on a GPU, then convert data to a gpuArray.
        if canUseGPU
            X = gpuArray(X);
        end

        % Evaluate the model loss and gradients using dlfeval and the
```

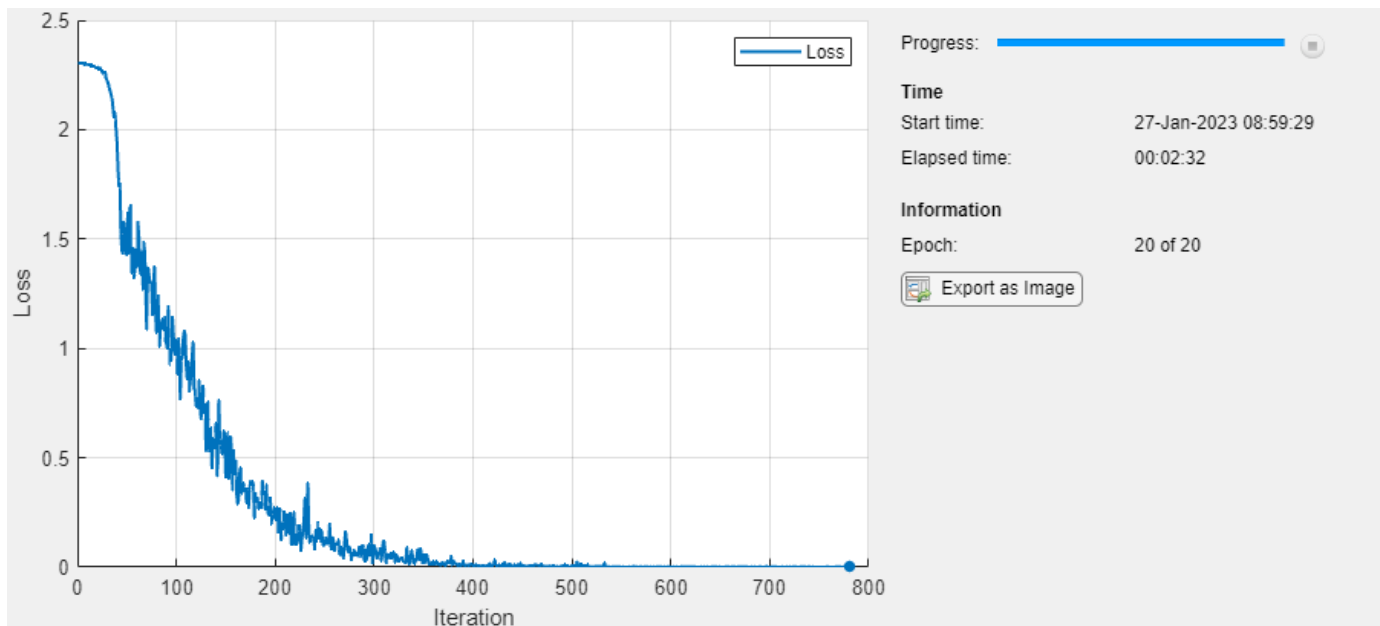
```

% modelLoss function.
[loss,gradients] = dlfeval(@modelLoss,net,X,T);

% Update the network parameters using the SGDM optimizer.
[net,vel] = sgdmupdate(net,gradients,vel);

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch + " of " + numEpochs);
monitor.Progress = 100 * iteration/numIterations;
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,TTest] = digitTest4DArrayData;
```

Convert the data to a `dlarray` with the dimension format "SSCB" (spatial, spatial, channel, batch). For GPU prediction, also convert the data to a `gpuArray`.

```

XTest = dlarray(XTest,"SSCB");
if canUseGPU
    XTest = gpuArray(XTest);
end

```

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

```

YTest = predict(net,XTest);
[~,idx] = max(extractdata(YTest),[],1);
YTest = classes(idx);

```

Evaluate the classification accuracy.

```
accuracy = mean(YTest==TTest)
accuracy = 0.9910
```

### Model Loss Function

The `modelLoss` function takes a `dlnetwork` object `net` and a mini-batch of input data `X` with corresponding labels `T`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `net`. To compute the gradients automatically, use the `dlgradient` function.

```
function [loss,gradients] = modelLoss(net,X,T)
Y = forward(net,X);
loss = crossentropy(Y,T);
gradients = dlgradient(loss,net.Learnables);
end
```

### Input Arguments

#### **net** — Network

`dlnetwork` object

Network, specified as a `dlnetwork` object.

The function updates the `Learnables` property of the `dlnetwork` object. `net.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 `net.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`.

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

Input	Learnable Parameters	Velocities
net	Table net.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 net.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 `sgdupdate` 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

### netUpdated — Updated network

dlnetwork object

Updated network, returned as a `dlnetwork` object.

The function updates the `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 1-1782 on the `trainingOptions` reference page.

## **Version History**

**Introduced in R2019b**

## **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”

## 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`.

## Version History

Introduced in R2018a

### See Also

`read` | `readByIndex` | `readall`

# 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** — Mini-batch queue

`minibatchqueue` object

Mini-batch queue, specified as a `minibatchqueue` object.

## Version History

**Introduced in R2020b**

## 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”

# 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 =
```

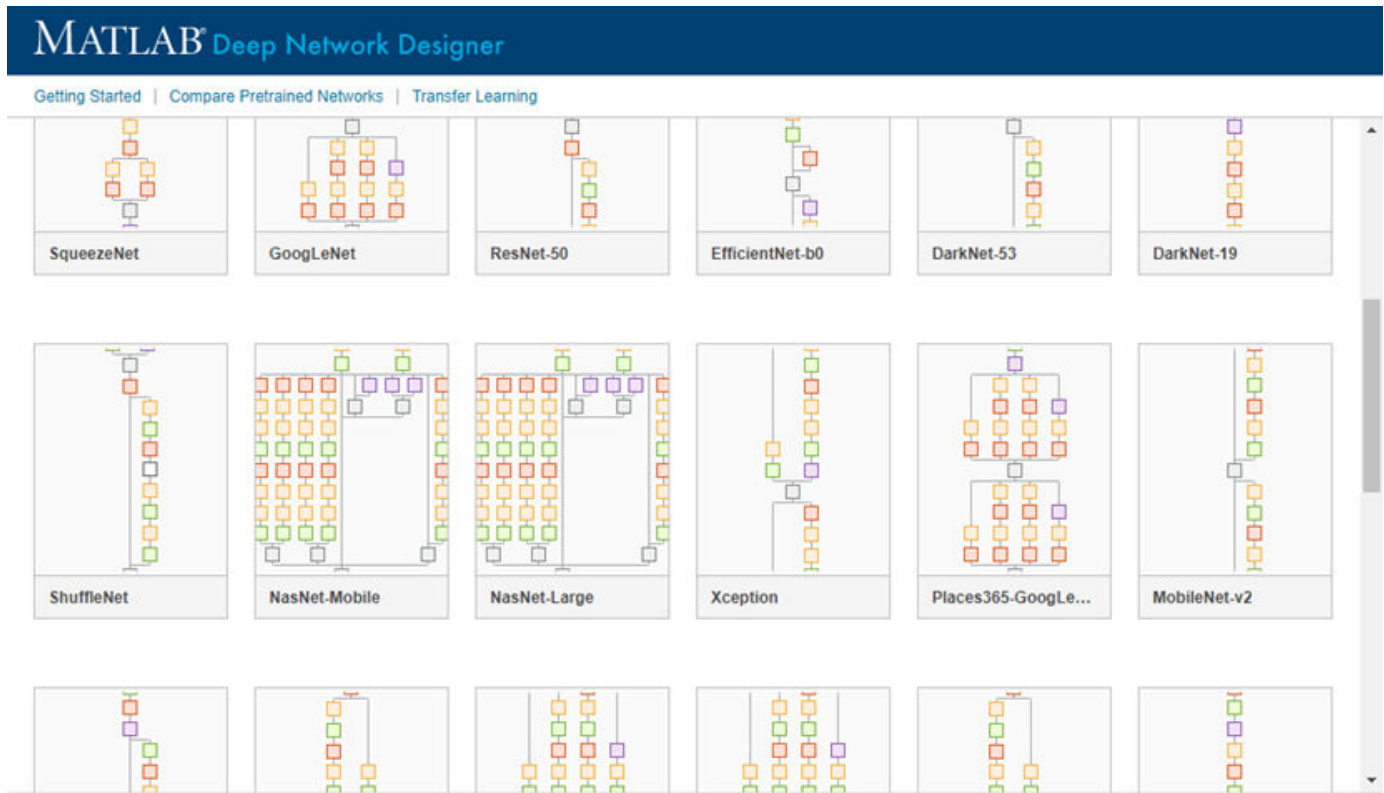
```
DAGNetwork with properties:
```

```
    Layers: [173x1 nnet.cnn.layer.Layer]
 Connections: [188x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(shufflenet)
```

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



If you need to download a neural network, pause on the desired neural 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.

## Version History

Introduced in R2019a

## 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"

## sigmoid

Apply sigmoid activation

### Syntax

```
Y = sigmoid(X)
```

### 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`
- 

`Y = sigmoid(X)` computes the sigmoid activation of the input `X` by applying the sigmoid transfer function. All values in `Y` 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);  
X = dlarray(X, 'SSCB');
```

Compute the sigmoid activation.

```
Y = sigmoid(X);
```

All values in  $Y$  now range between 0 and 1.

## Input Arguments

### **X** — Input data

`dlarray`

Input data, specified as a formatted `dlarray`, an unformatted `dlarray`, or a numeric array.

Data Types: `single` | `double`

## Output Arguments

### **Y** — Sigmoid activations

`dlarray`

Sigmoid activations, returned as a `dlarray`. All values in  $Y$  are between 0 and 1. The output  $Y$  has the same underlying data type as the input  $X$ .

If the input data  $X$  is a formatted `dlarray`,  $Y$  has the same dimension format as  $X$ . If the input data is not a formatted `dlarray`,  $Y$  is an unformatted `dlarray` with the same dimension order as the input data.

## Version History

Introduced in R2019b

## 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  $X$  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 darray Support”

# 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 the name `' '`.

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
```

## Algorithms

**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

$$P(C_k|x_n) = \frac{1}{1 + e^{-a_k}}.$$

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.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

`SigmoidLayer` objects apply an element-wise operation and supports input data of any format. The layer does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

**Introduced in R2020b**

### 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 Neural Network for Classification”

“Train Convolutional Neural Network for Regression”

“Deep Learning in MATLAB”

“Specify Layers of Convolutional Neural Network”

“List of Deep Learning Layers”



# softmax

Apply softmax activation to channel dimension

## Syntax

```
Y = softmax(X)
Y = softmax(X, '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 `darray` data. If you want to apply softmax within a `layerGraph` object or `Layer` array, use the following layer:

- `softmaxLayer`
- 

`Y = softmax(X)` computes the softmax activation of the input `X` by applying the softmax transfer function to the channel dimension of the input data. All values in `Y` are between 0 and 1, and sum to 1. The input `X` must be a formatted `darray`. The output `Y` is a formatted `darray` with the same dimension format as `X`.

`Y = softmax(X, 'DataFormat', FMT)` also specifies dimension format `FMT` when `X` is not a formatted `darray`. The output `Y` is an unformatted `darray` with the same dimension order as `X`.

## 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);
X = darray(X, 'CB');
```

Compute the softmax activation.

```
Y = softmax(X)
totalProb = sum(Y,1)

Y =
    10(C) x 2(B) darray

    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) darray

    1.0000    1.0000
```

All values in Y range between 0 and 1. The values over all channels sum to 1 for each observation.

## Input Arguments

### X — Input data

darray

Input data, specified as a formatted darray or an unformatted darray. When X is not a formatted darray, you must specify the dimension label format using 'DataFormat', FMT.

X 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 darray.

Example: 'DataFormat', 'SSCB'

Data Types: `char` | `string`

## Output Arguments

### Y — Softmax activations

`darray`

Softmax activations, returned as a `darray`. All values in Y are between 0 and 1. The output Y has the same underlying data type as the input X.

If the input data X is a formatted `darray`, Y has the same dimension format as X. If the input data is not a formatted `darray`, Y is an unformatted `darray` 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 1-1709 on the `softmaxLayer` reference page.

## Version History

Introduced in R2019b

## 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 X is a `darray` 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

`darray` | `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 darray Support”

# 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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	ReLU	ReLU
4	''	2-D 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

## Algorithms

### 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:

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

where  $0 \leq y_r \leq 1$  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:

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

where  $0 \leq P(c_r|x, \theta) \leq 1$  and  $\sum_{j=1}^k P(c_j|x, \theta) = 1$ . 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].

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `SoftmaxLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"CB" (channel, batch)	"CB" (channel, batch)
"SCB" (spatial, channel, batch)	"SCB" (spatial, channel, batch)
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"CBT" (channel, batch, time)	"CBT" (channel, batch, time)
"SCBT" (spatial, channel, batch, time)	"SCBT" (spatial, channel, batch, time)
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"CU" (channel, unspecified)	"CU" (channel, unspecified)
"SC" (spatial, channel)	"SC" (spatial, channel)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `SoftmaxLayer` objects also support these input and output format combinations.

Input Format	Output Format
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2016a

## 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 Neural Network for Classification”](#)

[“Train Convolutional Neural Network for Regression”](#)

[“Deep Learning in MATLAB”](#)

[“Specify Layers of Convolutional Neural Network”](#)

[“List of Deep Learning Layers”](#)

## 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);
```

True Class	airplane	923	4	21	8	4	1	5	5	23	6
	automobile	5	972	2					1	5	15
	bird	26	2	892	30	13	8	17	5	4	3
	cat	12	4	32	826	24	48	30	12	5	7
	deer	5	1	28	24	898	13	14	14	2	1
	dog	7	2	28	111	18	801	13	17		3
	frog	5		16	27	3	4	943	1	1	
	horse	9	1	14	13	22	17	3	915	2	4
	ship	37	10	4	4		1	2	1	931	10
	truck	20	39	3	3			2	1	9	923
		airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
		Predicted Class									

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"])
```

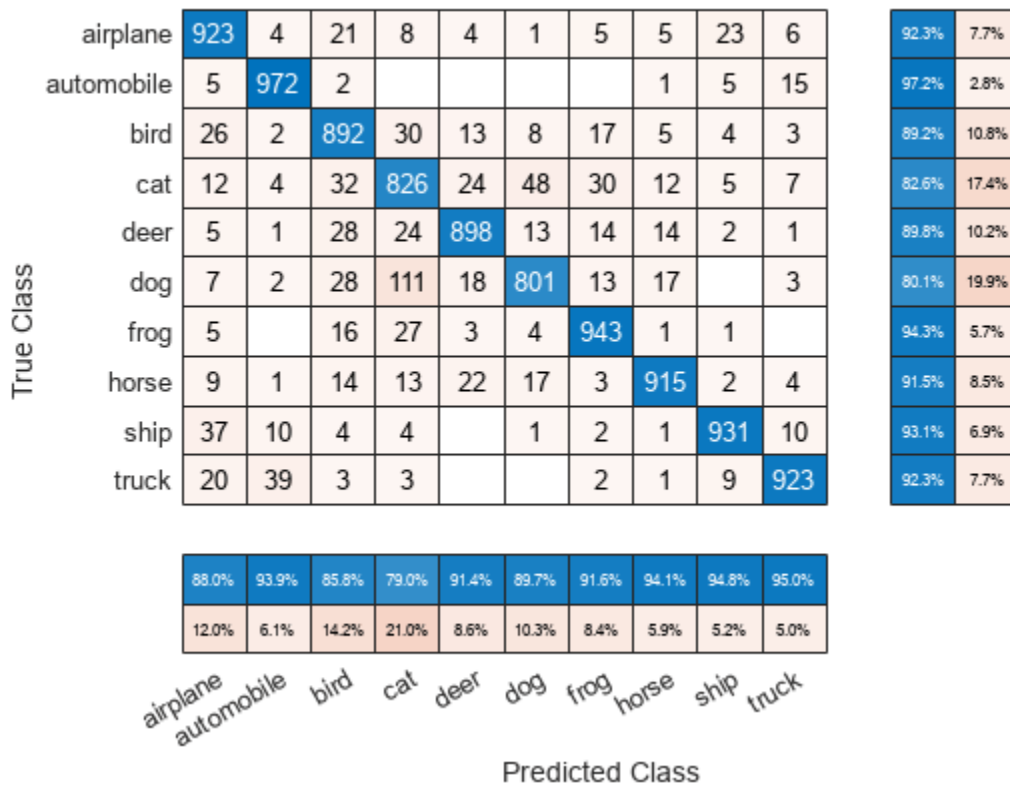
cat	826	48	12	24	32	30	12	5	4	7
dog	111	801	17	18	28	13	7		2	3
horse	13	17	915	22	14	3	9	2	1	4
deer	24	13	14	898	28	14	5	2	1	1
bird	30	8	5	13	892	17	26	4	2	3
frog	27	4	1	3	16	943	5	1		
airplane	8	1	5	4	21	5	923	23	4	6
ship	4	1	1		4	2	37	931	10	10
automobile			1		2		5	5	972	15
truck	3		1		3	2	20	9	39	923

Predicted Class

### 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');
```



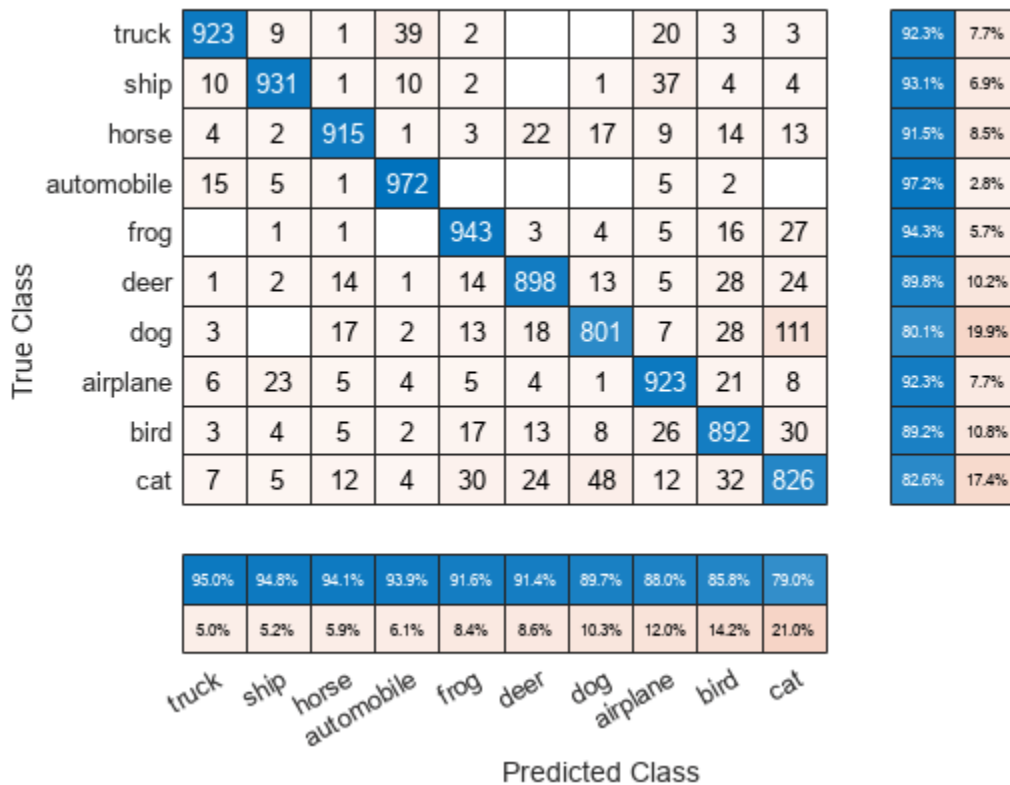
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. 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';
```

True Class	automobile	972		5	5	15	1		2			97.2%	2.8%								
	frog		943	1	5		1	3	16	27	4	94.3%	5.7%								
	ship	10	2	931	37	10	1		4	4	1	93.1%	6.9%								
	airplane	4	5	23	923	6	5	4	21	8	1	92.3%	7.7%								
	truck	39	2	9	20	923	1		3	3		92.3%	7.7%								
	horse	1	3	2	9	4	915	22	14	13	17	91.5%	8.5%								
	deer	1	14	2	5	1	14	898	28	24	13	89.8%	10.2%								
	bird	2	17	4	26	3	5	13	892	30	8	89.2%	10.8%								
	cat	4	30	5	12	7	12	24	32	826	48	82.6%	17.4%								
	dog	2	13		7	3	17	18	28	111	801	80.1%	19.9%								
												93.9%	91.6%	94.8%	88.0%	95.0%	94.1%	91.4%	85.8%	79.0%	89.7%
												6.1%	8.4%	5.2%	12.0%	5.0%	5.9%	8.6%	14.2%	21.0%	10.3%
												automobile	frog	ship	airplane	truck	horse	deer	bird	cat	dog
												Predicted Class									

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"])`

## Version History

**Introduced in R2018b**

### See Also

#### Functions

`categorical` | `confusionchart`

#### Properties

`ConfusionMatrixChart` Properties

#### Topics

“Deep Learning in MATLAB”

## 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 1-1734.

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 1-1719.

`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: [68x1 nnet.cnn.layer.Layer]
 Connections: [75x2 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*).

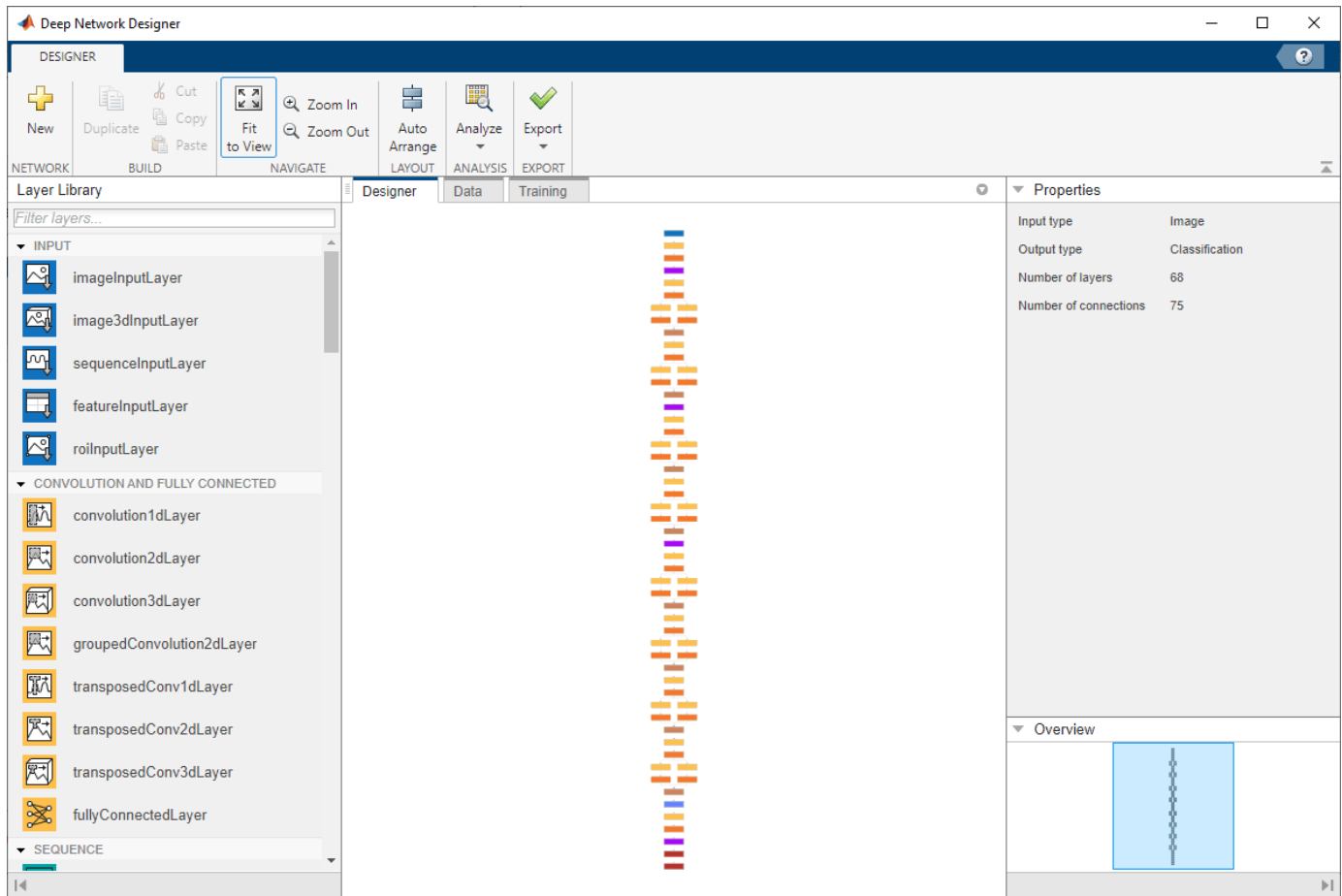
```
unzip("MerchData.zip");
```

#### 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.

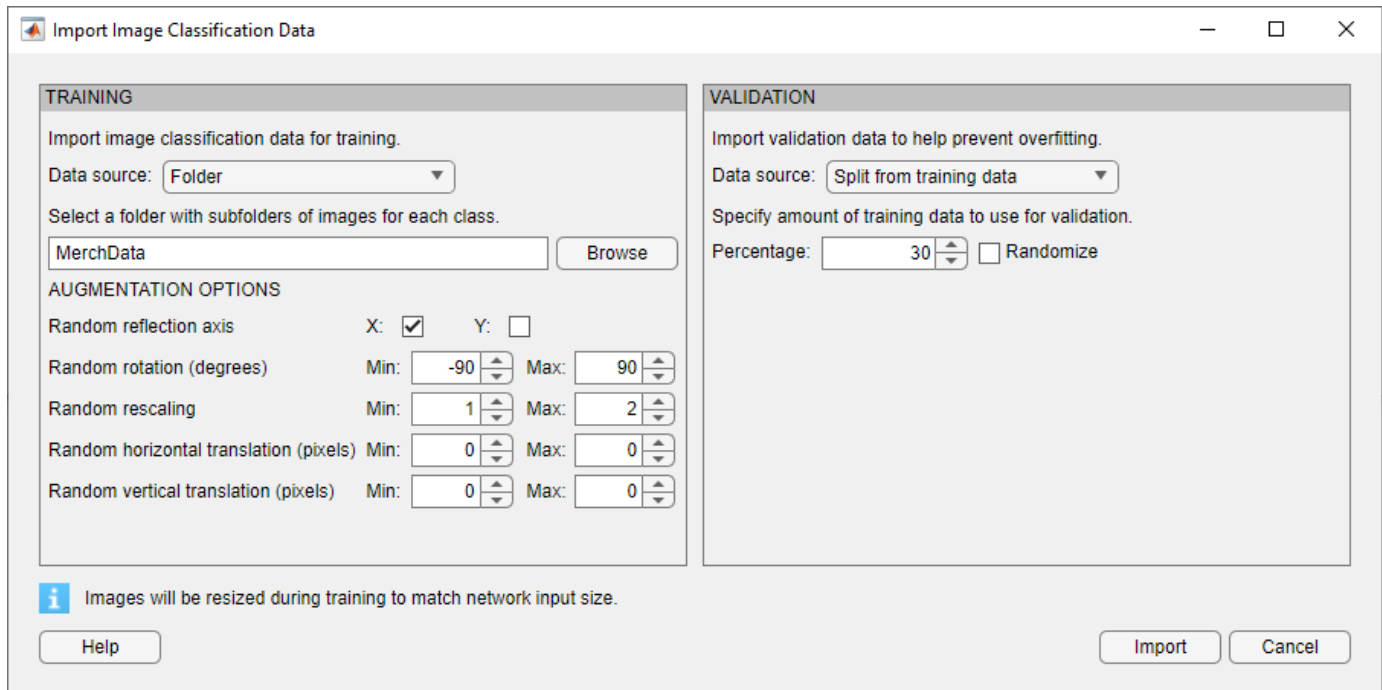
## Import Data

To load the data into Deep Network Designer, on the **Data** tab, click **Import Data > Import Image Classification Data**.

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.

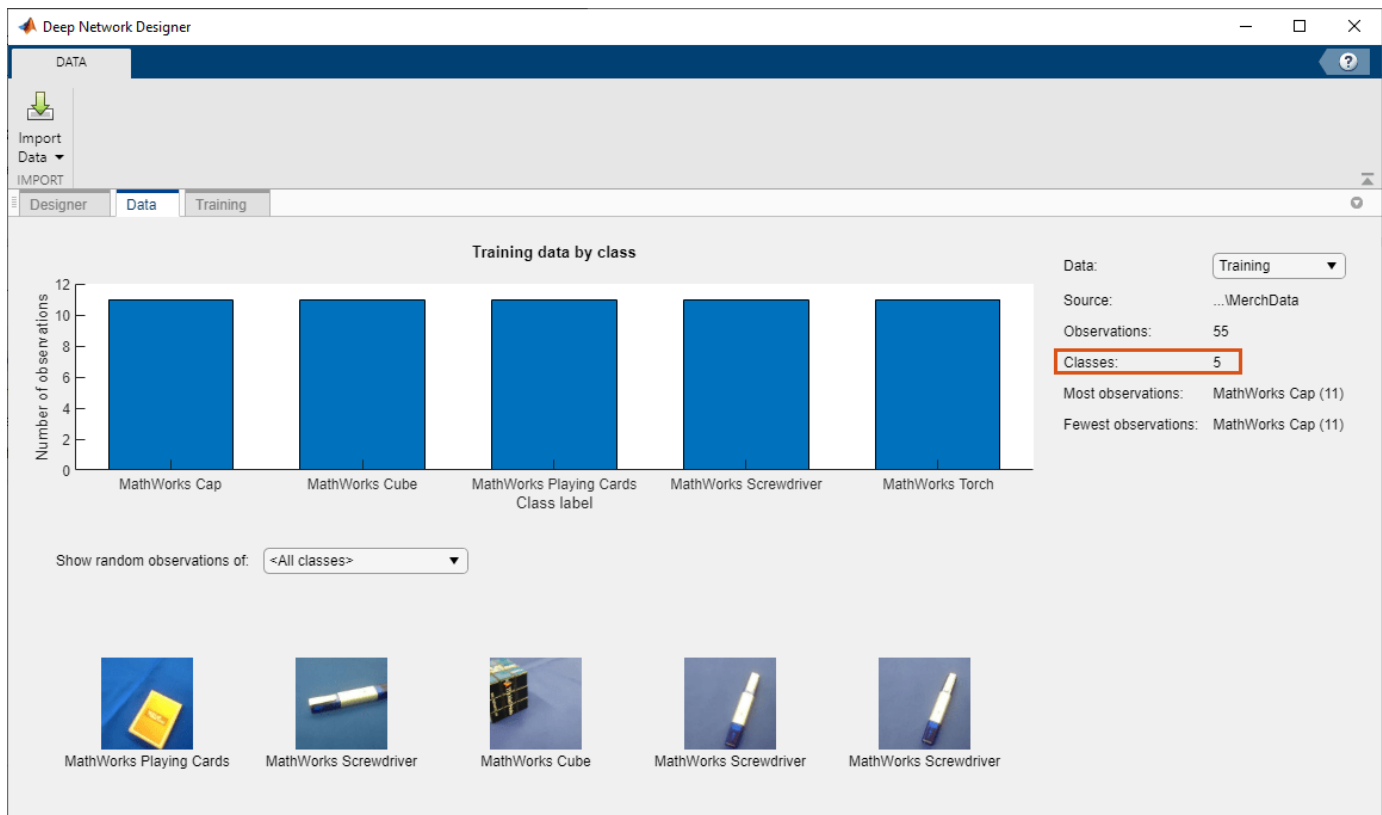
Specify augmentation operations to perform on the training images. For this example, apply a random reflection in the x-axis, a random rotation from the range  $[-90,90]$  degrees, and a random rescaling from the range  $[1,2]$ . Data augmentation helps prevent the network from overfitting and memorizing 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.



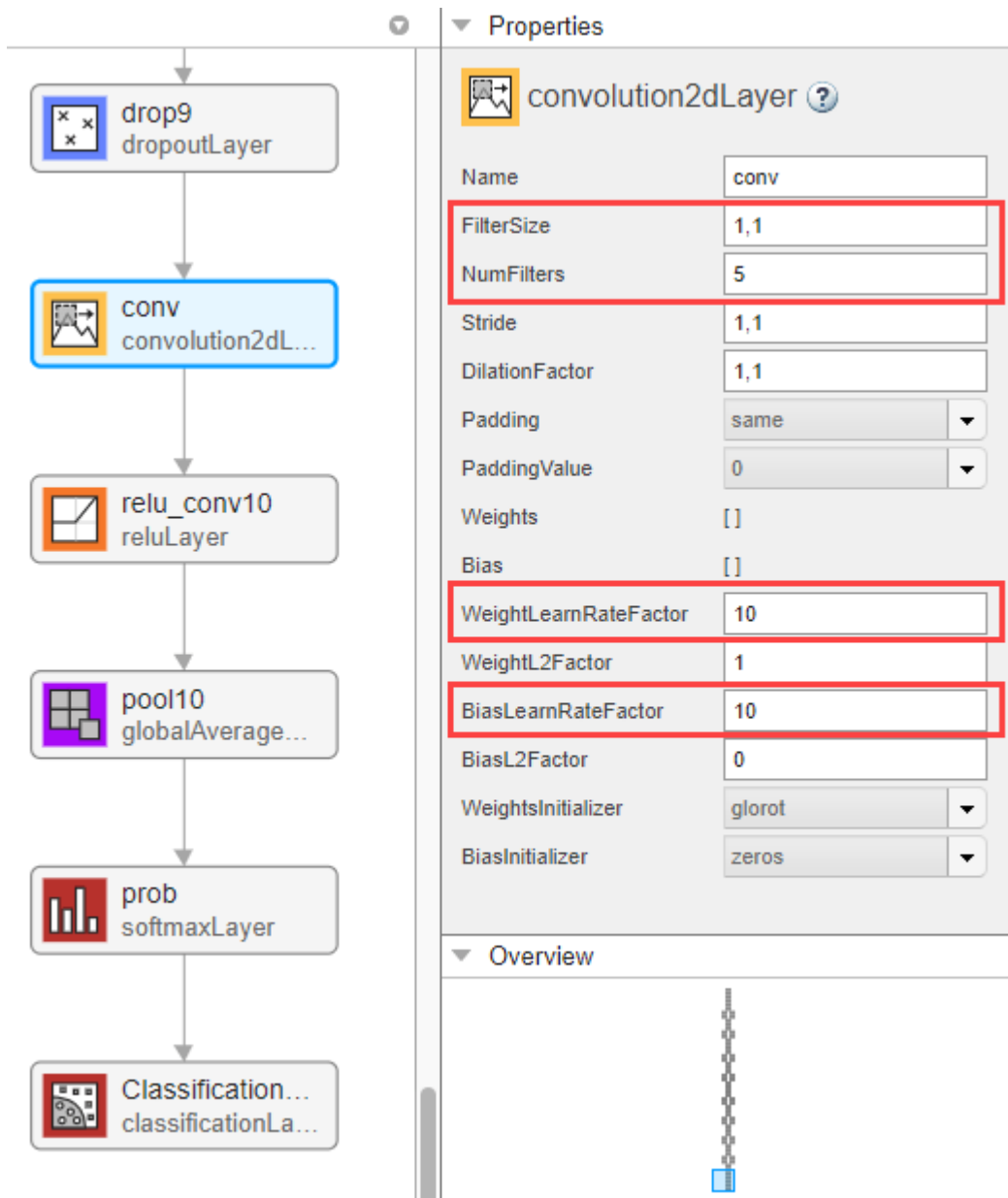
### Edit Network for Transfer Learning

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.

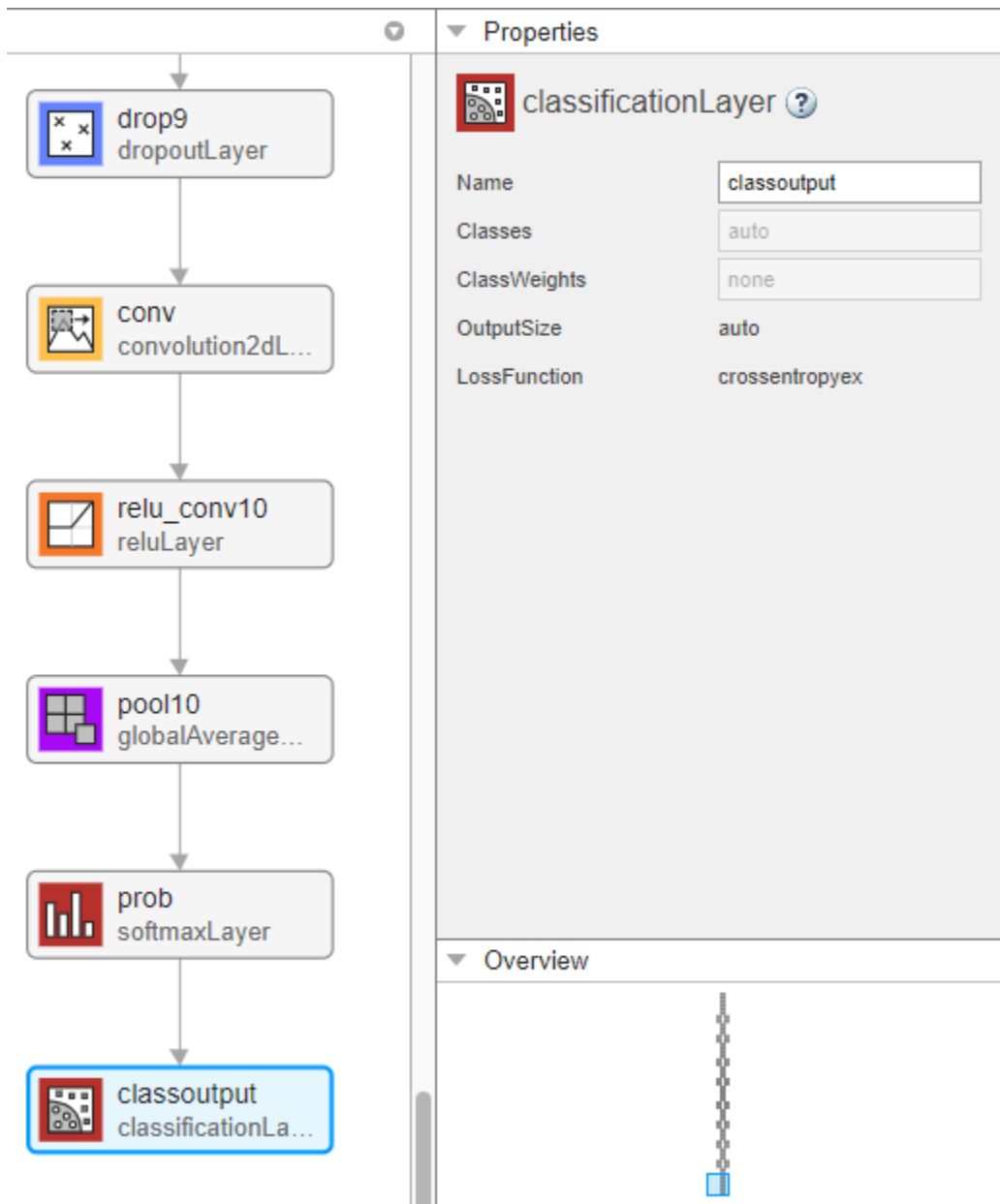
In most networks, the last layer with learnable weights is a fully connected layer. In some networks, such as SqueezeNet, the last learnable layer is the final convolutional layer instead. In this case, replace the convolutional layer with a new convolutional layer with the number of filters equal to 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.

Deep Learning Network Analyzer

Network from Deep Network Designer  
Analysis date: 13-Jul-2020 09:47:03

68 layers   0 warnings   0 errors

**ANALYSIS RESULT**

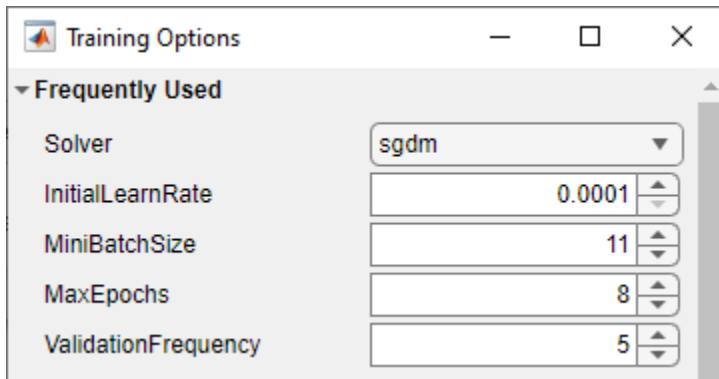
	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 64 3×3×3 convolutions with stride [2 2] and padding [0 0 0 0]	Convolution	113×113×64	Weights 3×3×3×64 Bias 1×1×64
3	relu_conv1 ReLU	ReLU	113×113×64	-
4	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	56×56×64	-
5	fire2-squeeze1x1 16 1×1×64 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 1×1×16 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 3×3×16 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 1×1×128 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 64 1×1×16 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	56×56×64	Weights 1×1×16×64 Bias 1×1×64
15	fire3-relu_expand1x1 ReLU	ReLU	56×56×64	-

## Train Network

Specify training options. Select the **Training** tab and click **Training Options**.

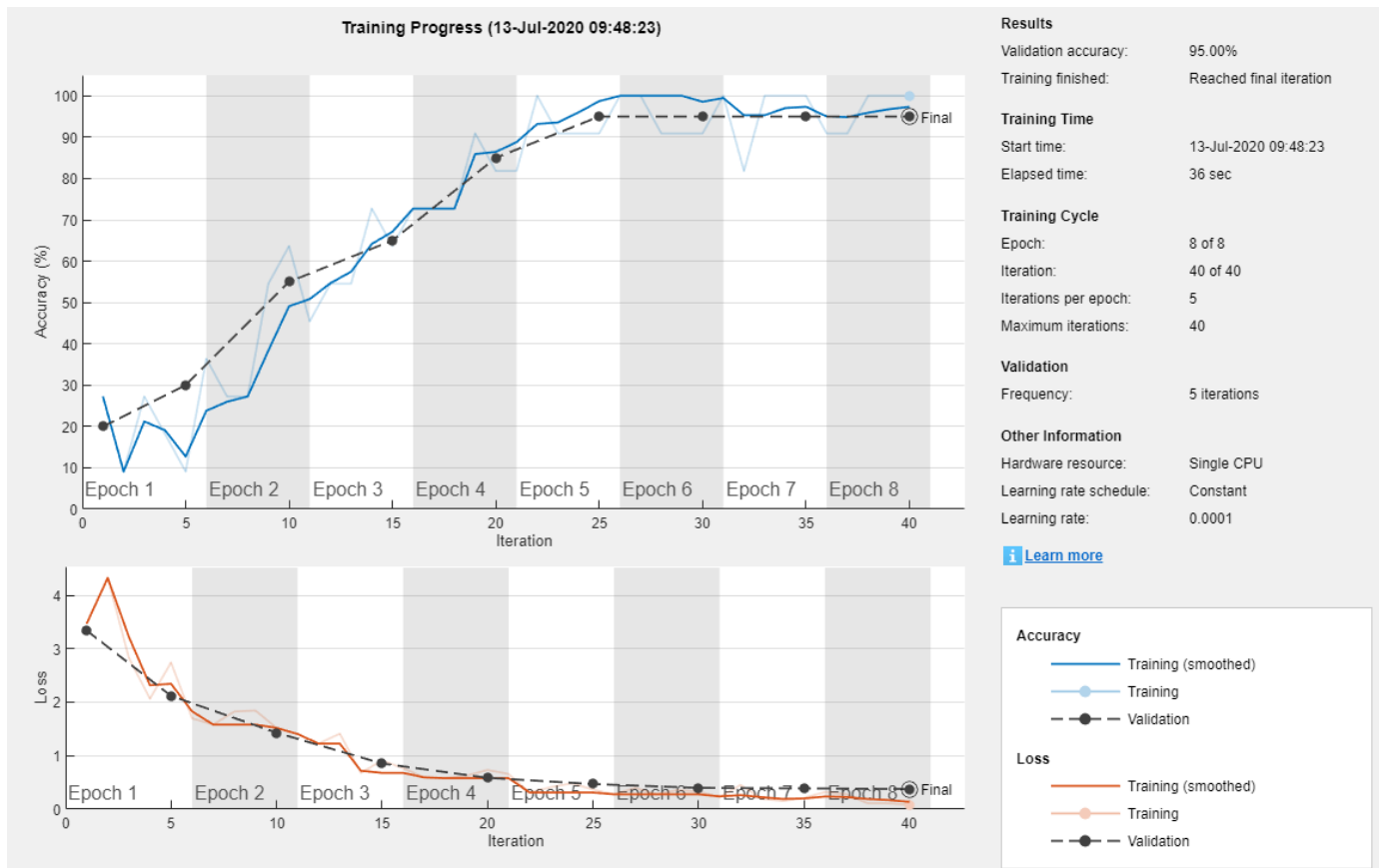
- Set the initial learn rate to a small value to slow down learning in the transferred layers.
- Specify the validation frequency so that the accuracy on the validation data is calculated once every epoch.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data 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, **MaxEpochs** to 8, and **ValidationFrequency** to 5. As there are 55 observations, set **MiniBatchSize** to 11.



To train the network with the specified training options, click **OK** 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 NaN
    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.

The screenshot shows the Deep Network Designer interface. On the left, a network diagram displays two layers: 'data imageInputLayer' and 'conv1 convolution2dL...'. On the right, the 'Properties' pane for the selected 'imageInputLayer' is shown. The 'InputSize' property is highlighted with a red box and is set to '227,227,3'. Other properties include 'Name' (data), 'Normalization' (zerocenter), 'NormalizationDimension' (auto), 'Mean' ([1×1×3 single]), and 'StandardDeviation' ([]).

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) + "%");

```

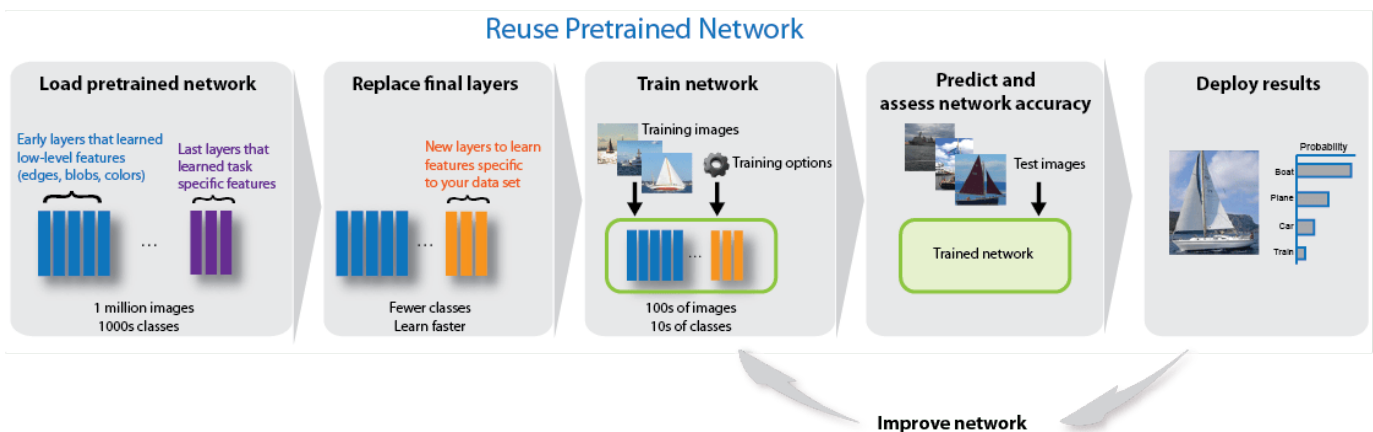


### Programmatic Transfer Learning Using SqueezeNet

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.

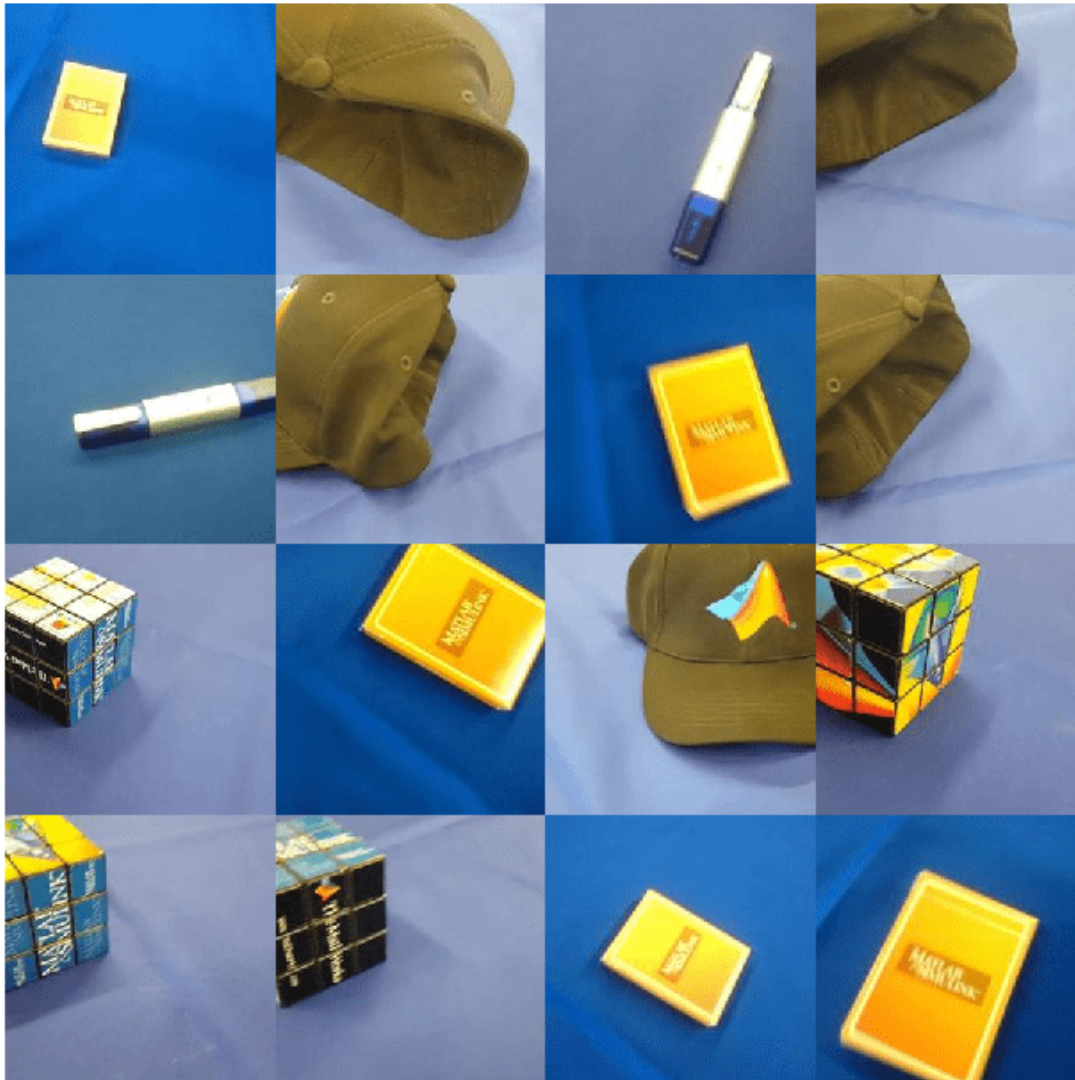
```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

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)
```



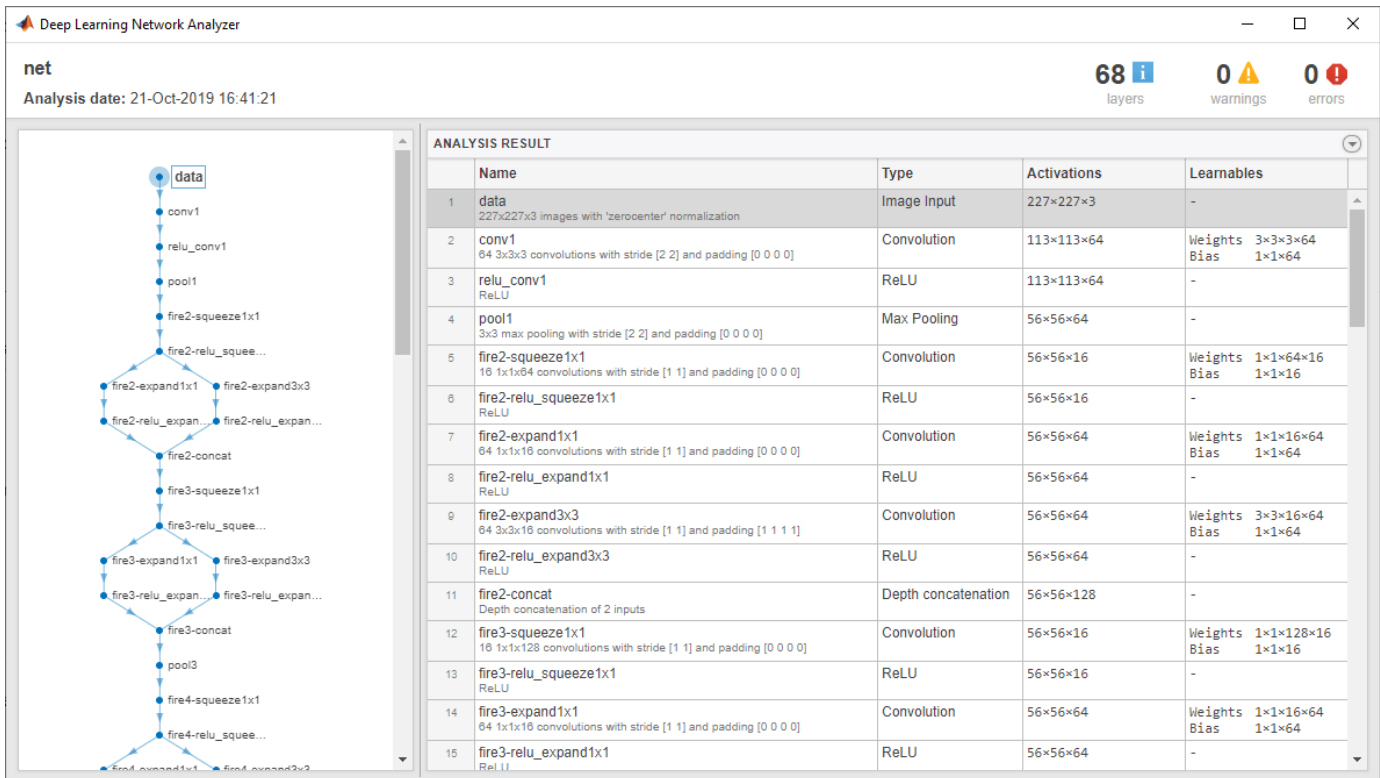
## Load Pretrained Network

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 = 1x3
```

```
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);
```

```
ans =  
1x2 Layer array with layers:
```

```

1 'conv10'                2-D Convolution          1000 1x1x512 convolutions w
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,'BiasLearnRateFactor',10);
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.

```

newClassificationLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassificationLayer);

```

### Train Network

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

```

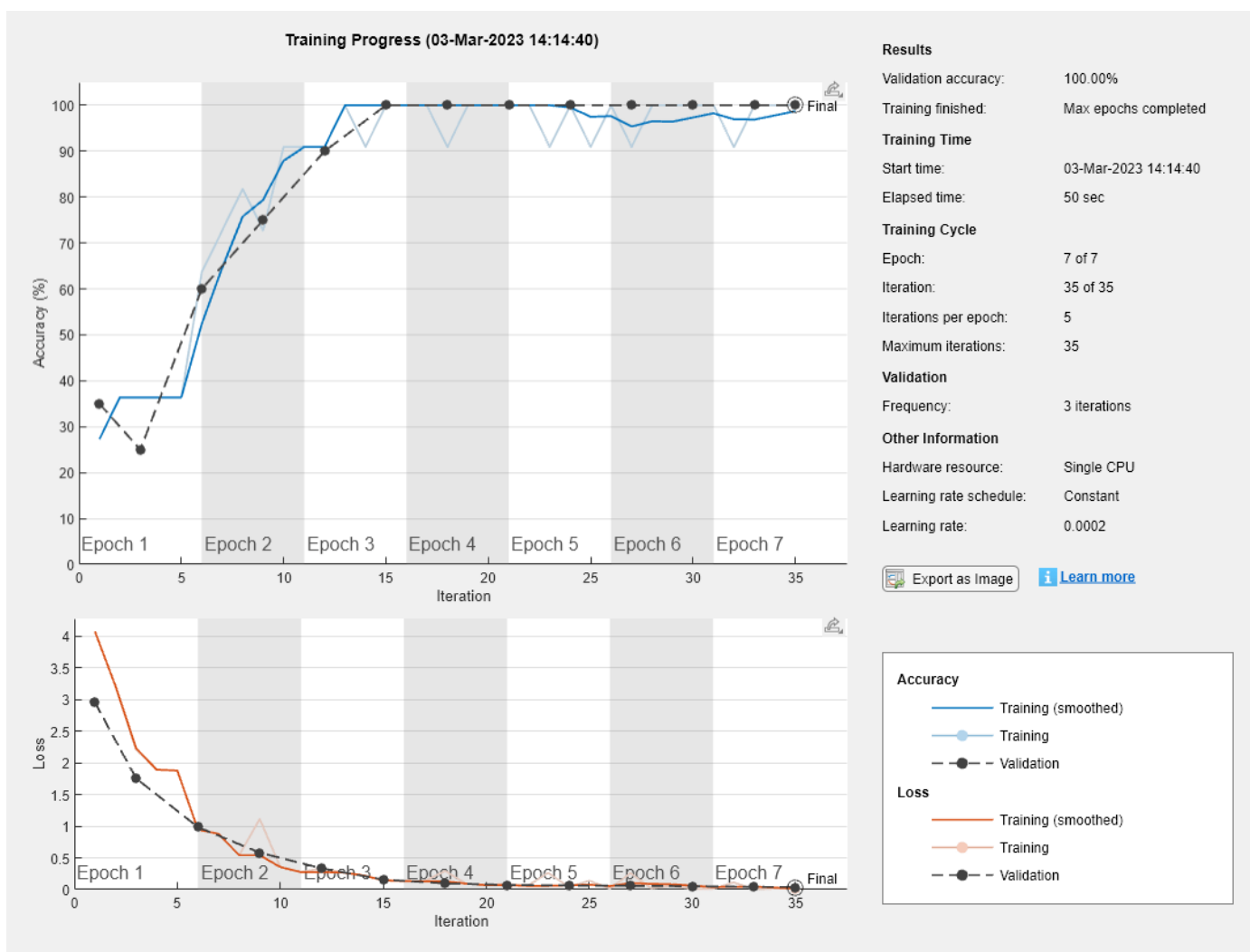
```

'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 Computing Requirements” (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
```

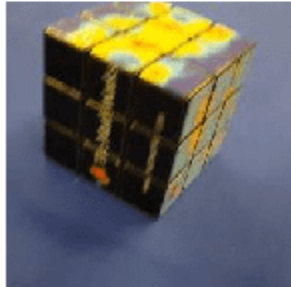
**MathWorks Playing Cards**



**MathWorks Playing Cards**



**MathWorks Cube**



**MathWorks Cube**



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)
```

bell pepper



## 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 requires only 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)
```



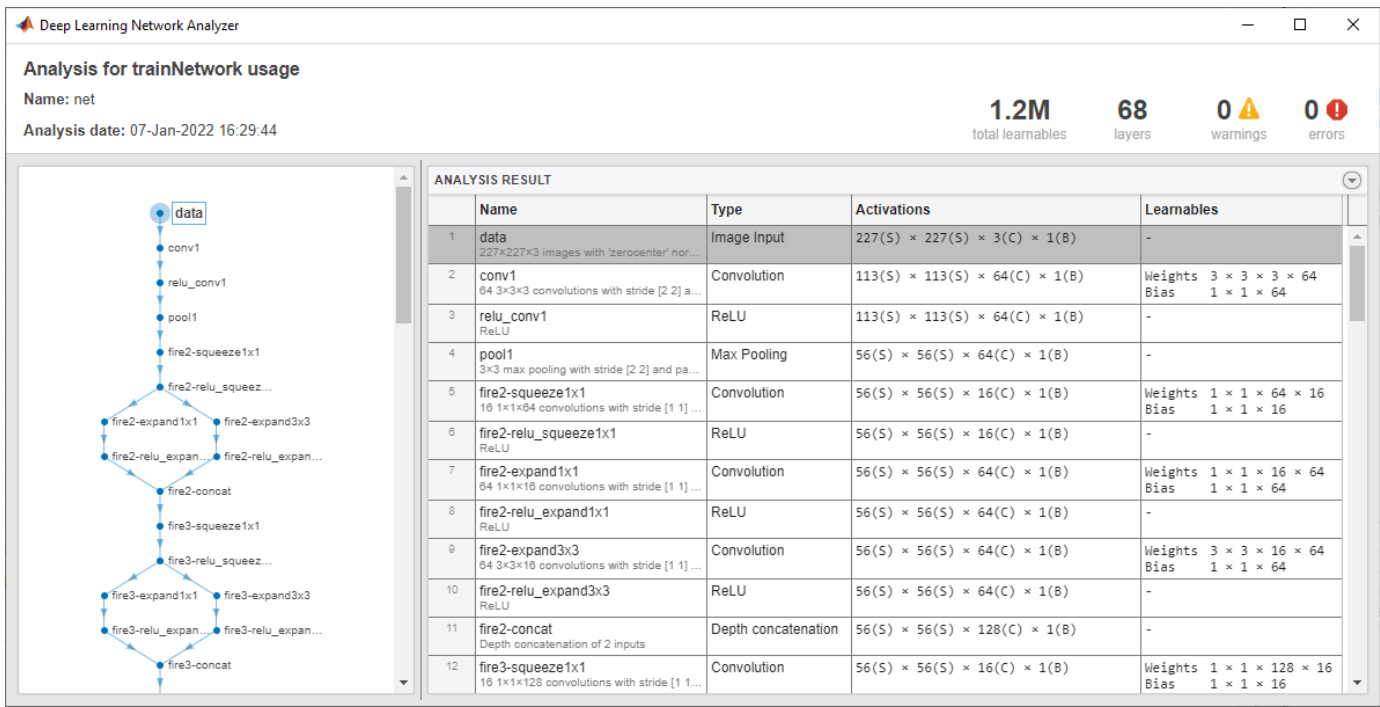
### Load Pretrained Network

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 = 1x3
```

```
    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 inputting them 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.

```
TTrain = imdsTrain.Labels;  
TTest = 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,TTrain);
```

### **Classify Test Images**

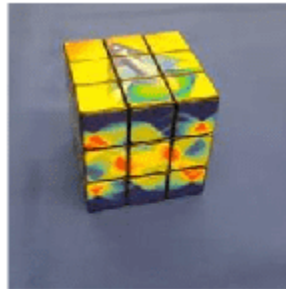
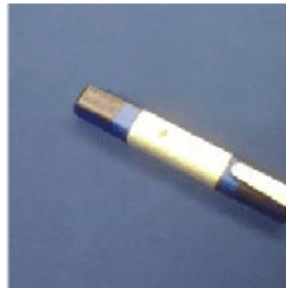
Classify the test images using the trained SVM model and the features extracted from the test images.

```
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
```



**MathWorks Cap****MathWorks Cube****MathWorks Playing Cards****MathWorks Screwdriver**

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

```
accuracy = mean(YPred == TTest)
```

```
accuracy = 0.9500
```

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.

## Version History

Introduced in R2018a

## 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"



# stripdims

Remove darray data format

## Syntax

```
Y = stripdims(X)
```

## Description

`Y = stripdims(X)` returns the darray `X` without any dimension labels. `Y` is an unformatted darray.

## Examples

### Remove Data Format from darray

Create a formatted darray.

```
d1X = darray(randn(3,2,1,2), 'SSTU')
```

```
d1X =
    3(S) x 2(S) x 1(T) x 2(U) darray
```

```
(:,:,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 `d1X` but has no dimension labels.

```
y = stripdims(d1X)
```

```
y =
    3x2x1x2 darray
```

```
(:,:,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

### **X** — Input data

`dLarray` object

Input data, specified as a formatted or unformatted `dLarray` object.

## Output Arguments

### **Y** — Unformatted `dLarray`

unformatted `dLarray` object

Unformatted `dLarray`, returned as an unformatted `dLarray` object that is the same as the input array `X`, but without any dimension labels. If `X` is unformatted, then `Y = X`.

## 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(X)` can decrease after a `stripdims` call because the function removes trailing singleton dimensions.

```
X = dLarray(ones(3,2), 'SCB');  
ndims(X)
```

```
ans =
```

```
3
```

```
X = stripdims(X);  
ndims(X)
```

```
ans =
```

```
2
```

## Version History

Introduced in R2019b

## 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](#)

## summary

Print network summary

### Syntax

```
summary(net)
```

### Description

`summary(net)` prints a summary of the `dlnetwork` object `net`. The summary shows whether the network is initialized, the total number of learnable parameters, and information about the network inputs.

Network input summaries require or input layers or an initialized network. Network learnable parameter summaries require an initialized network.

### Examples

#### Print Network Summary

Load the `dlnetwork` object `net` from the `dlnetDigits` MAT file.

```
load dlnetDigits
```

Print the network summary using the `summary` function.

```
summary(net)

    Initialized: true

    Number of learnables: 84k

    Inputs:
      1 'in' 28x28x1 images
```

### Input Arguments

#### **net** — Neural network

`dlnetwork` object

Neural network, specified as a `dlnetwork` object.

### Version History

Introduced in R2022b

### See Also

`dlnetwork` | `initialize` | `plot`

**Topics**

“Define Custom Training Loops, Loss Functions, and Networks”

“Train Generative Adversarial Network (GAN)”

“Automatic Differentiation Background”

# 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 the name ' '.

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	''	2-D Convolution	20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3	''	Batch Normalization	Batch normalization
4	''	Swish	Swish
5	''	2-D 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

## Algorithms

### 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".

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

`SwishLayer` objects apply an element-wise operation and supports input data of any format. The layer and does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

**Introduced in R2021a**

### 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 Neural Network for Classification”](#)

[“Train Convolutional Neural Network for Regression”](#)

[“Deep Learning in MATLAB”](#)

[“Compare Activation Layers”](#)

[“List of Deep Learning Layers”](#)

# **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 the name `''`.

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  ''  2-D Convolution      16 3x3 convolutions with stride [1 1] and padding [0 0 0 0]  
  3  ''  Batch Normalization  Batch normalization  
  4  ''  Tanh                 Hyperbolic tangent  
  5  ''  2-D Max Pooling     2x2 max pooling with stride [2 2] and padding [0 0 0 0]  
  6  ''  2-D Convolution      32 3x3 convolutions with stride [1 1] and padding [0 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
```

## Algorithms

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

`TanhLayer` objects apply an element-wise operation and supports input data of any format. The layer does not add or remove any dimensions, so it outputs data with the same format as its inputs.

## Version History

Introduced in R2019a

### 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 Neural 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”](#)

# taylorPrunableNetwork

Network that can be pruned by using first-order Taylor approximation

## Description

A `TaylorPrunableNetwork` object enables support for pruning of filters in convolution layers by using first-order Taylor approximation. To prune filters in a `dlnetwork` object, first convert it to a `TaylorPrunableNetwork` object and then use the associated object functions.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see Deep Learning Toolbox Model Quantization Library.

## Creation

### Syntax

```
prunableNet = taylorPrunableNetwork(net)
prunableNet = taylorPrunableNetwork(layers)
```

### Description

`prunableNet = taylorPrunableNetwork(net)` converts a `dlnetwork` object `net` to a `TaylorPrunableNetwork` object. The latter is a different representation of the same network that is suitable for pruning by using the Taylor pruning algorithm. If the input network cannot be pruned, this function produces an error.

`prunableNet = taylorPrunableNetwork(layers)` converts the network layers specified in `layers` to a `TaylorPrunableNetwork` object that is suitable for pruning by using the Taylor pruning algorithm. The input `layers` must be a `LayerGraph` object or a `Layer` array that can be converted to a `dlnetwork` object.

### Input Arguments

#### **net** — Neural network

`dlnetwork` object

Neural network, specified as a `dlnetwork` object.

#### **layers** — Network layers

`LayerGraph` object | `Layer` array

Network layers, specified as a `layerGraph` object or as a `Layer` array.

## Properties

#### **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 `darray` 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 `darray` 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.

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

### **NumPrunables** — Number of convolution layer filters that are suitable for pruning

nonnegative integer

Number of convolution layer filters in the network that are suitable for pruning by using first-order Taylor approximation, specified as a nonnegative integer.

## **Object Functions**

forward	Compute deep learning network output for training
predict	Compute deep learning network output for inference
updatePrunables	Remove filters from prunable layers based on importance scores
updateScore	Compute and accumulate Taylor-based importance scores for pruning
dlnetwork	Deep learning network for custom training loops

## **Examples**

### **Prune dlnetwork Object to Compress the Model**

This example shows how to prune a dlnetwork object by using a custom pruning loop.

#### **Load dlnetwork Object**

Load a trained dlnetwork object and the corresponding classes.

```
s = load("digitsCustom.mat");  
dlnet_1 = s.dlnet;  
classes = s.classes;
```

Inspect the layers of the dlnetwork object. The network has three convolution layers at locations 2, 5, and 8 of the Layer array.

```
layers_1 = dlnet_1.Layers
```

```
layers_1 =
```

```
12x1 Layer array with layers:
```

1	'input'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv1'	2-D Convolution	20 5x5x1 convolutions with stride [1 1] and padding
3	'bn1'	Batch Normalization	Batch normalization with 20 channels
4	'relu1'	ReLU	ReLU
5	'conv2'	2-D Convolution	20 3x3x20 convolutions with stride [1 1] and padding
6	'bn2'	Batch Normalization	Batch normalization with 20 channels
7	'relu2'	ReLU	ReLU
8	'conv3'	2-D Convolution	20 3x3x20 convolutions with stride [1 1] and padding
9	'bn3'	Batch Normalization	Batch normalization with 20 channels
10	'relu3'	ReLU	ReLU
11	'fc'	Fully Connected	10 fully connected layer
12	'softmax'	Softmax	softmax

#### **Load Data for Prediction**

Load the digits data for prediction.

```
dataFolder = fullfile(toolboxdir("nnet"), "ndemos", "nndatasets", "DigitDataset");
```



```

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

```

Partition the data into pruning and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```

[imdsPrune,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 images, use augmented image datastores.

```

inputSize = [28 28 1];
augimdsPrune = augmentedImageDatastore(inputSize(1:2),imdsPrune);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);

```

### Prune dlnetwork Object

Convert the `dlnetwork` object to a representation that is suitable for pruning by using the `taylorPrunableNetwork` function. This function returns a `TaylorPrunableNetwork` object that has the `NumPrunables` property set to 48. This indicates that 48 filters in the original model are suitable for pruning by using the Taylor pruning algorithm.

```

prunableNet_1 = taylorPrunableNetwork(dlnet_1)

```

```

prunableNet_1 =
    TaylorPrunableNetwork with properties:

        Learnables: [14x3 table]
           State: [6x3 table]
    InputNames: {'input'}
    OutputNames: {'softmax'}
    NumPrunables: 48

```

Create a `minibatchqueue` object that processes and manages mini-batches of images during pruning. 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 format 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 Computing Requirements" (Parallel Computing Toolbox).

```

miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(augimdsPrune, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    MiniBatchFormat=["SSCB" ""]);

```

Calculate Taylor-based importance scores of the prunable filters in the network by looping over the mini-batches of data. For each mini-batch:

- Calculate pruning activations and pruning gradients by using the `modelLoss` function defined at the end of this example
- Update importance scores of the prunable filters by using the `updateScore` function

```
while hasdata(mbq)
    [X,T] = next(mbq);
    [~,pruningActivations,pruningGradients] = dlfeval(@modelLoss,prunableNet_1,X,T);
    prunableNet_1 = updateScore(prunableNet_1,pruningActivations,pruningGradients);
end
```

Finally, remove filters with the lowest importance scores to create a new `TaylorPrunableNetwork` object by using the `updatePrunables` function. By default, a single call to this function removes 8 filters. Observe that the new network `prunableNet_2` has 40 prunable filters remaining.

```
prunableNet_2 = updatePrunables(prunableNet_1)
```

```
prunableNet_2 =
    TaylorPrunableNetwork with properties:
```

```
    Learnables: [14x3 table]
        State: [6x3 table]
    InputNames: {'input'}
    OutputNames: {'softmax'}
    NumPrunables: 40
```

To further compress the model, run the custom pruning loop and update prunables again.

### Extract Pruned `dlnetwork` Object

Use the `dlnetwork` function to extract the pruned `dlnetwork` object from the pruned `TaylorPrunableNetwork` object. You can now use this compressed `dlnetwork` object to perform inference.

```
dlnet_2 = dlnetwork(prunableNet_2);
```

Compare the convolution layers of the original and the pruned `dlnetwork` objects. Observe that the three convolution layers in the pruned network have fewer filters. These counts agree with the fact that, by default, a single call to the `updatePrunables` function removes 8 filters from the network.

```
conv_layers_1 = dlnet_1.Layers([2 5 8])
```

```
conv_layers_1 =
    3x1 Convolution2DLayer array with layers:
```

```
    1 'conv1' 2-D Convolution 20 5x5x1 convolutions with stride [1 1] and padding [0 0]
    2 'conv2' 2-D Convolution 20 3x3x20 convolutions with stride [1 1] and padding [1 1]
    3 'conv3' 2-D Convolution 20 3x3x20 convolutions with stride [1 1] and padding [1 1]
```

```
conv_layers_2 = dlnet_2.Layers([2 5 8])
```

```
conv_layers_2 =
    3x1 Convolution2DLayer array with layers:
```

```
    1 'conv1' 2-D Convolution 17 5x5x1 convolutions with stride [1 1] and padding [0 0]
```

```

2 'conv2' 2-D Convolution 18 3x3x17 convolutions with stride [1 1] and padding [1 1]
3 'conv3' 2-D Convolution 17 3x3x18 convolutions with stride [1 1] and padding [1 1]

```

## Supporting Functions

### Model Loss Function

The `modelLoss` function takes a `TaylorPrunableNetwork` object `net`, a mini-batch of input data `X` with corresponding targets `T` and returns activations in `net` and the gradients of the loss with respect to the activations in `net`. To compute the gradients automatically, this function uses the `dlgradient` function.

```

function [loss, pruningActivations, pruningGradients] = modelLoss(net,X,T)

% Calculate network output for training.
[out, ~, pruningActivations] = forward(net,X);

% Calculate loss.
loss = crossentropy(out,T);

% Compute pruning gradients.
pruningGradients = dlgradient(loss,pruningActivations);
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,T] = preprocessMiniBatch(dataX,dataT)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(dataX);

% Extract label data from cell and concatenate.
T = cat(2,dataT{1:end});

% One-hot encode labels.
T = onehotencode(T,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 concatenating 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(dataX)

```

```
% Concatenate.  
X = cat(4,dataX{1:end});  
  
% Normalize the images.  
X = X/255;  
  
end
```

## More About

### Layers Supported for Pruning

The Taylor pruning algorithm prunes filters from `convolution2dLayer` objects. Pruning convolutional filters can also reduce the number of learnable parameters in downstream layers, for example:

- `batchNormalizationLayer`
- `fullyConnectedLayer`
- `groupedConvolution2dLayer`
- `transposedConv2dLayer`

### Effect of Network Architecture on Pruning

For certain network architectures, data dependency between the prunable layers and other layers in the network might prevent pruning of filters. These are some example architectures that exhibit this behavior:

- Your network has a `convolution2dLayer`, a `groupNormalizationLayer` and another `convolution2dLayer` connected in sequence. The presence of the group normalization layer prevents pruning of filters of the first convolution layer, because doing so changes the shape of the input channels of the group normalization layer.
- Your network has a 2-D convolution layer, a softmax layer, and an output layer connected in sequence. This architecture prevents pruning of filters of the convolution layer because doing so changes the output size of the network.

## Algorithms

For an individual input data point in the pruning dataset, you use the `forward` function to calculate the output of the deep learning network and the activations of the prunable filters. Then you calculate the gradients of the loss with respect to these activations using automatic differentiation. You then pass the network, the activations, and the gradients to the `updateScore` function. For each prunable filter in the network, the `updateScore` function calculates the change in loss that occurs if that filter is pruned from the network (up to first-order Taylor approximation). Based on this change, the function associates an importance score with that filter and updates the `TaylorPrunableNetwork` object [1].

Inside the custom pruning loop, you accumulate importance scores for the prunable filters over all mini-batches of the pruning dataset. Then you pass the network object to the `updatePrunables` function. This function prunes the filters that have the lowest importance scores and hence have the smallest effect on the accuracy of the network output. The number of filters that a single call to the `updatePrunables` function prunes is determined by the optional name-value argument `MaxToPrune`, that has a default value of 8.

All these steps complete a single pruning iteration. To further compress your model, repeat these steps multiple times over a loop.

## Version History

Introduced in R2022a

## References

- [1] Molchanov, Pavlo, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. "Pruning Convolutional Neural Networks for Resource Efficient Inference." Preprint, submitted June 8, 2017. <https://arxiv.org/abs/1611.06440>.

## See Also

`predict` | `forward` | `updatePrunables` | `updateScore` | `dlnetwork`

## Topics

"Prune Filters in a Detection Network Using Taylor Scores"

## 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 neural 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 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: ''
CheckpointFrequency: 1
CheckpointFrequencyUnit: 'epoch'
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

This example shows how to monitor the training process of deep learning networks.

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.

This example shows how to monitor training progress for networks trained using the `trainNetwork` function. For networks trained using a custom training loop, use a `trainingProgressMonitor` object to plot metrics during training. For more information, see “Monitor Custom Training Loop Progress”.

When you set the `Plots` training option to “training-progress” 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.

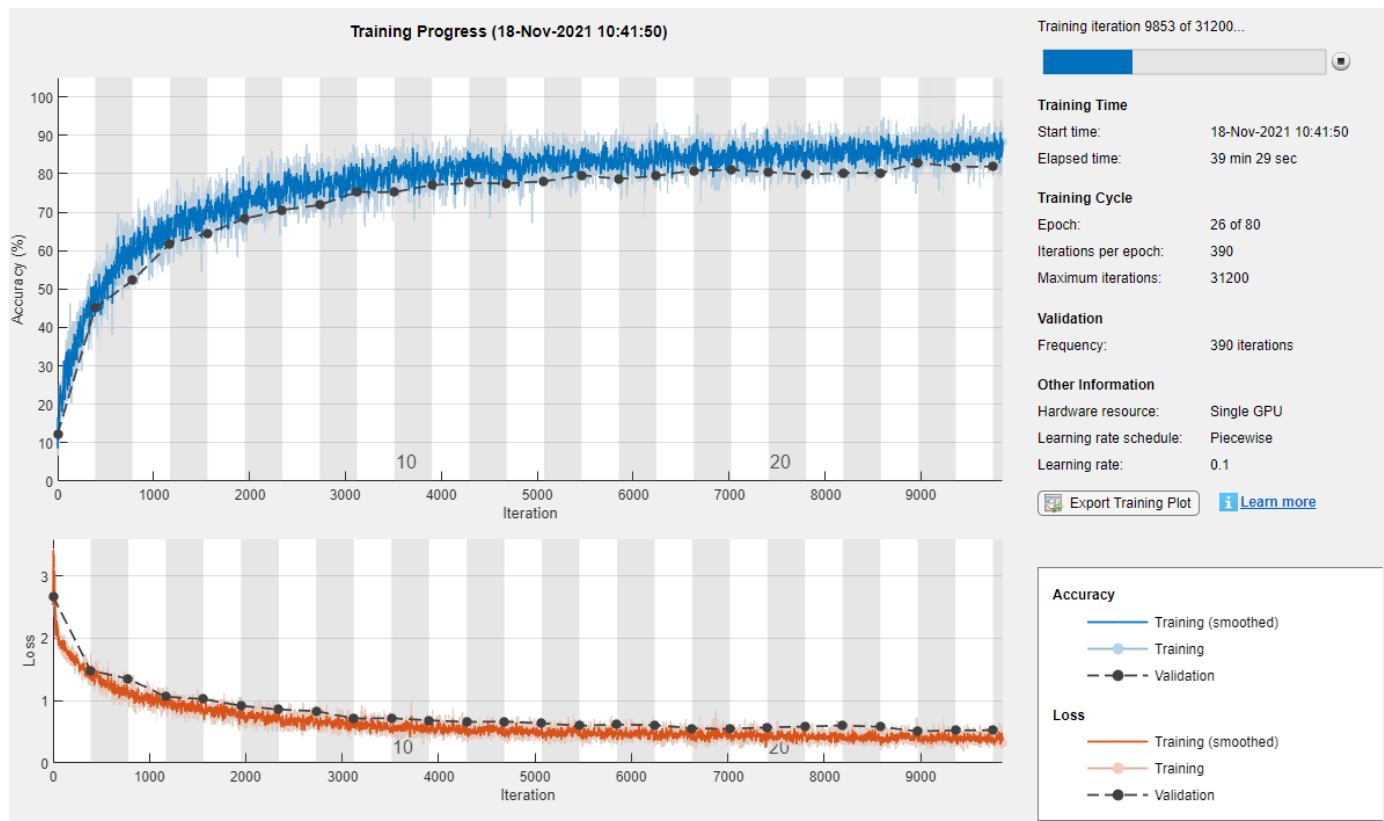
When training finishes, view the **Results** showing the finalized validation accuracy and the reason that training finished. If the `OutputNetwork` training option is "last-iteration" (default), the finalized metrics correspond to the last training iteration. If the `OutputNetwork` training option is "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 `BatchNormalizationStatistics` 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 `BatchNormalizationStatistics` 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".

To save the training progress plot, click **Export Training Plot** in the training window. You can save the plot as a PNG, JPEG, TIFF, or PDF file. You can also save the individual plots of loss, accuracy, and root mean squared error using the axes toolbar.





## 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")
```

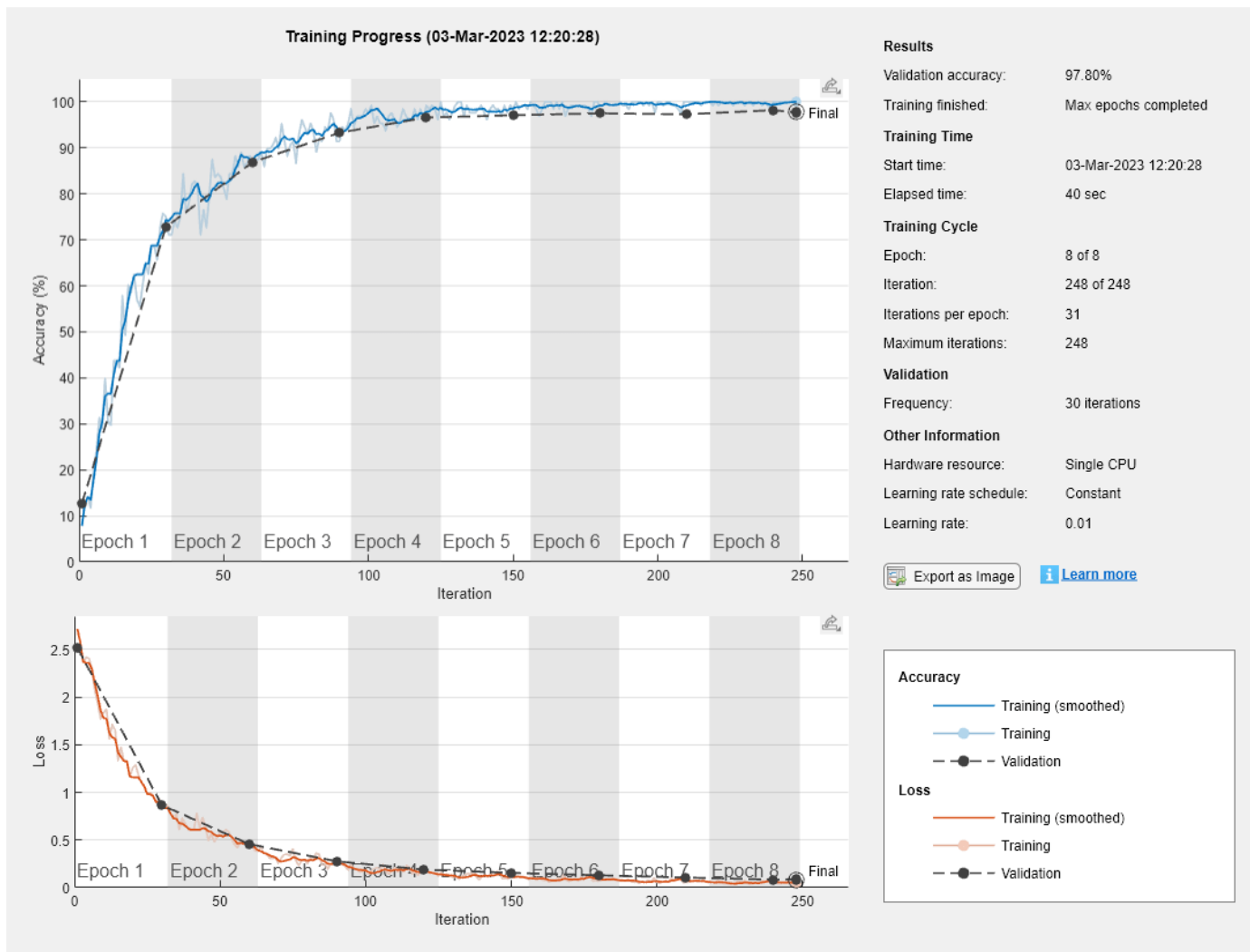
```
batchNormalizationLayer  
reluLayer  
fullyConnectedLayer(10)  
softmaxLayer  
classificationLayer];
```

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, set the `Plots` training option to "training-progress".

```
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 neural network

`'sgdm' | 'rmsprop' | 'adam'`

Solver for training neural 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` training option.
- 'rmsprop' — Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the `SquaredGradientDecayFactor` training option.
- 'adam' — Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the `GradientDecayFactor` and `SquaredGradientDecayFactor` training options, respectively.

For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-1782.

## 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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 neural network training

'none' (default) | 'training-progress'

Plots to display during neural 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 neural network. You can save the training plot as an image or PDF by clicking **Export Training Plot**. For more information on the training progress plot, see “Monitor Deep Learning 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 Neural Networks**

<b>Field</b>	<b>Description</b>
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 <code>ClassificationOutputLayer</code> 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 <code>ClassificationOutputLayer</code> 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 Neural 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 <code>RegressionOutputLayer</code> object, then the loss is the half-mean-squared-error.
Validation Loss	Loss on the validation data. If the output layer is a <code>RegressionOutputLayer</code> 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: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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 a positive integer. This option only has an effect when the `Verbose` training option is 1 (true).

If you validate the neural network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Mini-Batch Options****MaxEpochs — Maximum number of epochs**

30 (default) | 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **MiniBatchSize — Size of mini-batch**

128 (default) | 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.

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Shuffle — Option for data shuffling**

'once' (default) | '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 neural 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` training option to 'every-epoch'.

## **Validation**

### **ValidationData — Data to use for validation during training**

[] (default) | 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 neural 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 neural network validation.

If `ValidationData` is `[]`, then the software does not validate the neural network during training.

**ValidationFrequency — Frequency of neural network validation**

50 (default) | positive integer

Frequency of neural network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics. To specify validation data, use the `ValidationData` training option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ValidationPatience — Patience of validation stopping**

Inf (default) | positive integer

Patience of validation stopping of neural 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 neural network training stops. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned neural network depends on the `OutputNetwork` training option. To return the neural network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**OutputNetwork — Neural network to return when training completes**

'last-iteration' (default) | 'best-validation-loss'

Neural network to return when training completes, specified as one of the following:

- `'last-iteration'` - Return the neural network corresponding to the last training iteration.
- `'best-validation-loss'` - Return the neural network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify the `ValidationData` training option.

**Solver Options****InitialLearnRate — Initial learning rate**

positive scalar

Initial learning rate used for training, specified as 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 can take a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LearnRateSchedule — Option for dropping learning rate during training**

'none' (default) | 'piecewise'

Option for dropping the learning rate during training, specified as 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` training option to specify the value of this factor. Use the `LearnRateDropPeriod` training option to specify the number of epochs between multiplications.

### **LearnRateDropPeriod — Number of epochs for dropping the learning rate**

10 (default) | positive integer

Number of epochs for dropping the learning rate, specified as a positive integer. This option is valid only when the `LearnRateSchedule` training option 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` training option.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **LearnRateDropFactor — Factor for dropping the learning rate**

0.1 (default) | scalar from 0 to 1

Factor for dropping the learning rate, specified as a scalar from 0 to 1. This option is valid only when the `LearnRateSchedule` training option 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` training option.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **L2Regularization — Factor for L<sub>2</sub> regularization**

0.0001 (default) | nonnegative scalar

Factor for L<sub>2</sub> regularization (weight decay), specified as a nonnegative scalar. For more information, see “L2 Regularization” on page 1-1785.

You can specify a multiplier for the L<sub>2</sub> regularization for neural network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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 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. The default value works well for most tasks.

To specify the `Momentum` training option, `solverName` must be 'sgdm'.

For more information, see “Stochastic Gradient Descent with Momentum” on page 1-1783.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**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 a nonnegative scalar less than 1. The gradient decay rate is denoted by  $\beta_1$  in the “Adam” on page 1-1784 section.

To specify the GradientDecayFactor training option, solverName must be 'adam'.

The default value works well for most tasks.

For more information, see “Adam” on page 1-1784.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SquaredGradientDecayFactor — Decay rate of squared gradient moving average**

nonnegative scalar less than 1

Decay rate of squared gradient moving average for the Adam and RMSProp solvers, specified as a nonnegative scalar less than 1. The squared gradient decay rate is denoted by  $\beta_2$  in [4].

To specify the SquaredGradientDecayFactor training option, solverName must 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, see “Adam” on page 1-1784 and “RMSProp” on page 1-1783.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Epsilon — Denominator offset**

1e-8 (default) | positive scalar

Denominator offset for Adam and RMSProp solvers, specified as a positive scalar.

The solver adds the offset to the denominator in the neural network parameter updates to avoid division by zero. The default value works well for most tasks.

To specify the Epsilon training option, solverName must be 'adam' or 'rmsprop'.

For more information, see “Adam” on page 1-1784 and “RMSProp” on page 1-1783.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ResetInputNormalization — Option to reset input layer normalization**

1 (true) (default) | 0 (false)

Option to reset input layer normalization, specified as one of the following:

- 1 (true) — Reset the input layer normalization statistics and recalculate them at training time.
- 0 (false) — Calculate normalization statistics at training time when they are empty.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### 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  $\mu^*$  and  $\sigma^{2*}$  denote the updated mean and variance, respectively,  $\lambda_\mu$  and  $\lambda_{\sigma^2}$  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  $\mu$  and  $\sigma^2$  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 Inf or a positive scalar. If the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to the GradientThresholdMethod training option.

For more information, see Gradient Clipping on page 1-1784.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 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 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 1-1784.

## Sequence Options

### SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### 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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is 'sequence' for each recurrent 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

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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Hardware Options

### ExecutionEnvironment — Hardware resource for training neural network

`'auto'` (default) | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training neural 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 Computing Requirements” (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.

The `'multi-gpu'` and `'parallel'` options do not support neural networks containing custom layers with state parameters or built-in layers that are stateful at training time. For example:

- recurrent layers such as `LSTMlayer`, `BiLSTMlayer`, or `GRUlayer` objects when the `SequenceLength` training option is a positive integer
- `BatchNormalizationLayer` objects when the `BatchNormalizationStatistics` training option is set to `'moving'`

### WorkerLoad — Parallel worker load division

scalar from 0 to 1 | positive integer | numeric vector

Parallel worker load division between GPUs or CPUs, specified as one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for neural network training computation. If you train the neural 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 neural network training computation. If you train the neural 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 — Neural network training load for each worker in the parallel pool. For a vector  $W$ , worker  $i$  gets a fraction  $W(i)/\text{sum}(W)$  of the work (number of examples per mini-batch). If you

train a neural 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DispatchInBackground — Flag to enable background dispatch**

0 (false) (default) | 1 (true)

Flag to enable background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as 0 (false) or 1 (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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Checkpoints**

#### **CheckpointPath — Path for saving checkpoint neural networks**

"" (default) | character vector

Path for saving the checkpoint neural networks, specified as a character vector or string scalar.

- If you do not specify a path (that is, you use the default ""), then the software does not save any checkpoint neural networks.
- If you specify a path, then `trainNetwork` saves checkpoint neural networks to this path and assigns a unique name to each neural network. You can then load any checkpoint neural network and resume training from that neural network.

If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint neural networks. If the path you specify does not exist, then `trainingOptions` returns an error.

The `CheckpointFrequency` and `CheckpointFrequencyUnit` options specify the frequency of saving checkpoint neural networks.

For more information about saving neural network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Data Types: `char` | `string`

#### **CheckpointFrequency — Frequency of saving checkpoint neural networks**

1 (default) | positive integer

Frequency of saving checkpoint neural networks, specified as a positive integer.

If `CheckpointFrequencyUnit` is 'epoch', then the software saves checkpoint neural networks every `CheckpointFrequency` epochs.

If `CheckpointFrequencyUnit` is `'iteration'`, then the software saves checkpoint neural networks every `CheckpointFrequency` iterations.

This option only has an effect when `CheckpointPath` is nonempty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CheckpointFrequencyUnit — Checkpoint frequency unit**

`'epoch'` (default) | `'iteration'`

Checkpoint frequency unit, specified as `'epoch'` or `'iteration'`.

If `CheckpointFrequencyUnit` is `'epoch'`, then the software saves checkpoint neural networks every `CheckpointFrequency` epochs.

If `CheckpointFrequencyUnit` is `'iteration'`, then the software saves checkpoint neural networks every `CheckpointFrequency` iterations.

This option only has an effect when `CheckpointPath` is nonempty.

### **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 neural networks)
TrainingRMSE	RMSE on the current mini-batch (regression neural networks)
ValidationAccuracy	Accuracy on the validation data (classification neural networks)
ValidationRMSE	RMSE on the validation data (regression neural networks)
State	Current training state, with a possible value of <code>"start"</code> , <code>"iteration"</code> , or <code>"done"</code> .

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 `1` (true). If any output function returns `1` (true), then

training finishes and `trainNetwork` returns the latest neural 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` is '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 neural 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 neural 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 1-412 and “Specify Initial Weights and Biases in Fully Connected Layer” on page 1-766.

### 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,

$$\theta_{t+1} = \theta_t - \alpha \nabla E(\theta_t),$$



where  $\ell$  is the iteration number,  $\alpha > 0$  is the learning rate,  $\theta$  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` training options, 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  $\gamma$  determines the contribution of the previous gradient step to the current iteration. You can specify this value using the `Momentum` training option. To train a neural network using the stochastic gradient descent with momentum algorithm, specify 'sgdm' as the first input argument to `trainingOptions`. To specify the initial value of the learning rate  $\alpha$ , use the `InitialLearnRate` training option. 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,

$$v_{\ell} = \beta_2 v_{\ell-1} + (1 - \beta_2) [\nabla E(\theta_{\ell})]^2$$

$\beta_2$  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  $1/(1-\beta_2)$ , that is, 10, 100, and 1000 parameter updates, respectively. You can specify  $\beta_2$  by using the `SquaredGradientDecayFactor` training options. 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.  $\epsilon$  is a small constant added to avoid division by zero. You can specify  $\epsilon$  by using the `Epsilon` training option, but the default value usually works well. To use RMSProp to train a neural network, specify 'rmsprop' as the first input to `trainingOptions`.

## 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  $\beta_1$  and  $\beta_2$  decay rates using the `GradientDecayFactor` and `SquaredGradientDecayFactor` training options, respectively. Adam uses the moving averages to update the network parameters as

$$\theta_{\ell+1} = \theta_\ell - \frac{\alpha m_\ell}{\sqrt{v_\ell} + \epsilon}$$

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  $\epsilon$  by using the `Epsilon` training option. 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 'adam' as the first input to `trainingOptions`. The full Adam update also includes a mechanism to correct a bias that appears in the beginning of training. For more information, see [4].

Specify the learning rate  $\alpha$  for all optimization algorithms using the `InitialLearnRate` training option. 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.

## L<sub>2</sub> 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,  $\lambda$  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  $\lambda$  by using the `L2Regularization` training option. 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.

## Version History

### Introduced in R2016a

#### **R2022b: `trainNetwork` pads mini-batches to length of longest sequence before splitting when you specify `SequenceLength` training option as an integer**

Starting in R2022b, when you train a neural network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the neural network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

#### **R2018b: `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.

#### **R2018b: 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 Neural 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"

# 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 first input argument.


## Properties

### Plots and Display

#### Plots — Plots to display during neural network training

'none' (default) | 'training-progress'

Plots to display during neural 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 neural network. You can save the training plot as an image or PDF by clicking **Export Training Plot**. For more information on the training progress plot, see “Monitor Deep Learning 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 Neural Networks**

<b>Field</b>	<b>Description</b>
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 <code>ClassificationOutputLayer</code> 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 <code>ClassificationOutputLayer</code> 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 Neural 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 <code>RegressionOutputLayer</code> object, then the loss is the half-mean-squared-error.
Validation Loss	Loss on the validation data. If the output layer is a <code>RegressionOutputLayer</code> 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: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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 a positive integer. This option only has an effect when the `Verbose` training option is 1 (true).

If you validate the neural network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Mini-Batch Options

#### MaxEpochs — Maximum number of epochs

30 (default) | 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **MiniBatchSize — Size of mini-batch**

128 (default) | 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.

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Shuffle — Option for data shuffling**

'once' (default) | '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 neural 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` training option to 'every-epoch'.

## **Validation**

### **ValidationData — Data to use for validation during training**

[] (default) | 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 neural 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 neural network validation.



If `ValidationData` is `[]`, then the software does not validate the neural network during training.

### **ValidationFrequency — Frequency of neural network validation**

50 (default) | positive integer

Frequency of neural network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics. To specify validation data, use the `ValidationData` training option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **ValidationPatience — Patience of validation stopping**

Inf (default) | positive integer

Patience of validation stopping of neural 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 neural network training stops. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned neural network depends on the `OutputNetwork` training option. To return the neural network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputNetwork — Neural network to return when training completes**

'last-iteration' (default) | 'best-validation-loss'

Neural network to return when training completes, specified as one of the following:

- `'last-iteration'` - Return the neural network corresponding to the last training iteration.
- `'best-validation-loss'` - Return the neural network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify the `ValidationData` training option.

## **Solver Options**

### **InitialLearnRate — Initial learning rate**

0.001 (default) | positive scalar

Initial learning rate used for training, specified as a positive scalar.

If the learning rate is too low, then training can take a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **LearnRateScheduleSettings — Settings for learning rate schedule**

structure

This property is read-only.

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<sub>2</sub> regularization**

0.0001 (default) | nonnegative scalar

Factor for L<sub>2</sub> regularization (weight decay), specified as a nonnegative scalar. For more information, see “L2 Regularization” on page 1-1800.

You can specify a multiplier for the L<sub>2</sub> regularization for neural network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 a nonnegative scalar less than 1. The gradient decay rate is denoted by  $\beta_1$  in the “Adam” on page 1-1799 section.

The default value works well for most tasks.

For more information, see “Adam” on page 1-1799.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SquaredGradientDecayFactor — Decay rate of squared gradient moving average**

0.999 (default) | nonnegative scalar less than 1

Decay rate of squared gradient moving average for the Adam solver, specified as a nonnegative scalar less than 1. The squared gradient decay rate is denoted by  $\beta_2$  in the “Adam” on page 1-1799 section.

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.

For more information, see “Adam” on page 1-1799.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Epsilon — Denominator offset**

1e-8 (default) | positive scalar

Denominator offset for Adam solver, specified as a positive scalar.

The solver adds the offset to the denominator in the neural network parameter updates to avoid division by zero. The default value works well for most tasks.

For more information, see “Adam” on page 1-1799.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **ResetInputNormalization — Option to reset input layer normalization**

1 (true) (default) | 0 (false)

Option to reset input layer normalization, specified as one of the following:

- 1 (true) — Reset the input layer normalization statistics and recalculate them at training time.
- 0 (false) — Calculate normalization statistics at training time when they are empty.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **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  $\mu^*$  and  $\sigma^{2*}$  denote the updated mean and variance, respectively,  $\lambda_\mu$  and  $\lambda_{\sigma^2}$  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  $\mu$  and  $\sigma^2$  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 Inf or a positive scalar. If the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to the GradientThresholdMethod training option.

For more information, see Gradient Clipping on page 1-1800.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**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 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 1-1800.

**Sequence Options****SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

**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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is 'sequence' for each recurrent 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**

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 neural network.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## **Hardware Options**

### **ExecutionEnvironment — Hardware resource for training neural network**

'auto' (default) | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'

Hardware resource for training neural 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 Computing Requirements” (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.

The 'multi-gpu' and 'parallel' options do not support neural networks containing custom layers with state parameters or built-in layers that are stateful at training time. For example:

- recurrent layers such as `LSTMLayer`, `BiLSTMLayer`, or `GRULayer` objects when the `SequenceLength` training option is a positive integer
- `BatchNormalizationLayer` objects when the `BatchNormalizationStatistics` training option is set to `'moving'`

**WorkerLoad — Parallel worker load division**

scalar from 0 to 1 | positive integer | numeric vector

Parallel worker load division between GPUs or CPUs, specified as one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for neural network training computation. If you train the neural 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 neural network training computation. If you train the neural 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 — Neural network training load for each worker in the parallel pool. For a vector  $W$ , worker  $i$  gets a fraction  $W(i)/\text{sum}(W)$  of the work (number of examples per mini-batch). If you train a neural 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**DispatchInBackground — Flag to enable background dispatch**

0 (false) (default) | 1 (true)

Flag to enable background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as 0 (false) or 1 (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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Checkpoints****CheckpointPath — Path for saving checkpoint neural networks**

"" (default) | character vector

Path for saving the checkpoint neural networks, specified as a character vector or string scalar.

- If you do not specify a path (that is, you use the default `""`), then the software does not save any checkpoint neural networks.
- If you specify a path, then `trainNetwork` saves checkpoint neural networks to this path and assigns a unique name to each neural network. You can then load any checkpoint neural network and resume training from that neural network.

If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint neural networks. If the path you specify does not exist, then `trainingOptions` returns an error.

The `CheckpointFrequency` and `CheckpointFrequencyUnit` options specify the frequency of saving checkpoint neural networks.

For more information about saving neural network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Data Types: `char` | `string`

### **CheckpointFrequency — Frequency of saving checkpoint neural networks**

1 (default) | positive integer

Frequency of saving checkpoint neural networks, specified as a positive integer.

If `CheckpointFrequencyUnit` is `'epoch'`, then the software saves checkpoint neural networks every `CheckpointFrequency` epochs.

If `CheckpointFrequencyUnit` is `'iteration'`, then the software saves checkpoint neural networks every `CheckpointFrequency` iterations.

This option only has an effect when `CheckpointPath` is nonempty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CheckpointFrequencyUnit — Checkpoint frequency unit**

`'epoch'` (default) | `'iteration'`

Checkpoint frequency unit, specified as `'epoch'` or `'iteration'`.

If `CheckpointFrequencyUnit` is `'epoch'`, then the software saves checkpoint neural networks every `CheckpointFrequency` epochs.

If `CheckpointFrequencyUnit` is `'iteration'`, then the software saves checkpoint neural networks every `CheckpointFrequency` iterations.

This option only has an effect when `CheckpointPath` is nonempty.

### **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

Field	Description
ValidationLoss	Loss on the validation data
BaseLearnRate	Current base learning rate
TrainingAccuracy	Accuracy on the current mini-batch (classification neural networks)
TrainingRMSE	RMSE on the current mini-batch (regression neural networks)
ValidationAccuracy	Accuracy on the validation data (classification neural networks)
ValidationRMSE	RMSE on the validation data (regression neural 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 1 (true). If any output function returns 1 (true), then training finishes and `trainNetwork` returns the latest neural 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: ''
  CheckpointFrequency: 1
CheckpointFrequencyUnit: 'epoch'
ExecutionEnvironment: 'auto'
  WorkerLoad: []
  OutputFcn: []
  Plots: 'training-progress'
  SequenceLength: 'longest'
  SequencePaddingValue: 0
SequencePaddingDirection: 'right'
DispatchInBackground: 0
ResetInputNormalization: 1
BatchNormalizationStatistics: 'population'
OutputNetwork: 'last-iteration'

```

## Algorithms

### Adam

Adam (derived from *adaptive moment estimation*) [1] 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  $\beta_1$  and  $\beta_2$  decay rates using the `GradientDecayFactor` and `SquaredGradientDecayFactor` training options, respectively. Adam uses the moving averages to update the network parameters as

$$\theta_{\ell+1} = \theta_\ell - \frac{\alpha m_\ell}{\sqrt{|v_\ell|} + \epsilon}$$

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  $\epsilon$  by using the `Epsilon` training option. 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 'adam' as the first input to `trainingOptions`. The full Adam update also includes a mechanism to correct a bias that appears in the beginning of training. For more information, see [1].

Specify the learning rate  $\alpha$  for all optimization algorithms using the `InitialLearnRate` training option. 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”.

## **L<sub>2</sub> 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,  $\lambda$  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  $\lambda$  by using the `L2Regularization` training option. 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.

## **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.

## **Version History**

**Introduced in R2018a**

**R2022b: `trainNetwork` pads mini-batches to length of longest sequence before splitting when you specify `SequenceLength` training option as an integer**

Starting in R2022b, when you train a neural network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the neural network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

### **R2018b: 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.

## **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 Neural 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"

# 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 first input argument.


## Properties

### Plots and Display

#### Plots — Plots to display during neural network training

'none' (default) | 'training-progress'

Plots to display during neural 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 neural network. You can save the training plot as an image or PDF by clicking **Export Training Plot**. For more information on the training progress plot, see “Monitor Deep Learning 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 Neural Networks**

<b>Field</b>	<b>Description</b>
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 <code>ClassificationOutputLayer</code> 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 <code>ClassificationOutputLayer</code> 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 Neural 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 <code>RegressionOutputLayer</code> object, then the loss is the half-mean-squared-error.
Validation Loss	Loss on the validation data. If the output layer is a <code>RegressionOutputLayer</code> 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: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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 a positive integer. This option only has an effect when the `Verbose` training option is 1 (true).

If you validate the neural network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Mini-Batch Options

#### MaxEpochs — Maximum number of epochs

30 (default) | 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **MiniBatchSize — Size of mini-batch**

128 (default) | 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.

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Shuffle — Option for data shuffling**

'once' (default) | '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 neural 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` training option to 'every-epoch'.

## **Validation**

### **ValidationData — Data to use for validation during training**

[] (default) | 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 neural 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 neural network validation.

If `ValidationData` is `[]`, then the software does not validate the neural network during training.

**ValidationFrequency — Frequency of neural network validation**

50 (default) | positive integer

Frequency of neural network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics. To specify validation data, use the `ValidationData` training option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ValidationPatience — Patience of validation stopping**

Inf (default) | positive integer

Patience of validation stopping of neural 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 neural network training stops. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned neural network depends on the `OutputNetwork` training option. To return the neural network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**OutputNetwork — Neural network to return when training completes**

'last-iteration' (default) | 'best-validation-loss'

Neural network to return when training completes, specified as one of the following:

- `'last-iteration'` - Return the neural network corresponding to the last training iteration.
- `'best-validation-loss'` - Return the neural network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify the `ValidationData` training option.

**Solver Options****InitialLearnRate — Initial learning rate**

0.001 (default) | positive scalar

Initial learning rate used for training, specified as a positive scalar.

If the learning rate is too low, then training can take a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LearnRateScheduleSettings — Settings for learning rate schedule**

structure

This property is read-only.



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<sub>2</sub> regularization**

0.0001 (default) | nonnegative scalar

Factor for L<sub>2</sub> regularization (weight decay), specified as a nonnegative scalar. For more information, see “L2 Regularization” on page 1-1814.

You can specify a multiplier for the L<sub>2</sub> regularization for neural network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SquaredGradientDecayFactor — Decay rate of squared gradient moving average**

0.9 (default) | nonnegative scalar less than 1

Decay rate of squared gradient moving average for the RMSProp solver, specified as a nonnegative scalar less than 1.

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.

For more information, see “RMSProp” on page 1-1814.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Epsilon — Denominator offset**

1e-8 (default) | positive scalar

Denominator offset for the RMSProp solver, specified as a positive scalar.

The solver adds the offset to the denominator in the neural network parameter updates to avoid division by zero. The default value works well for most tasks.

For more information about the different solvers, see “RMSProp” on page 1-1814.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **ResetInputNormalization — Option to reset input layer normalization**

1 (true) (default) | 0 (false)

Option to reset input layer normalization, specified as one of the following:

- 1 (true) — Reset the input layer normalization statistics and recalculate them at training time.
- 0 (false) — Calculate normalization statistics at training time when they are empty.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **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  $\mu^*$  and  $\sigma^{2*}$  denote the updated mean and variance, respectively,  $\lambda_{\mu}$  and  $\lambda_{\sigma^2}$  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  $\mu$  and  $\sigma^2$  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 Inf or a positive scalar. If the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to the GradientThresholdMethod training option.

For more information, see Gradient Clipping on page 1-1815.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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 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 1-1815.

## Sequence Options

### SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### 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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is `'sequence'` for each recurrent 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**`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 neural network.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**Hardware Options****ExecutionEnvironment — Hardware resource for training neural network**`'auto' (default) | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'`

Hardware resource for training neural 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 Computing Requirements” (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.

The `'multi-gpu'` and `'parallel'` options do not support neural networks containing custom layers with state parameters or built-in layers that are stateful at training time. For example:

- recurrent layers such as `LSTMlayer`, `BiLSTMlayer`, or `GRUlayer` objects when the `SequenceLength` training option is a positive integer
- `BatchNormalizationLayer` objects when the `BatchNormalizationStatistics` training option is set to `'moving'`

**WorkerLoad — Parallel worker load division**`scalar from 0 to 1 | positive integer | numeric vector`

Parallel worker load division between GPUs or CPUs, specified as one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for neural network training computation. If you train the neural 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 neural network training computation. If you train the neural 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 — Neural network training load for each worker in the parallel pool. For a vector  $W$ , worker  $i$  gets a fraction  $W(i) / \text{sum}(W)$  of the work (number of examples per mini-batch). If you train a neural 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DispatchInBackground — Flag to enable background dispatch**

0 (false) (default) | 1 (true)

Flag to enable background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as 0 (false) or 1 (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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Checkpoints**

### **CheckpointPath — Path for saving checkpoint neural networks**

"" (default) | character vector

Path for saving the checkpoint neural networks, specified as a character vector or string scalar.

- If you do not specify a path (that is, you use the default ""), then the software does not save any checkpoint neural networks.
- If you specify a path, then `trainNetwork` saves checkpoint neural networks to this path and assigns a unique name to each neural network. You can then load any checkpoint neural network and resume training from that neural network.

If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint neural networks. If the path you specify does not exist, then `trainingOptions` returns an error.

The `CheckpointFrequency` and `CheckpointFrequencyUnit` options specify the frequency of saving checkpoint neural networks.

For more information about saving neural network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Data Types: char | string

**CheckpointFrequency — Frequency of saving checkpoint neural networks**

1 (default) | positive integer

Frequency of saving checkpoint neural networks, specified as a positive integer.

If CheckpointFrequencyUnit is 'epoch', then the software saves checkpoint neural networks every CheckpointFrequency epochs.

If CheckpointFrequencyUnit is 'iteration', then the software saves checkpoint neural networks every CheckpointFrequency iterations.

This option only has an effect when CheckpointPath is nonempty.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**CheckpointFrequencyUnit — Checkpoint frequency unit**

'epoch' (default) | 'iteration'

Checkpoint frequency unit, specified as 'epoch' or 'iteration'.

If CheckpointFrequencyUnit is 'epoch', then the software saves checkpoint neural networks every CheckpointFrequency epochs.

If CheckpointFrequencyUnit is 'iteration', then the software saves checkpoint neural networks every CheckpointFrequency iterations.

This option only has an effect when CheckpointPath is nonempty.

**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 neural networks)
TrainingRMSE	RMSE on the current mini-batch (regression neural networks)
ValidationAccuracy	Accuracy on the validation data (classification neural networks)

Field	Description
ValidationRMSE	RMSE on the validation data (regression neural 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 1 (true). If any output function returns 1 (true), then training finishes and `trainNetwork` returns the latest neural 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: ''
    CheckpointFrequency: 1
    CheckpointFrequencyUnit: 'epoch'
```

```
ExecutionEnvironment: 'auto'  
  WorkerLoad: []  
  OutputFcn: []  
    Plots: 'training-progress'  
  SequenceLength: 'longest'  
  SequencePaddingValue: 0  
  SequencePaddingDirection: 'right'  
  DispatchInBackground: 0  
  ResetInputNormalization: 1  
  BatchNormalizationStatistics: 'population'  
  OutputNetwork: 'last-iteration'
```

## Algorithms

### 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,

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) [\nabla E(\theta_t)]^2$$

$\beta_2$  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  $1/(1-\beta_2)$ , that is, 10, 100, and 1000 parameter updates, respectively. You can specify  $\beta_2$  by using the `SquaredGradientDecayFactor` training options. The RMSProp algorithm uses this moving average to normalize the updates of each parameter individually,

$$\theta_{t+1} = \theta_t - \frac{\alpha \nabla E(\theta_t)}{\sqrt{v_t} + \epsilon}$$

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.  $\epsilon$  is a small constant added to avoid division by zero. You can specify  $\epsilon$  by using the `Epsilon` training option, but the default value usually works well. To use RMSProp to train a neural network, specify `'rmsprop'` as the first input to `trainingOptions`.

### L<sub>2</sub> 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,  $\lambda$  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  $\lambda$  by using the `L2Regularization` training option. 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.

### 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.

## Version History

### Introduced in R2018a

#### **R2022b: `trainNetwork` pads mini-batches to length of longest sequence before splitting when you specify `SequenceLength` training option as an integer**

Starting in R2022b, when you train a neural network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the neural network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

#### **R2018b: `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.

## **See Also**

`trainNetwork` | `trainingOptions`

## **Topics**

“Create Simple Deep Learning Neural 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”

# 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 first input argument.


## Properties

### Plots and Display

#### Plots — Plots to display during neural network training

'none' (default) | 'training-progress'

Plots to display during neural 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 neural network. You can save the training plot as an image or PDF by clicking **Export Training Plot**. For more information on the training progress plot, see “Monitor Deep Learning 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 Neural Networks**

<b>Field</b>	<b>Description</b>
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 <code>ClassificationOutputLayer</code> 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 <code>ClassificationOutputLayer</code> 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 Neural 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 <code>RegressionOutputLayer</code> object, then the loss is the half-mean-squared-error.
Validation Loss	Loss on the validation data. If the output layer is a <code>RegressionOutputLayer</code> 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: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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 a positive integer. This option only has an effect when the `Verbose` training option is 1 (true).

If you validate the neural network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Mini-Batch Options

#### MaxEpochs — Maximum number of epochs

30 (default) | 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **MiniBatchSize — Size of mini-batch**

128 (default) | 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.

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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Shuffle — Option for data shuffling**

'once' (default) | '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 neural 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` training option to 'every-epoch'.

## **Validation**

### **ValidationData — Data to use for validation during training**

[] (default) | 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 neural 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 neural network validation.

If `ValidationData` is `[]`, then the software does not validate the neural network during training.

### **ValidationFrequency — Frequency of neural network validation**

50 (default) | positive integer

Frequency of neural network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics. To specify validation data, use the `ValidationData` training option.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **ValidationPatience — Patience of validation stopping**

Inf (default) | positive integer

Patience of validation stopping of neural 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 neural network training stops. If `ValidationPatience` is `Inf`, then the values of the validation loss do not cause training to stop early.

The returned neural network depends on the `OutputNetwork` training option. To return the neural network with the lowest validation loss, set the `OutputNetwork` training option to `"best-validation-loss"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OutputNetwork — Neural network to return when training completes**

'last-iteration' (default) | 'best-validation-loss'

Neural network to return when training completes, specified as one of the following:

- `'last-iteration'` - Return the neural network corresponding to the last training iteration.
- `'best-validation-loss'` - Return the neural network corresponding to the training iteration with the lowest validation loss. To use this option, you must specify the `ValidationData` training option.

## **Solver Options**

### **InitialLearnRate — Initial learning rate**

0.01 (default) | positive scalar

Initial learning rate used for training, specified as a positive scalar.

If the learning rate is too low, then training can take a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **LearnRateScheduleSettings — Settings for learning rate schedule**

structure

This property is read-only.

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<sub>2</sub> regularization**

`0.0001` (default) | nonnegative scalar

Factor for L<sub>2</sub> regularization (weight decay), specified as a nonnegative scalar. For more information, see “L2 Regularization” on page 1-1829.

You can specify a multiplier for the L<sub>2</sub> regularization for neural network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 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. The default value works well for most tasks.

For more information, see “Stochastic Gradient Descent with Momentum” on page 1-1829.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **ResetInputNormalization — Option to reset input layer normalization**

`1` (true) (default) | `0` (false)

Option to reset input layer normalization, specified as one of the following:

- `1` (true) — Reset the input layer normalization statistics and recalculate them at training time.
- `0` (false) — Calculate normalization statistics at training time when they are empty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **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

$$\mu^* = \lambda_\mu \widehat{\mu} + (1 - \lambda_\mu)\mu$$

$$\sigma^{2*} = \lambda_\sigma 2\widehat{\sigma^2} + (1 - \lambda_\sigma)\sigma^2$$

where  $\mu^*$  and  $\sigma^{2*}$  denote the updated mean and variance, respectively,  $\lambda_\mu$  and  $\lambda_\sigma$  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  $\mu$  and  $\sigma^2$  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 Inf or a positive scalar. If the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to the GradientThresholdMethod training option.

For more information, see Gradient Clipping on page 1-1829.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### 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 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 1-1829.

## Sequence Options

### SequenceLength — Option to pad or truncate 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 neural 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 length of the longest sequence 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. If the specified sequence length does not evenly divide the sequence lengths of the data, then the mini-batches containing the ends of those sequences have length shorter than the specified sequence length. 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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **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 recurrent layers process sequence data one time step at a time, when the recurrent 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 neural networks (when the `OutputMode` property is `'sequence'` for each recurrent 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**

`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 neural network.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Hardware Options**

#### **ExecutionEnvironment — Hardware resource for training neural network**

`'auto'` (default) | `'cpu'` | `'gpu'` | `'multi-gpu'` | `'parallel'`

Hardware resource for training neural 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 Computing Requirements” (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.

The 'multi-gpu' and 'parallel' options do not support neural networks containing custom layers with state parameters or built-in layers that are stateful at training time. For example:

- recurrent layers such as `LSTMLayer`, `BiLSTMLayer`, or `GRULayer` objects when the `SequenceLength` training option is a positive integer
- `BatchNormalizationLayer` objects when the `BatchNormalizationStatistics` training option is set to 'moving'

#### **WorkerLoad — Parallel worker load division**

scalar from 0 to 1 | positive integer | numeric vector

Parallel worker load division between GPUs or CPUs, specified as one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for neural network training computation. If you train the neural 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 neural network training computation. If you train the neural 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 — Neural network training load for each worker in the parallel pool. For a vector  $W$ , worker  $i$  gets a fraction  $W(i)/\text{sum}(W)$  of the work (number of examples per mini-batch). If you train a neural 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.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DispatchInBackground — Flag to enable background dispatch**

`0` (false) (default) | `1` (true)

Flag to enable background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as `0` (false) or `1` (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”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Checkpoints**

### **CheckpointPath — Path for saving checkpoint neural networks**

`''` (default) | character vector

Path for saving the checkpoint neural networks, specified as a character vector or string scalar.

- If you do not specify a path (that is, you use the default `''`), then the software does not save any checkpoint neural networks.
- If you specify a path, then `trainNetwork` saves checkpoint neural networks to this path and assigns a unique name to each neural network. You can then load any checkpoint neural network and resume training from that neural network.

If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint neural networks. If the path you specify does not exist, then `trainingOptions` returns an error.

The `CheckpointFrequency` and `CheckpointFrequencyUnit` options specify the frequency of saving checkpoint neural networks.

For more information about saving neural network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Data Types: `char` | `string`

### **CheckpointFrequency — Frequency of saving checkpoint neural networks**

`1` (default) | positive integer

Frequency of saving checkpoint neural networks, specified as a positive integer.

If `CheckpointFrequencyUnit` is `'epoch'`, then the software saves checkpoint neural networks every `CheckpointFrequency` epochs.

If `CheckpointFrequencyUnit` is `'iteration'`, then the software saves checkpoint neural networks every `CheckpointFrequency` iterations.

This option only has an effect when `CheckpointPath` is nonempty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**CheckpointFrequencyUnit — Checkpoint frequency unit**

'epoch' (default) | 'iteration'

Checkpoint frequency unit, specified as 'epoch' or 'iteration'.

If CheckpointFrequencyUnit is 'epoch', then the software saves checkpoint neural networks every CheckpointFrequency epochs.

If CheckpointFrequencyUnit is 'iteration', then the software saves checkpoint neural networks every CheckpointFrequency iterations.

This option only has an effect when CheckpointPath is nonempty.

**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 neural networks)
TrainingRMSE	RMSE on the current mini-batch (regression neural networks)
ValidationAccuracy	Accuracy on the validation data (classification neural networks)
ValidationRMSE	RMSE on the validation data (regression neural 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 1 (true). If any output function returns 1 (true), then training finishes and `trainNetwork` returns the latest neural 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: ''  
        CheckpointFrequency: 1  
        CheckpointFrequencyUnit: 'epoch'  
        ExecutionEnvironment: 'auto'  
        WorkerLoad: []  
        OutputFcn: []  
        Plots: 'training-progress'  
        SequenceLength: 'longest'  
        SequencePaddingValue: 0  
        SequencePaddingDirection: 'right'  
        DispatchInBackground: 0  
        ResetInputNormalization: 1  
        BatchNormalizationStatistics: 'population'  
        OutputNetwork: 'last-iteration'
```

## Algorithms

### 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  $\gamma$  determines the contribution of the previous gradient step to the current iteration. You can specify this value using the `Momentum` training option. To train a neural network using the stochastic gradient descent with momentum algorithm, specify `'sgdm'` as the first input argument to `trainingOptions`. To specify the initial value of the learning rate  $\alpha$ , use the `InitialLearnRate` training option. 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”.

### L<sub>2</sub> 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,  $\lambda$  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  $\lambda$  by using the `L2Regularization` training option. 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.

### 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.

## Version History

### Introduced in R2016a

#### **R2022b: `trainNetwork` pads mini-batches to length of longest sequence before splitting when you specify `SequenceLength` training option as an integer**

Starting in R2022b, when you train a neural network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the neural network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

#### **R2018b: `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.

## See Also

`trainNetwork` | `trainingOptions`

### Topics

“Create Simple Deep Learning Neural 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”



# trainingProgressMonitor

Monitor and plot training progress for deep learning custom training loops

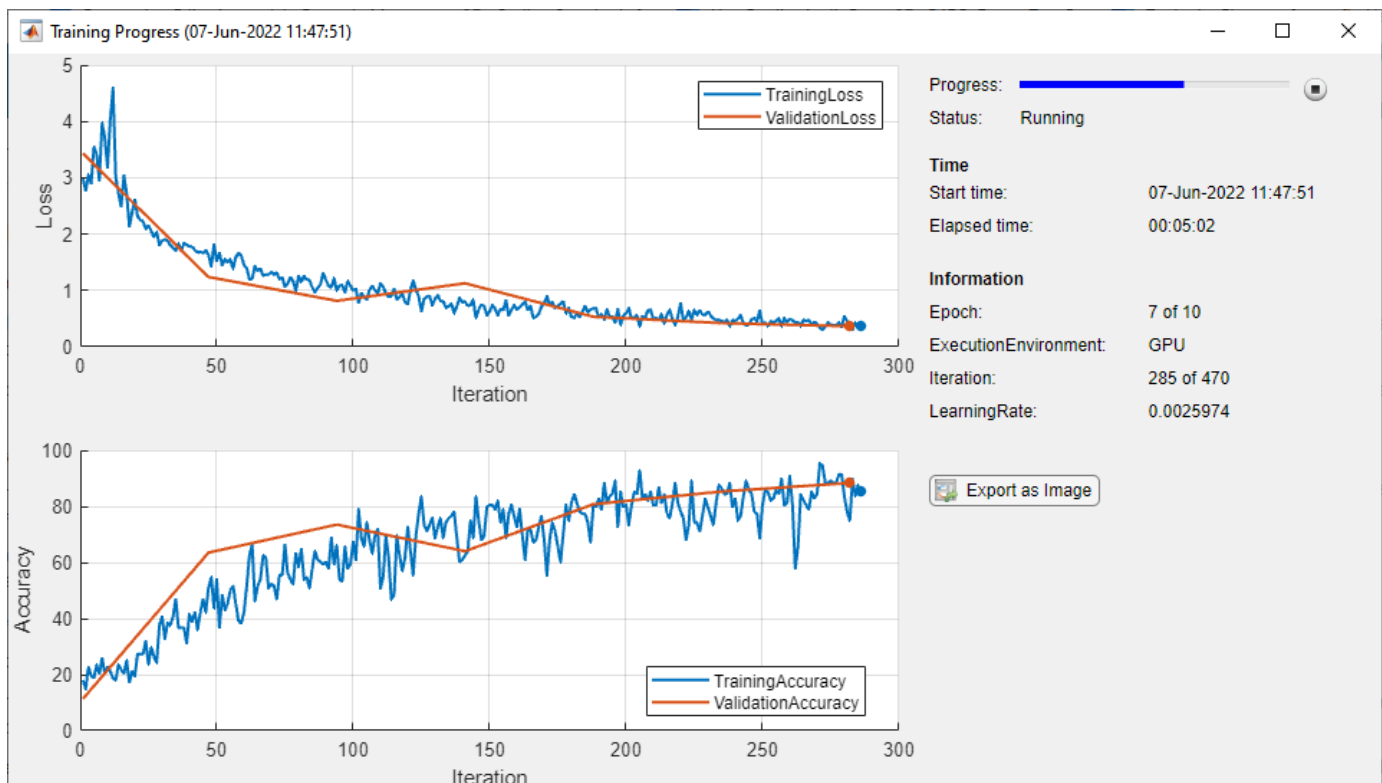
## Description

Use a TrainingProgressMonitor object to track training progress when using a custom training loop.

You can use a TrainingProgressMonitor object to:

- Create animated custom metric plots and record custom metrics during training.
- Display and record training information during training.
- Stop training early.
- Track training progress with a progress bar.
- Track elapsed time.

This image shows an example of the Training Progress window during training. For more information about configuring the Training Progress window and an example showing how to generate this figure, see “Monitor Custom Training Loop Progress”.



## Creation

### Syntax

```
monitor = trainingProgressMonitor  
monitor = trainingProgressMonitor(Name=Value)
```

### Description

`monitor = trainingProgressMonitor` creates a `TrainingProgressMonitor` object that you can use to track the training progress and create training plots.

`monitor = trainingProgressMonitor(Name=Value)` sets the `Metrics`, `Info`, `Visible`, `Progress`, `Status`, and `XLabel` properties using one or more name-value arguments.

### Properties

#### Metrics — Metric names

"" (default) | string scalar | character vector | string array | cell array of character vectors

Metric names, specified as a string scalar, character vector, string array, or cell array of character vectors. Valid names begin with a letter, and contain letters, digits, and underscores. Each metric appears in its own training subplot. To plot more than one metric in a single subplot, use the `groupSubPlot` function.

Example: `["TrainingLoss", "ValidationLoss"];`

Data Types: `char` | `string` | `cell`

#### Info — Information names

"" (default) | string scalar | character vector | string array | cell array of character vectors

Information names, specified as a string scalar, character vector, string array, or cell array of character vectors. Valid names begin with a letter, and contain letters, digits, and underscores. These names appear in the Training Progress window but do not appear as training plots.

Example: `["GradientDecayFactor", "SquaredGradientDecayFactor"];`

Data Types: `char` | `string` | `cell`

#### Stop — Request to stop training

false or 0 (default) | true or 1

This property is read-only.

Request to stop training, specified as a numeric or logical 0 (false) or 1 (true). The value of this property changes to 1 when you click the **Stop** button in the Training Progress window. The **Stop** button only appears if you set the `Visible` property to 'on' or 1 (true).

Data Types: `logical`

#### 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 Training Progress window.
- 'off' — Hide the Training Progress window without deleting it. You still can access the properties of an invisible object.

Example: 'off'

### **Progress — Training progress percentage**

0 (default) | scalar in the range [0, 100] | d`l`array object in the range [0, 100]

Training progress percentage, specified as a scalar or d`l`array object in the range [0, 100].

Example: 17;

### **XLabel — Horizontal axis label**

"" (default) | string scalar | character vector

Horizontal axis label in the training plot, specified as a string scalar or character vector.

Example: "Iteration";

Data Types: char | string | cell

### **Status — Training status**

"" (default) | string scalar | character vector

Training status, specified as a string scalar or character vector.

Example: "Running";

Data Types: char | string | cell

### **MetricData — Metric values**

structure

This property is read-only.

Metric values, specified as a structure. Use the `Metrics` property to specify the field names for the structure. Each field contains a matrix with two columns. The first column contains the custom training loop step values and the second column contains the metric values recorded by the `recordMetrics` function.

Data Types: struct

### **InfoData — Information values**

structure

This property is read-only.

Information values, specified as a structure. Use the `Info` property to specify the field names for the structure. Each field is a column vector that contains the values updated by the `updateInfo` function.

Data Types: struct

## Object Functions

groupSubPlot	Group metrics in training plot
recordMetrics	Record metric values for custom training loops
updateInfo	Update information values for custom training loops

## Examples

### Track Progress and Produce Training Plots

Use a `TrainingProgressMonitor` object to track training progress and produce training plots for custom training loops.

Create a `TrainingProgressMonitor` object. The monitor automatically tracks the start time and the elapsed time. The timer starts when you create the object.

---

**Tip** To ensure that the elapsed time accurately reflects the training time, make sure you create the `TrainingProgressMonitor` object close to the start of your custom training loop.

---

```
monitor = trainingProgressMonitor;
```

Before you start the training, specify names for the information and metric values.

```
monitor.Info = ["LearningRate", "Epoch", "Iteration"];  
monitor.Metrics = ["TrainingLoss", "ValidationLoss", "TrainingAccuracy", "ValidationAccuracy"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot, and group the training and validation accuracy in the same plot.

```
monitor.XLabel = "Iteration";  
groupSubPlot(monitor, "Loss", ["TrainingLoss", "ValidationLoss"]);  
groupSubPlot(monitor, "Accuracy", ["TrainingAccuracy", "ValidationAccuracy"]);
```

During training:

- Evaluate the `Stop` property at the start of each step in your custom training loop. When you click the **Stop** button in the Training Progress window, the `Stop` property changes to 1. Training stops if your training loop exits when the `Stop` property is 1.
- Update the information values. The updated values appear in the Training Progress window.
- Record the metric values. The recorded values appear in the training plot.
- Update the training progress percentage based on the fraction of iterations completed.

---

**Note** The following example code is a template. You must edit this training loop to compute your metric and information values. For a complete example that you can run in MATLAB, see “Monitor Custom Training Loop Progress During Training”.

---

```
epoch = 0;  
iteration = 0;  
  
monitor.Status = "Running";
```

```

while epoch < maxEpochs && ~monitor.Stop
    epoch = epoch + 1;

    while hasData(mbq) && ~monitor.Stop
        iteration = iteration + 1;

        % Add code to calculate metric and information values.
        % lossTrain = ...

        updateInfo(monitor, ...
            LearningRate=learnRate, ...
            Epoch=string(epoch) + " of " + string(maxEpochs), ...
            Iteration=string(iteration) + " of " + string(numIterations));

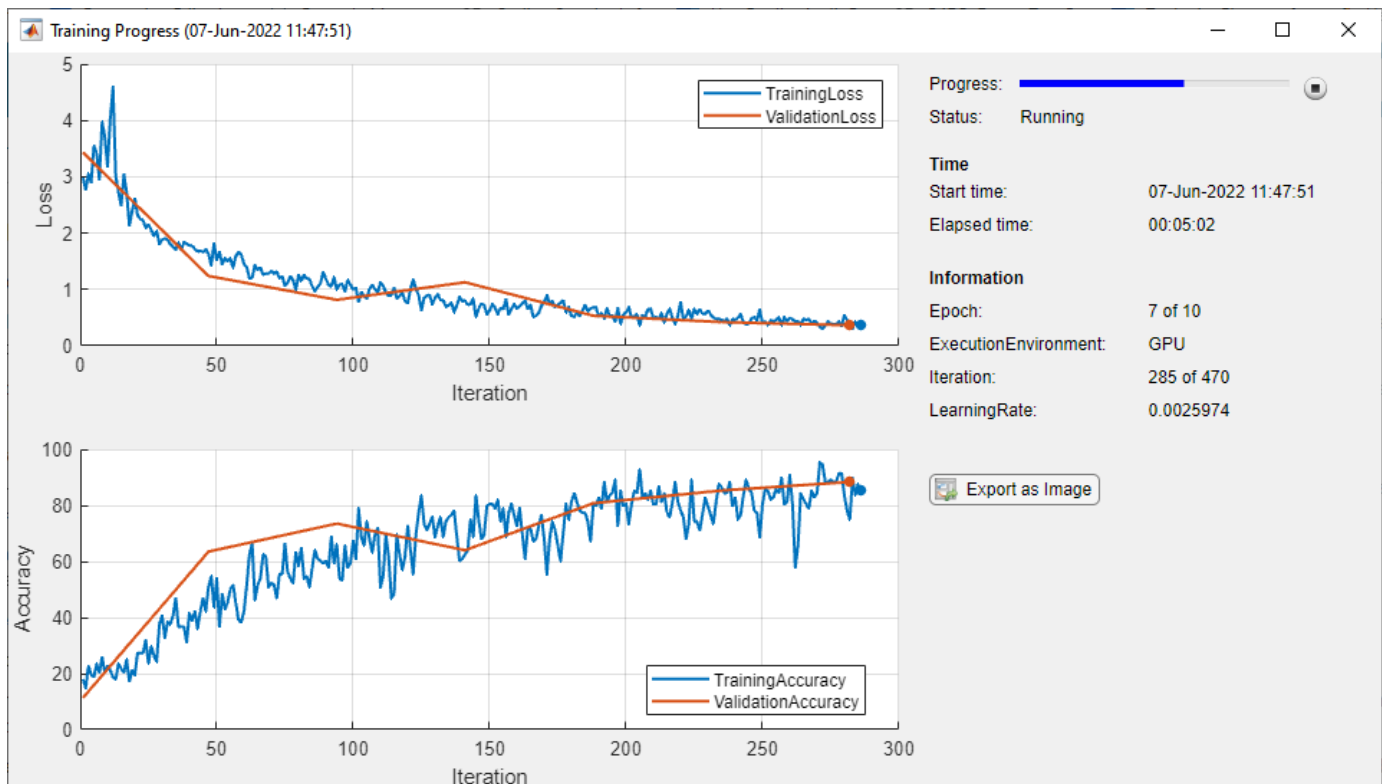
        recordMetrics(monitor, iteration, ...
            TrainingLoss=lossTrain, ...
            TrainingAccuracy=accuracyTrain, ...
            ValidationLoss=lossValidation, ...
            ValidationAccuracy=accuracyValidation);

        monitor.Progress = 100*iteration/numIterations;
    end
end

```

The Training Progress window shows animated plots of the metrics, and the information values, training progress bar, and elapsed time.

- The training plots update each time you call `recordMetrics`.
- The values under **Information** update each time you call `updateInfo`.
- The elapsed time updates each time you call `recordMetrics` or `updateInfo`, and when you update the property.



### Prepare Plotting Code for Custom Training Experiment

A `TrainingProgressMonitor` object has the same properties and object functions as an `experiments.Monitor` object. Therefore, you can easily adapt your plotting code for use in an **Experiment Manager** setup script.

How you monitor training depends on where you are training.

- If you are using a custom training loop script, you must create and manage a `TrainingProgressMonitor` object yourself.
- If you are using a custom training experiment, **Experiment Manager** creates an `experiments.Monitor` object for each trial of your experiment. By default, **Experiment Manager** saves the `experiments.Monitor` object as the variable `monitor`.

In **Experiment Manager**, you can use the `experiments.Monitor` object in place of the `TrainingProgressMonitor` object in your custom training loop code.

For example, suppose your training script creates a `TrainingProgressMonitor` object to track and plot training and validation loss.

```
monitor = trainingProgressMonitor( ...
    Metrics=["TrainingLoss","ValidationLoss"], ...
    XLabel="Iteration");

groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
```

```
iteration = 1;
recordMetrics(monitor, iteration, TrainingLoss=loss, ValidationLoss=lossVal);
```

To adapt this code for use in **Experiment Manager** with an `experiments.Monitor` object:

- Convert any code that sets properties using `Name=Value` syntax to use dot notation.
- Delete the call to `trainingProgressMonitor`. This is because **Experiment Manager** creates a monitor for you.

Use the adapted code inside your **Experiment Manager** setup function.

```
% Inside custom training experiment setup function
```

```
monitor.Metrics=["TrainingLoss", "ValidationLoss"];
monitor.XLabel = "Iteration";

groupSubPlot(monitor, "Loss", ["TrainingLoss", "ValidationLoss"]);

iteration = 1;
recordMetrics(monitor, iteration, TrainingLoss=loss, ValidationLoss=lossVal);
```

---

**Note** **Experiment Manager** accesses the monitor object as the second input argument of the training function. You must check that the second input argument matches the variable name of your monitor object. For more information, see “Configure Custom Training Experiment” on page 1-54.

---

## Tips

- The information values appear in the Training Progress window and the training plot shows a record of the metric values. Use information values for text and for numerical values that you want to display in the training window but not in the training plot.
- When you click the **Stop** button in the Training Progress window, the `Stop` property is set to 1 (`true`). This stops training if your training loop exits when the `Stop` property is 1. For example, to enable early stopping, include the following code in your custom training loop.

```
while numEpochs < maxEpochs && ~monitor.Stop
% Custom training loop code.
end
```

- The elapsed time updates each time you call `recordMetrics` or `updateInfo`, and when you update the `Progress` property.

## Version History

Introduced in R2022b

## See Also

`groupSubPlot` | `updateInfo` | `recordMetrics` | `experiments.Monitor`

## Topics

“Monitor Custom Training Loop Progress”

“Train Network Using Custom Training Loop”

“Detect Issues During Deep Neural Network Training”

## 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 = trainNetwork(mixed, 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) neural network for sequence and time-series data
- a multilayer perceptron (MLP) neural 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 neural 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 Computing Requirements” (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 neural 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) neural 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 = trainNetwork(mixed, layers, options)` trains a neural network with multiple inputs with mixed data types with the data and responses specified by `mixed`.

`[net, info] = trainNetwork( ___ )` also returns information on the training using any of the previous syntaxes.

## Examples

### Train Network for Image Classification

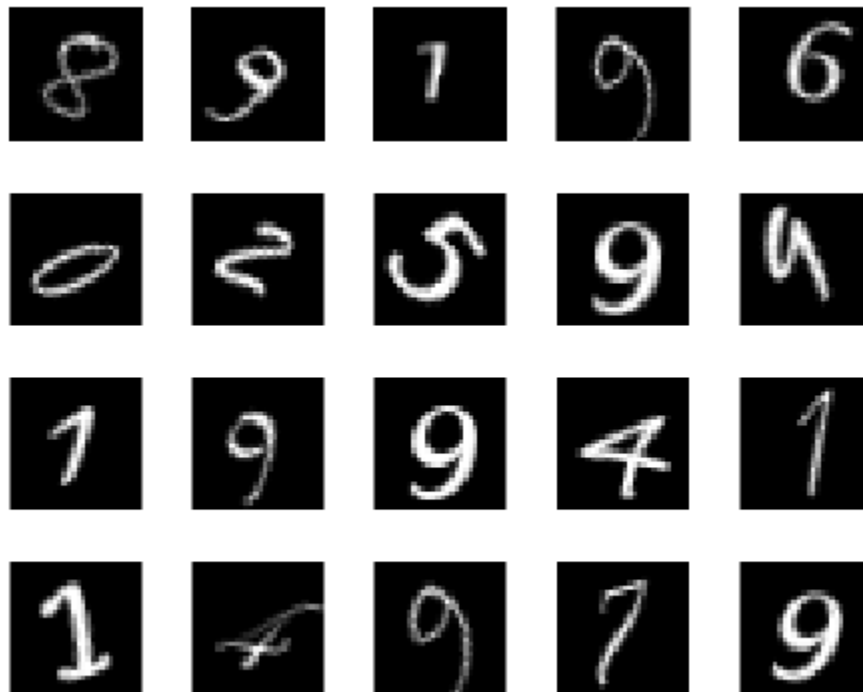
Load the data as an `ImageDatastore` object.

```
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.

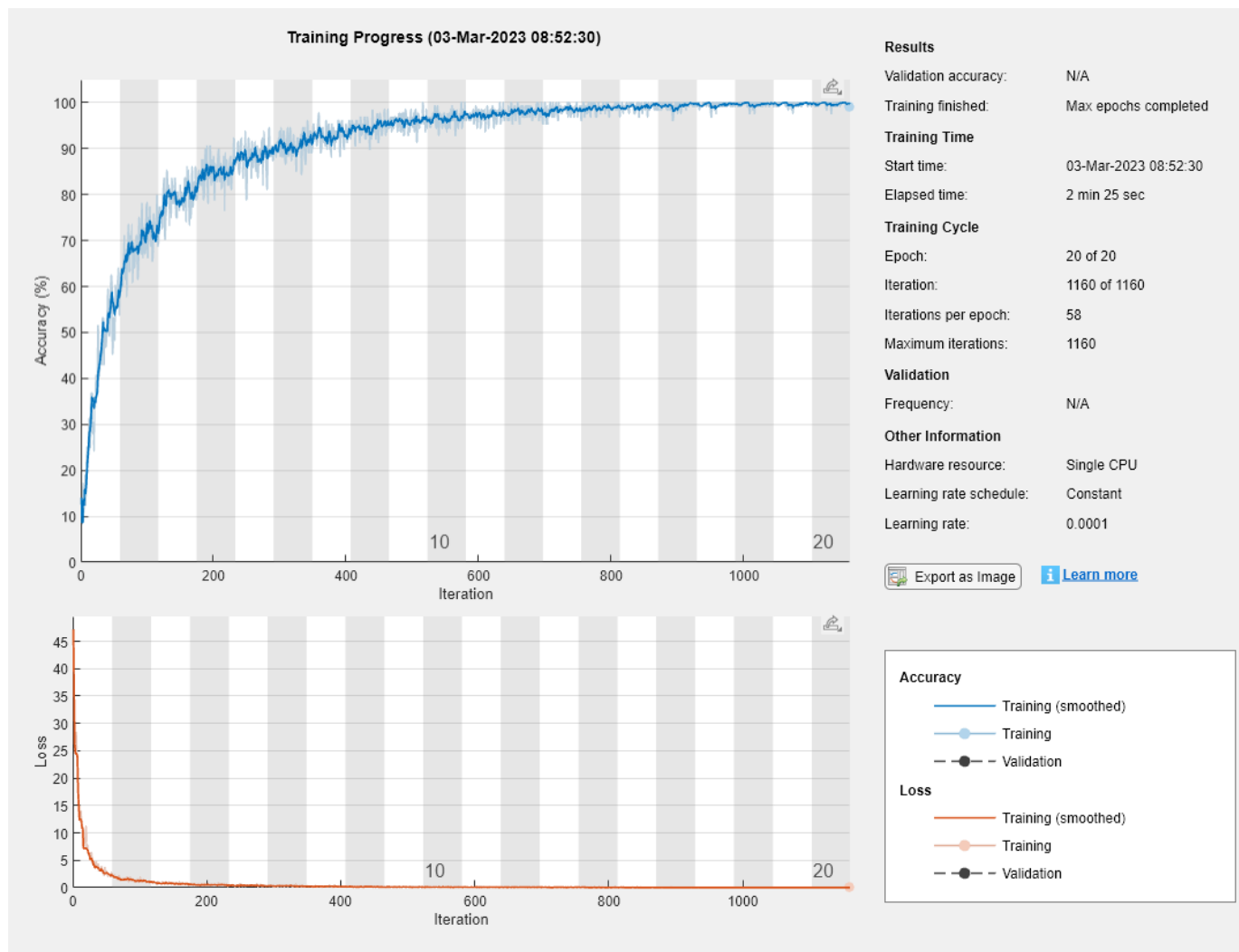
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

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 = 0.9416
```

## 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.

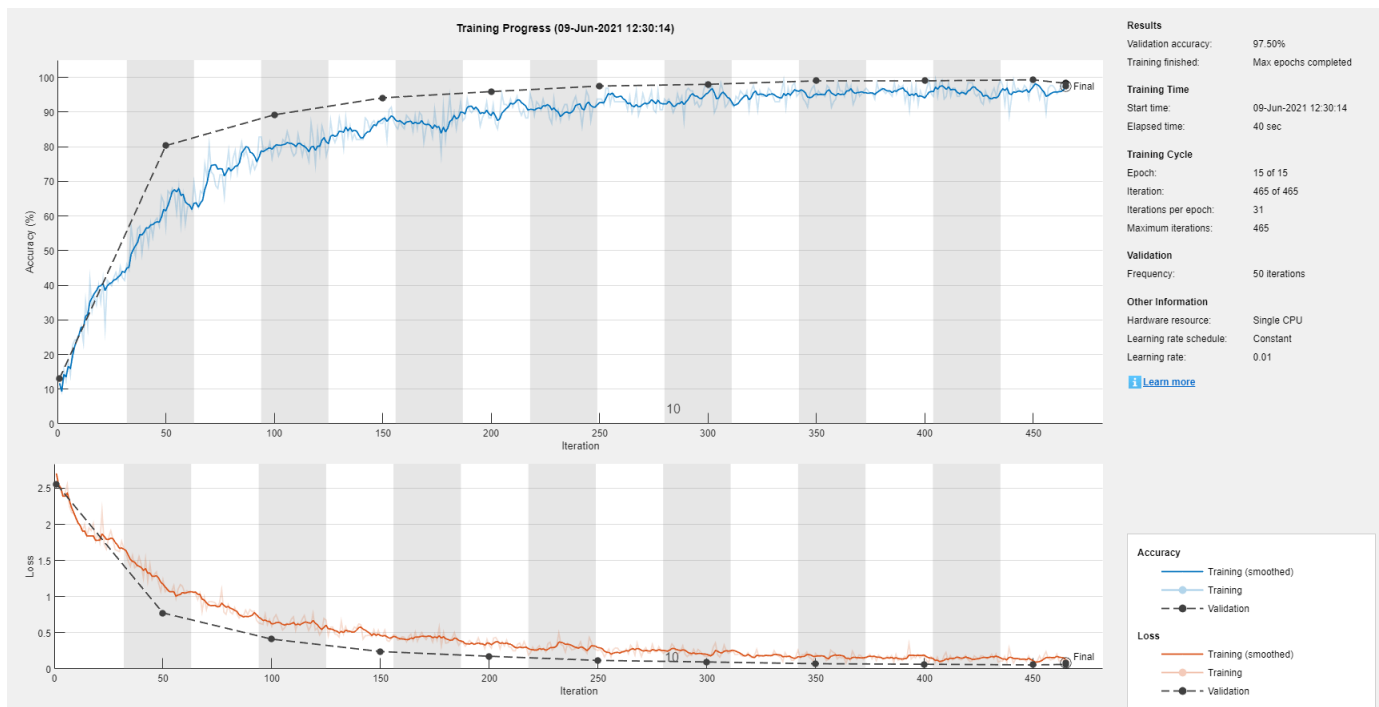
```
layers = [  
    imageInputLayer(imageSize)  
  
    convolution2dLayer(3,8,'Padding','same')  
    batchNormalizationLayer  
    reluLayer  
  
    maxPooling2dLayer(2,'Stride',2)  
  
    convolution2dLayer(3,16,'Padding','same')  
    batchNormalizationLayer  
    reluLayer  
  
    maxPooling2dLayer(2,'Stride',2)  
  
    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:

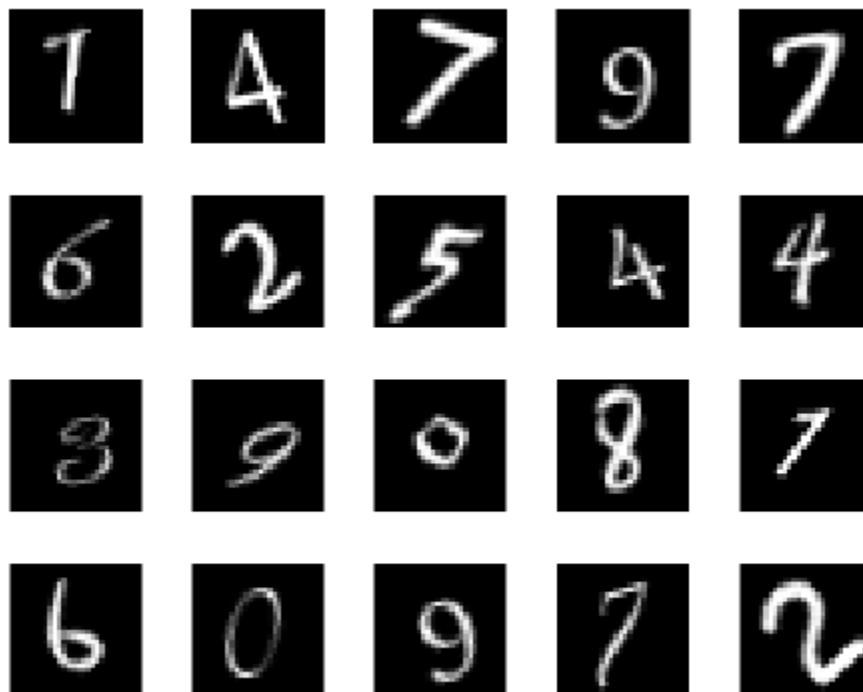
- 28 is the height and width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`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.

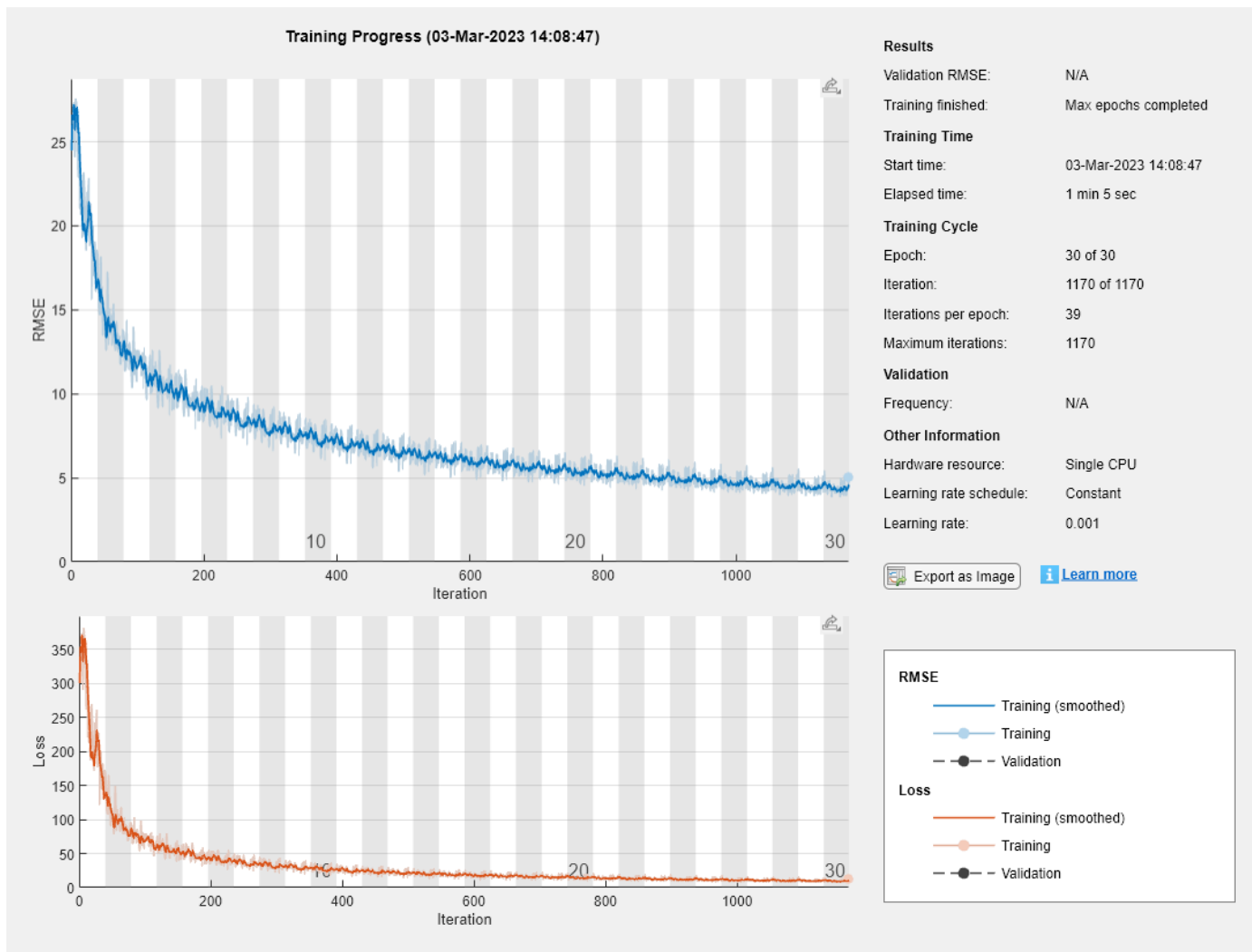
```
layers = [ ...  
    imageInputLayer([28 28 1])  
    convolution2dLayer(12,25)  
    reluLayer  
    fullyConnectedLayer(1)  
    regressionLayer];
```

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
      6.0339
```

### Train Network for Sequence Classification

Train a deep learning LSTM network for sequence-to-label classification.

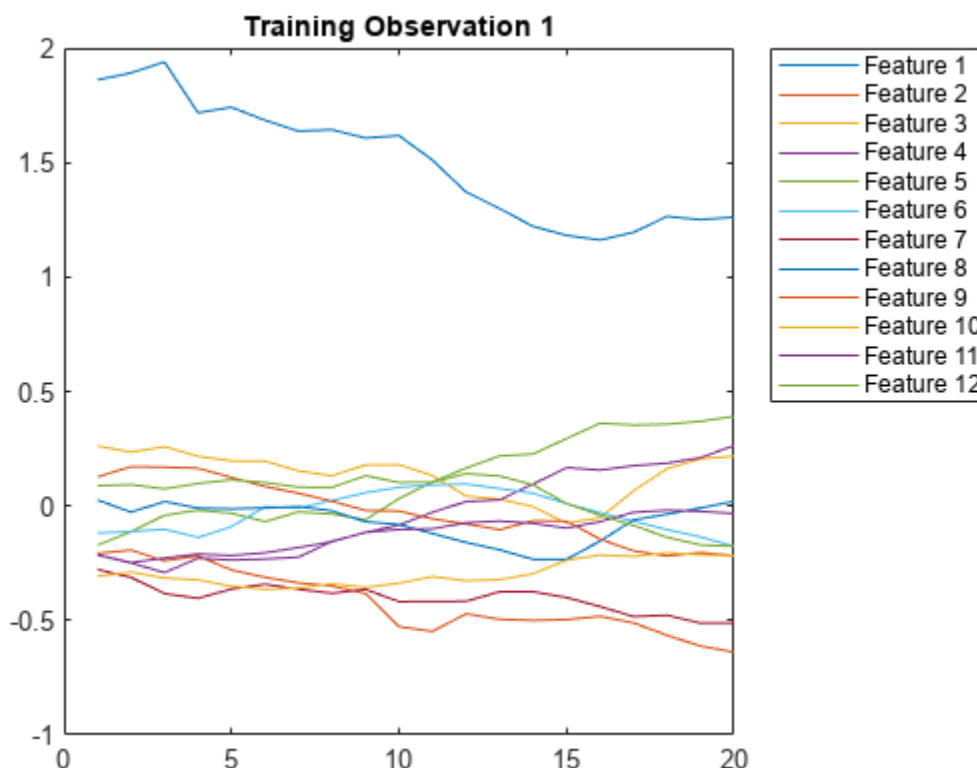


Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with 12 features corresponding to LPC cepstrum coefficients. `Y` is a categorical vector of labels 1,2,...,9. The entries in `XTrain` are matrices with 12 rows (one row for 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.

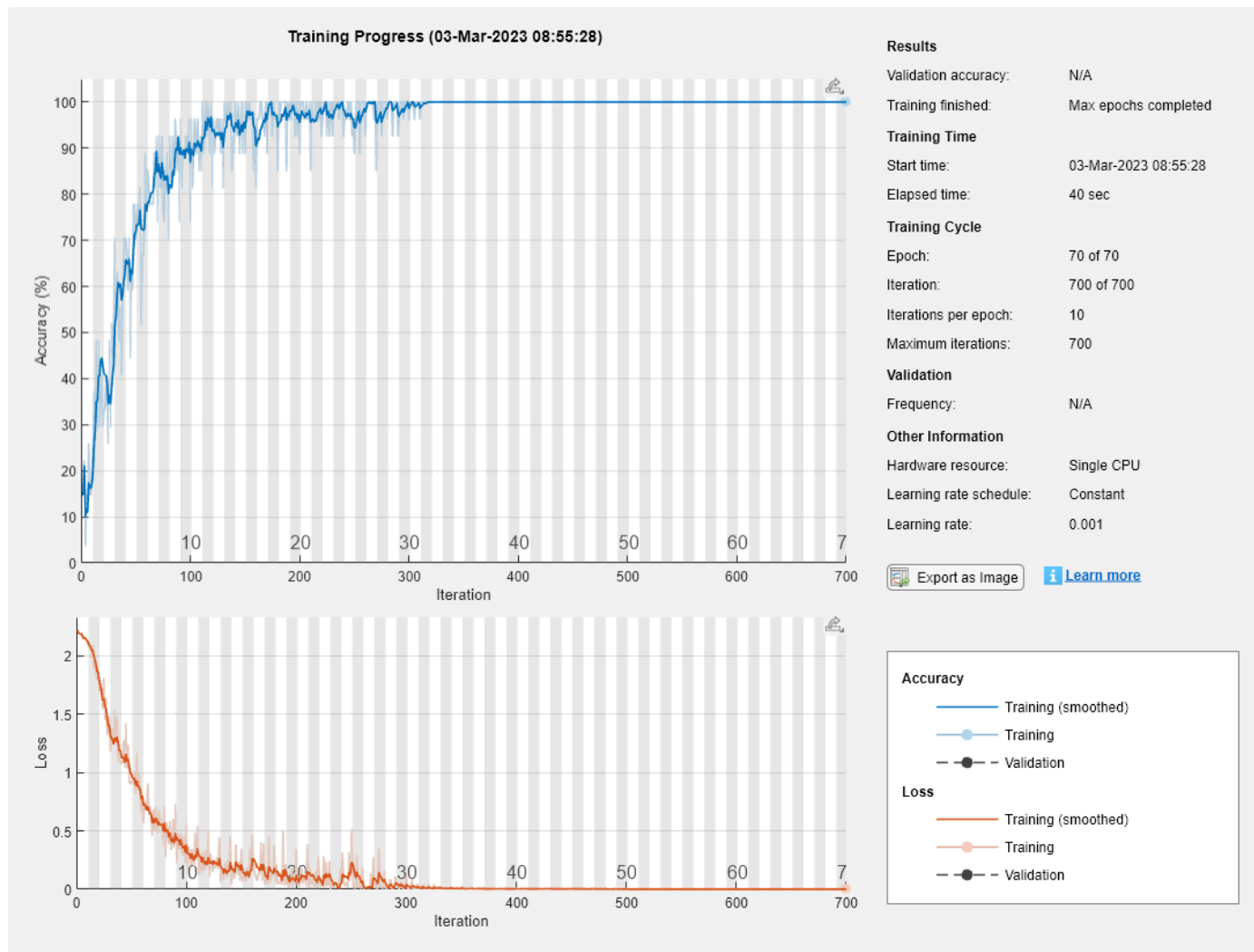
Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

```
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 = 0.9432
```

### 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
```

`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)
```

SigMean	SigMedian	SigRMS	SigVar	SigPeak	SigPeak2Peak	SigSkewness	SigKurtosis
-0.94876	-0.9722	1.3726	0.98387	0.81571	3.6314	-0.041525	2.7187
-0.97537	-0.98958	1.3937	0.99105	0.81571	3.6314	-0.023777	2.7187
1.0502	1.0267	1.4449	0.98491	2.8157	3.6314	-0.04162	2.7187
1.0227	1.0045	1.4288	0.99553	2.8157	3.6314	-0.016356	2.7187
1.0123	1.0024	1.4202	0.99233	2.8157	3.6314	-0.014701	2.7187
1.0275	1.0102	1.4338	1.0001	2.8157	3.6314	-0.02659	2.7187
1.0464	1.0275	1.4477	1.0011	2.8157	3.6314	-0.042849	2.7187
1.0459	1.0257	1.4402	0.98047	2.8157	3.6314	-0.035405	2.7187

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 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];
```

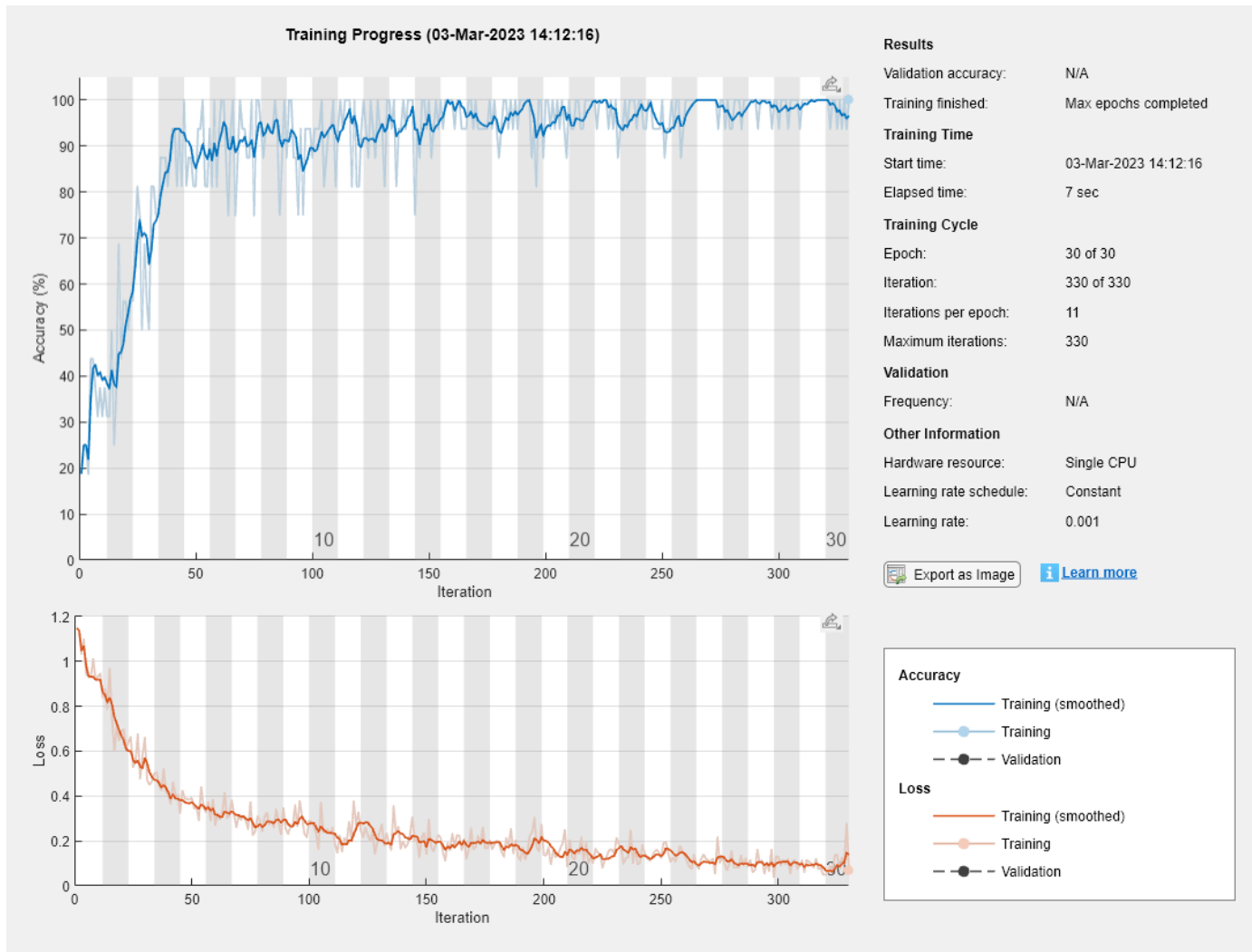
Specify the training options.

```
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 of images saved on disk.	<p>Train image classification neural network with images saved on disk, where the images are the same size.</p> <p>When the images are different sizes, use an <b>AugmentedImageDatastore</b> object.</p> <p><b>ImageDatastore</b> objects support image classification tasks only. To use image datastores for regression neural networks, create a transformed or combined datastore that contains the images and responses using the <b>transform</b> and <b>combine</b> functions, respectively.</p>
	AugmentedImageDatastore	Datastore that applies random affine geometric transformations, including resizing, rotation, reflection, shear, and translation.	<ul style="list-style-type: none"> <li>• Train image classification neural network with images saved on disk, where the images are different sizes.</li> <li>• Train image classification neural network and generate new data using augmentations.</li> </ul>

Data Type		Description	Example Usage
	TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function.	<ul style="list-style-type: none"> <li>• Train image regression neural network.</li> <li>• Train neural networks with multiple inputs.</li> <li>• Transform datastores with outputs not supported by <code>trainNetwork</code>.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
	CombinedDatastore	Datastore that reads from two or more underlying datastores.	<ul style="list-style-type: none"> <li>• Train image regression neural network.</li> <li>• Train neural networks with multiple inputs.</li> <li>• Combine predictors and responses from different data sources.</li> </ul>
	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 <code>responses</code> 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 <code>responses</code> argument.	Train neural network using data stored in a table.

For neural 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 details, see “Develop 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 the `imageDatastore` function for preprocessing or resizing, as this option is usually significantly slower.

You can use other built-in datastores for training deep learning neural networks by using the `transform` and `combine` functions. These functions can convert the data read from datastores to the format required by `trainNetwork`.

For neural networks with multiple inputs, the datastore must be a `TransformedDatastore` or `CombinedDatastore` object.

The required format of the datastore output depends on the neural network architecture.

Neural 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 neural network with one input and one output:  <code>data = read(ds)</code>  <code>data =</code>  4×2 table  <table style="margin-left: 40px; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center; border-bottom: 1px solid black;">Predictors</th> <th style="text-align: center; border-bottom: 1px solid black;">Response</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">2</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">7</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">9</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">9</td> </tr> </tbody> </table>	Predictors	Response	{224×224×3 double}	2	{224×224×3 double}	7	{224×224×3 double}	9	{224×224×3 double}	9
	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 neural network with one input and one output:  <code>data = read(ds)</code>  <code>data =</code>  4×2 cell array  <table style="margin-left: 40px; border-collapse: collapse;"> <tbody> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">{[2]}</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">{[7]}</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">{[9]}</td> </tr> <tr> <td style="text-align: center;">{224×224×3 double}</td> <td style="text-align: center;">{[9]}</td> </tr> </tbody> </table>	{224×224×3 double}	{[2]}	{224×224×3 double}	{[7]}	{224×224×3 double}	{[9]}	{224×224×3 double}	{[9]}			
{224×224×3 double}	{[2]}											
{224×224×3 double}	{[7]}											
{224×224×3 double}	{[9]}											
{224×224×3 double}	{[9]}											

Neural Network Architecture	Datastore Output	Example Output
Multiple input layers	<p>Cell array with (<code>numInputs</code> + 1) columns, where <code>numInputs</code> is the number of neural network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input and the last column specifies the responses.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the layer graph <code>layers</code>.</p>	<p>Cell array for neural network with two inputs and one output.</p> <pre>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</pre>

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

Data	Format
2-D images	<i>h-by-w-by-c</i> numeric array, where <i>h</i> , <i>w</i> , and <i>c</i> are the height, width, and number of channels of the images, respectively.
3-D images	<i>h-by-w-by-d-by-c</i> numeric array, where <i>h</i> , <i>w</i> , <i>d</i> , and <i>c</i> 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	<ul style="list-style-type: none"> <li>Numeric scalar</li> <li>Numeric vector</li> <li>3-D numeric array representing a 2-D image</li> <li>4-D numeric array representing a 3-D image</li> </ul>

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	<ul style="list-style-type: none"> <li>• Numeric scalar</li> <li>• Two or more columns of scalar values</li> <li>• 1-by-1 cell array containing a <math>h</math>-by-<math>w</math>-by-<math>c</math> numeric array representing a 2-D image</li> <li>• 1-by-1 cell array containing a <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math> numeric array representing a 3-D image</li> </ul>

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 NaNs, then they are propagated through the neural network during training. In these cases, the training usually fails to converge.
- 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”.
- To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | table

Complex Number Support: Yes

### 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.	<ul style="list-style-type: none"> <li>Transform datastores with outputs not supported by <code>trainNetwork</code>.</li> <li>Apply custom transformations to datastore output.</li> </ul>
	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.	<p>Train neural network using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>
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. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for training deep learning neural 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	<pre>data = read(ds)  data =      4x2 table        Predictors      Response       _____      _____     {12x50 double}         2     {12x50 double}         7     {12x50 double}         9     {12x50 double}         9</pre>
Cell array	<pre>data = read(ds)  data =      4x2 cell array      {12x50 double}    {[2]}     {12x50 double}    {[7]}     {12x50 double}    {[9]}     {12x50 double}    {[9]}</pre>

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

Data	Format of Predictors
Vector sequence	<i>c</i> -by- <i>s</i> matrix, where <i>c</i> is the number of features of the sequence and <i>s</i> is the sequence length.
1-D image sequence	<p><i>h</i>-by-<i>c</i>-by-<i>s</i> array, where <i>h</i> and <i>c</i> correspond to the height and number of channels of the image, respectively, and <i>s</i> is the sequence length.</p> <p>Each sequence in the mini-batch must have the same sequence length.</p>

<b>Data</b>	<b>Format of Predictors</b>
2-D image sequence	<p><math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>s</math> array, where <math>h</math>, <math>w</math>, and <math>c</math> correspond to the height, width, and number of channels of the image, respectively, and <math>s</math> is the sequence length.</p> <p>Each sequence in the mini-batch must have the same sequence length.</p>
3-D image sequence	<p><math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math>-by-<math>s</math> array, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> correspond to the height, width, depth, and number of channels of the image, respectively, and <math>s</math> is the sequence length.</p> <p>Each sequence in the mini-batch must have the same sequence length.</p>

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.

<b>Task</b>	<b>Format of Responses</b>
Sequence-to-label classification	Categorical scalar
Sequence-to-one regression	Scalar
Sequence-to-vector regression	Numeric row vector
Sequence-to-sequence classification	<ul style="list-style-type: none"> <li>1-by-<math>s</math> sequence of categorical labels, where <math>s</math> is the sequence length of the corresponding predictor sequence.</li> <li><math>h</math>-by-<math>w</math>-by-<math>s</math> sequence of categorical labels, where <math>h</math>, <math>w</math>, and <math>s</math> are the height, width, and sequence length of the corresponding predictor sequence, respectively.</li> <li><math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>s</math> sequence of categorical labels, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>s</math> are the height, width, depth, and sequence length of the corresponding predictor sequence, respectively.</li> </ul> <p>Each sequence in the mini-batch must have the same sequence length.</p>

Task	Format of Responses
Sequence-to-sequence regression	<ul style="list-style-type: none"> <li>• <math>R</math>-by-<math>s</math> matrix, where <math>R</math> is the number of responses and <math>s</math> is the sequence length of the corresponding predictor sequence.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>R</math>-by-<math>s</math> sequence of numeric responses, where <math>R</math> is the number of responses, and <math>h</math>, <math>w</math>, and <math>s</math> are the height, width, and sequence length of the corresponding predictor sequence, respectively.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>R</math>-by-<math>s</math> sequence of numeric responses, where <math>R</math> is the number of responses, and <math>h</math>, <math>w</math>, <math>d</math>, and <math>s</math> are the height, width, depth, and sequence length of the corresponding predictor sequence, respectively.</li> </ul> <p>Each sequence in the mini-batch must have the same sequence length.</p>

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.

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.



Input	Description
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.

The `trainNetwork` function supports neural networks with at most one sequence input layer.

### Tip

- If the predictors or the responses contains NaNs, then they are propagated through the neural network during training. In these cases, the training usually fails to converge.
- For regression tasks, normalizing the responses often helps to stabilize and speed up training. For more information, see “Train Convolutional Neural Network for Regression”.
- To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Complex Number Support: Yes

### features – Feature data

`datastore` | numeric array | table

Feature data, specified as one of the following:

Data Type	Description	Example Usage
Datastore	<code>TransformedDatastore</code>	<ul style="list-style-type: none"> <li>• Train neural networks with multiple inputs.</li> <li>• Transform datastores with outputs not supported by <code>trainNetwork</code>.</li> <li>• Apply custom transformations to datastore output.</li> </ul>
	<code>CombinedDatastore</code>	<ul style="list-style-type: none"> <li>• Train neural networks with multiple inputs.</li> <li>• Combine predictors and responses from different data sources.</li> </ul>

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”.
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 <code>responses</code> 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 <code>responses</code> 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. For details, see “Develop Custom Mini-Batch Datastore”.

You can use other built-in datastores for training deep learning neural 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 neural 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 neural network architecture.

Neural Network Architecture	Datastore Output	Example Output																		
Single input layer	<p>Table or cell array with two columns.</p> <p>The first and second columns specify the predictors and responses, respectively.</p> <p>Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.</p> <p>Custom mini-batch datastores must output tables.</p>	<p>Table for neural network with one input and one output:</p> <pre>data = read(ds) data =</pre> <pre>4x2 table</pre> <table border="1"> <thead> <tr> <th>Predictors</th> <th>Response</th> </tr> </thead> <tbody> <tr> <td>{24x1 double}</td> <td>2</td> </tr> <tr> <td>{24x1 double}</td> <td>7</td> </tr> <tr> <td>{24x1 double}</td> <td>9</td> </tr> <tr> <td>{24x1 double}</td> <td>9</td> </tr> </tbody> </table> <p>Cell array for neural network with one input and one output:</p> <pre>data = read(ds) data =</pre> <pre>4x2 cell array</pre> <table border="1"> <tbody> <tr> <td>{24x1 double}</td> <td>{[2]}</td> </tr> <tr> <td>{24x1 double}</td> <td>{[7]}</td> </tr> <tr> <td>{24x1 double}</td> <td>{[9]}</td> </tr> <tr> <td>{24x1 double}</td> <td>{[9]}</td> </tr> </tbody> </table>	Predictors	Response	{24x1 double}	2	{24x1 double}	7	{24x1 double}	9	{24x1 double}	9	{24x1 double}	{[2]}	{24x1 double}	{[7]}	{24x1 double}	{[9]}	{24x1 double}	{[9]}
Predictors	Response																			
{24x1 double}	2																			
{24x1 double}	7																			
{24x1 double}	9																			
{24x1 double}	9																			
{24x1 double}	{[2]}																			
{24x1 double}	{[7]}																			
{24x1 double}	{[9]}																			
{24x1 double}	{[9]}																			
Multiple input layers	<p>Cell array with (numInputs + 1) columns, where numInputs is the number of neural network inputs.</p> <p>The first numInputs columns specify the predictors for each input and the last column specifies the responses.</p> <p>The order of inputs is given by the InputNames property of the layer graph layers.</p>	<p>Cell array for neural network with two inputs and one output:</p> <pre>data = read(ds) data =</pre> <pre>4x3 cell array</pre> <table border="1"> <tbody> <tr> <td>{24x1 double}</td> <td>{28x1 double}</td> <td>{}</td> </tr> <tr> <td>{24x1 double}</td> <td>{28x1 double}</td> <td>{}</td> </tr> <tr> <td>{24x1 double}</td> <td>{28x1 double}</td> <td>{}</td> </tr> <tr> <td>{24x1 double}</td> <td>{28x1 double}</td> <td>{}</td> </tr> </tbody> </table>	{24x1 double}	{28x1 double}	{}	{24x1 double}	{28x1 double}	{}	{24x1 double}	{28x1 double}	{}	{24x1 double}	{28x1 double}	{}						
{24x1 double}	{28x1 double}	{}																		
{24x1 double}	{28x1 double}	{}																		
{24x1 double}	{28x1 double}	{}																		
{24x1 double}	{28x1 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	<ul style="list-style-type: none"> <li>Scalar</li> <li>Numeric vector</li> </ul>

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 <code>responses</code> argument, then the predictors must be in the first <code>numFeatures</code> columns of the table, where <code>numFeatures</code> is the number of features of the input data.	Categorical label
Feature regression		One or more columns of scalar values

For classification neural 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 neural 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”.
- Responses must not contain NaNs. If the predictor data contains NaNs, then they are propagated through the training. However, in most cases, the training fails to converge.
- To input complex-valued data into a neural network, the `SplitComplexInputs` option of the input layer must be 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `table`

Complex Number Support: Yes

### **mixed** — Mixed data

datastore

Mixed data and responses, specified as one of the following:

Data Type	Description	Example Usage
TransformedDatastore	Datastore that transforms batches of data read from an underlying datastore using a custom transformation function.	<ul style="list-style-type: none"> <li>Train neural networks with multiple inputs.</li> <li>Transform outputs of datastores not supported by <code>trainNetwork</code> to have the required format.</li> <li>Apply custom transformations to datastore output.</li> </ul>
CombinedDatastore	Datastore that reads from two or more underlying datastores.	<ul style="list-style-type: none"> <li>Train neural networks with multiple inputs.</li> <li>Combine predictors and responses from different data sources.</li> </ul>
Custom mini-batch datastore	Custom datastore that returns mini-batches of data.	<p>Train neural network using data in a format that other datastores do not support.</p> <p>For details, see “Develop Custom Mini-Batch Datastore”.</p>

You can use other built-in datastores for training deep learning neural 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”.

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 neural network architecture.

Datastore Output	Example Output
<p>Cell array with <math>(\text{numInputs} + 1)</math> columns, where <code>numInputs</code> is the number of neural network inputs.</p> <p>The first <code>numInputs</code> columns specify the predictors for each input and the last column specifies the responses.</p> <p>The order of inputs is given by the <code>InputNames</code> property of the layer graph <code>layers</code>.</p>	<pre>data = read(ds) data =     4x3 cell array     {24x1 double}    {28x1 double}    {[2]}     {24x1 double}    {28x1 double}    {[2]}     {24x1 double}    {28x1 double}    {[9]}     {24x1 double}    {28x1 double}    {[9]}</pre>

For image, sequence, and feature predictor input, the format of the predictors must match the formats described in the `images`, `sequences`, or `features` argument descriptions, respectively.

Similarly, the format of the responses must match the formats described in the `images`, `sequences`, or `features` argument descriptions that corresponds to the type of task.

The `trainNetwork` function supports neural networks with at most one sequence input layer.

For an example showing how to train a neural network with multiple inputs, see “Train Network on Image and Feature Data”.

---

**Tip**

- To convert a numeric array to a datastore, use `ArrayDatastore`.
  - When combining layers in a neural network with mixed types of data, you may need to reformat the data before passing it to a combination layer (such as a concatenation or an addition layer). To reformat the data, you can use a `flatten` layer to flatten the spatial dimensions into the channel dimension, or create a `FunctionLayer` object or custom layer that reformats and reshapes.
- 

**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 or 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	<i>N</i> -by-1 categorical vector of labels, where <i>N</i> is the number of observations.
	Feature classification	
	Sequence-to-label classification	

Task	Format	
	Sequence-to-sequence classification	<p><math>N</math>-by-1 cell array of categorical sequences of labels, where <math>N</math> is the number of observations. Each sequence must have the same number of time steps as the corresponding predictor sequence.</p> <p>For sequence-to-sequence classification tasks with one observation, sequences can also be a vector. In this case, responses must be a categorical row-vector of labels.</p>
Regression	2-D image regression	<ul style="list-style-type: none"> <li>• <math>N</math>-by-<math>R</math> matrix, where <math>N</math> is the number of images and <math>R</math> is the number of responses.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>c</math>-by-<math>N</math> numeric array, where <math>h</math>, <math>w</math>, and <math>c</math> are the height, width, and number of channels of the images, respectively, and <math>N</math> is the number of images.</li> </ul>
	3-D image regression	<ul style="list-style-type: none"> <li>• <math>N</math>-by-<math>R</math> matrix, where <math>N</math> is the number of images and <math>R</math> is the number of responses.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>c</math>-by-<math>N</math> numeric array, where <math>h</math>, <math>w</math>, <math>d</math>, and <math>c</math> are the height, width, depth, and number of channels of the images, respectively, and <math>N</math> is the number of images.</li> </ul>
	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	<p><math>N</math>-by-1 cell array of numeric sequences, where <math>N</math> is the number of sequences, with sequences given by one of the following:</p> <ul style="list-style-type: none"> <li>• <math>R</math>-by-<math>s</math> matrix, where <math>R</math> is the number of responses and <math>s</math> is the sequence length of the corresponding predictor sequence.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>R</math>-by-<math>s</math> array, where <math>h</math> and <math>w</math> are the height and width of the output, respectively, <math>R</math> is the number of responses, and <math>s</math> is the sequence length of the corresponding predictor sequence.</li> <li>• <math>h</math>-by-<math>w</math>-by-<math>d</math>-by-<math>R</math>-by-<math>s</math> array, where <math>h</math>, <math>w</math>, and <math>d</math> are the height, width, and depth of the output, respectively, <math>R</math> is the number of responses, and <math>s</math> is the sequence length of the corresponding predictor sequence.</li> </ul> <p>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.</p>

---

**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 NaNs. If the predictor data contains NaNs, then they are propagated through the training. However, in most cases, the training fails to converge.

---

### Layers — Neural network layers

Layer array | LayerGraph object

Neural network layers, specified as a Layer array or a LayerGraph object.

To create a neural network with all layers connected sequentially, you can use a Layer array as the input argument. In this case, the returned neural network is a SeriesNetwork object.



A directed acyclic graph (DAG) neural network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG neural network, specify the neural network architecture as a `LayerGraph` object and then use that layer graph as the input argument to `trainNetwork`.

The `trainNetwork` function supports neural networks with at most one sequence input layer.

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 neural network

`SeriesNetwork` object | `DAGNetwork` object

Trained neural network, returned as a `SeriesNetwork` object or a `DAGNetwork` object.

If you train the neural network using a `Layer` array, then `net` is a `SeriesNetwork` object. If you train the neural 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 neural network
- `FinalValidationAccuracy` — Validation accuracy of returned neural network
- `OutputNetworkIteration` — Iteration number of returned neural 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 neural network
- `FinalValidationRMSE` — Validation RMSE of returned neural network
- `OutputNetworkIteration` — Iteration number of returned neural 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 neural 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 neural network perform different operations than during training. For more information, see `batchNormalizationLayer`.

## More About

### Save Checkpoint Neural Networks and Resume Training

Deep Learning Toolbox enables you to save neural networks as `.mat` files during training. This periodic saving is especially useful when you have a large neural 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 neural network. If you want `trainNetwork` to save checkpoint neural networks, then you must specify the name of the path by using the `CheckpointPath` option of `trainingOptions`. If the path that you specify does not exist, then `trainingOptions` returns an error.

`trainNetwork` automatically assigns unique names to checkpoint neural 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 neural network. You can load a checkpoint neural 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 neural network as an input argument to `trainNetwork`. For example:

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

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

### Floating-Point Arithmetic

When you train a neural 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 neural networks using both CPUs and GPUs.

## Reproducibility

To provide the best performance, deep learning using a GPU in MATLAB is not guaranteed to be deterministic. Depending on your network architecture, under some conditions you might get different results when using a GPU to train two identical networks or make two predictions using the same network and data.

## Version History

### Introduced in R2016a

#### **R2022b: trainNetwork pads mini-batches to length of longest sequence before splitting when you specify SequenceLength training option as an integer**

Starting in R2022b, when you train a neural network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the neural network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

#### **R2021a: 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 neural 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.

#### **R2021b: trainNetwork automatically stops training when loss is NaN**

*Behavior changed in R2021b*

When you train a neural network using the `trainNetwork` function, training automatically stops when the loss is NaN. Usually, a loss value of NaN introduces NaN values to the neural network learnable parameters, which in turn can cause the neural network to fail to train or to make valid predictions. This change helps identify issues with the neural network before training completes.

In previous releases, the neural 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 neural network on a GPU by using `trainingOptions` to set the 'ExecutionEnvironment' to "auto" or "gpu" and supply the options to `trainNetwork`.
- The `ExecutionEnvironment` option must be "auto" or "gpu" when the input data is:
  - A `gpuArray`
  - A cell array containing `gpuArray` objects
  - A table containing `gpuArray` objects
  - A datastore that outputs cell arrays containing `gpuArray` objects
  - A datastore that outputs tables containing `gpuArray` objects

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 Neural Network for Classification”

“Transfer Learning Using Pretrained Network”

“Train Convolutional Neural Network for Regression”

“Sequence Classification Using Deep Learning”

“Train Network on Image and Feature Data”

“Deep Learning in MATLAB”

“Define Custom Deep Learning Layers”

“List of Deep Learning Layers”

# transposedConv1dLayer

Transposed 1-D convolution layer

## Syntax

```
layer = transposedConv1dLayer(filterSize,numFilters)
layer = transposedConv1dLayer(filterSize,numFilters,Name=Value)
```

## Description

A transposed 1-D convolution layer upsamples one-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 = transposedConv1dLayer(filterSize,numFilters)` returns a 1-D transposed convolution layer and sets the `FilterSize` and `NumFilters` properties.

`layer = transposedConv1dLayer(filterSize,numFilters,Name=Value)` returns a 1-D transposed convolutional layer and specifies additional options using one or more name-value arguments.

## Examples

### Create 1-D Transposed Convolutional Layer

Create a 1-D transposed convolutional layer with 96 filters of length 11 and a stride of 4.

```
layer = transposedConv1dLayer(11,96,Stride=4)
```

```
layer =
  TransposedConvolution1dLayer with properties:
```

```
    Name: ''
```

```
Hyperparameters
```

```
  FilterSize: 11
  NumChannels: 'auto'
  NumFilters: 96
  Stride: 4
  CroppingMode: 'manual'
  CroppingSize: [0 0]
```

```
Learnable Parameters
```

```
  Weights: []
  Bias: []
```

```
Show all properties
```

## Input Arguments

### **filterSize** — Length of filters

positive integer

Length of the filters, specified as a positive integer. The filter size defines the size of the local regions to which the neurons connect in the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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 layer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **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: `transposedConv1dLayer(11,96,Stride=4)` creates a 1-D transposed convolutional layer with 96 filters of length 11 and a stride of 4.

### **Transposed Convolution**

#### **Stride** — Upsampling factor

1 (default) | positive integer

Upsampling factor of the input, specified as a positive integer that corresponds to the horizontal stride.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **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 length of the layer input. If `Cropping` is "same", then the software automatically sets the `CroppingMode` property of the layer to 'same'.

The software trims an equal amount from the left and right, when possible. 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 the left and right edges.
- A vector of nonnegative integers [`l` `r`] — Crop `l` and `r` from the left and right, respectively.

If you set the `Cropping` option to a numeric value, then the software automatically sets the `CroppingMode` property of the layer to 'manual'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

**NumChannels — Number of input channels**

"auto" (default) | positive integer

Number of input channels, specified as one of the following:

- "auto" — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. NumChannels and the number of channels in the layer input data must match. For example, if the input is an RGB image, then NumChannels must be 3. If the input is the output of a convolutional layer with 16 filters, then NumChannels must be 16.

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  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize} * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with a mean of zero and a variance of  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$ .
- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

### **Weights — Layer weights**

`[]` (default) | numeric array

Layer weights for the transposed convolution operation, specified as a `FilterSize-by-NumFilters-by-NumChannels` numeric array or `[]`.

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 transposed convolutional operation, specified as a `1-by-NumFilters` numeric array or `[]`.

The layer biases are learnable parameters. When you train a neural 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`.

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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

Data Types: `char` | `string`

## **Output Arguments**

### **layer — Transposed 1-D convolution layer**

`TransposedConvolution1DLayer` object

Transposed 1-D convolution layer, returned as a `TransposedConvolution1DLayer` object.

## **Algorithms**

### **1-D Transposed Convolutional Layer**

A transposed 1-D convolution layer upsamples one-dimensional feature maps.

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

A 1-D transposed convolution layer upsamples a single dimension only. The dimension that the layer upsamples 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 upsamples the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer upsamples 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 upsamples the spatial dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)

In `dlnetwork` objects, `TransposedConvolution1DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)

## Version History

Introduced in R2022a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## See Also

`trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `biLstmLayer` | `gruLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer` | `convolution1dLayer`

## Topics

"Time Series Anomaly Detection Using Deep Learning"  
 "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 Neural Networks"  
 "List of Deep Learning Layers"  
 "Deep Learning Tips and Tricks"

# 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 two-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer performs the transpose of convolution and does not perform deconvolution.

`layer = transposedConv2dLayer(filterSize,numFilters)` returns a 2-D transposed convolution layer and sets the `FilterSize` and `NumFilters` properties.

`layer = transposedConv2dLayer(filterSize,numFilters,Name,Value)` returns a 2-D transposed 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

positive integer | vector of two positive integers

Height and width of the filters, specified as a positive integer or a vector of two positive integers [h w], where h is the height and w is the width. The filter size defines the size of the local regions to which the neurons connect in the input.

If `filterSize` is a positive integer, then the software uses the same value for both dimensions.

Example: [5 6] specifies filters of height and width of 5 and 6, respectively.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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 layer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `transposedConv2dLayer(11,96,'Stride',4)` creates a 2-D transposed convolutional layer with 96 filters of size 11 and a stride of 4.

### 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 that corresponds to both the vertical and horizontal stride.

Example: `'Stride',[2 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### 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]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

### **NumChannels — Number of input channels**

`"auto"` (default) | positive integer

Number of input channels, specified as one of the following:

- `"auto"` — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

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 Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumFilters}$ .
- `'he'` - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$ .
- `'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 biases**

`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the biases, specified as one of the following:

- `'zeros'` — Initialize the biases with zeros.
- `'ones'` — Initialize the biases with ones.
- `'narrow-normal'` — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.

- **Function handle** — Initialize the biases 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 biases.

The layer only initializes the biases when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

### **Weights — Layer weights**

`[]` (default) | numeric array

Layer weights for the transposed convolution operation, specified as a `filterSize(1)`-by-`filterSize(2)`-by-`numFilters`-by-`NumChannels` numeric array or `[]`.

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 transposed convolutional operation, specified as a 1-by-1-by-`numFilters` numeric array or `[]`.

The layer biases are learnable parameters. When you train a neural 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`.

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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

Data Types: `char` | `string`

## **Output Arguments**

### **layer — Transposed 2-D convolution layer**

`TransposedConvolution2DLayer` object

Transposed 2-D convolution layer, returned as a `TransposedConvolution2DLayer` object.

## **Algorithms**

### **2-D Transposed Convolutional Layer**

A transposed 2-D convolution layer upsamples two-dimensional feature maps.

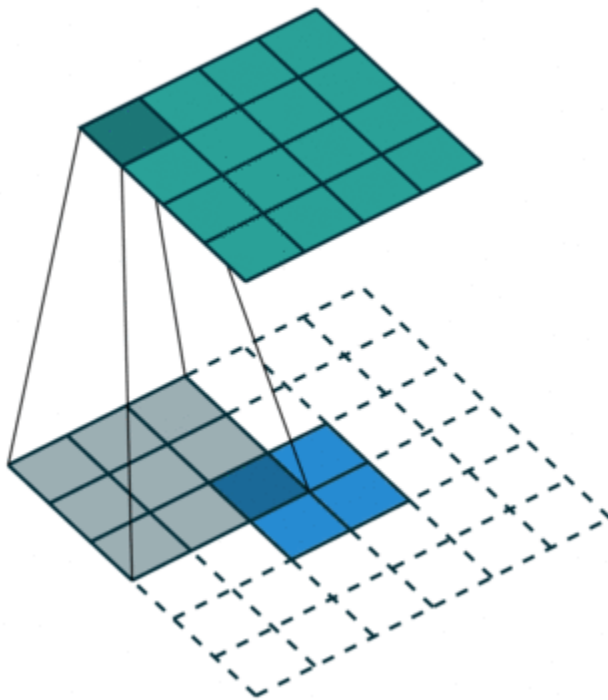
The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.



Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

This image shows a 3-by-3 filter upsampling 2-by-2 input. The lower map represents the input and the upper map represents the output. <sup>2</sup>



### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

<sup>2</sup> Image credit: Convolution arithmetic (License)

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)

In `dlnetwork` objects, `TransposedConvolution2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

### Introduced in R2017b

#### R2019a: 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## 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 Neural Network for Classification”

“Deep Learning in MATLAB”

“Compare Layer Weight Initializers”

“List of Deep Learning Layers”

# 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 performs the transpose of convolution and does not perform deconvolution.

`layer = transposedConv3dLayer(filterSize,numFilters)` returns a 3-D transposed convolution layer and sets the `FilterSize` and `NumFilters` properties.

`layer = transposedConv3dLayer(filterSize,numFilters,Name,Value)` returns a 3-D transposed 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**

positive integer | vector of three positive integers

Height, width, and depth of the filters, specified as a positive integer or a vector of three positive integers [h w d], where h is the height, w is the width, and d is the depth. The filter size defines the size of the local regions to which the neurons connect in the input.

If `filterSize` is a scalar, then the software uses the same value for all three dimensions.

Example: [5 6 7] specifies filters with a height, width, and depth of 5, 6, and 7 respectively.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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 layer.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `transposedConv3dLayer(11,96,'Stride',4)` creates a 3-D transposed convolutional layer with 96 filters of size 11 and a stride of 4.

### **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.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### **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.

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 2]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | string

### **NumChannels — Number of input channels**

"auto" (default) | positive integer

Number of input channels, specified as one of the following:

- "auto" — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. **NumChannels** and the number of channels in the layer input data must match. For example, if the input is an RGB image, then **NumChannels** must be 3. If the input is the output of a convolutional layer with 16 filters, then **NumChannels** must be 16.

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 Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$ .
- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

### **Weights — Layer weights**

[] (default) | numeric array

Layer weights for the transposed convolution operation, specified as a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`numFilters`-by-`NumChannels` numeric array or [].

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 transposed convolutional operation, specified as a 1-by-1-by-1-by-`numFilters` numeric array or [].

The layer biases are learnable parameters. When you train a neural 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`.

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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

Data Types: `char` | `string`



## Output Arguments

### Layer — Transposed 3-D convolution layer

TransposedConvolution3DLayer object

Transposed 3-D convolution layer, returned as a TransposedConvolution3dLayer object.

## Algorithms

### 3-D Transposed Convolutional Layer

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution3DLayer` objects 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 property set to 0 (false), then the layer receives an unformatted d`l`array object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In d`l`network objects, TransposedConvolution3DLayer objects also support these input and output format combinations.

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2019a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## 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"

# TransposedConvolution1DLayer

Transposed 1-D convolution layer

## Description

A transposed 1-D convolution layer upsamples one-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer performs the transpose of convolution and does not perform deconvolution.

## Creation

Create a transposed convolution 1-D output layer using `transposedConv1dLayer`.

## Properties

### Transposed Convolution

#### FilterSize — Length of filters

positive integer

Length of the filters, specified as a positive integer. The filter size defines the size of the local regions to which the neurons connect in the input.

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 layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the layer output.

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`

#### 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 length of the layer input.

To specify the cropping size, use the `Cropping` option of `transposedConv1dLayer`.

### **NumChannels — Number of input channels**

`'auto'` (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- `'auto'` — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

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  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize} * \text{NumFilters}$ .
- `'he'` - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with a mean of zero and a variance of  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize} * \text{NumChannels}$ .
- `'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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases when the `Bias` property is empty.

Data Types: char | string | function\_handle

**Weights — Layer weights**

[] (default) | numeric array

Layer weights for the transposed convolution operation, specified as a `FilterSize-by-NumFilters-by-NumChannels` numeric array or [].

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 transposed convolutional operation, specified as a `1-by-NumFilters` numeric array or [].

The layer biases are learnable parameters. When you train a neural 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`.

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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

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 1-D Transposed Convolutional Layer**

Create a 1-D transposed convolutional layer with 96 filters of length 11 and a stride of 4.

```
layer = transposedConv1dLayer(11,96,Stride=4)
```

```
layer =
```

```
  TransposedConvolution1DLayer with properties:
```

```
      Name: ''
```

```
  Hyperparameters
```

```
    FilterSize: 11
```

```
    NumChannels: 'auto'
```

```
    NumFilters: 96
```

```
    Stride: 4
```

```
    CroppingMode: 'manual'
```

```
    CroppingSize: [0 0]
```

```
  Learnable Parameters
```

```
    Weights: []
```

```
    Bias: []
```

```
  Show all properties
```

## Algorithms

### 1-D Transposed Convolutional Layer

A transposed 1-D convolution layer upsamples one-dimensional feature maps.

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

A 1-D transposed convolution layer upsamples a single dimension only. The dimension that the layer upsamples 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 upsamples the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations), the layer upsamples 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 upsamples the spatial dimension.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution1DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats 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)

In `dlnetwork` objects, `TransposedConvolution1DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SC" (spatial, channel)	"SC" (spatial, channel)
"CT" (channel, time)	"CT" (channel, time)
"SCT" (spatial, channel, time)	"SCT" (spatial, channel, time)

## Version History

Introduced in R2022a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## See Also

`transposedConv1dLayer` | `trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `lstmLayer` | `biLstmLayer` | `gruLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer` | `convolution1dLayer`

## Topics

"Time Series Anomaly Detection Using Deep Learning"  
 "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 Neural Networks"

“List of Deep Learning Layers”  
“Deep Learning Tips and Tricks”

# TransposedConvolution2DLayer

Transposed 2-D convolution layer

## Description

A transposed 2-D convolution layer upsamples two-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 2-D 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.

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 layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the layer output.

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.

### **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 input channels**

`'auto'` (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- `'auto'` — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

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 Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumFilters}$ .
- `'he'` - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{NumChannels}$ .
- `'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 biases

`'zeros'` (default) | `'narrow-normal'` | `'ones'` | function handle

Function to initialize the biases, specified as one of the following:

- `'zeros'` — Initialize the biases with zeros.
- `'ones'` — Initialize the biases with ones.
- `'narrow-normal'` — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name ' '.

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);
```

## Algorithms

### 2-D Transposed Convolutional Layer

A transposed 2-D convolution layer upsamples two-dimensional feature maps.

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

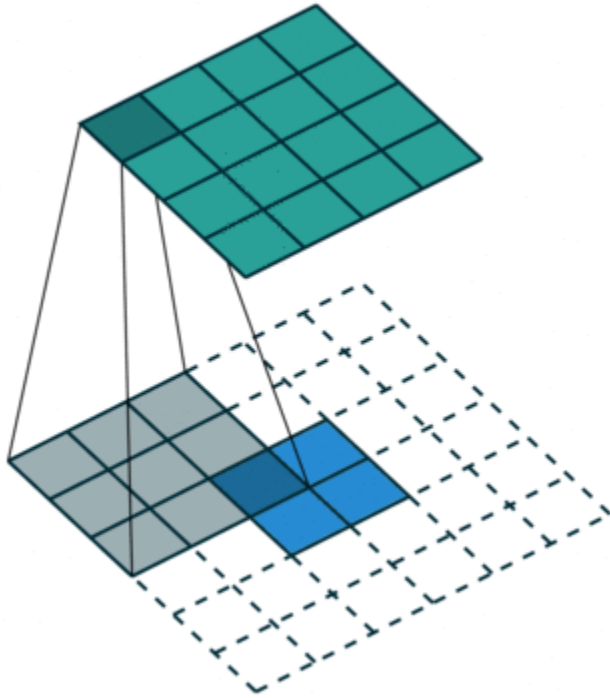
By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

This image shows a 3-by-3 filter upsampling 2-by-2 input. The lower map represents the input and the upper map represents the output.<sup>3</sup>

---

<sup>3</sup> Image credit: Convolution arithmetic (License)





### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The format consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution2DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSCB" (spatial, spatial, channel, batch)	"SSCB" (spatial, spatial, channel, batch)
"SSC" (spatial, spatial, channel)	"SSC" (spatial, spatial, channel)

In `dlnetwork` objects, `TransposedConvolution2DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSCBT" (spatial, spatial, channel, batch, time)	"SSCBT" (spatial, spatial, channel, batch, time)
"SSCT" (spatial, spatial, channel, time)	"SSCT" (spatial, spatial, channel, time)

## Version History

### Introduced in R2017b

#### R2019a: 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'.

#### R2019a: 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

## See Also

`averagePooling2dLayer` | `transposedConv2dLayer` | `maxPooling2dLayer` | `convolution2dLayer`

## Topics

"Create Simple Deep Learning Neural Network for Classification"

“Deep Learning in MATLAB”  
“Specify Layers of Convolutional Neural Network”  
“List of Deep Learning Layers”  
“Compare Layer Weight Initializers”

# 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 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

This property is read-only.

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 layer output.

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.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **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 input channels**

'auto' (default) | positive integer

This property is read-only.

Number of input channels, specified as one of the following:

- 'auto' — Automatically determine the number of input channels at training time.
- Positive integer — Configure the layer for the specified number of input channels. `NumChannels` and the number of channels in the layer input data must match. For example, if the input is an RGB image, then `NumChannels` must be 3. If the input is the output of a convolutional layer with 16 filters, then `NumChannels` must be 16.

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 Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance  $2/(\text{numIn} + \text{numOut})$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$  and  $\text{numOut} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumFilters}$ .
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance  $2/\text{numIn}$ , where  $\text{numIn} = \text{FilterSize}(1) * \text{FilterSize}(2) * \text{FilterSize}(3) * \text{NumChannels}$ .
- '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 biases**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the biases, specified as one of the following:

- 'zeros' — Initialize the biases with zeros.
- 'ones' — Initialize the biases with ones.
- 'narrow-normal' — Initialize the biases by independently sampling from a normal distribution with a mean of zero and a standard deviation of 0.01.
- Function handle — Initialize the biases 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 biases.

The layer only initializes the biases 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 neural 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. The software determines the global  $L_2$  regularization factor based on the settings you specify 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 the name `''`.

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
```

## Algorithms

### 3-D Transposed Convolutional Layer

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

The *standard* convolution operation *downsamples* the input by applying sliding convolutional filters to the input. By flattening the input and output, you can express the convolution operation as  $Y = CX + B$  for the convolution matrix  $C$  and bias vector  $B$  that can be derived from the layer weights and biases.

Similarly, the *transposed* convolution operation *upsamples* the input by applying sliding convolutional filters to the input. To upsample the input instead of downsampling using sliding filters, the layer zero-pads each edge of the input with padding that has the size of the corresponding filter edge size minus 1.

By flattening the input and output, the transposed convolution operation is equivalent to  $Y = C^T X + B$ , where  $C$  and  $B$  denote the convolution matrix and bias vector for standard convolution derived from the layer weights and biases, respectively. This operation is equivalent to the backward function of a standard convolution layer.

### Layer Input and Output Formats

Layers in a layer array or layer graph pass data to subsequent layers as formatted `darray` objects. The format of a `darray` object is a string of characters, in which each character describes the corresponding dimension of the data. The formats consists of one or more of these characters:

- "S" — Spatial
- "C" — Channel
- "B" — Batch
- "T" — Time
- "U" — Unspecified

For example, 2-D image data represented as a 4-D array, where the first two dimensions correspond to the spatial dimensions of the images, the third dimension corresponds to the channels of the images, and the fourth dimension corresponds to the batch dimension, can be described as having the format "SSCB" (spatial, spatial, channel, batch).

You can interact with these `darray` objects in automatic differentiation workflows such as 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 `TransposedConvolution3DLayer` objects 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` property set to `0` (false), then the layer receives an unformatted `darray` object with dimensions ordered corresponding to the formats in this table.

Input Format	Output Format
"SSSCB" (spatial, spatial, spatial, channel, batch)	"SSSCB" (spatial, spatial, spatial, channel, batch)
"SSSC" (spatial, spatial, spatial, channel)	"SSSC" (spatial, spatial, spatial, channel)

In `dlnetwork` objects, `TransposedConvolution3DLayer` objects also support these input and output format combinations.

Input Format	Output Format
"SSSCBT" (spatial, spatial, spatial, channel, batch, time)	"SSSCBT" (spatial, spatial, spatial, channel, batch, time)
"SSSCT" (spatial, spatial, spatial, channel, time)	"SSSCT" (spatial, spatial, spatial, channel, time)

## Version History

Introduced in R2019a

## 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. <https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>
- [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. <https://doi.org/10.1109/ICCV.2015.123>

**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"](#)

## 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 1-1926.

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")
```

Function containing the imported ONNX network architecture was saved to the file `squeezenetFcn.m`. To learn how to use this function, type: `help squeezenetFcn`.

```
params =  
    ONNXParameters with properties:  
  
        Learnables: [1x1 struct]  
        Nonlearnables: [1x1 struct]  
        State: [1x1 struct]  
        NumDimensions: [1x1 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: [1x1x512x1000 dlarray]
```

```
conv10_B: [1000x1 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");
```

The network is ready for training. Specify the training options.

```
velocity = [];  
numEpochs = 5;  
miniBatchSize = 16;
```

```

initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;

```

Calculate the total number of iterations for the training progress monitor.

```

numObservations = size(YTrain,2);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
numIterations = numEpochs*numIterationsPerEpoch;

```

Initialize the `TrainingProgressMonitor` object. Because the timer starts when you create the monitor object, make sure that you create the object immediately after the training loop.

```

monitor = trainingProgressMonitor(Metrics="Loss",Info="Epoch",XLabel="Iteration");

```

Train the network.

```

epoch = 0;
iteration = 0;
executionEnvironment = "cpu"; % Change to "gpu" to train on a GPU.

% Loop over epochs.
while epoch < numEpochs && ~monitor.Stop

    epoch = epoch + 1;

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

    % Loop over mini-batches.
    i = 0;
    while i < numIterationsPerEpoch && ~monitor.Stop
        i = i + 1;
        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);

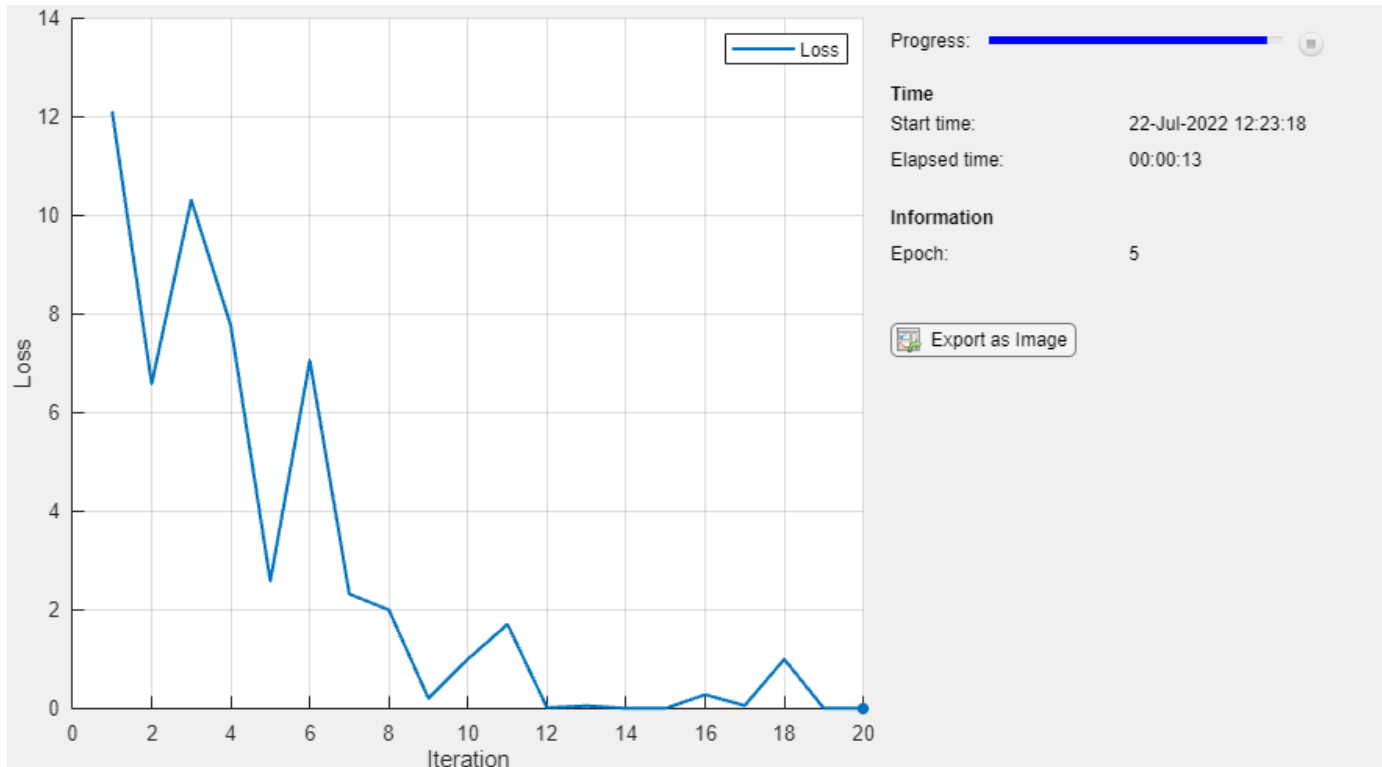
        % Update the network parameters using the SGDM optimizer.
        [params.Learnables,velocity] = sgdmupdate(params.Learnables,gradients,velocity);
    end
end

```

```

% Update the training progress monitor.
recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch);
monitor.Progress = 100 * iteration/numIterations;
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`.

## Version History

Introduced in R2020b

## See Also

`importONNXFunction` | `ONNXParameters` | `freezeParameters`

# TFLiteModel

TensorFlow Lite model

## Description

A `TFLiteModel` object enables support for simulation and code generation for deep learning inference by using TensorFlow Lite models

Use a `TFLiteModel` object with the `predict` function in your MATLAB code to perform inference in MATLAB execution, code generation, or MATLAB Function block in Simulink models. For more information, see “Prerequisites for Deep Learning with TensorFlow Lite Models”.

To use this object, you must install the Deep Learning Toolbox Interface for TensorFlow Lite support package.

## Creation

To create a `TFLiteModel` object from a pretrained TensorFlow Lite model file, use the `loadTFLiteModel` function.

## Properties

### **ModelName** — Name of the TensorFlow Lite model file

character vector

Name of the TensorFlow Lite model file, specified as a character vector.

### **NumInputs** — Number of inputs of the TensorFlow Lite model

numeric scalar

Number of inputs the TensorFlow Lite model accepts, specified as a integer-valued numeric scalar.

### **NumOutputs** — Number of outputs of the TensorFlow Lite model

numeric scalar

Number of outputs the TensorFlow Lite model produces, specified as a integer-valued numeric scalar.

### **InputSize** — Size of inputs of the TensorFlow model

cell array containing numeric arrays

Size of inputs of the TensorFlow model, specified as a cell array containing numeric arrays.

### **OutputSize** — Size of outputs of the TensorFlow model

cell array containing numeric arrays

Size of outputs of the TensorFlow model, specified as a cell array containing numeric arrays.

### **NumThreads** — Number of computational threads used for inference computation

numeric scalar

Number of computational threads used for running inference with the TensorFlow Lite model, specified as an integer-valued numeric scalar.

The default value of this property is equal to the value returned by the `maxNumCompThreads` function.

### Mean — Mean for input data normalization

127.5 (default) | double scalar

Mean value to which the input data is normalized, specified as a double scalar. If the input data is not normalized, you must set this property to 0. Otherwise, set this property based on how the input data is normalized.

### StandardDeviation — Standard deviation for input data normalization

127.5 (default) | double scalar

Standard deviation to which the input data is normalized, specified as a double scalar. If the input data is not normalized, you must set this property to 1. Otherwise, set this property based on how the input data is normalized.

## Object Functions

`predict` Compute deep learning network output for inference by using a TensorFlow Lite model

## Examples

### Perform Inference with TensorFlow Lite Model

Suppose that your current working directory contains a TensorFlow Lite Model named `mobilenet_v1_0.5_224.tflite`.

Load the model by using the `loadTFLite` function. Inspect the object this function creates.

```
net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
disp(net)
```

```
TFLiteModel with properties:
    ModelName: 'mobilenet_v1_0.5_224.tflite'
    NumInputs: 1
    NumOutputs: 1
    InputSize: {[224 224 3]}
    OutputSize: {[1001 1]}
    NumThreads: 8
            Mean: 127.5000
    StandardDeviation: 127.5000
```

Create a MATLAB function that can perform inference using the object `net`. This function loads the Mobilenet-V1 model into a persistent network object. Then the function performs prediction by passing the network object to the `predict` function. Subsequent calls to this function reuse this the persistent object.

```
function out = tflite_predict(in)
persistent net;
if isempty(net)
    net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
```

```
end  
out = predict(net,in);  
end
```

For an example that shows how to generate code for this function and deploy on Raspberry Pi hardware, see “Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”.

## Version History

Introduced in R2022a

### Extended Capabilities

#### C/C++ Code Generation

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

Usage notes and limitations:

- If code generation target is the host computer, the default value of the NumThreads property is equal to the value returned by the maxNumCompThreads function.
- If code generation target is a non-host hardware board, the default value of the NumThreads property is -1. To optimize performance, in your entry-point function, set this property based on the number of threads available on your hardware board.

### See Also

loadTFLiteModel | predict

#### Topics

“Deploy Pose Estimation Application Using TensorFlow Lite Model (TFLite) Model on Host and Raspberry Pi”

“Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”

“Deploy Super Resolution Application That Uses TensorFlow Lite (TFLite) Model on Host and Raspberry Pi”

“Prerequisites for Deep Learning with TensorFlow Lite Models”

# predict

Compute deep learning network output for inference by using a TensorFlow Lite model

## Syntax

```
Y = predict(net,X)
[Y1,...,YN] = predict(net,X)
___ = predict( ___,Name=Value)
```

## Description

`Y = predict(net,X)` returns the network output `Y` during inference given the input data `X` and the network `net` with a single input and a single output.

To use this function, you must install the Deep Learning Toolbox Interface for TensorFlow Lite support package.

`[Y1,...,YN] = predict(net,X)` returns the `N` outputs `Y1`, ..., `YN` during inference for networks that have `N` outputs.

`___ = predict( ___,Name=Value)` provides you options to control the data type (`int8/uint8` vs `single`) of inputs and outputs when `net` is a quantized model. There is an additional option to enable or disable the execution of `predict` on Windows platform for quantized models. All these name-value arguments are ignored if `net` is not a quantized model.

---

**Tip** For prediction with `SeriesNetwork` and `DAGNetwork` objects, see `predict`.

---

## Examples

### Perform Inference with TensorFlow Lite Model

Suppose that your current working directory contains a TensorFlow Lite Model named `mobilenet_v1_0.5_224.tflite`.

Load the model by using the `loadTFLite` function. Inspect the object this function creates.

```
net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
disp(net)
```

```
TFLiteModel with properties:
    ModelName: 'mobilenet_v1_0.5_224.tflite'
    NumInputs: 1
    NumOutputs: 1
    InputSize: {[224 224 3]}
    OutputSize: {[1001 1]}
    NumThreads: 8
             Mean: 127.5000
    StandardDeviation: 127.5000
```

Create a MATLAB function that can perform inference using the object `net`. This function loads the Mobilenet-V1 model into a persistent network object. Then the function performs prediction by passing the network object to the `predict` function. Subsequent calls to this function reuse this the persistent object.

```
function out = tflite_predict(in)
persistent net;
if isempty(net)
    net = loadTFLiteModel('mobilenet_v1_0.5_224.tflite');
end
out = predict(net,in);
end
```

For an example that shows how to generate code for this function and deploy on Raspberry Pi hardware, see “Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”.

## Input Arguments

### **net** — Object representing TensorFlow Lite model

TFLiteModel object

TFLiteModel object that represents the TensorFlow Lite model file.

### **X** — Input to the network

single | double | int8 | uint8

Image or sequence input to the network, specified as a numeric array.

- For image classification networks, the input must be of shape (H,W,C,N), where H is height, W is width, C is channel, and N is batch size.
- For recurrent neural networks, the input must be of shape (D, N, S), where D is channel or feature dimension, N is batch size, and S is timestamp or sequence length.

### 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `Y = predict(net,X,QuantizeInputs = true)`

### **QuantizeInputs** — Whether to convert inputs to a quantized model to int8/uint8 values

false (default) | true

Whether to convert inputs to a quantized model to int8/uint8 values before performing inference computation, specified as one of the values in this table.

Value	Description
false	This value is the default value.  The <code>predict</code> function directly uses the inputs that you provide.

Value	Description
true	The <code>predict</code> function quantizes inputs to the type expected by the <code>InputType</code> property of the model before performing inference computation.

This option only applies to quantized TFLite models and is ignored if the model is not quantized. For quantized TFLite models, you must set `QuantizeInputs` to `true` if input to the network is `single` or `double`.

For code generation workflows, you must specify this option as a constant.

Data Types: `logical`

### **DequantizeOutputs – Whether to convert outputs of a quantized model to single values**

`false` (default) | `true`

Whether to convert outputs of a quantized model to single data type before returning them, specified as one of the values in this table.

Value	Description
false	This value is the default value.  The <code>predict</code> function returns <code>int8/uint8</code> values.
true	The <code>predict</code> function converts outputs to single data type before returning them.

This option only applies to quantized TFLite models and is ignored if the model is not quantized.

For code generation workflows, you must specify this option as a constant.

Data Types: `logical`

### **EnableINT8InferenceOnWindows – Whether to allow predict to run on Windows platform for quantized models**

`false` (default) | `true`

Whether to allow `predict` to run on Windows platform for quantized models, specified as one of the values in this table.

Value	Description
false	This value is the default value.  Running the <code>predict</code> function with a quantized model produces an error on Windows platform. This default behavior allows you to detect existing issues with TFLite library v2.8.0 for quantized models on Windows platform.
true	Running the <code>predict</code> function with a quantized model on Windows platform bypasses the default error and executes to completion.

This option only applies to quantized TFLite models and is ignored if the model is not quantized. This option is also ignored on Linux platform or in embedded deployment workflows.

For code generation workflows, you must specify this option as a constant.

Data Types: `logical`

## Output Arguments

### Y — Output data

`single | int8 | uint8`

Output data, specified as a numeric array.

When performing inference with quantized TensorFlow Lite models, the output data is normalized in one of these ways:

- Signed 8-bit integer type outputs are normalized as  $\text{output}[i] = (\text{prediction}[i] + 128) / 256.0$ .
- Unsigned 8-bit integer type outputs are normalized as  $\text{output}[i] = \text{prediction}[i] / 255.0$ .

## Version History

Introduced in R2022a

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

For code generation workflows, you must specify `QuantizeInputs`, `DequantizeOutputs` and `EnableINT8InferenceOnWindows` options as constant.

## See Also

`loadTFLiteModel` | `TFLiteModel`

### Topics

“Deploy Pose Estimation Application Using TensorFlow Lite Model (TFLite) Model on Host and Raspberry Pi”

“Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi”

“Deploy Super Resolution Application That Uses TensorFlow Lite (TFLite) Model on Host and Raspberry Pi”

“Prerequisites for Deep Learning with TensorFlow Lite Models”



# updateInfo

**Package:** experiments

Update information columns in experiment results table

## Syntax

```
updateInfo(monitor, infoName=infoValue)
updateInfo(monitor, infoName1=infoValue1, ..., infoNameN=infoValueN)
updateInfo(monitor, infoStructure)
```

## Description

`updateInfo(monitor, infoName=infoValue)` updates the specified information column for a trial in the **Experiment Manager** results table.

`updateInfo(monitor, infoName1=infoValue1, ..., infoNameN=infoValueN)` updates multiple information columns for a trial.

`updateInfo(monitor, infoStructure)` updates the information columns using the values specified by the structure `infoStructure`.

## 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 = 100 * (iteration/numIterations);
```

### 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`

### **infoValue** — Information column value

numeric scalar | string | character vector | `dlarray`

Information column value, specified as a numeric scalar, string, character vector, or `dlarray` object.

### **infoStructure** — 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=squ  
aredGradientDecayFactor)
```

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.

## **Version History**

**Introduced in R2021a**

### **See Also**

#### **Apps**

**Experiment Manager**

#### **Objects**

`experiments.Monitor`

#### **Functions**

`groupSubPlot` | `recordMetrics` | `struct`

## updateInfo

**Package:** deep

Update information values for custom training loops

### Syntax

```
updateInfo(monitor,infoName=infoValue)
updateInfo(monitor,infoName1=infoValue1,...,infoNameN=infoValueN)
updateInfo(monitor,infoStructure)
```

### Description

`updateInfo(monitor,infoName=infoValue)` updates the specified information in the Training Progress window and saves the values in the `InfoData` property of the `TrainingProgressMonitor` object `monitor`.

`updateInfo(monitor,infoName1=infoValue1,...,infoNameN=infoValueN)` updates multiple information values.

`updateInfo(monitor,infoStructure)` updates the information using the values specified by the structure `infoStructure`.

### Examples

#### Track Progress and Produce Training Plots

Use a `TrainingProgressMonitor` object to track training progress and produce training plots for custom training loops.

Create a `TrainingProgressMonitor` object. The monitor automatically tracks the start time and the elapsed time. The timer starts when you create the object.

---

**Tip** To ensure that the elapsed time accurately reflects the training time, make sure you create the `TrainingProgressMonitor` object close to the start of your custom training loop.

---

```
monitor = trainingProgressMonitor;
```

Before you start the training, specify names for the information and metric values.

```
monitor.Info = ["LearningRate","Epoch","Iteration"];
monitor.Metrics = ["TrainingLoss","ValidationLoss","TrainingAccuracy","ValidationAccuracy"];
```

Specify the horizontal axis label for the training plot. Group the training and validation loss in the same subplot, and group the training and validation accuracy in the same plot.

```
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Loss",["TrainingLoss","ValidationLoss"]);
groupSubPlot(monitor,"Accuracy",["TrainingAccuracy","ValidationAccuracy"]);
```

During training:

- Evaluate the **Stop** property at the start of each step in your custom training loop. When you click the **Stop** button in the Training Progress window, the **Stop** property changes to 1. Training stops if your training loop exits when the **Stop** property is 1.
- Update the information values. The updated values appear in the Training Progress window.
- Record the metric values. The recorded values appear in the training plot.
- Update the training progress percentage based on the fraction of iterations completed.

---

**Note** The following example code is a template. You must edit this training loop to compute your metric and information values. For a complete example that you can run in MATLAB, see “Monitor Custom Training Loop Progress During Training”.

---

```
epoch = 0;
iteration = 0;

monitor.Status = "Running";

while epoch < maxEpochs && ~monitor.Stop
    epoch = epoch + 1;

    while hasData(mbq) && ~monitor.Stop
        iteration = iteration + 1;

        % Add code to calculate metric and information values.
        % lossTrain = ...

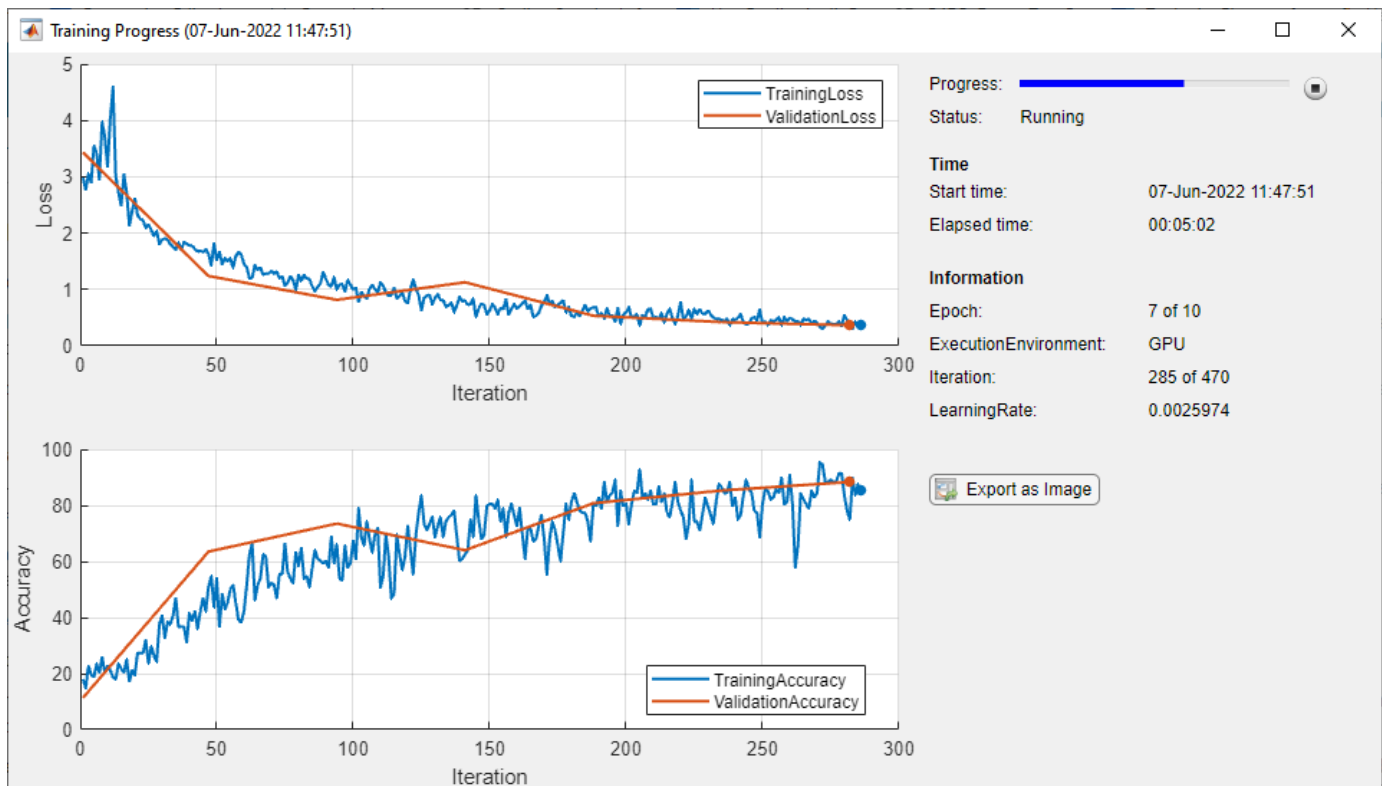
        updateInfo(monitor, ...
            LearningRate=learnRate, ...
            Epoch=string(epoch) + " of " + string(maxEpochs), ...
            Iteration=string(iteration) + " of " + string(numIterations));

        recordMetrics(monitor, iteration, ...
            TrainingLoss=lossTrain, ...
            TrainingAccuracy=accuracyTrain, ...
            ValidationLoss=lossValidation, ...
            ValidationAccuracy=accuracyValidation);

        monitor.Progress = 100*iteration/numIterations;
    end
end
```

The Training Progress window shows animated plots of the metrics, and the information values, training progress bar, and elapsed time.

- The training plots update each time you call `recordMetrics`.
- The values under **Information** update each time you call `updateInfo`.
- The elapsed time updates each time you call `recordMetrics` or `updateInfo`, and when you update the property.



### Specify Information Values by Using Structure

Use a structure to update the information values.

```
structure.GradientDecayFactor = gradientDecayFactor;
structure.SquaredGradientDecayFactor = squaredGradientDecayFactor;
updateInfo(monitor, structure);
```

### Input Arguments

#### monitor — Training progress monitor

TrainingProgressMonitor object

Training progress monitor, specified as a TrainingProgressMonitor object.

#### infoName — Information name

string scalar | character vector

Information name, specified as a string scalar or character vector. This name must be an element of the Info property of monitor.

Data Types: char | string | cell

#### infoValue — Information value

numeric scalar | string scalar | character vector | dIarray object

Information value, specified as a numeric scalar, string scalar, character vector, or dIarray object.

## **infoStructure — Information names and values**

structure

Information names and values, specified as a structure. Names must be elements of the `Info` property of `monitor` and can appear in any order in the structure.

Example:

```
struct(GradientDecayFactor=gradientDecayFactor,SquaredGradientDecayFactor=squaredGradientDecayFactor)
```

Data Types: `struct`

## **Tips**

- The information values appear in the Training Progress window and the training plot shows a record of the metric values. Use information values for text and for numerical values that you want to display in the training window but not in the training plot.

## **Version History**

**Introduced in R2022b**

## **See Also**

`groupSubPlot` | `trainingProgressMonitor` | `recordMetrics`

## **Topics**

“Monitor Custom Training Loop Progress”

“Train Network Using Custom Training Loop”

“Detect Issues During Deep Neural Network Training”

## updatePrunables

**Package:** `deep.prune`

Remove filters from prunable layers based on importance scores

### Syntax

```
prunableNet_new = updatePrunables(prunableNet)
prunableNet_new = updatePrunables(prunableNet,MaxToPrune=maxToPrune)
```

### Description

`prunableNet_new = updatePrunables(prunableNet)` removes up to 8 prunable filters from the convolution layers of `prunableNet` and returns an updated `TaylorPrunableNetwork` object. This function removes filters that have the lowest Taylor-based importance scores.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see [Deep Learning Toolbox Model Quantization Library](#).

`prunableNet_new = updatePrunables(prunableNet,MaxToPrune=maxToPrune)` removes up to `maxToPrune` prunable filters from the convolution layers of `prunableNet` and returns an updated `TaylorPrunableNetwork` object.

### Examples

#### Prune dlnetwork Object to Compress the Model

This example shows how to prune a `dlnetwork` object by using a custom pruning loop.

#### Load dlnetwork Object

Load a trained `dlnetwork` object and the corresponding classes.

```
s = load("digitsCustom.mat");
dlnet_1 = s.dlnet;
classes = s.classes;
```

Inspect the layers of the `dlnetwork` object. The network has three convolution layers at locations 2, 5, and 8 of the `Layer` array.

```
layers_1 = dlnet_1.Layers
```

```
layers_1 =
    12x1 Layer array with layers:
```

1	'input'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv1'	2-D Convolution	20 5x5x1 convolutions with stride [1 1] and padding
3	'bn1'	Batch Normalization	Batch normalization with 20 channels
4	'relu1'	ReLU	ReLU



5	'conv2'	2-D Convolution	20 3x3x20 convolutions with stride [1 1] and padding
6	'bn2'	Batch Normalization	Batch normalization with 20 channels
7	'relu2'	ReLU	ReLU
8	'conv3'	2-D Convolution	20 3x3x20 convolutions with stride [1 1] and padding
9	'bn3'	Batch Normalization	Batch normalization with 20 channels
10	'relu3'	ReLU	ReLU
11	'fc'	Fully Connected	10 fully connected layer
12	'softmax'	Softmax	softmax

### Load Data for Prediction

Load the digits data for prediction.

```
dataFolder = fullfile(toolboxdir("nnet"), "nndemos", "nndatasets", "DigitDataset");

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

Partition the data into pruning and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```
[imdsPrune, 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 images, use augmented image datastores.

```
inputSize = [28 28 1];
augimdsPrune = augmentedImageDatastore(inputSize(1:2), imdsPrune);
augimdsValidation = augmentedImageDatastore(inputSize(1:2), imdsValidation);
```

### Prune dlNetwork Object

Convert the `dlNetwork` object to a representation that is suitable for pruning by using the `taylorPrunableNetwork` function. This function returns a `TaylorPrunableNetwork` object that has the `NumPrunables` property set to 48. This indicates that 48 filters in the original model are suitable for pruning by using the Taylor pruning algorithm.

```
prunableNet_1 = taylorPrunableNetwork(dlnet_1)
```

```
prunableNet_1 =
    TaylorPrunableNetwork with properties:
```

```
    Learnables: [14x3 table]
    State: [6x3 table]
    InputNames: {'input'}
    OutputNames: {'softmax'}
    NumPrunables: 48
```

Create a `minibatchqueue` object that processes and manages mini-batches of images during pruning. 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 format 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 Computing Requirements” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(augimdsPrune, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    MiniBatchFormat=["SSCB" ""]);
```

Calculate Taylor-based importance scores of the prunable filters in the network by looping over the mini-batches of data. For each mini-batch:

- Calculate pruning activations and pruning gradients by using the `modelLoss` function defined at the end of this example
- Update importance scores of the prunable filters by using the `updateScore` function

```
while hasdata(mbq)
    [X,T] = next(mbq);
    [~,pruningActivations,pruningGradients] = dlfeval(@modelLoss,prunableNet_1,X,T);
    prunableNet_1 = updateScore(prunableNet_1,pruningActivations,pruningGradients);
end
```

Finally, remove filters with the lowest importance scores to create a new `TaylorPrunableNetwork` object by using the `updatePrunables` function. By default, a single call to this function removes 8 filters. Observe that the new network `prunableNet_2` has 40 prunable filters remaining.

```
prunableNet_2 = updatePrunables(prunableNet_1)

prunableNet_2 =
    TaylorPrunableNetwork with properties:

        Learnables: [14x3 table]
           State: [6x3 table]
    InputNames: {'input'}
    OutputNames: {'softmax'}
    NumPrunables: 40
```

To further compress the model, run the custom pruning loop and update prunables again.

### Extract Pruned `dlnetwork` Object

Use the `dlnetwork` function to extract the pruned `dlnetwork` object from the pruned `TaylorPrunableNetwork` object. You can now use this compressed `dlnetwork` object to perform inference.

```
dlnet_2 = dlnetwork(prunableNet_2);
```

Compare the convolution layers of the original and the pruned `dlnetwork` objects. Observe that the three convolution layers in the pruned network have fewer filters. These counts agree with the fact that, by default, a single call to the `updatePrunables` function removes 8 filters from the network.

```
conv_layers_1 = dlnet_1.Layers([2 5 8])
```

```

conv_layers_1 =
  3x1 Convolution2DLayer array with layers:

      1  'conv1'  2-D Convolution  20 5x5x1 convolutions with stride [1 1] and padding [0 0]
      2  'conv2'  2-D Convolution  20 3x3x20 convolutions with stride [1 1] and padding [1 1]
      3  'conv3'  2-D Convolution  20 3x3x20 convolutions with stride [1 1] and padding [1 1]

conv_layers_2 = dlnet_2.Layers([2 5 8])

conv_layers_2 =
  3x1 Convolution2DLayer array with layers:

      1  'conv1'  2-D Convolution  17 5x5x1 convolutions with stride [1 1] and padding [0 0]
      2  'conv2'  2-D Convolution  18 3x3x17 convolutions with stride [1 1] and padding [1 1]
      3  'conv3'  2-D Convolution  17 3x3x18 convolutions with stride [1 1] and padding [1 1]

```

## Supporting Functions

### Model Loss Function

The `modelLoss` function takes a `TaylorPrunableNetwork` object `net`, a mini-batch of input data `X` with corresponding targets `T` and returns activations in `net` and the gradients of the loss with respect to the activations in `net`. To compute the gradients automatically, this function uses the `dlgradient` function.

```

function [loss, pruningActivations, pruningGradients] = modelLoss(net,X,T)

% Calculate network output for training.
[out, ~, pruningActivations] = forward(net,X);

% Calculate loss.
loss = crossentropy(out,T);

% Compute pruning gradients.
pruningGradients = dlgradient(loss,pruningActivations);
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,T] = preprocessMiniBatch(dataX,dataT)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(dataX);

% Extract label data from cell and concatenate.
T = cat(2,dataT{1:end});

% One-hot encode labels.

```

```
T = onehotencode(T,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 concatenating 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(dataX)
```

```
% Concatenate.
```

```
X = cat(4,dataX{1:end});
```

```
% Normalize the images.
```

```
X = X/255;
```

```
end
```

## Input Arguments

### **prunableNet** — Network for pruning by using first-order Taylor approximation

TaylorPrunableNetwork object

Network for pruning by using first-order Taylor approximation, specified as a TaylorPrunableNetwork object.

### **maxToPrune** — Maximum number of filters to be pruned

8 (default) | numeric scalar

Maximum number of filters to be pruned, specified as a numeric integer-valued scalar

## Output Arguments

### **prunableNet\_new** — Pruned network

TaylorPrunableNetwork object

Network object from which filters with low Taylor-based importance scores have been removed, specified as a TaylorPrunableNetwork object.

## Version History

Introduced in R2022a

## See Also

[taylorPrunableNetwork](#) | [updateScore](#) | [dlnetwork](#) | [forward](#) | [predict](#)

## Topics

“Prune Filters in a Detection Network Using Taylor Scores”

# updateScore

**Package:** deep.prune

Compute and accumulate Taylor-based importance scores for pruning

## Syntax

```
prunableNet_new = updateScore(prunableNet,pruningActivations,
pruningGradients)
```

## Description

`prunableNet_new = updateScore(prunableNet,pruningActivations, pruningGradients)` computes and accumulates Taylor-based importance scores of convolution filters in prunable layers. This function returns another `TaylorPrunableNetwork` object whose state contains these updated scores.

To get robust estimates of the importance scores of the convolution filters in your network, execute `updateScore` several times on the same prunable network for different mini-batches of data.

To prune a deep neural network, you require the Deep Learning Toolbox Model Quantization Library support package. This support package is a free add-on that you can download using the Add-On Explorer. Alternatively, see [Deep Learning Toolbox Model Quantization Library](#).

## Examples

### Prune dlnetwork Object to Compress the Model

This example shows how to prune a `dlnetwork` object by using a custom pruning loop.

#### Load dlnetwork Object

Load a trained `dlnetwork` object and the corresponding classes.

```
s = load("digitsCustom.mat");
dlnet_1 = s.dlnet;
classes = s.classes;
```

Inspect the layers of the `dlnetwork` object. The network has three convolution layers at locations 2, 5, and 8 of the `Layer` array.

```
layers_1 = dlnet_1.Layers
```

```
layers_1 =
    12x1 Layer array with layers:
```

1	'input'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv1'	2-D Convolution	20 5x5x1 convolutions with stride [1 1] and padding
3	'bn1'	Batch Normalization	Batch normalization with 20 channels
4	'relu1'	ReLU	ReLU

```

5 'conv2'      2-D Convolution      20 3x3x20 convolutions with stride [1 1] and padding
6 'bn2'       Batch Normalization  Batch normalization with 20 channels
7 'relu2'     ReLU                  ReLU
8 'conv3'     2-D Convolution      20 3x3x20 convolutions with stride [1 1] and padding
9 'bn3'       Batch Normalization  Batch normalization with 20 channels
10 'relu3'    ReLU                  ReLU
11 'fc'       Fully Connected     10 fully connected layer
12 'softmax'  Softmax              softmax

```

### Load Data for Prediction

Load the digits data for prediction.

```

dataFolder = fullfile(toolboxdir("nnet"), "nndemos", "nndatasets", "DigitDataset");

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

```

Partition the data into pruning and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```
[imdsPrune, 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 images, use augmented image datastores.

```

inputSize = [28 28 1];
augimdsPrune = augmentedImageDatastore(inputSize(1:2), imdsPrune);
augimdsValidation = augmentedImageDatastore(inputSize(1:2), imdsValidation);

```

### Prune dlnetwork Object

Convert the `dlnetwork` object to a representation that is suitable for pruning by using the `taylorPrunableNetwork` function. This function returns a `TaylorPrunableNetwork` object that has the `NumPrunables` property set to 48. This indicates that 48 filters in the original model are suitable for pruning by using the Taylor pruning algorithm.

```
prunableNet_1 = taylorPrunableNetwork(dlnet_1)
```

```
prunableNet_1 =
TaylorPrunableNetwork with properties:
```

```

Learnables: [14x3 table]
State: [6x3 table]
InputNames: {'input'}
OutputNames: {'softmax'}
NumPrunables: 48

```

Create a `minibatchqueue` object that processes and manages mini-batches of images during pruning. 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 format 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 Computing Requirements” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(augimdsPrune, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    MiniBatchFormat=["SSCB" ""]);
```

Calculate Taylor-based importance scores of the prunable filters in the network by looping over the mini-batches of data. For each mini-batch:

- Calculate pruning activations and pruning gradients by using the `modelLoss` function defined at the end of this example
- Update importance scores of the prunable filters by using the `updateScore` function

```
while hasdata(mbq)
    [X,T] = next(mbq);
    [~,pruningActivations,pruningGradients] = dlfeval(@modelLoss,prunableNet_1,X,T);
    prunableNet_1 = updateScore(prunableNet_1,pruningActivations,pruningGradients);
end
```

Finally, remove filters with the lowest importance scores to create a new `TaylorPrunableNetwork` object by using the `updatePrunables` function. By default, a single call to this function removes 8 filters. Observe that the new network `prunableNet_2` has 40 prunable filters remaining.

```
prunableNet_2 = updatePrunables(prunableNet_1)
```

```
prunableNet_2 =
    TaylorPrunableNetwork with properties:

        Learnables: [14x3 table]
           State: [6x3 table]
    InputNames: {'input'}
    OutputNames: {'softmax'}
    NumPrunables: 40
```

To further compress the model, run the custom pruning loop and update prunables again.

### Extract Pruned `dlnetwork` Object

Use the `dlnetwork` function to extract the pruned `dlnetwork` object from the pruned `TaylorPrunableNetwork` object. You can now use this compressed `dlnetwork` object to perform inference.

```
dlnet_2 = dlnetwork(prunableNet_2);
```

Compare the convolution layers of the original and the pruned `dlnetwork` objects. Observe that the three convolution layers in the pruned network have fewer filters. These counts agree with the fact that, by default, a single call to the `updatePrunables` function removes 8 filters from the network.

```
conv_layers_1 = dlnet_1.Layers([2 5 8])
```

```
conv_layers_1 =  
  3x1 Convolution2DLayer array with layers:  
  
    1  'conv1'  2-D Convolution  20 5x5x1 convolutions with stride [1 1] and padding [0 0  
    2  'conv2'  2-D Convolution  20 3x3x20 convolutions with stride [1 1] and padding [1 1  
    3  'conv3'  2-D Convolution  20 3x3x20 convolutions with stride [1 1] and padding [1 1  
  
conv_layers_2 = dlnet_2.Layers([2 5 8])  
  
conv_layers_2 =  
  3x1 Convolution2DLayer array with layers:  
  
    1  'conv1'  2-D Convolution  17 5x5x1 convolutions with stride [1 1] and padding [0 0  
    2  'conv2'  2-D Convolution  18 3x3x17 convolutions with stride [1 1] and padding [1 1  
    3  'conv3'  2-D Convolution  17 3x3x18 convolutions with stride [1 1] and padding [1 1
```

## Supporting Functions

### Model Loss Function

The `modelLoss` function takes a `TaylorPrunableNetwork` object `net`, a mini-batch of input data `X` with corresponding targets `T` and returns activations in `net` and the gradients of the loss with respect to the activations in `net`. To compute the gradients automatically, this function uses the `dlgradient` function.

```
function [loss, pruningActivations, pruningGradients] = modelLoss(net,X,T)  
  
% Calculate network output for training.  
[out, ~, pruningActivations] = forward(net,X);  
  
% Calculate loss.  
loss = crossentropy(out,T);  
  
% Compute pruning gradients.  
pruningGradients = dlgradient(loss,pruningActivations);  
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,T] = preprocessMiniBatch(dataX,dataT)  
  
% Preprocess predictors.  
X = preprocessMiniBatchPredictors(dataX);  
  
% Extract label data from cell and concatenate.  
T = cat(2,dataT{1:end});  
  
% One-hot encode labels.
```



```
T = onehotencode(T,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 concatenating 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(dataX)
```

```
% Concatenate.
```

```
X = cat(4,dataX{1:end});
```

```
% Normalize the images.
```

```
X = X/255;
```

```
end
```

## Input Arguments

### **prunableNet** — Network for pruning by using first-order Taylor approximation

TaylorPrunableNetwork object

Network for pruning by using first-order Taylor approximation, specified as a TaylorPrunableNetwork object.

### **pruningActivations** — Activations of the pruning layers

cell array containing darray objects

Activations of the pruning layers, specified as a cell array containing darray objects. To retrieve these values, call the `forward` function on the prunable network.

### **pruningGradients** — Gradients of loss with respect to activations

cell array containing darray objects

Gradients of loss with respect to `pruningActivations`, specified as a cell array containing darray objects. To calculate `pruningGradients`, first calculate the loss and then use the `dlgradient` function.

## Output Arguments

### **prunableNet\_new** — Updated network for pruning

TaylorPrunableNetwork object

Network object for pruning that been updated to contain the accumulated Taylor-based importance scores of the prunable filters, specified as a TaylorPrunableNetwork object.

## Version History

Introduced in R2022a

**See Also**

taylorPrunableNetwork | updatePrunables | dlnetwork | forward | predict

**Topics**

“Prune Filters in a Detection Network Using Taylor Scores”

# validate

Quantize and validate a deep neural network

## Syntax

```
valResults = validate(quantObj, valData)
valResults = validate(quantObj, valData, quantOpts)
```

## Description

`valResults = 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`, using the data specified by `valData`.

`valResults = validate(quantObj, valData, quantOpts)` quantizes and validates the network with additional options specified by `quantOpts`.

This function requires Deep Learning Toolbox Model Quantization Library. 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 squeezeNetmerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
  Connections: [75x2 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=121x5 table
    Optimized Layer Name      Network Layer Name      Learnables / Activations      MinValue
```

```

_____
{'conv1_Weights'      }      {'conv1'      }      "Weights"      -0.9198
{'conv1_Bias'        }      {'conv1'      }      "Bias"         -0.0792
{'fire2-squeeze1x1_Weights'}  {'fire2-squeeze1x1'}  "Weights"      -1.3
{'fire2-squeeze1x1_Bias' }  {'fire2-squeeze1x1'}  "Bias"         -0.1164
{'fire2-expand1x1_Weights'}  {'fire2-expand1x1' }  "Weights"      -0.740
{'fire2-expand1x1_Bias' }  {'fire2-expand1x1' }  "Bias"         -0.06005
{'fire2-expand3x3_Weights'}  {'fire2-expand3x3' }  "Weights"      -0.7439
{'fire2-expand3x3_Bias' }  {'fire2-expand3x3' }  "Bias"         -0.05177
{'fire3-squeeze1x1_Weights'}  {'fire3-squeeze1x1'}  "Weights"      -0.771
{'fire3-squeeze1x1_Bias' }  {'fire3-squeeze1x1'}  "Bias"         -0.1013
{'fire3-expand1x1_Weights'}  {'fire3-expand1x1' }  "Weights"      -0.7203
{'fire3-expand1x1_Bias' }  {'fire3-expand1x1' }  "Bias"         -0.06702
{'fire3-expand3x3_Weights'}  {'fire3-expand3x3' }  "Weights"      -0.6144
{'fire3-expand3x3_Bias' }  {'fire3-expand3x3' }  "Bias"         -0.05361
{'fire4-squeeze1x1_Weights'}  {'fire4-squeeze1x1'}  "Weights"      -0.742
{'fire4-squeeze1x1_Bias' }  {'fire4-squeeze1x1'}  "Bias"         -0.1088
:

```

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: [1x1 struct]
    Statistics: [2x2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2x2 table
    NetworkImplementation    MetricOutput
    _____
    {'Floating-Point'}      1
    {'Quantized'           }      1
```

```
valResults.Statistics
```

```
ans=2x2 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 Network for FPGA Deployment

Reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. This example shows how to use Deep Learning Toolbox Model Quantization Library and Deep Learning HDL Toolbox to deploy the `int8` network to a target FPGA board.

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning.

## Load Pretrained Network

Load the pretrained LogoNet network and analyze the network architecture.

```
snet = getLogoNetwork;  
deepNetworkDesigner(snet);
```

## Load Data

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 for calibration and validation. Expedite calibration and validation 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. The FPGA validation workflow has a maximum limit of one image when validating the quantized network. Reduce the FPGA validation data set to a single image.

```
curDir = pwd;  
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_simulation = validationData.subset(1:5);  
validationData_FPGA = validationData.subset(1:1);
```

## Generate Calibration Result File for the Network

Create a `dlquantizer` (Deep Learning HDL Toolbox) object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj_simulation = dlquantizer(snet,'ExecutionEnvironment',"FPGA",'Simulation','on');  
dlQuantObj_FPGA = dlquantizer(snet,'ExecutionEnvironment',"FPGA");
```

Use the `calibrate` (Deep Learning HDL Toolbox) function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases. The `calibrate` function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj_simulation,calibrationData_reduced)
```

ans=35x5 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"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045961
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02953
{'imageinput' }	{'imageinput' }	"Activations"	( )
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

```
calibrate(dlQuantObj_FPGA,calibrationData_reduced)
```

ans=35x5 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"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045942
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.0013998
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045961
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.00164
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051399
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052319
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.05010
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.0017564
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.050700
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.02953
{'imageinput' }	{'imageinput' }	"Activations"	( )
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.34
:			

### Create Target Object

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 use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\n2020.2\bin\vivado.bat');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, you can also use the JTAG interface.

```
% hTarget = dlhdl.Target('Xilinx', 'Interface', 'JTAG');
```

### Create dlQuantizationOptions Object

Create a `dlquantizationOptions` object. Specify the target bitstream and target board interface. The default metric function is a Top-1 accuracy metric function.

```
options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);  
options_simulation = dlquantizationOptions;
```

To use a custom metric function, specify the metric function in the `dlquantizationOptions` object.

```
options_FPGA = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_FPGA)});  
options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_simulation)});
```

### Validate Quantized Network

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_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulation);
```

```
Compiling leg: conv_1>>relu_4 ...  
Compiling leg: conv_1>>relu_4 ... complete.  
Compiling leg: maxpool_4 ...  
Compiling leg: maxpool_4 ... complete.  
Compiling leg: fc_1>>fc_3 ...  
Compiling leg: fc_1>>fc_3 ... complete.
```

```
prediction_simulation = struct with fields:  
    NumSamples: 5  
    MetricResults: [1x1 struct]  
    Statistics: []
```

For validation on an FPGA, the `validate` function:

- Programs the FPGA board by using the output of the `compile` method and the programming file
- Downloads the network weights and biases



- Compares the performance of the network before and after quantization

```
prediction_FPGA = dlQuantObj_FPGA.validate(validationData_FPGA,options_FPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
  1  'imageinput'    Image Input          227x227x3 images with 'zerocenter' normalization
  2  'conv_1'       Convolution          96 5x5x3 convolutions with stride [1 1] and padding
  3  'relu_1'      ReLU                 ReLU
  4  'maxpool_1'   Max Pooling          3x3 max pooling with stride [2 2] and padding
  5  'conv_2'     Convolution          128 3x3x96 convolutions with stride [1 1] and padding
  6  'relu_2'     ReLU                 ReLU
  7  'maxpool_2'   Max Pooling          3x3 max pooling with stride [2 2] and padding
  8  'conv_3'     Convolution          384 3x3x128 convolutions with stride [1 1] and padding
  9  'relu_3'     ReLU                 ReLU
 10  'maxpool_3'   Max Pooling          3x3 max pooling with stride [2 2] and padding
 11  'conv_4'     Convolution          128 3x3x384 convolutions with stride [2 2] and padding
 12  'relu_4'     ReLU                 ReLU
 13  'maxpool_4'   Max Pooling          3x3 max pooling with stride [2 2] and padding
 14  'fc_1'       Fully Connected      2048 fully connected layer
 15  'relu_5'     ReLU                 ReLU
 16  'dropout_1'  Dropout              50% dropout
 17  'fc_2'       Fully Connected      2048 fully connected layer
 18  'relu_6'     ReLU                 ReLU
 19  'dropout_2'  Dropout              50% dropout
 20  'fc_3'       Fully Connected      32 fully connected layer
 21  'softmax'    Softmax              softmax
 22  'classoutput' Classification Output crossentropyex with 'adidas' and 31 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
### Compiling layer group: conv_1>>relu_4 ...
### Compiling layer group: conv_1>>relu_4 ... complete.
### Compiling layer group: maxpool_4 ...
### Compiling layer group: maxpool_4 ... complete.
### Compiling layer group: fc_1>>fc_3 ...
### Compiling layer group: fc_1>>fc_3 ... complete.

### Allocating external memory buffers:

      offset_name          offset_address      allocated_space
-----
"InputDataOffset"        "0x00000000"        "12.0 MB"
"OutputResultOffset"    "0x00c00000"        "4.0 MB"
"SchedulerDataOffset"   "0x01000000"        "4.0 MB"
"SystemBufferOffset"    "0x01400000"        "36.0 MB"
"InstructionDataOffset" "0x03800000"        "8.0 MB"
"ConvWeightDataOffset"  "0x04000000"        "12.0 MB"
"FCWeightDataOffset"    "0x04c00000"        "12.0 MB"
"EndOffset"              "0x05800000"        "Total: 88.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Deep learning network programming has been skipped as the same network is already loaded on the target
```

```
### Finished writing input activations.
### Running single input activation.
```

Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	248358	274080	90.62
DSPs	384	2520	15.24
Block RAM	581	912	63.71

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

```
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	40142478	0.18247	1	40142478
__imageinput_norm	216472	0.00098		
__conv_1	6825671	0.03103		
__maxpool_1	3755088	0.01707		
__conv_2	10440701	0.04746		
__maxpool_2	1447840	0.00658		
__conv_3	9405685	0.04275		
__maxpool_3	1765856	0.00803		
__conv_4	1819636	0.00827		
__maxpool_4	28098	0.00013		
__fc_1	2651288	0.01205		
__fc_2	1696632	0.00771		
__fc_3	89511	0.00041		

\* The clock frequency of the DL processor is: 220MHz

Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	168645	274080	61.53
DSPs	800	2520	31.75
Block RAM	453	912	49.67

\* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

```
### Finished writing input activations.
### Running single input activation.
```

```
prediction_FPGA = struct with fields:
  NumSamples: 1
  MetricResults: [1x1 struct]
  Statistics: [2x7 table]
```

## View Performance of Quantized Neural Network

Display the accuracy of the quantized network.

```
prediction_simulation.MetricResults.Result
```

```
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}      1
  {'Quantized'      }      1
```

```
prediction_FPGA.MetricResults.Result
```

```
ans=2x2 table
  NetworkImplementation  MetricOutput
  _____  _____
  {'Floating-Point'}      1
  {'Quantized'      }      1
```

Display the performance of the quantized network in frames per second.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
```

```
ans = 19.0828
```

## Quantize a Neural Network for CPU Target

This example shows how to quantize and validate a neural network for a CPU target. This workflow is similar to other execution environments, but before validating you must establish a `raspi` connection and specify it as target using `dlquantizationOptions`.

First, load your network. This example uses the pretrained network `squeezenet`.

```
load squeezenetmerch
net

net =
  DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 table]
    InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Then define your calibration and validation data, `calDS` and `valDS` respectively.

```
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 the dlquantizer object and specify a CPU execution environment.

```
dq = dlquantizer(net,'ExecutionEnvironment','CPU')

dq =
  dlquantizer with properties:
    NetworkObject: [1x1 DAGNetwork]
    ExecutionEnvironment: 'CPU'
```

Calibrate the network.

```
calResults = calibrate(dq,aug_calData,'UseGPU','off')
```

```
calResults=122x5 table
    Optimized Layer Name      Network Layer Name      Learnables / Activations      MinValue
    _____            _____            _____            _____
    {'conv1_Weights'         } {'conv1'                } "Weights"                  -0.9198
    {'conv1_Bias'            } {'conv1'                } "Bias"                     -0.0792
    {'fire2-squeeze1x1_Weights'} {'fire2-squeeze1x1'     } "Weights"                  -1.3
    {'fire2-squeeze1x1_Bias' } {'fire2-squeeze1x1'     } "Bias"                     -0.1164
    {'fire2-expand1x1_Weights'} {'fire2-expand1x1'      } "Weights"                  -0.740
    {'fire2-expand1x1_Bias'  } {'fire2-expand1x1'      } "Bias"                     -0.06005
    {'fire2-expand3x3_Weights'} {'fire2-expand3x3'      } "Weights"                  -0.7439
    {'fire2-expand3x3_Bias'  } {'fire2-expand3x3'      } "Bias"                     -0.05177
    {'fire3-squeeze1x1_Weights'} {'fire3-squeeze1x1'     } "Weights"                  -0.771
    {'fire3-squeeze1x1_Bias' } {'fire3-squeeze1x1'     } "Bias"                     -0.1013
    {'fire3-expand1x1_Weights'} {'fire3-expand1x1'      } "Weights"                  -0.7203
    {'fire3-expand1x1_Bias'  } {'fire3-expand1x1'      } "Bias"                     -0.06702
    {'fire3-expand3x3_Weights'} {'fire3-expand3x3'      } "Weights"                  -0.6144
    {'fire3-expand3x3_Bias'  } {'fire3-expand3x3'      } "Bias"                     -0.05361
    {'fire4-squeeze1x1_Weights'} {'fire4-squeeze1x1'     } "Weights"                  -0.742
    {'fire4-squeeze1x1_Bias' } {'fire4-squeeze1x1'     } "Bias"                     -0.1088
    :
```

Use the MATLAB Support Package for Raspberry Pi Hardware function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name or address of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiname','username','password')
```

For example,

```
r = raspi('gpcoder-raspberrypi-7','pi','matlab')

r =
  raspi with properties:
    DeviceAddress: 'gpcoder-raspberrypi-7'
```

```

        Port: 18734
        BoardName: 'Raspberry Pi 3 Model B+'
        AvailableLEDs: {'led0'}
        AvailableDigitalPins: [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
        AvailableSPIChannels: {}
        AvailableI2CBuses: {}
        AvailableWebcams: {}
        I2CBusSpeed:
AvailableCANInterfaces: {}

Supported peripherals

```

Specify `raspi` object as the target for the quantized network.

```

opts = dlquantizationOptions('Target',r)

opts =
dlquantizationOptions with properties:

    MetricFcn: {}
    Bitstream: ''
    Target: [1x1 raspi]

```

Validate the quantized network with the `validate` function.

```

valResults = validate(dq, aug_valData, opts)

### Starting application: 'codegen\lib\validate_predict_int8\pil\validate_predict_int8.elf'
To terminate execution: clear validate_predict_int8_pil
### Launching application validate_predict_int8.elf...
### Host application produced the following standard output (stdout) and standard error (stderr)

valResults = struct with fields:
    NumSamples: 20
    MetricResults: [1x1 struct]
    Statistics: []

```

Examine the validation output to see the performance of the quantized network.

```

valResults.MetricResults.Result

ans=2x2 table
    NetworkImplementation    MetricOutput
    _____    _____
    {'Floating-Point'}      0.95
    {'Quantized' }          0.95

```

## Quantize YOLO v3 Object Detector

This example shows how to quantize a `yolov3objectDetector` (Computer Vision Toolbox) object using preprocessed calibration and validation data.

First, download a pretrained YOLO v3 object detector.

```
detector = downloadPretrainedNetwork();
```

This example uses a small labeled data set that contains one or two labeled instances of a vehicle. Many of these images come from the Caltech Cars 1999 and 2001 data sets, created by Pietro Perona and used with permission.

Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

Add the full path to the local vehicle data folder.

```
vehicleDataset.imageFilename = fullfile(pwd, vehicleDataset.imageFilename);
```

Create an `imageDatastore` for loading the images and a `boxLabelDatastore` (Computer Vision Toolbox) for the ground truth bounding boxes.

```
imds = imageDatastore(vehicleDataset.imageFilename);
blDs = boxLabelDatastore(vehicleDataset(:,2));
```

Use the `combine` function to combine both the datastores into a `CombinedDatastore`.

```
combinedDS = combine(imds, blDs);
```

Split the data into calibration and validation data.

```
calData = combinedDS.subset(1:32);
valData = combinedDS.subset(33:64);
```

Use the `preprocess` (Computer Vision Toolbox) method of `yoloV3ObjectDetector` (Computer Vision Toolbox) object with `transform` function to prepare the data for calibration and validation.

The `transform` function returns a `TransformedDatastore` object.

```
processedCalData = transform(calData, @(data)preprocess(detector,data));
processedValData = transform(valData, @(data)preprocess(detector,data));
```

Create the `dlquantizer` object.

```
dq = dlquantizer(detector, 'ExecutionEnvironment', 'MATLAB');
```

Calibrate the network.

```
calResults = calibrate(dq, processedCalData, 'UseGPU', 'off')
```

*calResults=135x5 table*

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.92189
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.09627
{'fire2-squeeze1x1_Weights' }	{'fire2-squeeze1x1' }	"Weights"	-1.375
{'fire2-squeeze1x1_Bias' }	{'fire2-squeeze1x1' }	"Bias"	-0.1206
{'fire2-expand1x1_Weights' }	{'fire2-expand1x1' }	"Weights"	-0.7527

```

{'fire2-expand1x1_Bias' } {'fire2-expand1x1' } "Bias" -0.05925
{'fire2-expand3x3_Weights' } {'fire2-expand3x3' } "Weights" -0.7527
{'fire2-expand3x3_Bias' } {'fire2-expand3x3' } "Bias" -0.06221
{'fire3-squeeze1x1_Weights' } {'fire3-squeeze1x1' } "Weights" -0.758
{'fire3-squeeze1x1_Bias' } {'fire3-squeeze1x1' } "Bias" -0.1020
{'fire3-expand1x1_Weights' } {'fire3-expand1x1' } "Weights" -0.7156
{'fire3-expand1x1_Bias' } {'fire3-expand1x1' } "Bias" -0.06931
{'fire3-expand3x3_Weights' } {'fire3-expand3x3' } "Weights" -0.6007
{'fire3-expand3x3_Bias' } {'fire3-expand3x3' } "Bias" -0.05804
{'fire4-squeeze1x1_Weights' } {'fire4-squeeze1x1' } "Weights" -0.73
{'fire4-squeeze1x1_Bias' } {'fire4-squeeze1x1' } "Bias" -0.1118
:

```

Validate the quantized network with the `validate` function.

```

valResults = validate(dq, processedValData)

valResults = struct with fields:
    NumSamples: 32
    MetricResults: [1x1 struct]
    Statistics: []

function detector = downloadPretrainedNetwork()
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehicleExample_21aSPKG.zip';
    websave('yolov3SqueezeNetVehicleExample_21aSPKG.zip', pretrainedURL);

    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');

    pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
    detector = pretrained.detector;
end

```

## Input Arguments

### **quantObj** — Network to quantize

`dlquantizer` object

Network to quantize, specified as a `dlquantizer` object.

### **valData** — Data to use for validation of quantized network

`imageDatastore` object | `augmentedImageDatastore` object | `pixelLabelImageDatastore` object | `CombinedDatastore` object | `TransformedDatastore` object

Data to use for validation of quantized network, specified as an `imageDatastore` object, an `augmentedImageDatastore` object, a `pixelLabelImageDatastore` object, a `CombinedDatastore` object, or a `TransformedDatastore` object.

You must preprocess the data used for validation of a `yolov3objectDetector` object using the `preprocess` function. For an example of using preprocessed data for validation of a `yolov3objectDetector`, see “Quantize YOLO v3 Object Detector” on page 1-1963.

`validate` accepts a `CombinedDatastore` or `TransformedDatastore` object as input data for validating quantized `yolov3objectDetector` and `yolov4objectDetector` objects. The

CombinedDatastore and TransformedDatastore used for validation must contain an imageDatastore or augmentedImageDatastore as the first datastore and a boxLabelDatastore as the second datastore. For more information on valid datastores, see “Prepare Data for Quantizing Networks”.

**quantOpts — Options for quantizing network**

dlquantizationOptions object

Options for quantizing the network, specified as a dlquantizationOptions object.

**Output Arguments**

**valResults — Performance of quantized network**

struct

Performance of quantized network, returned as a struct. The struct contains these fields.

- NumSamples — The number of sample inputs used to validate the network, specified by valData.
- 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	Metric function used to determine the performance of the quantized network, specified in the dlquantizationOptions object.
Result	Table indicating the results of the metric function before and after quantization.  The first row in the table, 'Floating-Point', contains information for the original floating-point implementation. The second row, 'Quantized', contains information for the quantized implementation. The output of the metric function is displayed in the MetricOutput column.

- Statistics — Table indicating the learnable parameter memory used, in bytes, by the original floating-point implementation of the network and the quantized implementation.

When the ExecutionEnvironment for the dlquantizer object is set to FPGA, the Statistics table is a table indicating these values for the original floating-point and quantized network implementations:

- Frames per second
- Number of convolution threads
- Number of fully connected threads
- Lookup table (LUT) resource utilization percentage



- Block RAM resource utilization percentage
- DSP resource utilization percentage

## Algorithms

The `validate` function determines the default 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 <code>evaluateDetectionPrecision</code> .
Regression	MSE — Mean squared error of the network
Semantic Segmentation	<code>evaluateSemanticSegmentation</code> — 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 'Simulation' property of the `dlquantizer` object is set to 'on', the default metric function uses at most 5 files from the validation datastore. Custom metric functions, specified using a `dlquantizationOptions` object, do not have this restriction.

## Version History

Introduced in R2020a

### R2022a: Validate the performance of quantized network for CPU target

You can now use the `dlquantizer` object and the `validate` function to quantize a network and generate code for CPU targets.

## See Also

### Apps

Deep Network Quantizer

### Functions

`calibrate` | `dlquantizer` | `dlquantizationOptions` | `quantize` | `quantizationDetails` | `estimateNetworkMetrics`

### Topics

“Quantization Workflow Prerequisites”

“Prepare Data for Quantizing Networks”

“Quantization of Deep Neural Networks”

“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”

“Quantize Network for FPGA Deployment” (Deep Learning HDL Toolbox)

“Generate int8 Code for Deep Learning Networks” (MATLAB Coder)

# verifyNetworkRobustness

Verify adversarial robustness of deep learning network

## Syntax

```
result = verifyNetworkRobustness(net,XLower,XUpper,label)
```

## Description

`result = verifyNetworkRobustness(net,XLower,XUpper,label)` verifies whether the network `net` is adversarially robust with respect to the class `label` when the input is between `XLower` and `XUpper`. For more information, see “Adversarial Examples” on page 1-1973.

A network is robust to adversarial examples for a specific input if the predicted class does not change when the input is perturbed between `XLower` and `XUpper`. For more information, see “Algorithms” on page 1-1973.

The `verifyNetworkRobustness` function requires the Deep Learning Toolbox Verification Library support package. If this support package is not installed, use the **Add-On Explorer**. To open the **Add-On Explorer**, go to the MATLAB Toolstrip and click **Add-Ons > Get Add-Ons**.

## Examples

### Verify Network Robustness

Verify the adversarial robustness of an image classification network.

Load a pretrained classification network. This network is a `dlnetwork` object that has been trained to predict the class label of images of handwritten digits.

```
load("digitsRobustClassificationMLPNet.mat");
```

Prepare the network for verification by removing the softmax layer. When you remove layers from a `dlnetwork` object, the software returns the network as an uninitialized `dlnetwork` object. To initialize the network, use the `initialize` function.

```
netRobust = removeLayers(netRobust,"softmax");
netRobust = initialize(netRobust);
```

Load the test data.

```
[XTest,TTest] = digitTest4DArrayData;
```

Select the first ten images.

```
X = XTest(:,:,,1:10);
label = TTest(1:10);
```

Convert the test data to a `dlarray` object.

```
X = dlarray(X,"SSCB");
```

Verify the network robustness to an input perturbation between -0.01 and 0.01 for each pixel. Create lower and upper bounds for the input.

```
perturbation = 0.01;  
XLower = X - perturbation;  
XUpper = X + perturbation;
```

Verify the network robustness for each test image.

```
result = verifyNetworkRobustness(netRobust,XLower,XUpper,label);  
summary(result)
```

```
verified      10  
violated      0  
unproven      0
```

### Approximate Maximum Adversarial Perturbation

Find the maximum adversarial perturbation that you can apply to an input without changing the predicted class.

Load a pretrained classification network. This network is a `dlnetwork` object that has been trained to predict the class of images of handwritten digits.

```
load("digitsRobustClassificationMLPNet.mat");
```

Prepare the network for verification by removing the softmax layer. When you remove layers from a `dlnetwork` object, the software returns the network as an uninitialized `dlnetwork` object. To initialize the network, use the `initialize` function.

```
netRobust = removeLayers(netRobust,"softmax");  
netRobust = initialize(netRobust);
```

Load the test data.

```
[XTest,TTest] = digitTest4DArrayData;
```

Select a test image.

```
idx = 3;  
X = XTest(:,:,:,idx);  
label = TTest(idx);
```

Create lower and upper bounds for a range of perturbation values.

```
perturbationRange = 0:0.005:0.05;  
  
for i = 1:numel(perturbationRange)  
XLower(:,:,:,i) = X - perturbationRange(i);  
XUpper(:,:,:,i) = X + perturbationRange(i);  
end
```

Repeat the class label for each set of bounds.

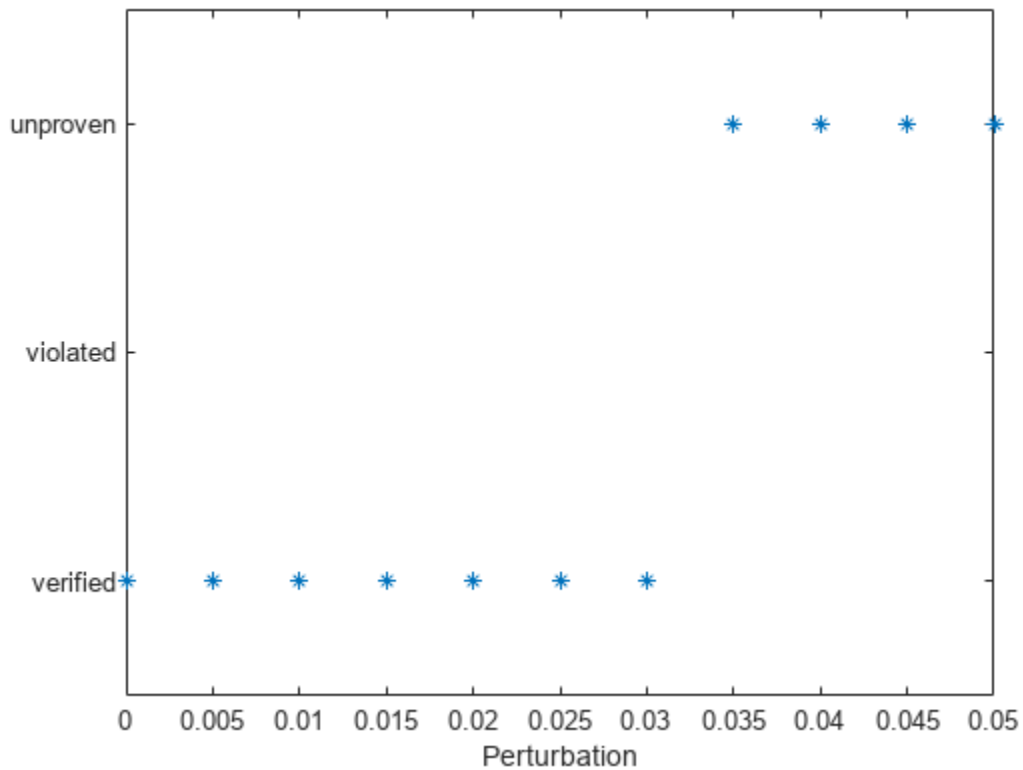
```
label = repmat(label,numel(perturbationRange),1);
```

Convert the bounds to `dLarray` objects.

```
XLower = dLarray(XLower, "SSCB");
XUpper = dLarray(XUpper, "SSCB");
```

Verify the adversarial robustness for each perturbation.

```
result = verifyNetworkRobustness(netRobust, XLower, XUpper, label);
plot(perturbationRange, result, "*")
xlabel("Perturbation")
```



Find the maximum perturbation value for which the function returns verified.

```
maxIdx = find(result=="verified",1,"last");
maxPerturbation = perturbationRange(maxIdx)

maxPerturbation = 0.0300
```

## Input Arguments

### **net** — Network

initialized `dLnetwork` object

Network, specified as an initialized `dLnetwork` object. To initialize a `dLnetwork` object, use the `initialize` function.

The function supports networks with these layers:

- `imageInputLayer` with `Normalization="none"`
- `featureInputLayer` with `Normalization="none"`
- `fullyConnectedLayer`
- `convolution2dLayer` with `Dilation` set to `[1 1]` or `1` and `PaddingValue` set to `0`
- `reluLayer`
- `tanhLayer`
- `batchNormalizationLayer`
- `averagePooling2dLayer` with `PaddingValue` set to `0`
- `globalAveragePooling2dLayer`
- `dropoutLayer`
- `sigmoidLayer`

The function does not support networks with multiple inputs and multiple outputs.

The function verifies the network using the final layer. For most applications, use the final fully connected layer for verification. If your network has a different layer as its final layer (for example, softmax), remove the layer before calling the function.

**XLower — Input lower bound**

formatted `darray` object

Input lower bound, specified as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The lower and upper bounds, `XLower` and `XUpper`, must have the same size and format. The function computes the results across the batch ("B") dimension of the input lower and upper bounds.

**XUpper — Input upper bound**

formatted `darray` object

Input upper bound, specified as a formatted `darray` object. For more information about `darray` formats, see the `fmt` input argument of `darray`.

The lower and upper bounds, `XLower` and `XUpper`, must have the same size and format. The function computes the results across the batch ("B") dimension of the input lower and upper bounds.

**label — Class label**

numeric index | categorical

Class label, specified as a numeric index or a categorical, or a vector of these values. The length of `label` must match the size of the batch ("B") dimension of the lower and upper bounds.

The function verifies that the predicted class that the network returns matches `label` for any input in the range defined by the lower and upper bounds.

---

**Note** If you specify `label` as a categorical, then the order of the categories must match the order of the outputs in the network.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

## Output Arguments

### result — Verification result

categorical array

Verification result, returned as a categorical array. For each set of input lower and upper bounds, the function returns the corresponding element of this array as one of these values:

- "verified" — The network is robust to adversarial inputs between the specified bounds.
- "violated" — The network is not robust to adversarial inputs between the specified bounds.
- "unproven" — Unable to prove whether the network is robust to adversarial inputs between the specified bounds.

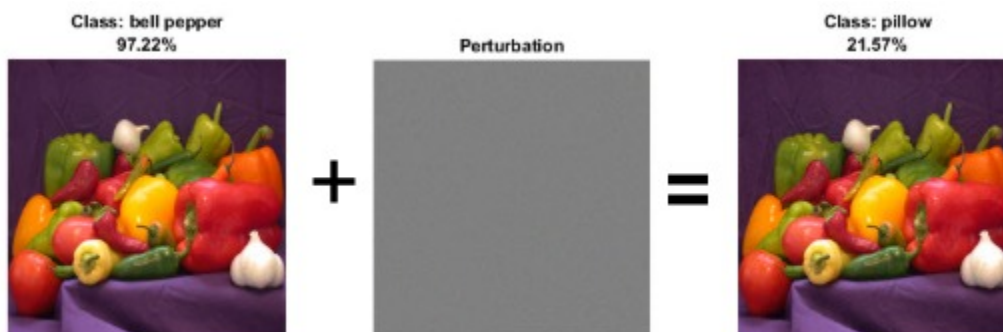
The function computes the results across the batch ("B") dimension of the input lower and upper bounds. If you supply  $k$  upper bounds, lower bounds, and labels, then `result(k)` corresponds to the verification result for the  $k$ th input lower and upper bounds with respect to `label(k)`. For more information, see "Algorithms" on page 1-1973.

## More About

### Adversarial Examples

Neural networks can be susceptible to a phenomenon known as *adversarial examples* [1], where very small changes to an input can cause the network predictions to significantly change. For example, making a small change to an image that causes the network to misclassify it. These changes are often imperceptible to humans.

For example, this image shows that adding an imperceptible perturbation to an image of peppers means that the classification changes from "bell pepper" to "pillow".



A network is *adversarially robust* if the output of the network does not change significantly when the input is perturbed. For classification tasks, adversarial robustness means that the output of the fully connected layer with the highest value does not change, and therefore the predicted class does not change.

## Algorithms

To verify the robustness of a network for an input, the function checks that when the input is perturbed between the specified lower and upper bound, the output does not significantly change.

Let  $X$  be an input with respect to which you want to test the robustness of the network. To use the `verifyNetworkRobustness` function, you must specify a lower and upper bound for the input. For example, let  $\epsilon$  be a small perturbation. You can define a lower and upper bound for the input as  $X_{\text{lower}} = X - \epsilon$  and  $X_{\text{upper}} = X + \epsilon$ , respectively.

To verify the adversarial robustness of the network, the function checks that, for all inputs between  $X_{\text{lower}}$  and  $X_{\text{upper}}$ , no adversarial example exists. To check for adversarial examples, the function uses these steps.

- 1 Create an input set using the lower and upper input bounds.
- 2 Pass the input set through the network and return an output set. To reduce computational overhead, the function performs abstract interpretation by approximating the output of each layer using the DeepPoly [2] method.
- 3 Check if the specified label remains the same for the entire input set. Because the algorithm uses overapproximation when it computes the output set, the result can be unproven if part of the output set corresponds to an adversarial example.

If you specify multiple pairs of input lower and upper bounds, then the function verifies the robustness for each pair of input bounds.

---

**Note** Because of floating-point round-off error, the verification results might be slightly different when working with networks produced using C/C++ code generation.

---

## Version History

Introduced in R2022b

## References

- [1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples." Preprint, submitted March 20, 2015. <https://arxiv.org/abs/1412.6572>.
- [2] Singh, Gagandeep, Timon Gehr, Markus Püschel, and Martin Vechev. "An Abstract Domain for Certifying Neural Networks". *Proceedings of the ACM on Programming Languages* 3, no. POPL (January 2, 2019): 1-30. <https://doi.org/10.1145/3290354>.

## See Also

`estimateNetworkOutputBounds` | `dlarray` | `dlnetwork`

## Topics

- "Verify Robustness of Deep Learning Neural Network"
- "Generate Untargeted and Targeted Adversarial Examples for Image Classification"
- "Train Image Classification Network Robust to Adversarial Examples"
- "Train Network Using Custom Training Loop"
- "Out-of-Distribution Detection for Deep Neural Networks"



# quantize

Quantize deep neural network

## Syntax

```
quantizedNetwork = quantize(quantObj)  
quantizedNetwork = quantize(quantObj,Name,Value)
```

## Description

`quantizedNetwork = quantize(quantObj)` quantizes a deep neural network using a calibrated `dlquantizer` object, `quantObj`. The quantized neural network object, `quantizedNetwork`, enables visibility of the quantized layers, weights, and biases of the network, as well as simulatable quantized inference behavior.

`quantizedNetwork = quantize(quantObj,Name,Value)` specifies additional options using one or more name name-value arguments.

This function requires Deep Learning Toolbox Model Quantization Library. To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

## Examples

### Emulate Target Agnostic Quantized Network

This example shows how to create a target agnostic, simulatable quantized deep neural network in MATLAB.

Target agnostic quantization allows you to see the effect quantization has on your neural network without target hardware or target-specific quantization schemes. Creating a target agnostic quantized network is useful if you:

- Do not have access to your target hardware.
- Want to preview whether or not your network is suitable for quantization.
- Want to find layers that are sensitive to quantization.

Quantized networks emulate quantized behavior for quantization-compatible layers. Network architecture like layers and connections are the same as the original network, but inference behavior uses limited precision types. Once you have quantized your network, you can use the `quantizationDetails` function to retrieve details on what was quantized.

Load the pretrained network. `net` is a SqueezeNet network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch  
net  
  
net =  
    DAGNetwork with properties:
```

```
Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
InputNames: {'data'}
OutputNames: {'new_classoutput'}
```

You can use the `quantizationDetails` function to see that the network is not quantized.

```
qDetailsOriginal = quantizationDetails(net)
```

```
qDetailsOriginal = struct with fields:
    IsQuantized: 0
    TargetLibrary: ""
    QuantizedLayerNames: [0x0 string]
    QuantizedLearnables: [0x3 table]
```

Unzip and load the `MerchData` images as an image datastore.

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

Define calibration and validation data to use for quantization. The output size of the images are changed for both calibration and validation data according to network requirements.

```
[calData,valData] = splitEachLabel(imds,0.7,'randomized');
augCalData = augmentedImageDatastore([227 227],calData);
augValData = augmentedImageDatastore([227 227],valData);
```

Create `dlquantizer` object and specify the network to quantize. Set the execution environment to MATLAB. How the network is quantized depends on the execution environment. The MATLAB execution environment is agnostic to the target hardware and allows you to prototype quantized behavior.

```
quantObj = dlquantizer(net,'ExecutionEnvironment','MATLAB');
```

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,augCalData);
```

Use the `quantize` method to quantize the network object and return a simulatable quantized network.

```
qNet = quantize(quantObj)
```

```
qNet =
Quantized DAGNetwork with properties:

    Layers: [68x1 nnet.cnn.layer.Layer]
Connections: [75x2 table]
InputNames: {'data'}
```

```
OutputNames: {'new_classoutput'}
```

Use the `quantizationDetails` method to extract quantization details.

You can use the `quantizationDetails` function to see that the network is now quantized.

```
qDetailsQuantized = quantizationDetails(qNet)
```

```
qDetailsQuantized = struct with fields:
    IsQuantized: 1
    TargetLibrary: "none"
    QuantizedLayerNames: [26x1 string]
    QuantizedLearnables: [52x3 table]
```

Make predictions using the original, single-precision floating-point network, and the quantized INT8 network.

```
predOriginal = classify(net, augValData);           % Predictions for the non-quantized network
predQuantized = classify(qNet, augValData);        % Predictions for the quantized network
```

Compute the relative accuracy of the quantized network as compared to the original network.

```
ccrQuantized = mean(predQuantized == valData.Labels)*100
ccrQuantized = 100
```

```
ccrOriginal = mean(predOriginal == valData.Labels)*100
ccrOriginal = 100
```

For this validation data set, the quantized network gives the same predictions as the floating-point network.

## Emulate GPU Target Behavior for Quantized Network

This example shows how to emulate the behavior of a quantized network for GPU deployment. Once you quantize your network for a GPU execution environment, you can emulate the GPU target behavior without the GPU hardware. Doing so allows you to examine your quantized network structure and behavior without generating code for deployment.

Emulated quantized networks are not smaller than the original network.

Load the pretrained network. `net` is a SqueezeNet convolutional neural network that has been retrained using transfer learning to classify images in the `MerchData` data set.

```
load squeezeNetMerch
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph, "new_classoutput");
dlNet = dlNetwork(lgraph)

dlNet =
    dlNetwork with properties:
```

```
Layers: [67x1 nnet.cnn.layer.Layer]
Connections: [74x2 table]
Learnables: [52x3 table]
State: [0x3 table]
InputNames: {'data'}
OutputNames: {'prob'}
Initialized: 1
```

View summary with `summary`.

Define calibration and validation data to use for quantization.

Use the calibration data 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.

Use the validation data to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

For this example, use the images in the MerchData data set. Split the data into calibration and validation data sets.

```
unzip("MerchData.zip");
imds = imageDatastore("MerchData", ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");
classes = categories(imds.Labels);
[calData, valData] = splitEachLabel(imds, 0.7, "randomized");
```

Create a `dlquantizer` object and specify the network to quantize. How the network is quantized depends on the execution environment. Set `ExecutionEnvironment` to `GPU` to perform quantization specific to GPU target hardware.

```
quantObj = dlquantizer(dlnet, ExecutionEnvironment="GPU");
```

Use the `calibrate` function to exercise the network object with sample inputs and collect range information.

```
calResults = calibrate(quantObj, calData);
```

Use the `quantize` method to quantize the network object and return a simulatable quantized network.

```
qNet = quantize(quantObj)
```

```
qNet =
  Quantized dlnetwork with properties:

    Layers: [67x1 nnet.cnn.layer.Layer]
Connections: [74x2 table]
Learnables: [52x3 table]
State: [0x3 table]
InputNames: {'data'}
OutputNames: {'prob'}
Initialized: 1
```

View summary with `summary`.  
 Use the `quantizationDetails` function to extract quantization details.

You can use the `quantizationDetails` method to see that the network is now quantized.

```
qDetails = quantizationDetails(qNet)

qDetails = struct with fields:
    IsQuantized: 1
    TargetLibrary: "cudnn"
    QuantizedLayerNames: [55x1 string]
    QuantizedLearnables: [35x3 table]
```

The `TargetLibrary` field shows that the quantized network emulates the CUDA® Deep Neural Network library (cuDNN).

The `QuantizedLayerNames` field displays a list of layers that have been quantized.

```
qDetails.QuantizedLayerNames(1:5)

ans = 5x1 string
    "conv1"
    "relu_conv1"
    "pool1"
    "fire2-squeeze1x1"
    "fire2-relu_squeeze1x1"
```

The `QuantizedLearnables` field contains additional details on quantized network learnable parameters. In this example, the 2-D convolutional layer, `conv1`, has had the weights scaled and casted to `int8`. The bias is scaled and remains in single precision. The values of quantized learnables are returned as stored integer values.

```
qDetails.QuantizedLearnables
```

```
ans=35x3 table
      Layer      Parameter      Value
-----
"conv1"      "Weights"      {3x3x3x64  int8  }
"conv1"      "Bias"          {1x1x64    single}
"fire2-squeeze1x1"  "Weights"      {1x1x64x16 int8  }
"fire2-squeeze1x1"  "Bias"          {1x1x16    single}
"fire2-expand1x1"   "Weights"      {1x1x16x64 int8  }
"fire2-expand3x3"   "Weights"      {3x3x16x64 int8  }
"fire3-squeeze1x1"  "Weights"      {1x1x128x16 int8  }
"fire3-squeeze1x1"  "Bias"          {1x1x16    single}
"fire3-expand1x1"   "Weights"      {1x1x16x64 int8  }
"fire3-expand3x3"   "Weights"      {3x3x16x64 int8  }
"fire4-squeeze1x1"  "Weights"      {1x1x128x32 int8  }
"fire4-squeeze1x1"  "Bias"          {1x1x32    single}
"fire4-expand1x1"   "Weights"      {1x1x32x128 int8  }
"fire4-expand3x3"   "Weights"      {3x3x32x128 int8  }
"fire5-squeeze1x1"  "Weights"      {1x1x256x32 int8  }
"fire5-squeeze1x1"  "Bias"          {1x1x32    single}
      :
```

You can use the quantized network to emulate how a network quantized for GPU target hardware would perform a classification task.

Make predictions using the original, single-precision floating-point network. To accelerate the computation by compiling and executing a MEX function on the GPU, use the acceleration option "mex" of the predict function.

```
XTest = readall(valData);  
XTest = cat(4,XTest{:});  
XTest = dldarray(gpuArray(single(XTest)), "SSCB");  
TTest = valData.Labels;
```

```
YTestOriginal = predict(dlnet,XTest,Acceleration="mex");
```

Generating MEX for cudnn target.

```
YTestOriginal = onehotdecode(YTestOriginal,classes,3);
```

Make predictions using the quantized INT8 network. Use the acceleration option "mex" of the predict function. MEX acceleration is supported for quantized networks based on quantization objects with ExecutionEnvironment set to GPU.

```
YTestQuantized = predict(qNet,XTest,Acceleration="mex");
```

Generating MEX for cudnn target.

```
YTestQuantized = onehotdecode(YTestQuantized,classes,3);
```

Compute the relative accuracy of the quantized network as compared to the original network.

```
ccrOriginal = mean(squeeze(YTestOriginal) == valData.Labels)
```

```
ccrOriginal = 1
```

```
ccrQuantized = mean(squeeze(YTestQuantized) == valData.Labels)
```

```
ccrQuantized = 1
```

The quantized network shows no drop in accuracy.

## Emulate FPGA Target Behavior for Quantized Network

This example shows how to emulate the behavior of a quantized network for FPGA deployment. Once you quantize your network for an FPGA execution environment, you can emulate the FPGA target behavior without any FPGA hardware. This action allows you to examine your quantized network structure and behavior without generating code for deployment.

Load the pretrained network.

```
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;
```

Define calibration and validation data to use for quantization.

Use the calibration data to collect the dynamic ranges of the weights and biases in the convolution and fully connected layers, the dynamic ranges of the activations in all the layers, and the dynamic ranges of the parameters for some layers. For the best quantization results, the calibration data must be representative of inputs to the network.

Use the validation data to test the network after quantization. Test the network to determine the effects of the limited range and precision of the quantized layers and layer parameters in the network.

This example uses the images in the `logos_dataset` data set. Create an `imageDatastore` object, then split the data into calibration and validation data sets.

```
unzip("logos_dataset.zip");
imageData = imageDatastore(fullfile(pwd,"logos_dataset"),...
    IncludeSubfolders=true,FileExtensions=".JPG",LabelSource="foldernames");
[calData,valData] = splitEachLabel(imageData,0.7,"randomized");
```

Create a `dlquantizer` object and specify the network to quantize. Set the execution environment for the quantized network to FPGA.

```
quantObj = dlquantizer(net,ExecutionEnvironment="FPGA");
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information.

```
calResults = calibrate(quantObj,calData,UseGPU="off");
```

Use the `quantize` function to quantize the network object and return a quantized network for simulation.

```
qNet = quantize(quantObj)
```

```
qNet =
  Quantized DAGNetwork with properties:

    Layers: [22x1 nnet.cnn.layer.Layer]
  Connections: [21x2 table]
    InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Use the `quantizationDetails` function to extract quantization details.

Use the `quantizationDetails` method to extract quantization details.

You can use the `quantizationDetails` function to confirm that the network is now quantized. The `TargetLibrary` field shows that the quantized network emulates an FPGA target.

```
qDetails = quantizationDetails(qNet)

qDetails = struct with fields:
    IsQuantized: 1
    TargetLibrary: "fpga"
  QuantizedLayerNames: [17x1 string]
  QuantizedLearnables: [14x3 table]
```

The `QuantizedLayerNames` field displays a list of quantized layers.

`qDetails.QuantizedLayerNames`

```
ans = 17x1 string
"conv_1"
"relu_1"
"maxpool_1"
"conv_2"
"relu_2"
"maxpool_2"
"conv_3"
"relu_3"
"maxpool_3"
"conv_4"
"relu_4"
"maxpool_4"
"fc_1"
"relu_5"
"fc_2"
"relu_6"
"fc_3"
```

The `QuantizedLearnables` field contains additional details about the quantized network learnable parameters. In this example, the 2-D convolutional layers and fully connected layers have their weights scaled and cast to `int8`. The bias is scaled and remains in `int32`. The `quantizationDetails` function returns the values of the quantized learnables as stored integer values.

`qDetails.QuantizedLearnables`

```
ans=14x3 table
   Layer      Parameter      Value
   -----      -
"conv_1"    "Weights"    {5x5x3x96      int8 }
"conv_1"    "Bias"       {1x1x96        int32}
"conv_2"    "Weights"    {3x3x96x128    int8 }
"conv_2"    "Bias"       {1x1x128       int32}
"conv_3"    "Weights"    {3x3x128x384   int8 }
"conv_3"    "Bias"       {1x1x384       int32}
"conv_4"    "Weights"    {3x3x384x128   int8 }
"conv_4"    "Bias"       {1x1x128       int32}
"fc_1"      "Weights"    {5x5x128x2048  int8 }
"fc_1"      "Bias"       {1x1x2048      int32}
"fc_2"      "Weights"    {1x1x2048x2048 int8 }
"fc_2"      "Bias"       {1x1x2048      int32}
"fc_3"      "Weights"    {1x1x2048x32   int8 }
"fc_3"      "Bias"       {1x1x32        int32}
```

You can use the quantized network to emulate a network quantized for FPGA target hardware performing a classification task.

```
ypred = qNet.classify(valData);
ccr = mean(ypred == valData.Labels)

ccr = 1
```



## Input Arguments

### **quantObj — Network to quantize**

`dlquantizer` object

`dlquantizer` object containing the network to quantize, calibrated using the `calibrate` object function. The “ExecutionEnvironment” on page 1-0 must be set to 'GPU' 'FPGA', or 'MATLAB'.

### **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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `quantizedNetwork = quantize(quantObj, 'ExponentScheme', 'Histogram')`

### **ExponentScheme — Exponent selection scheme**

'MinMax' (default) | 'Histogram'

Exponent selection scheme, specified as one of these values:

- 'MinMax' — Evaluate the exponent based on the range information in the calibration statistics and avoid overflows.
- 'Histogram' — Distribution-based scaling which evaluates the exponent to best fit the calibration data.

Example: 'ExponentScheme', 'Histogram'

## Output Arguments

### **quantizedNetwork — Quantized neural network**

`dlnetwork` object | `DAGNetwork` object | `yolov2objectDetector` object | `yolov3objectDetector` object | `yolov4objectDetector` object | `ssdobjectDetector` object

Quantized neural network, returned as a `dlnetwork`, `DAGNetwork`, `yolov2objectDetector`, `yolov3objectDetector`, `yolov4objectDetector`, or a `ssdobjectDetector` object.

## Limitations

- The `quantize` function does not support quantization of networks using `dlquantizer` objects with “ExecutionEnvironment” on page 1-0 set to 'CPU'.
- Code generation does not support quantized deep neural networks produced by the `quantize` function.

## Version History

Introduced in R2022a

**R2022a: Quantize `dlquantizer` objects calibrated in R2022a and later**

The `quantize` function supports quantization of `dlquantizer` objects that are calibrated in R2022a and later.

## **R2022b: quantize support for FPGA execution environment**

Use the `quantize` method to create a simulatable quantized network when the `ExecutionEnvironment` property of `dlquantizer` is set to `FPGA`. The simulatable quantized network enables visibility of the quantized layers, weights, and biases of the network, as well as simulatable quantized inference behavior.

## **R2023a: Quantize dlquantizer objects that specify a dlnetwork**

The `quantize` function now supports quantization of `dlnetwork` objects using a calibration `dlquantizer` object.

## **See Also**

### **Apps**

**Deep Network Quantizer**

### **Functions**

`dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate` | `quantizationDetails` | `estimateNetworkMetrics`

### **Topics**

“Quantization of Deep Neural Networks”

“Quantize Layers in Object Detectors and Generate CUDA Code”

“Classify Images on FPGA Using Quantized Neural Network” (Deep Learning HDL Toolbox)

“Generate INT8 Code for Deep Learning Network on Raspberry Pi” (MATLAB Coder)

# 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 =
```

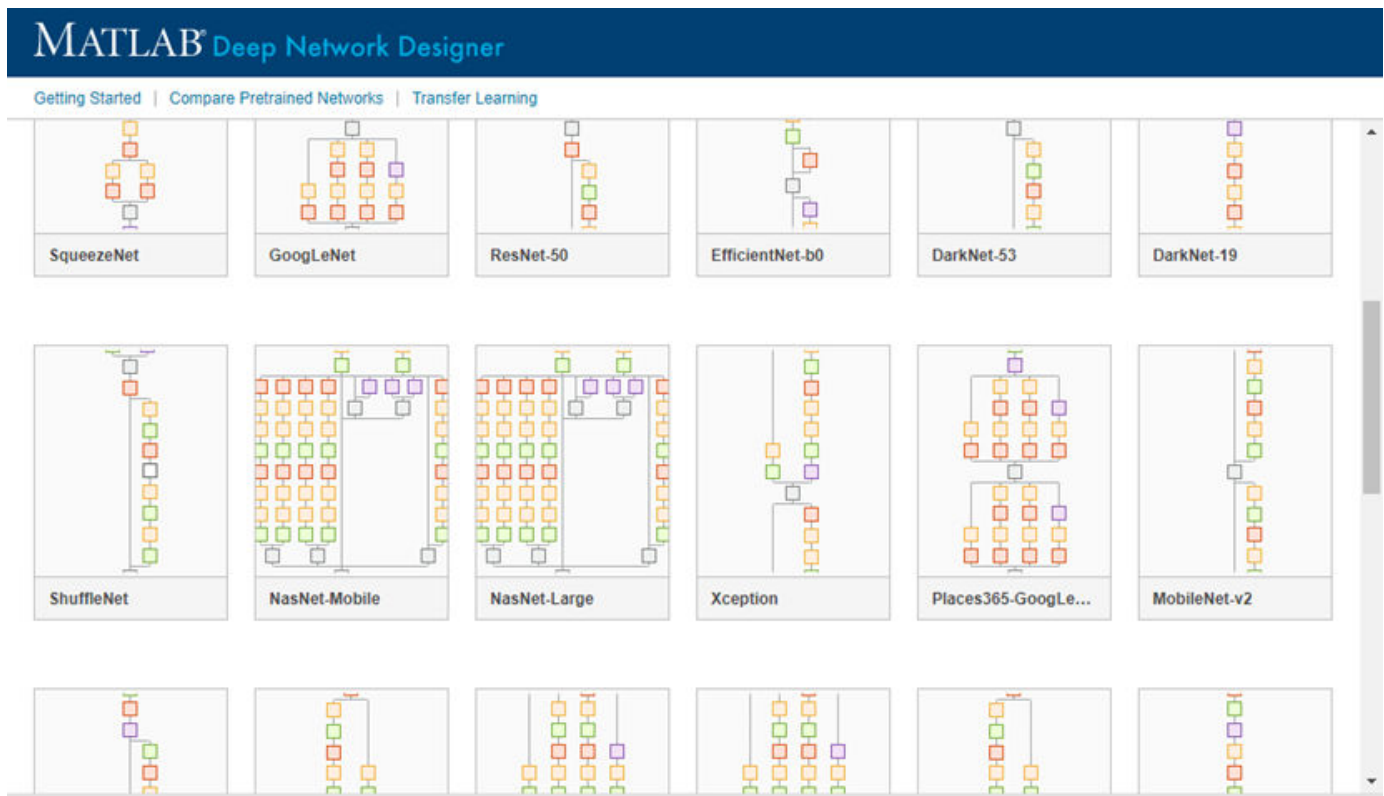
```
SeriesNetwork with properties:
```

```
Layers: [41x1 nnet.cnn.layer.Layer]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(vgg16)
```

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



If you need to download a neural network, pause on the desired neural 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: [41x1 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 padding
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 padding
8	'relu2_1'	ReLU	ReLU
9	'conv2_2'	Convolution	128 3x3x128 convolutions with stride [1 1] and padding
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 padding
13	'relu3_1'	ReLU	ReLU
14	'conv3_2'	Convolution	256 3x3x256 convolutions with stride [1 1] and padding
15	'relu3_2'	ReLU	ReLU
16	'conv3_3'	Convolution	256 3x3x256 convolutions with stride [1 1] and padding
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 padding
20	'relu4_1'	ReLU	ReLU
21	'conv4_2'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding
22	'relu4_2'	ReLU	ReLU
23	'conv4_3'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding
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 padding
27	'relu5_1'	ReLU	ReLU
28	'conv5_2'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding
29	'relu5_2'	ReLU	ReLU
30	'conv5_3'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding
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 = 10x1 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.

## Version History

Introduced in R2017a

## 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/](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”

## 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 =
```

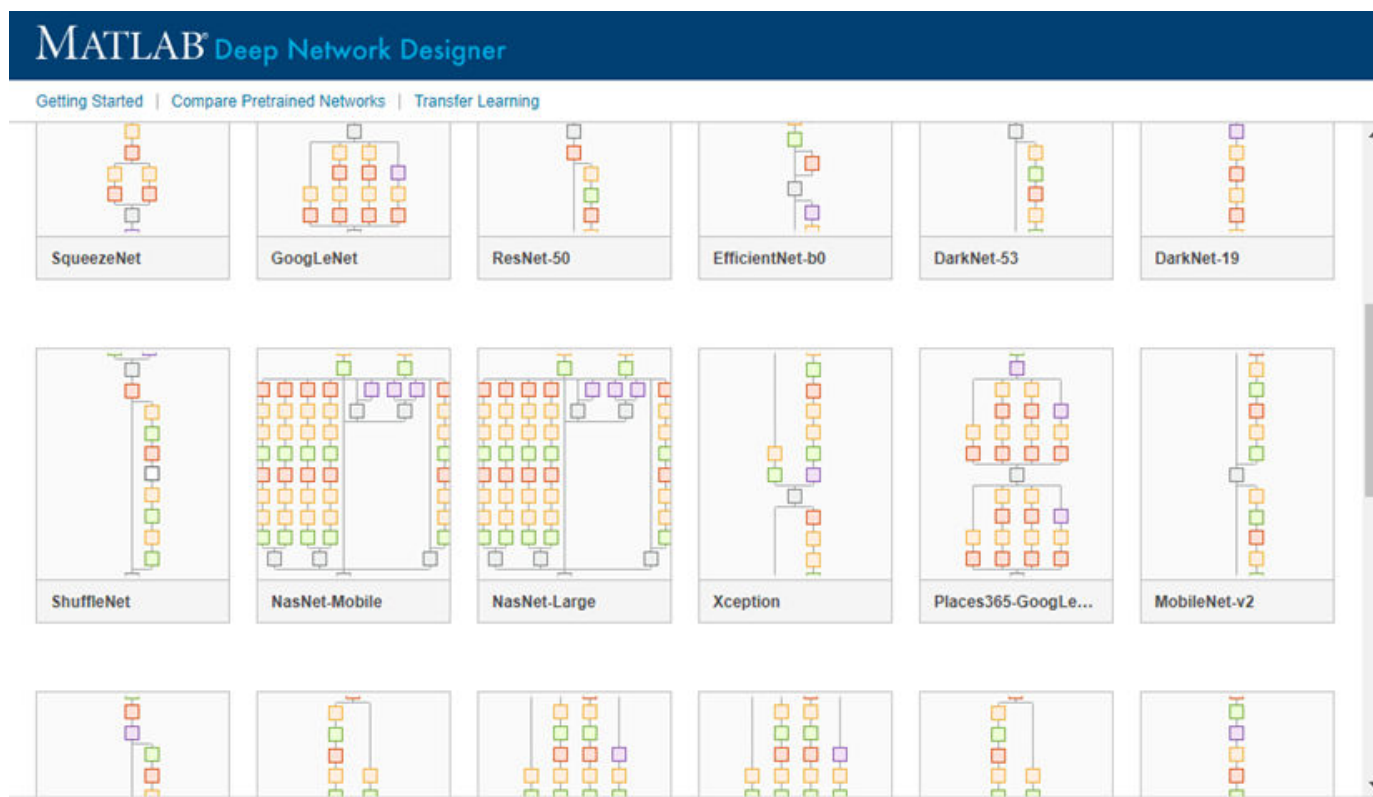
```
SeriesNetwork with properties:
```

```
Layers: [47x1 nnet.cnn.layer.Layer]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(vgg19)
```

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



If you need to download a neural network, pause on the desired neural 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: [47x1 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 padding
     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 padding
     8 'relu2_1'    ReLU            ReLU
     9 'conv2_2'    Convolution     128 3x3x128 convolutions with stride [1 1] and padding
    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 padding
    13 'relu3_1'    ReLU            ReLU
    14 'conv3_2'    Convolution     256 3x3x256 convolutions with stride [1 1] and padding
    15 'relu3_2'    ReLU            ReLU
    16 'conv3_3'    Convolution     256 3x3x256 convolutions with stride [1 1] and padding
    17 'relu3_3'    ReLU            ReLU
    18 'conv3_4'    Convolution     256 3x3x256 convolutions with stride [1 1] and padding
    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 padding
    22 'relu4_1'    ReLU            ReLU
    23 'conv4_2'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    24 'relu4_2'    ReLU            ReLU
    25 'conv4_3'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    26 'relu4_3'    ReLU            ReLU
    27 'conv4_4'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    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 padding
    31 'relu5_1'    ReLU            ReLU
    32 'conv5_2'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    33 'relu5_2'    ReLU            ReLU
    34 'conv5_3'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    35 'relu5_3'    ReLU            ReLU
    36 'conv5_4'    Convolution     512 3x3x512 convolutions with stride [1 1] and padding
    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.

## Version History

Introduced in R2017a

## 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/](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”

# 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: [24x1 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: [24x1 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.

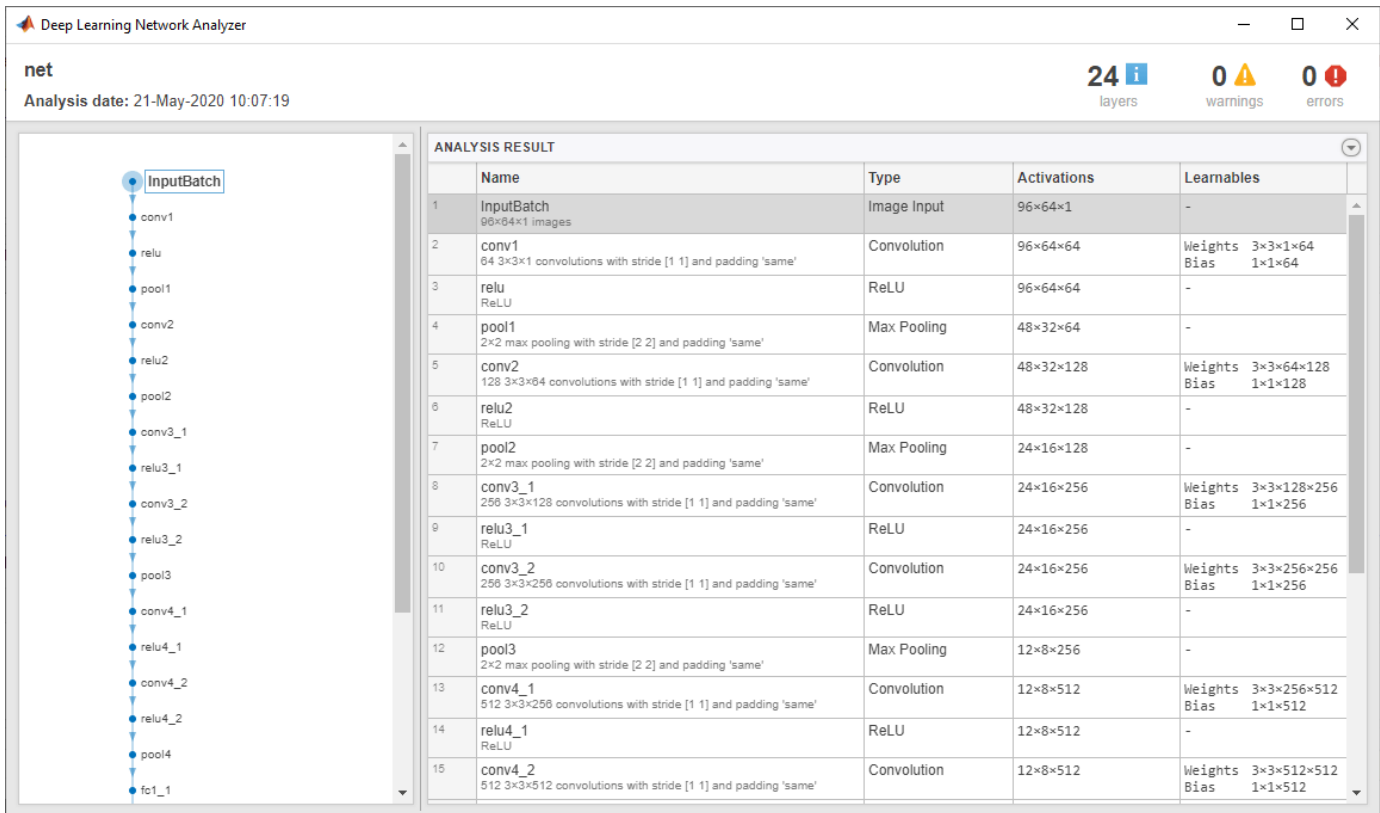
```
net.Layers
```

```
ans =
  24x1 Layer array with layers:

    1  'InputBatch'      Image Input      96x64x1 images
    2  'conv1'           Convolution      64 3x3x1 convolutions with stride [1 1] and padding
    3  'relu'            ReLU
    4  'pool1'           Max Pooling      2x2 max pooling with stride [2 2] and padding
    5  'conv2'           Convolution      128 3x3x64 convolutions with stride [1 1] and padding
    6  'relu2'           ReLU
    7  'pool2'           Max Pooling      2x2 max pooling with stride [2 2] and padding
    8  'conv3_1'         Convolution      256 3x3x128 convolutions with stride [1 1] and padding
    9  'relu3_1'         ReLU
   10  'conv3_2'         Convolution      256 3x3x256 convolutions with stride [1 1] and padding
   11  'relu3_2'         ReLU
   12  'pool3'           Max Pooling      2x2 max pooling with stride [2 2] and padding
   13  'conv4_1'         Convolution      512 3x3x256 convolutions with stride [1 1] and padding
   14  'relu4_1'         ReLU
   15  'conv4_2'         Convolution      512 3x3x512 convolutions with stride [1 1] and padding
   16  'relu4_2'         ReLU
   17  'pool4'           Max Pooling      2x2 max pooling with stride [2 2] and padding
   18  'fc1_1'           Fully Connected  4096 fully connected layer
   19  'relu5_1'         ReLU
   20  'fc1_2'           Fully Connected  4096 fully connected layer
   21  'relu5_2'         ReLU
   22  'fc2'             Fully Connected  128 fully connected layer
   23  'EmbeddingBatch'  ReLU
   24  'regressionoutput' Regression Output mean-squared-error
```

Use `analyzeNetwork` to visually explore the network.

```
analyzeNetwork(net)
```



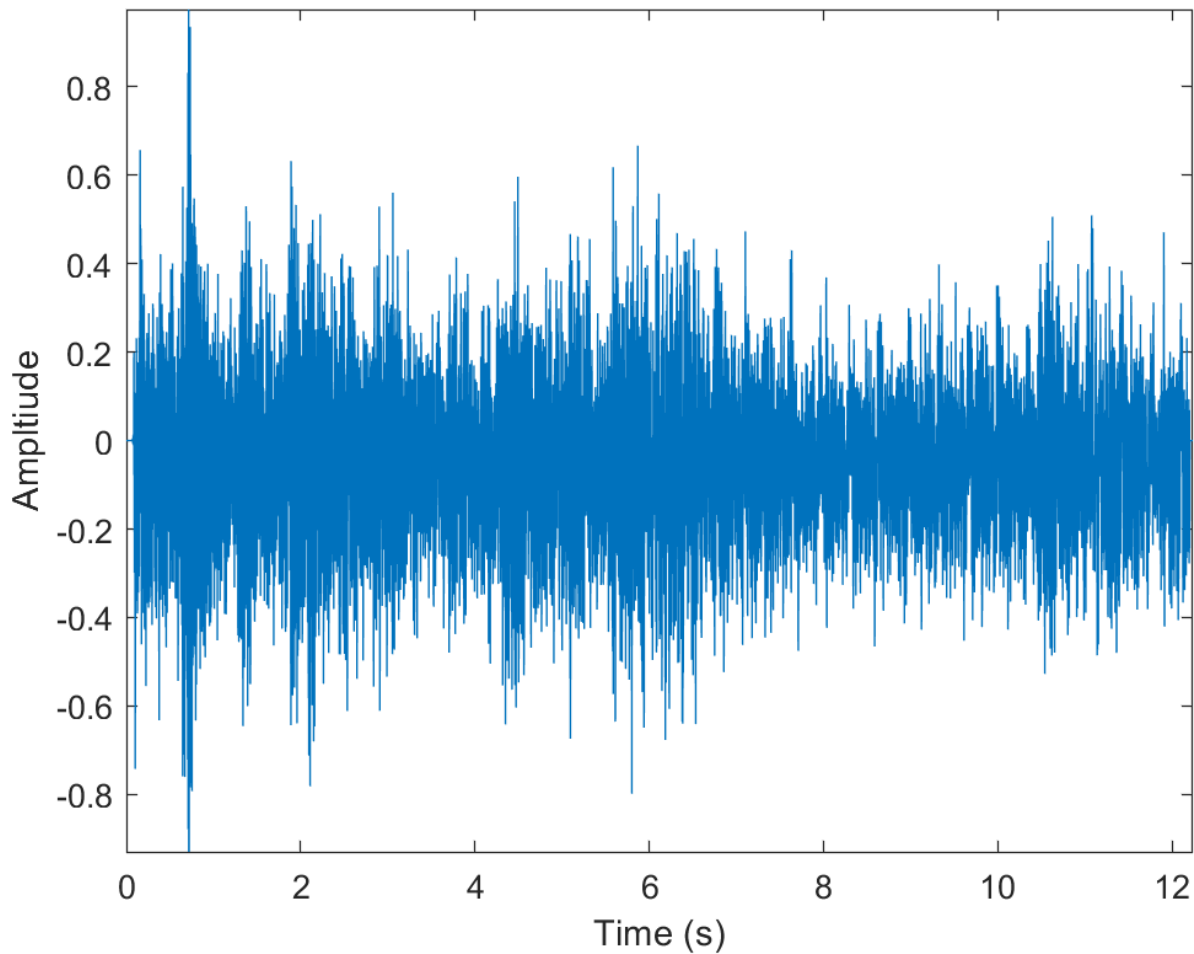
## Extract Features Using VGGish

Read in an audio signal to extract feature embeddings from it.

```
[audioIn, fs] = audioread(Ambiance-16-44p1-...);
```

Plot and listen to the audio signal.

```
t = (0:numel(audioIn)-1)/fs;
plot(t, audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis tight
```

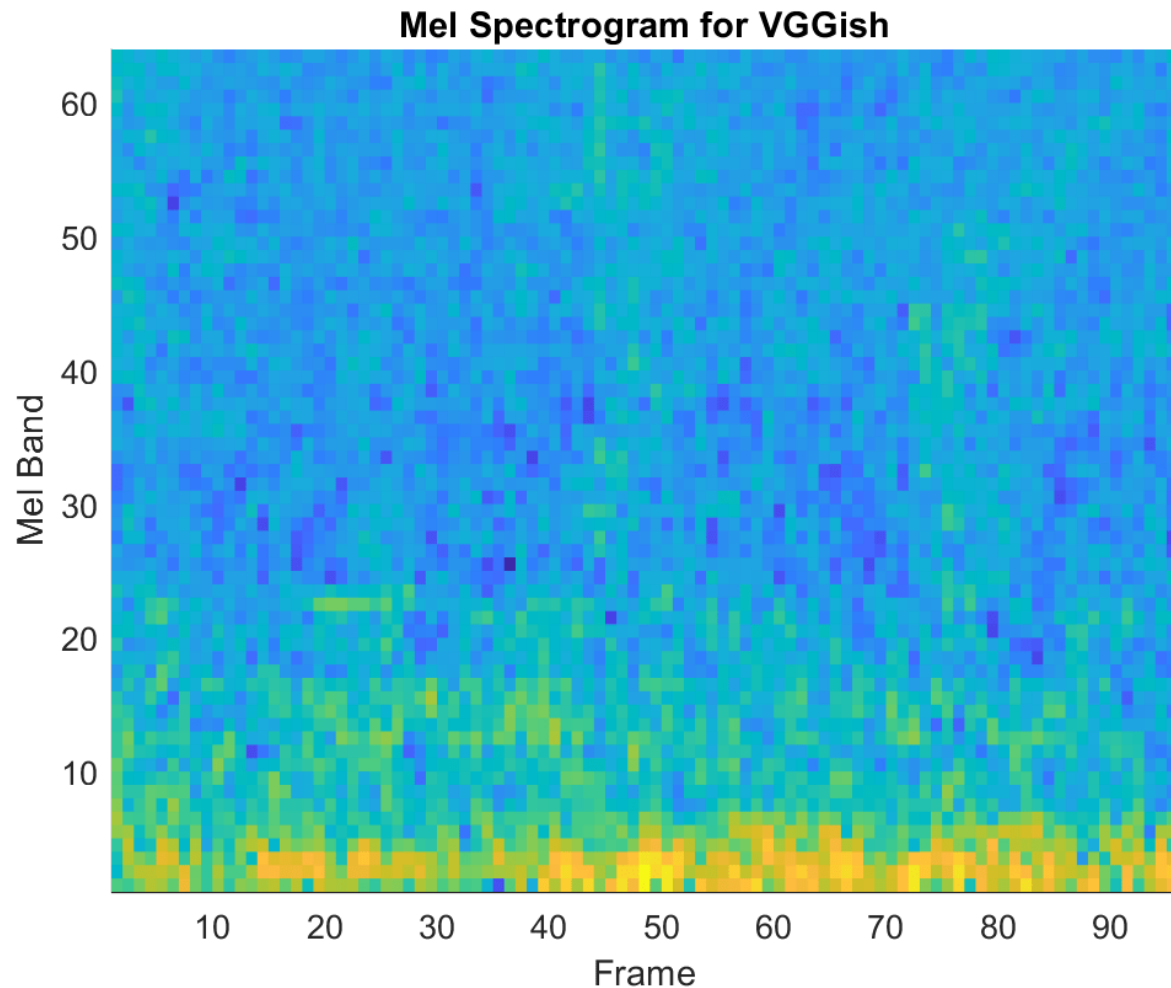


```
% To play the sound, call soundsc(audioIn,fs)
```

VGGish requires you to preprocess the audio signal to match the input format used to train the network. The preprocessing steps include resampling the audio signal and computing an array of mel spectrograms. To learn more about mel spectrograms, see `melSpectrogram` (Audio Toolbox). Use `vggishPreprocess` to preprocess the signal and extract the mel spectrograms to be passed to VGGish. Visualize one of these spectrograms chosen at random.

```
spectrograms = vggishPreprocess(audioIn,fs);  
  
arbitrarySpect = spectrograms(:,:,1,randi(size(spectrograms,4)));  
surf(arbitrarySpect,EdgeColor="none")  
view(90,-90)  
xlabel("Mel Band")  
ylabel("Frame")  
title("Mel Spectrogram for VGGish")  
axis tight
```





Create a VGGish neural network. Using the `vggish` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

```
net = vggish;
```

Call `predict` with the network on the preprocessed mel spectrogram images to extract feature embeddings. 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,spectrograms);  
[numFrames,numFeatures] = size(features)
```

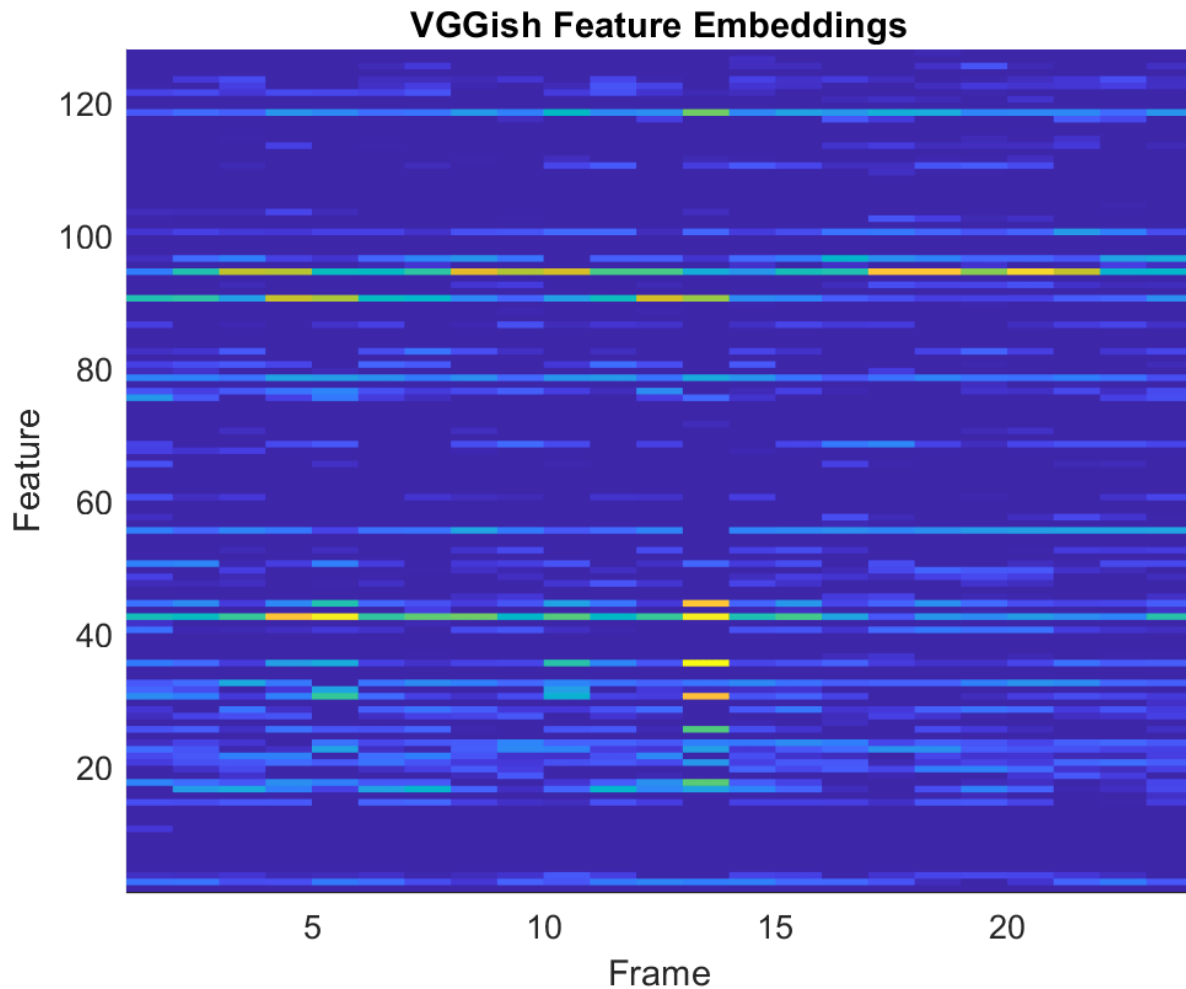
```
numFrames = 24
```

```
numFeatures = 128
```

Visualize the VGGish feature embeddings.

```
surf(features,EdgeColor="none")  
view([90 -90])
```

```
xlabel("Feature")  
ylabel("Frame")  
title("VGGish Feature Embeddings")  
axis tight
```



### 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).

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio", "ESC-10.zip");  
unzip(downloadFolder, tempdir)  
dataLocation = fullfile(tempdir, "ESC-10");
```

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(dataLocation,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 = height(labelTable);
```

Call `splitEachLabel` (Audio Toolbox) to split the data set into train 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. Use `vggishPreprocess` (Audio Toolbox) to extract the spectrograms from the train set. There are multiple spectrograms for each audio signal. Replicate the labels so that they are in one-to-one correspondence with the spectrograms.

```
overlapPercentage = 75 ;

trainFeatures = [];
trainLabels = [];
while hasdata(adsTrain)
    [audioIn,fileInfo] = read(adsTrain);
    features = vggishPreprocess(audioIn,fileInfo.SampleRate,OverlapPercentage=overlapPercentage)
    numSpectrograms = size(features,4);
    trainFeatures = cat(4,trainFeatures,features);
    trainLabels = cat(2,trainLabels, repelem(fileInfo.Label,numSpectrograms));
end
```

Extract spectrograms from the validation set and replicate the labels.

```
validationFeatures = [];
validationLabels = [];
segmentsPerFile = zeros(numel(adsValidation.Files), 1);
idx = 1;
while hasdata(adsValidation)
    [audioIn,fileInfo] = read(adsValidation);
    features = vggishPreprocess(audioIn,fileInfo.SampleRate,OverlapPercentage=overlapPercentage)
    numSpectrograms = size(features,4);
    validationFeatures = cat(4,validationFeatures,features);
    validationLabels = cat(2,validationLabels, repelem(fileInfo.Label,numSpectrograms));

    segmentsPerFile(idx) = numSpectrograms;
    idx = idx + 1;
end
```

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. Set the `WeightLearnRateFactor` and `BiasLearnRateFactor` of the new fully connected layer to 10 so that learning is faster in the new layer than in the transferred layers.

```
lgraph = addLayers(lgraph,[ ...
    fullyConnectedLayer(numClasses,Name="FCFinal",WeightLearnRateFactor=10,BiasLearnRateFactor=10)
    softmaxLayer(Name="softmax")
    classificationLayer(Name="classOut")]);
```

Use `connectLayers` to append the fully connected, softmax, and classification layers to the layer graph.

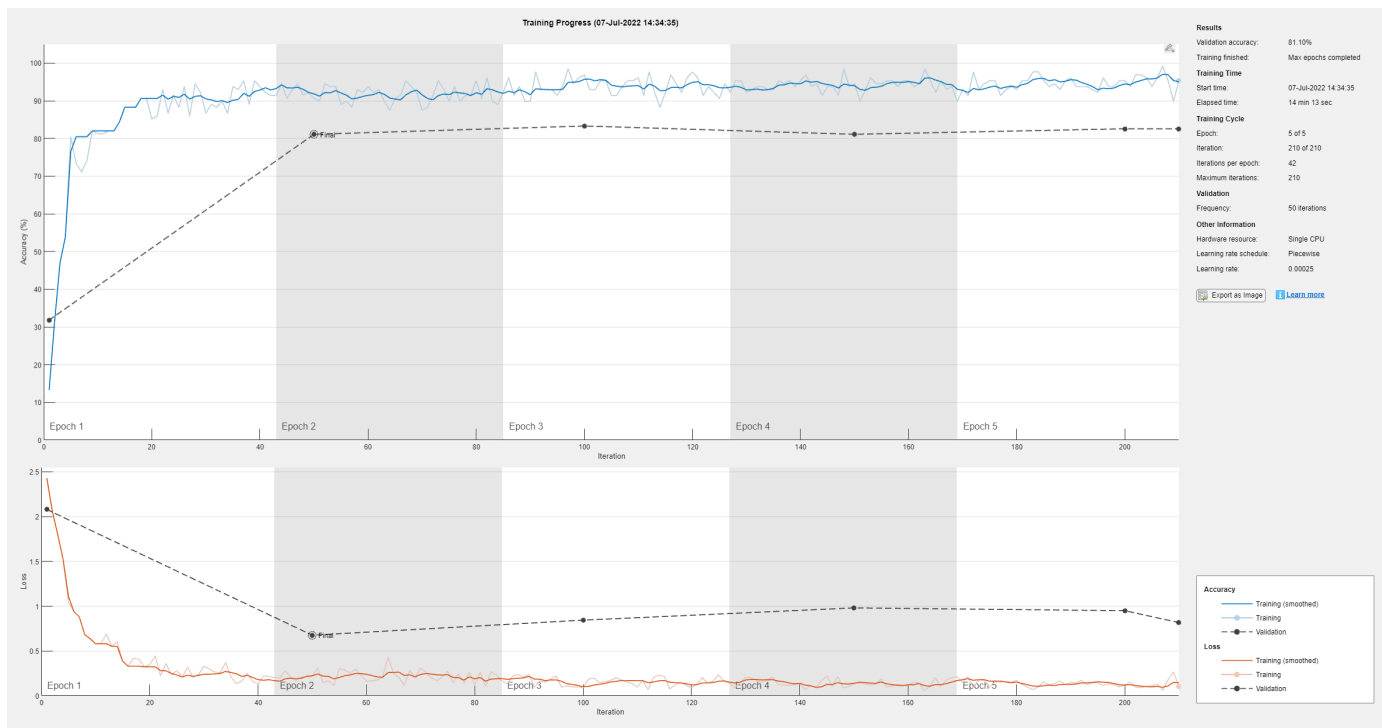
```
lgraph = connectLayers(lgraph,"EmbeddingBatch","FCFinal");
```

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, ...
    OutputNetwork="best-validation-loss", ...
    Verbose=false, ...
    Plots="training-progress");
```

To train the network, use `trainNetwork`.

```
[trainedNet, netInfo] = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```



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]);
confusionchart(adsValidation.Labels,validationPredictionsPerFile, ...
    Title=sprintf("Confusion Matrix for Validation Data \nAccuracy = %0.2f %%",mean(validationPredictionsPerFile)), ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized")
```

**Confusion Matrix for Validation Data**  
Accuracy = 87.50 %

True Class	chainsaw	clock_tick	crackling_fire	crying_baby	dog	helicopter	rain	rooster	sea_waves	sneezing		
chainsaw	7					1					87.5%	12.5%
clock_tick		8									100.0%	
crackling_fire	1		6					1			75.0%	25.0%
crying_baby				8							100.0%	
dog					1	6					75.0%	25.0%
helicopter	1		1			6					75.0%	25.0%
rain							8				100.0%	
rooster								8			100.0%	
sea_waves									8		100.0%	
sneezing					3					5	62.5%	37.5%

77.8%	100.0%	85.7%	66.7%	100.0%	85.7%	88.9%	100.0%	100.0%	83.3%
22.2%		14.3%	33.3%		14.3%	11.1%			16.7%
chainsaw	clock_tick	crackling_fire	crying_baby	dog	helicopter	rain	rooster	sea_waves	sneezing

Predicted Class

## Output Arguments

### net — Pretrained VGGish neural network

SeriesNetwork object

Pretrained VGGish neural network, returned as a SeriesNetwork object.

## Version History

Introduced in R2020b

## 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

**Signal Labeler**

### Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

### Functions

`audioFeatureExtractor` | `classifySound` | `melSpectrogram` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`



# 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 =
```

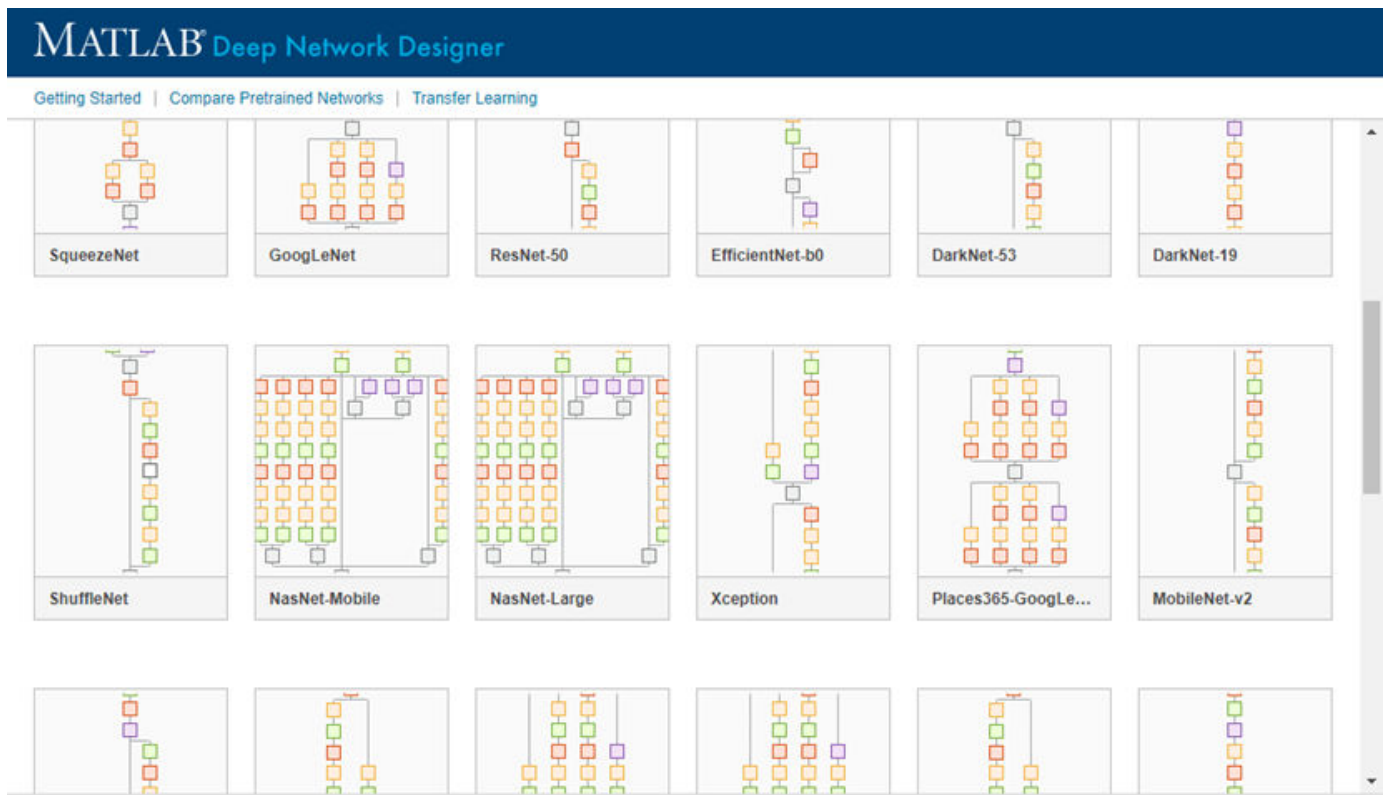
```
DAGNetwork with properties:
```

```
Layers: [171x1 nnet.cnn.layer.Layer]  
Connections: [182x2 table]
```

Visualize the network using Deep Network Designer.

```
deepNetworkDesigner(xception)
```

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



If you need to download a neural network, pause on the desired neural 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.

**Version History**

Introduced in R2019a

**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”

# 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: [86x1 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: [86x1 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           96x64x1 images
   2  'conv2d'           Convolution           32 3x3x1 convolutions with stride
   3  'b'                 Batch Normalization  Batch normalization with 32 channels
   4  'activation'       ReLU                  ReLU
   5  'depthwise_conv2d' Grouped Convolution  32 groups of 1 3x3x1 convolutions
   6  'L11'              Batch Normalization  Batch normalization with 32 channels
   7  'activation_1'     ReLU                  ReLU
   8  'conv2d_1'         Convolution           64 1x1x32 convolutions with stride
   9  'L12'              Batch Normalization  Batch normalization with 64 channels
  10  'activation_2'     ReLU                  ReLU
  11  'depthwise_conv2d_1' Grouped Convolution  64 groups of 1 3x3x1 convolutions
  12  'L21'              Batch Normalization  Batch normalization with 64 channels
  13  'activation_3'     ReLU                  ReLU
  14  'conv2d_2'         Convolution           128 1x1x64 convolutions with stride
  15  'L22'              Batch Normalization  Batch normalization with 128 channels
  16  'activation_4'     ReLU                  ReLU
  17  'depthwise_conv2d_2' Grouped Convolution  128 groups of 1 3x3x1 convolutions
  18  'L31'              Batch Normalization  Batch normalization with 128 channels
  19  'activation_5'     ReLU                  ReLU
  20  'conv2d_3'         Convolution           128 1x1x128 convolutions with stride
  21  'L32'              Batch Normalization  Batch normalization with 128 channels
  22  'activation_6'     ReLU                  ReLU
  23  'depthwise_conv2d_3' Grouped Convolution  128 groups of 1 3x3x1 convolutions
  24  'L41'              Batch Normalization  Batch normalization with 128 channels
  25  'activation_7'     ReLU                  ReLU
  26  'conv2d_4'         Convolution           256 1x1x128 convolutions with stride
  27  'L42'              Batch Normalization  Batch normalization with 256 channels
  28  'activation_8'     ReLU                  ReLU
  29  'depthwise_conv2d_4' Grouped Convolution  256 groups of 1 3x3x1 convolutions
  30  'L51'              Batch Normalization  Batch normalization with 256 channels
  31  'activation_9'     ReLU                  ReLU
  32  'conv2d_5'         Convolution           256 1x1x256 convolutions with stride
  33  'L52'              Batch Normalization  Batch normalization with 256 channels
  34  'activation_10'    ReLU                  ReLU
  35  'depthwise_conv2d_5' Grouped Convolution  256 groups of 1 3x3x1 convolutions
  36  'L61'              Batch Normalization  Batch normalization with 256 channels
  37  'activation_11'    ReLU                  ReLU
  38  'conv2d_6'         Convolution           512 1x1x256 convolutions with stride
```

39	'L62'	Batch Normalization	Batch normalization with 512 channels
40	'activation_12'	ReLU	ReLU
41	'depthwise_conv2d_6'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
42	'L71'	Batch Normalization	Batch normalization with 512 channels
43	'activation_13'	ReLU	ReLU
44	'conv2d_7'	Convolution	512 1x1x512 convolutions with stride 1
45	'L72'	Batch Normalization	Batch normalization with 512 channels
46	'activation_14'	ReLU	ReLU
47	'depthwise_conv2d_7'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
48	'L81'	Batch Normalization	Batch normalization with 512 channels
49	'activation_15'	ReLU	ReLU
50	'conv2d_8'	Convolution	512 1x1x512 convolutions with stride 1
51	'L82'	Batch Normalization	Batch normalization with 512 channels
52	'activation_16'	ReLU	ReLU
53	'depthwise_conv2d_8'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
54	'L91'	Batch Normalization	Batch normalization with 512 channels
55	'activation_17'	ReLU	ReLU
56	'conv2d_9'	Convolution	512 1x1x512 convolutions with stride 1
57	'L92'	Batch Normalization	Batch normalization with 512 channels
58	'activation_18'	ReLU	ReLU
59	'depthwise_conv2d_9'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
60	'L101'	Batch Normalization	Batch normalization with 512 channels
61	'activation_19'	ReLU	ReLU
62	'conv2d_10'	Convolution	512 1x1x512 convolutions with stride 1
63	'L102'	Batch Normalization	Batch normalization with 512 channels
64	'activation_20'	ReLU	ReLU
65	'depthwise_conv2d_10'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
66	'L111'	Batch Normalization	Batch normalization with 512 channels
67	'activation_21'	ReLU	ReLU
68	'conv2d_11'	Convolution	512 1x1x512 convolutions with stride 1
69	'L112'	Batch Normalization	Batch normalization with 512 channels
70	'activation_22'	ReLU	ReLU
71	'depthwise_conv2d_11'	Grouped Convolution	512 groups of 1 3x3x1 convolutions with stride 1
72	'L121'	Batch Normalization	Batch normalization with 512 channels
73	'activation_23'	ReLU	ReLU
74	'conv2d_12'	Convolution	1024 1x1x512 convolutions with stride 1
75	'L122'	Batch Normalization	Batch normalization with 1024 channels
76	'activation_24'	ReLU	ReLU
77	'depthwise_conv2d_12'	Grouped Convolution	1024 groups of 1 3x3x1 convolutions with stride 1
78	'L131'	Batch Normalization	Batch normalization with 1024 channels
79	'activation_25'	ReLU	ReLU
80	'conv2d_13'	Convolution	1024 1x1x1024 convolutions with stride 1
81	'L132'	Batch Normalization	Batch normalization with 1024 channels
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 'Sound'

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.

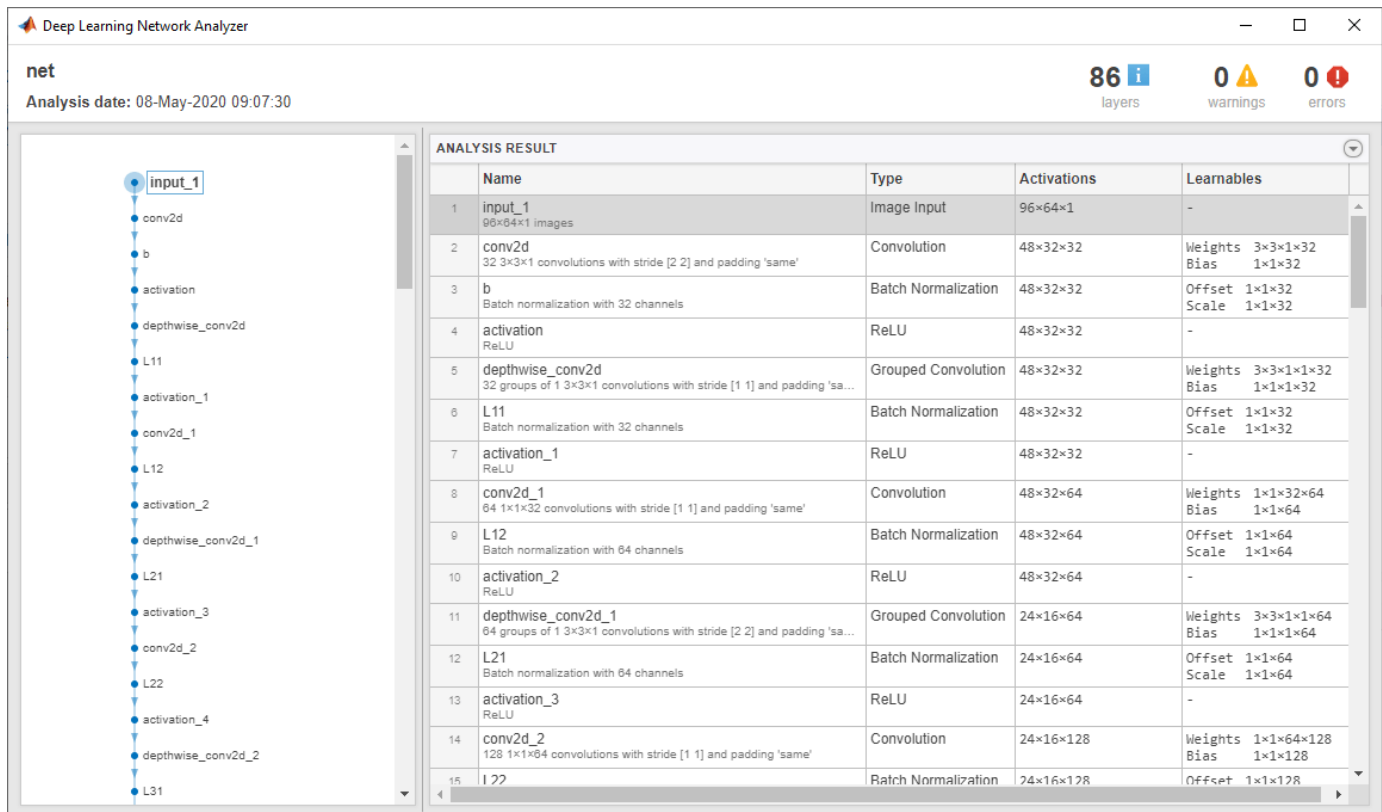
```
net.Layers(end).Classes(1:10)
```

```
ans = 10x1 categorical
    Speech
    Child speech, kid speaking
```

Conversation  
 Narration, monologue  
 Babbling  
 Speech synthesizer  
 Shout  
 Bellow  
 Whoop  
 Yell

Use analyzeNetwork to visually explore the network.

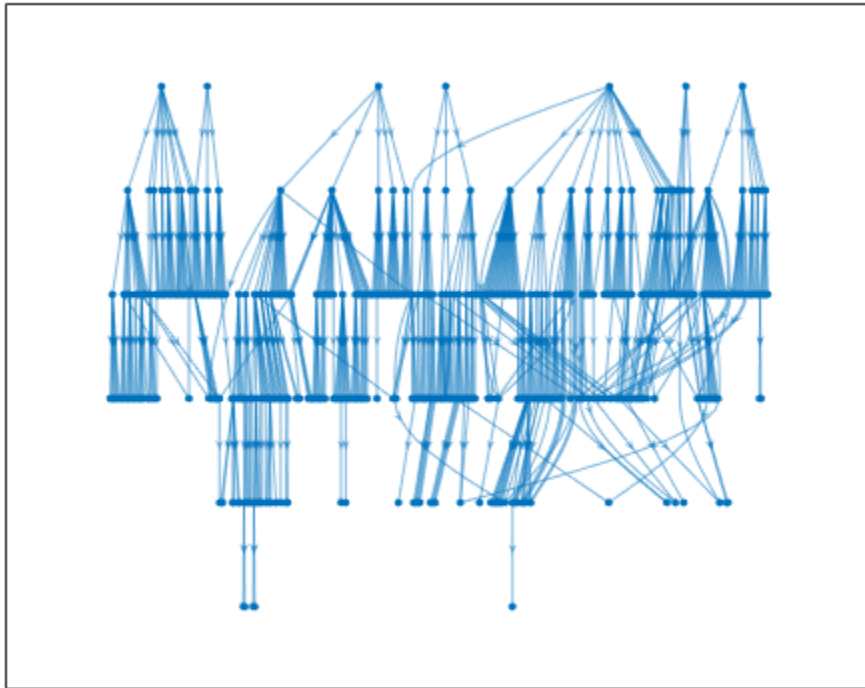
```
analyzeNetwork(net)
```



YAMNet was released with a corresponding sound class ontology, which you can explore using the `yamnetGraph` (Audio Toolbox) object.

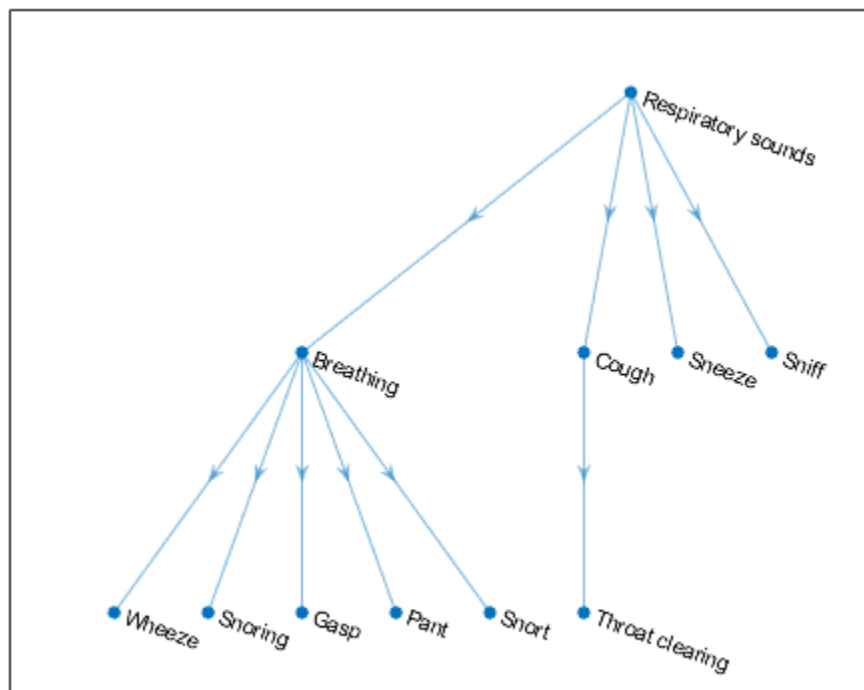
```
ygraph = yamnetGraph;  
p = plot(ygraph);  
layout(p, 'layered')
```





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)
```



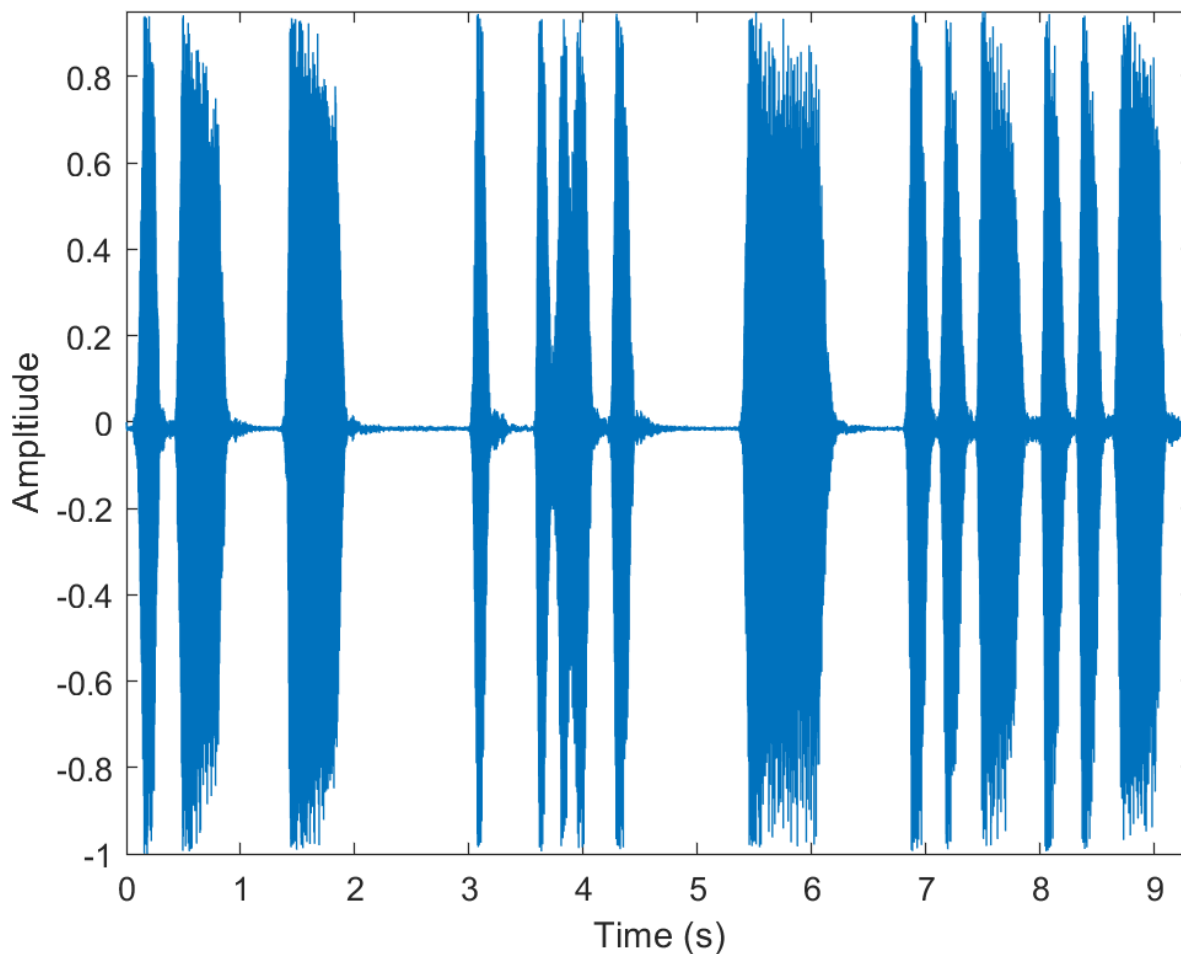
### Classify Sounds Using YAMNet

Read in an audio signal to classify it.

```
[audioIn,fs] = audioread(TrainWhistle-16-44...);
```

Plot and listen to the audio signal.

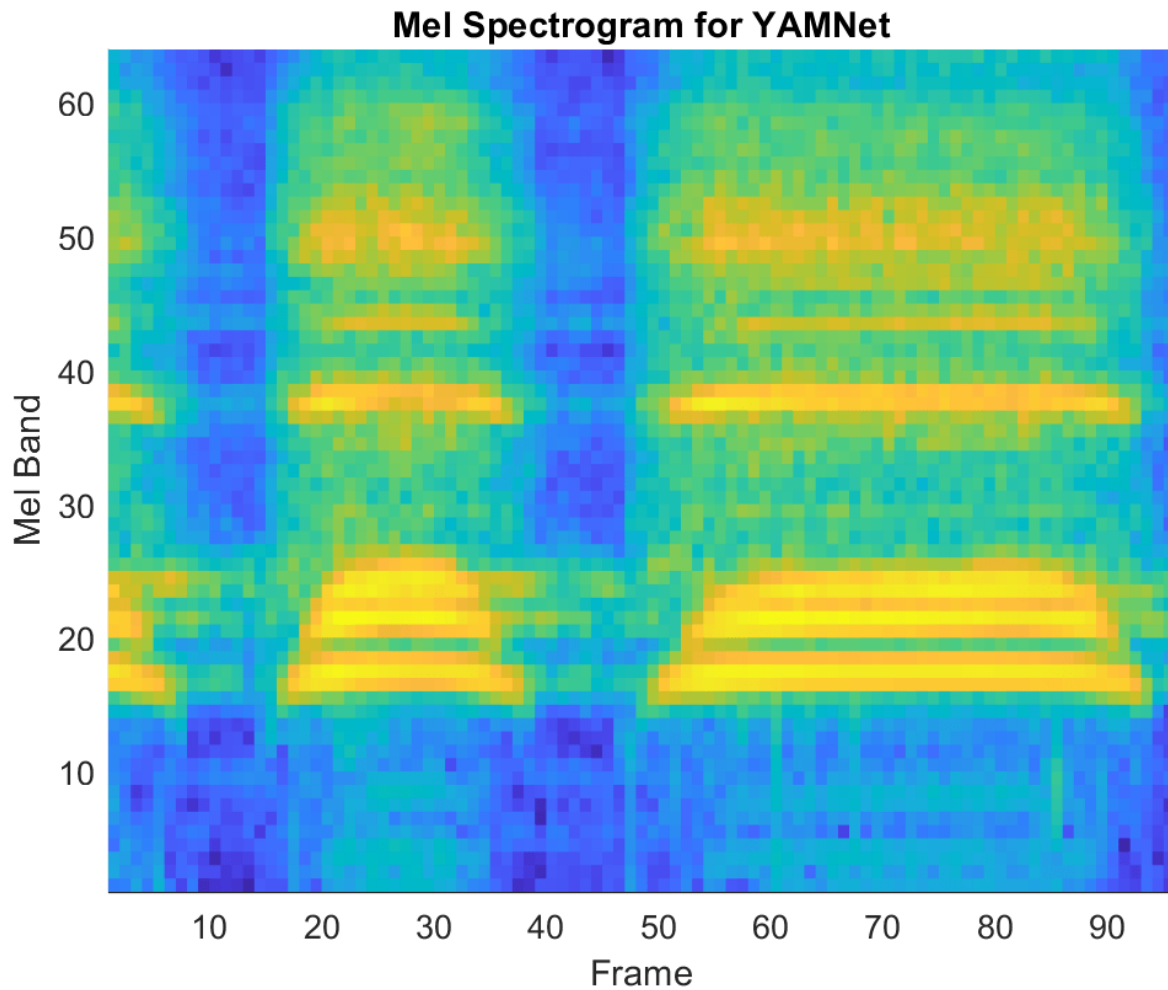
```
t = (0:numel(audioIn)-1)/fs;  
plot(t,audioIn)  
xlabel("Time (s)")  
ylabel("Amplitude")  
axis tight
```



```
% To play the sound, call soundsc(audioIn,fs)
```

YAMNet requires you to preprocess the audio signal to match the input format used to train the network. The preprocessing steps include resampling the audio signal and computing an array of mel spectrograms. To learn more about mel spectrograms, see `melSpectrogram` (Audio Toolbox). Use `yamnetPreprocess` to preprocess the signal and extract the mel spectrograms to be passed to YAMNet. Visualize one of these spectrograms chosen at random.

```
spectrograms = yamnetPreprocess(audioIn,fs);  
  
arbitrarySpect = spectrograms(:,:,1,randi(size(spectrograms,4)));  
surf(arbitrarySpect,EdgeColor="none")  
view([90 -90])  
xlabel("Mel Band")  
ylabel("Frame")  
title("Mel Spectrogram for YAMNet")  
axis tight
```



Create a YAMNet neural network. Using the `yamnet` function requires installing the pretrained YAMNet network. If the network is not installed, the function provides a link to download the pretrained model. Call `classify` with the network on the preprocessed mel spectrogram images.

```
net = yamnet;  
classes = classify(net,spectrograms);
```

Calling `classify` returns a label for each of the spectrogram images in the input. Classify the sound as the most frequently occurring label in the output of `classify`.

```
mySound = mode(classes)  
  
mySound = categorical  
Whistle
```

## Transfer Learning Using YAMNet

Download and unzip the air compressor data set [1] on page 1-2022. 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);
```

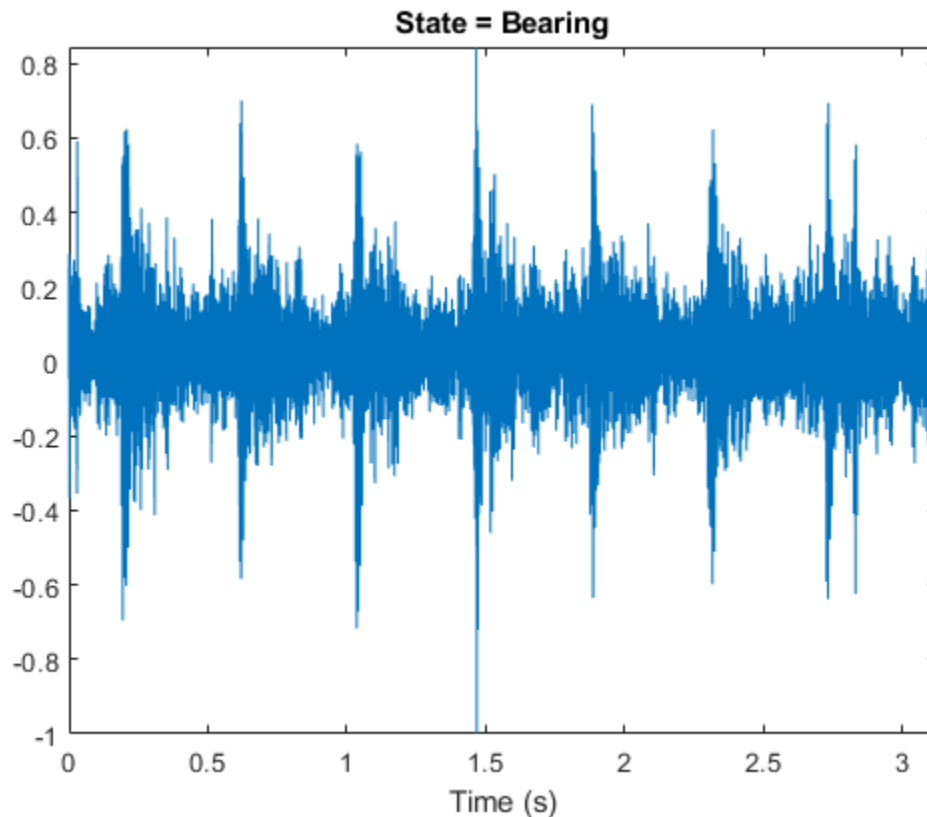
Read an audio file from the datastore and save the sample rate for later use. Reset the datastore to return the read pointer to the beginning of the data set. Listen to the audio signal and plot the signal in the time domain.

```
[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);

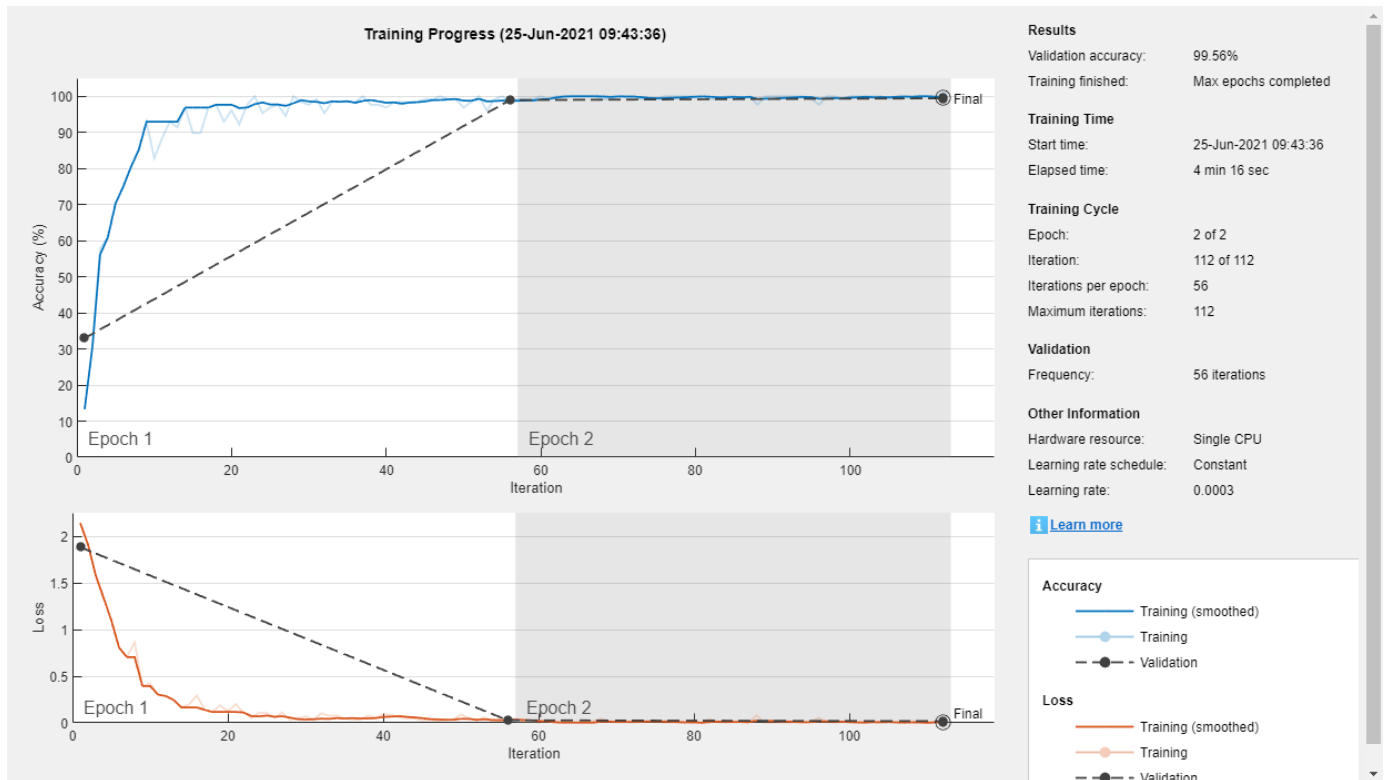
newClassificationLayer = classificationLayer("Name", "Sounds", "Classes", uniqueLabels);
lgraph = replaceLayer(lgraph, "Sound", newClassificationLayer);
```

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.

## Version History

Introduced in R2020b



## 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

**Signal Labeler**

### Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

### Functions

`audioFeatureExtractor` | `classifySound` | `designAuditoryFilterBank` | `melSpectrogram` | `vggish` | `vggishPreprocess` | `yamnetGraph` | `yamnetPreprocess`



# 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

<code>net</code>	Network
<code>P</code>	Network inputs
<code>T</code>	Network targets (default = zeros)
<code>Pi</code>	Initial input delay conditions (default = zeros)
<code>Ai</code>	Initial layer delay conditions (default = zeros)

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

<code>net</code>	Updated network
<code>Y</code>	Network outputs
<code>E</code>	Network errors
<code>Pf</code>	Final input delay conditions
<code>Af</code>	Final layer delay conditions
<code>tr</code>	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,

<code>P</code>	<code>Ni-by-TS</code> cell array	Each element <code>P{i,ts}</code> is an <code>Ri-by-Q</code> matrix.
<code>T</code>	<code>Nt-by-TS</code> cell array	Each element <code>T{i,ts}</code> is a <code>Vi-by-Q</code> matrix.

$P_i$	$N_i$ -by-ID cell array	Each element $P_i\{i, k\}$ is an $R_i$ -by- $Q$ matrix.
$A_i$	$N_L$ -by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by- $Q$ matrix.
$Y$	$N_o$ -by-TS cell array	Each element $Y\{i, ts\}$ is a $U_i$ -by- $Q$ matrix.
$E$	$N_o$ -by-TS cell array	Each element $E\{i, ts\}$ is a $U_i$ -by- $Q$ matrix.
$P_f$	$N_i$ -by-ID cell array	Each element $P_f\{i, k\}$ is an $R_i$ -by- $Q$ matrix.
$A_f$	$N_L$ -by-LD cell array	Each element $A_f\{i, k\}$ is an $S_i$ -by- $Q$ matrix.

where

$N_i$	=	<code>net.numInputs</code>
$N_L$	=	<code>net.numLayers</code>
$N_o$	=	<code>net.numOutputs</code>
ID	=	<code>net.numInputDelays</code>
LD	=	<code>net.numLayerDelays</code>
TS	=	Number of time steps
$Q$	=	Batch size
$R_i$	=	<code>net.inputs{i}.size</code>
$S_i$	=	<code>net.layers{i}.size</code>
$U_i$	=	<code>net.outputs{i}.size</code>

The columns of  $P_i$ ,  $P_f$ ,  $A_i$ , and  $A_f$  are ordered from oldest delay condition to most recent:

$P_i\{i, k\}$	=	Input $i$ at time $ts = k - ID$
$P_f\{i, k\}$	=	Input $i$ at time $ts = TS + k - ID$
$A_i\{i, k\}$	=	Layer output $i$ at time $ts = k - LD$
$A_f\{i, k\}$	=	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 $R_i$ )-by- $Q$ matrix
$T$	(sum of $V_i$ )-by- $Q$ matrix
$P_i$	(sum of $R_i$ )-by-( $ID*Q$ ) matrix
$A_i$	(sum of $S_i$ )-by-( $LD*Q$ ) matrix
$Y$	(sum of $U_i$ )-by- $Q$ matrix
$E$	(sum of $U_i$ )-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.

## **Version History**

Introduced before R2006a

### **See Also**

sim | init | train | revert

## 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)
```

## Version History

Introduced in R2010b

### See Also

`adapt`



## 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

### Remove and Add Delay to Network

This example shows how to create, train, and simulate 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.

#### Time Delay

```
[X,T] = simpleseries_dataset;  
net1 = timedelaynet(1:2,20);  
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);  
net1 = train(net1,Xs,Ts,Xi);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

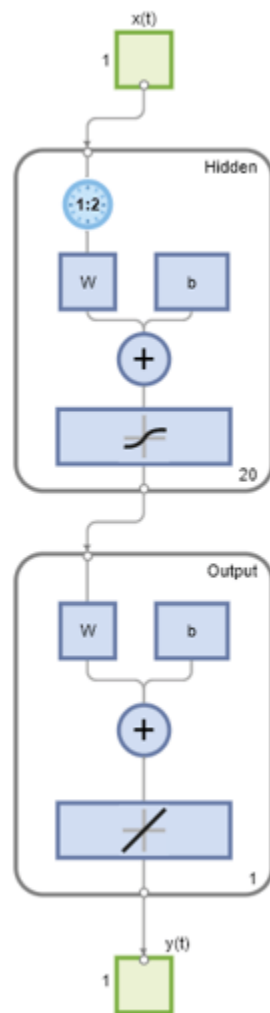
Performance Training State

Error Histogram Regression

Time-Series Response Error

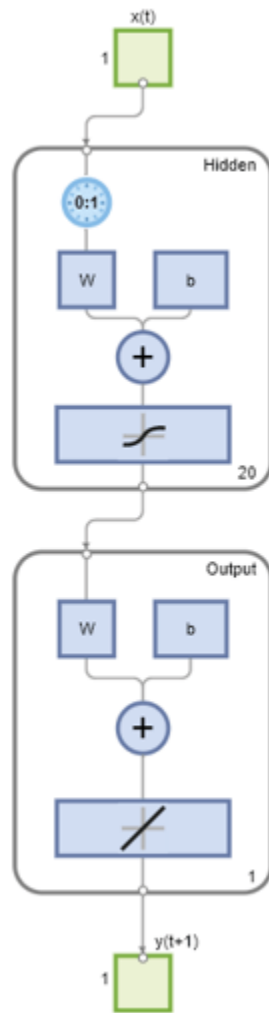
Input-Error Cross-correlation

```
y1 = net1(Xs,Xi);
view(net1)
```



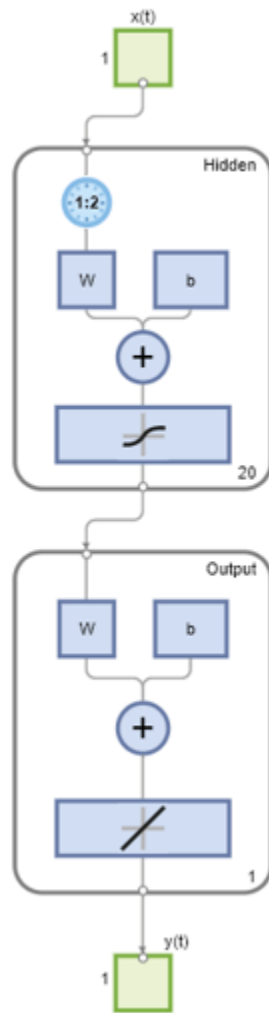
### Remove Delay

```
net2 = removedelay(net1);  
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);  
y2 = net2(Xs,Xi);  
view(net2)
```



### Add Delay

```
net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
```



## Version History

Introduced in R2010b

### See Also

[closeloop](#) | [openloop](#) | [removedelay](#)

## 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,

<code>pos</code>	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  $P_i$  and  $P_j$  from a set of  $S$  vectors is

$$D_{ij} = \max(\text{abs}(P_i - P_j))$$

## Version History

Introduced before R2006a

### See Also

`dist` | `linkdist` | `mandist` | `sim`

## 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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an $R \times Q$ matrix (or $N \times TS$ cell array of $R \times Q$ matrices)
<code>T</code>	Targets, an $S \times Q$ matrix (or $M \times TS$ cell array of $S \times Q$ matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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,  $R_i$  and  $S_i$  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)
```

## Version History

Introduced in R2010b

**See Also**

`defaultderiv` | `fpderiv` | `num2deriv` | `num5deriv` | `staticderiv`



# 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);
```

Network Diagram

**Training Results**  
 Training finished: Reached maximum number of epochs ✓

**Training Progress**

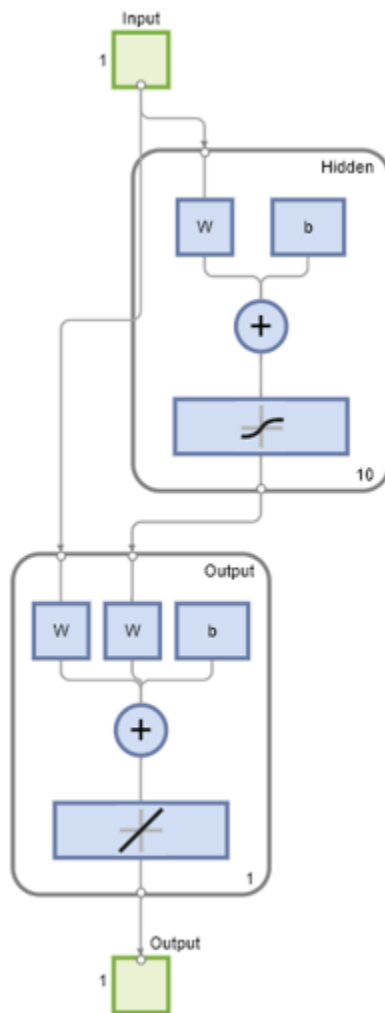
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:06	-
Performance	108	2.11e-07	0
Gradient	204	2.33e-05	1e-07
Mu	0.001	1e-06	1e+10
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

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-Ribière 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.

**Version History**

Introduced in R2010b

**See Also**

feedforwardnet | network

**Topics**

“Create, Configure, and Initialize Multilayer Shallow Neural Networks”

“Neural Network Object Properties”

“Neural Network Subobject Properties”

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `numelements` | `getelements` | `setelements` | `catsignals` | `catsamples` | `cattimesteps`

# 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')
```

## Version History

**Introduced in R2010b**

**See Also**

`nndata` | `numsamples` | `getsamples` | `setsamples` | `catelements` | `catsignals` | `cattimesteps`



# 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)
```

## Version History

**Introduced in R2010b**

## See Also

[nndata](#) | [numsignals](#) | [getsignals](#) | [setsignals](#) | [catelements](#) | [catsamples](#) | [cattimesteps](#)

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

[nndata](#) | [numtimesteps](#) | [gettimesteps](#) | [settimesteps](#) | [catelements](#) | [catsignals](#) | [catsamples](#)

# 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)
```

## Version History

**Introduced in R2010b**

## See Also

`nndata`

## 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);
```

Network Diagram

### Training Results

Training finished: Reached minimum gradient ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	686	1000	▲
Elapsed Time	-	00:00:07	-	
Performance	2.05	2.08e-11	0	
Gradient	3.24	9.97e-08	1e-07	
Mu	0.001	1e-08	1e+10	
Validation Checks	0	0	6	▼

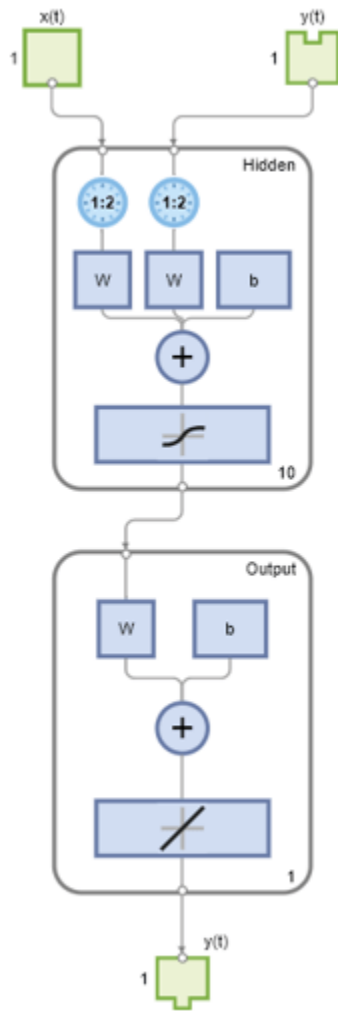
### Training Algorithms

Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

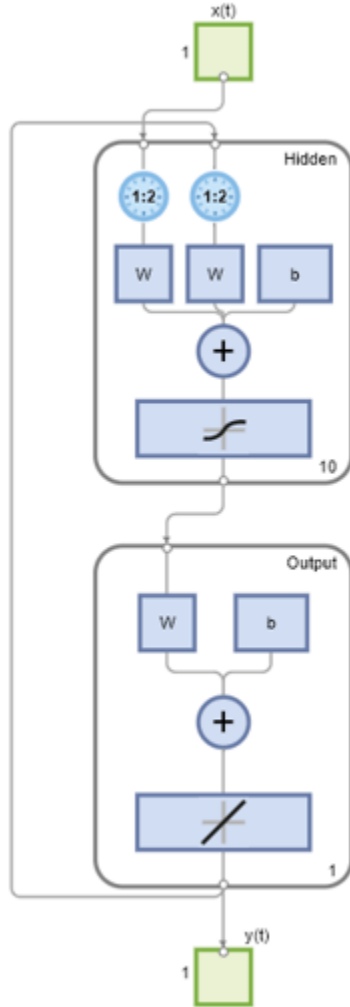
### Training Plots

Performance Training State  
Error Histogram Regression  
Time-Series Response Error  
Input-Error Cross-correlation

```
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`.

## Version History

Introduced in R2010b

### See Also

`narnet` | `narxnet` | `noloop` | `openLoop`

## 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  $(N_1*N_2*...)$  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  $A_i$ .

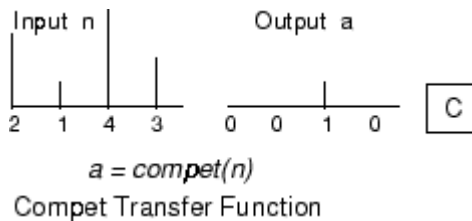
## Version History

Introduced before R2006a

## compet

Competitive transfer function

### Graph and Symbol



### 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.

$A = \text{compet}(N, FP)$  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';
```

## Version History

Introduced before R2006a

### See Also

[sim](#) | [softmax](#)

## 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,

<code>numClasses</code>	Number of classes to classify inputs (default = 5)
<code>kohonenLR</code>	Learning rate for Kohonen weights (default = 0.01)
<code>conscienceLR</code>	Learning rate for conscience bias (default = 0.001)

and returns a competitive layer with `numClasses` neurons.

### Examples


#### Create and Train a Competitive Layer

This example shows how to train a competitive layer to classify 150 iris flowers into 6 classes.

```
inputs = iris_dataset;  
net = competlayer(6);  
net = train(net, inputs);
```

Network Diagram

### Training Results

Training finished: Reached maximum number of epochs 

### Training Progress

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	1000	1000	▲
Elapsed Time	-	00:01:32	-	▼

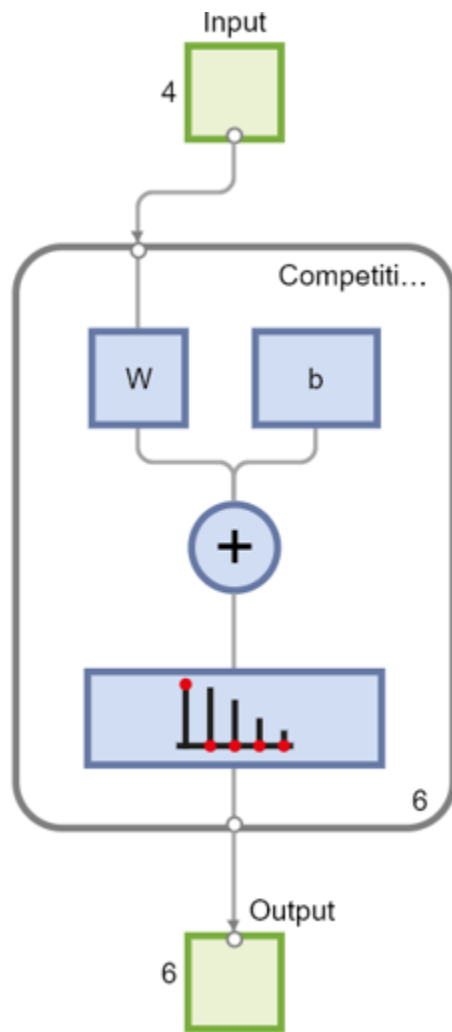
### Training Algorithms

Data Division: Random Weight/Bias Rule `trainru`

Performance: Mean Squared Error `mse`

Calculations: MATLAB

`view(net)`



```
outputs = net(inputs);  
classes = vec2ind(outputs);
```

## Version History

Introduced in R2010b

### See Also

[selforgmap](#) | [patternnet](#) | [lvqnet](#)

## 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)
```

### Version History

Introduced before R2006a

**See Also**

seq2con | concur



## concur

Create concurrent bias vectors

### Syntax

```
concur(B,Q)
```

### Description

```
concur(B,Q)
```

B	S-by-1 bias vector (or an N1-by-1 cell array of vectors)
Q	Concurrent size

and returns an S-by-B matrix of copies of B (or an N1-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))
```

## Version History

Introduced before R2006a

### See Also

`con2seq` | `netprod` | `netsum` | `seq2con` | `sim`

## 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.

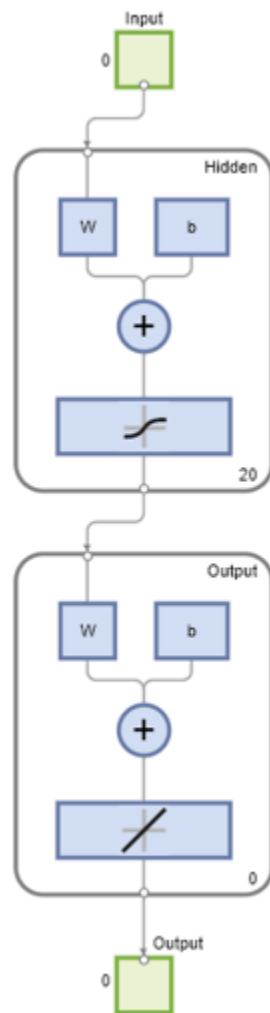
`net = configure(net,'outputs',t,i)` configures the outputs specified with the index vector `i`. If `i` is not specified all targets are configured.

### Examples

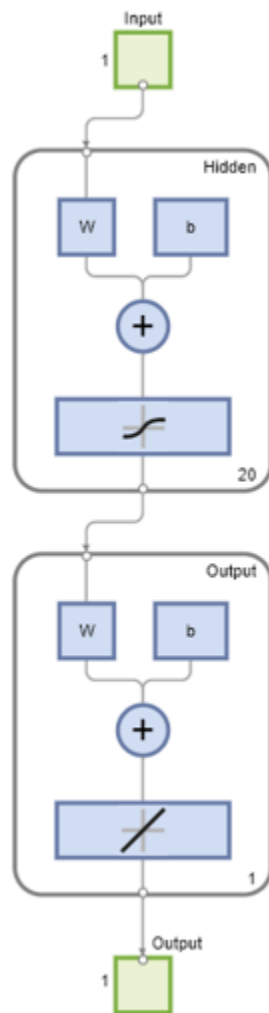
#### Configure Network with configure

This example shows how to manually configure a network for a simple fitting problem instead of using the `train` function.

```
[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.

**Version History****Introduced in R2010b****See Also**

isconfigured | unconfigure | init | train

## 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)
```

**Version History**

Introduced in R2006a

**See Also**

plotconfusion | roc

## 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`.

## **Version History**

**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:  $perf = \text{sum}(ce(:))/\text{numel}(ce)$ .

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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	37	1000

**Training Algorithms**  
Data Division: Random dividerand  
Training: Scaled Conjugate Gradient trainscg  
Performance: Cross Entropy crossentropy  
Calculations: MEX

**Training Plots**

Performance Training State

Error Histogram Confusion

Receiver Operating Characteristic

```
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  $N_i$ -by- $Q$  matrix. In each of these cases,  $N$  or  $N_i$  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  $N_i$ -by- $Q$  matrix. In each of these cases,  $N$  or  $N_i$  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 ( $N \geq 2$ ): 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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).

### Version History

**Introduced in R2013b**

#### **See Also**

mae | mse | patternnet | sae | softmax | sse

# 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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an R-by-Q matrix (or N-by-TS cell array of Ri-by-Q matrices)
<code>T</code>	Targets, an S-by-Q matrix (or M-by-TS cell array of Si-by-Q matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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)
```

## Version History

**Introduced in R2010b**

**See Also**

`bttderiv` | `fpderiv` | `num2deriv` | `num5deriv` | `staticderiv`



# 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

$$d = \text{sum}((x-y).^2).^0.5$$

**Version History**

Introduced before R2006a

**See Also**

`sim` | `dotprod` | `negdist` | `normprod` | `mandist` | `linkdist`

## 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,

<code>delays</code>	Row vector of increasing 0 or positive delays (default = 1:2)
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (default = 10)
<code>trainFcn</code>	Training function (default = 'trainlm')

and returns a distributed delay neural network.

### Examples

#### Distributed Delay Network

This example shows how to train a distributed delay neural network 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);
```

Network Diagram

### Training Results

Training finished: Met validation criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	8	1000
Elapsed Time	-	00:00:02	-
Performance	0.645	0.0137	0
Gradient	1.61	0.00434	1e-07
Mu	0.001	0.0001	1e+10
Validation Checks	0	6	6

### Training Algorithms

Data Division: Random dividerand

Training: Levenberg-Marquardt trainlm

Performance: Mean Squared Error mse

Calculations: MEX

### Training Plots

Performance

Training State

Error Histogram

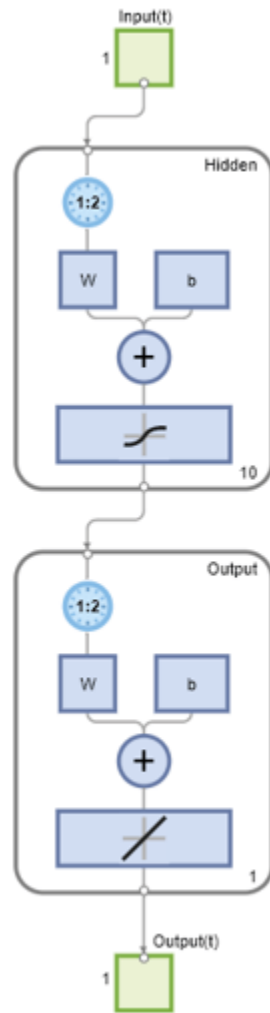
Regression

Time-Series Response

Error

Input-Error Cross-correlation

view(net)



```
Y = net(Xs,Xi,Ai);  
perf = perform(net,Y,Ts)  
  
perf = 0.0323
```

## Version History

Introduced in R2010b

### See Also

preparets | removedelay | timedelaynet | narnet | narxnet

## 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
```

## Version History

Introduced in R2008a

### See Also

divideind | divideint | dividerand | dividetrain

## 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
```

**Version History**

Introduced in R2008a

**See Also**

`divideblock` | `divideint` | `dividerand` | `dividetrain`

## 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,

<code>Q</code>	Number of targets to divide up.
<code>trainRatio</code>	Ratio of vectors for training. Default = 0.7.
<code>valRatio</code>	Ratio of vectors for validation. Default = 0.15.
<code>testRatio</code>	Ratio of vectors for testing. Default = 0.15.

and returns

<code>trainInd</code>	Training indices
<code>valInd</code>	Validation indices
<code>testInd</code>	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
```

## Version History

Introduced in R2008a

### See Also

`divideblock` | `divideind` | `dividerand` | `dividetrain`

# 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
```

### Version History

Introduced in R2008a

#### See Also

`divideblock` | `divideind` | `divideint` | `dividetrain`

## 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
```

## Version History

**Introduced in R2010b**

### See Also

divideblock | divideind | divideint | dividerand

## 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.

$Z = \text{dotprod}(W, P, FP)$  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.

$\text{dim} = \text{dotprod}('size', S, R, FP)$  takes the layer dimension S, input dimension R, and function parameters, and returns the weight size [S-by-R].

$\text{dw} = \text{dotprod}('dw', W, P, Z, FP)$  returns the derivative of Z with respect to W.

$\text{info} = \text{dotprod}('code')$  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.

## Version History

**Introduced before R2006a**

### See Also

`sim` | `dist` | `feedforwardnet` | `negdist` | `normprod`

## 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 net input (column) vectors,  $N$ , and returns an  $S$ -by- $Q$  matrix,  $A$ , of output vectors, where each element of  $N$  is squashed from the interval  $[-\infty \infty]$  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';
```

## Version History

Introduced in R2012b

### See Also

[elliotsig](#) | [logsig](#) | [tansig](#)



# elliott2sig

Elliot 2 symmetric sigmoid transfer function

## Syntax

```
A = elliott2sig(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 = elliott2sig(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  $[-\infty \infty]$  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 = elliott2sig(n);
```

Plot the transfer function:

```
n = -5:0.01:5;
plot(n, elliott2sig(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 = 'elliott2sig';
```

## Version History

Introduced in R2012b

## See Also

`elliotsig` | `logsig` | `tansig`

## 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 (`narntnet`) 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,

<code>layerdelays</code>	Row vector of increasing 0 or positive delays (default = 1:2)
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (default = 10)
<code>trainFcn</code>	Training function (default = 'trainlm')

and returns an Elman neural network.

### Examples

#### Train Elman Neural Network

This example shows how to train an Elman neural network 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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	165	1000
Elapsed Time	-	00:00:04	-
Performance	0.114	0.000313	0
Gradient	0.646	0.00547	1e-05
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Gradient Descent with Momentum and Adaptive LR ...  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

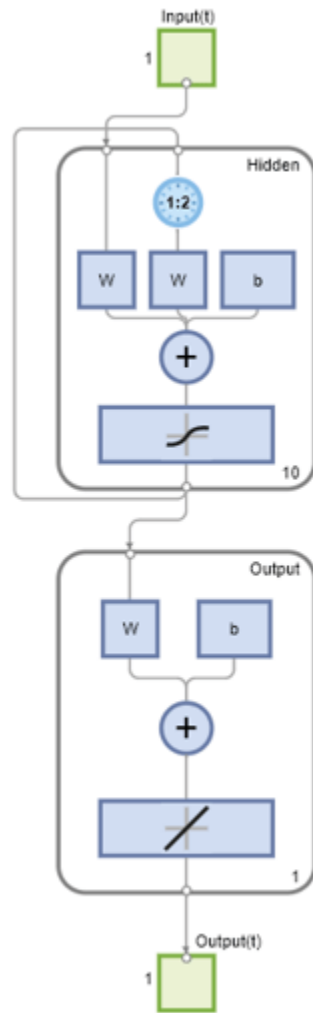
Regression

Time-Series Response

Error

Input-Error Cross-correlation

view(net)



```
Y = net(Xs,Xi,Ai);  
perf = perform(net,Ts,Y)  
  
perf = 9.0122e-04
```

## Version History

Introduced in R2010b

### See Also

[preparets](#) | [removedelay](#) | [timedelaynet](#) | [layreclnet](#) | [narnet](#) | [narxnet](#)

# 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])
```

## Version History

Introduced before R2006a

## See Also

plotes

## extends

Extend time series data to given number of timesteps

### Syntax

```
extends(x, ts, v)
```

### Description

`extends(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 = extends(x, 25, 0)
```

## Version History

Introduced in R2010b

### See Also

[nndata](#) | [catsamples](#) | [preparets](#)

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	19	1000
Elapsed Time	-	00:00:08	-
Performance	324	9.25e-05	0
Gradient	414	0.000292	1e-07
Mu	0.001	1e-05	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

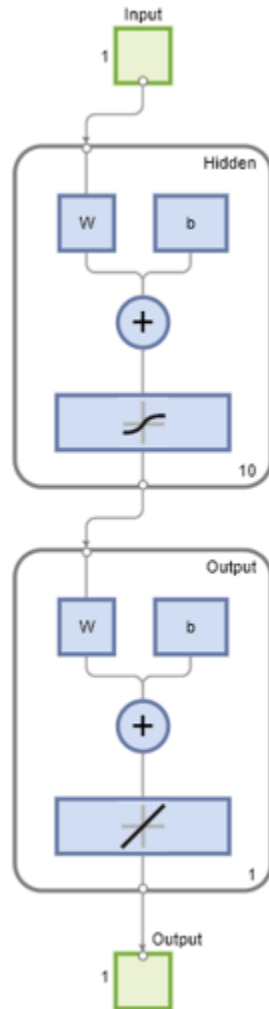
**Training Plots**

Performance      Training State  
Error Histogram      Regression

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-Ribière 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.

**Version History**

Introduced in R2010b

**See Also**

fitnet | network | patternnet | cascadeforwardnet

**Topics**

“Neural Network Object Properties”

“Neural Network Subobject Properties”

## 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 `fixunknowns` 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.

## Version History

**Introduced in R2006a**

**See Also**

mapminmax | mapstd | processpca

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

[getwb](#) | [setwb](#) | [separatewb](#)

## 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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an R-by-Q matrix (or N-by-TS cell array of Ri-by-Q matrices)
<code>T</code>	Targets, an S-by-Q matrix (or M-by-TS cell array of Si-by-Q matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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)
```

## Version History

Introduced in R2010b



**See Also**

`bttderiv` | `defaultderiv` | `num2deriv` | `num5deriv` | `staticderiv`

## 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,

<code>net</code>	Neural network
<code>toMatrix</code>	True if result is to be in matrix form
<code>columnSample</code>	True if samples are to be represented as columns, false if rows
<code>cellTime</code>	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)
```

## Version History

Introduced in R2010b

### See Also

`tonndata`

# 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]}
```

## Version History

Introduced in R2010b

## See Also

`gsubtract` | `gmultiply` | `gdivide` | `gnegate` | `gsqrt`

## 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]}
```

## Version History

Introduced in R2010b

### See Also

`gadd` | `gsubtract` | `gmultiply` | `gnegate` | `gsqrt`

# 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 “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).

## Version History

Introduced before R2006a

## See Also

`preparets`

# 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 “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**

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`

## Version History

Introduced in R2013b

## 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”

## 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])
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `numelements` | `setelements` | `catelements` | `getsamples` | `gettimesteps` | `getsignals`

# 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])
```

## Version History

Introduced in R2010b

## See Also

[nndata](#) | [numsamples](#) | [setsamples](#) | [catsamples](#) | [getelements](#) | [gettimesteps](#) | [getsignals](#)

## 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, then the result is the rows of `x` with indices specified by `ind`.

### 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)
```

### Version History

**Introduced in R2010b**

### See Also

[nndata](#) | [numsignals](#) | [setsignals](#) | [catsignals](#) | [getelements](#) | [getsamples](#) | [gettimesteps](#)

## 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);
```

## **Version History**

**Introduced in R2010b**

### **See Also**

[gensim](#) | [setsiminit](#) | [nndata2sim](#) | [sim2nndata](#)

# 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, then the result is the columns of `x` with indices specified by `ind`.

## 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)
```

## Version History

Introduced in R2010b

## See Also

`nndata` | `numtimesteps` | `settimesteps` | `cattimesteps` | `getelements` | `getsamples` | `getsignals`

## 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)
```

## Version History

Introduced in R2010b

### See Also

[setwb](#) | [formwb](#) | [separatewb](#)



# 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]}
```

## Version History

Introduced in R2010b

## See Also

gadd | gsubtract | gdivide | gnegate | gsqrt

## 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
```

### Version History

**Introduced in R2010b**

### See Also

gadd | gsubtract | gdivide | gmultiply | gsqrt

## 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)
```

## Version History

Introduced in R2012b

**See Also**

nndata2gpu

# 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
------------	-------------------------------

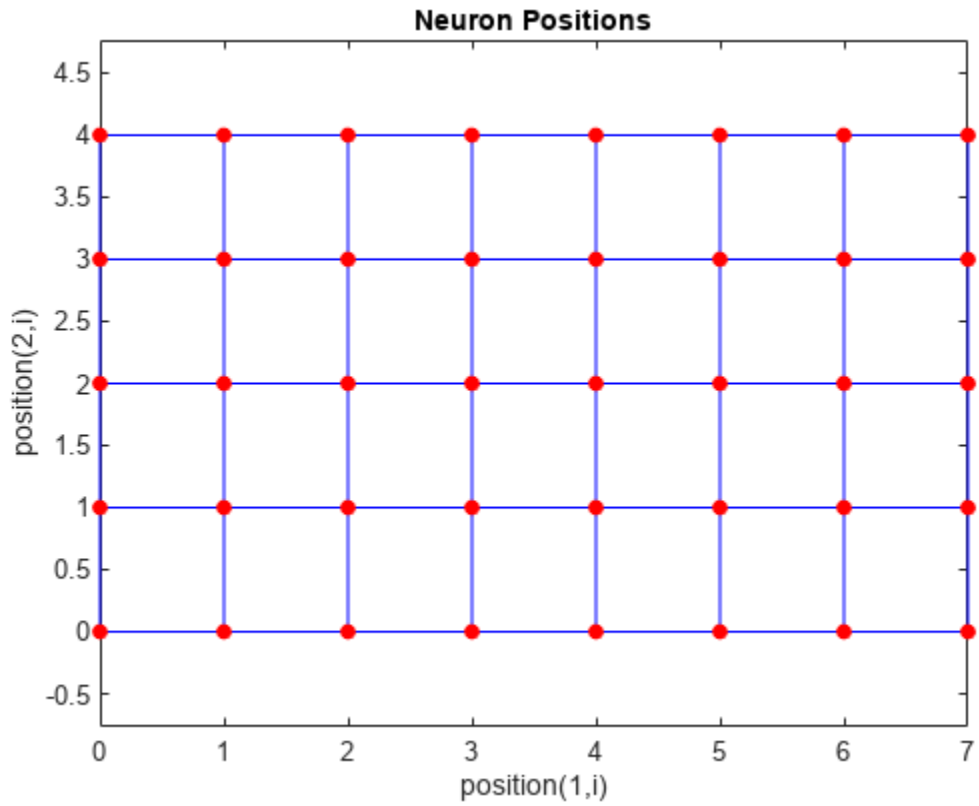
and returns an N-by-S matrix of N coordinate vectors where N is the number of dimensions and S is the product of dimensions.

## 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)
```



## Version History

Introduced before R2006a

## See Also

hextop | randtop | tritop

# 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=2x2 cell array  
    {[    1]}    {[1.4142]}  
    {[1.7321]}    {[    2]}
```

## Version History

**Introduced in R2010b**

## See Also

`gadd` | `gsubtract` | `gdivide` | `gmultiply` | `gnegate`

## 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]}
```

### Version History

Introduced in R2010b

### See Also

`gadd` | `gmultiply` | `gdivide` | `gnegate` | `gsqrt`



# hardlim

Hard-limit transfer function

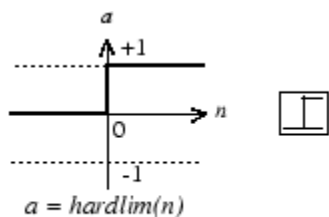
## Syntax

```
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 the corresponding elements in  $N$  are 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 the corresponding elements in N are greater than or equal to 0.

## Algorithms

$$\text{hardlim}(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

## Version History

Introduced before R2006a

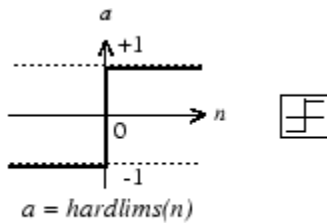
## See Also

sim | hardlims

# hardlims

Symmetric hard-limit transfer function

## Graph and Symbol



Symmetric Hard-Limit Transfer Function

## Syntax

$A = \text{hardlims}(N, FP)$

## Description

`hardlims` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

$A = \text{hardlims}(N, FP)$  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  $n \geq 0$ , -1 otherwise.

### Version History

Introduced before R2006a

### See Also

`sim` | `hardlim`

# 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
------------	-------------------------------

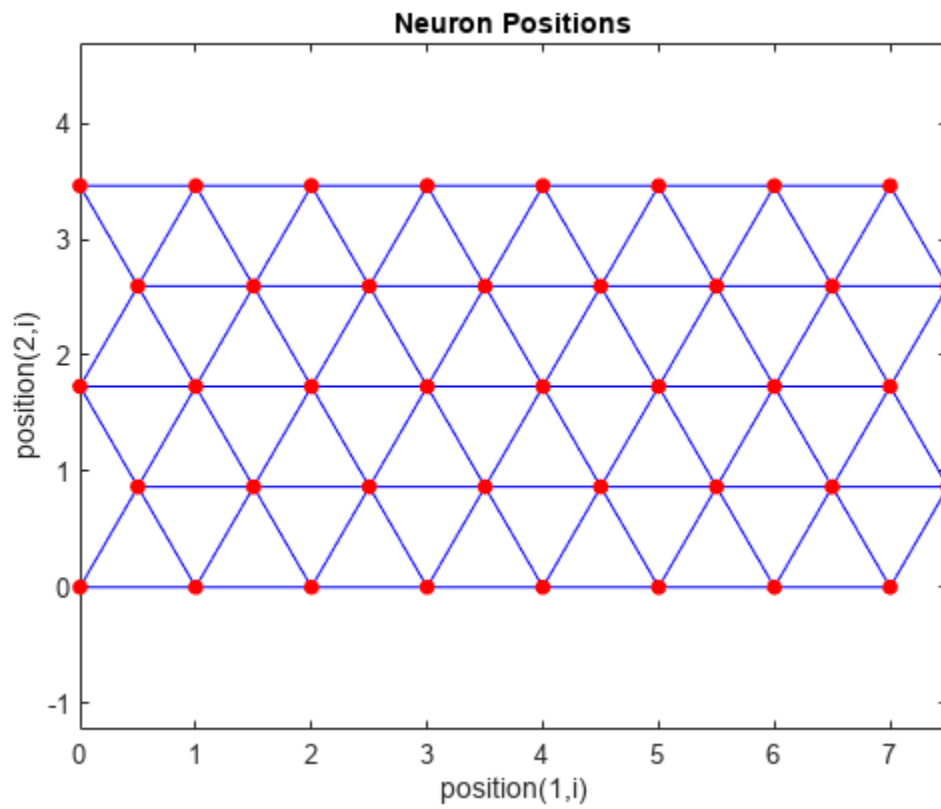
and returns an N-by-S matrix of N coordinate vectors where N is the number of dimensions and S is the product of dimensions.

## 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)
```



## Version History

Introduced before R2006a

## See Also

[gridtop](#) | [randtop](#) | [tritop](#)

# 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.

### Version History

Introduced before R2006a

### See Also

[vec2ind](#) | [sub2ind](#) | [ind2sub](#)



# 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.

### Version History

Introduced before R2006a

#### See Also

`sim` | `adapt` | `train` | `initlay` | `initnw` | `initwb` | `rands` | `revert`

# 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.

## **Version History**

**Introduced before R2006a**

### **See Also**

`competlayer` | `init` | `initlay` | `initwb` | `learncon`

# 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

<code>net</code>	Neural network
------------------	----------------

and returns the network with each layer updated.

`info = initlay('code')` returns useful information for each supported `code` character vector:

<code>'pnames'</code>	Names of initialization parameters
<code>'pdefaults'</code>	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`.

## Version History

Introduced before R2006a

**See Also**

`cascadeforwardnet` | `feedforwardnet` | `init` | `initnw` | `initwb`

# 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.

## Version History

Introduced in R2010b

## See Also

`lvqnet` | `init`

## 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  $[-\infty, \infty]$ . 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,

<code>net</code>	Neural network
<code>i</code>	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.

## Version History

Introduced before R2006a

### See Also

`cascaforwardnet` | `feedforwardnet` | `init` | `initlay` | `initwb`

## 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,

<code>net</code>	Neural network
<code>i</code>	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.

## Version History

Introduced before R2006a

### See Also

`init` | `initlay` | `initnw` | `linearlayer` | `perceptron`

# 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])
```

## Version History

Introduced before R2006a

## See Also

initwb | initlay | init

## 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)
```

## Version History

Introduced in R2010b

### See Also

[configure](#) | [unconfigure](#)

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Reached minimum gradient ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	461	1000
Elapsed Time	-	00:00:11	-
Performance	0.103	6.12e-11	0
Gradient	0.606	9.98e-08	1e-07
Mu	0.001	1e-08	1e+10
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

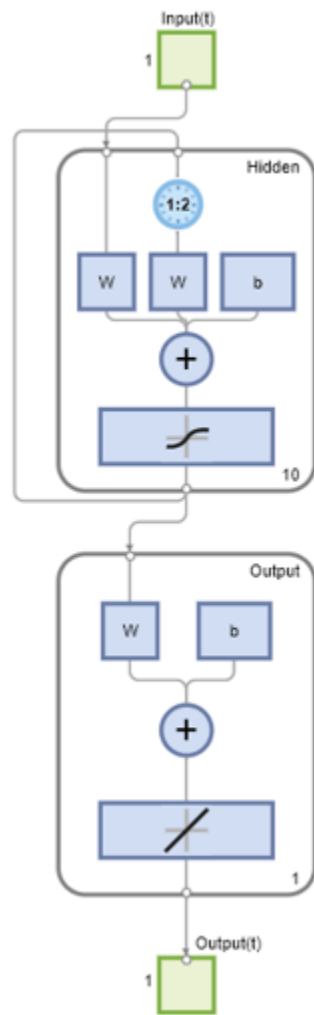
Regression

Time-Series Response

Error

Input-Error Cross-correlation

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.

<b>Training Function</b>	<b>Algorithm</b>
'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-Ribière 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

## **Version History**

**Introduced in R2010b**

### **See Also**

preparets | removedelay | distdelaynet | timedelaynet | narnet | narxnet



# 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.)

### Version History

Introduced before R2006a

**See Also**

learnk | learnos | adapt | train

# learnngd

Gradient descent weight and bias learning function

## Syntax

```
[dW,LS] = learnngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnngd('code')
```

## Description

learnngd is the gradient descent weight and bias learning function.

[dW,LS] = learnngd(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 learnngd's learning parameter, shown here with its default value.

LP.lr - 0.01	Learning rate
--------------	---------------

info = learnngd('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 `learnngd` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```
dW = learnngd([],[],[],[],[],[],[],gW,[],[],lp,[])
```

## Algorithms

`learnngd` 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  $dw = lr * gW$ .

## Version History

Introduced before R2006a

## See Also

`adapt` | `learnngdm` | `train`

## 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:

$$dW = mc*dW_{prev} + (1-mc)*lr*gW$$

The previous weight change  $dW_{prev}$  is stored and read from the learning state  $LS$ .

## Version History

Introduced before R2006a

### See Also

`adapt` | `learngd` | `train`

## 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'
```

## Version History

Introduced before R2006a

## References

Hebb, D.O., *The Organization of Behavior*, New York, Wiley, 1949

## See Also

`learnhd` | `adapt` | `train`

## 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:

$$dw = lr*a*p' - dr*w$$

## Version History

Introduced before R2006a

## See Also

`learnh` | `adapt` | `train`

## 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:

$$dw = lr * a * (p - w)$$

## Version History

Introduced before R2006a

## References

Grossberg, S., *Studies of the Mind and Brain*, Dordrecht, Holland, Reidel Press, 1982

## See Also

`learnk` | `learnos` | `adapt` | `train`

## 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:

$$dw = lr*(p' - w), \text{ if } a \sim \theta; = 0, \text{ otherwise}$$

## Version History

Introduced before R2006a

## References

Kohonen, T., *Self-Organizing and Associative Memory*, New York, Springer-Verlag, 1984

## See Also

`learnis` | `learnos` | `adapt` | `train`

## 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:

$$dw(i,:) = +lr*(p-w(i,:)) \text{ if } gA(i) = 0; = -lr*(p-w(i,:)) \text{ if } gA(i) = -1$$

## Version History

Introduced before R2006a

## See Also

`learnlv2` | `adapt` | `train`

## 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  $d_i$  between the winning neuron and the input  $p$  is roughly equal to the distance  $d_j$  from the runnerup neuron to the input  $p$  according to the given window,

$$\min(d_i/d_j, d_j/d_i) > (1-\text{window})/(1+\text{window})$$

then move the winning neuron  $i$  weights away from the input vector, and move the runnerup neuron  $j$  weights toward the input according to

$$\begin{aligned} dw(i,:) &= -lp.lr*(p'-w(i,:)) \\ dw(j,:) &= +lp.lr*(p'-w(j,:)) \end{aligned}$$

## **Version History**

**Introduced before R2006a**

### **See Also**

learnlv1 | adapt | train

# 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'
```

## Version History

Introduced before R2006a

## References

Grossberg, S., *Studies of the Mind and Brain*, Dordrecht, Holland, Reidel Press, 1982

## See Also

`learnis` | `learnk` | `adapt` | `train`

# 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:

$$\begin{aligned} dw &= 0, \text{ if } e = 0 \\ &= p', \text{ if } e = 1 \\ &= -p', \text{ if } e = -1 \end{aligned}$$

This can be summarized as

$$dw = e * p'$$

### Version History

Introduced before R2006a

### References

Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C., Spartan Press, 1961

### See Also

`adapt` | `learnpn` | `train`



# 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'
```

## Version History

Introduced before R2006a

## See Also

`adapt` | `learnp` | `train`

# 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:

$$a2(i,q) = \begin{cases} 1, & \text{if } a(i,q) = 1 \\ 0.5, & \text{if } a(j,q) = 1 \text{ and } D(i,j) \leq nd \\ 0, & \text{otherwise} \end{cases}$$

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.

## Version History

Introduced before R2006a

**See Also**

adapt | train

## 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 `learnsomb` 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.

## Version History

Introduced in R2008a

**See Also**

`adapt` | `selforgmap` | `train`



# 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'
```

## Version History

Introduced before R2006a

## 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`

## 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);
```

Network Diagram

### Training Results

Training finished: Reached maximum number of epochs ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:07	-
Performance	1	0.24	0
Gradient	NaN	NaN	1e-06
Validation Checks	0	0	6

### Training Algorithms

Data Division: Batch Weight/Bias Rule `trainb`

Performance: Mean Squared Error `mse`

Calculations: MATLAB

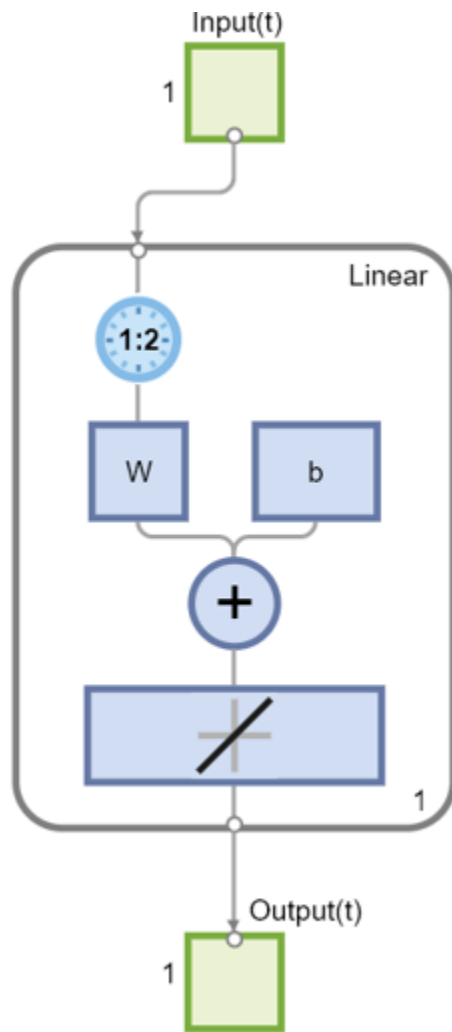
### Training Plots

Performance

Training State

Error Histogram

`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.

## Version History

**Introduced in R2010b**

### **See Also**

preparets | removedelay | timedelaynet | narnet | narxnet

## 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,

<code>pos</code>	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  $P_i$  and  $P_j$  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
```



## **Version History**

Introduced before R2006a

### **See Also**

dist | mandist | selforgmap | sim

## logsig

Log-sigmoid transfer function

### Syntax

```
A = logsig(N)
dA_dN = logsig('dn',N,A,FP)
info = logsig(code)
```

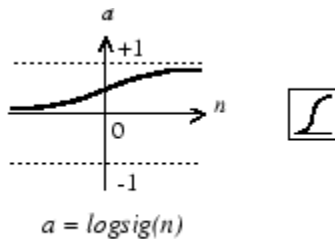
### Description

---

**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

$$\text{logsig}(n) = 1 / (1 + \exp(-n))$$

## Version History

Introduced before R2006a

### See Also

sim | tansig

## 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,

<code>hiddenSize</code>	Size of hidden layer (default = 10)
<code>lvqLR</code>	LVQ learning rate (default = 0.01)
<code>lvqLF</code>	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

This example shows how to train an LVQ network to classify iris flowers.

```
[x,t] = iris_dataset;  
net = lvqnet(10);  
net.trainParam.epochs = 50;  
net = train(net,x,t);
```

Network Diagram

### Training Results

Training finished: Reached maximum number of epochs ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	50	50	▲
Elapsed Time	-	00:00:16	-	
Performance	0.444	0.0356	0	
Validation Checks	0	0	6	▼

### Training Algorithms

Data Division: Random Weight/Bias Rule `trainr`

Performance: Mean Squared Error `mse`

Calculations: MATLAB

### Training Plots

Performance

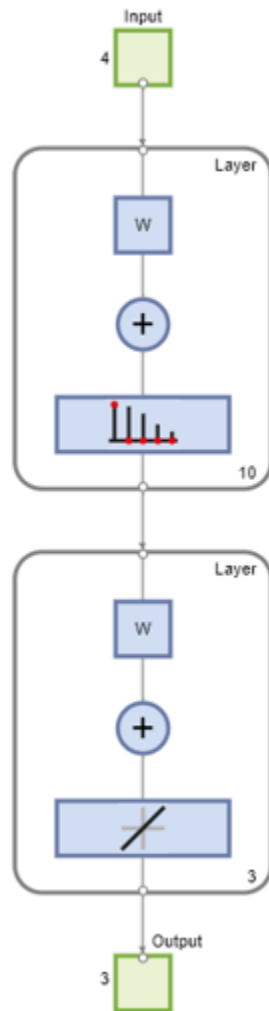
Training State

Error Histogram

Confusion

Receiver Operating Characteristic

`view(net)`



```
y = net(x);  
perf = perform(net,y,t)  
  
perf = 0.0489  
classes = vec2ind(y);
```

## Version History

Introduced in R2010b

### See Also

[competlayer](#) | [patternnet](#) | [selforgmap](#)

# 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.

## Version History

Introduced in R2010b

## See Also

lvqnet | initlvq

## 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.

## Version History

Introduced before R2006a

### **See Also**

`mse` | `perceptron`

## 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

$$D = \text{sum}(\text{abs}(x-y))$$

## **Version History**

**Introduced before R2006a**

### **See Also**

`dist` | `linkdist` | `sim`

# 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  $x_{\max}=x_{\min}$  or if either  $x_{\max}$  or  $x_{\min}$  are non-finite, then  $y=x$  and no change occurs.)

$$y = (y_{\max}-y_{\min})*(x-x_{\min})/(x_{\max}-x_{\min}) + y_{\min};$$

## Version History

Introduced in R2006a

### See Also

`fixunknowns` | `mapstd` | `processpca`

## 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

The function assumes that `X` only contains finite real values. Additionally, the result for each row depends on the elements in that row.

- If the elements of the row are not all equal, then the function transforms the mean and standard deviation of each row to `ymean` (default 0) and `ystd` (default 1).

$$y = (x - xmean) * (ystd / xstd) + ymean;$$

- If the elements of the row are all equal, then the function does not modify the data in that row (`y=x`).

## **Version History**

**Introduced in R2006a**

### **See Also**

`fixunknowns` | `mapminmax` | `processpca`

# 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')
```

## Version History

Introduced before R2006a

## See Also

learnwh | linearlayer

## 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]})
```

## Version History

Introduced in R2010b

### See Also

meansqr | sumabs | sumsqr

# 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]})
```

## Version History

Introduced in R2010b

## See Also

meanabs | sumabs | sumsqr

## 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 Q input (column) vectors with values in the ranges = [Pmin Pmax] where Q is the batch size

and returns an S-by-R matrix with rows set to  $(P_{min}+P_{max})' / 2$ .

### 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])
```

### Version History

Introduced before R2006a

### See Also

`initwb` | `initlay` | `init`

# 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 = rand(4,5)
mm = minmax(x)
x = rand(1;2),3,4)
mm = minmax(x)
```

## Version History

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 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
```

Train the network and evaluate performance.

```
net = train(net, x, t);  
y = net(x);  
perf = perform(net, t, y)
```

Alternatively, you can call `mse` directly.

```
perf = mse(net, t, y, 'regularization', 0.01)
```

## 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.

## Version History

Introduced before R2006a

## See Also

`mae`

## 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);
```

Network Diagram

**Training Results**  
Training finished: Reached minimum gradient ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	169	1000

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

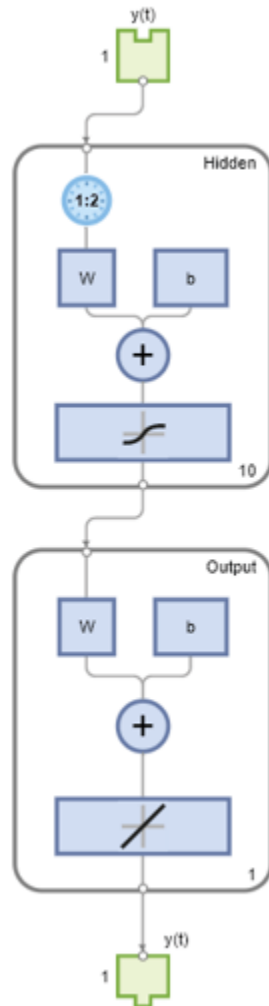
Regression

Time-Series Response

Error

Display the trained network.

```
view(net)
```



Calculate the network output  $Y$ , final input states  $X_f$ , and final layer states  $A_f$  of the open-loop network from the network input  $X_s$ , initial input states  $X_i$ , and initial layer states  $A_i$ .

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
```

Calculate the network performance.

```
perf = perform(net,Ts,Y)
```

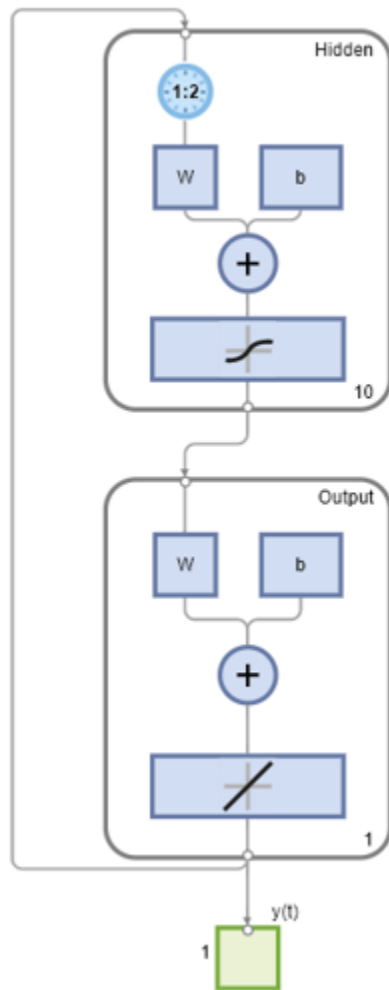
```
perf = 1.0100e-09
```

To predict the output for the next 20 time steps, first simulate the network in closed-loop mode. The final input states  $X_f$  and layer states  $A_f$  of the open-loop network  $net$  become the initial input states  $X_{ic}$  and layer states  $A_{ic}$  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  $X_{ic}$  and  $A_{ic}$ .

```
Yc = netc(cell(0,20),Xic,Aic)
```

$Yc=1 \times 20$  cell array

```
{[0.8346]} {[0.3329]} {[0.9084]} {[1.0000]} {[0.3190]} {[0.7329]} {[0.9801]}
```

## 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
'trainscg'	Scaled Conjugate Gradient
'traincgb'	Conjugate Gradient with Powell/Beale Restarts
'traincgf'	Fletcher-Powell Conjugate Gradient
'traincgp'	Polak-Ribière 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

## Version History

Introduced in R2010b

### See Also

preparets | removedelay | timedelaynet | narxnet | closeloop | network | train | openloop

### Topics

“Multistep Neural Network Prediction”

# 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);
```



Network Diagram

### Training Results

Training finished: Met validation criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	10	1000
Elapsed Time	-	00:00:03	-
Performance	0.615	0.00697	0
Gradient	1.24	0.00312	1e-07
Mu	0.001	0.0001	1e+10
Validation Checks	0	6	6

### Training Algorithms

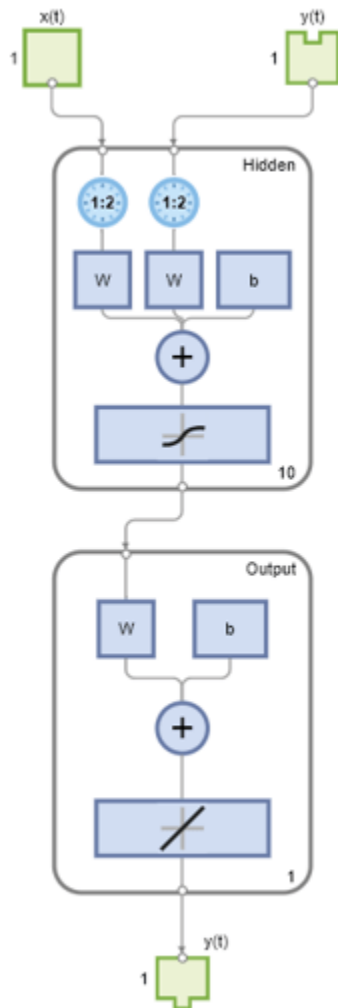
Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

### Training Plots

Performance Training State  
 Error Histogram Regression  
 Time-Series Response Error  
 Input-Error Cross-correlation

Display the trained network.

```
view(net)
```



Calculate the network output  $Y$ , final input states  $X_f$ , and final layer states  $A_f$  of the open-loop network from the network input  $X_s$ , initial input states  $X_i$ , and initial layer states  $A_i$ .

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
```

Calculate the network performance.

```
perf = perform(net,Ts,Y)
```

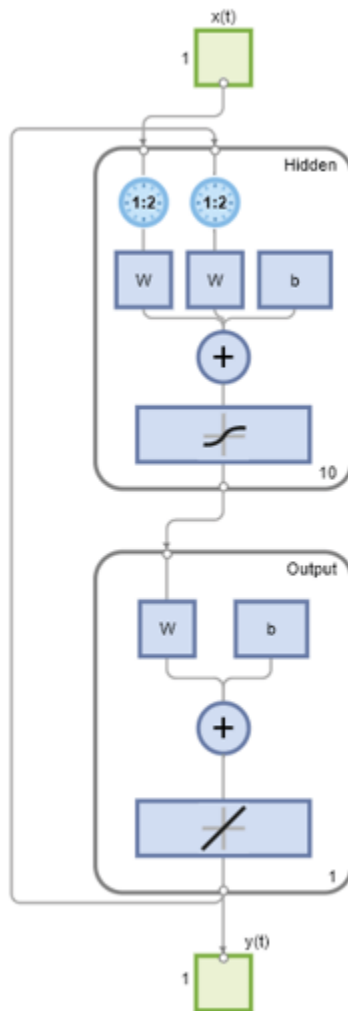
```
perf = 0.0153
```

To predict the output for the next 20 time steps, first simulate the network in closed-loop mode. The final input states  $X_f$  and layer states  $A_f$  of the open-loop network `net` become the initial input states  $X_{ic}$  and layer states  $A_{ic}$  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*

```
{[-0.0156]} { [0.1133]} { [-0.1472]} { [-0.0706]} { [0.0355]} { [-0.2829]} { [0.1133]}
```

## 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-Ribière 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

**Version History**

Introduced in R2010b

**See Also**

closeloop | narnet | openloop | preparets | removedelay | timedelaynet | network | train

**Topics**

“Multistep Neural Network Prediction”  
 “Design Time Series NARX Feedback Neural Networks”

# nctool

Open Neural Net Clustering app

## Syntax

```
nctool  
nctool("close")
```

## 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”.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

nctool("close") closes the Neural Net Clustering app.

## Algorithms

The **Neural Net Clustering** app 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.

## Version History

**Introduced in R2008a**

## See Also

**Neural Net Clustering** | nftool | nprtool | ntstool

## Topics

“Cluster Data with a Self-Organizing Map”

## 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:

$$z = -\sqrt{\sum(w-p)^2}$$

## **Version History**

Introduced before R2006a

### **See Also**

`competlayer` | `dist` | `dotprod` | `selforgmap` | `sim`

## 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';
```

## Version History

Introduced in R2006a



**See Also**

tansig | logsig

## 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

$Z_i$	S-by-Q matrices in a row cell array
-------	-------------------------------------

and returns an element-wise product of  $Z_1$  to  $Z_n$ .

`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  $Z_1$  and  $Z_2$  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.

## Version History

**Introduced before R2006a**

### See Also

`sim` | `netsum` | `concur`

## 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,

<code>Zi</code>	S-by-Q matrices in a row cell array
<code>FP</code>	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';
```

## **Version History**

**Introduced before R2006a**

### **See Also**

[cascadeforwardnet](#) | [feedforwardnet](#) | [netprod](#) | [netinv](#)

## network

Create custom shallow neural network

### Syntax

```
net = network
net =
network(numInputs, numLayers, biasConnect, inputConnect, layerConnect, outputConnect)
```

### 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, outputConnect)` takes these optional arguments (shown with default values):

<code>numInputs</code>	Number of inputs, 0
<code>numLayers</code>	Number of layers, 0
<code>biasConnect</code>	<code>numLayers</code> -by-1 Boolean vector, zeros
<code>inputConnect</code>	<code>numLayers</code> -by- <code>numInputs</code> Boolean matrix, zeros
<code>layerConnect</code>	<code>numLayers</code> -by- <code>numLayers</code> Boolean matrix, zeros
<code>outputConnect</code>	1-by- <code>numLayers</code> Boolean vector, zeros

and returns

<code>net</code>	New network with the given property values
------------------	--

### Properties

#### Architecture Properties

<code>net.numInputs</code>	0 or a positive integer	Number of inputs.
----------------------------	-------------------------	-------------------

<code>net.numLayers</code>	0 or a positive integer	Number of layers.
<code>net.biasConnect</code>	<code>numLayer</code> -by-1 Boolean vector	If <code>net.biasConnect(i)</code> is 1, then layer <code>i</code> has a bias, and <code>net.biases{i}</code> is a structure describing that bias.
<code>net.inputConnect</code>	<code>numLayer</code> -by- <code>numInputs</code> Boolean vector	If <code>net.inputConnect(i, j)</code> is 1, then layer <code>i</code> has a weight coming from input <code>j</code> , and <code>net.inputWeights{i, j}</code> is a structure describing that weight.
<code>net.layerConnect</code>	<code>numLayer</code> -by- <code>numLayers</code> Boolean vector	If <code>net.layerConnect(i, j)</code> is 1, then layer <code>i</code> has a weight coming from layer <code>j</code> , and <code>net.layerWeights{i, j}</code> is a structure describing that weight.
<code>net.outputConnect</code>	1-by- <code>numLayers</code> Boolean vector	If <code>net.outputConnect(i)</code> is 1, then the network has an output from layer <code>i</code> , and <code>net.outputs{i}</code> is a structure describing that output.
<code>net.numOutputs</code>	0 or a positive integer (read only)	Number of network outputs according to <code>net.outputConnect</code> .
<code>net.numInputDelays</code>	0 or a positive integer (read only)	Maximum input delay according to all <code>net.inputWeights{i, j}.delays</code> .
<code>net.numLayerDelays</code>	0 or a positive number (read only)	Maximum layer delay according to all <code>net.layerWeights{i, j}.delays</code> .

### Subject Structure Properties

<code>net.inputs</code>	<code>numInputs</code> -by-1 cell array	<code>net.inputs{i}</code> is a structure defining input <code>i</code> .
<code>net.layers</code>	<code>numLayers</code> -by-1 cell array	<code>net.layers{i}</code> is a structure defining layer <code>i</code> .
<code>net.biases</code>	<code>numLayers</code> -by-1 cell array	If <code>net.biasConnect(i)</code> is 1, then <code>net.biases{i}</code> is a structure defining the bias for layer <code>i</code> .
<code>net.inputWeights</code>	<code>numLayers</code> -by- <code>numInputs</code> cell array	If <code>net.inputConnect(i, j)</code> is 1, then <code>net.inputWeights{i, j}</code> is a structure defining the weight to layer <code>i</code> from input <code>j</code> .
<code>net.layerWeights</code>	<code>numLayers</code> -by- <code>numLayers</code> cell array	If <code>net.layerConnect(i, j)</code> is 1, then <code>net.layerWeights{i, j}</code> is a structure defining the weight to layer <code>i</code> from layer <code>j</code> .
<code>net.outputs</code>	1-by- <code>numLayers</code> cell array	If <code>net.outputConnect(i)</code> is 1, then <code>net.outputs{i}</code> is a structure defining the network output from layer <code>i</code> .

### Function Properties

<code>net.adaptFcn</code>	Name of a network adaption function or ''
<code>net.initFcn</code>	Name of a network initialization function or ''

<code>net.performFcn</code>	Name of a network performance function or ''
<code>net.trainFcn</code>	Name of a network training function or ''

**Parameter Properties**

<code>net.adaptParam</code>	Network adaption parameters
<code>net.initParam</code>	Network initialization parameters
<code>net.performParam</code>	Network performance parameters
<code>net.trainParam</code>	Network training parameters

**Weight and Bias Value Properties**

<code>net.IW</code>	<code>numLayers-by-numInputs</code> cell array of input weight values
<code>net.LW</code>	<code>numLayers-by-numLayers</code> cell array of layer weight values
<code>net.b</code>	<code>numLayers-by-1</code> cell array of bias values

**Other Properties**

<code>net.userData</code>	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, the 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];
```

After training your network, you can use `sim` to simulate the network. For example, simulate the network for a two-element input vector.

```
p = [0.5; -0.1];  
y = sim(net,p)
```

## Version History

Introduced before R2006a

### See Also

`sim`

### Topics

"Neural Network Object Properties"

"Neural Network Subobject Properties"

## 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  $0.8326/\text{spread}$ , resulting in radial basis functions that cross 0.5 at weighted inputs of  $\pm \text{spread}$ . 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)
```

## **Version History**

Introduced before R2006a

## **References**

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York, Van Nostrand Reinhold, 1993, pp. 155-61

## **See Also**

sim | newrb | newrbe | newpnn

## 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  $P_{i,k}$  is an  $R_i$ -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	$N_i$ -by- $TS$ cell array	Each element $P_{i,ts}$ is an $R_i$ -by- $Q$ input matrix
T	$N_t$ -by- $TS$ cell array	Each element $T_{i,ts}$ is a $V_i$ -by- $Q$ matrix
Pi	$N_i$ -by-ID cell array	Each element $P_{i,k}$ is an $R_i$ -by- $Q$ matrix, default = <code>[]</code>

and returns a linear network with  $ID$  input delays,  $N_i$  network inputs, and  $N_l$  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  $P_i$ .

```
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:

$$[W \ b] * [P; \text{ones}] = T$$

## Version History

Introduced before R2006a

## See Also

`sim`

## 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  $0.8326/\text{spread}$ , resulting in radial basis functions that cross 0.5 at weighted inputs of  $\pm \text{spread}$ . The second-layer weights  $W2$  are set to T.

## **Version History**

Introduced before R2006a

## **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

## 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 `radbases` 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 `radbases` 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 `radbases` neuron is added with weights equal to that vector.
- 4 The `purelin` layer weights are redesigned to minimize error.

## **Version History**

Introduced before R2006a

**See Also**

sim | newrbe | newgrnn | newpnn

## 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  $0.8326/\text{spread}$ , resulting in radial basis functions that cross 0.5 at weighted inputs of  $\pm \text{spread}$ .

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:

$$[W\{2,1\} \ b\{2\}] * [A\{1\}; \text{ones}] = T$$

## **Version History**

**Introduced before R2006a**

### **See Also**

sim | newrb | newgrnn | newpnn

# nftool

Open Neural Net Fitting app

## Syntax

```
nftool  
nftool("close")
```

## 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”.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

nftool("close") closes the Neural Net Fitting app.

## Algorithms

The **Neural Net Fitting** app leads you through solving a data fitting problem, solving it with a two-layer feed-forward network trained with Levenberg-Marquardt.

## Version History

**Introduced in R2006a**

## See Also

**Neural Net Fitting** | nctool | nprtool | ntstool

## Topics

“Fit Data with a Shallow Neural Network”

## 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)
```

## Version History

Introduced in R2010b

### See Also

nndata | nnsz

## nncorr

Cross correlation between neural network time series

### Syntax

```
nncorr(a,b,maxlag,'flag')
```

### Description

`nncorr(a,b,maxlag,'flag')` takes these arguments,

<code>a</code>	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of <code>N</code> .
<code>b</code>	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of <code>M</code> .
<code>maxlag</code>	Maximum number of time lags
<code>flag</code>	Type of normalization (default = 'none')

and returns an `N`-by-`M` cell array where each  $\{i, j\}$  element is a  $2*\text{maxlag}+1$  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  $1/(N - \text{abs}(k))$ , 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)
```

## Version History

Introduced in R2010b

**See Also**

confusion | regression



# 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)
```

## Version History

Introduced in R2010b

## See Also

`nnsim` | `tonndata` | `fromnndata` | `nndata2sim` | `sim2nndata`

## 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 ith 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)
```

## Version History

Introduced in R2012b

**See Also**

gpu2nndata

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

nndata | sim2nndata | nnsim

## nnsizes

Number of neural data elements, samples, timesteps, and signals

### Syntax

```
[N,Q,TS,M] = nnsizes(X)
```

### Description

`[N,Q,TS,M] = nnsizes(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] = nnsizes(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] = nnsizes(x)
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `numelements` | `numsamples` | `numsignals` | `numtimesteps`

## nnstart

Open Neural Network Start app

### Syntax

```
nnstart  
nnstart("close")
```

### 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.

nnstart("close") closes the Neural Network Start app.

## Version History

**Introduced in R2010b**

### See Also

nctool | nftool | nprtool | ntstool

# nntool

(Removed) Open Network/Data Manager

---

**Note** nntool has been removed. 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.

## Version History

**Introduced before R2006a**

**R2022a: nntool has been removed**

*Errors starting in R2022a*

nntool has been removed. 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

## nntraintool

(Removed) Neural network training tool

---

**Note** `nntraintool` has been removed. 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.

### Version History

**Introduced in R2008a**

**R2022a: `nntraintool` has been removed**

*Errors starting in R2022a*

`nntraintool` has been removed. To train a network and open the training window, use `train` instead.

### See Also

`train`



## 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

#### Convert NARX Network to No Loop Form

This example shows how to create and train 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);
```

Network Diagram

**Training Results**  
Training finished: Reached minimum gradient ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	313	1000
Elapsed Time	-	00:00:04	-
Performance	0.559	3.16e-12	0
Gradient	1.6	8.99e-08	1e-07
Mu	0.001	1e-09	1e+10
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

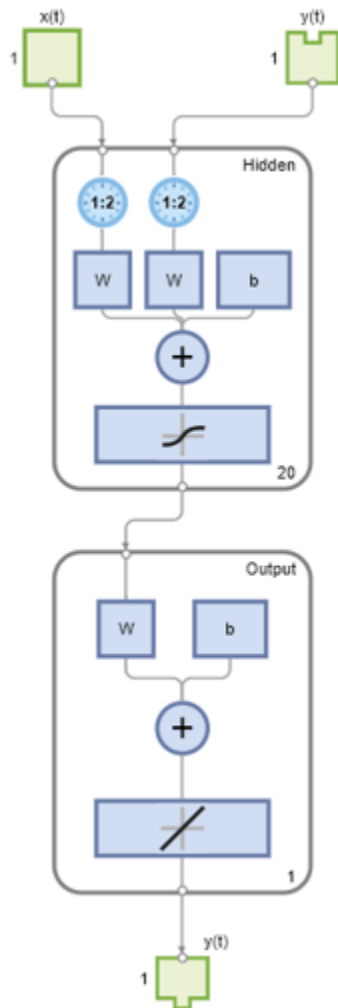
Performance      Training State

Error Histogram      Regression

Time-Series Response      Error

Input-Error Cross-correlation

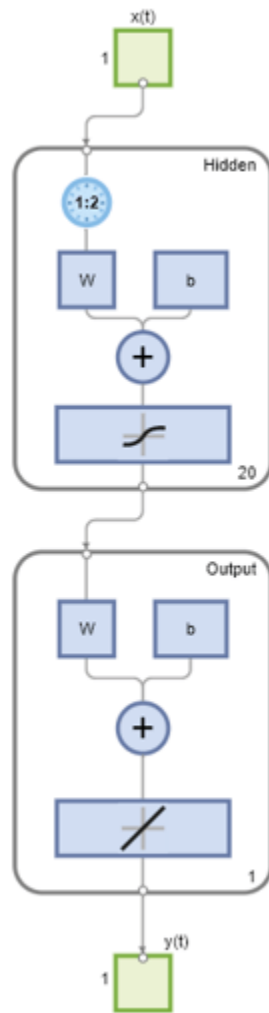
```
view(net)
```



```
Y = net(Xs,Xi,Ai);
```

Now the network is converted to no loop form. The output and second input are no longer associated.

```
net = nolop(net);
view(net)
```



```
[Xs,Xi,Ai] = preparets(net,X,T);
Y = net(Xs,Xi,Ai);
```

## Version History

Introduced in R2010b

### See Also

closeloop | openloop

## 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.

### Version History

Introduced before R2006a

### See Also

`normr`

## 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.

$z = w*p/sum(p)$

## **Version History**

**Introduced before R2006a**

## **See Also**

dotprod

## 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.

## Version History

Introduced before R2006a



**See Also**

normc

## nprtool

Open Neural Net Pattern Recognition app

### Syntax

```
nprtool  
nprtool("close")
```

### Description

`nprtool` opens the **Neural Net Pattern Recognition** app. For more information and an example of its usage, see “Pattern Recognition with a Shallow Neural Network”.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

`nprtool("close")` closes the Neural Net Pattern Recognition app.

### Algorithms

The **Neural Net Pattern Recognition** app leads you through solving a pattern-recognition classification problem using a two-layer feed-forward `patternnet` network with sigmoid output neurons.

## Version History

**Introduced in R2008a**

### See Also

**Neural Net Pattern Recognition** | `nctool` | `nftool` | `ntstool`

### Topics

“Pattern Recognition with a Shallow Neural Network”

# ntstool

Open Neural Net Time Series app

## 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”.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

ntstool("close") closes the Neural Net Time Series app.

## Version History

Introduced in R2010b

## See Also

**Neural Net Time Series** | nctool | nftool | nprtool

## Topics

“Shallow Neural Network Time-Series Prediction and Modeling”

## 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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an $R \times Q$ matrix (or $N \times TS$ cell array of $R \times Q$ matrices)
<code>T</code>	Targets, an $S \times Q$ matrix (or $M \times TS$ cell array of $S \times Q$ matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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,  $R_i$  and  $S_i$  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)
```

## **Version History**

**Introduced in R2010b**

### **See Also**

bttderiv | defaultderiv | fpderiv | num5deriv | staticderiv

## 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.

$$y_1 = y(x + 2dx)$$

$$y_2 = y(x + dx)$$

$$y_3 = y(x - dx)$$

$$y_4 = y(x - 2dx)$$

$$\frac{dy}{dx} = \frac{-y_1 + 8y_2 - 8y_3 + y_4}{12dx}$$

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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an $R \times Q$ matrix (or $N \times TS$ cell array of $R \times Q$ matrices)
<code>T</code>	Targets, an $S \times Q$ matrix (or $M \times TS$ cell array of $S \times Q$ matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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,  $R_i$  and  $S_i$  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)
```

## **Version History**

**Introduced in R2010b**

### **See Also**

[bttderiv](#) | [defaultderiv](#) | [fpderiv](#) | [num2deriv](#) | [staticderiv](#)

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

[nndata](#) | [nnsample](#) | [getelements](#) | [setelements](#) | [catelements](#) | [numsamples](#) | [numsignals](#) | [numtimesteps](#)



# 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)
```

## Version History

**Introduced in R2010b**

## See Also

`numnan` | `nndata` | `nnsizes`

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `nnsz`

# 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)
```

## Version History

**Introduced in R2010b**

## See Also

`nndata` | `nnsample` | `getsamples` | `setsamples` | `catsamples` | `numelements` | `numsignals` | `numtimesteps`

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `nnsample` | `getsignals` | `setsignals` | `catsignals` | `numelements` | `numsamples` | `numtimesteps`

# 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)
```

## Version History

**Introduced in R2010b**

## See Also

`nndata` | `nnsz` | `gettimesteps` | `settimesteps` | `cattimesteps` | `numelements` | `numsamples` | `numsignals`

## 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

This example shows how to create a NARX network in open-loop form, convert it to closed-loop form, and then convert it 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);
```

Network Diagram

### Training Results

Training finished: Reached minimum gradient ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	686	1000
Elapsed Time	-	00:00:05	-
Performance	2.05	2.08e-11	0
Gradient	3.24	9.97e-08	1e-07
Mu	0.001	1e-08	1e+10
Validation Checks	0	0	6

### Training Algorithms

Data Division: Random dividerand

Training: Levenberg-Marquardt trainlm

Performance: Mean Squared Error mse

Calculations: MEX

### Training Plots

Performance

Training State

Error Histogram

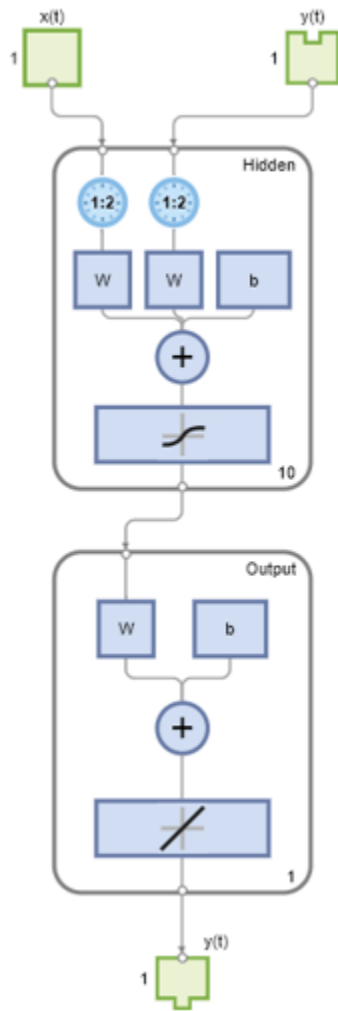
Regression

Time-Series Response

Error

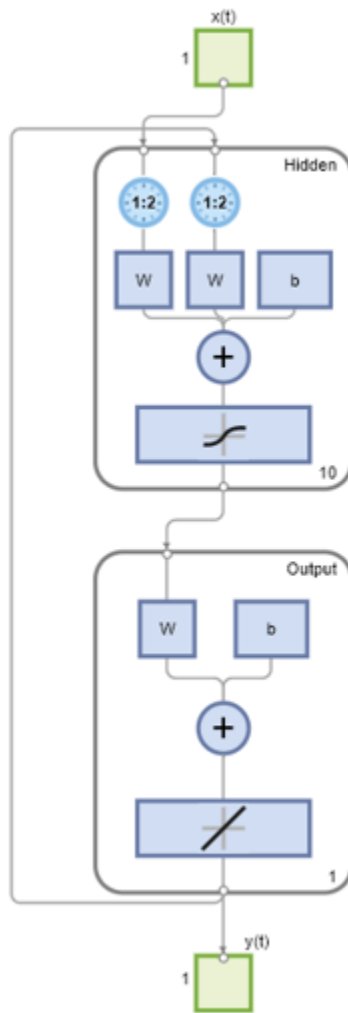
Input-Error Cross-correlation

view(net)

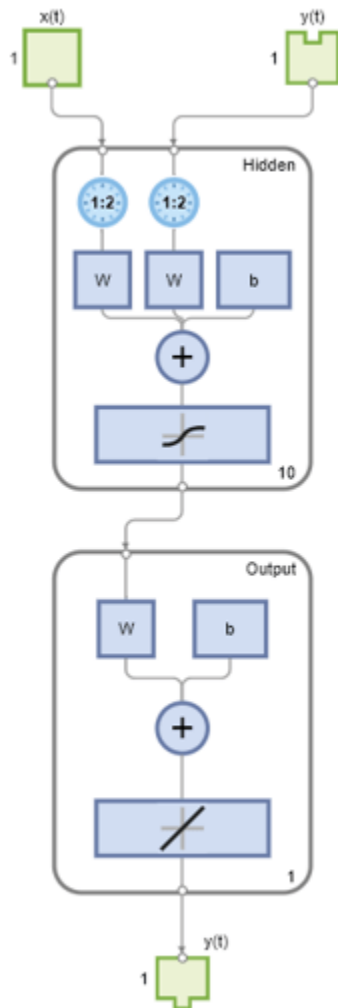


```
Yopen = net(Xs,Xi,Ai);  
net = closeloop(net);  
view(net)
```





```
[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`.

## Version History

Introduced in R2010b

### See Also

`closeLoop` | `narnet` | `narxnet` | `noLoop`

# 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	15	1000
Elapsed Time	-	00:00:03	-
Performance	0.549	0.00804	0
Gradient	0.406	0.0154	1e-06
Validation Checks	0	6	6

**Training Algorithms**  
Data Division: Random dividerand  
Training: Scaled Conjugate Gradient trainscg  
Performance: Cross Entropy crossentropy  
Calculations: MEX

**Training Plots**

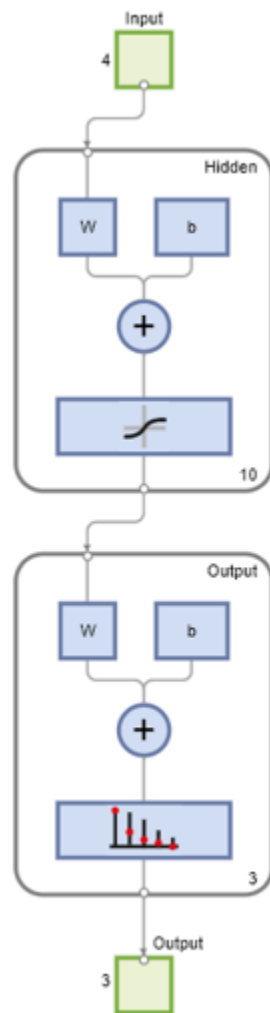
Performance Training State

Error Histogram Confusion

Receiver Operating Characteristic

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-Ribière 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.

## Version History

Introduced in R2010b

### See Also

[competlayer](#) | [lvqnet](#) | [network](#) | [nprtool](#) | [selforgmap](#)

### Topics

[“Pattern Recognition with a Shallow Neural Network”](#)

[“Neural Network Object Properties”](#)

[“Neural Network Subobject Properties”](#)

## 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);
```



Network Diagram

### Training Results

Training finished: Met performance criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	3	1000	▲
Elapsed Time	-	00:00:03	-	
Performance	0.25	0	0	▼

### Training Algorithms

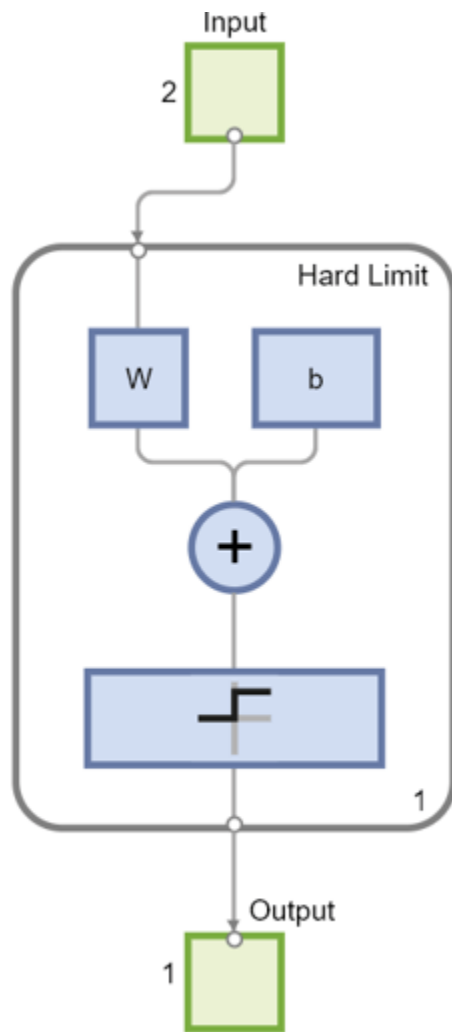
Data Division: Training Only `dividetrain`  
Training: Cyclical Weight/Bias Rule `trainc`  
Performance: Mean Absolute Error `mae`  
Calculations: MATLAB

### Training Plots

Performance Training State

Confusion

```
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.

## Version History

Introduced in R2010b

**See Also**

[preparets](#) | [removedelay](#) | [patternnet](#) | [timedelaynet](#) | [narnet](#) | [narxnet](#)

## 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.

**Version History****Introduced in R2010b****See Also**

train | configure | init

## plotconfusion

Plot classification confusion matrix

### Syntax

```
plotconfusion(targets, outputs)
plotconfusion(targets, outputs, name)
plotconfusion(targets1, outputs1, name1, targets2, outputs2, name2, ..., targetsn, ou
tputsn, 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, ou tputsn, 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.

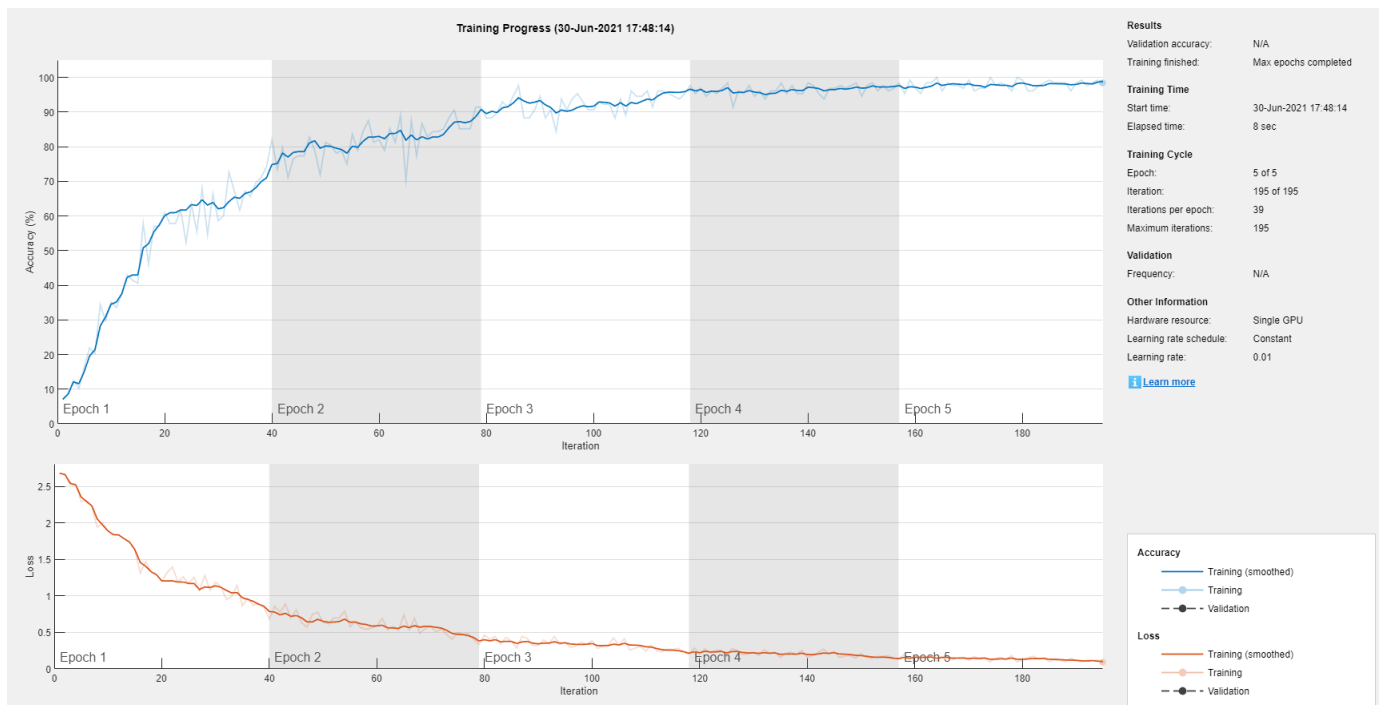
```
layers = [
    imageInputLayer([28 28 1])

    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);
```

Network Diagram

### Training Results

Training finished: Met validation criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	14	1000
Elapsed Time	-	00:00:03	-
Performance	1.33	0.0368	0
Gradient	1.79	0.0108	1e-06
Validation Checks	0	6	6

### Training Algorithms

Data Division: Random dividerand  
 Training: Scaled Conjugate Gradient trainscg  
 Performance: Cross Entropy crossentropy  
 Calculations: MEX

### Training Plots

Performance

Training State

Error Histogram

Confusion

Receiver Operating Characteristic

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
```

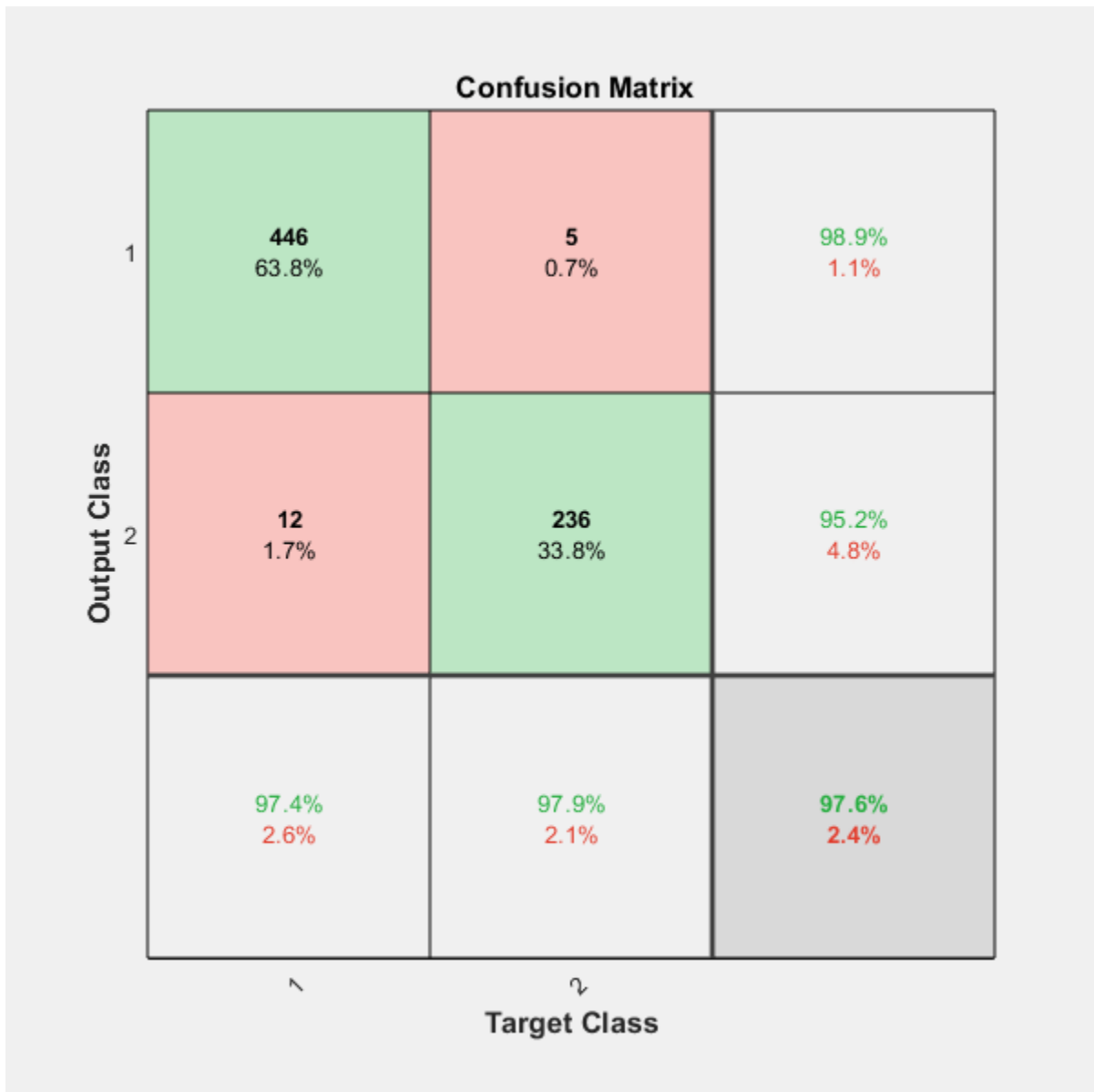
```

0.9980  0.9979  0.9894  0.0578  0.9614  0.9960  0.0026  0.0023  0.0084  0.
0.0020  0.0021  0.0106  0.9422  0.0386  0.0040  0.9974  0.9977  0.9916  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

## Version History

Introduced in R2008a

**See Also**

`trainNetwork` | `trainingOptions`

## 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.

### Version History

Introduced before R2006a

### See Also

errsurf | plotes

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000
Elapsed Time	-	00:00:02	-
Performance	0.427	0.00692	0
Gradient	1.11	0.00215	1e-07
Mu	0.001	0.0001	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

Regression

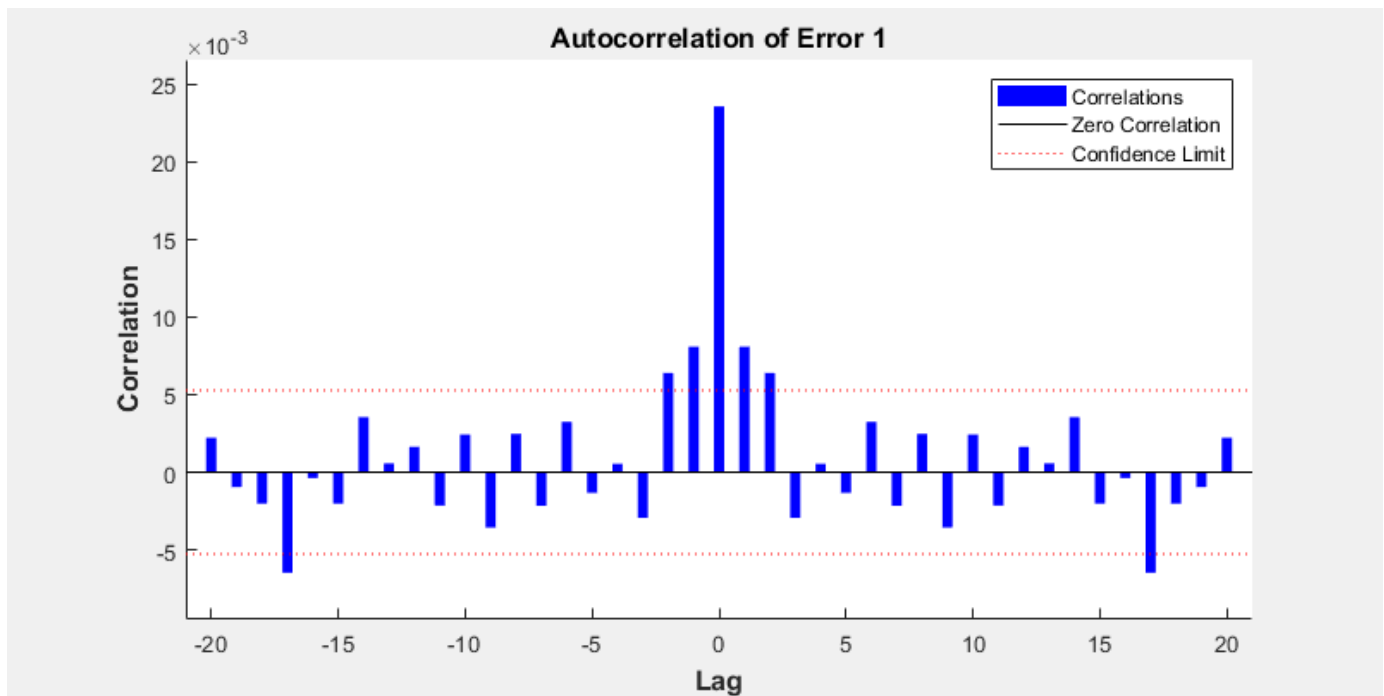
Time-Series Response

Error

Input-Error Cross-correlation

```
Y = net(Xs, Xi, Ai);
E = gsubtract(Ts, Y);
ploterrcorr(E)
```





## Version History

Introduced in R2010b

## See Also

`plotinerrcorr` | `plotresponse`

## 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.

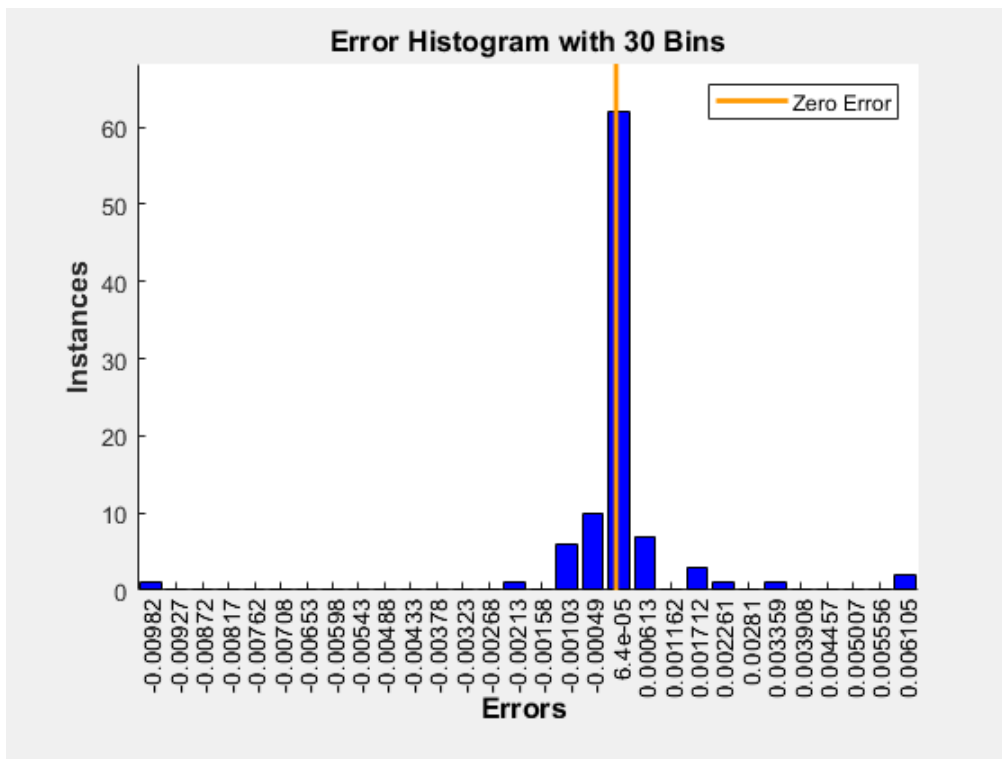
### Examples

#### 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.

## Version History

Introduced in R2010b

### See Also

plotconfusion | ploterrcorr | plotinerrcorr

## plots

Plot error surface of single-input neuron

### Syntax

```
plots(WV,BV,ES,V)
```

### Description

`plots(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])

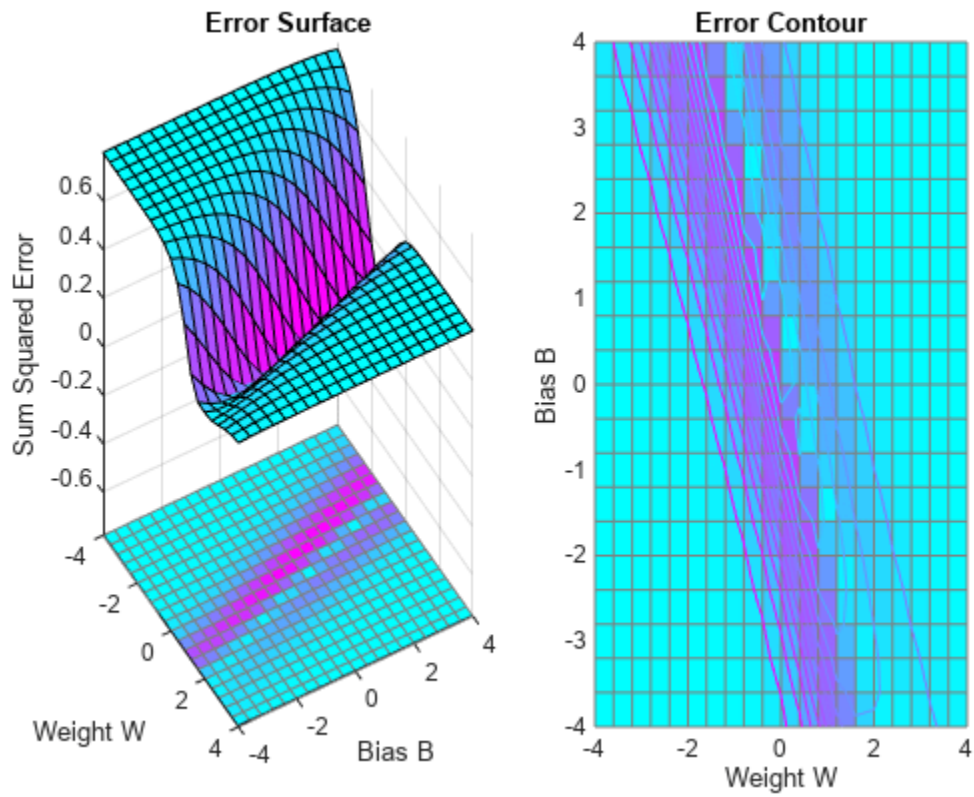
and plots the error surface with a contour underneath.

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');  
plots(wv,bv,ES,[60 30])
```



## Version History

Introduced before R2006a

## See Also

errsurf

## 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

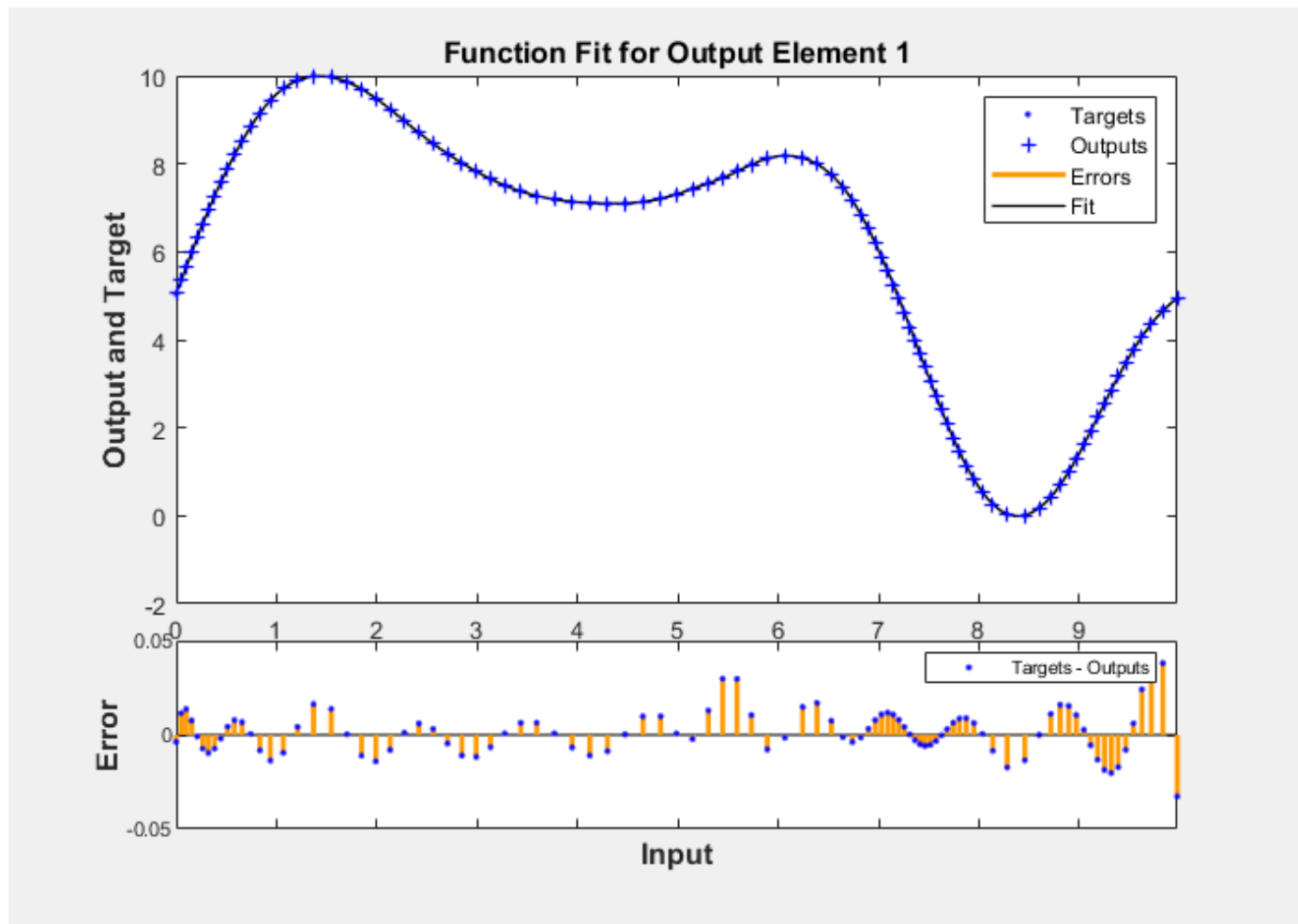
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	19	1000
Elapsed Time	-	00:00:05	-
Performance	324	9.25e-05	0
Gradient	414	0.000292	1e-07
Mu	0.001	1e-05	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

Performance Training State  
Error Histogram Regression

```
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.



## **Version History**

Introduced in R2008a

### **See Also**

plottrainstate

## 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion 

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000

**Training Algorithms**  
Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

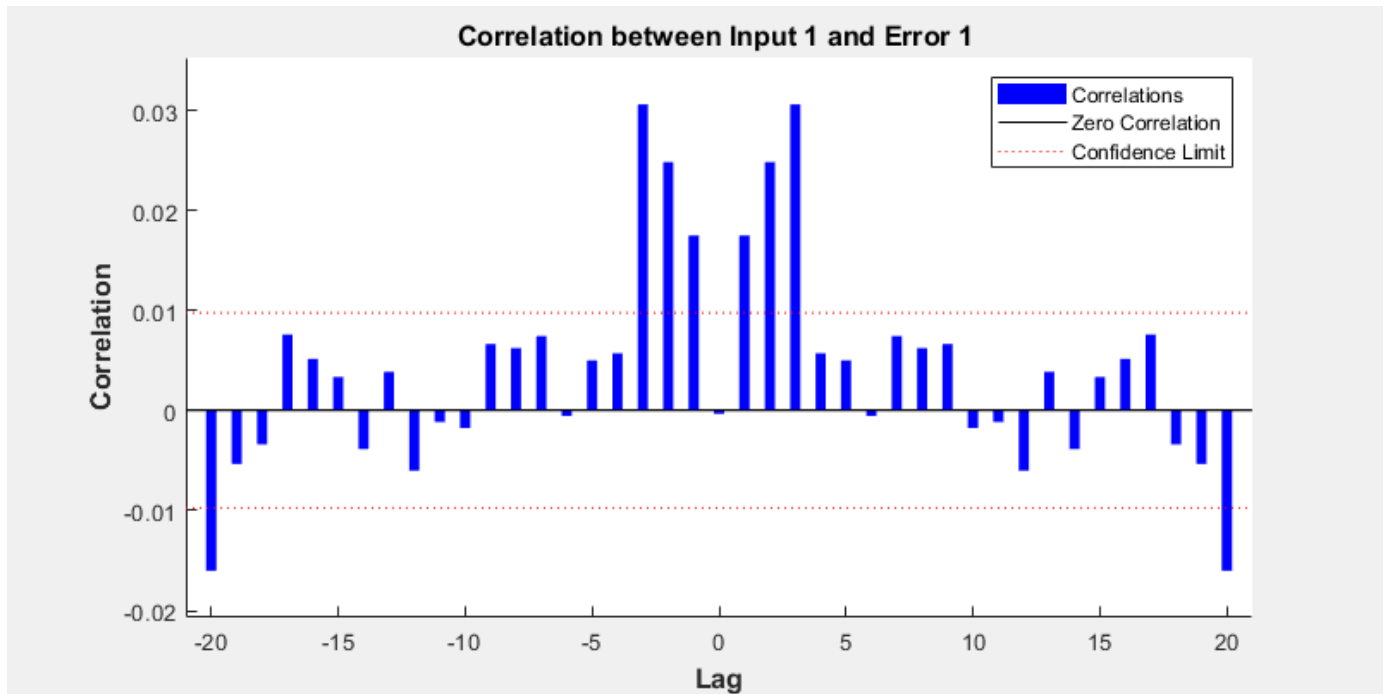
Performance Training State

Error Histogram Regression

Time-Series Response Error

Input-Error Cross-correlation

```
Y = net(Xs, Xi, Ai);
E = gsubtract(Ts, Y);
plotinerrcorr(Xs, E)
```



## Version History

Introduced in R2010b

### See Also

`ploterrcorr` | `plotresponse` | `ploterrhist`

## 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.

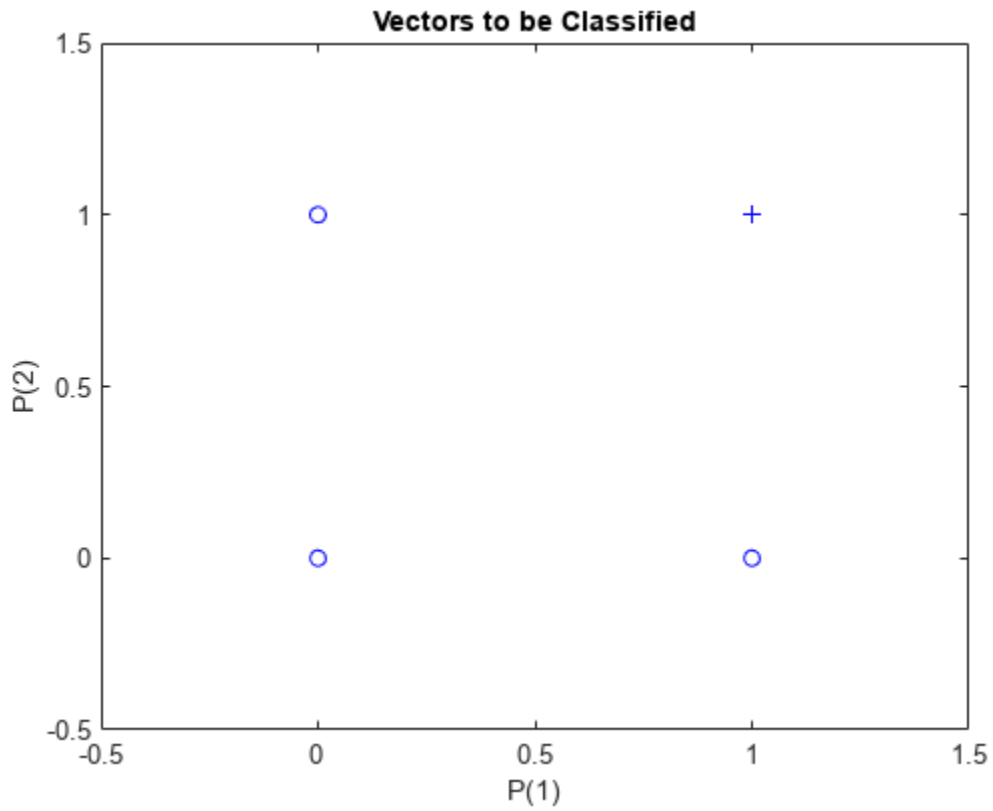
This function does not change the current axis and is intended to be called after plotpv.

### Examples

#### Plot Classification Line

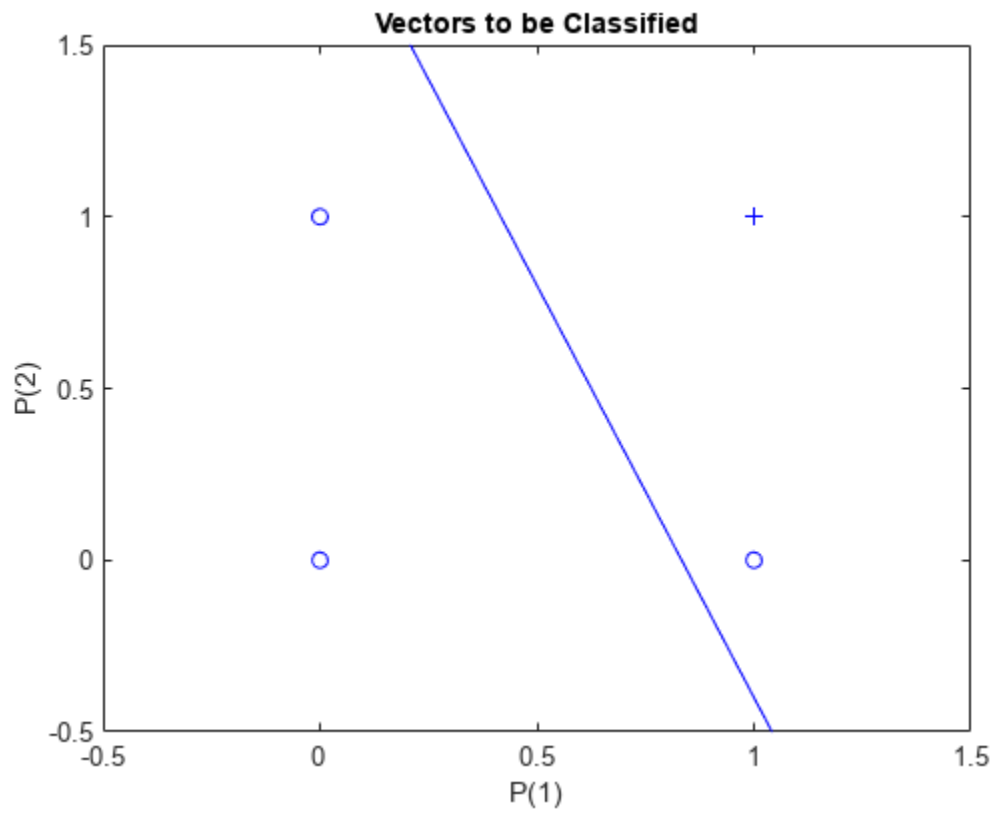
The code below defines and plots the inputs and targets for a perceptron:

```
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})
```



## Version History

Introduced before R2006a

## See Also

plotpv

## 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);
```



Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

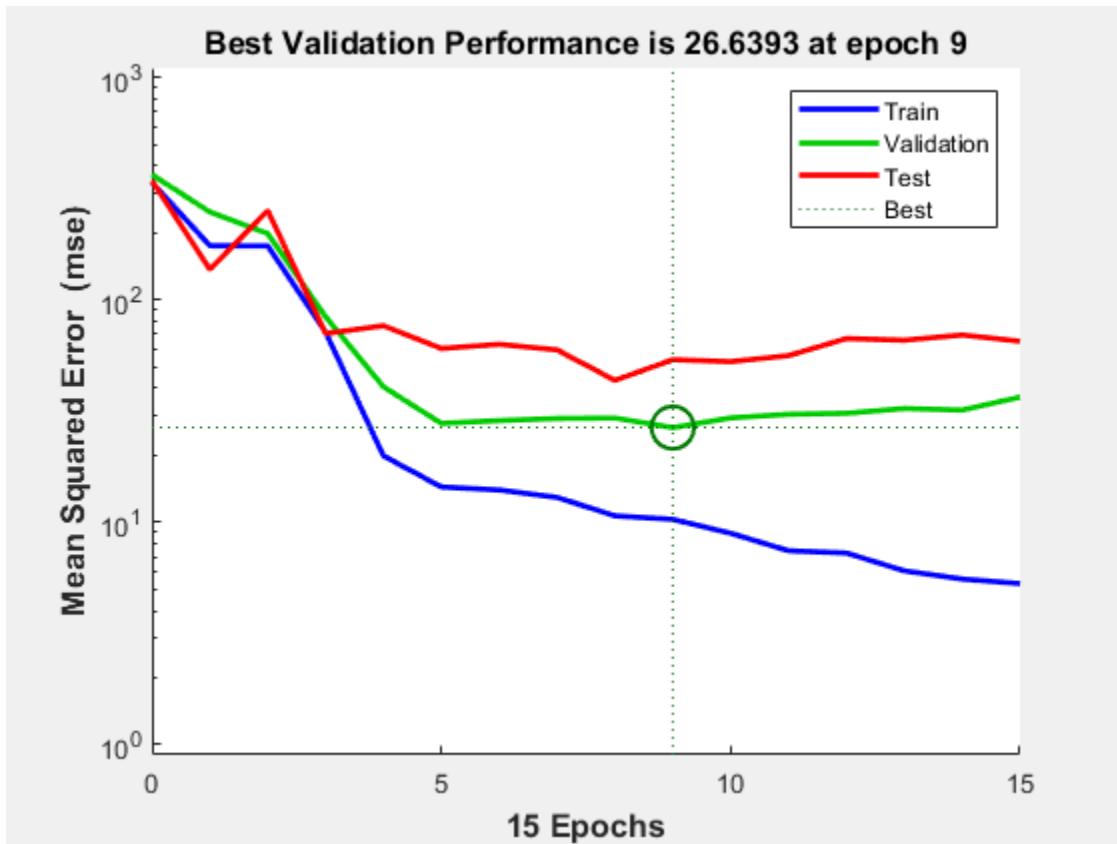
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	15	1000
Elapsed Time	-	00:00:03	-
Performance	337	5.3	0
Gradient	1.72e+03	62.2	1e-07
Mu	0.001	0.001	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

Performance Training State  
Error Histogram Regression

```
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`)

## **Version History**

Introduced in R2008a

### **See Also**

plottrainstate

## 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]
---	--

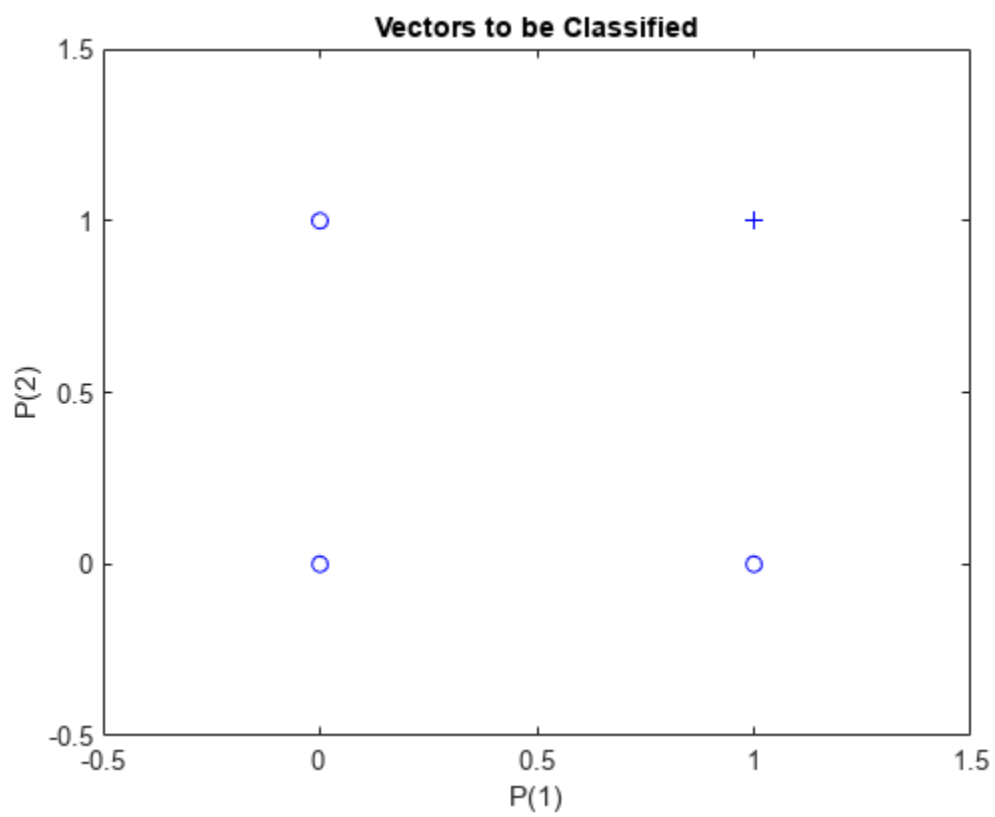
and plots the column vectors with limits set by V.

### 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)
```



## Version History

Introduced before R2006a

## See Also

plotpc

## 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

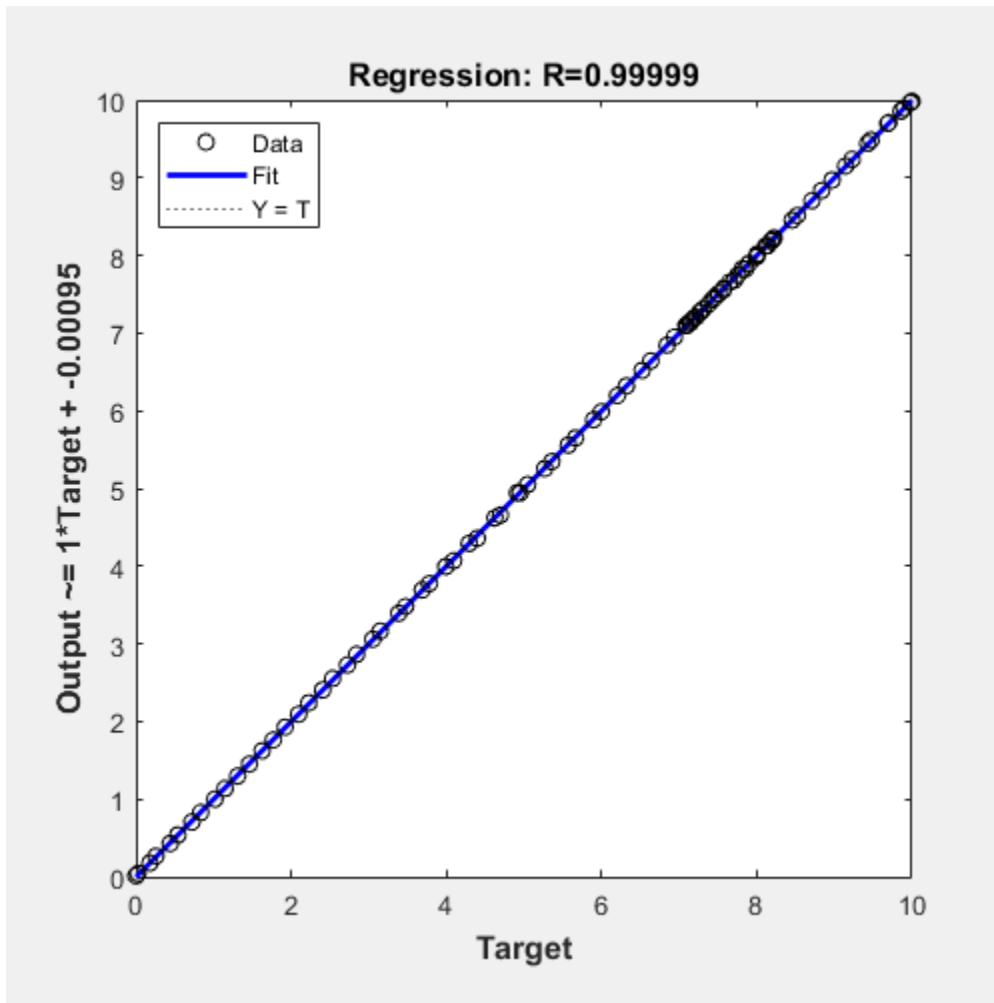
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	19	1000
Elapsed Time	-	00:00:03	-
Performance	324	9.25e-05	0
Gradient	414	0.000292	1e-07
Mu	0.001	1e-05	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

Performance Training State  
Error Histogram Regression

```
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.

## Version History

Introduced in R2008a



**See Also**

plottrainstate

## 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000

**Training Algorithms**  
Data Division: Random dividerand  
Training: Levenberg-Marquardt trainlm  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

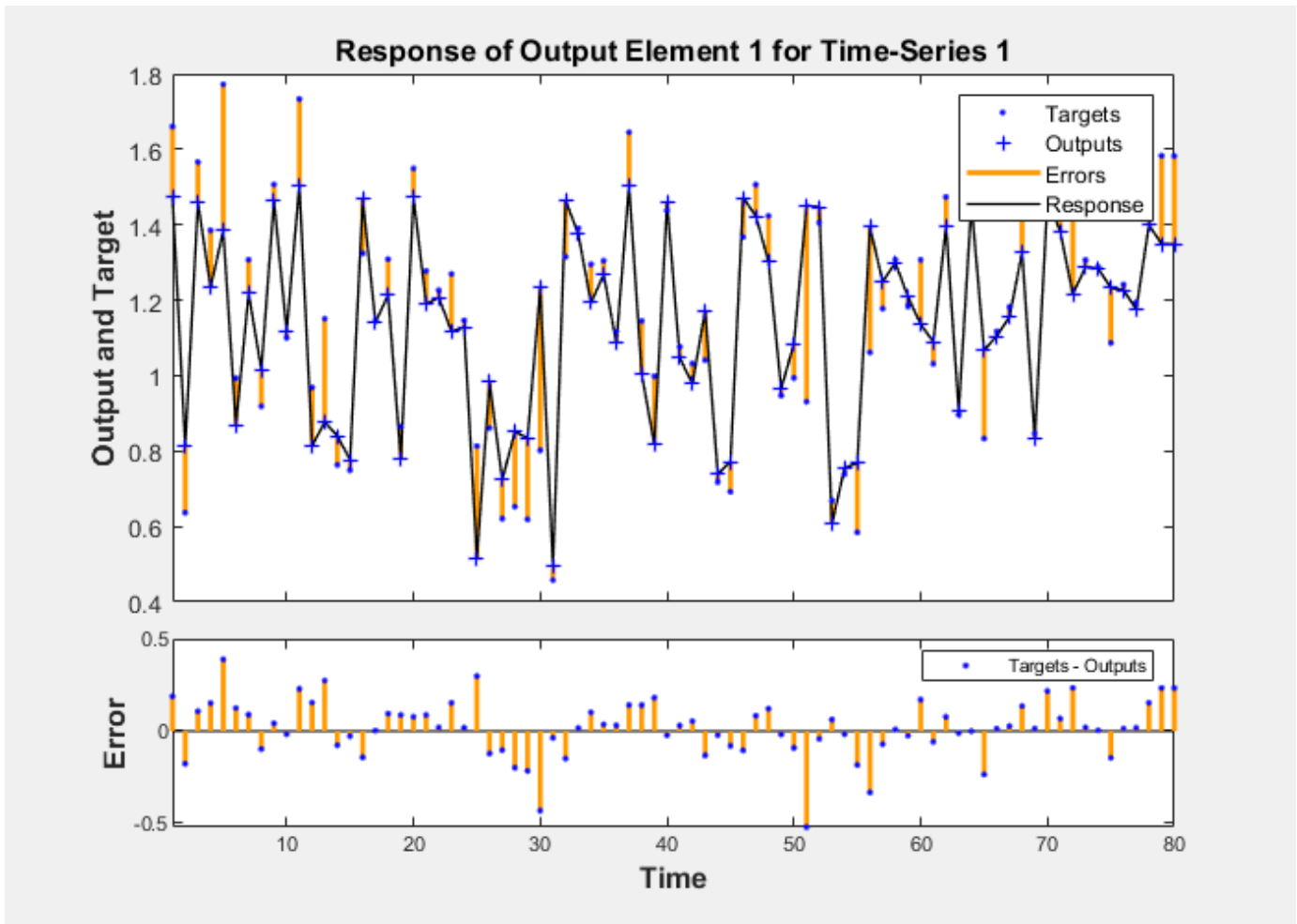
Performance Training State

Error Histogram Regression

Time-Series Response Error

Input-Error Cross-correlation

```
Y = net(Xs,Xi,Ai);
plotresponse(Ts,Y)
```



## Version History

Introduced in R2010b

## See Also

`ploterrcorr` | `plotinerrcorr` | `ploterrhist`

# 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.

---

**Tip** `plotroc` does not support categorical targets. To plot ROC metrics for categorical targets, use `rocmetrics`.

---

`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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	38	1000
Elapsed Time	-	00:00:03	-
Performance	0.529	0.0138	0
Gradient	0.401	0.00141	1e-06
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Scaled Conjugate Gradient trainscg  
 Performance: Cross Entropy crossentropy  
 Calculations: MEX

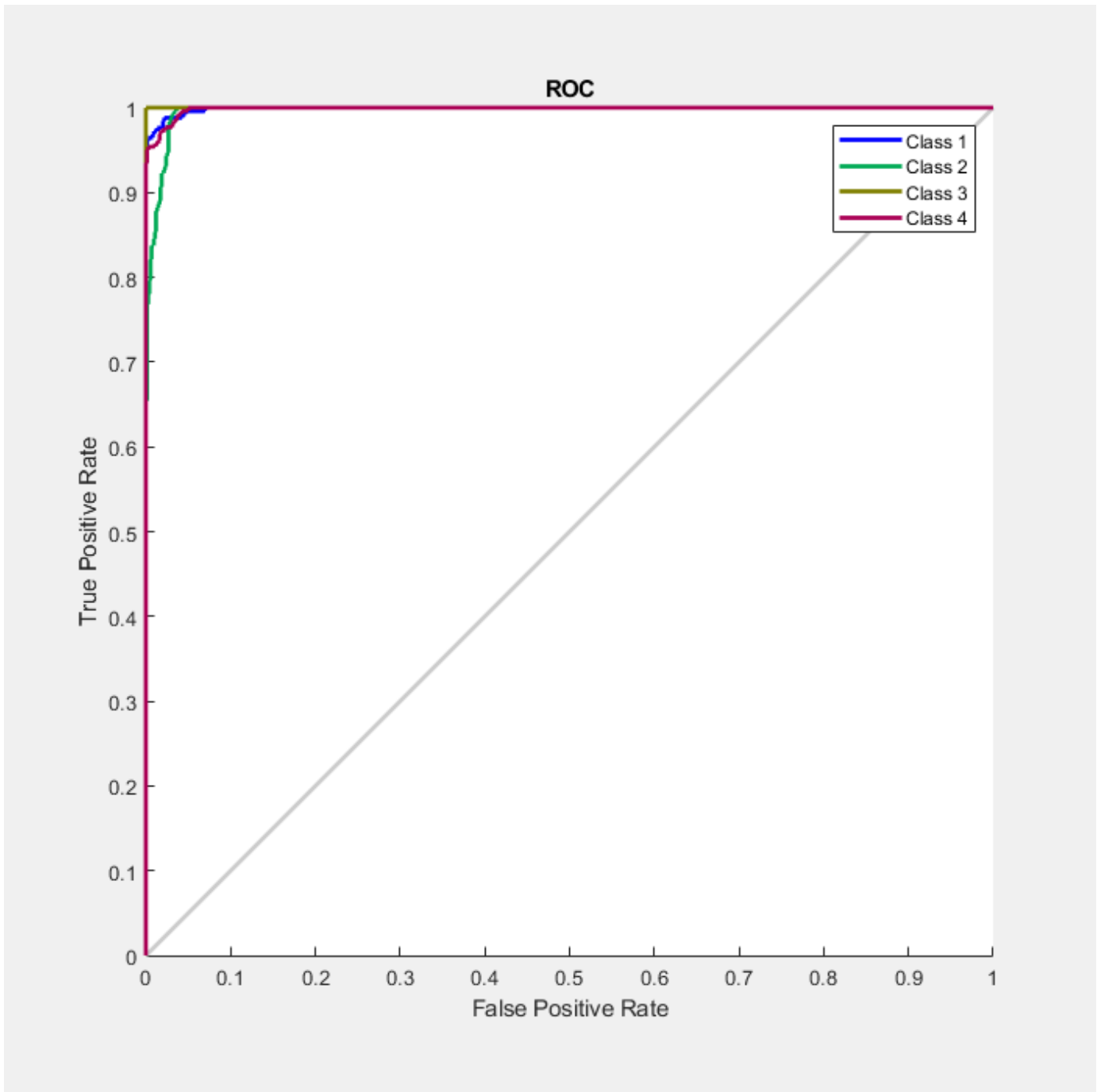
**Training Plots**

Performance      Training State

Error Histogram      Confusion

Receiver Operating Characteristic

```
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.

**Version History**

**Introduced in R2008a**

**See Also**

roc



# 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)

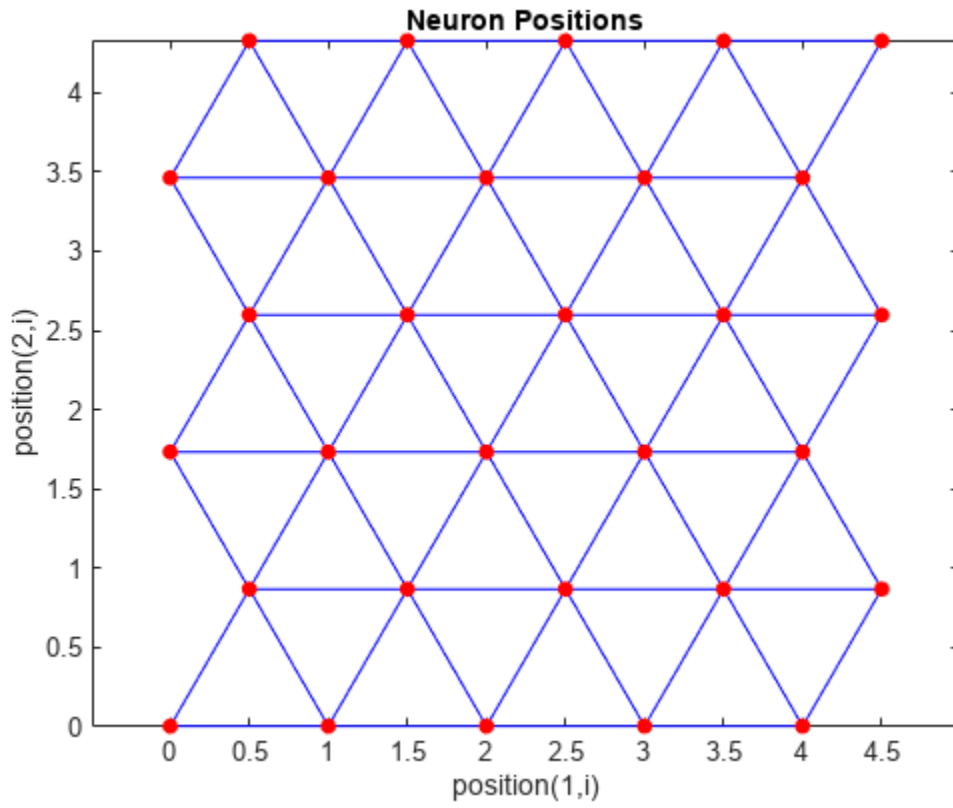
and plots the neuron's weight vectors with connections between weight vectors whose neurons are within a distance of 1.

## Examples

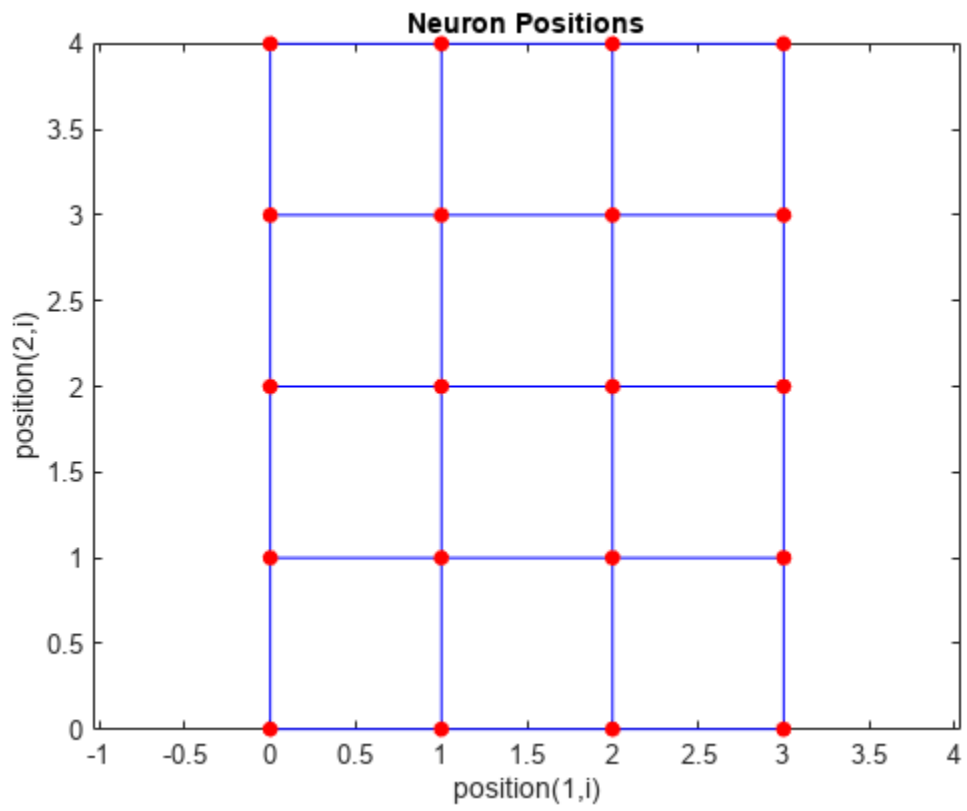
### Plot Self-Organizing Maps

These examples generate plots of various layer topologies.

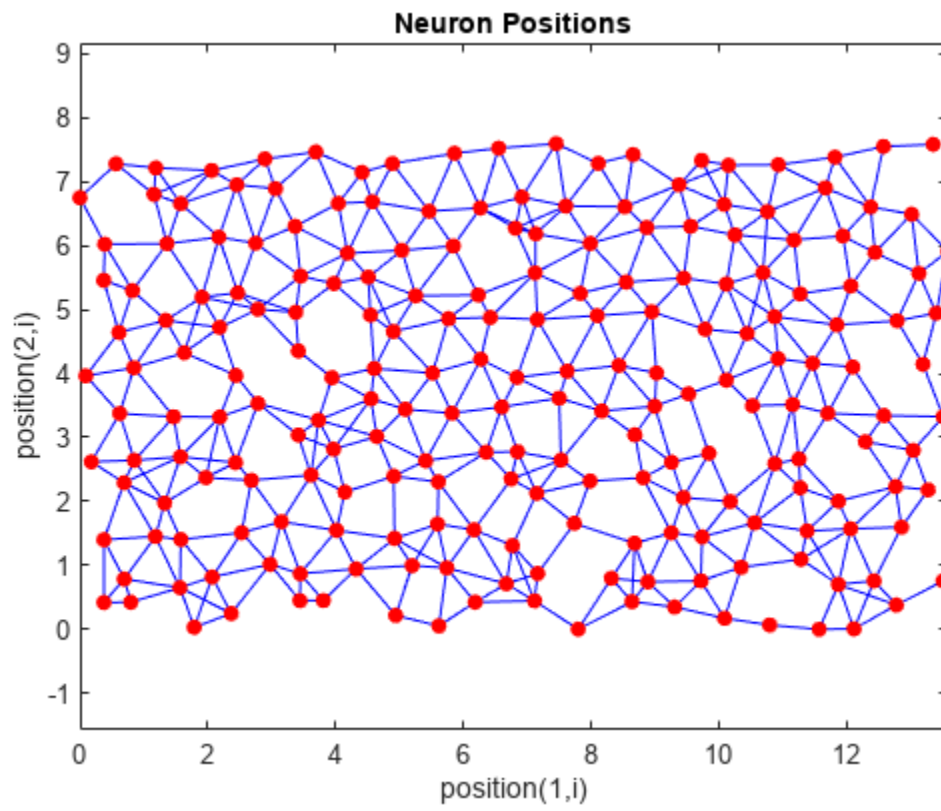
```
pos = hextop([5 6]);
plotsom(pos)
```



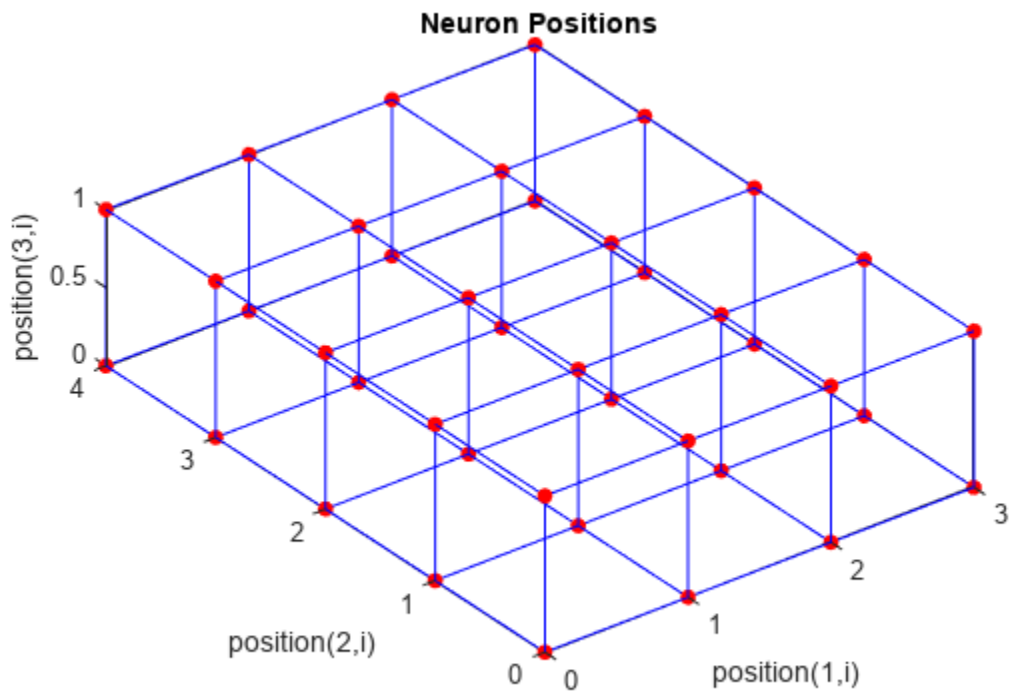
```
pos = gridtop([4 5]);  
plotsom(pos)
```



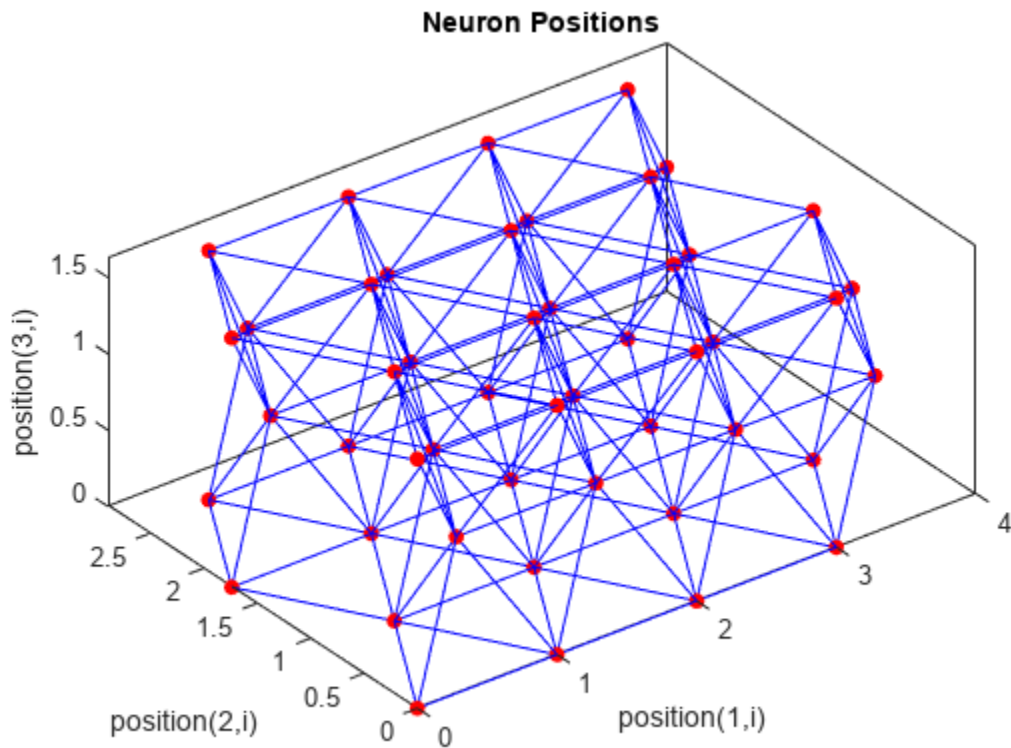
```
pos = randtop([18 12]);  
plotsom(pos)
```



```
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.

## Version History

Introduced before R2006a

## See Also

`learnsom`

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Reached maximum number of epochs ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	200	200	▲
Elapsed Time	-	00:00:03	-	▼

**Training Algorithms**  
 Data Division: Batch Weight/Bias Rule trainbu  
 Performance: Mean Squared Error mse  
 Calculations: MATLAB

**Training Plots**

SOM

SOM Neighbor Connections

SOM Neighbor Distances

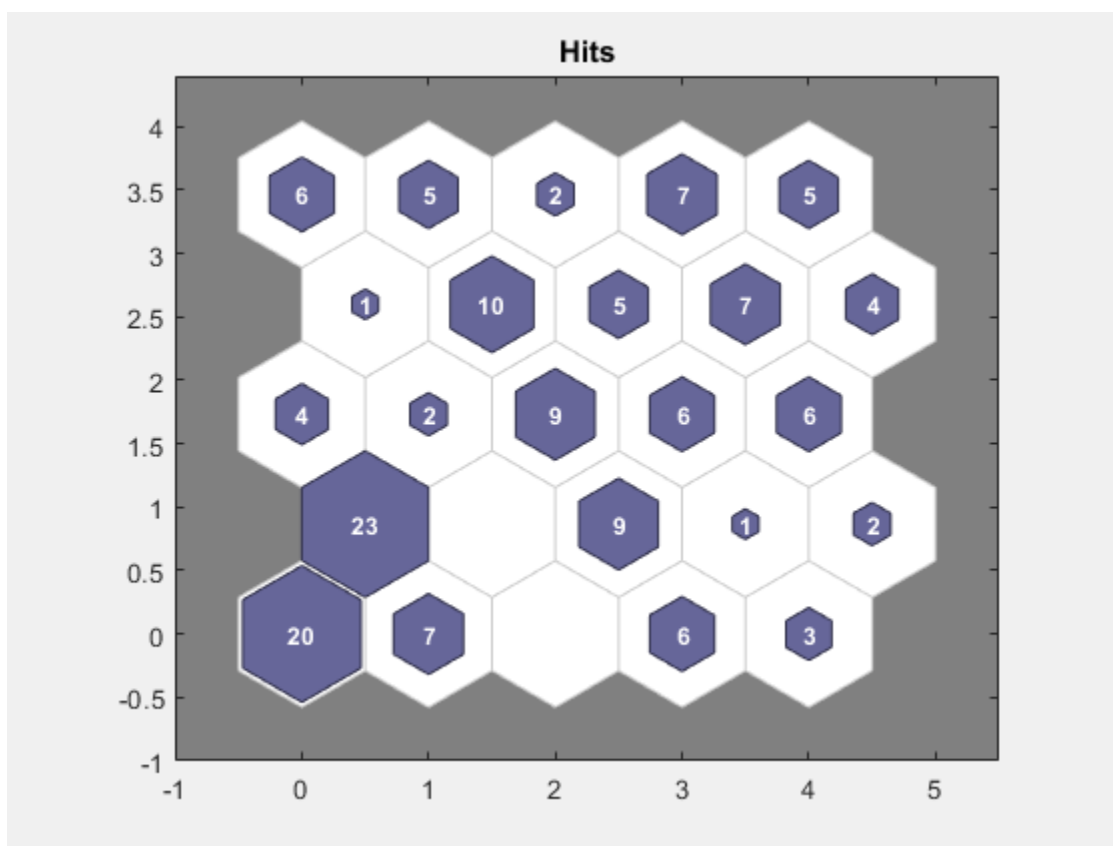
SOM Input Planes

SOM Sample Hits

SOM Weight Positions

`plotsomhits(net,x)`





## Version History

Introduced in R2008a

## See Also

plotsomplanes

# 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);
```

Network Diagram

### Training Results

Training finished: Reached maximum number of epochs 

### Training Progress

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	200	200	▲
Elapsed Time	-	00:00:03	-	▼

### Training Algorithms

Data Division: Batch Weight/Bias Rule trainbu

Performance: Mean Squared Error mse

Calculations: MATLAB

### Training Plots

SOM

SOM Neighbor Connections

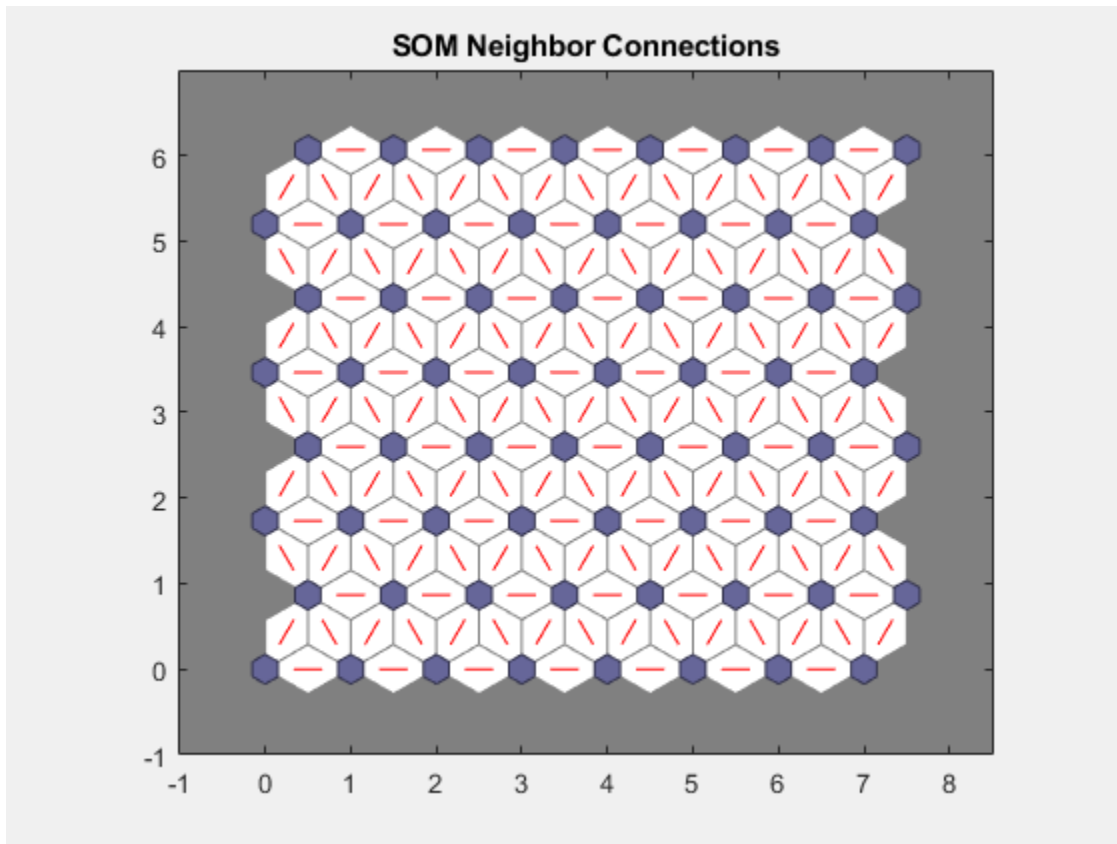
SOM Neighbor Distances

SOM Input Planes

SOM Sample Hits

SOM Weight Positions

plotsomnc(net)



## Version History

Introduced in R2008a

## See Also

`plotsomnd` | `plotsomplanes` | `plotsomhits`

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Reached maximum number of epochs ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200

**Training Algorithms**  
 Data Division: Batch Weight/Bias Rule trainbu  
 Performance: Mean Squared Error mse  
 Calculations: MATLAB

**Training Plots**

SOM

SOM Neighbor Connections

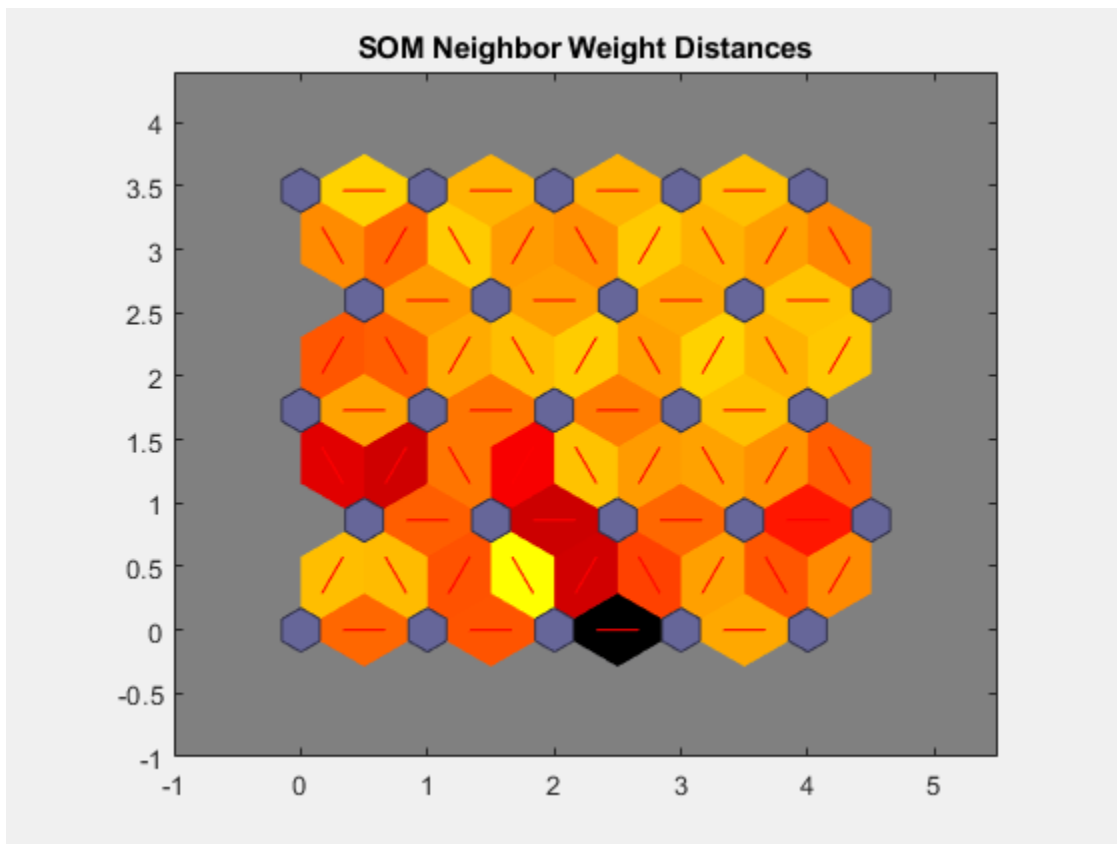
SOM Neighbor Distances

SOM Input Planes

SOM Sample Hits

SOM Weight Positions

plotsomnd(net)



## Version History

Introduced in R2008a

## See Also

[plotsomhits](#) | [plotsomnc](#) | [plotsomplanes](#)

## 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);
```



Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs 

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	200	200	▲
				▼

**Training Algorithms**  
Data Division: Batch Weight/Bias Rule trainbu  
Performance: Mean Squared Error mse  
Calculations: MATLAB

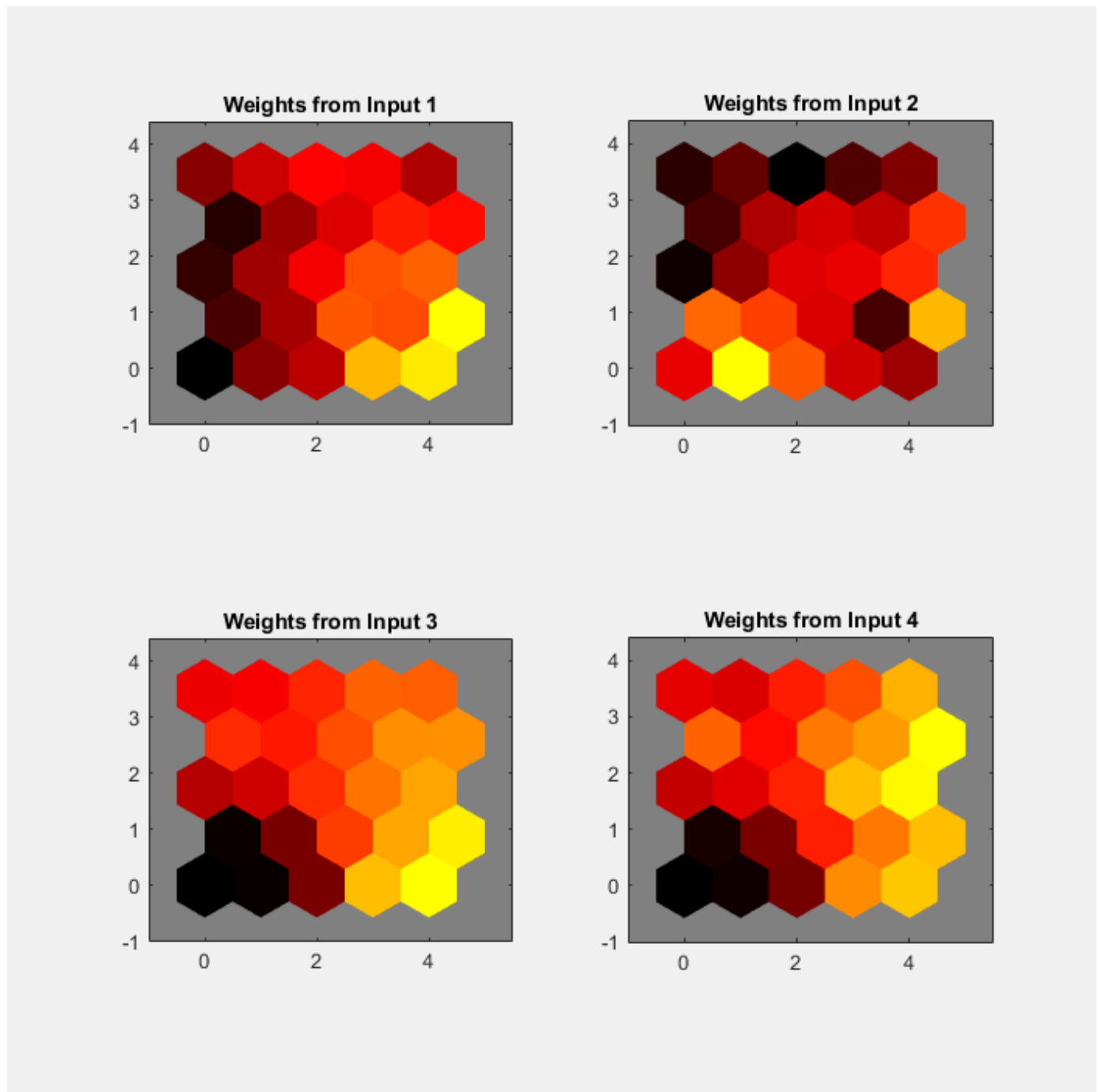
**Training Plots**

SOM SOM Neighbor Connections

SOM Neighbor Distances SOM Input Planes

SOM Sample Hits SOM Weight Positions

plotsomplanes(net)



## Version History

Introduced in R2008a

### See Also

`plotsomhits` | `plotsomnc` | `plotsomnd`

# 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);
```

Network Diagram

**Training Results**  
 Training finished: Reached maximum number of epochs ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	200	200	▲
Elapsed Time	-	00:00:04	-	▼

**Training Algorithms**  
 Data Division: Batch Weight/Bias Rule `trainbu`  
 Performance: Mean Squared Error `mse`  
 Calculations: MATLAB

**Training Plots**

SOM

SOM Neighbor Connections

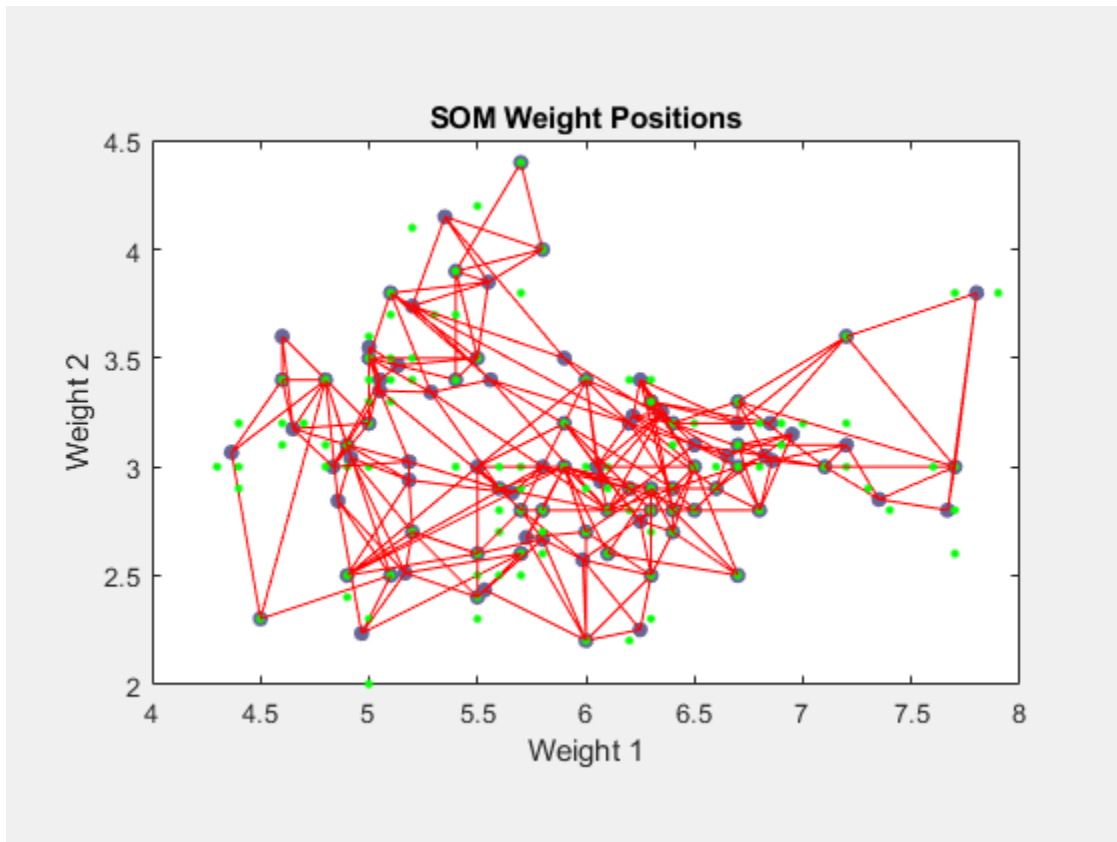
SOM Neighbor Distances

SOM Input Planes

SOM Sample Hits

SOM Weight Positions

`plotsompos(net,x)`



## Version History

Introduced in R2008a

## See Also

`plotsomnd` | `plotsomplanes` | `plotsomhits`

## plotsomtop

Plot self-organizing map topology

### Syntax

```
plotsomtop(net)
```

### Description

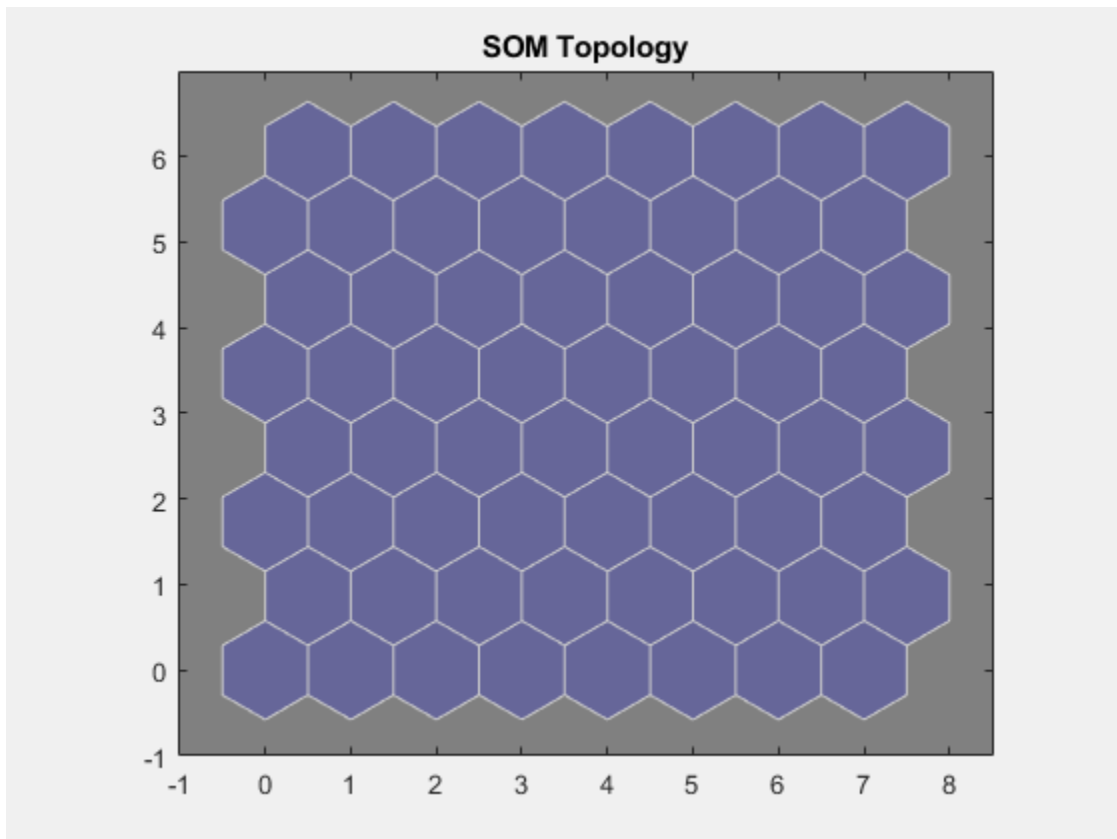
`plotsomtop(net)` plots the topology of a SOM layer.

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)
```



## **Version History**

Introduced in R2008a

### **See Also**

plotsomnd | plotsomplanes | plotsomhits

## 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);
```



Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	15	1000
Elapsed Time	-	00:00:03	-
Performance	337	5.3	0
Gradient	1.72e+03	62.2	1e-07
Mu	0.001	0.001	1e+10
Validation Checks	0	6	6

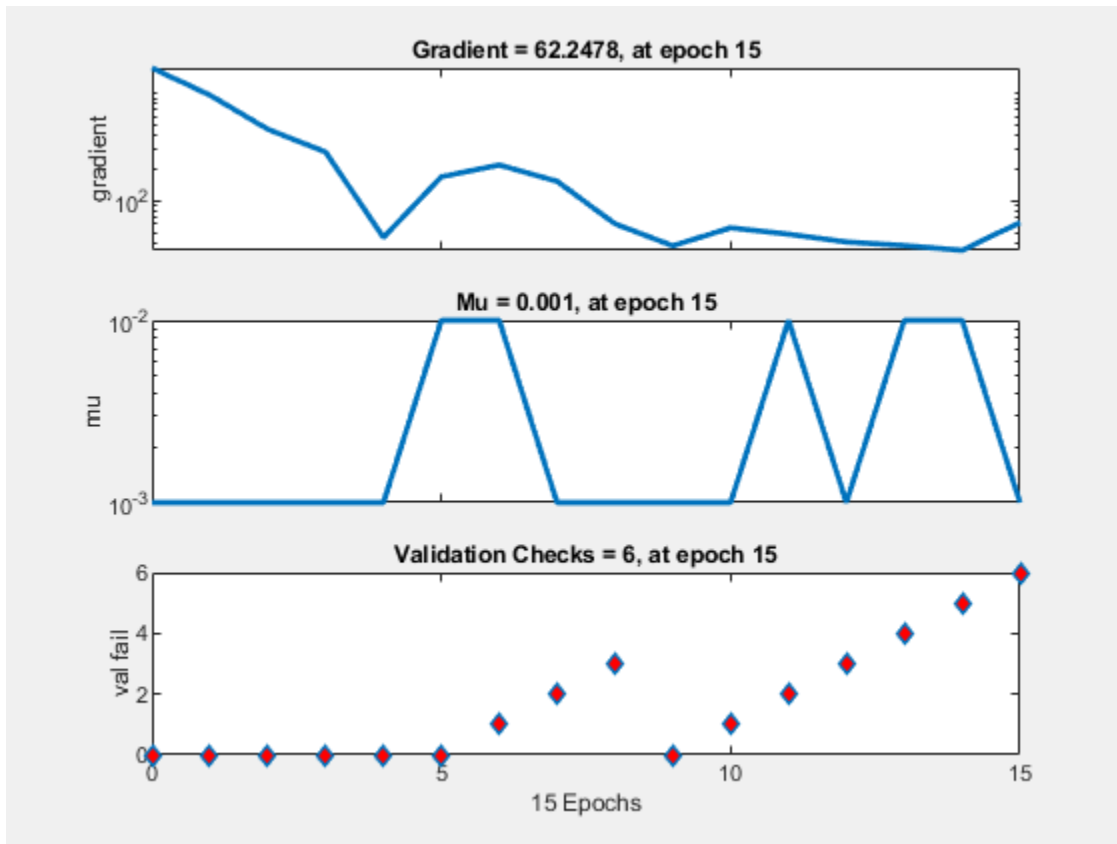
**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance Training State

Error Histogram Regression

plottrainstate(tr)



## Version History

Introduced in R2008a

### See Also

`plotfit` | `plotperform` | `plotregression`

# plotv

(To be removed) Plot vectors as lines from origin

---

**Note** `plotv` will be removed in a future release. Use `plot` instead. For information on updating your code, see “Version History”

---

## 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)

Line Style	Description
--	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
c	Cyan
r	Red
g	Green
b	Blue
w	White
k	Black

## Version History

Introduced before R2006a

### R2023a: To be removed

*Not recommended starting in R2023a*

`plotv` will be removed. Use `plot` instead. The `plot` function offers greater flexibility over figure properties and is consistent with other MATLAB plotting functions.

There are several differences between these functions that require updates to your code. For example, let `M` be a matrix containing three 2-element vectors.

```
M = [-0.4 0.7 0.2 ;  
     -0.5 0.1 0.5];
```

Plot the column vectors as lines from the origin.

Using plotv	Using plot
<code>plotv(M, '-')</code>	<code>origin = zeros(1,size(M,2)); plot([origin; M(1,:)],[origin; M(2,:)]);</code>

## See Also

`plot` | `plotvec` | `plotfit`

## 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.

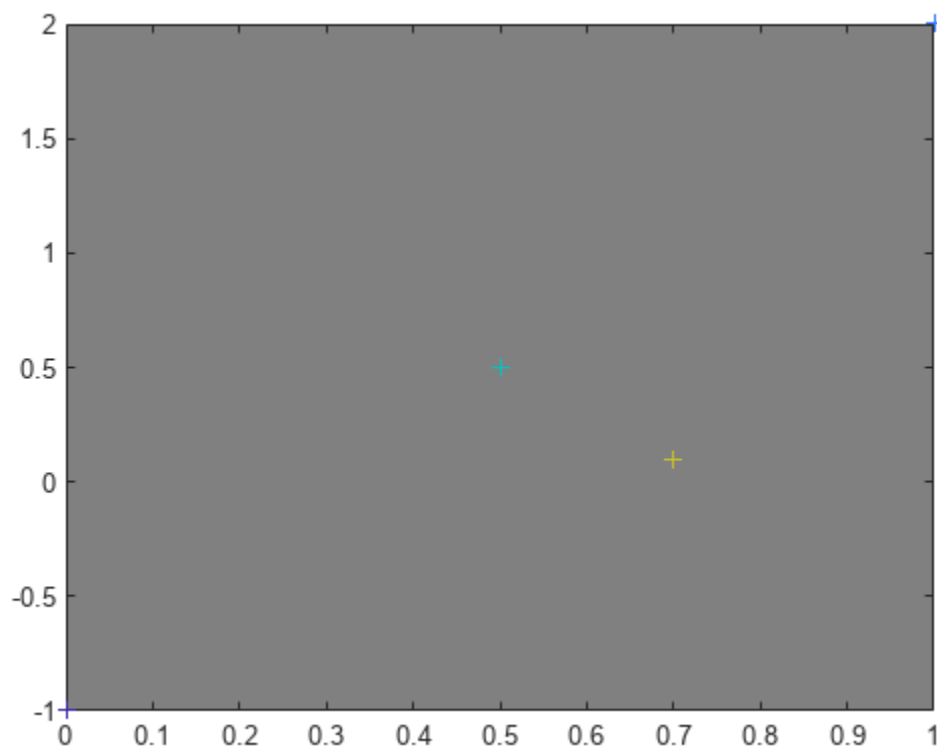
`plotvec(X)` only takes a matrix  $X$  and plots each  $i$ th vector in  $X$  with marker '+' using the index  $i$  as the color coordinate.

### 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)
```



## Version History

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.

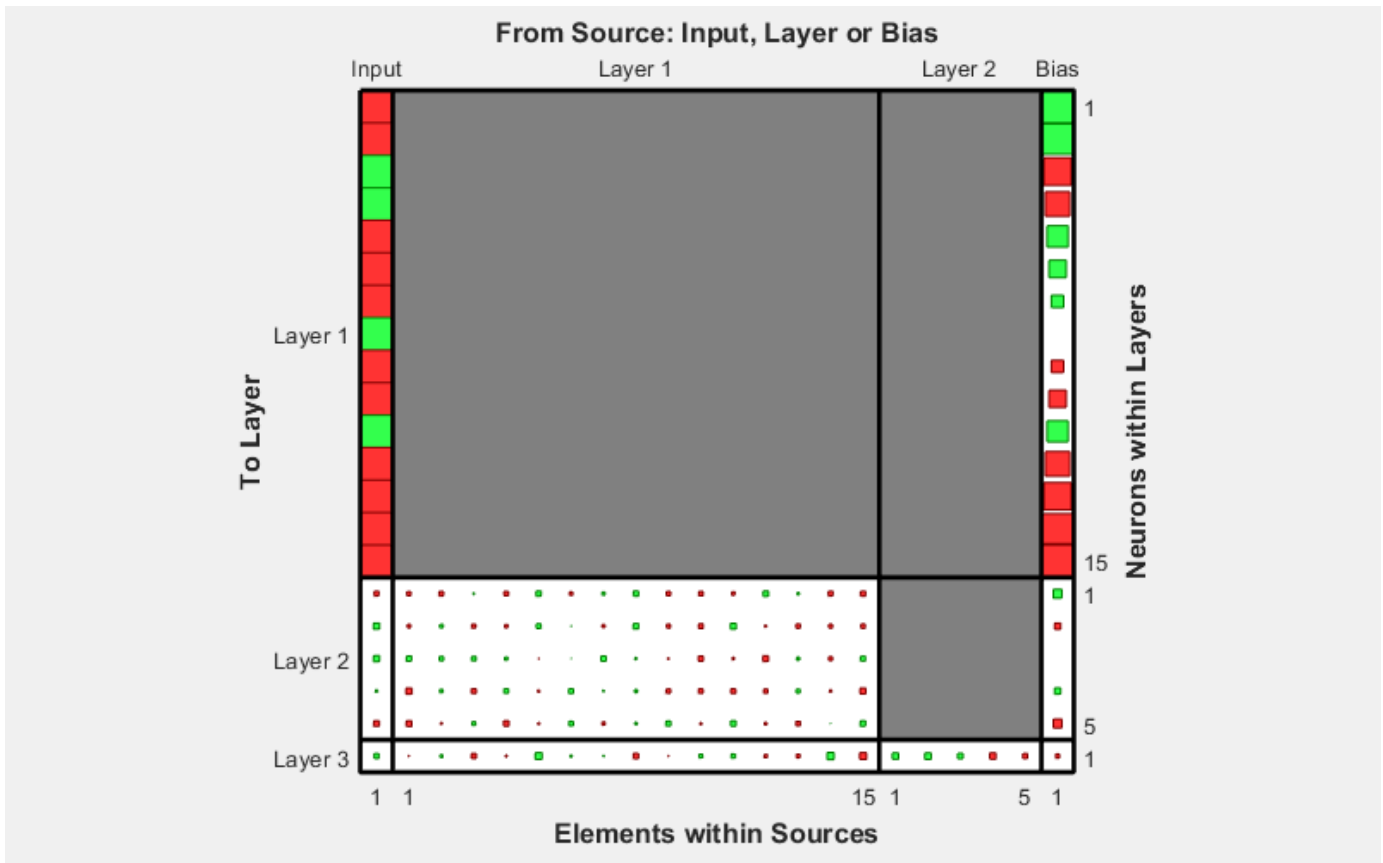
### Examples

#### Plot Weights and Biases

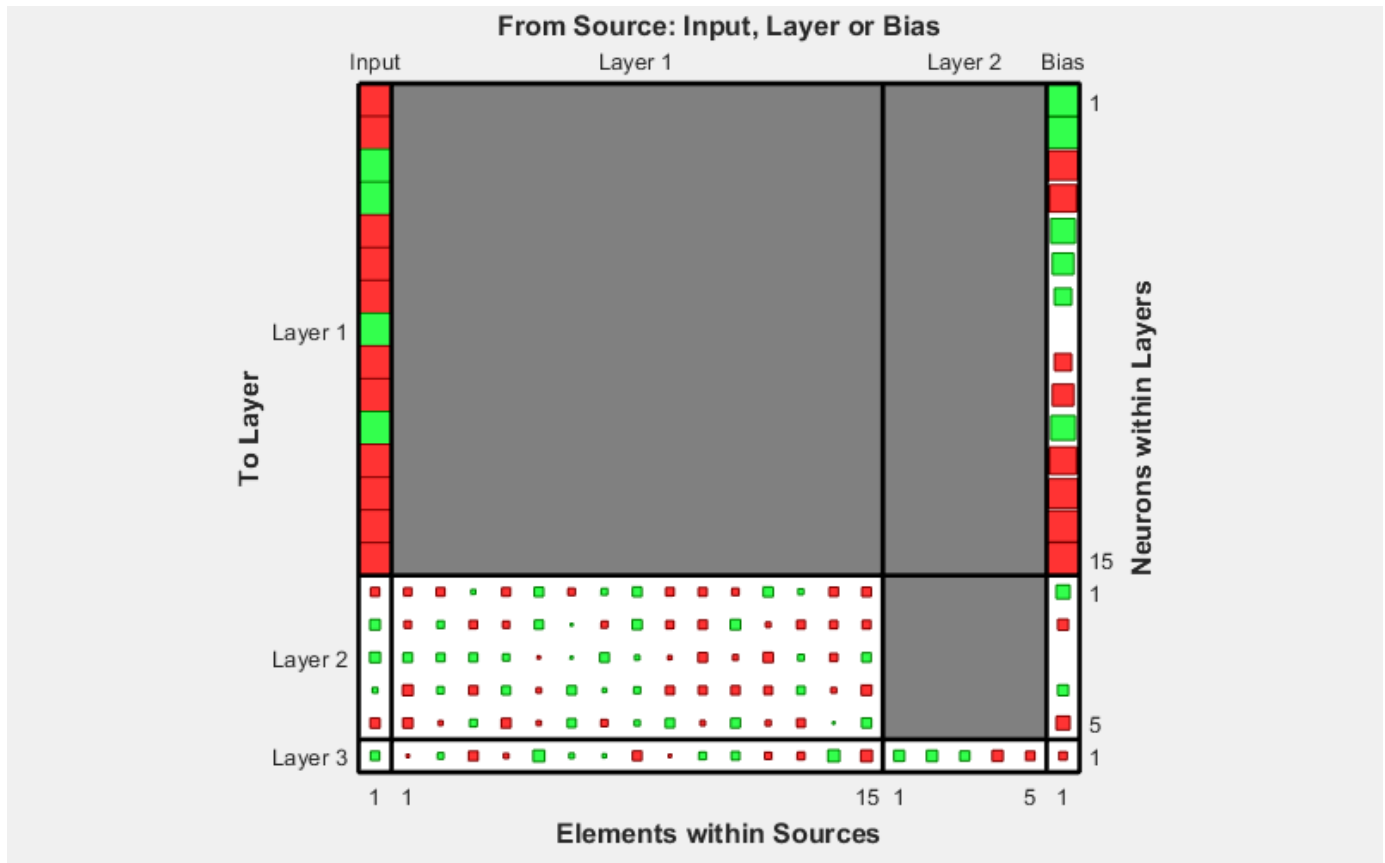
Here a cascade-forward network is configured for particular data and its weights and biases are plotted in several ways.

```
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
```

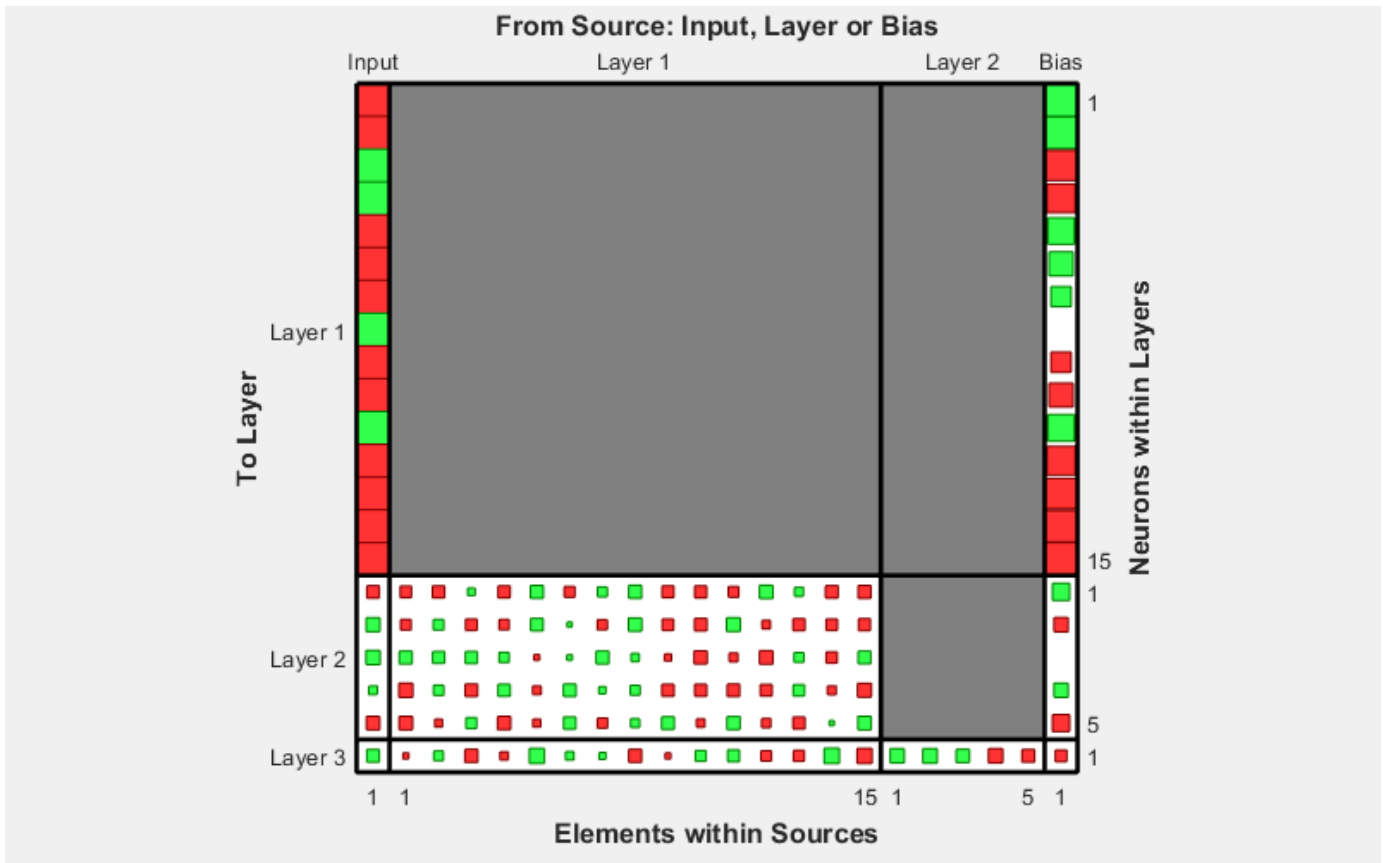




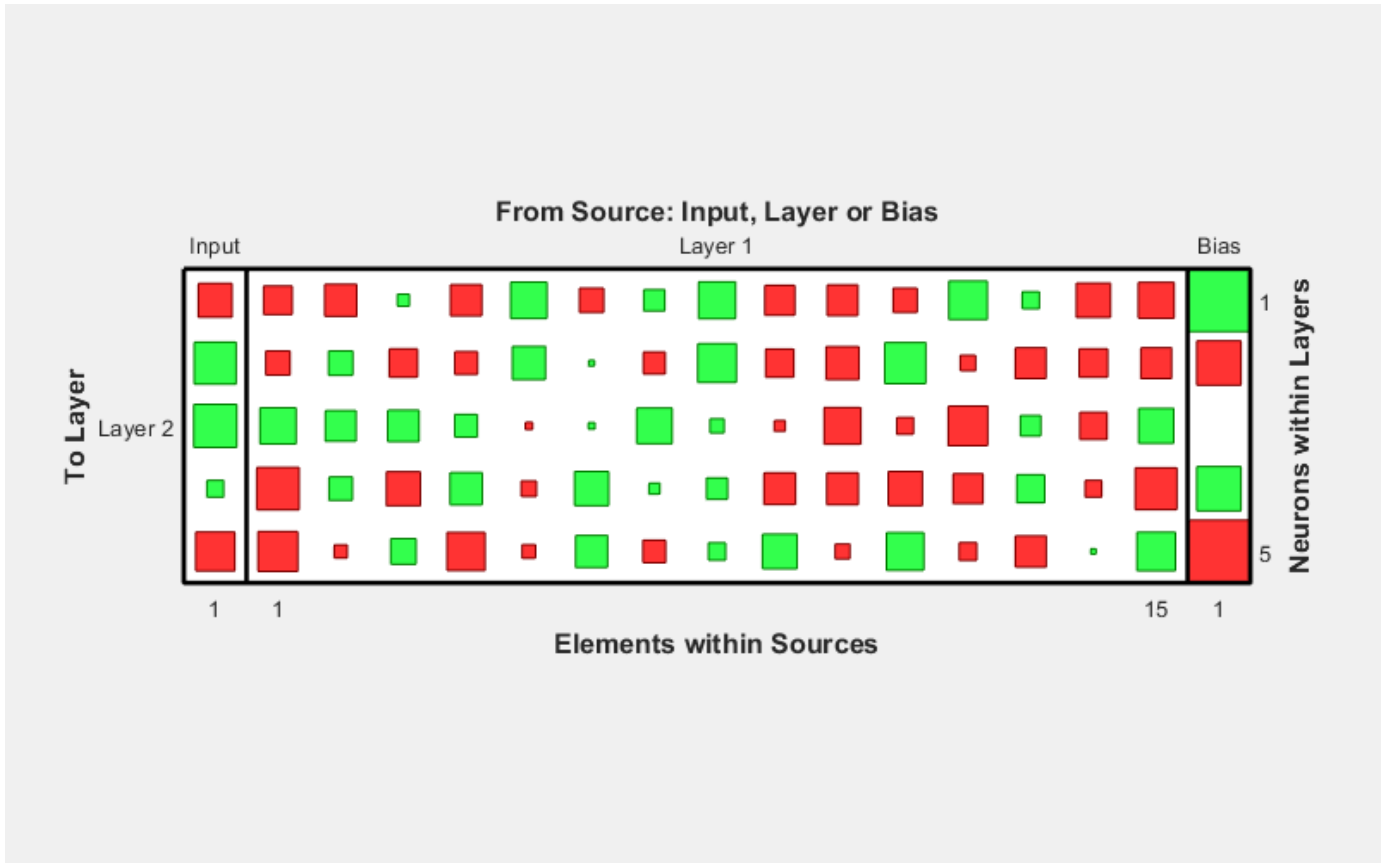
plotwb(net, 'root', 3)



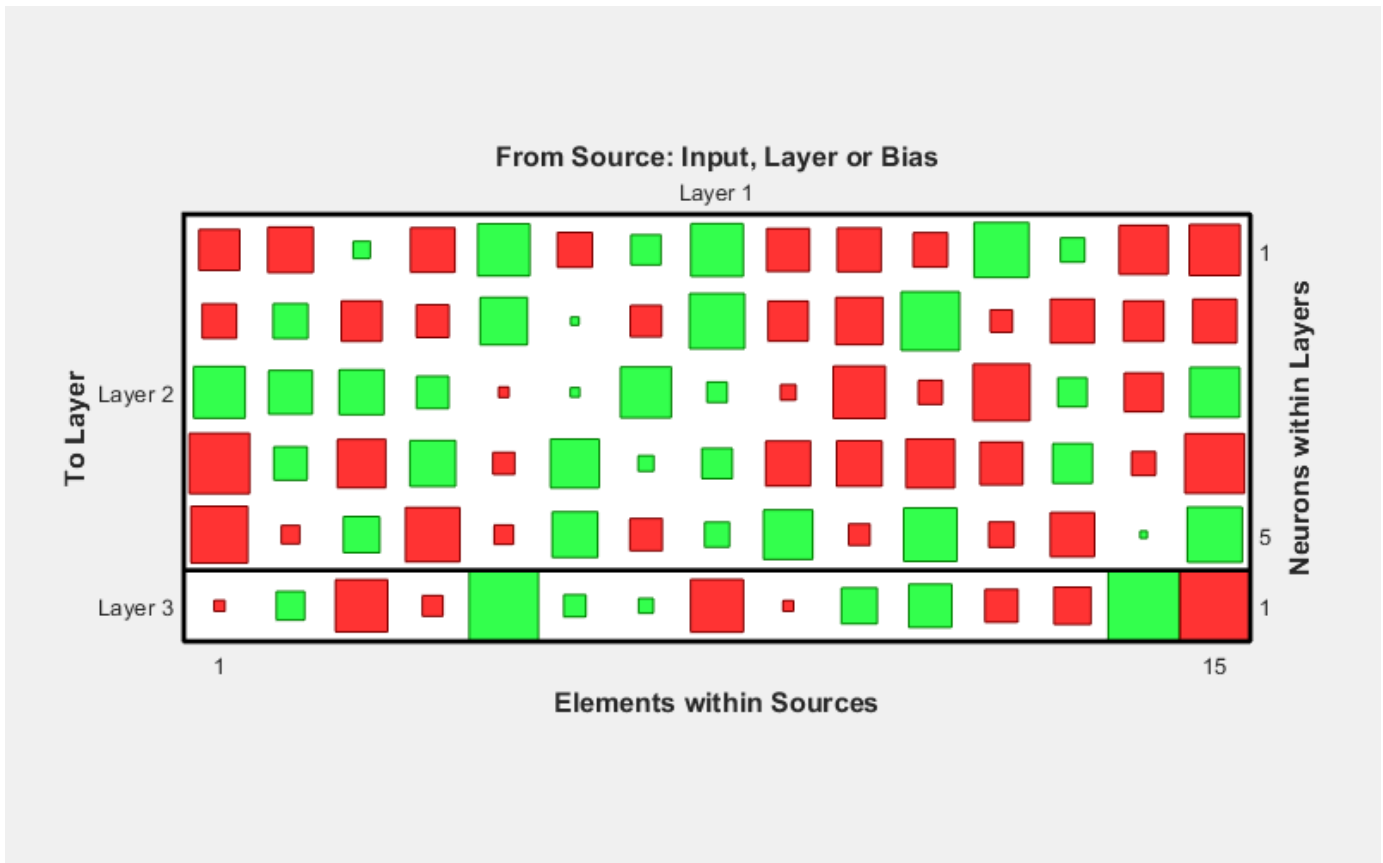
plotwb(net, 'root', 4)



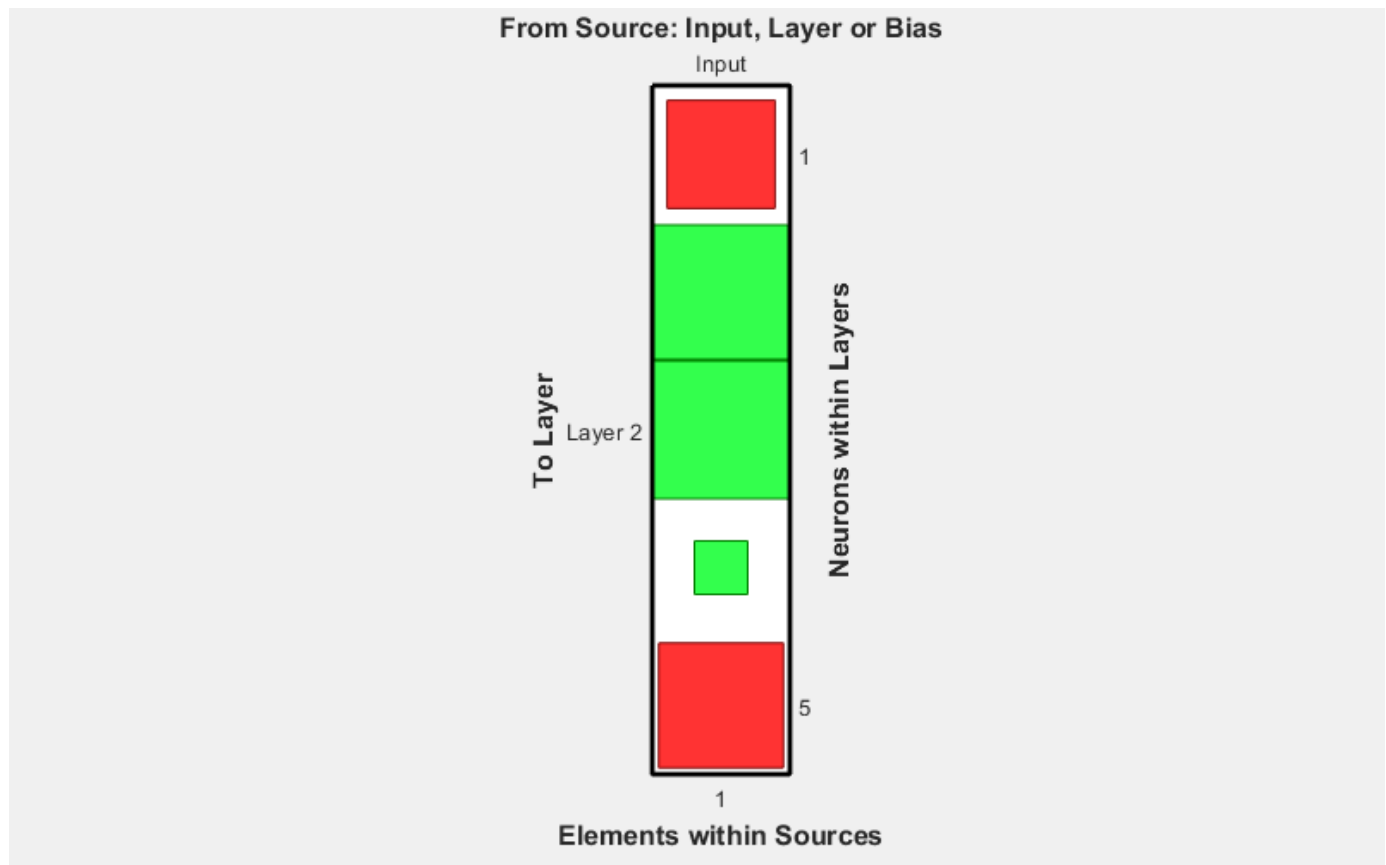
```
plotwb(net, 'toLayers', 2)
```



```
plotwb(net, 'fromLayers', 1)
```



```
plotwb(net, 'toLayers', 2, 'fromInputs', 1)
```



## Version History

Introduced in R2010b

### See Also

plotsomplanes

## 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)
```

### Version History

Introduced before R2006a

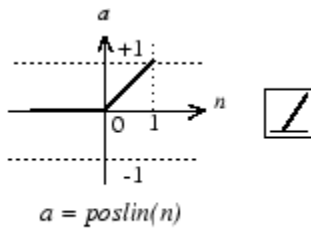
### See Also

normc | normr

## 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.

$$\text{poslin}(n) = \begin{cases} n, & \text{if } n \geq 0 \\ 0, & \text{if } n < 0 \end{cases}$$

## Version History

Introduced before R2006a

## See Also

`sim` | `purelin` | `satlin` | `satlins`

## 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

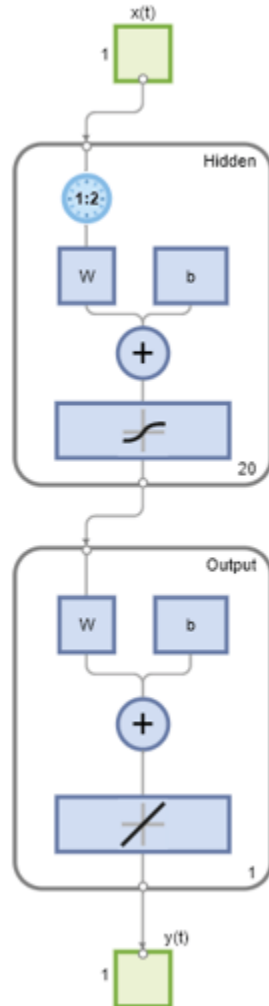
#### Prepare Data for Open- and Closed-Loop Networks

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);
```

Network Diagram

**Training Results**  
 Training finished: Reached minimum gradient ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	444	1000
Elapsed Time	-	00:00:03	-
Performance	1.36	3.88e-11	0
Gradient	4.66	9.97e-08	1e-07
Mu	0.001	1e-08	1e+10
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

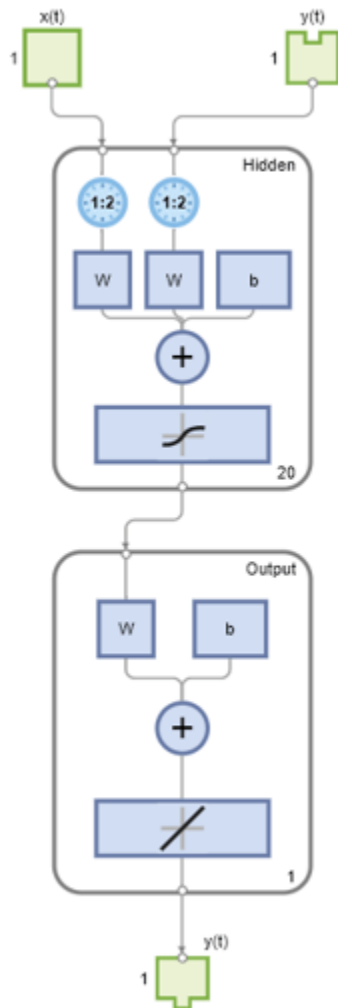
Regression

Time-Series Response

Error

Input-Error Cross-correlation

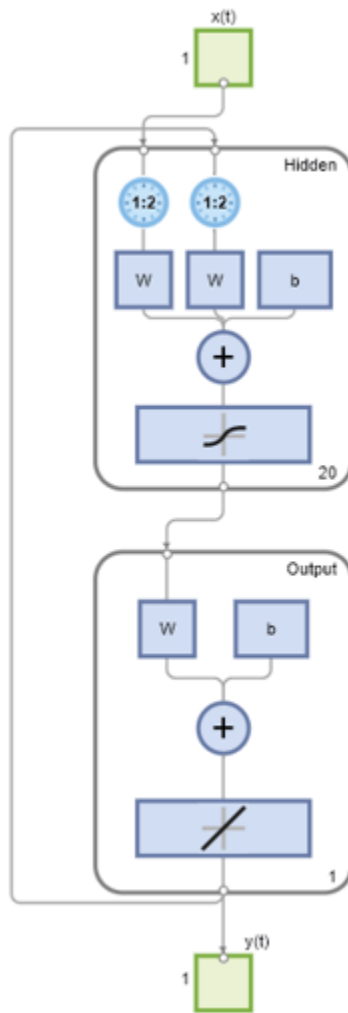
view(net)



```
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.

**Version History****Introduced in R2010b**

**See Also**

`adddelay` | `closeloop` | `narnet` | `narxnet` | `openloop` | `removedelay` | `timedelaynet`



## 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

$$y = 2*(x-\text{minx})/(\text{maxx}-\text{minx}) - 1;$$

Values in rows with all the same value are set to 0.

## Version History

Introduced in R2006a

## See Also

`fixunknowns` | `mapminmax` | `mapstd`

## 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

<code>net</code>	The same network with zero-sized subobjects removed
<code>pi</code>	Indices of pruned inputs
<code>pl</code>	Indices of pruned layers
<code>po</code>	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);
```

## **Version History**

**Introduced in R2010b**

### **See Also**

prunedata | gensim

## 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,

<code>net</code>	Pruned neural network
<code>pi</code>	Indices of pruned inputs
<code>pl</code>	Indices of pruned layers
<code>po</code>	Indices of pruned outputs
<code>X</code>	Input data
<code>Xi</code>	Initial input delay states
<code>Ai</code>	Initial layer delay states
<code>T</code>	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);
```

## **Version History**

**Introduced in R2010b**

### **See Also**

prune | gensim

## purelin

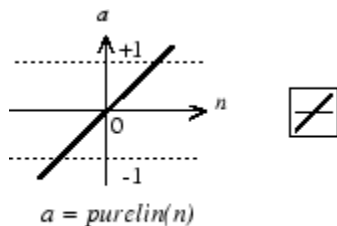
Linear transfer function

### Syntax

```
A = purelin(N)
info = purelin('code')
```

### Description

$A = \text{purelin}(N)$  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('fpname')` 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

$a = \text{purelin}(n) = n$

## Version History

Introduced before R2006a

## See Also

sim | satlin | satlins

## 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)
```

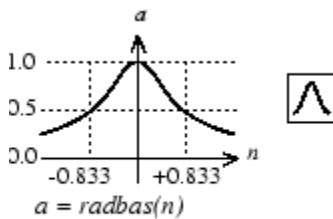
### Version History

Introduced before R2006a

# radbas

Radial basis transfer function

## Graph and Symbol



Radial Basis Function

## Syntax

$A = \text{radbas}(N, FP)$

## Description

radbas is a neural transfer function. Transfer functions calculate a layer's output from its net input.

$A = \text{radbas}(N, FP)$  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

$a = \text{radbas}(n) = \exp(-n^2)$

## Version History

Introduced before R2006a

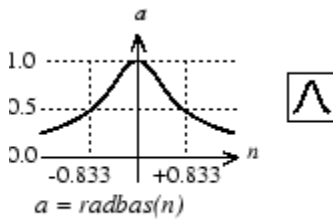
**See Also**

sim | radbasn | tribas

# radbasn

Normalized radial basis transfer function

## Graph and Symbol



Radial Basis Function

## Syntax

$A = \text{radbasn}(N, FP)$

## 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.

$A = \text{radbasn}(N, FP)$  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

$$a = \text{radbasn}(n) = \frac{\exp(-n^2)}{\sum(\exp(-n^2))}$$

## **Version History**

**Introduced in R2010b**

### **See Also**

sim | radbas | tribas

## 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

## Version History

Introduced before R2006a

### See Also

randnr

## 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
```

## Version History

Introduced before R2006a

### See Also

randnc



## 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`.

## Version History

Introduced before R2006a

**See Also**

`randsmall` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

## 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.

## Version History

Introduced in R2010b

**See Also**

`rands` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

# 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
------------	-------------------------------

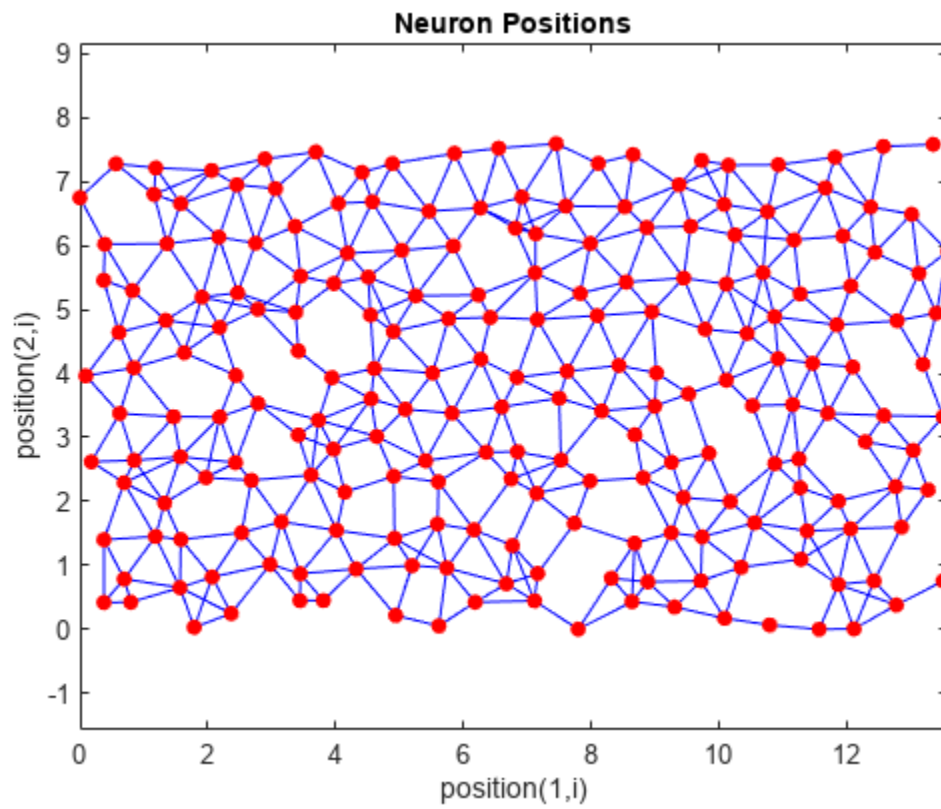
and returns an N-by-S matrix of N coordinate vectors, where N is the number of dimensions and S is the product of dimensions.

## 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)
```



## Version History

Introduced before R2006a

## See Also

[gridtop](#) | [hextop](#) | [tritop](#)

# 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	657	1000
Elapsed Time	-	00:00:06	-
Performance	250	4.37e-08	0
Gradient	476	6.01e-06	1e-07
Mu	0.001	1e-09	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance      Training State  
Error Histogram      Regression

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)
```

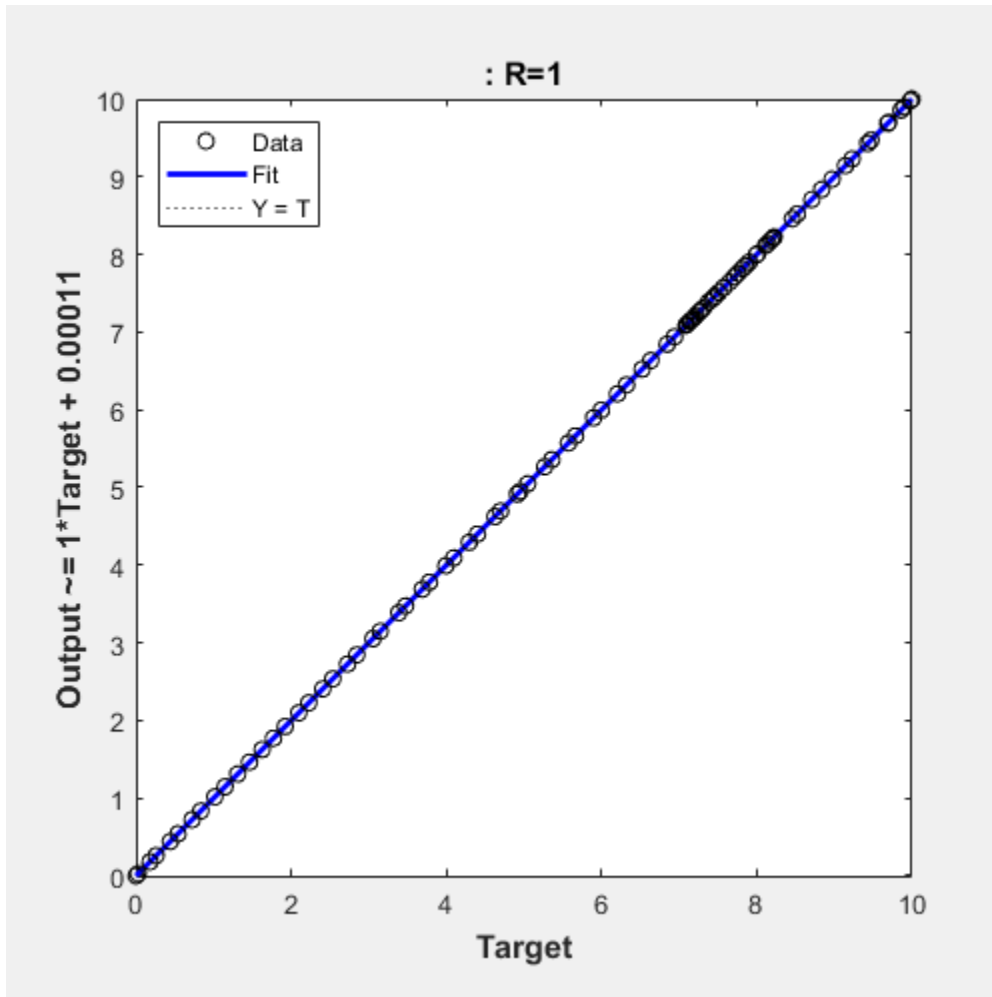
```
r = 1.0000
```



```
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.

## Version History

### **Introduced in R2010b**

### **R2020b: 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`

## 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
```

## Version History

Introduced in R2006a

### See Also

[fixunknowns](#) | [mapminmax](#) | [mapstd](#) | [processpca](#)

## 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 shows how to create, train, and simulate 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.

#### Time Delay

```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

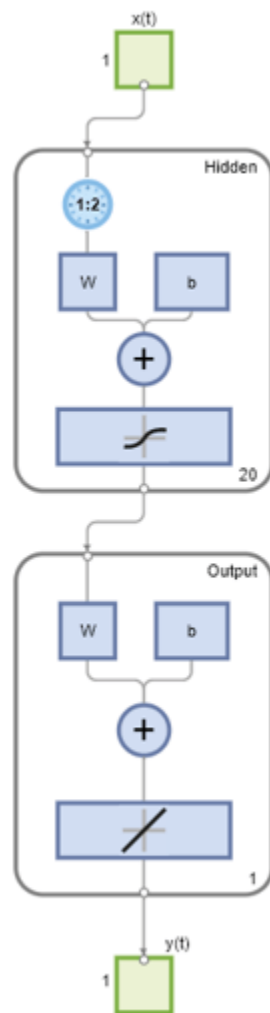
Performance Training State

Error Histogram Regression

Time-Series Response Error

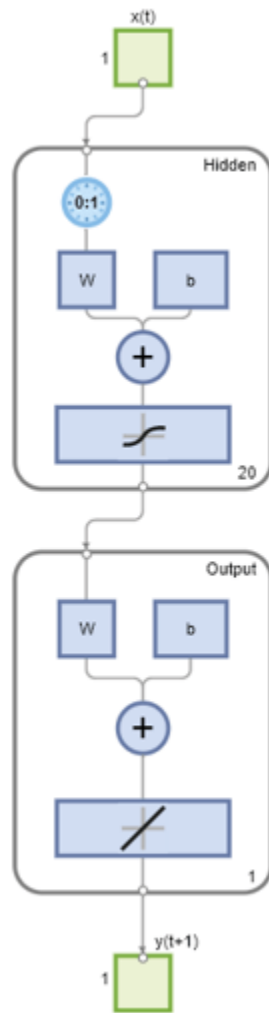
Input-Error Cross-correlation

```
y1 = net1(Xs,Xi);
view(net1)
```



### Remove Delay

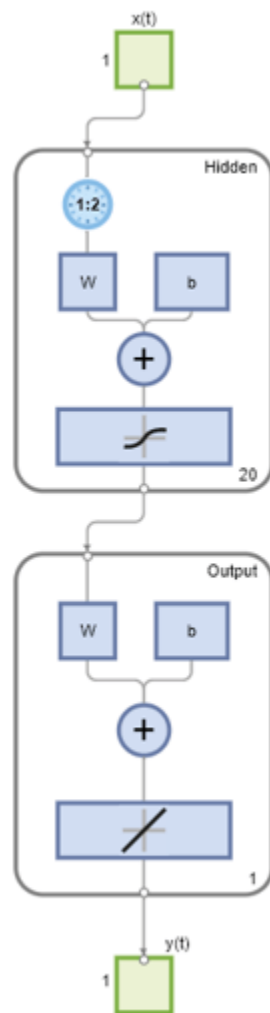
```
net2 = removedelay(net1);  
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);  
y2 = net2(Xs,Xi);  
view(net2)
```



### Add Delay

```
net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
```





## Version History

Introduced in R2010b

### See Also

[adddelay](#) | [closeloop](#) | [openloop](#)

## 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,

<code>X</code>	N-by-Q matrix
<code>ind</code>	Vector of row indices to remove (default is <code>[]</code> )

and returns

<code>Y</code>	M-by-Q matrix, where <code>M == N - length(ind)</code>
<code>PS</code>	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.

## Version History

Introduced in R2006a

## See Also

[fixunknowns](#) | [mapminmax](#) | [mapstd](#) | [processpca](#)

## 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}
```

## Version History

Introduced before R2006a

### See Also

`init` | `sim` | `adapt` | `train`

## 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  $[0, 1]$ .

---

**Tip** `roc` does not support categorical targets. To compute ROC metrics for categorical targets, use `rocmetrics`.

---

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.

**Version History**

Introduced in R2008a

**See Also**

plotroc | confusion

## 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,

<code>net</code>	Neural network
<code>t</code>	Matrix or cell array of target vectors
<code>y</code>	Matrix or cell array of output vectors
<code>ew</code>	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.

### **Version History**

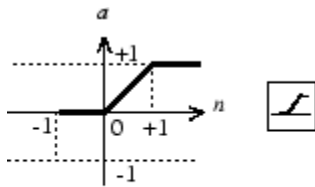
**Introduced in R2010b**



# satlin

Saturating linear transfer function

## Graph and Symbol



$$a = \text{satlin}(n)$$

Satlin Transfer Function

## Syntax

$A = \text{satlin}(N, FP)$

## Description

`satlin` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

$A = \text{satlin}(N, FP)$  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
```

### Version History

Introduced before R2006a

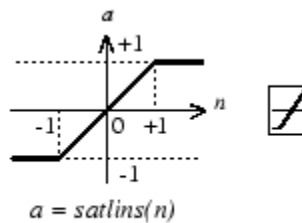
### See Also

[sim](#) | [poslin](#) | [satlins](#) | [purelin](#)

# satlins

Symmetric saturating linear transfer function

## Graph and Symbol



Satlins Transfer Function

## Syntax

$A = \text{satlins}(N, FP)$

## Description

`satlins` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

$A = \text{satlins}(N, FP)$  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
```

### Version History

Introduced before R2006a

### See Also

sim | satlin | poslin | purelin

# 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.

$Z = \text{scalprod}(W, P)$  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  $Z = w * P$ .

$\text{dim} = \text{scalprod}('size', S, R, FP)$  takes the layer dimension S, input dimension R, and function parameters, and returns the weight size [1-by-1].

$\text{dw} = \text{scalprod}('dw', W, P, Z, FP)$  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.

## Version History

Introduced in R2006a

**See Also**

dotprod | sim | dist | negdist | normprod

# 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);
```

Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value	
Epoch	0	200	200	▲
Elapsed Time	-	00:00:04	-	▼

**Training Algorithms**  
Data Division: Batch Weight/Bias Rule trainbu  
Performance: Mean Squared Error mse  
Calculations: MATLAB

**Training Plots**

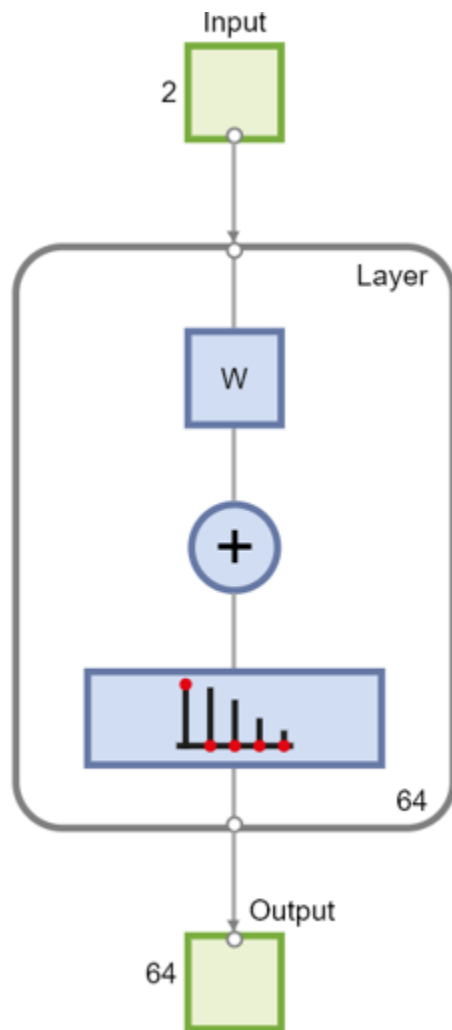
SOM SOM Neighbor Connections

SOM Neighbor Distances SOM Input Planes

SOM Sample Hits SOM Weight Positions

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.

**Version History**

**Introduced in R2010b**

**See Also**

lvqnet | competlayer | nctool

## separatwb

Separate biases and weight values from weight/bias vector

### Syntax

```
[b,IW,LW] = separatwb(net,wb)
```

### Description

[b,IW,LW] = separatwb(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] = separatwb(net,wb)
```

## Version History

**Introduced in R2010b**

### See Also

getwb | formwb | setwb

## 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)
```

## Version History

Introduced before R2006a

### See Also

`con2seq` | `concur`

# 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)
```

## Version History

**Introduced in R2010b**

## See Also

nndata | numelements | getelements | catelements | setsamples | setsignals | settimesteps

## setsamples

Set neural network data samples

### Syntax

```
setsamples(x,i,v)
```

### Description

`setsamples(x,i,v)` takes these arguments,

<code>x</code>	Neural network matrix or cell array data
<code>i</code>	Indices
<code>v</code>	Neural network data to store into <code>x</code>

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)
```

## Version History

**Introduced in R2010b**

### See Also

[nndata](#) | [numsamples](#) | [getsamples](#) | [catsamples](#) | [setelements](#) | [setsignals](#) | [settimesteps](#)

# setsignals

Set neural network data signals

## Syntax

```
setsignals(x,i,v)
```

## Description

`setsignals(x,i,v)` takes these arguments,

<code>x</code>	Neural network matrix or cell array data
<code>i</code>	Indices
<code>v</code>	Neural network data to store into <code>x</code>

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)
```

## Version History

Introduced in R2010b

## See Also

`nndata` | `numsignals` | `getsignals` | `catsignals` | `setelements` | `setsamples` | `settimesteps`

## 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,

<code>sysName</code>	The name of the Simulink system containing the neural network block
<code>netName</code>	The name of the Simulink neural network block
<code>net</code>	The original neural network
<code>xi</code>	Initial input delay states
<code>ai</code>	Initial layer delay states
<code>Q</code>	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);
```

## **Version History**

**Introduced in R2010b**

### **See Also**

[gensim](#) | [getsiminit](#) | [nndata2sim](#) | [sim2nndata](#)

## settimesteps

Set neural network data timesteps

### Syntax

```
settimesteps(x,i,v)
```

### Description

`settimesteps(x,i,v)` takes these arguments,

<code>x</code>	Neural network matrix or cell array data
<code>i</code>	Indices
<code>v</code>	Neural network data to store into <code>x</code>

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)
```

## Version History

Introduced in R2010b

### See Also

`nndata` | `numtimesteps` | `gettimesteps` | `cattimesteps` | `setelements` | `setsamples` | `setsignals`

## 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

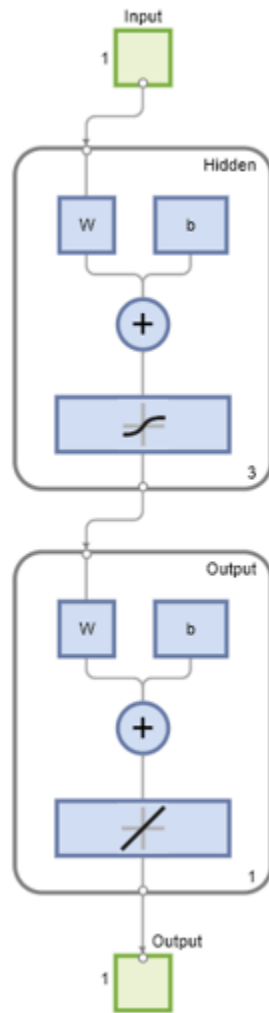
### Examples

#### 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

## **Version History**

**Introduced in R2010b**

### **See Also**

getwb | formwb | separatwb

## 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

<code>net</code>	Network
<code>X</code>	Network inputs
<code>Xi</code>	Initial input delay conditions (default = zeros)
<code>Ai</code>	Initial layer delay conditions (default = zeros)
<code>T</code>	Network targets (default = zeros)

and returns

<code>Y</code>	Network outputs
<code>Xf</code>	Final input delay conditions
<code>Af</code>	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 $R_i$ -by- $Q$ matrix.
$X_i$	Ni-by-ID cell array	Each element $X_i\{i, k\}$ is an $R_i$ -by- $Q$ matrix.
$A_i$	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by- $Q$ matrix.
T	No-by-TS cell array	Each element $X\{i, ts\}$ is a $U_i$ -by- $Q$ matrix.
Y	No-by-TS cell array	Each element $Y\{i, ts\}$ is a $U_i$ -by- $Q$ matrix.
$X_f$	Ni-by-ID cell array	Each element $X_f\{i, k\}$ is an $R_i$ -by- $Q$ matrix.
$A_f$	Nl-by-LD cell array	Each element $A_f\{i, k\}$ is an $S_i$ -by- $Q$ matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
No	=	<code>net.numOutputs</code>
ID	=	<code>net.numInputDelays</code>
LD	=	<code>net.numLayerDelays</code>
TS	=	Number of time steps
Q	=	Batch size
$R_i$	=	<code>net.inputs{i}.size</code>
$S_i$	=	<code>net.layers{i}.size</code>
$U_i$	=	<code>net.outputs{i}.size</code>

The columns of  $X_i$ ,  $A_i$ ,  $X_f$ , and  $A_f$  are ordered from oldest delay condition to most recent:

$X_i\{i, k\}$	=	Input $i$ at time $ts = k - ID$
$X_f\{i, k\}$	=	Input $i$ at time $ts = TS + k - ID$
$A_i\{i, k\}$	=	Layer output $i$ at time $ts = k - LD$
$A_f\{i, k\}$	=	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 $R_i$ )-by- $Q$ matrix
$X_i$	(sum of $R_i$ )-by-( $ID*Q$ ) matrix
$A_i$	(sum of $S_i$ )-by-( $LD*Q$ ) matrix
T	(sum of $U_i$ )-by- $Q$ matrix
Y	(sum of $U_i$ )-by- $Q$ matrix
$X_f$	(sum of $R_i$ )-by-( $ID*Q$ ) matrix
$A_f$	(sum of $S_i$ )-by-( $LD*Q$ ) matrix

$[Y, X_f, A_f] = \text{sim}(\text{net}, \{Q \text{ TS}\}, X_i, A_i)$  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);
```



Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	15	1000
Elapsed Time	-	00:00:03	-
Performance	337	5.3	0
Gradient	1.72e+03	62.2	1e-07
Mu	0.001	0.001	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

Regression

`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
```

## Version History

Introduced before R2006a

### See Also

`init` | `adapt` | `train` | `revert`

## 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)
```

## Version History

**Introduced in R2010b**

### See Also

`nndata` | `nndata2sim`

# softmax

Soft max transfer function

## Syntax

```
A = softmax(N)
info = softmax(code)
```

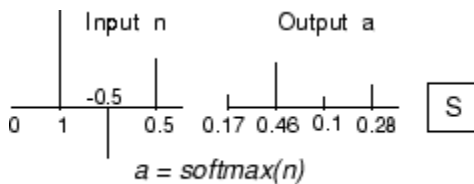
## Description

---

**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' | 'fpname' | '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.
- 'fpname' 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

$$a = \text{softmax}(n) = \exp(n) / \text{sum}(\exp(n))$$

## Version History

Introduced before R2006a

## See Also

sim | compet

## 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 $D_{ij}$ -by- $Q$ matrix.
Tl	Nl-by-TS cell array	Each element $P\{i, ts\}$ is a $V_i$ -by- $Q$ matrix.
V	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by- $Q$ matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
LD	=	<code>net.numLayerDelays</code>
Ri	=	<code>net.inputs{i}.size</code>
Si	=	<code>net.layers{i}.size</code>
Vi	=	<code>net.targets{i}.size</code>
Dij	=	<code>Ri * length(net.inputWeights{i,j}.delays)</code>

## 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.

## Version History

Introduced before R2006a

## 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

## 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 $D_{ij}$ -by-Q matrix.
Tl	Nl-by-TS cell array	Each element $P\{i, ts\}$ is a $V_i$ -by-Q matrix.
Ai	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by-Q matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
LD	=	<code>net.numLayerDelays</code>
Ri	=	<code>net.inputs{i}.size</code>
Si	=	<code>net.layers{i}.size</code>
Vi	=	<code>net.targets{i}.size</code>
Dij	=	<code>Ri * length(net.inputWeights{i,j}.delays)</code>

## 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.

### Version History

Introduced before R2006a

### References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

### See Also

`srchbac` | `srchcha` | `srchgol` | `srchhyb`

## 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 $D_{ij}$ -by- $Q$ matrix.
Tl	Nl-by-TS cell array	Each element $P\{i, ts\}$ is a $V_i$ -by- $Q$ matrix.
Ai	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by- $Q$ matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
LD	=	<code>net.numLayerDelays</code>
Ri	=	<code>net.inputs{i}.size</code>
Si	=	<code>net.layers{i}.size</code>
Vi	=	<code>net.targets{i}.size</code>
Dij	=	<code>Ri * length(net.inputWeights{i,j}.delays)</code>

## 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).

## Version History

Introduced before R2006a

## 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

## 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 $D_{ij}$ -by-Q matrix.
Tl	Nl-by-TS cell array	Each element $P\{i, ts\}$ is a $V_i$ -by-Q matrix.
Ai	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by-Q matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
LD	=	<code>net.numLayerDelays</code>
Ri	=	<code>net.inputs{i}.size</code>
Si	=	<code>net.layers{i}.size</code>
Vi	=	<code>net.targets{i}.size</code>
Dij	=	<code>Ri * length(net.inputWeights{i,j}.delays)</code>

## 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).

### Version History

Introduced before R2006a

### References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

### See Also

`srchbac` | `srchbre` | `srchcha` | `srchhyb`

## 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 $D_{ij}$ -by- $Q$ matrix.
Tl	Nl-by-TS cell array	Each element $P\{i, ts\}$ is a $V_i$ -by- $Q$ matrix.
Ai	Nl-by-LD cell array	Each element $A_i\{i, k\}$ is an $S_i$ -by- $Q$ matrix.

where

Ni	=	<code>net.numInputs</code>
Nl	=	<code>net.numLayers</code>
LD	=	<code>net.numLayerDelays</code>
Ri	=	<code>net.inputs{i}.size</code>
Si	=	<code>net.layers{i}.size</code>
Vi	=	<code>net.targets{i}.size</code>
Dij	=	<code>Ri * length(net.inputWeights{i,j}.delays)</code>

## 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).

## Version History

Introduced before R2006a

## References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York Springer-Verlag, 1985

## See Also

srchbac | srchbre | srchcha | srchgol

## **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.

### Version History

Introduced before R2006a

### See Also

`mse` | `mae`



# 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 (elmnet) 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,

<code>net</code>	Neural network
<code>X</code>	Inputs, an $R \times Q$ matrix (or $N \times TS$ cell array of $R \times Q$ matrices)
<code>T</code>	Targets, an $S \times Q$ matrix (or $M \times TS$ cell array of $S \times Q$ matrices)
<code>Xi</code>	Initial input delay states (optional)
<code>Ai</code>	Initial layer delay states (optional)
<code>EW</code>	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,  $R_i$  and  $S_i$  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)
```

## **Version History**

**Introduced in R2010b**

### **See Also**

`bttderiv` | `defaultderiv` | `fpderiv` | `num2deriv`

# 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]})
```

## Version History

Introduced in R2010b

## See Also

[meanabs](#) | [meansqr](#) | [sumsqr](#)

## 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.

## **Version History**

Introduced before R2006a

### **See Also**

meanabs | meansqr | sumabs

## tansig

Hyperbolic tangent sigmoid transfer function

### Syntax

`A = tansig(N)`

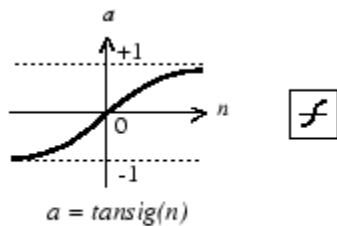
### Description

---

**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  $[-\infty \infty]$  to the interval  $[-1 \ 1]$  with an "S-shaped" function.

## Algorithms

$$a = \text{tansig}(N) = 2/(1+\exp(-2*N))-1$$

This is mathematically equivalent to  $\tanh(N)$ .

## Version History

Introduced before R2006a

## 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

## 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])
```

## Version History

**Introduced in R2010b**

### See Also

[nndata](#) | [extendts](#) | [preparets](#)



# 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

This example shows how to train a time delay network.

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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	10	1000
Elapsed Time	-	00:00:03	-
Performance	0.183	0.014	0
Gradient	0.548	0.00121	1e-07
Mu	0.001	0.0001	1e+10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

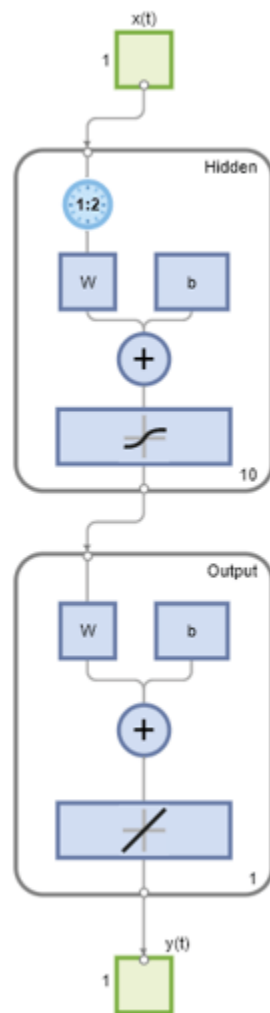
Regression

Time-Series Response

Error

Input-Error Cross-correlation

view(net)

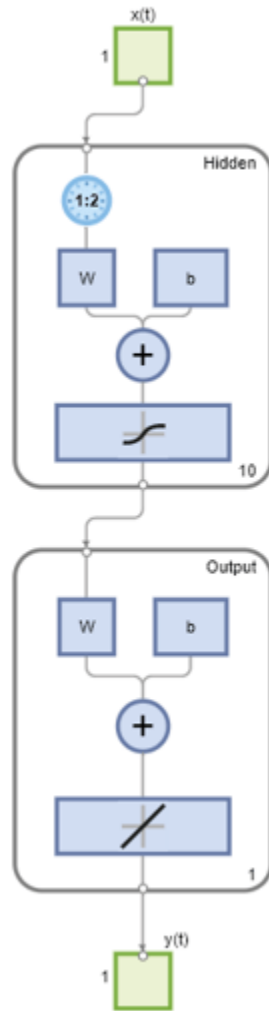


Calculate the network performance.

```
[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-Ribière 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

## Version History

Introduced in R2010b

### See Also

preparets | removedelay | distdelaynet | narnet | narxnet

## 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 = rand(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)
```

## Version History

Introduced in R2010b

### See Also

nndata | fromnndata | nndata2sim | sim2nndata

# 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
- $N_i = \text{net.numInputs}$
- 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  $R_i$ )-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  $R_i$ -by-Q matrix, where  $R_i = \text{net.inputs}\{i\}.size$ .

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
- $N_o = \text{net.numOutputs}$

- $TS$  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  $U_i$ )-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  $U_i$ -by- $Q$  matrix, where  $U_i = \text{net.outputs}\{i\}.\text{size}$ .

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.

---

### **$X_i$ – Initial input delay conditions**

zeros (default) | cell array | matrix

Initial input delay conditions, specified as an  $N_i$ -by- $ID$  cell array or an  $R$ -by- $(ID*Q)$  matrix, where

- $ID = \text{net.numInputDelays}$
- $N_i = \text{net.numInputs}$
- $R$  is the input size
- $Q$  is the batch size

For cell array input, the columns of  $X_i$  are ordered from the oldest delay condition to the most recent:  $X_i\{i, k\}$  is the input  $i$  at time  $ts = k - ID$ .

$X_i$  is also optional and need only be used for networks that have input or layer delays.

### **$A_i$ – Initial layer delay conditions**

zeros (default) | cell array | matrix

Initial layer delay conditions, specified as a  $N_l$ -by- $LD$  cell array or a (sum of  $S_i$ )-by- $(LD*Q)$  matrix, where

- $N_l = \text{net.numLayers}$
- $LD = \text{net.numLayerDelays}$
- $S_i = \text{net.layers}\{i\}.\text{size}$
- $Q$  is the batch size

For cell array input, the columns of  $A_i$  are ordered from the oldest delay condition to the most recent:  $A_i\{i, k\}$  is the layer output  $i$  at time  $ts = k - LD$ .

### **EW — Error weights**

cell array

Error weights, specified as a  $N_o$ -by- $TS$  cell array or a (sum of  $U_i$ )-by- $Q$  matrix, where

- $N_o = \text{net.numOutputs}$
- $TS$  is the number of time steps
- $U_i = \text{net.outputs}\{i\}.\text{size}$
- $Q$  is the batch size

For cell array input, each element  $EW\{i, ts\}$  is a  $U_i$ -by- $Q$  matrix, where

- $U_i = \text{net.outputs}\{i\}.\text{size}$
- $Q$  is the batch size

The error weights  $EW$  can also have a size of 1 in place of all or any of  $N_o$ ,  $TS$ ,  $U_i$  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  $U_i$ )-by-1, then each output element has a different importance, with all samples treated with the same importance.

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as  $Name_1=Value_1, \dots, Name_N=Value_N$ , 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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.

## Version History

Introduced before R2006a

### See Also

`init` | `revert` | `sim` | `adapt`

# 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.min_grad</code>	1e-6	Minimum performance gradient
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

### Version History

Introduced before R2006a

### See Also

`linearlayer` | `train`



# 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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	30	1000
Elapsed Time	-	00:00:04	-
Performance	337	15	0
Gradient	1.72e+03	25.7	1e-06
Validation Checks	0	6	6
Step Size	100	0.1	1e-06
Resets	0	0	4

**Training Algorithms**  
Data Division: Random dividerand  
Training: BFGS Quasi-Newton trainbfg  
Performance: Mean Squared Error mse  
Calculations: MEX

**Training Plots**

Performance Training State  
Error Histogram Regression

`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 `feedforwardnet` 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:

$$dX = -H \setminus gX;$$

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`.
- Validation performance (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## References

[1] Gill, Murray, & Wright, *Practical Optimization*, 1981

**See Also**

`cascadeforwardnet` | `feedforwardnet` | `traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp`  
| `traingcf` | `traingcb` | `trainscg` | `traingcp` | `trainoss`

# 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,

$$\begin{aligned} jj &= jX * jX \\ je &= jX * E \\ dX &= -(jj+I*mu) \setminus je \end{aligned}$$

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`.

## Version History

Introduced before R2006a

## 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` | `traingcf` | `traingcb` | `traingcg` | `traingcp` | `traingbf`

## 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

### Version History

Introduced in R2010b

### See Also

`train` | `trainb`

# 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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.

### Version History

Introduced before R2006a

### See Also

`competlayer` | `train`

# 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.time</code>	inf	Maximum time to train in seconds
<code>net.trainParam.min_grad</code>	1e-10	Minimum performance gradient
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.searchFcn</code>	'srchcha'	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.sca1_tol</code>	20	Divide into <code>delta</code> to determine tolerance for linear search.
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines sufficient reduction in <code>perf</code>
<code>net.trainParam.beta</code>	0.1	Scale factor that determines sufficiently large step size
<code>net.trainParam.delta</code>	0.01	Initial step size in interval location step
<code>net.trainParam.gama</code>	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> )
<code>net.trainParam.low_lim</code>	0.1	Lower limit on change in step size
<code>net.trainParam.up_lim</code>	0.5	Upper limit on change in step size
<code>net.trainParam.maxstep</code>	100	Maximum step length
<code>net.trainParam.minstep</code>	1.0e-6	Minimum step length

<code>net.trainParam.bmax</code>	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);
```



Network Diagram

### Training Results

Training finished: Met validation criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	43	1000
Elapsed Time	-	00:00:03	-
Performance	337	15.3	0
Gradient	1.72e+03	10.9	1e-10
Validation Checks	0	6	6
Step Size	100	0.0472	1e-06

### Training Algorithms

Data Division: Random dividerand

Training: Conjugate Gradient with Beale-Powell Restarts trai...

Performance: Mean Squared Error mse

Calculations: MEX

### Training Plots

Performance

Training State

Error Histogram

Regression

`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:

$$|\mathbf{g}_{k-1}^T \mathbf{g}_k| \geq 0.2 \|\mathbf{g}_k\|^2$$

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-Ribière (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:

$$X = X + a \cdot dX;$$

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

$$dX = -gX + dX_{old} \cdot Z;$$

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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## 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

## 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.time</code>	inf	Maximum time to train in seconds
<code>net.trainParam.min_grad</code>	1e-10	Minimum performance gradient
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.searchFcn</code>	'srchcha'	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.sca1_tol</code>	20	Divide into <code>delta</code> to determine tolerance for linear search.
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines sufficient reduction in <code>perf</code>
<code>net.trainParam.beta</code>	0.1	Scale factor that determines sufficiently large step size
<code>net.trainParam.delta</code>	0.01	Initial step size in interval location step
<code>net.trainParam.gama</code>	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> )
<code>net.trainParam.low_lim</code>	0.1	Lower limit on change in step size
<code>net.trainParam.up_lim</code>	0.5	Upper limit on change in step size
<code>net.trainParam.maxstep</code>	100	Maximum step length
<code>net.trainParam.minstep</code>	1.0e-6	Minimum step length

<code>net.trainParam.bmax</code>	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);
```

Network Diagram

**Training Results**  
Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	51	1000
Elapsed Time	-	00:00:03	-
Performance	337	15.2	0
Gradient	1.72e+03	5.89	1e-10
Validation Checks	0	6	6
Step Size	100	0.187	1e-06

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Conjugate Gradient Backpropagation with Fletcher-Ree  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance Training State

Error Histogram Regression

```
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  $\beta_k$  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

$$Z = \text{normnew\_sqr}/\text{norm\_sqr};$$

where  $\text{norm\_sqr}$  is the norm square of the previous gradient and  $\text{normnew\_sqr}$  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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## 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`



# traincgp

Conjugate gradient backpropagation with Polak-Ribière 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-Ribière 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.time</code>	inf	Maximum time to train in seconds
<code>net.trainParam.min_grad</code>	1e-10	Minimum performance gradient
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.searchFcn</code>	'srchcha'	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.sca1_tol</code>	20	Divide into <code>delta</code> to determine tolerance for linear search.
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines sufficient reduction in <code>perf</code>
<code>net.trainParam.beta</code>	0.1	Scale factor that determines sufficiently large step size
<code>net.trainParam.delta</code>	0.01	Initial step size in interval location step
<code>net.trainParam.gama</code>	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> )
<code>net.trainParam.low_lim</code>	0.1	Lower limit on change in step size
<code>net.trainParam.up_lim</code>	0.5	Upper limit on change in step size
<code>net.trainParam.maxstep</code>	100	Maximum step length
<code>net.trainParam.minstep</code>	1.0e-6	Minimum step length

<code>net.trainParam.bmax</code>	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);
```

Network Diagram

### Training Results

Training finished: Met validation criterion ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	26	1000
Elapsed Time	-	00:00:03	-
Performance	337	17.4	0
Gradient	1.72e+03	6.59	1e-10
Validation Checks	0	6	6
Step Size	100	0.0298	1e-06

### Training Algorithms

Data Division: Random dividerand

Training: Conjugate Gradient with Polak-Ribiere Restarts trai...

Performance: Mean Squared Error mse

Calculations: MEX

### Training Plots

Performance

Training State

Error Histogram

Regression

`y = net(x);`

## More About

### Conjugate Gradient Backpropagation with Polak-Ribière Updates

Another version of the conjugate gradient algorithm was proposed by Polak and Ribière. 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-Ribière update, the constant  $\beta_k$  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-Ribière 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-Ribière (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-Ribière 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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## 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`

## 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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

### See Also

`traingdm` | `traingda` | `traingdx` | `trainlm`

## 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 `nnd12v1` [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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

**See Also**

traingd | traingdm | traingdx | trainlm

## 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.lr</code>	0.01	Learning rate
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.mc</code>	0.9	Momentum constant
<code>net.trainParam.min_grad</code>	1e-5	Minimum performance gradient
<code>net.trainParam.show</code>	25	Epochs between showing progress
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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,

$$dX = mc*dX_{prev} + lr*(1-mc)*dperf/dX$$

where  $dX_{prev}$  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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

**See Also**

`traingd` | `traingda` | `traingdx` | `trainlm`



# 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,

$$dX = mc*dX_{prev} + lr*mc*dperf/dX$$

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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

### See Also

`traingd` | `traingda` | `traingdm` | `trainlm`

## 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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	15	1000

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: Levenberg-Marquardt trainlm  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

Performance

Training State

Error Histogram

Regression

`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

$$\mathbf{H} = \mathbf{J}^T \mathbf{J} \quad (2-1)$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T \mathbf{e} \quad (2-2)$$

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  $\mu$  is zero, this is just Newton's method, using the approximate Hessian matrix. When  $\mu$  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,  $\mu$  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  $\mathbf{jX}$  of performance `perf` with respect to the weight and bias variables  $\mathbf{X}$ . Each variable is adjusted according to Levenberg-Marquardt,

$$\begin{aligned} \mathbf{jj} &= \mathbf{jX} * \mathbf{jX} \\ \mathbf{je} &= \mathbf{jX} * \mathbf{E} \\ \mathbf{dX} &= -(\mathbf{jj} + \mathbf{I} * \mu) \setminus \mathbf{je} \end{aligned}$$

where  $\mathbf{E}$  is all errors and  $\mathbf{I}$  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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## **Version History**

**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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.min_grad</code>	1e-10	Minimum performance gradient
<code>net.trainParam.searchFcn</code>	'srchbac'	Name of line search routine to use
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	inf	Maximum time to train in seconds

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.sca1_tol</code>	20	Divide into <code>delta</code> to determine tolerance for linear search.
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines sufficient reduction in <code>perf</code>
<code>net.trainParam.beta</code>	0.1	Scale factor that determines sufficiently large step size
<code>net.trainParam.delta</code>	0.01	Initial step size in interval location step
<code>net.trainParam.gama</code>	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> )
<code>net.trainParam.low_lim</code>	0.1	Lower limit on change in step size
<code>net.trainParam.up_lim</code>	0.5	Upper limit on change in step size
<code>net.trainParam.maxstep</code>	100	Maximum step length
<code>net.trainParam.minstep</code>	1.0e-6	Minimum step length

<code>net.trainParam.bmax</code>	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);
```

Network Diagram

**Training Results**  
 Training finished: Met validation criterion ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	12	1000
Elapsed Time	-	00:00:03	-
Performance	337	23.5	0
Gradient	1.72e+03	35.3	1e-10
Validation Checks	0	6	6

**Training Algorithms**  
 Data Division: Random dividerand  
 Training: One Step Secant trainoss  
 Performance: Mean Squared Error mse  
 Calculations: MEX

**Training Plots**

`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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## 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` | `traingcf` | `traingcb` | `trainscg` | `traingcp` | `trainbfg`

## 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.max_fail</code>	6	Maximum validation failures
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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.

### Version History

Introduced before R2006a

### See Also

`train`



# 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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## 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` | `traingcp` | `traingcf` | `traingcb` | `traingcg` | `trainoss` | `trainbfg`

# 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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.

### **Version History**

**Introduced in R2010b**

### **See Also**

`train` | `trainr`

## 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:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.time</code>	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.

## **Version History**

**Introduced before R2006a**

## **See Also**

`train` | `trainb` | `trainc` | `trainr`



# 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 (validation error) has increased more than `max_fail` times since the last time it decreased (when using validation).

## Version History

Introduced before R2006a

## References

[1] Moller. *Neural Networks*, Vol. 6, 1993, pp. 525-533

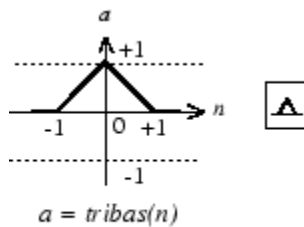
## See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainbfg` | `traincgp` | `trainoss`

## 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,

<code>N</code>	S-by-Q matrix of net input (column) vectors
<code>FP</code>	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
```

## Version History

Introduced before R2006a

## See Also

sim | radbas

## 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:

<code>dimensions</code>	Row vector of dimension sizes
-------------------------	-------------------------------

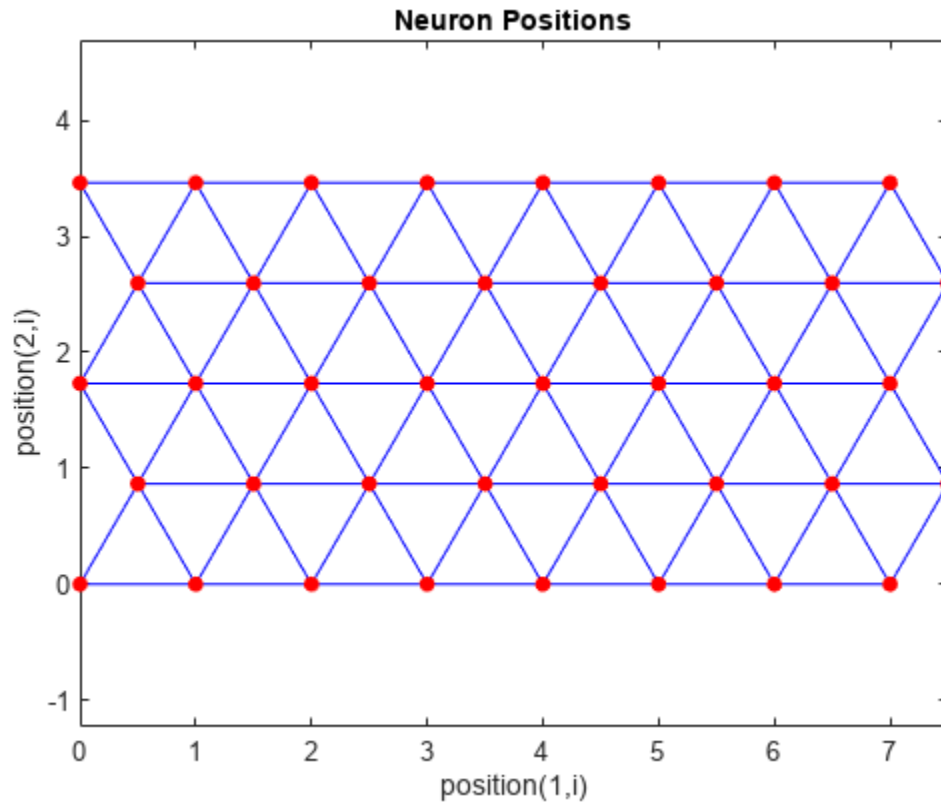
and returns an N-by-S matrix of N coordinate vectors, where N is the number of dimensions and S is the product of dimensions.

### 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)
```



## Version History

Introduced in R2010b

## See Also

[gridtop](#) | [hextop](#) | [randtop](#)

## 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.

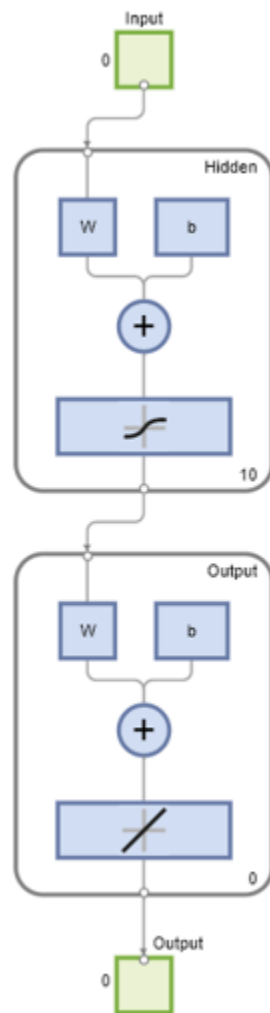
### Examples

#### Configure and Unconfigure Network

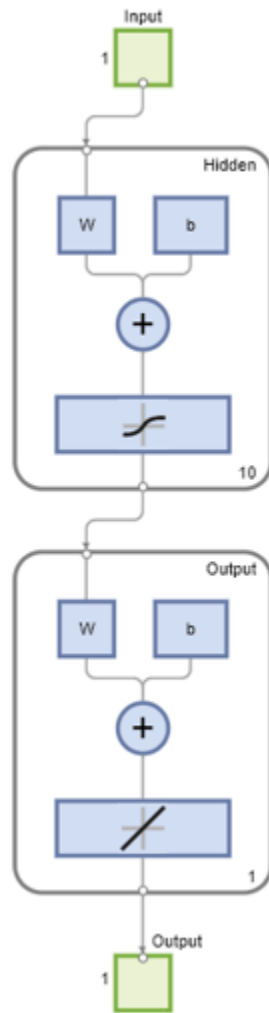
This example shows how to configure a network for a simple fitting problem, and then unconfigure the network.

```
[x,t] = simplefit_dataset;
net = fitnet(10);
view(net)
```

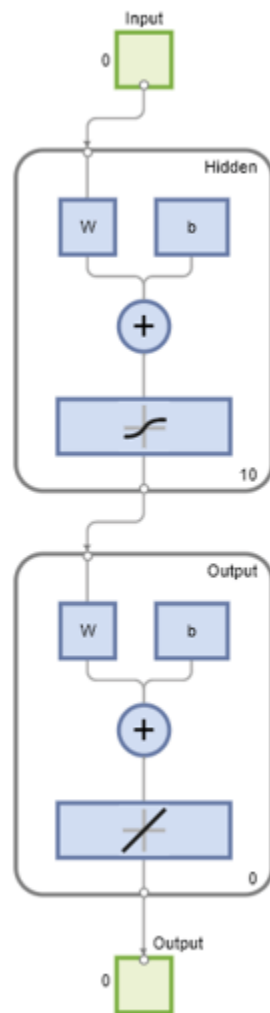




```
net = configure(net,x,t);  
view(net)
```



```
net = unconfigure(net);  
view(net)
```



## Version History

Introduced in R2010b

### See Also

`configure` | `isconfigured`

## vec2ind

Convert vectors to indices

### 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
```

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

## 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.

## Version History

Introduced before R2006a

## See Also

[ind2vec](#) | [sub2ind](#) | [ind2sub](#)

# view

View shallow neural network

## Syntax

```
view(net)
```

## Description

`view(net)` opens a window that shows your shallow neural network (specified by `net`) as a graphical diagram.

---

**Tip** To visualize deep learning networks, see **Deep Network Designer**.

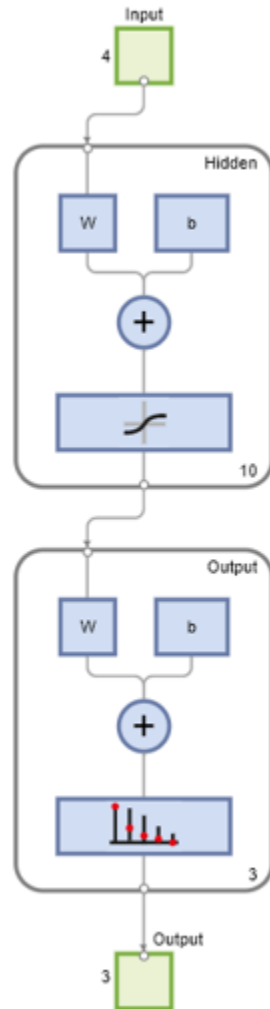
---

## Examples

### 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)
```



## Input Arguments

**net** — Neural network  
network object

Neural network, specified as a network object. To create a network object, use for example, `feedforwardnet` or `narxnet`.

## Version History

Introduced in R2008a

## See Also

`feedforwardnet` | `narxnet`

**Topics**

“Body Fat Estimation”



# 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.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

Two-layer feedforward network with sigmoid hidden neurons and linear output neurons, suitable for regression tasks.

**Model Summary**

Train a neural network to map predictors to continuous responses.

**Data**  
 Predictors: bodyfatInputs - [13x252 double]  
 Responses: bodyfatTargets - [1x252 double]  
 bodyfatInputs: double array of 252 observations with 13 features.  
 bodyfatTargets: double array of 252 observations with 1 features.

**Algorithm**  
 Data division: Random  
 Training algorithm: Levenberg-Marquardt  
 Performance: Mean squared error

**Training Results**  
 Training start time: 12-Jul-2021 10:42:52  
 Layer size: 10

	Observations	MSE	R
Training	176	13.9529	0.8906
Validation	38	19.0443	0.8725
Test	38	21.0854	0.8527

## 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	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Fitting</b> app uses the <code>trainlm</code> function.</p>
Bayesian regularization	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Fitting</b> app uses the <code>trainbr</code> function.</p>
Scaled conjugate gradient backpropagation	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Fitting</b> app uses the <code>trainscg</code> function.</p>

## 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.

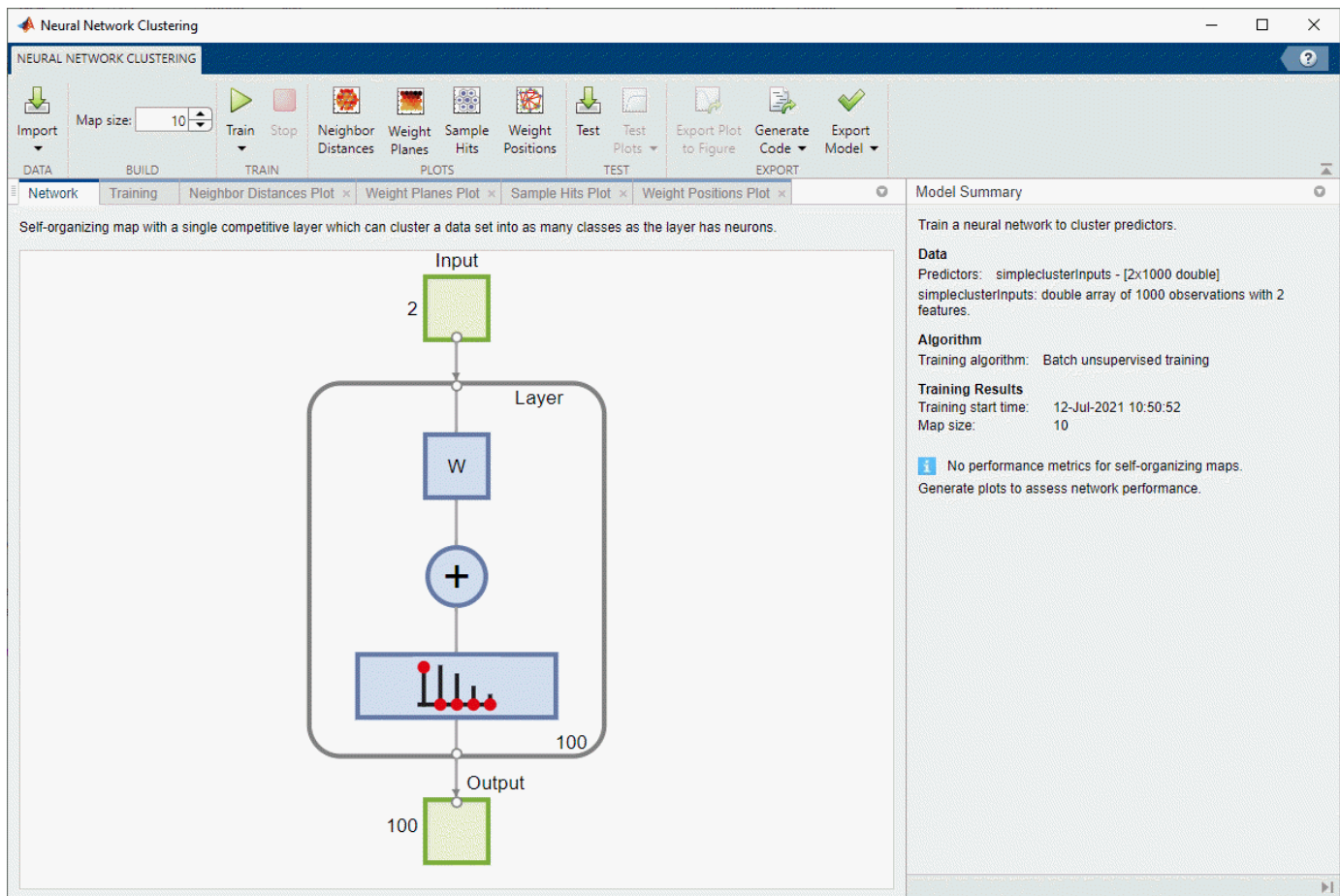
Using this app, you can:

- 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.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---



## 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 <b>Neural Net Clustering</b> app uses the <code>trainbu</code> 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.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

Two-layer feedforward network with sigmoid hidden neurons and softmax output neurons, suitable for classification tasks.

**Data**  
 Predictors: irisInputs - [4x150 double]  
 Responses: irisTargets - [3x150 double]  
 irisInputs: double array of 150 observations with 4 features.  
 irisTargets: double array of 150 observations with 3 classes.

**Algorithm**  
 Data division: Random  
 Training algorithm: Scaled conjugate gradient  
 Performance: Cross-entropy error

**Training Results**  
 Training start time: 12-Jul-2021 10:50:19  
 Layer size: 10

	Observations	Cross-entropy	Error
Training	104	0.0108	0.0096
Validation	23	0.0347	0.0435
Test	23	0.0231	0.0435

## 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

- “Pattern Recognition 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 <b>Neural Net Pattern Recognition</b> app uses the <code>trainscg</code> function.

## See Also

### Apps

[Neural Net Fitting](#) | [Neural Net Clustering](#) | [Neural Net Time Series](#)

### Functions

`patternnet` | `trainscg`

### Topics

“Pattern Recognition 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.

---

**Note** To interactively build, visualize, and train deep learning neural networks, use the **Deep Network Designer** app. For more information, see “Get Started with Deep Network Designer”.

---

Neural Network Time Series

Training data: 70 %  
 Validation data: 15  
 Test data: 15

Layer size: 10  
 Time delay: 2

Train Stop Training State Performance Error Histogram Response Error Autocorrelation Input-Error Correlation Test Test Plots Export Plot to Figure Generate Code Export Model

Network Training Error Histogram Plot x Response Plot x Error Autocorrelation Plot x Input-Error Correlation Plot x

Nonlinear autoregressive neural network with external input (NARX).

Model Summary

Train a neural network to predict series  $y(t)$  from past values of  $y(t)$  and past values of another series  $x(t)$ .

**Data**  
 Predictors: simplenarxInputs - [1x100 cell]  
 Responses: simplenarxTargets - [1x100 cell]  
 simplenarxInputs: cell array of 100 time steps with 1 features.  
 simplenarxTargets: cell array of 100 time steps with 1 features.

**Algorithm**  
 Data division: Random  
 Training algorithm: Levenberg-Marquardt  
 Performance: Mean squared error

**Training Results**  
 Training start time: 12-Jul-2021 10:51:40  
 Layer size: 10  
 Time delay: 2

	Observations	MSE	R
Training	68	2.5325e-11	1.0000
Validation	15	9.2037e-11	1.0000
Test	15	1.3443e-10	1.0000

## 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	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Time Series</b> app uses the <code>trainlm</code> function.</p>
Bayesian regularization	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Time Series</b> app uses the <code>trainbr</code> function.</p>
Scaled conjugate gradient backpropagation	<p>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.</p> <p>To implement this algorithm, the <b>Neural Net Time Series</b> app uses the <code>trainscg</code> function.</p>

## 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: [12x20 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 = 1x1 cell array
    {12x20 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 = 1x2 cell array
      {12x20 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).

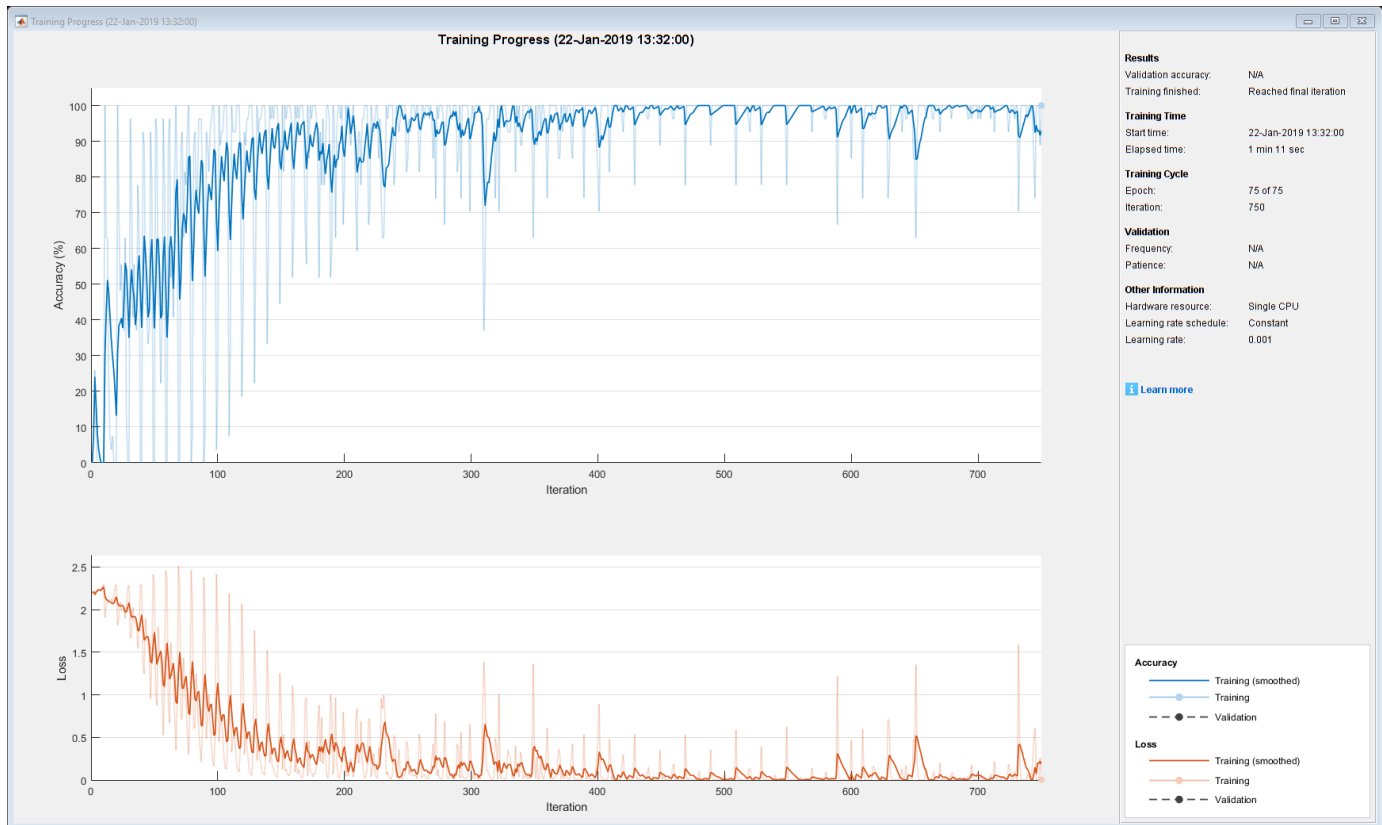
```
miniBatchSize = 27;

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
```

## Version History

### Introduced in R2018a

### **R2019a: `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”](#)

## read

**Class:** `matlab.io.datastore.MiniBatchable`

**Package:** `matlab.io.datastore`

(Not recommended) Read data from custom 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.

## Version History

### Introduced in R2018a

#### **R2019a: 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 `all` 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”

# 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 specified by index from datastore

## 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.

## Version History

### Introduced in R2018a

#### **R2019a: `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 `ttl` 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”

## readByIndex

**Class:** matlab.io.datastore.BackgroundDispatchable

**Package:** matlab.io.datastore

(Not recommended) Return observations specified by index from datastore

---

**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.

## Version History

### Introduced in R2018a

#### **R2019a: 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 all 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”

# 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 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.

## Version History

### Introduced in R2018a

#### **R2019a: `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 `all` 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”

## partitionByIndex

**Class:** matlab.io.datastore.PartitionableByIndex

**Package:** matlab.io.datastore

(Not recommended) Partition 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.

## Version History

### Introduced in R2018a

### R2019a: `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”

# 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);
```

Network Diagram

**Training Results**  
Training finished: Reached minimum gradient ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	721	1000
Elapsed Time	-	00:00:14	-
Performance	1.47	0.049	0
Gradient	0.578	9.65e-07	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Training Only `dividetrain`  
 Training: Scaled Conjugate Gradient `trainscg`  
 Performance: Mean Squared Error with L2 and Sparsity Regularizers  
 Calculations: MEX

**Training Plots**  
Performance

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');
```

Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	400	400
Elapsed Time	-	00:00:09	-
Performance	0.725	0.0193	0
Gradient	0.496	0.00144	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
Data Division: Training Only `dividetrain`  
Training: Scaled Conjugate Gradient `trainscg`  
Performance: Mean Squared Error with L2 and Sparsity Regularizers  
Calculations: MEX

**Training Plots**  
Performance

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);
```

Network Diagram

**Training Results**  
Training finished: Reached minimum gradient ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	315	1000
Elapsed Time	-	00:00:04	-
Performance	195	131	0
Gradient	4.73e+03	8.97e-07	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Training Only `dividetrain`  
 Training: Scaled Conjugate Gradient `trainscg`  
 Performance: Mean Squared Error with L2 and Sparsity Regularizers  
 Calculations: MEX

**Training Plots**  
 Performance

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);
```

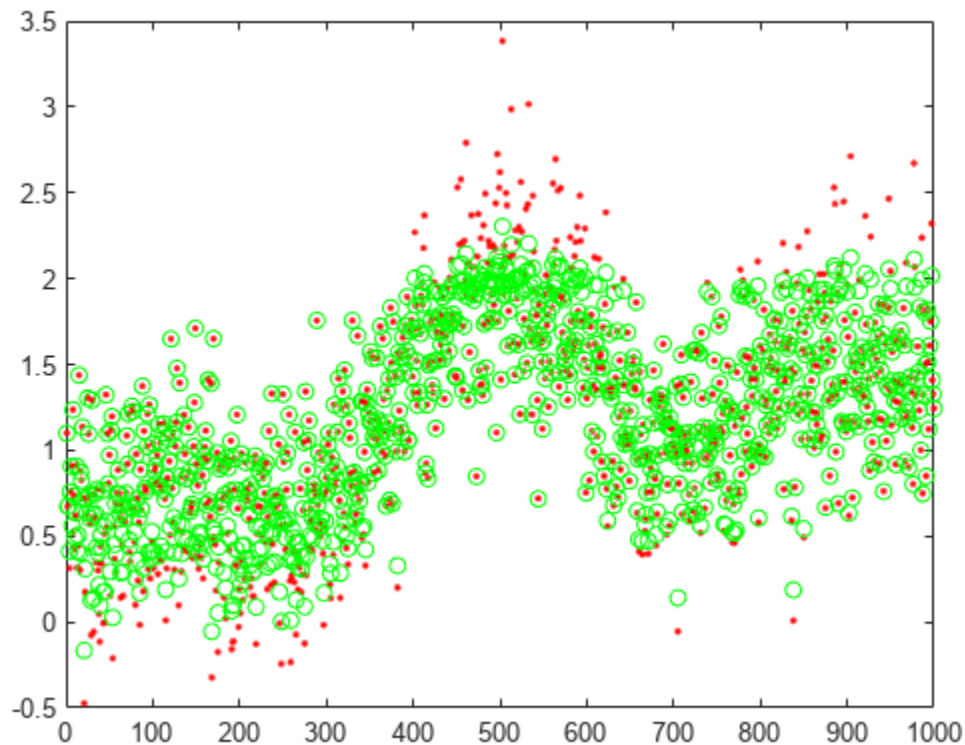
Predict the test data using the trained autoencoder, `autoenc`.



```
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);
```

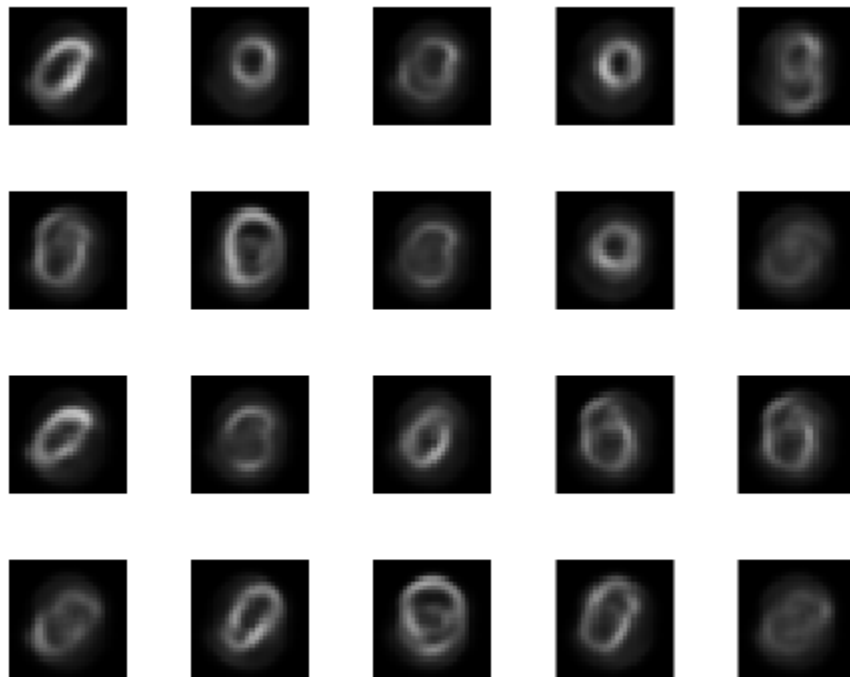
View the actual test data.

```
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 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.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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 $f(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \geq 1 \end{cases}$

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 $f(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \geq 1 \end{cases}$
'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 2-613 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**

'msesparsed' (default)

Loss function to use for training, specified as the comma-separated pair consisting of 'LossFunction' and 'msesparsed'. It corresponds to the mean squared error function adjusted for training a sparse autoencoder on page 2-613 as follows:

$$E = \frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K (x_{kn} - \hat{x}_{kn})^2 + \lambda * \underbrace{\rho_{L_2}^{\text{weights}}}_{\text{regularization}} + \beta * \underbrace{\rho_{\text{sparsity}}^{\text{sparsity}}}_{\text{regularization}},$$

where  $\lambda$  is the coefficient for the  $L_2$  regularization term on page 2-613 and  $\beta$  is the coefficient for the sparsity regularization term on page 2-613. You can specify the values of  $\lambda$  and  $\beta$  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 2-613.

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 2-613 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. 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  $\hat{x}$ .

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:

$$z = h^{(1)}(W^{(1)}x + b^{(1)}),$$

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:

$$\hat{x} = h^{(2)}(W^{(2)}z + b^{(2)}),$$

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,  $W^{(2)} \in \mathbb{R}^{D_x \times D^{(1)}}$  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:

$$\hat{\rho}_i = \frac{1}{n} \sum_{j=1}^n z_i^{(1)}(x_j) = \frac{1}{n} \sum_{j=1}^n h(w_i^{(1)T} x_j + b_i^{(1)}),$$

where  $n$  is the total number of training examples.  $x_j$  is the  $j$ th training example,  $w_i^{(1)T}$  is the  $i$ th row of the weight matrix  $W^{(1)}$ , and  $b_i^{(1)}$  is the  $i$ th 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  $\hat{\rho}_i$  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,  $\hat{\rho}_i$ , of a neuron  $i$  and its desired value,  $\rho$ , are not close in value [2]. One such sparsity regularization term can be the Kullback-Leibler divergence.

$$\Omega_{sparsity} = \sum_{i=1}^{D^{(1)}} KL(\rho \parallel \hat{\rho}_i) = \sum_{i=1}^{D^{(1)}} \rho \log\left(\frac{\rho}{\hat{\rho}_i}\right) + (1 - \rho) \log\left(\frac{1 - \rho}{1 - \hat{\rho}_i}\right)$$

Kullback-Leibler divergence is a function for measuring how different two distributions are. In this case, it takes the value zero when  $\rho$  and  $\hat{\rho}_i$  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  $\rho$  and  $\hat{\rho}_i$  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<sub>2</sub> 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<sub>2</sub> regularization term and is defined by:

$$\Omega_{weights} = \frac{1}{2} \sum_{l=1}^L \sum_{j=1}^{n_l} \sum_{i=1}^{k_l} (w_{ji}^{(l)})^2,$$

where  $L$  is the number of hidden layers,  $n_l$  is the output size of layer  $l$ , and  $k_l$  is the input size of layer  $l$ . The  $L_2$  regularization term is the sum of the squared elements of the weight matrices for each layer.

### Cost Function

The cost function for training a sparse autoencoder on page 2-613 is an adjusted mean squared error function as follows:

$$E = \underbrace{\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K (x_{kn} - \hat{x}_{kn})^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{L_2}}_{\text{regularization}} + \beta * \underbrace{\Omega_{sparsity}}_{\text{regularization}},$$

where  $\lambda$  is the coefficient for the  $L_2$  regularization term on page 2-613 and  $\beta$  is the coefficient for the sparsity regularization term on page 2-613. You can specify the values of  $\lambda$  and  $\beta$  by using the `L2WeightRegularization` and `SparsityRegularization` name-value pair arguments, respectively, while training an autoencoder.

## Version History

Introduced in R2015b

### 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"



# 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.

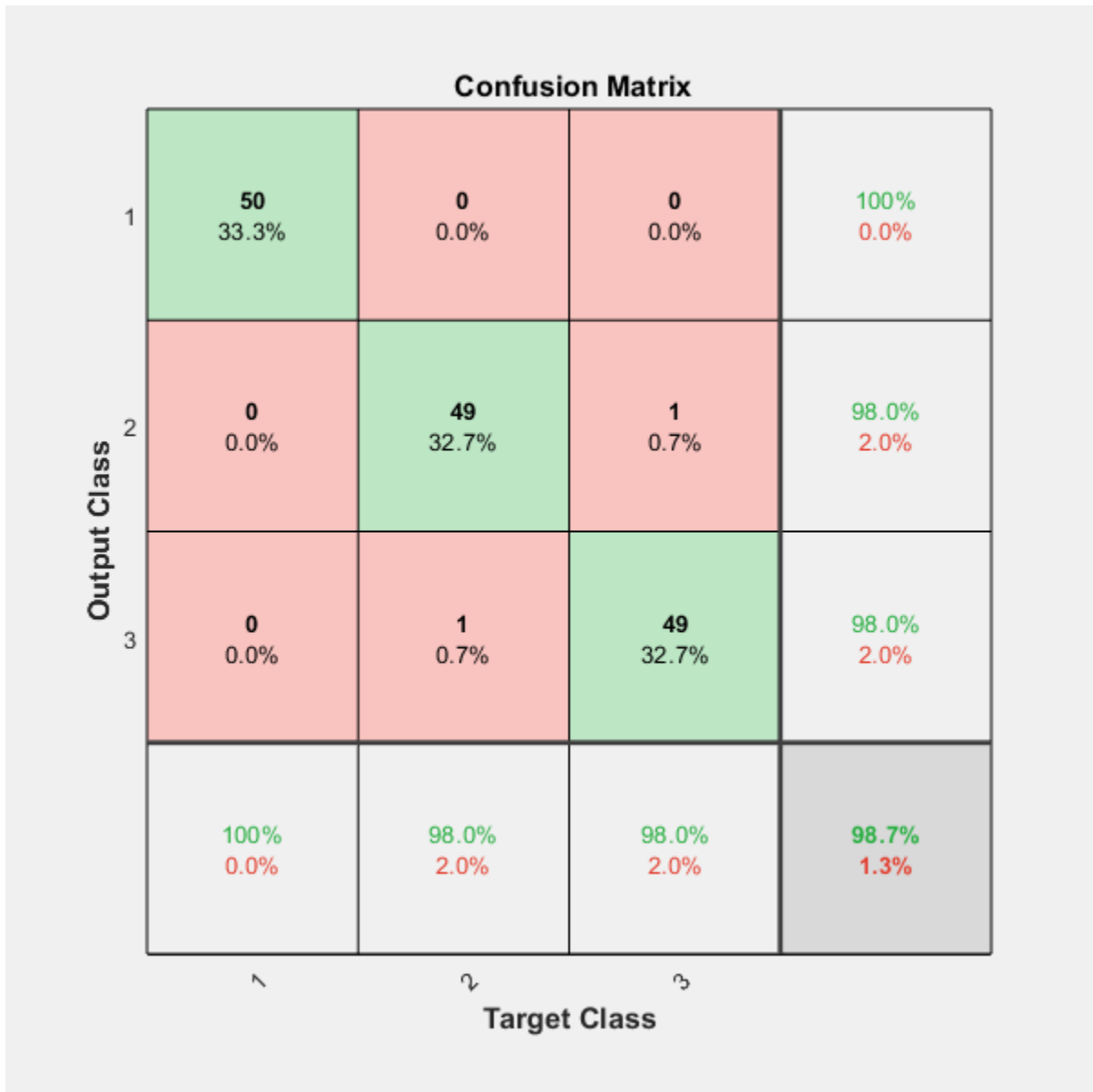
```
net = trainSoftmaxLayer(X,T);
```

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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

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:

$$E = \frac{1}{n} \sum_{j=1}^n \sum_{i=1}^k (t_{ij} - y_{ij})^2,$$

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  $\mathbf{x}_j$ .

The cross entropy function is given by:

$$E = \frac{1}{n} \sum_{j=1}^n \sum_{i=1}^k t_{ij} \ln y_{ij} + (1 - t_{ij}) \ln(1 - y_{ij}).$$

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`.

### **Version History**

**Introduced in R2015b**

### **See Also**

`trainAutoencoder` | `stack`

# 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

<code>decode</code>	Decode encoded data
<code>encode</code>	Encode input data
<code>generateFunction</code>	Generate a MATLAB function to run the autoencoder
<code>generateSimulink</code>	Generate a Simulink model for the autoencoder
<code>network</code>	Convert <code>Autoencoder</code> object into <code>network</code> object
<code>plotWeights</code>	Plot a visualization of the weights for the encoder of an autoencoder
<code>predict</code>	Reconstruct the inputs using trained autoencoder
<code>stack</code>	Stack encoders from several autoencoders together
<code>view</code>	View autoencoder

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects](#).

## Version History

**Introduced in R2015b**

### See Also

`trainAutoencoder`

### Topics

Class Attributes

Property Attributes

## decode

**Class:** Autoencoder

Decode encoded data

### Syntax

```
Y = decode(autoenc,Z)
```

### Description

`Y = decode(autoenc,Z)` returns the decoded data on page 2-623 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:

$$z = h^{(1)}(W^{(1)}x + b^{(1)}),$$

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,  $\hat{x}$ , as follows:

$$\hat{x} = h^{(2)}(W^{(2)}z + b^{(2)}),$$

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,  $W^{(2)} \in \mathbb{R}^{D_x \times D^{(1)}}$  is a weight matrix, and  $b^{(2)} \in \mathbb{R}^{D_x}$  is a bias vector.

## Version History

**Introduced in R2015b**

### See Also

encode | trainAutoencoder

## encode

**Class:** Autoencoder

Encode input data

### Syntax

```
Z = encode(autoenc, Xnew)
```

### Description

`Z = encode(autoenc, Xnew)` returns the encoded data on page 2-625, `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:

$$z = h^{(1)}(W^{(1)}x + b^{(1)}),$$

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.

## Version History

**Introduced in R2015b**

### See Also

[trainAutoencoder](#) | [decode](#) | [stack](#)

# 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 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.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

### **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`.

## Version History

Introduced in R2015b

## See Also

`genFunction` | `generateSimulink`

# 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 = digitSmallCellArrayData;
```

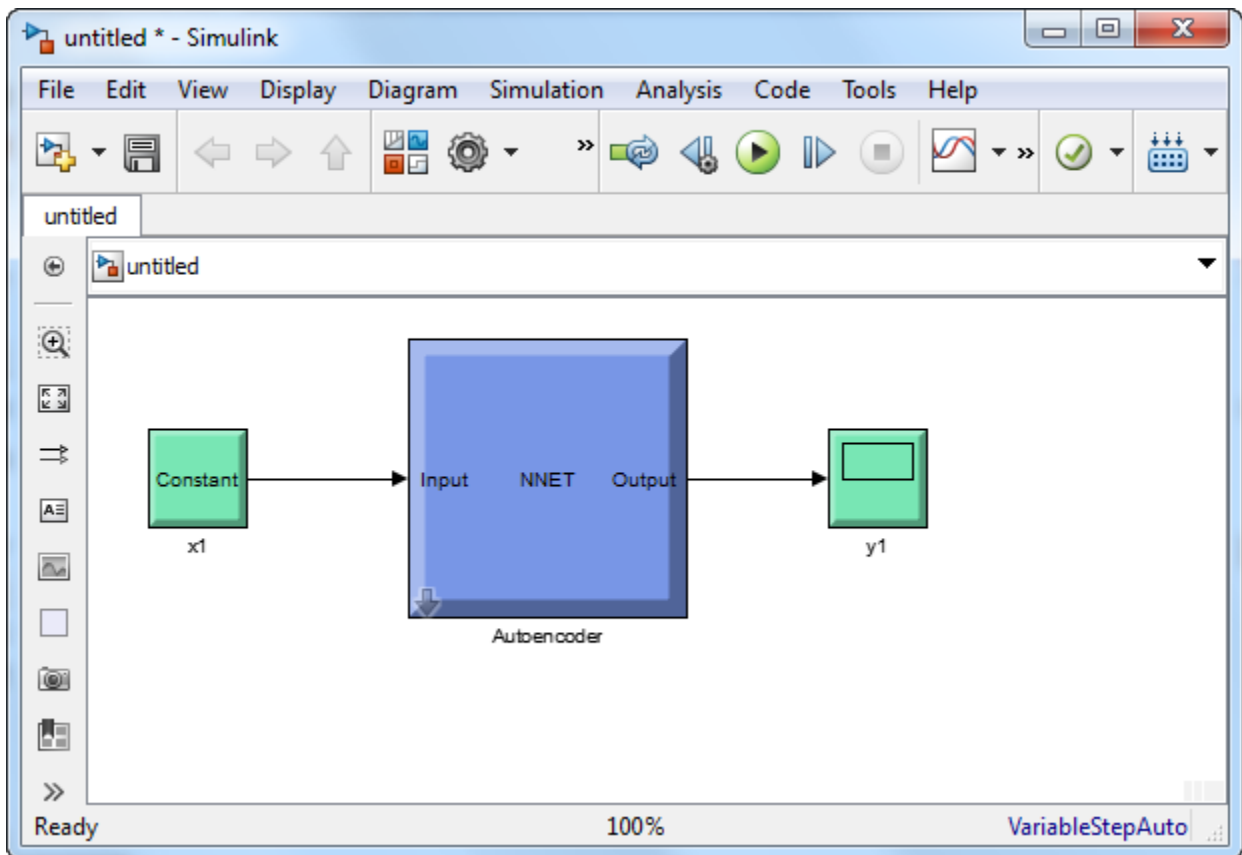
The training data is a 1-by-500 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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)
```



## Version History

Introduced in R2015b

### See Also

`trainAutoencoder`

## 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);
```



Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:07	-
Performance	482	1.67	0
Gradient	1.11e+03	0.349	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
Data Division: Training Only `dividetrain`  
Training: Scaled Conjugate Gradient `trainscg`  
Performance: Mean Squared Error with L2 and Sparsity Regularizers  
Calculations: MEX

**Training Plots**  
Performance

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
```

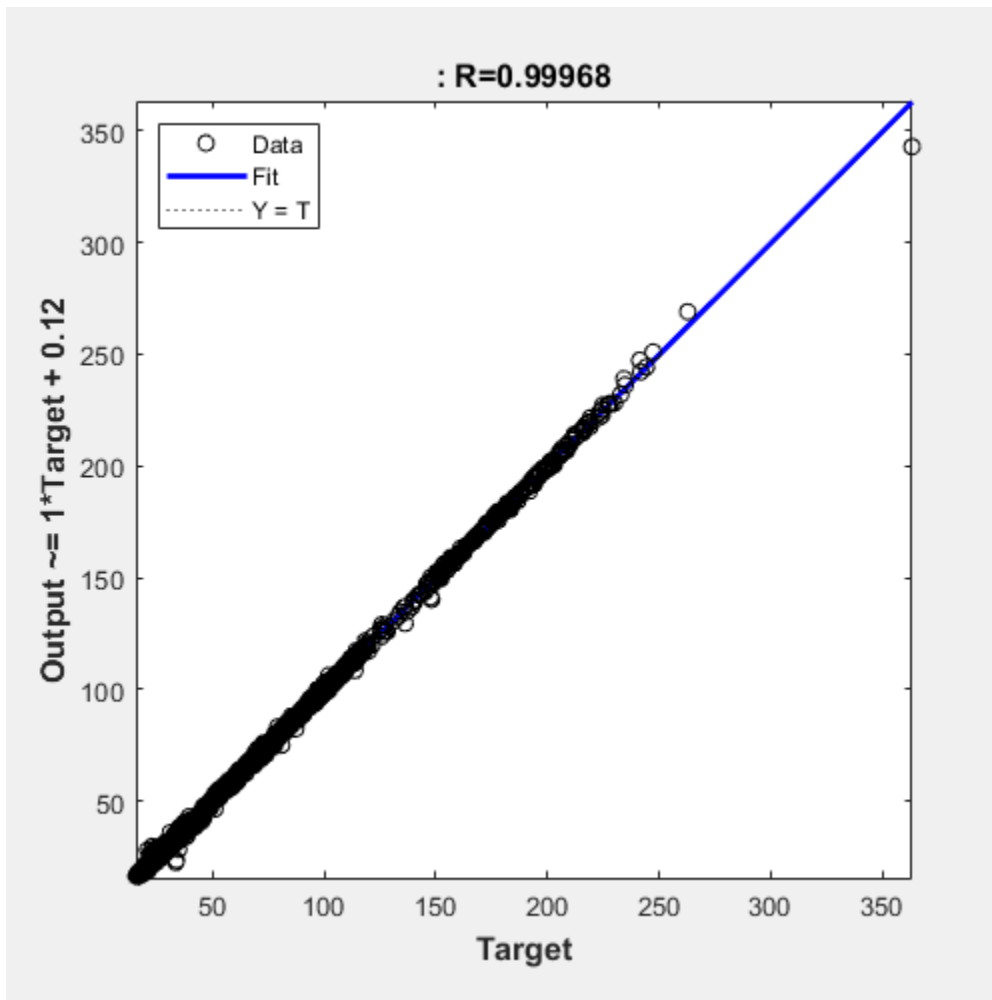
```
S = 0.9983
```

```
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.

## **Version History**

**Introduced in R2015b**

### **See Also**

`trainAutoencoder` | `Autoencoder`

## 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;
```

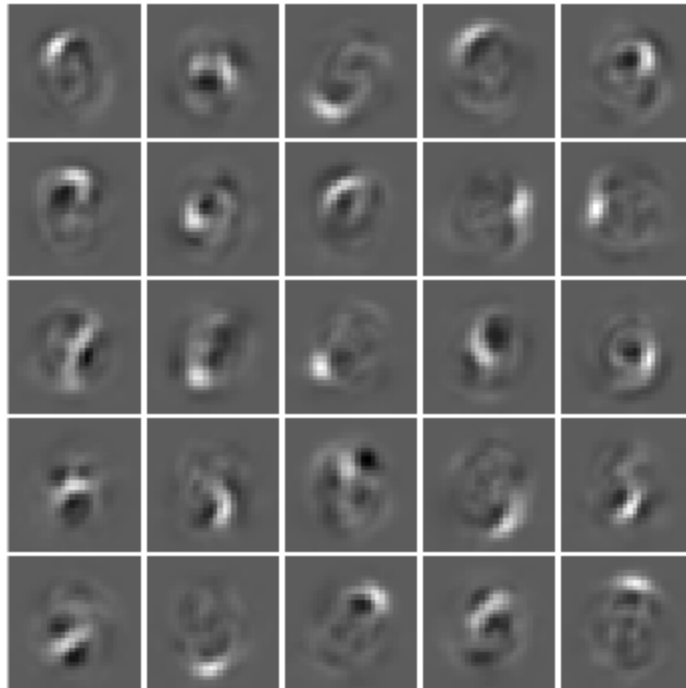
The training data is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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.

## Version History

Introduced in R2015b

## See Also

`trainAutoencoder`

## 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');
```

Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✓

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:06	-
Performance	3.14	1.62	0
Gradient	19.1	0.00181	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
Data Division: Training Only `dividetrain`  
Training: Scaled Conjugate Gradient `trainscg`  
Performance: Mean Squared Error with L2 and Sparsity Regularizers  
Calculations: MEX

**Training Plots**  
Performance

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.

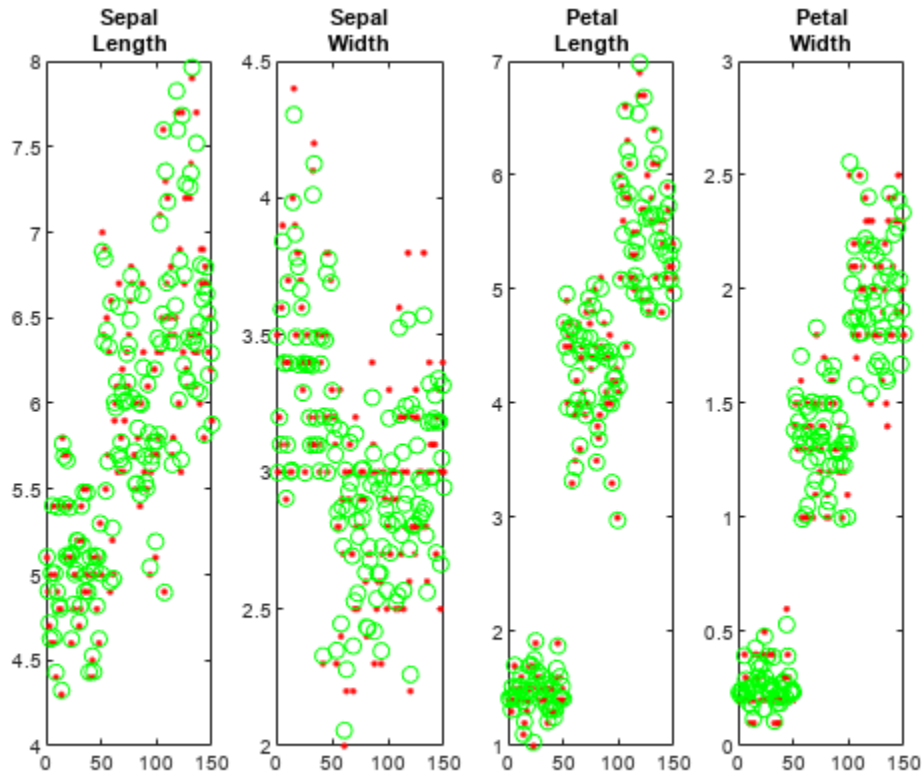
```
for i = 1:4
h(i) = subplot(1,4,i);
```



```

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;
```

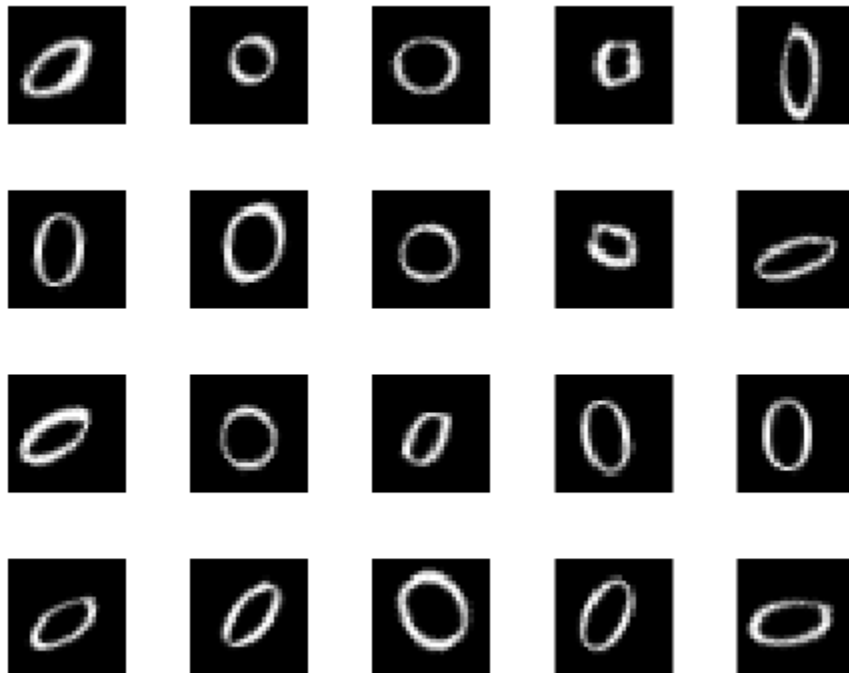
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, autoenc.

```
xReconstructed = predict(autoenc,XTest);
```

View the actual test data.

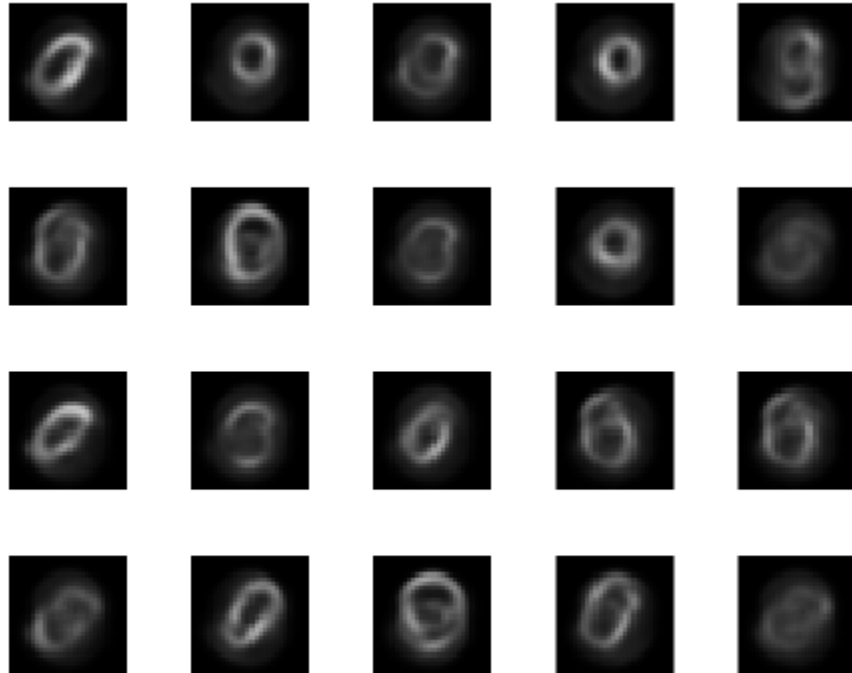
```
figure;  
for i = 1:20  
    subplot(4,5,i);  
    imshow(XTest{i});  
end
```



View the reconstructed test data.

```
figure;  
for i = 1:20  
    subplot(4,5,i);
```

```
imshow(xReconstructed{i});  
end
```



## Version History

Introduced in R2015b

### See Also

[trainAutoencoder](#)

## 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');
```

Network Diagram

**Training Results**  
 Training finished: Reached maximum number of epochs ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:12	-
Performance	6.12	0.0642	0
Gradient	2.13	0.00306	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Training Only `dividetrain`  
 Training: Scaled Conjugate Gradient `trainscg`  
 Performance: Mean Squared Error with L2 and Sparsity Regularizers  
 Calculations: MEX

**Training Plots**  
 Performance

Extract the features in the hidden layer.

```
features = encode(autoenc,X);
```

Train a softmax layer for classification using the features .

```
softnet = trainSoftmaxLayer(features,T);
```

Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✔

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:02	-
Performance	0.366	0.0137	0
Gradient	0.0202	1.38e-05	1e-06
Validation Checks	0	0	6

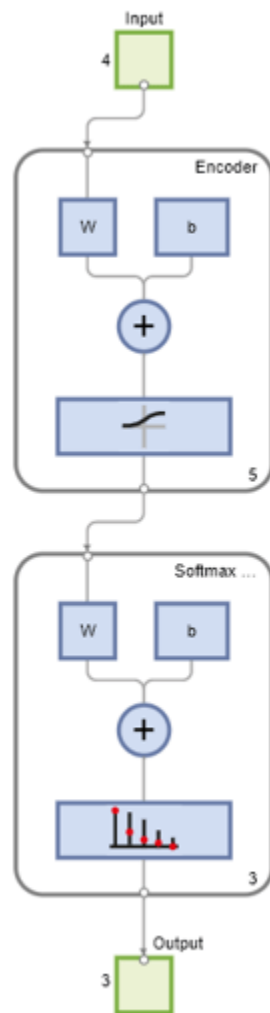
**Training Algorithms**  
 Data Division: Training Only `dividetrain`  
 Training: Scaled Conjugate Gradient `trainscg`  
 Performance: Cross Entropy `crossentropy`  
 Calculations: MEX

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`.

## Version History

Introduced in R2015b



## **See Also**

[trainAutoencoder](#) | [Autoencoder](#)

## **Topics**

["Train Stacked Autoencoders for Image Classification"](#)

## 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');
```

Network Diagram

**Training Results**  
Training finished: Reached maximum number of epochs ✔

**Training Progress**

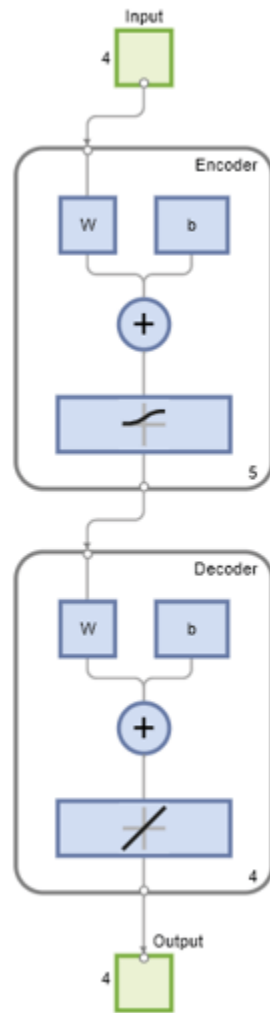
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:05	-
Performance	6.12	0.0642	0
Gradient	2.13	0.00306	1e-06
Validation Checks	0	0	6

**Training Algorithms**  
 Data Division: Training Only `dividetrain`  
 Training: Scaled Conjugate Gradient `trainscg`  
 Performance: Mean Squared Error with L2 and Sparsity Regularizers  
 Calculations: MEX

**Training Plots**  
Performance

View the autoencoder.

```
view(autoenc)
```



## Version History

Introduced in R2015b

### See Also

`trainAutoencoder`

# 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;
```

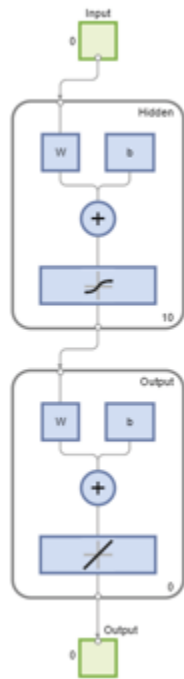
The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

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);
```

Network Diagram

**Training Results**

Training finished: Met validation criterion 

**Training Progress**

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	19	1000
Elapsed Time	-	00:00:02	-
Performance	324	9.25e-05	0
Gradient	414	0.000292	1e-07
Mu	0.001	1e-05	1e+10
Validation Checks	0	6	6

**Training Algorithms**

Data Division: Random dividerand

Training: Levenberg-Marquardt trainlm

Performance: Mean Squared Error mse

Calculations: MEX

**Training Plots**

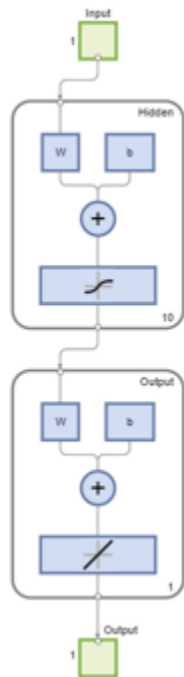
Performance Training State

Error Histogram Regression

Fit

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);
```



Network Diagram

### Training Results

Training finished: Reached maximum number of epochs ✔

### Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	1000	1000
Elapsed Time	-	00:00:04	-
Performance	77.6	1.77e-10	0
Gradient	144	2.1e-06	1e-07
Mu	0.005	500	1e+10
Effective # Param	31	28.3	0
Sum Squared P...	2.76e+03	358	0

### Training Algorithms

Data Division: Random dividerand  
 Training: Bayesian Regularization trainbr  
 Performance: Mean Squared Error mse  
 Calculations: MEX

### Training Plots

```
y = net(x);
perf = perform(net,y,t)
```

```
perf = 3.3416e-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-Ribière 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.

## Version History

Introduced in R2010b

## See Also

[train](#) | [feedforwardnet](#) | [network](#) | [nftool](#) | [trainlm](#) | [perform](#)

## Topics

[“Fit Data with a Shallow Neural Network”](#)

[“Neural Network Object Properties”](#)

[“Neural Network Subobject Properties”](#)

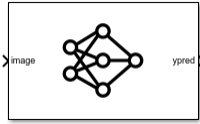


# Deep Learning Blocks

---

## Image Classifier

Classify data using a trained deep learning neural network



### Libraries:

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 NaNs, 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'

## Version History

Introduced in R2020b

## Extended Capabilities

**C/C++ Code Generation**

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

Usage notes and limitations:



- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C++. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to None generates generic C++ code that does not depend on third-party libraries.
- 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:

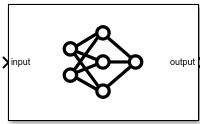
- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to C++.
- For a list of networks and layers supported for CUDA code generation, see “Supported Networks, Layers, and Classes” (GPU Coder).
- To learn more about generating code for Simulink models containing the Image Classifier block, see “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” (GPU Coder).

### See Also

Predict

## Predict

Predict responses using a trained deep learning neural network



### Libraries:

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- <code>numFeatures</code> numeric array, where $N$ is the number of observations and <code>numFeatures</code> is the number of features of the input data.

If the array contains NaNs, 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  
squeezeenet (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:** 'squeezeenet'

**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:** ''

## Version History

Introduced in R2020b

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C++. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to None generates generic C++ code that does not depend on third-party libraries.
- 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:

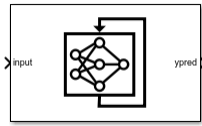
- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to C++.
- For a list of networks and layers supported for CUDA code generation, see “Supported Networks, Layers, and Classes” (GPU Coder).
- To learn more about generating code for Simulink 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

## Stateful Classify

Classify data using a trained deep learning recurrent neural network



### Libraries:

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.

To reset the state of recurrent neural network to its initial state, place the Stateful Classify block inside a Resettable Subsystem block and use the Reset control signal as trigger.

### 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'

## Version History

Introduced in R2021a

## Extended Capabilities

**C/C++ Code Generation**

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



Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C++. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to None generates generic C++ code that does not depend on third-party libraries.
- 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:

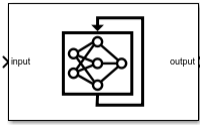
- The **Language** parameter in the **Configuration Parameters > Code Generation** general category must be set to C++.
- GPU code generation supports this block only when targeting the cuDNN library.

### See Also

Predict | Image Classifier | Stateful Predict

## Stateful Predict

Predict responses using a trained recurrent neural network



### Libraries:

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.

To reset the state of recurrent neural network to its initial state, place the Stateful Predict block inside a Resettable Subsystem block and use the Reset control signal as trigger.

## 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'

## **Version History**

**Introduced in R2021a**

## **Extended Capabilities**

### **C/C++ Code Generation**

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

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to C++. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to None generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation > Interface** pane must be enabled.
- Code generation for the Stateful Predict block does not support dlnetwork.
- 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++.
- GPU code generation supports this block only when targeting the cuDNN library.

- Code generation for the Stateful Predict block does not support `dlnetwork`.

**See Also**

[Predict](#) | [Image Classifier](#) | [Stateful Classify](#)

