Deep Learning Toolbox™

Getting Started Guide

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth

MATLAB®

R2022**b**

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Deep Learning Toolbox™ Getting Started Guide

Revision History

June 1992	First printing	
April 1993	Second printing	
January 1997	Third printing	
July 1997	Fourth printing	
January 1998	Fifth printing	Revised for Version 3 (Release 11)
September 2000	Sixth printing	Revised for Version 4 (Release 12)
June 2001	Seventh printing	Minor revisions (Release 12.1)
July 2002	Online only	Minor revisions (Release 13)
January 2003	Online only	Minor revisions (Release 13SP1)
June 2004	Online only	Revised for Version 4.0.3 (Release 14)
October 2004	Online only	Revised for Version 4.0.4 (Release 14SP1)
October 2004	Eighth printing	Revised for Version 4.0.4
March 2005	Online only	Revised for Version 4.0.5 (Release 14SP2)
March 2006	Online only	Revised for Version 5.0 (Release 2006a)
September 2006	Ninth printing	Minor revisions (Release 2006b)
March 2007	Online only	Minor revisions (Release 2007a)
September 2007	Online only	Revised for Version 5.1 (Release 2007b)
March 2008	Online only	Revised for Version 6.0 (Release 2008a)
October 2008	Online only	Revised for Version 6.0.1 (Release 2008b)
March 2009	Online only	Revised for Version 6.0.2 (Release 2009a)
September 2009	Online only	Revised for Version 6.0.3 (Release 2009b)
March 2010	Online only	Revised for Version 6.0.4 (Release 2010a)
September 2010	Tenth printing	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)

Contents

1 Getting Started

Getting Started

Deep Learning Toolbox Product Description

Design, train, and analyze deep learning networks

Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps. You can use convolutional neural networks (ConvNets, CNNs) and long short-term memory (LSTM) networks to perform classification and regression on image, time-series, and text data. You can build network architectures such as generative adversarial networks (GANs) and Siamese networks using automatic differentiation, custom training loops, and shared weights. With the Deep Network Designer app, you can design, analyze, and train networks graphically. The Experiment Manager app helps you manage multiple deep learning experiments, keep track of training parameters, analyze results, and compare code from different experiments. You can visualize layer activations and graphically monitor training progress.

You can import networks and layer graphics from TensorFlow™ 2, TensorFlow-Keras, and PyTorch®, the ONNX™ (Open Neural Network Exchange) model format, and Caffe. You can also export Deep Learning Toolbox networks and layer graphs to TensorFlow 2 and the ONNX model format. The toolbox supports transfer learning with DarkNet-53, ResNet-50, NASNet, SqueezeNet and many other pretrained models.

You can speed up training on a single- or multiple-GPU workstation (with Parallel Computing Toolbox™), or scale up to clusters and clouds, including NVIDIA® GPU Cloud and Amazon EC2® GPU instances (with MATLAB® Parallel Server™).

Get Started with Deep Network Designer

This example shows how to use Deep Network Designer to adapt a pretrained GoogLeNet 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 for Training

In the workspace, unzip the data.

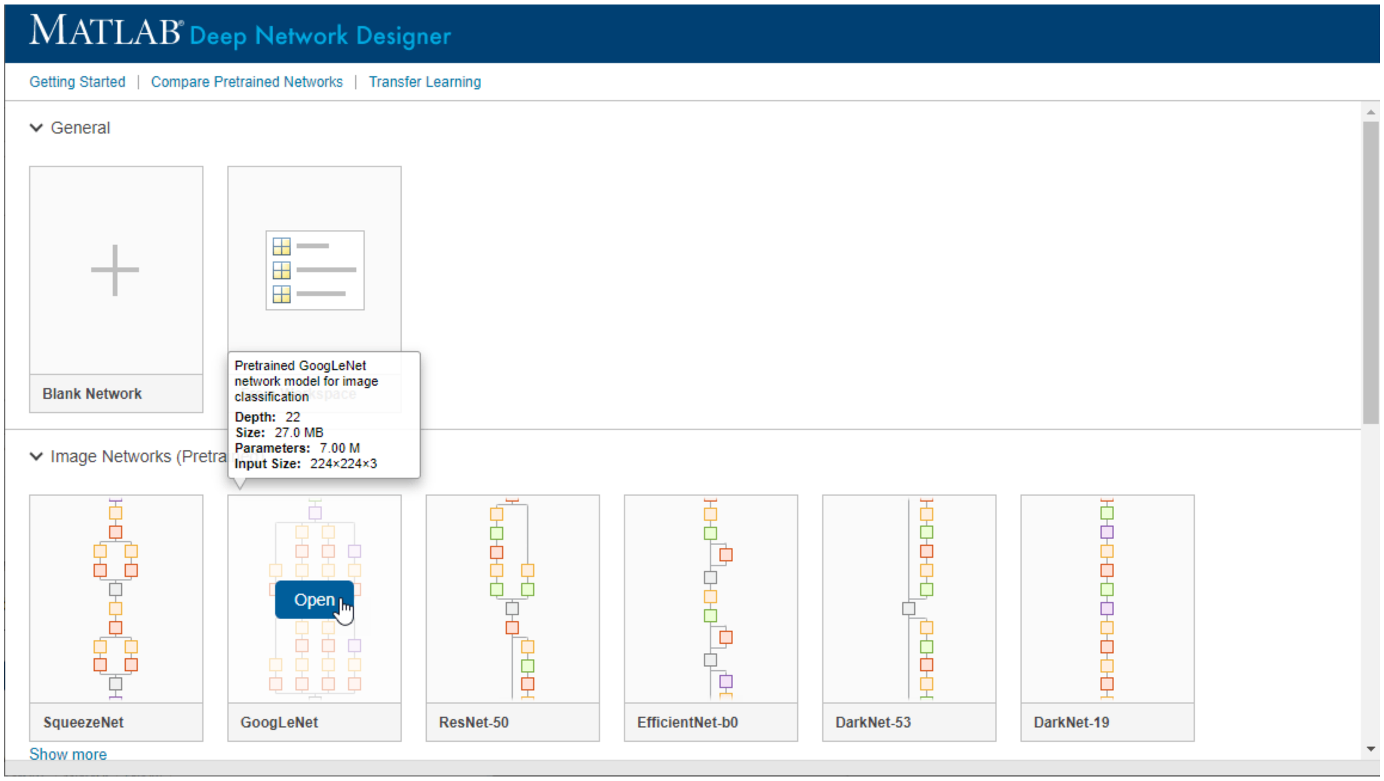
```
unzip('MerchData.zip');
```

Select a Pretrained Network

Open Deep Network Designer.

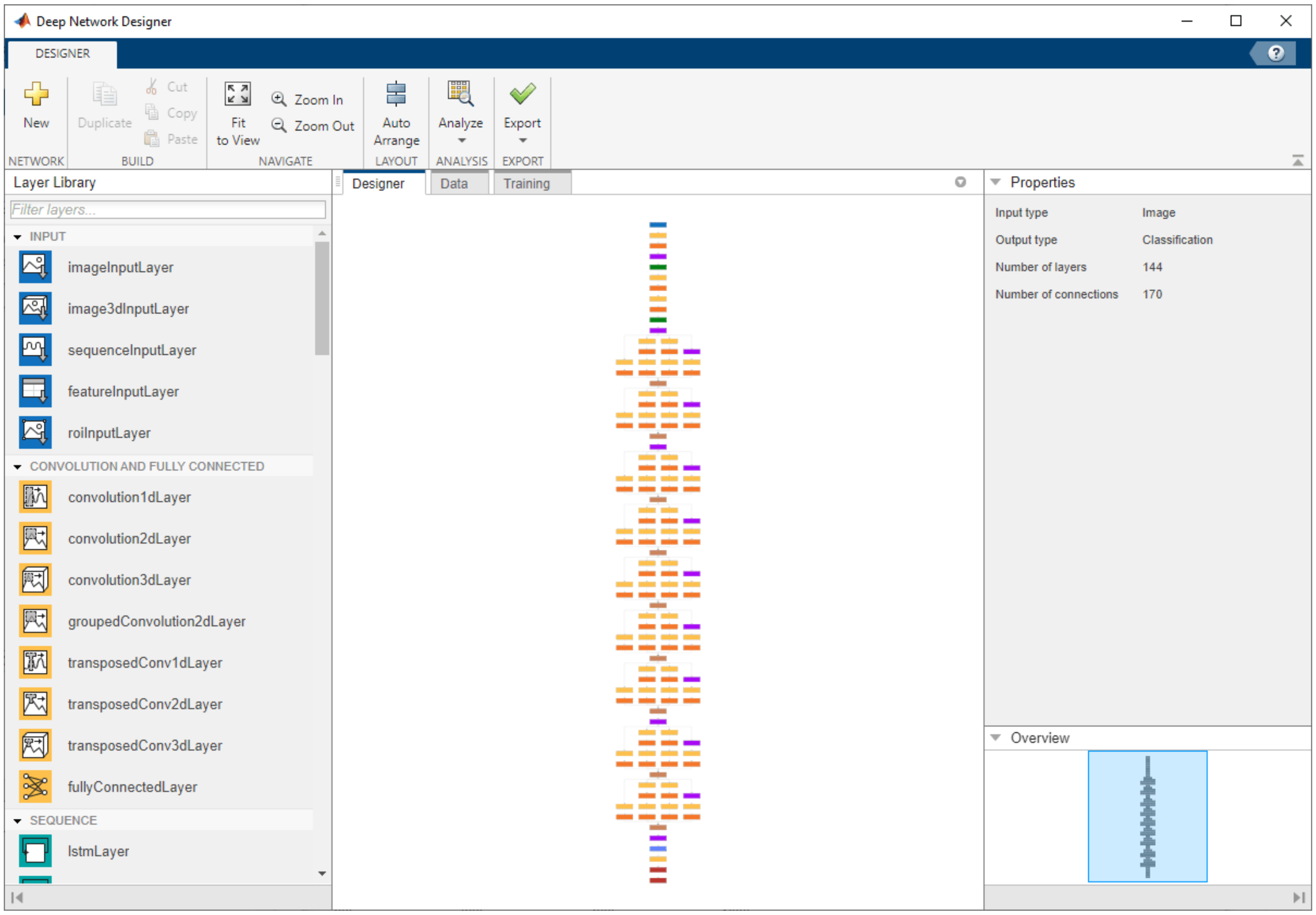
```
deepNetworkDesigner
```

Load a pretrained GoogLeNet network by selecting it from the Deep Network Designer Start Page. If you need to download the network, then click **Install** to open the Add-On Explorer.



Pretrained GoogLeNet network model for image classification
Depth: 22
Size: 27.0 MB
Parameters: 7.00 M
Input Size: 224×224×3

Deep Network Designer displays a zoomed-out view of the whole network. Explore the network plot. To zoom in with the mouse, use **Ctrl**+scroll wheel.



Input type	Image
Output type	Classification
Number of layers	144
Number of connections	170

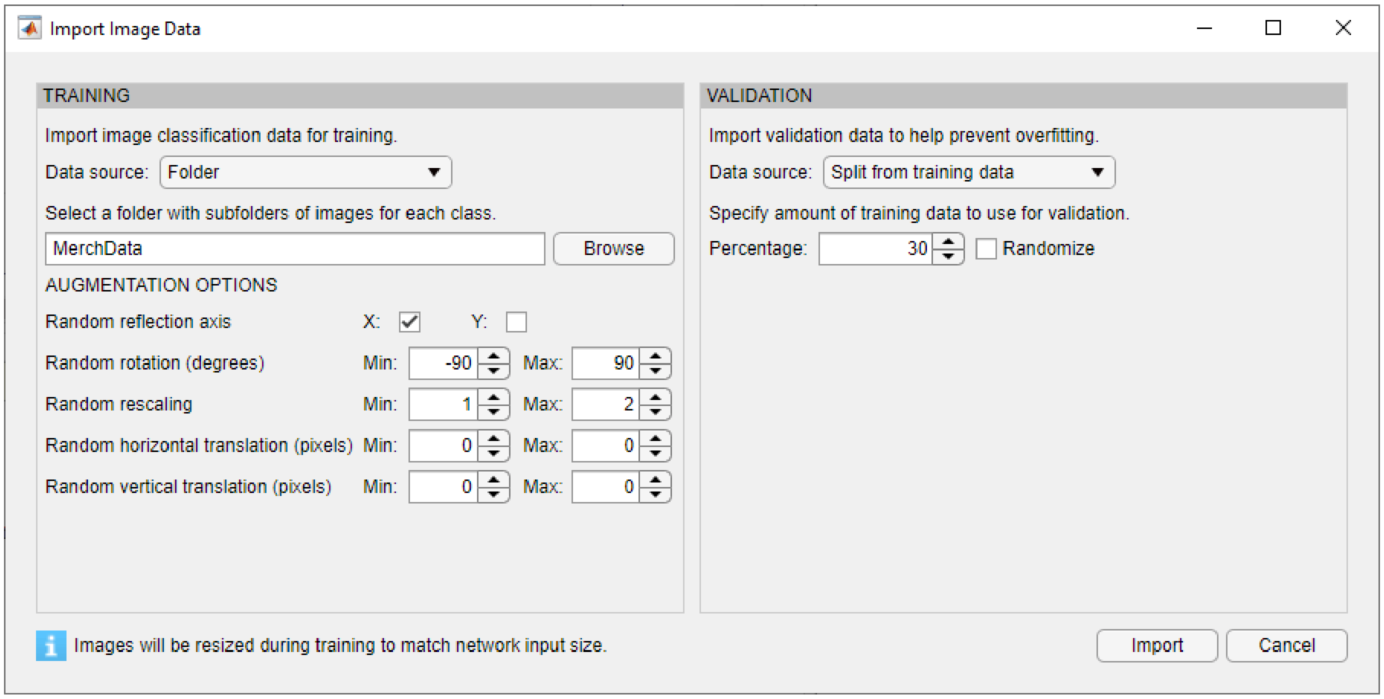
Load Data Set

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

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

The dialog box also allows you to split the validation data from within the app. 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].


Import Image Data

TRAINING

Import image classification data for training.

Data source: Folder

Select a folder with subfolders of images for each class.

MerchData

AUGMENTATION OPTIONS

Random reflection axis	X:	✓	Y:	
Random rotation (degrees)	Min:	-90	Max:	90
Random rescaling	Min:	1	Max:	2
Random horizontal translation (pixels)	Min:	0	Max:	0
Random vertical translation (pixels)	Min:	0	Max:	0

VALIDATION

Import validation data to help prevent overfitting.

Data source: Split from training data

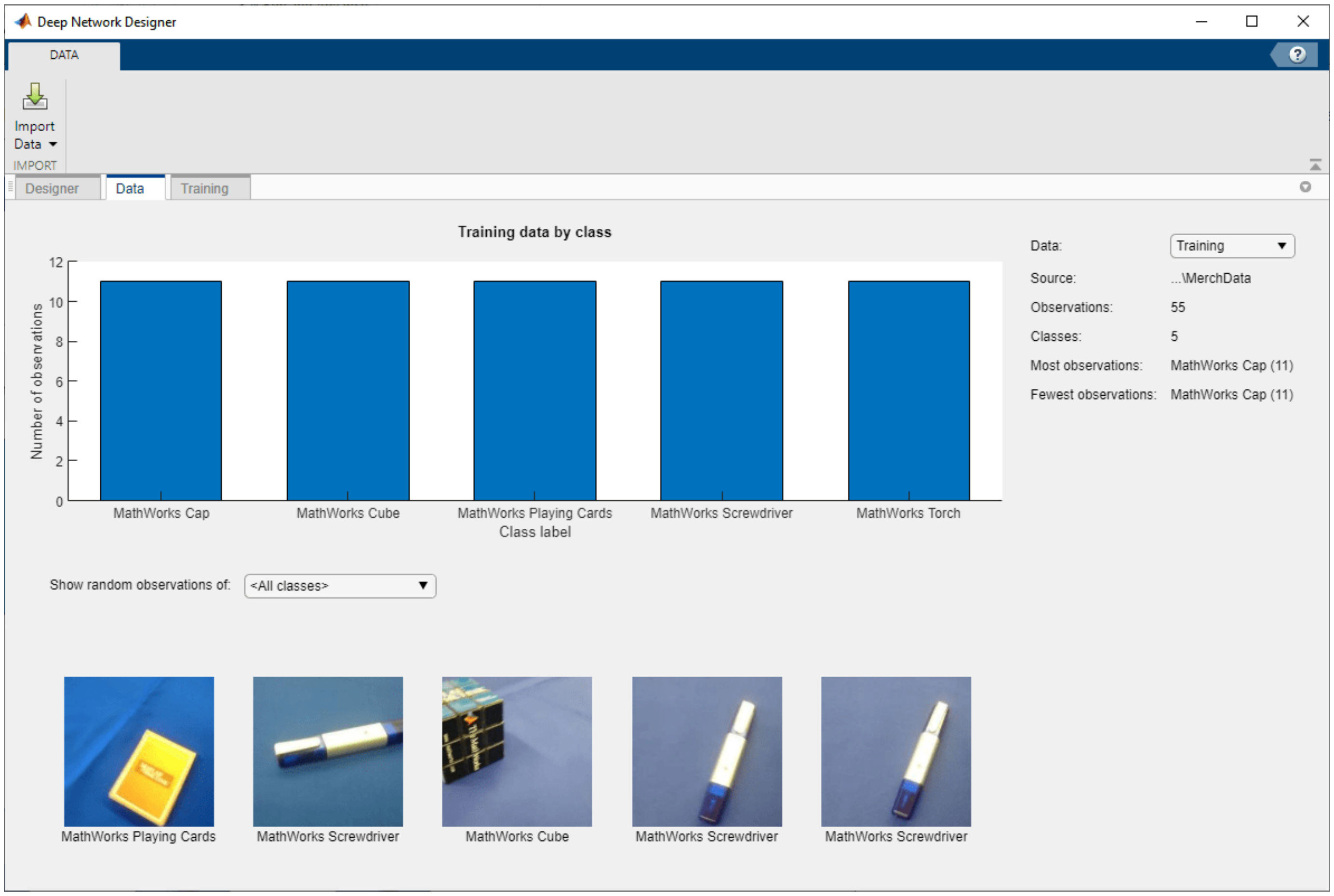
Specify amount of training data to use for validation.

Percentage: 30 Randomize

Images will be resized during training to match network input size.

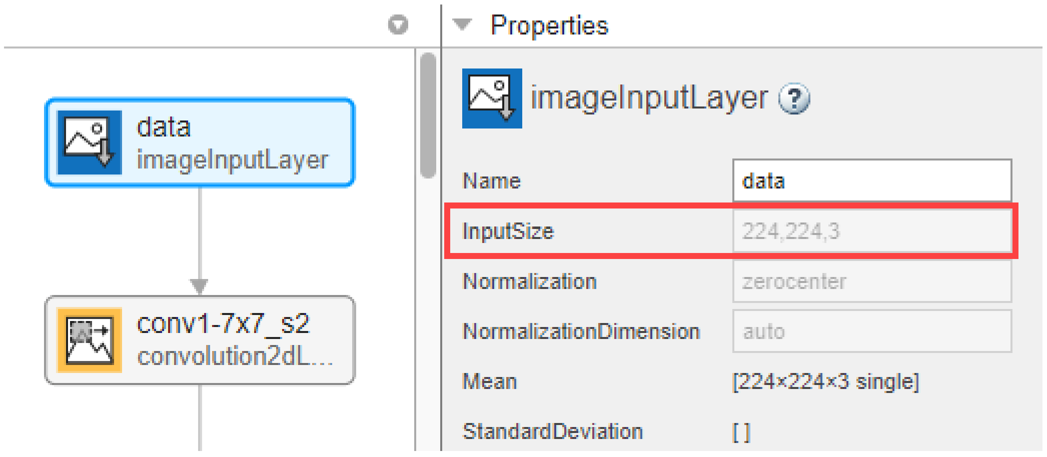
Click **Import** to import the data into Deep Network Designer.

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



Data:	Training
Source:	...\MerchData
Observations:	55
Classes:	5
Most observations:	MathWorks Cap (11)
Fewest observations:	MathWorks Cap (11)

Deep Network Designer resizes the images during training to match the network input size. To view the network input size, in the **Designer** tab, click the `imageInputLayer`. This network has an input size of 224-by-224.



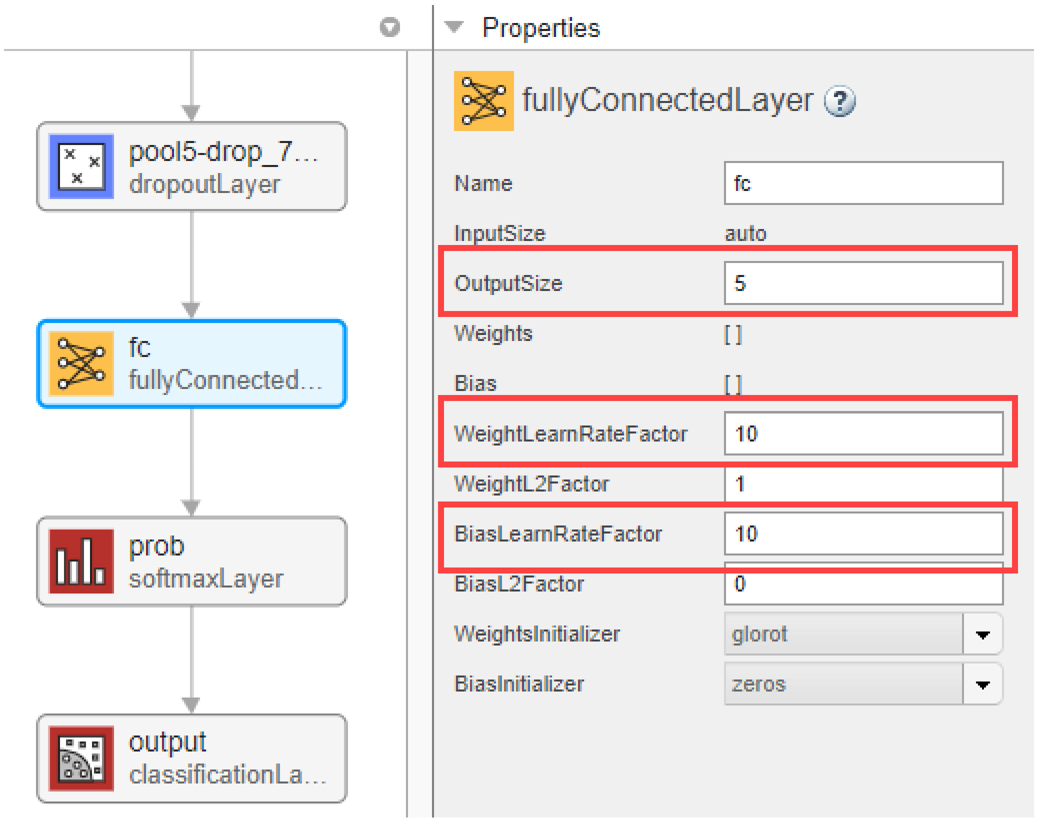
Name	data
InputSize	224,224,3
Normalization	zerocenter
NormalizationDimension	auto
Mean	[224×224×3 single]
StandardDeviation	[ ]

Edit Network for Transfer Learning

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

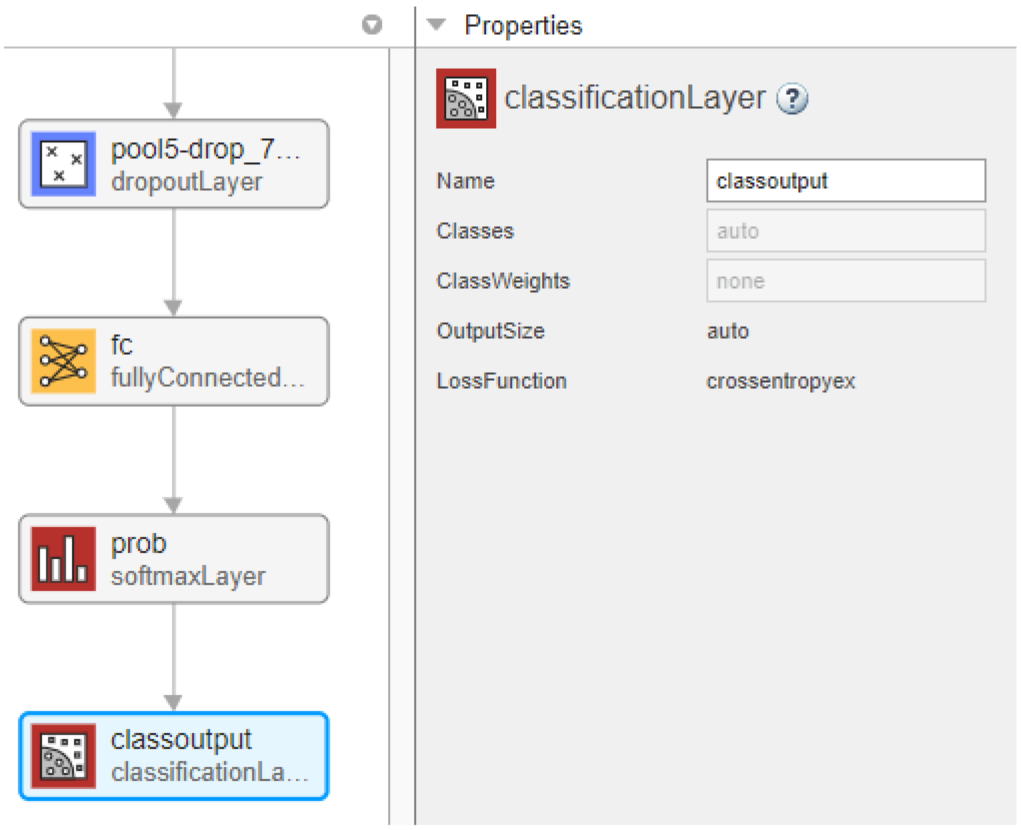
In the **Designer** tab, drag a new `fullyConnectedLayer` from the **Layer Library** onto the canvas. Set `OutputSize` to the number of classes in the new data, in this example, `5`.

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



Name	fc
InputSize	auto
OutputSize	5
Weights	[]
Bias	[]
WeightLearnRateFactor	10
WeightL2Factor	1
BiasLearnRateFactor	10
BiasL2Factor	0
WeightsInitializer	glorot
BiasInitializer	zeros

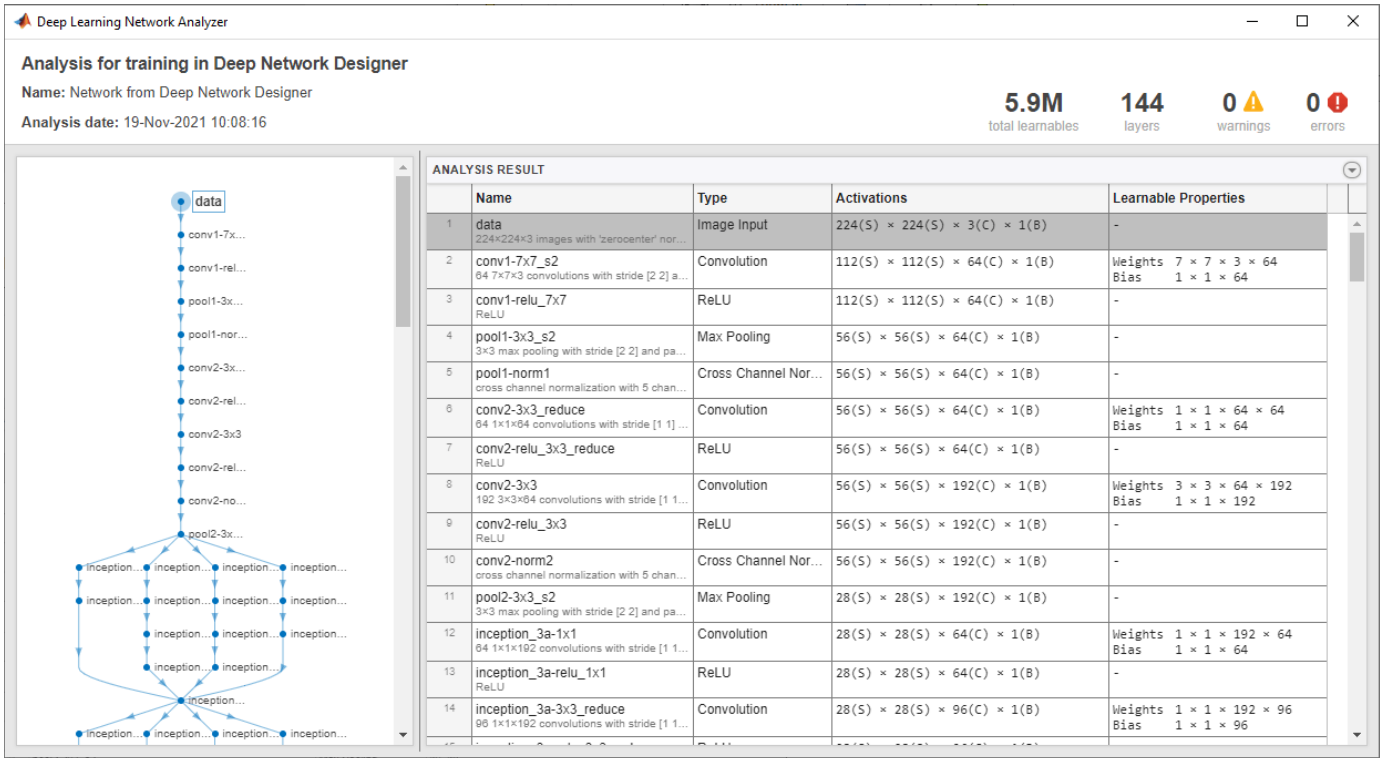
Replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original `output` layer and connect your new layer instead.



Name	classoutput
Classes	auto
ClassWeights	none
OutputSize	auto
LossFunction	crossentropyex

Check Network

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



Analysis for training in Deep Network Designer

Name: Network from Deep Network Designer

Analysis date: 19-Nov-2021 10:08:16

5.9M total learnables | 144 layers | 0 warnings | 0 errors

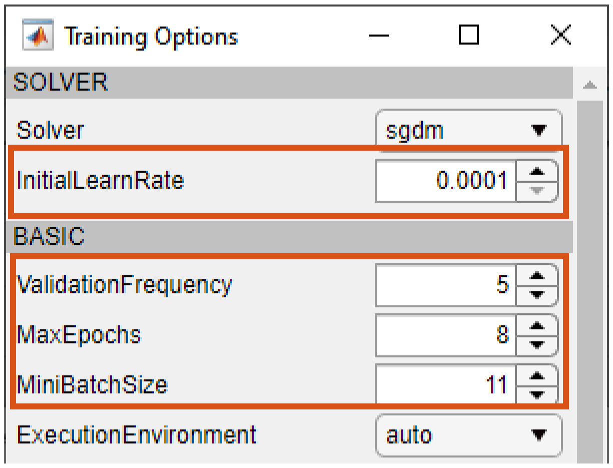
	Name	Type	Activations	Learnable Properties
1	data 224×224×3 images with 'zerocenter' nor...	Image Input	224(S) × 224(S) × 3(C) × 1(B)	-
2	conv1-7x7_s2 64 7×7×3 convolutions with stride [2 2] a...	Convolution	112(S) × 112(S) × 64(C) × 1(B)	Weights 7 × 7 × 3 × 64 Bias 1 × 1 × 64
3	conv1-relu_7x7 ReLU	ReLU	112(S) × 112(S) × 64(C) × 1(B)	-
4	pool1-3x3_s2 3×3 max pooling with stride [2 2] and pa...	Max Pooling	56(S) × 56(S) × 64(C) × 1(B)	-
5	pool1-norm1 cross channel normalization with 5 chan...	Cross Channel Nor...	56(S) × 56(S) × 64(C) × 1(B)	-
6	conv2-3x3_reduce 64 1×1×64 convolutions with stride [1 1] ...	Convolution	56(S) × 56(S) × 64(C) × 1(B)	Weights 1 × 1 × 64 × 64 Bias 1 × 1 × 64
7	conv2-relu_3x3_reduce ReLU	ReLU	56(S) × 56(S) × 64(C) × 1(B)	-
8	conv2-3x3 192 3×3×64 convolutions with stride [1 1...	Convolution	56(S) × 56(S) × 192(C) × 1(B)	Weights 3 × 3 × 64 × 192 Bias 1 × 1 × 192
9	conv2-relu_3x3 ReLU	ReLU	56(S) × 56(S) × 192(C) × 1(B)	-
10	conv2-norm2 cross channel normalization with 5 chan...	Cross Channel Nor...	56(S) × 56(S) × 192(C) × 1(B)	-
11	pool2-3x3_s2 3×3 max pooling with stride [2 2] and pa...	Max Pooling	28(S) × 28(S) × 192(C) × 1(B)	-
12	inception_3a-1x1 64 1×1×192 convolutions with stride [1 1...	Convolution	28(S) × 28(S) × 64(C) × 1(B)	Weights 1 × 1 × 192 × 64 Bias 1 × 1 × 64
13	inception_3a-relu_1x1 ReLU	ReLU	28(S) × 28(S) × 64(C) × 1(B)	-
14	inception_3a-3x3_reduce 96 1×1×192 convolutions with stride [1 1...	Convolution	28(S) × 28(S) × 96(C) × 1(B)	Weights 1 × 1 × 192 × 96 Bias 1 × 1 × 96

Train Network

To train the network with the default settings, on the **Training** tab, click **Train**.

If you want greater control over the training, click **Training Options** and choose the settings to train with. The default training options are better suited for large data sets. For small data sets, use smaller values for the mini-batch size and the validation frequency. For more information on selecting training options, see `trainingOptions`.

For this example, set **InitialLearnRate** to `0.0001`, **ValidationFrequency** to `5`, and **MaxEpochs** to `8`. As there are 55 observations, set **MiniBatchSize** to `11` to divide the training data evenly and ensure the whole training set is used during each epoch.



SOLVER

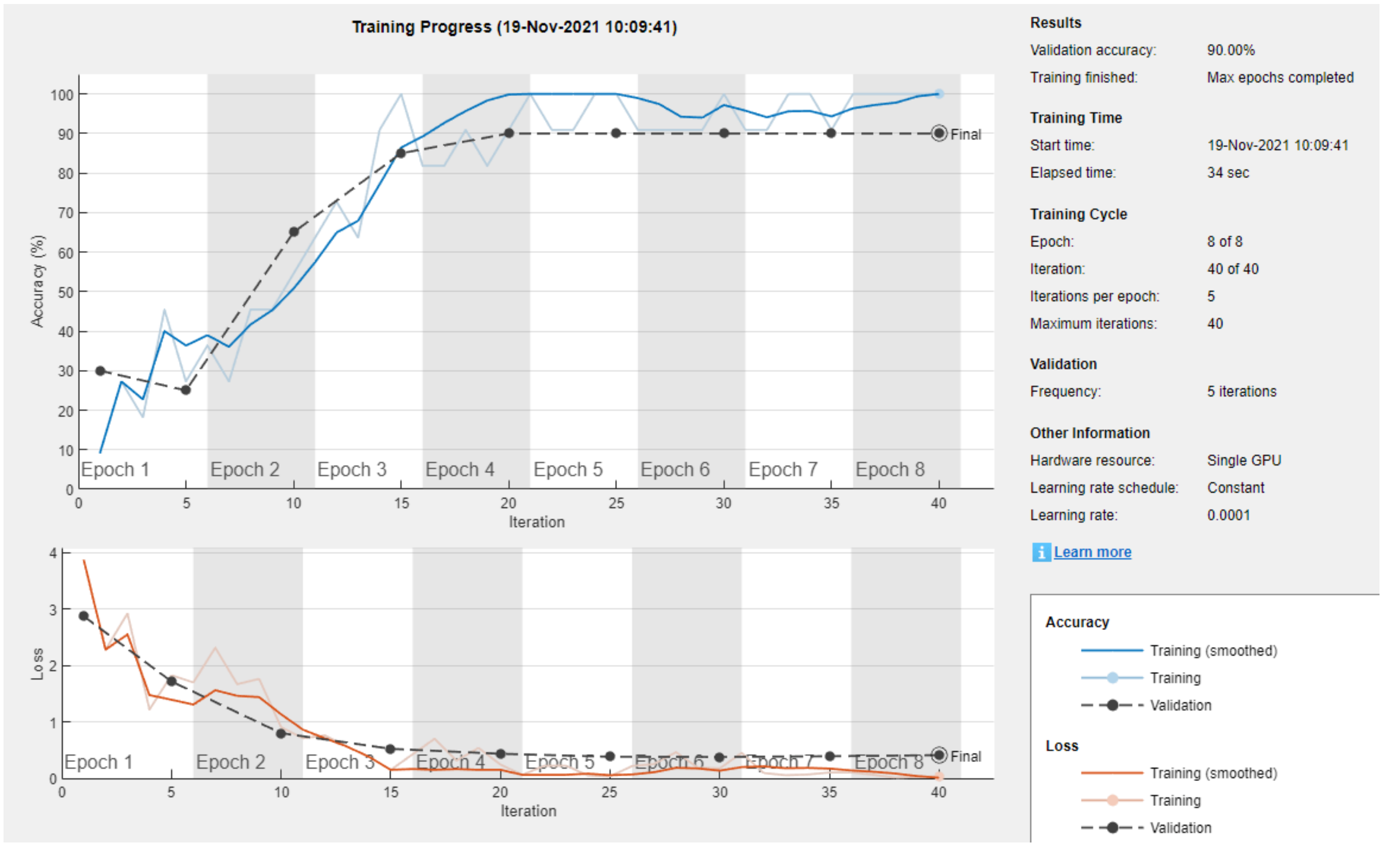
Solver	sgdm
InitialLearnRate	0.0001

BASIC

ValidationFrequency	5
MaxEpochs	8
MiniBatchSize	11
ExecutionEnvironment	auto

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

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



Results

Validation accuracy:	90.00%
Training finished:	Max epochs completed

Training Time

Start time:	19-Nov-2021 10:09:41
Elapsed time:	34 sec

Training Cycle

Epoch:	8 of 8
Iteration:	40 of 40
Iterations per epoch:	5
Maximum iterations:	40

Validation

Frequency:	5 iterations

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.0001

Export Results from Training

To export the results from training, 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`.

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**.

Test Trained Network

Select a new image to classify using the trained network.

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

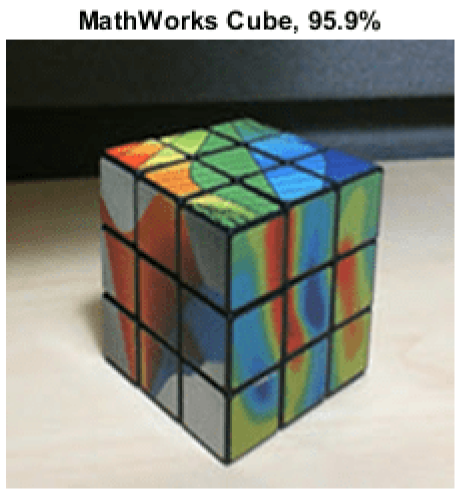
Resize the test image to match the network input size.

```
I = imresize(I, [224 224]);
```

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) + "%");
```



For more information, including on other pretrained networks, see Deep Network Designer.

See Also
Deep Network Designer

More About
- “Create Simple Image Classification Network Using Deep Network Designer” on page 1-29
- “Build Networks with Deep Network Designer”
- “Deep Learning Tips and Tricks”
- “List of Deep Learning Layers”

Try Deep Learning in 10 Lines of MATLAB Code

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

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

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

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

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

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

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

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

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

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

See Also
`trainNetwork` | `trainingOptions` | `alexnet`

More About
- “Classify Webcam Images Using Deep Learning”
- “Classify Image Using Pretrained Network” on page 1-14
- “Get Started with Transfer Learning” on page 1-16
- “Transfer Learning with Deep Network Designer”
- “Create Simple Image Classification Network” on page 1-26
- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 1-34

Classify Image Using Pretrained Network

This example shows how to classify an image using the pretrained deep convolutional neural network GoogLeNet.

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

Load Pretrained Network

Load the pretrained GoogLeNet network. You can also choose to load a different pretrained network for image classification. This step requires the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package. If you do not have the required support packages installed, then the software provides a download link.

```
net = googlenet;
```

Read and Resize Image

The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the network input size is the `InputSize` property of the image input layer.

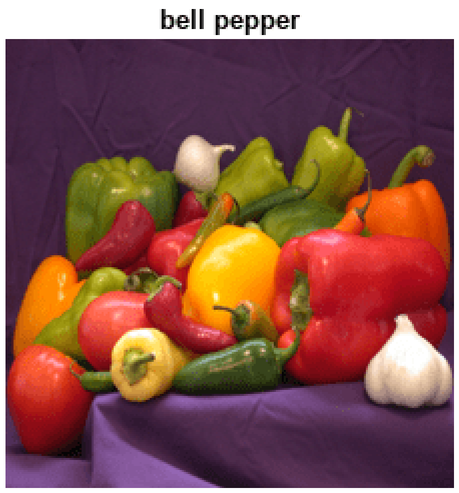
Read the image that you want to classify and resize it to the input size of the network. This resizing slightly changes the aspect ratio of the image.

```
I = imread("peppers.png");
inputSize = net.Layers(1).InputSize;
I = imresize(I,inputSize(1:2));
```

Classify and Display Image

Classify and display the image with the predicted label.

```
label = classify(net,I);
figure
imshow(I)
title(string(label))
```



For a more detailed example showing how to also display the top predictions with their associated probabilities, see “Classify Image Using GoogLeNet”.

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

References

1 Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

2 *BVLC GoogLeNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

See Also

`googlenet` | `classify` | **Deep Network Designer**

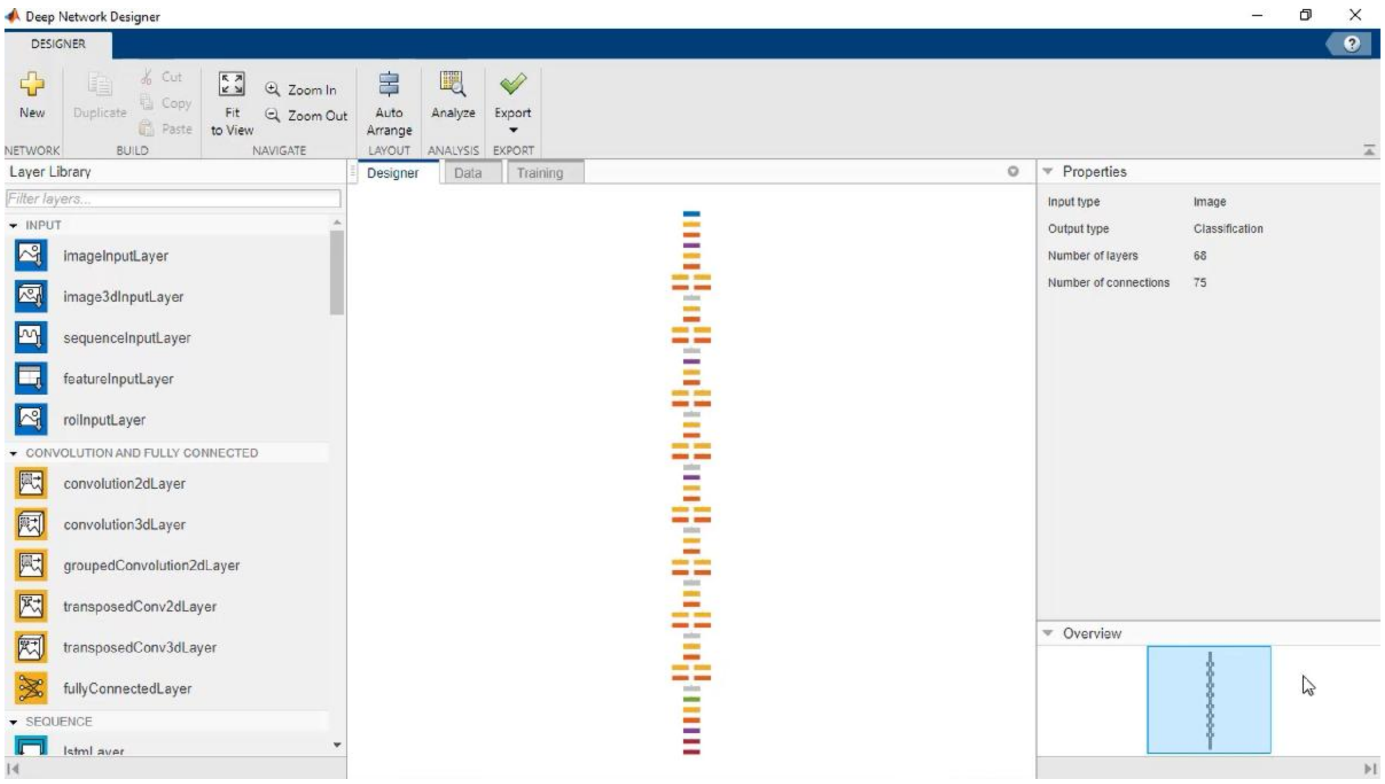
More About

- “Classify Image Using GoogLeNet”
- “Try Deep Learning in 10 Lines of MATLAB Code” on page 1-12
- “Get Started with Transfer Learning” on page 1-16
- “Create Simple Image Classification Network Using Deep Network Designer” on page 1-29
- “Transfer Learning with Deep Network Designer”
- “Create Simple Image Classification Network” on page 1-26
- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 1-34

Get Started with Transfer Learning

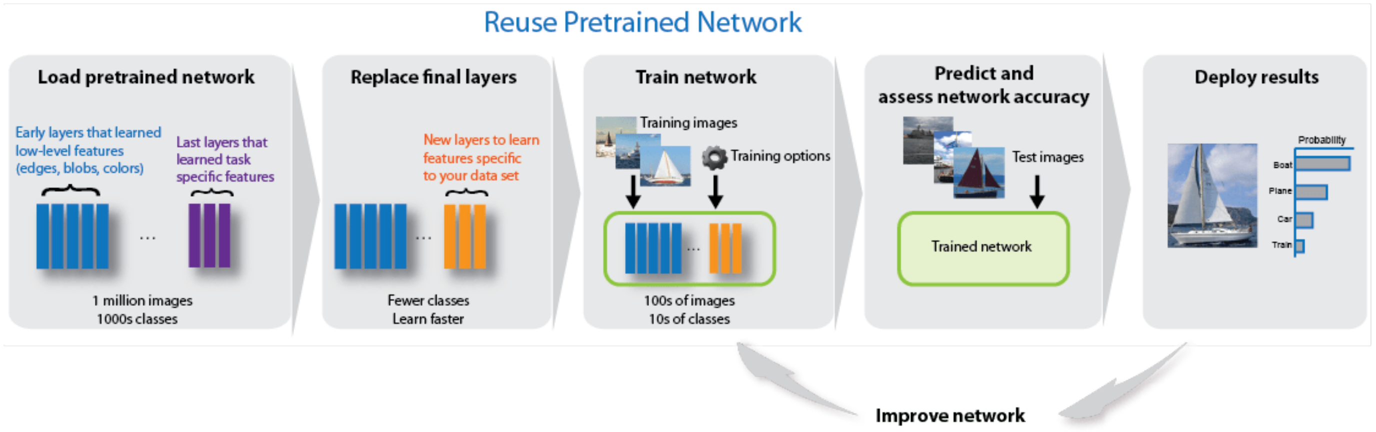
This example shows how to use transfer learning to retrain SqueezeNet, a pretrained convolutional neural network, to classify a new set of images. Try this example to see how simple it is to get started with deep learning in MATLAB®.

For a visual walkthrough of the example, watch the video.



Input type Image
Output type Classification
Number of layers 68
Number of connections 75

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.



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*).

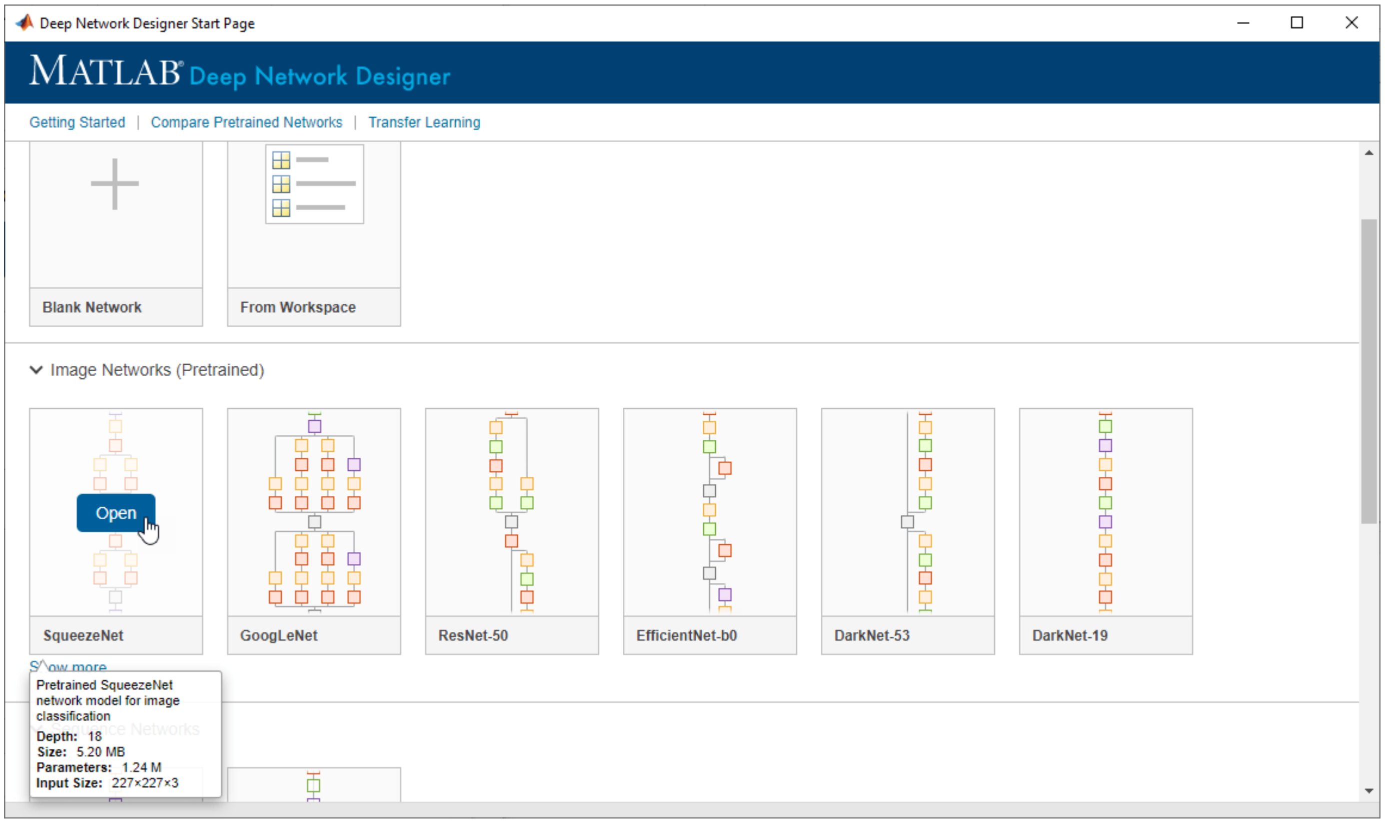
```
unzip("MerchData.zip");
```

Load Pretrained Network

Open Deep Network Designer.

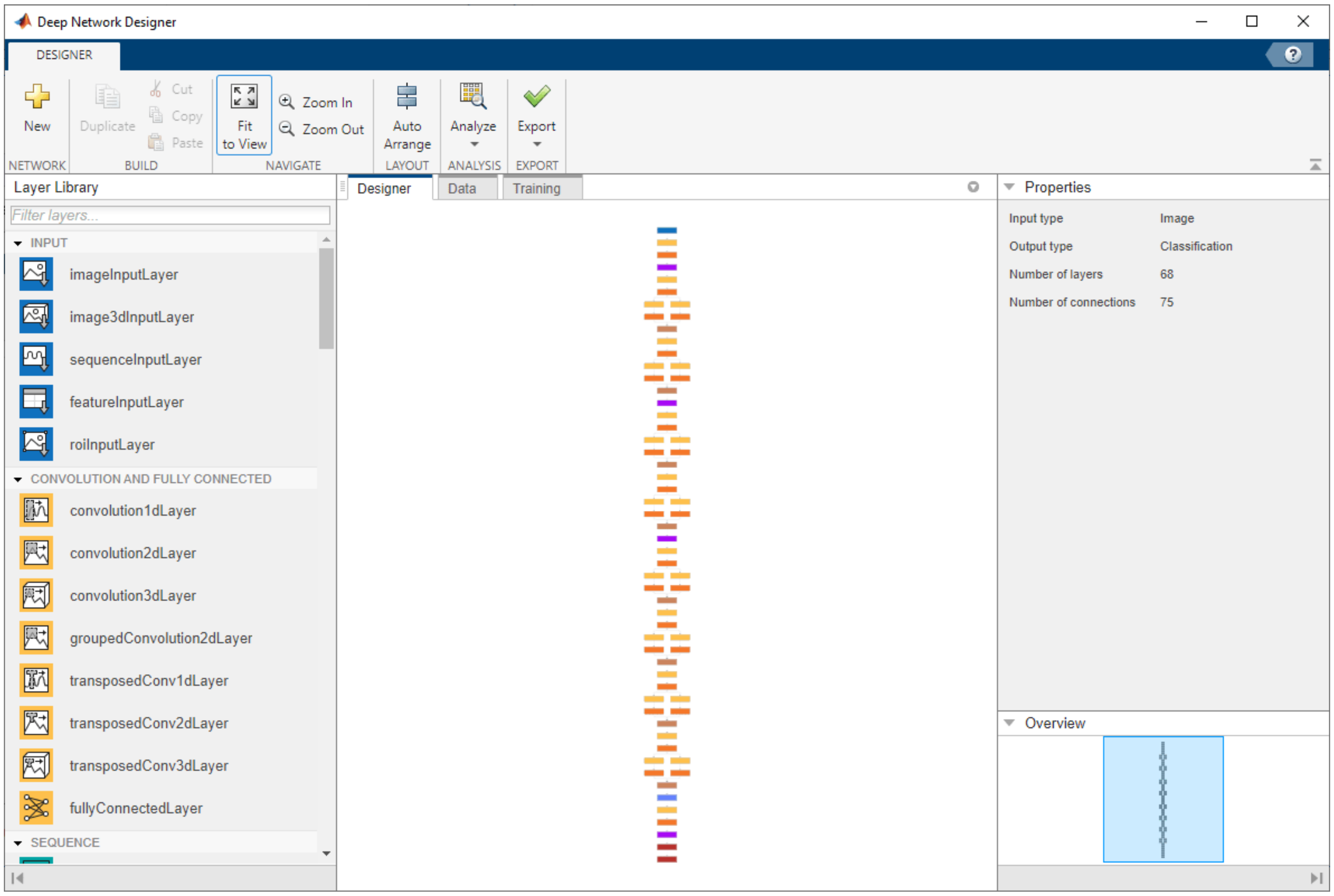
```
deepNetworkDesigner
```

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


Deep Network Designer Start Page
MATLAB Deep Network Designer
Getting Started | Compare Pretrained Networks | Transfer Learning
Blank Network
From Workspace
Image Networks (Pretrained)
Open
SqueezeNet
GoogLeNet
ResNet-50
EfficientNet-b0
DarkNet-53
DarkNet-19
Show more

Pretrained SqueezeNet network model for image classification
Depth: 18
Size: 5.20 MB
Parameters: 1.24 M
Input Size: 227×227×3

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



Input type	Image
Output type	Classification
Number of layers	68
Number of connections	75

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

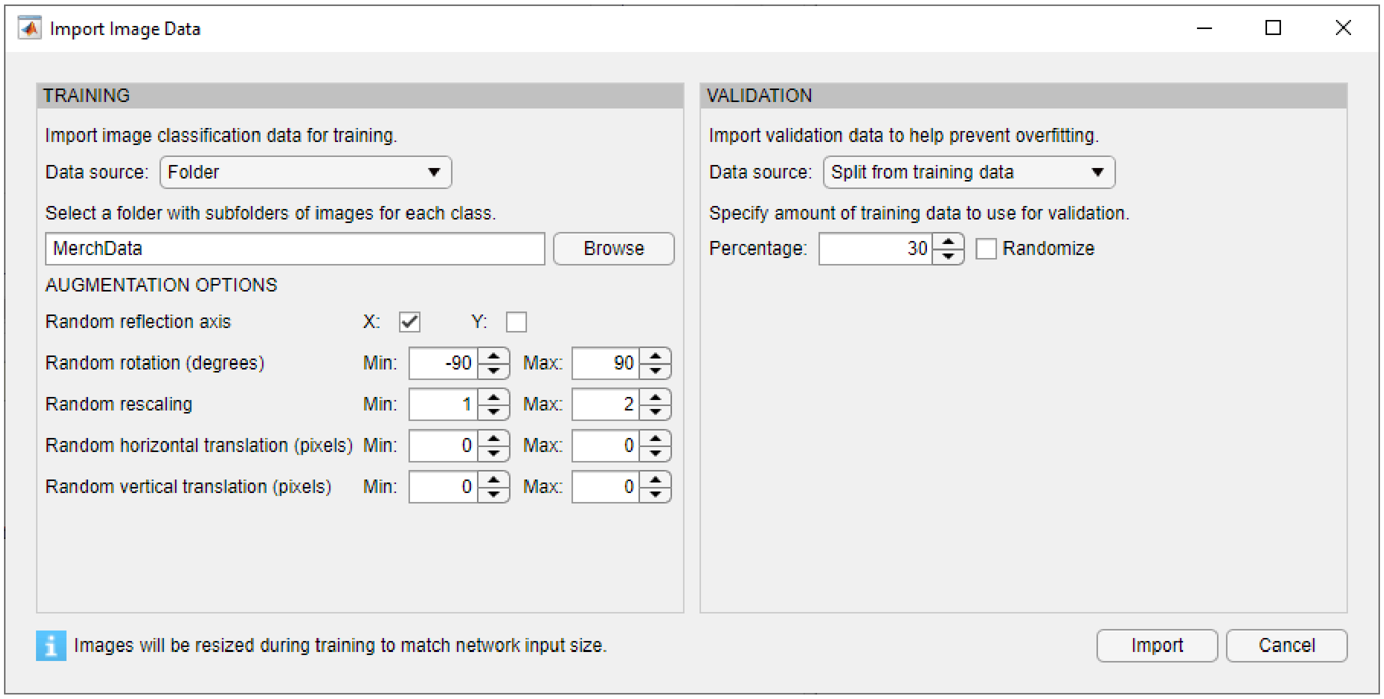
Import Data

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

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

Divide the data into 70% training data and 30% validation data.

Specify augmentation operations to perform on the training images. Data augmentation helps prevent the network from overfitting and memorizing the exact details of 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].



Import Image Data

TRAINING

Import image classification data for training.

Data source: Folder

Select a folder with subfolders of images for each class.

MerchData

AUGMENTATION OPTIONS

Random reflection axis	X:	✓	Y:	
Random rotation (degrees)	Min:	-90	Max:	90
Random rescaling	Min:	1	Max:	2
Random horizontal translation (pixels)	Min:	0	Max:	0
Random vertical translation (pixels)	Min:	0	Max:	0

VALIDATION

Import validation data to help prevent overfitting.

Data source: Split from training data

Specify amount of training data to use for validation.

Percentage: 30 Randomize

Images will be resized during training to match network input size.

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

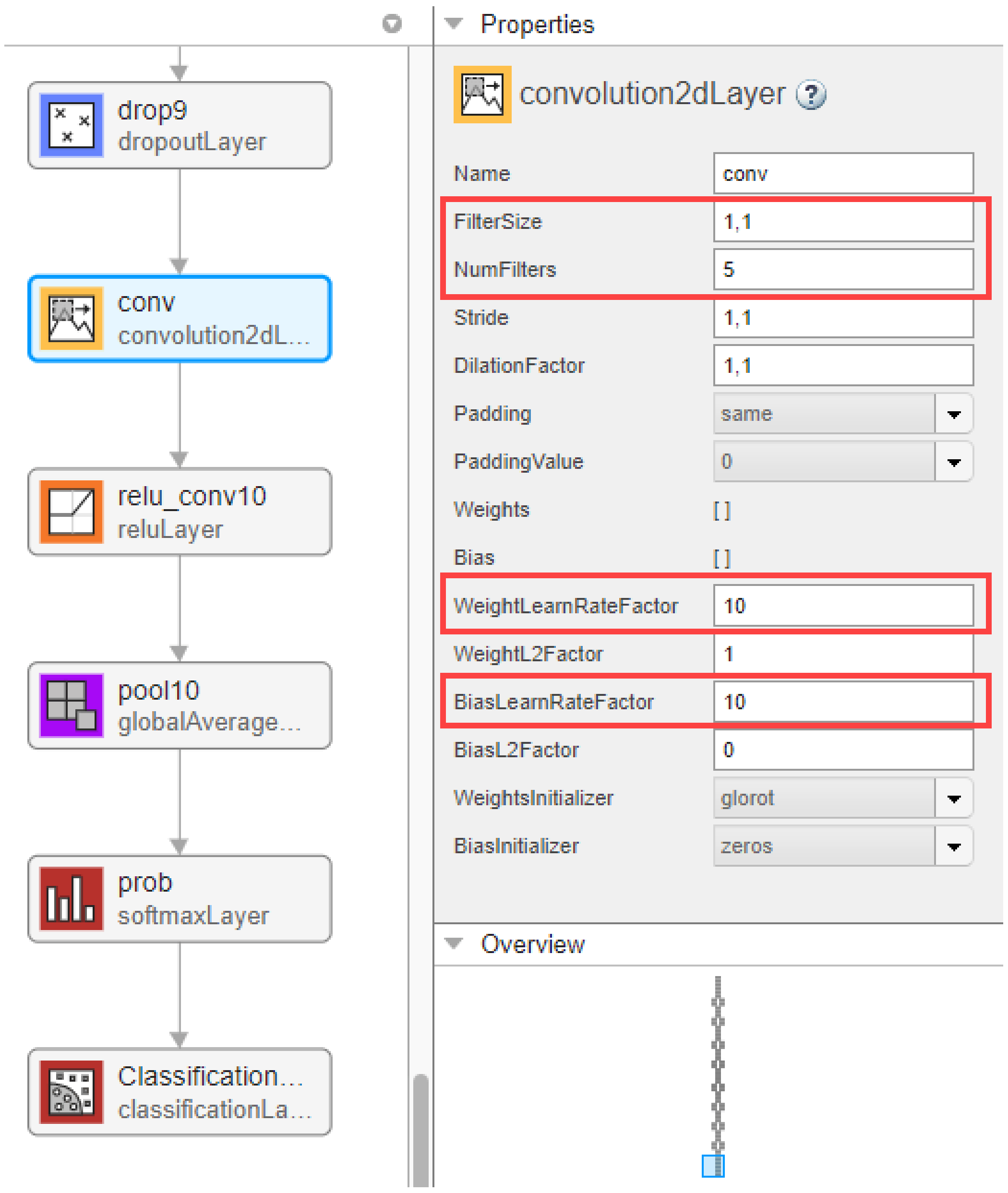
Edit Network for Transfer Learning

To retrain SqueezeNet to classify new images, replace the last 2-D convolutional layer and the final classification layer of the network. In SqueezeNet, these layers have the names `'conv10'` and `'ClassificationLayer_predictions'`, respectively.

On the **Designer** pane, drag a new `convolution2dLayer` onto the canvas. To match the original convolutional layer, set `FilterSize` to `1,1`. Edit `NumFilters` to be the number of classes in the new data, in this example, `5`.

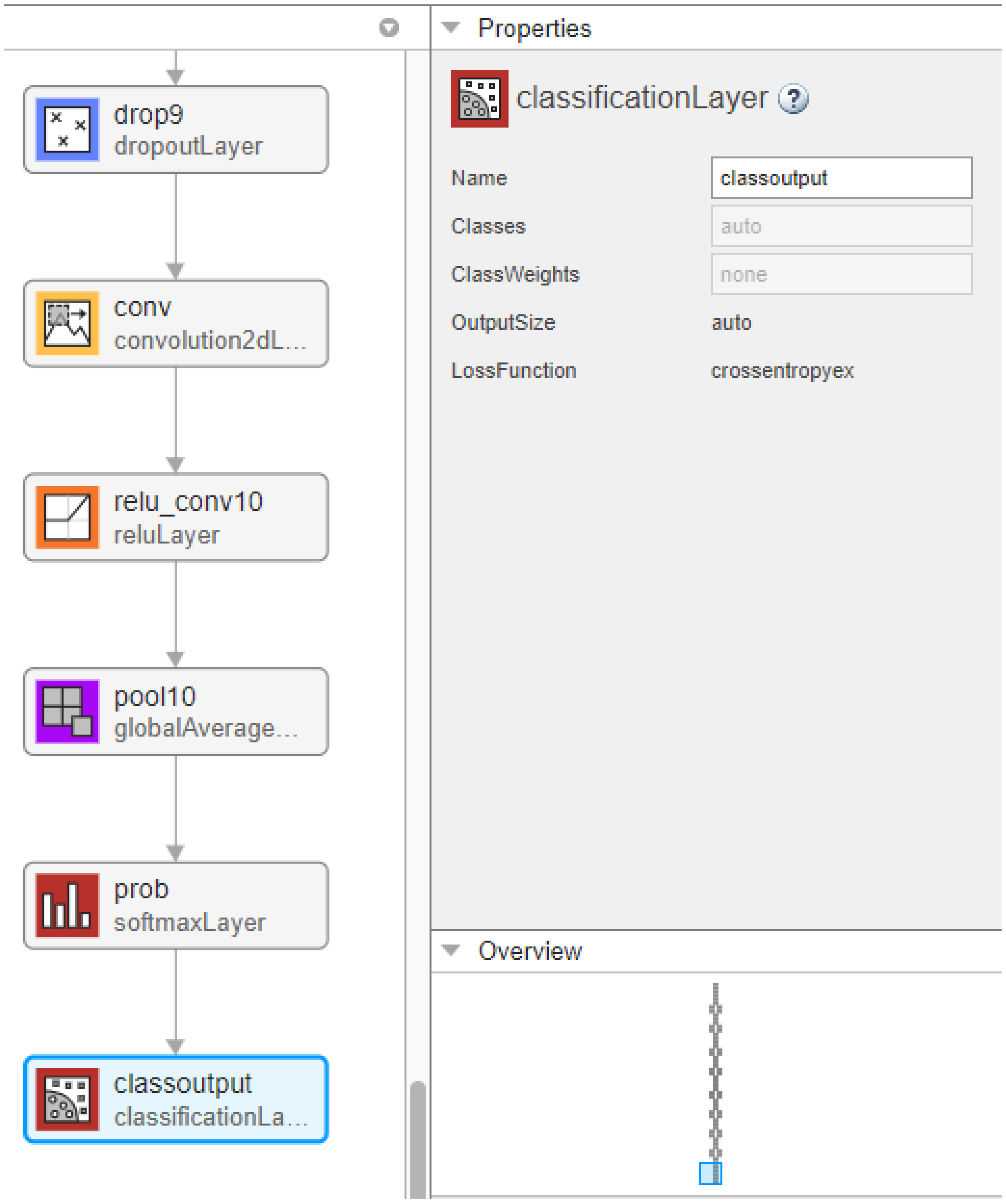
Change the learning rates so that learning is faster in the new layer than in the transferred layers by setting `WeightLearnRateFactor` and `BiasLearnRateFactor` to `10`.

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



Name	conv
FilterSize	1,1
NumFilters	5
Stride	1,1
DilationFactor	1,1
Padding	same
PaddingValue	0
Weights	[]
Bias	[]
WeightLearnRateFactor	10
WeightL2Factor	1
BiasLearnRateFactor	10
BiasL2Factor	0
WeightsInitializer	glorot
BiasInitializer	zeros

Replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original output layer and connect your new layer in its place.



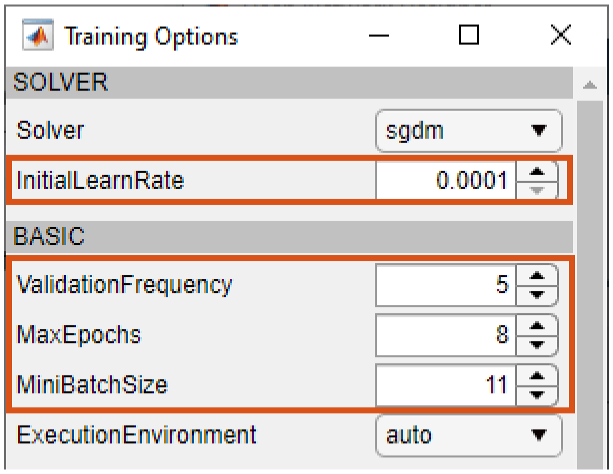
classificationLayer

Name	classoutput
Classes	auto
ClassWeights	none
OutputSize	auto
LossFunction	crossentropyex

Train Network

To choose the 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. In the previous step, you increased the learning rate factors for the 2-D 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.

For this example, set **InitialLearnRate** to `0.0001`, **ValidationFrequency** to `5`, **MaxEpochs** to `8`. As there are 55 observations, set **MiniBatchSize** to `11` to divide the training data evenly and ensure the whole training set is used during each epoch.



SOLVER

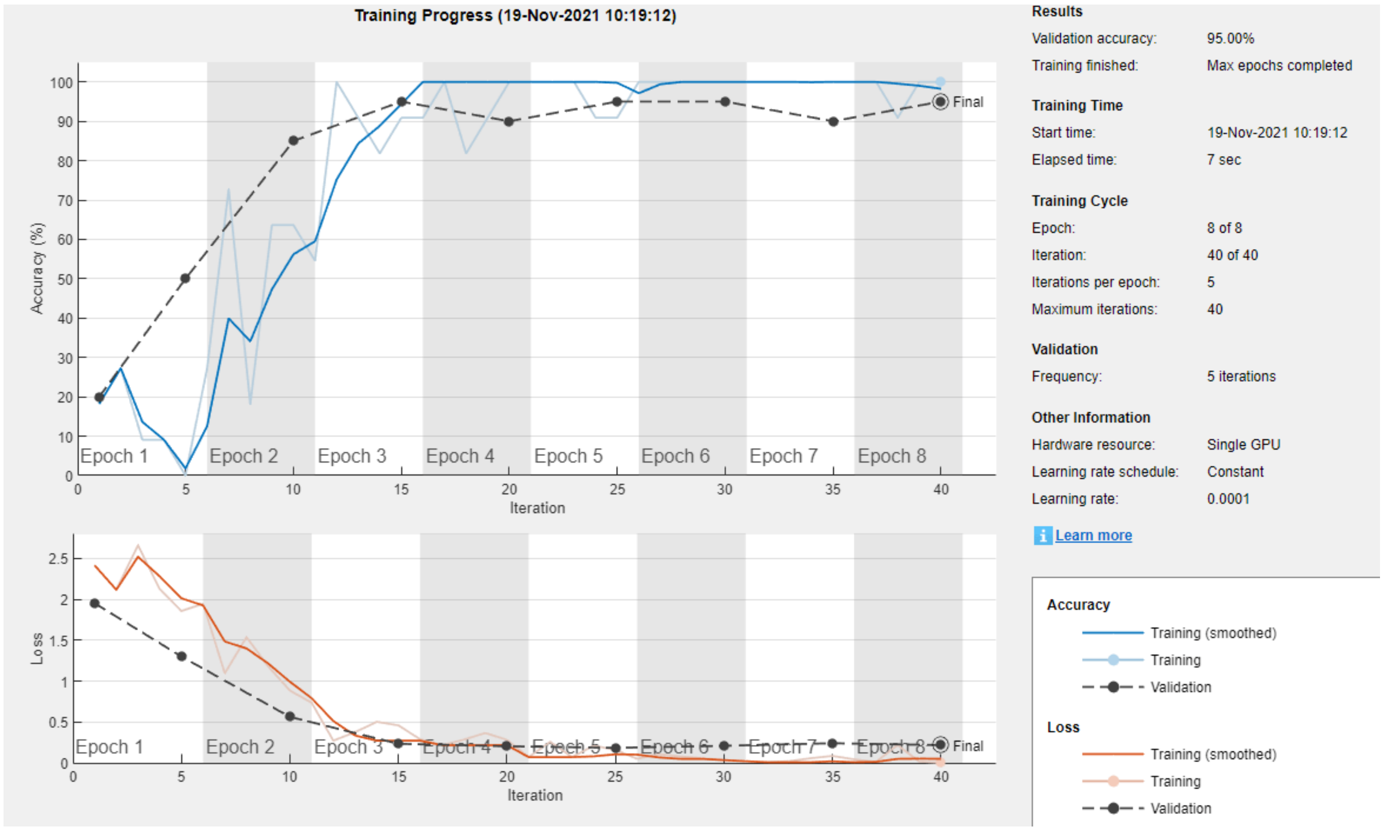
Solver	sgdm
InitialLearnRate	0.0001

BASIC

ValidationFrequency	5
MaxEpochs	8
MiniBatchSize	11
ExecutionEnvironment	auto

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

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



Results

Validation accuracy:	95.00%
Training finished:	Max epochs completed

Training Time

Start time:	19-Nov-2021 10:19:12
Elapsed time:	7 sec

Training Cycle

Epoch:	8 of 8
Iteration:	40 of 40
Iterations per epoch:	5
Maximum iterations:	40

Validation

Frequency:	5 iterations

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.0001

Export Results and Generate MATLAB Code

To export the results from training, 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`.

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.

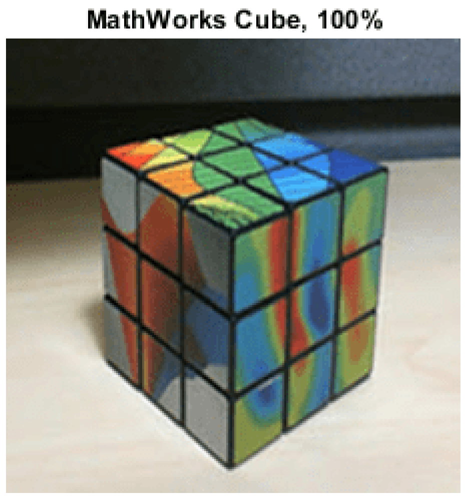
```
I = imread("MerchDataTest.jpg");
```

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) + "%");
```



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.

See Also

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

More About

- "Try Deep Learning in 10 Lines of MATLAB Code" on page 1-12
- "Classify Image Using Pretrained Network" on page 1-14
- "Transfer Learning with Deep Network Designer"
- "Create Simple Image Classification Network Using Deep Network Designer" on page 1-29
- "Create Simple Image Classification Network" on page 1-26
- "Create Simple Sequence Classification Network Using Deep Network Designer"
- "Generate Experiment Using Deep Network Designer"

Create Simple Image Classification Network

This example shows how to create and train a simple convolutional neural network for deep learning classification. Convolutional neural networks are essential tools for deep learning and are especially suited for image recognition.

The example demonstrates how to:

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

For an example showing how to interactively create and train a simple image classification network, see "Create Simple Image Classification Network Using Deep Network Designer" on page 1-29.

Load Data

Load the digit sample data as an image datastore. The `imageDatastore` function automatically labels the images based on folder names.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');

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

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label. `splitEachLabel` splits the image datastore into two new datastores for training and validation.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomized');
```

Define Network Architecture

Define the convolutional neural network architecture. Specify the size of the images in the input layer of the network and the number of classes in the fully connected layer before the classification layer. Each image is 28-by-28-by-1 pixels and there are 10 classes.

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

layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

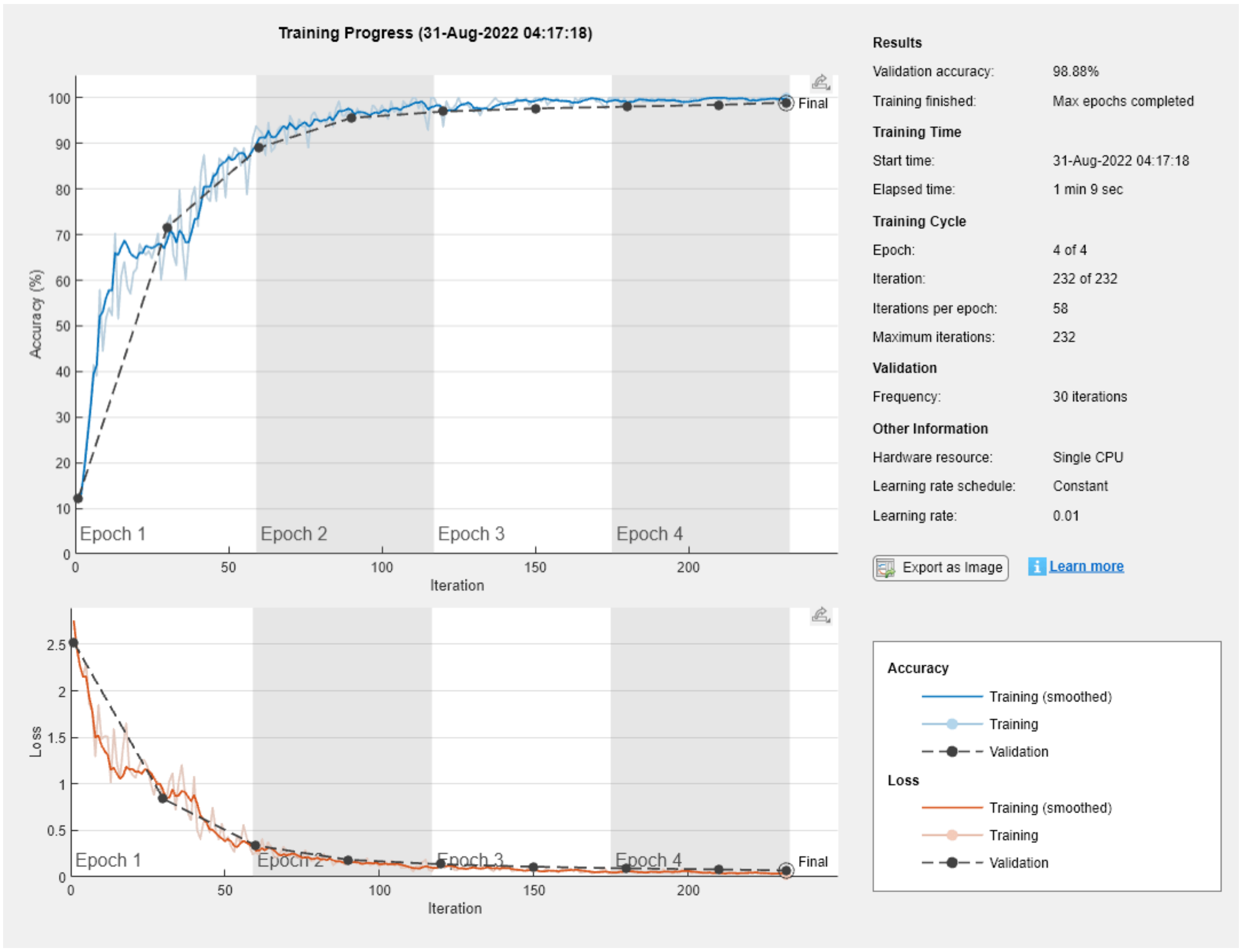
For more information about deep learning layers, see “List of Deep Learning Layers”.

Train Network

Specify the training options and train the network.

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`.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',4, ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');

net = trainNetwork(imdsTrain,layers,options);
```



Results

Validation accuracy:	98.88%
Training finished:	Max epochs completed
Training Time	
Start time:	31-Aug-2022 04:17:18
Elapsed time:	1 min 9 sec
Training Cycle	
Epoch:	4 of 4
Iteration:	232 of 232
Iterations per epoch:	58
Maximum iterations:	232
Validation	
Frequency:	30 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.01

For more information about training options, see “Set Up Parameters and Train Convolutional Neural Network”.

Test Network

Classify the validation data and calculate the classification accuracy.

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

```
accuracy = 0.9888
```

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

See Also
`trainNetwork` | `trainingOptions`

More About
- “Start Deep Learning Faster Using Transfer Learning”
- “Create Simple Image Classification Network Using Deep Network Designer” on page 1-29
- “Try Deep Learning in 10 Lines of MATLAB Code” on page 1-12
- “Classify Image Using Pretrained Network” on page 1-14
- “Get Started with Transfer Learning” on page 1-16
- “Transfer Learning with Deep Network Designer”
- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 1-34

Create Simple Image Classification Network Using Deep Network Designer

This example shows how to create and train a simple convolutional neural network for deep learning classification using Deep Network Designer. Convolutional neural networks are essential tools for deep learning and are especially suited for image recognition.

In this example, you:

- Import image data.
- Define the network architecture.
- Specify training options.
- Train the network.

Load Data

Load the digit sample data as an image datastore. The `imageDatastore` function automatically labels the images based on folder names. The data set has 10 classes and each image in the data set is 28-by-28-by-1 pixels.

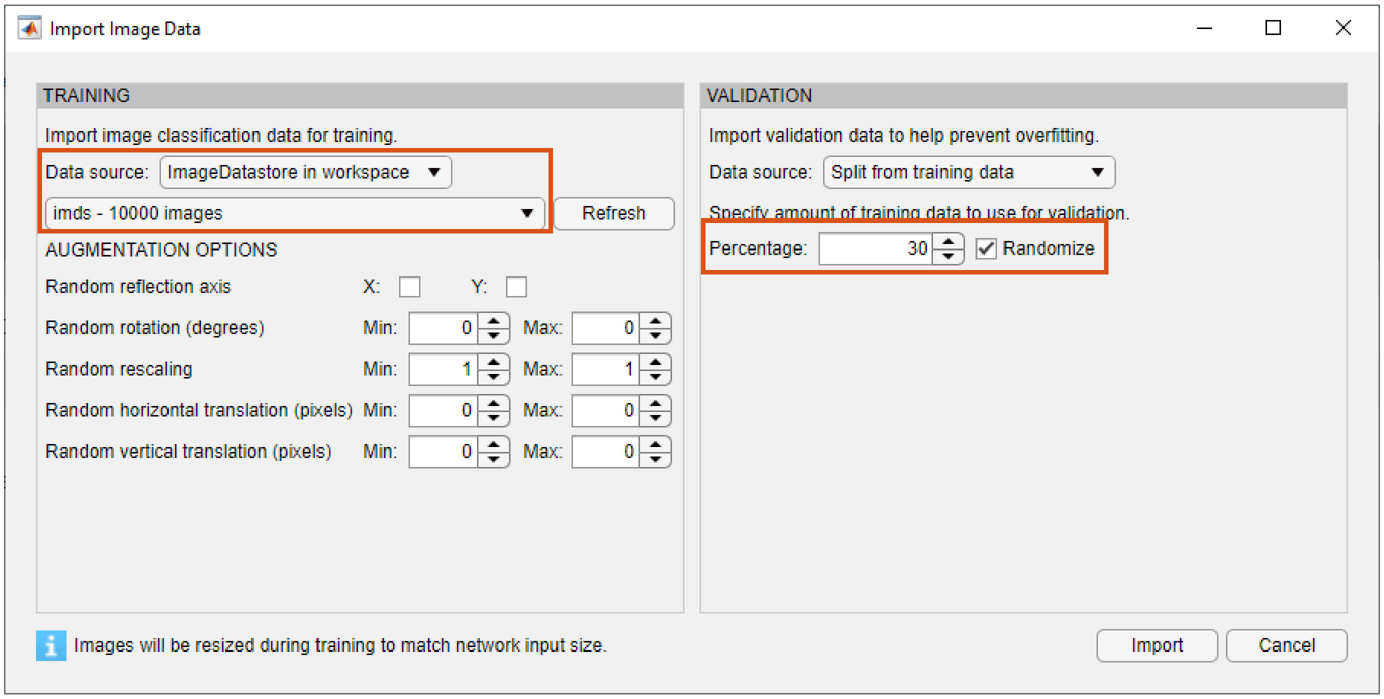
```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');

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

Open Deep Network Designer. Create a network, import and visualize data, and train the network using Deep Network Designer.

```
deepNetworkDesigner
```

To create a blank network, pause on **Blank Network** and click **New**.

To import the image datastore, select the **Data** tab and click **Import Data** > **Import Image Data**. Select `imds` as the data source. Set aside 30% of the training data to use as validation data. Randomly allocate observations to the training and validation sets by selecting **Randomize**.


Import Image Data

TRAINING

Import image classification data for training.

Data source: ImageDatastore in workspace

imds - 10000 images

Refresh

AUGMENTATION OPTIONS

Random reflection axis	X: ☐		Y: ☐	
Random rotation (degrees)	Min:	0	Max:	0
Random rescaling	Min:	1	Max:	1
Random horizontal translation (pixels)	Min:	0	Max:	0
Random vertical translation (pixels)	Min:	0	Max:	0

VALIDATION

Import validation data to help prevent overfitting.

Data source: Split from training data

Specify amount of training data to use for validation.

Percentage: 30 ☑ Randomize

Images will be resized during training to match network input size.

Import Cancel

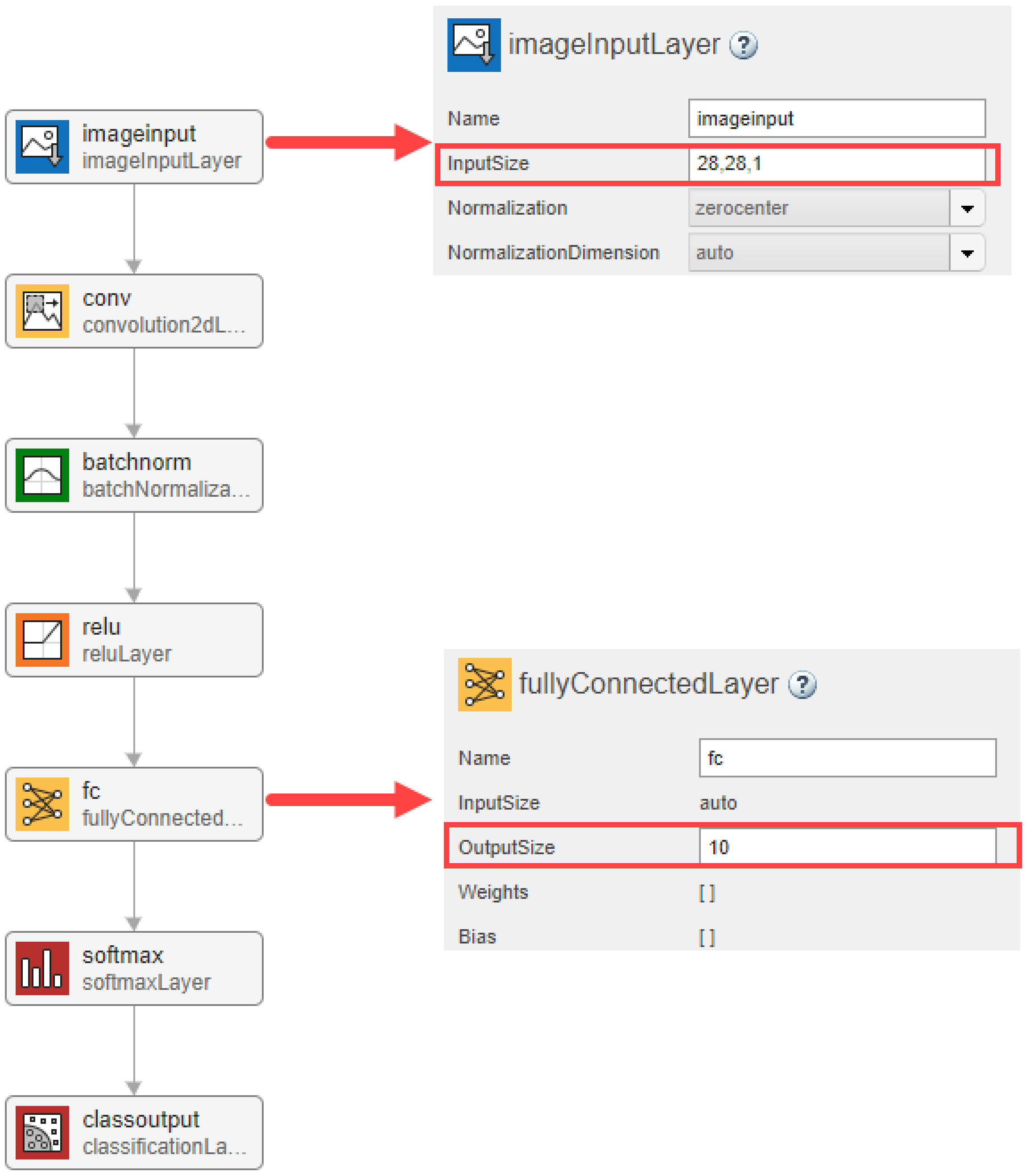
Import the data by clicking **Import**.

Define Network Architecture

In the **Designer** pane, define the convolutional neural network architecture. Drag layers from the **Layer Library** and connect them. To quickly search for layers, use the **Filter layers** search box in the **Layer Library** pane. To edit the properties of a layer, click the layer and edit the values in the **Properties** pane.

Connect layers in this order:

1. `imageInputLayer` with the `InputSize` property set to `28,28,1`
2. `convolution2dLayer`
3. `batchNormalizationLayer`
4. `reluLayer`
5. `fullyConnectedLayer` with the `OutputSize` property set to `10`
6. `softmaxLayer`
7. `classificationLayer`



imageInputLayer

Name	imageinput
InputSize	28,28,1
Normalization	zerocenter
NormalizationDimension	auto

fullyConnectedLayer

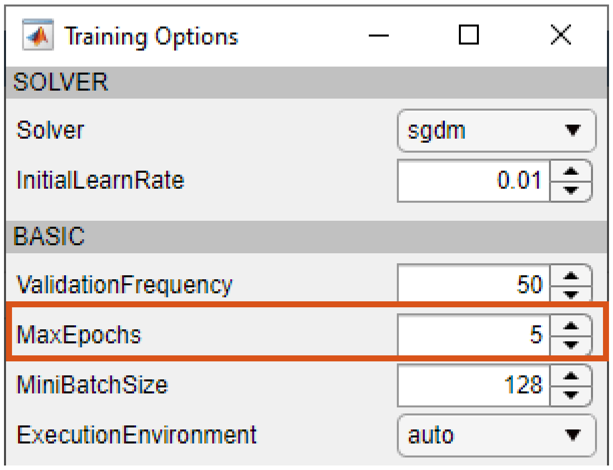
Name	fc
InputSize	auto
OutputSize	10
Weights	[]
Bias	[]

For more information about deep learning layers, see “List of Deep Learning Layers”.

Train Network

Specify the training options and train the network.

On the **Training** tab, click **Training Options**. For this example, set the maximum number of epochs to 5 and keep the other default settings. Set the training options by clicking **Close**. For more information about training options, see “Set Up Parameters and Train Convolutional Neural Network”.



Training Options

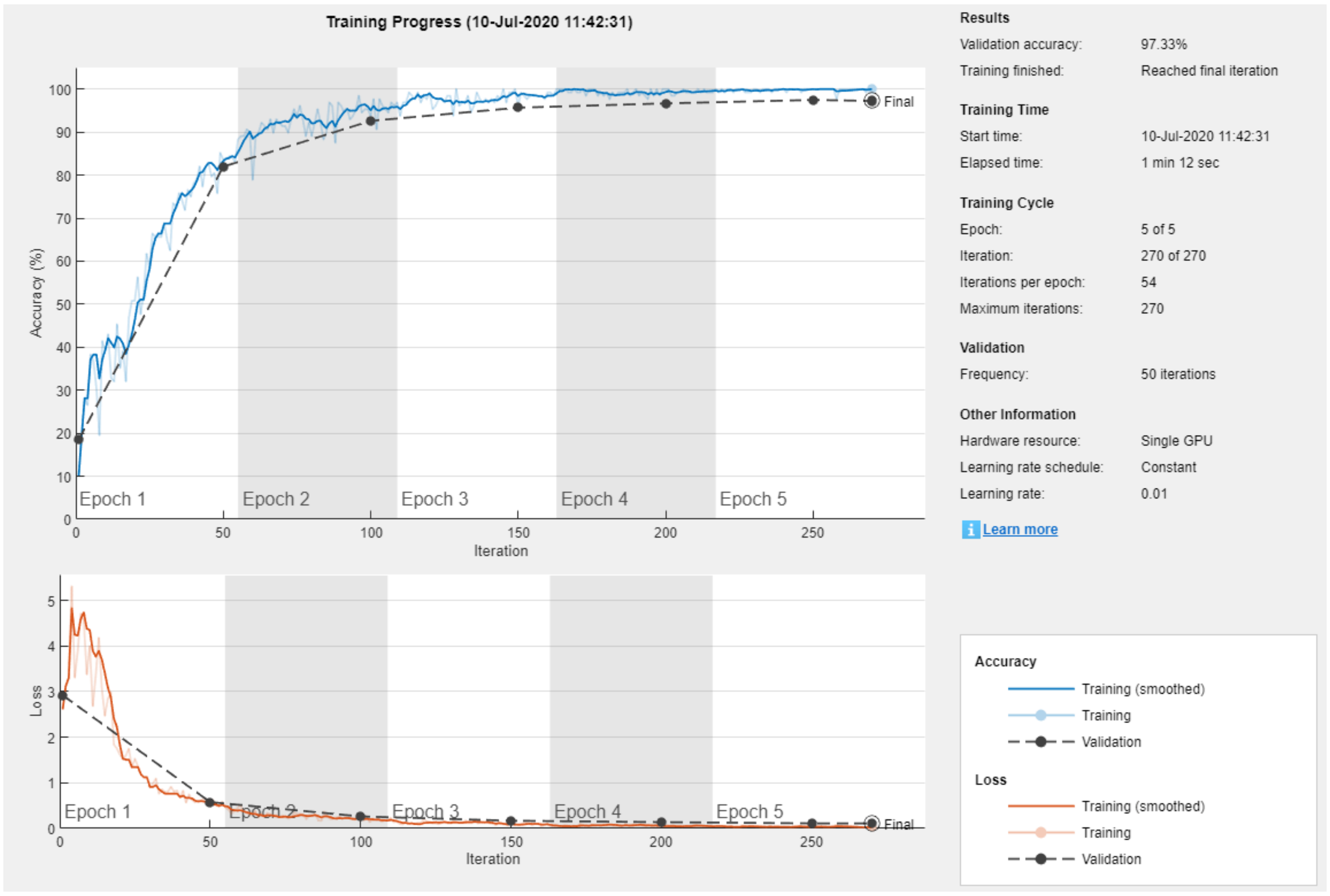
SOLVER

Solver	sgdm
InitialLearnRate	0.01

BASIC

ValidationFrequency	50
MaxEpochs	5
MiniBatchSize	128
ExecutionEnvironment	auto

Train the network by clicking **Train**.



Results

Validation accuracy:	97.33%
Training finished:	Reached final iteration

Training Time

Start time:	10-Jul-2020 11:42:31
Elapsed time:	1 min 12 sec

Training Cycle

Epoch:	5 of 5
Iteration:	270 of 270
Iterations per epoch:	54
Maximum iterations:	270

Validation

Frequency:	50 iterations

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.01

Accuracy
- Training (smoothed)
- Training
- Validation

Loss
- Training (smoothed)
- Training
- Validation

The accuracy is the fraction of labels that the network predicts correctly. In this case, more than 97% of the predicted labels match the true labels of the validation set.

To export the trained network to the workspace, on the **Training** tab, click **Export**.

For next steps in deep learning, you can try using pretrained networks for other tasks. Solve new classification problems on your image data with transfer learning. For example, see “Get Started with Transfer Learning” on page 1-16. To learn more about pretrained networks, see “Pretrained Deep Neural Networks”.

See Also

`trainingOptions` | **Deep Network Designer**

More About

- “Create Simple Image Classification Network” on page 1-26
- “Start Deep Learning Faster Using Transfer Learning”
- “Try Deep Learning in 10 Lines of MATLAB Code” on page 1-12
- “Classify Image Using Pretrained Network” on page 1-14
- “Get Started with Transfer Learning” on page 1-16
- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 1-34

Create Simple Sequence Classification Network Using Deep Network Designer

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

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

The example demonstrates how to:

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

Load Data

Load the Japanese Vowels data set, as described in [1] on page 1-39 and [2] on page 1-39. The predictors are cell arrays containing sequences of varying length with a feature dimension of 12. The labels are categorical vectors of labels 1,2,...,9.

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

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

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

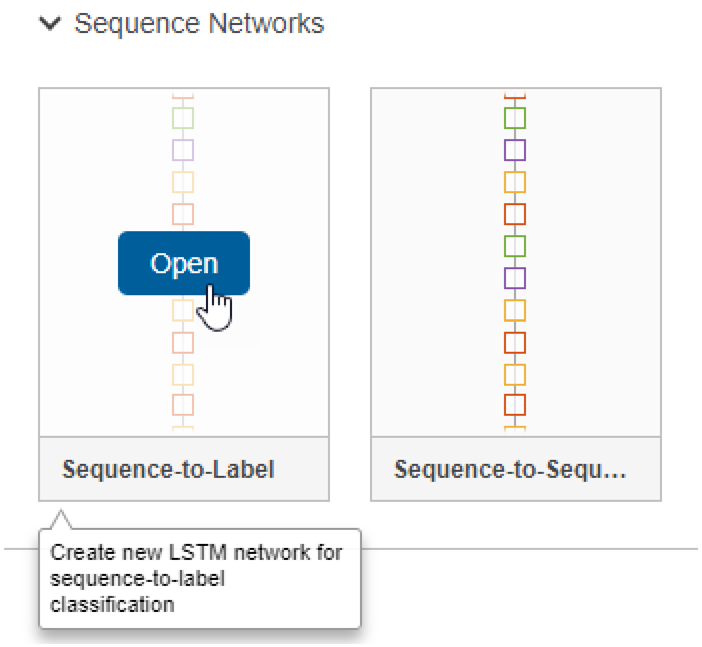
```
ans=5×1 cell array
    {12×20 double}
    {12×26 double}
    {12×22 double}
    {12×20 double}
    {12×21 double}
```

Define Network Architecture

Open Deep Network Designer.

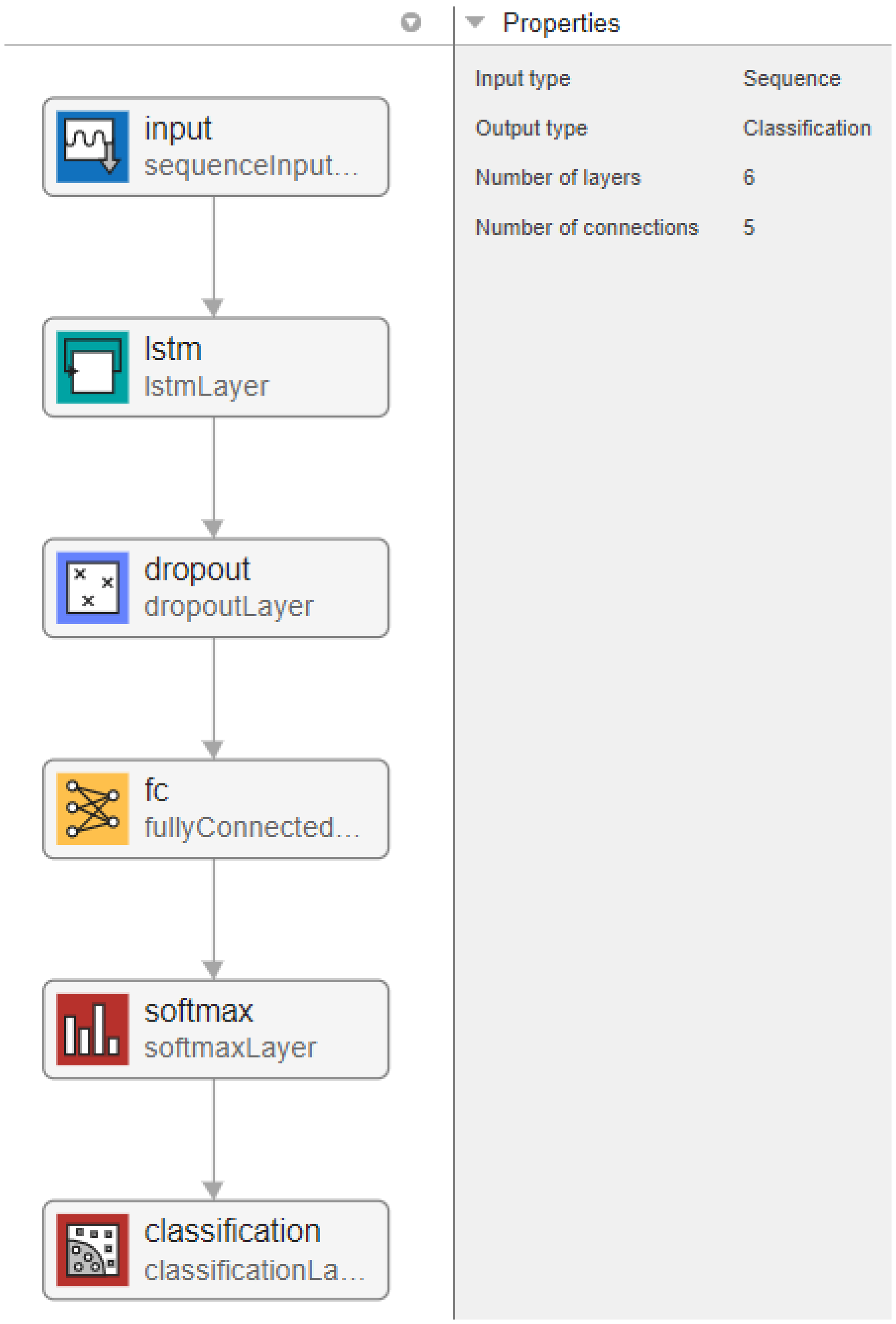
```
deepNetworkDesigner
```

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



Create new LSTM network for sequence-to-label classification

Deep Network Designer displays the prebuilt network.

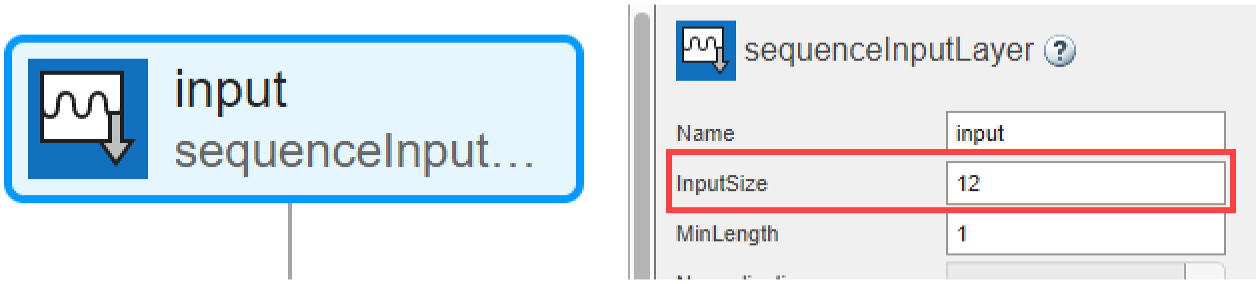


Properties

Input type	Sequence
Output type	Classification
Number of layers	6
Number of connections	5

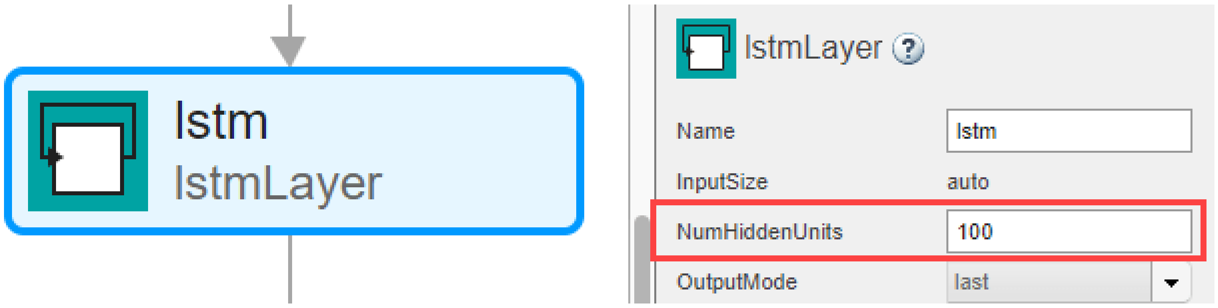
You can easily adapt this sequence network for the Japanese Vowels data set.

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



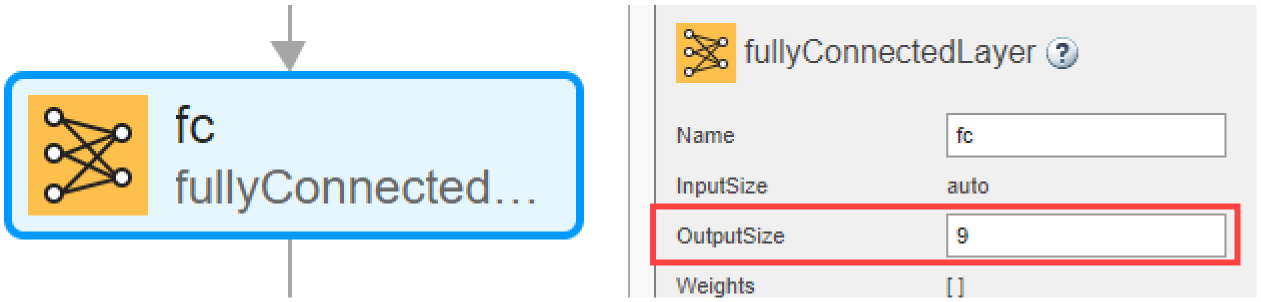
Name	input
InputSize	12
MinLength	1

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



Name	lstm
InputSize	auto
NumHiddenUnits	100
OutputMode	last

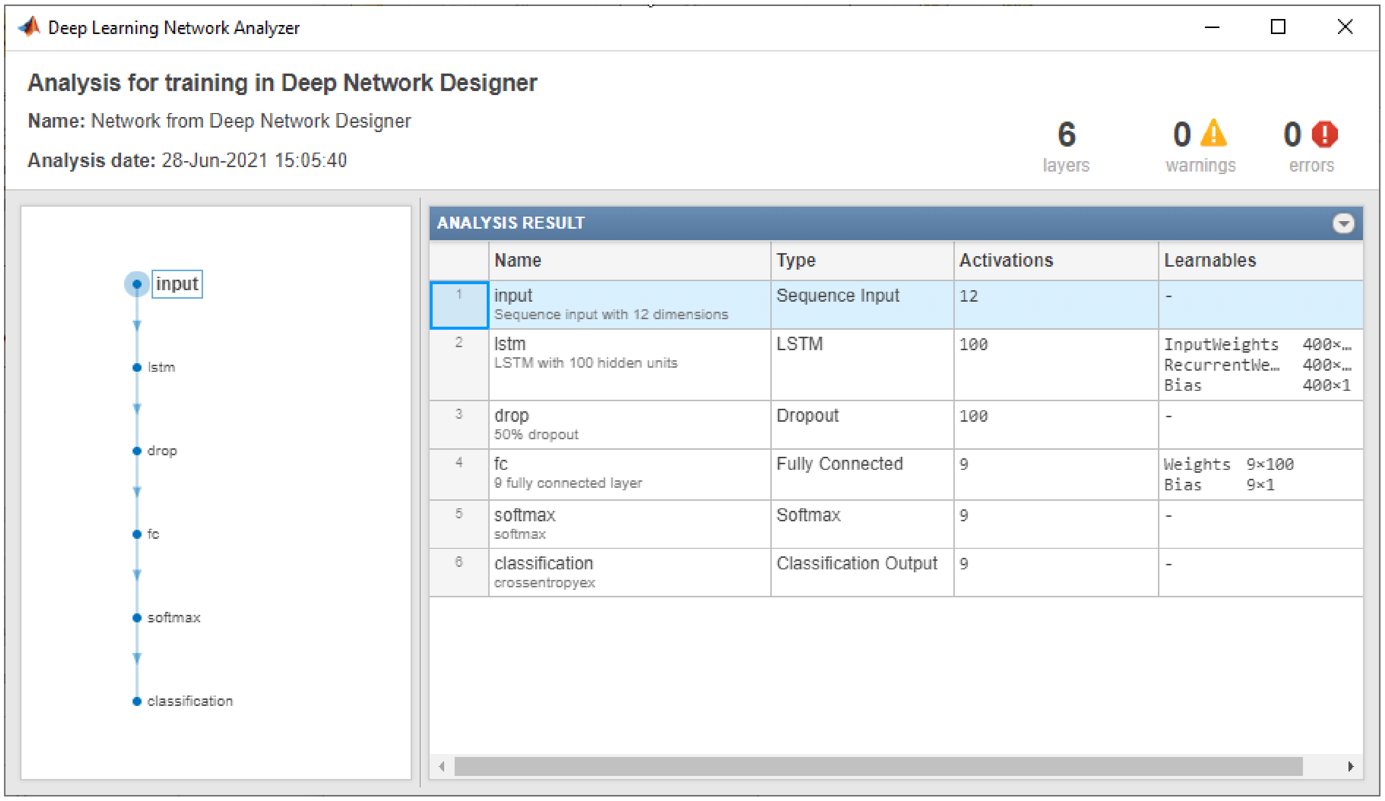
Select **fullyConnectedLayer** and check that **OutputSize** is set to 9, the number of classes.



Name	fc
InputSize	auto
OutputSize	9
Weights	[]

Check Network Architecture

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


Deep Learning Network Analyzer

Analysis for training in Deep Network Designer

Name: Network from Deep Network Designer

Analysis date: 28-Jun-2021 15:05:40

6 layers, 0 warnings, 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	input Sequence input with 12 dimensions	Sequence Input	12	-
2	lstm LSTM with 100 hidden units	LSTM	100	InputWeights 400×… RecurrentWe… 400×… Bias 400×1
3	drop 50% dropout	Dropout	100	-
4	fc 9 fully connected layer	Fully Connected	9	Weights 9×100 Bias 9×1
5	softmax softmax	Softmax	9	-
6	classification crossentropyex	Classification Output	9	-

Export Network Architecture

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

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

Train Network

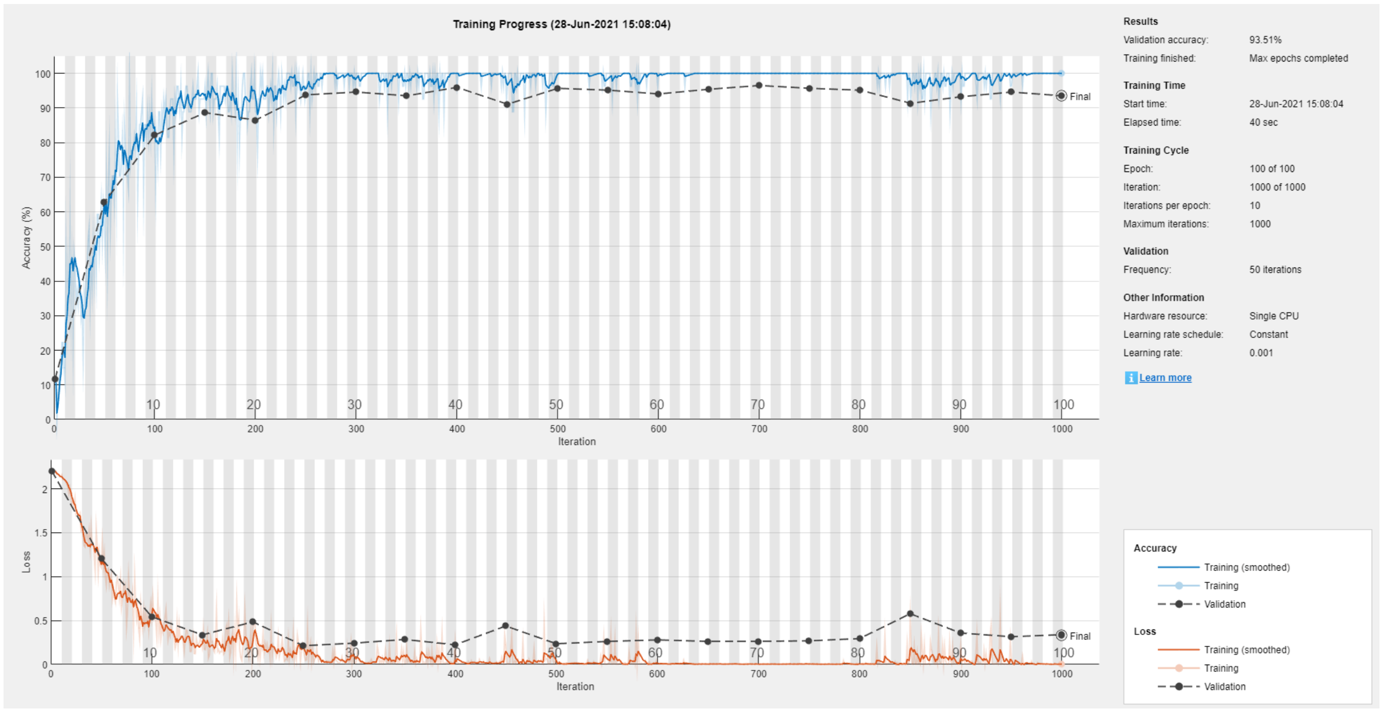
Specify the training options and train the network.

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

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

Train the network.

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



Results
Validation accuracy: 93.51%
Training finished: Max epochs completed

Training Time
Start time: 28-Jun-2021 15:08:04
Elapsed time: 40 sec

Training Cycle
Epoch: 100 of 100
Iteration: 1000 of 1000
Iterations per epoch: 10
Maximum iterations: 1000

Validation
Frequency: 50 iterations

Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.001

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

Test Network

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

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

```
acc = 0.9405
```

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

For an example showing how to use convolutional networks to classify sequence data, see “Train Speech Command Recognition Model Using Deep Learning”.

References

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

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

See Also

`trainingOptions` | `trainNetwork` | `lstmLayer`

More About

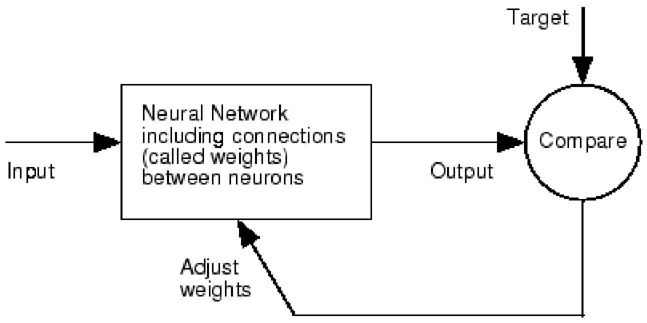
- "Long Short-Term Memory Networks"
- "Try Deep Learning in 10 Lines of MATLAB Code" on page 1-12
- "Classify Image Using Pretrained Network" on page 1-14
- "Get Started with Transfer Learning" on page 1-16
- "Transfer Learning with Deep Network Designer"
- "Create Simple Image Classification Network Using Deep Network Designer" on page 1-29

Shallow Networks for Pattern Recognition, Clustering and Time Series

In this section...
"Shallow Network Apps and Functions in Deep Learning Toolbox" on page 1-41
"Deep Learning Toolbox Applications" on page 1-42
"Shallow Neural Network Design Steps" on page 1-43

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the connections between elements largely determine the network function. You can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Typically, neural networks are adjusted, or trained, so that a particular input leads to a specific target output. The next figure illustrates such a situation. Here, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically, many such input/target pairs are needed to train a network.


Target
Neural Network including connections (called weights) between neurons
Input
Output
Compare
Adjust weights

Neural networks have been trained to perform complex functions in various fields, including pattern recognition, identification, classification, speech, vision, and control systems.

Neural networks can also be trained to solve problems that are difficult for conventional computers or human beings. The toolbox emphasizes the use of neural network paradigms that build up to—or are themselves used in— engineering, financial, and other practical applications.

The following topics explain how to use graphical tools for training neural networks to solve problems in function fitting, pattern recognition, clustering, and time series. Using these tools can give you an excellent introduction to the use of the Deep Learning Toolbox software:

- "Fit Data with a Shallow Neural Network" on page 1-45
- "Classify Patterns with a Shallow Neural Network" on page 1-63
- "Cluster Data with a Self-Organizing Map" on page 1-77
- "Shallow Neural Network Time-Series Prediction and Modeling" on page 1-89

Shallow Network Apps and Functions in Deep Learning Toolbox

There are four ways you can use the Deep Learning Toolbox software.

- The first way is through its tools. These tools provide a convenient way to access the capabilities of the toolbox for the following tasks:
- - Function fitting (`nftool`)
  - Pattern recognition (`nprtool`)
  - Data clustering (`nctool`)
  - Time-series analysis (`ntstool`)
- The second way to use the toolbox is through basic command-line operations. The command-line operations offer more flexibility than the tools, but with some added complexity. If this is your first experience with the toolbox, the tools provide the best introduction. In addition, the tools can generate scripts of documented MATLAB code to provide you with templates for creating your own customized command-line functions. The process of using the tools first, and then generating and modifying MATLAB scripts, is an excellent way to learn about the functionality of the toolbox.
- The third way to use the toolbox is through customization. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox. You can create networks with arbitrary connections, and you still be able to train them using existing toolbox training functions (as long as the network components are differentiable).
- The fourth way to use the toolbox is through the ability to modify any of the functions contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

These four levels of toolbox usage span the novice to the expert: simple tools guide the new user through specific applications, and network customization allows researchers to try novel architectures with minimal effort. Whatever your level of neural network and MATLAB knowledge, there are toolbox features to suit your needs.

Automatic Script Generation

The tools themselves form an important part of the learning process for the Deep Learning Toolbox software. They guide you through the process of designing neural networks to solve problems in four important application areas, without requiring any background in neural networks or sophistication in using MATLAB. In addition, the tools can automatically generate both simple and advanced MATLAB scripts that can reproduce the steps performed by the tool, but with the option to override default settings. These scripts can provide you with templates for creating customized code, and they can aid you in becoming familiar with the command-line functionality of the toolbox. It is highly recommended that you use the automatic script generation facility of these tools.

Deep Learning Toolbox Applications

It would be impossible to cover the total range of applications for which neural networks have provided outstanding solutions. The remaining sections of this topic describe only a few of the applications in function fitting, pattern recognition, clustering, and time series analysis. The following table provides an idea of the diversity of applications for which neural networks provide state-of-the-art solutions.

Industry	Business Applications
Aerospace	High-performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, and aircraft component fault detection
Automotive	Automobile automatic guidance system, and warranty activity analysis

Industry	Business Applications
Banking	Check and other document reading and credit application evaluation
Defense	Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, and signal/image identification
Electronics	Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, and nonlinear modeling
Entertainment	Animation, special effects, and market forecasting
Financial	Real estate appraisal, loan advising, mortgage screening, corporate bond rating, credit-line use analysis, credit card activity tracking, portfolio trading program, corporate financial analysis, and currency price prediction
Industrial	Prediction of industrial processes, such as the output gases of furnaces, replacing complex and costly equipment used for this purpose in the past
Insurance	Policy application evaluation and product optimization
Manufacturing	Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, and dynamic modeling of chemical process system
Medical	Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, and emergency-room test advisement
Oil and gas	Exploration
Robotics	Trajectory control, forklift robot, manipulator controllers, and vision systems
Securities	Market analysis, automatic bond rating, and stock trading advisory systems
Speech	Speech recognition, speech compression, vowel classification, and text-to-speech synthesis
Telecommunications	Image and data compression, automated information services, real-time translation of spoken language, and customer payment processing systems
Transportation	Truck brake diagnosis systems, vehicle scheduling, and routing systems

Shallow Neural Network Design Steps

In the remaining sections of this topic, you will follow the standard steps for designing neural networks to solve problems in four application areas: function fitting, pattern recognition, clustering, and time series analysis. The work flow for any of these problems has seven primary steps. (Data collection in step 1, while important, generally occurs outside the MATLAB environment.)

1. Collect data
2. Create the network
3. Configure the network
4. Initialize the weights and biases
5. Train the network
6. Validate the network
7. Use the network

You will follow these steps using both the GUI tools and command-line operations in the following sections:

- “Fit Data with a Shallow Neural Network” on page 1-45
- “Classify Patterns with a Shallow Neural Network” on page 1-63
- “Cluster Data with a Self-Organizing Map” on page 1-77
- “Shallow Neural Network Time-Series Prediction and Modeling” on page 1-89

Fit Data with a Shallow Neural Network

Neural networks are good at fitting functions. In fact, there is proof that a fairly simple neural network can fit any practical function.

Suppose, for instance, that you have data from a health clinic. You want to design a network that can predict the percentage of body fat of a person, given 13 anatomical measurements. You have a total of 252 example people for which you have those 13 items of data and their associated percentages of body fat.

You can solve this problem in two ways:

- Use the **Neural Net Fitting** app, as described in “Fit Data Using the Neural Net Fitting App” on page 1-45.
- Use command-line functions, as described in “Fit Data Using Command-Line Functions” on page 1-54.

It is generally best to start with the app, and then use the app to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each of the neural network apps has access to many sample data sets that you can use to experiment with the toolbox (see “Sample Data Sets for Shallow Neural Networks” on page 1-113). If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

Tip To interactively build and train deep networks, use **Deep Network Designer**.

Defining a Problem

To define a fitting (regression) problem for the toolbox, arrange a set of input vectors (predictors) as columns in a matrix. Then, arrange a set of responses (the correct output vectors for each of the input vectors) into a second matrix. For example, you can define a regression problem with four observations, each with two input features and a single response, as follows:

```
predictors = [0 1 0 1; 0 0 1 1];
responses = [0 0 0 1];
```

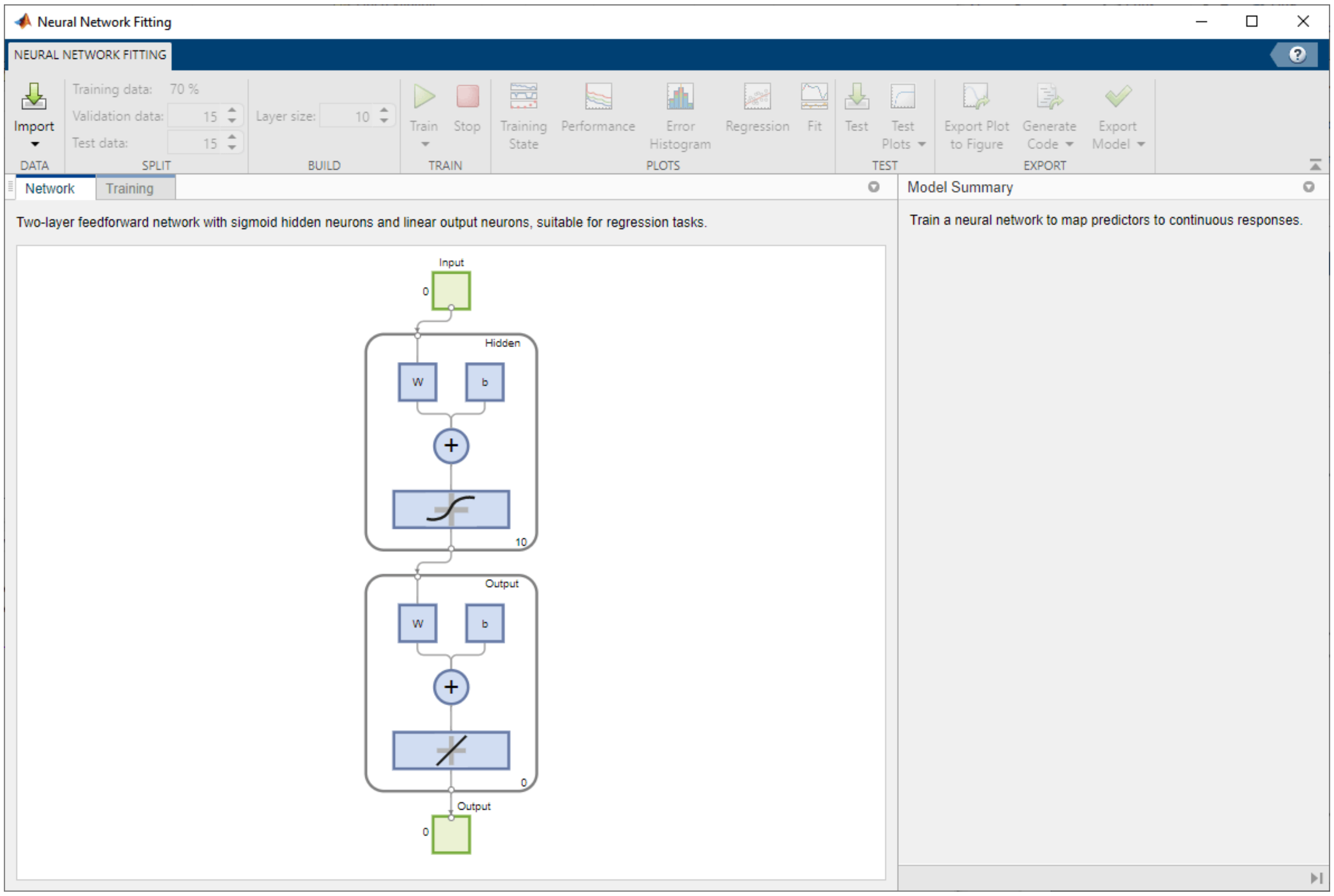
The next section shows how to train a network to fit a data set, using the **Neural Net Fitting** app. This example uses an example data set provided with the toolbox.

Fit Data Using the Neural Net Fitting App

This example shows how to train a shallow neural network to fit data using the **Neural Net Fitting** app.

Open the **Neural Net Fitting** app using `nftool`.

```
nftool
```



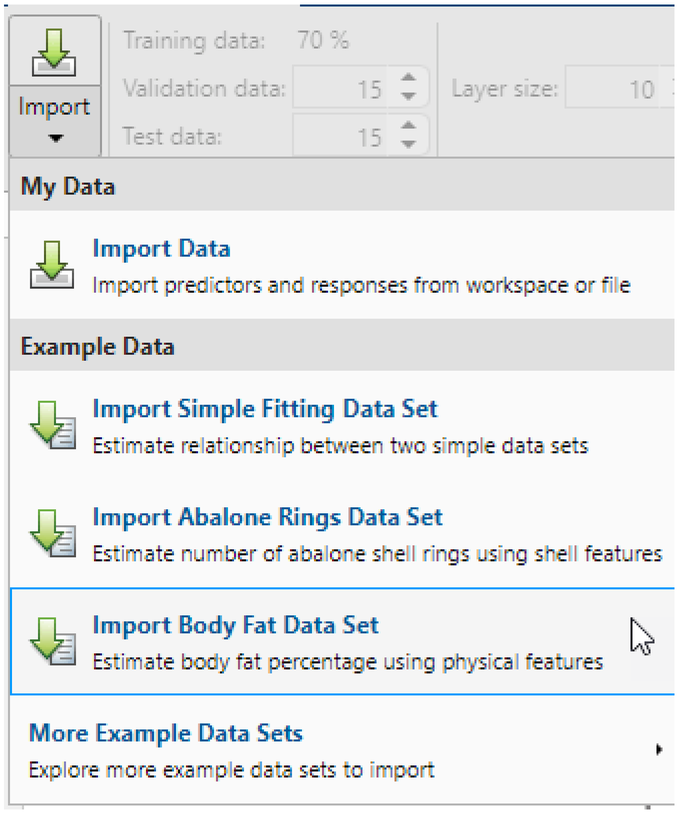
Two-layer feedforward network with sigmoid hidden neurons and linear output neurons, suitable for regression tasks.

Train a neural network to map predictors to continuous responses.

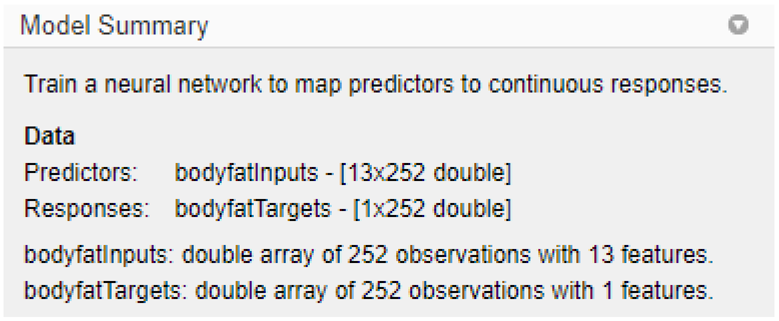
Select Data

The **Neural Net Fitting** app has example data to help you get started training a neural network.

To import example body fat data, select **Import** > **Import Body Fat Data Set**. You can use this data set to train a neural network to estimate the body fat of someone from various measurements. If you import your own data from file or the workspace, you must specify the predictors and responses, and whether the observations are in rows or columns.


Import
Training data: 70 %
Validation data: 15
Test data: 15
Layer size: 10
My Data
Import Data
Import predictors and responses from workspace or file
Example Data
Import Simple Fitting Data Set
Estimate relationship between two simple data sets
Import Abalone Rings Data Set
Estimate number of abalone shell rings using shell features
Import Body Fat Data Set
Estimate body fat percentage using physical features
More Example Data Sets
Explore more example data sets to import

Information about the imported data appears in the **Model Summary**. This data set contains 252 observations, each with 13 features. The responses contain the body fat percentage for each observation.



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.

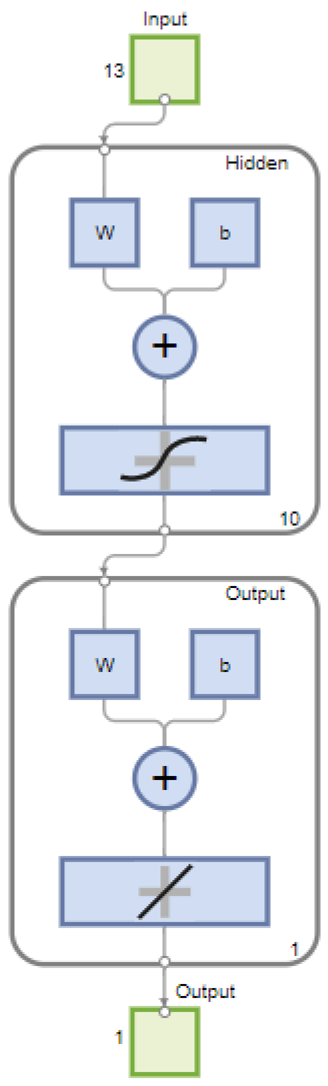
Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training”.

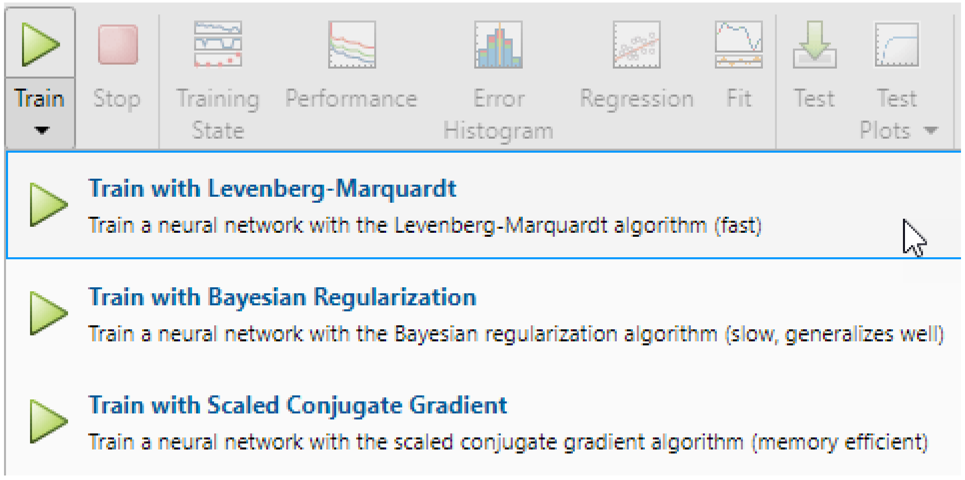
Create Network

The network is a two-layer feedforward network with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. The **Layer size** value defines the number of hidden neurons. Keep the default layer size, 10. You can see the network architecture in the **Network** pane. The network plot updates to reflect the input data. In this example, the data has 13 inputs (features) and one output.



Train Network

To train the network, select **Train > Train with Levenberg-Marquardt**. This is the default training algorithm and the same as clicking **Train**.



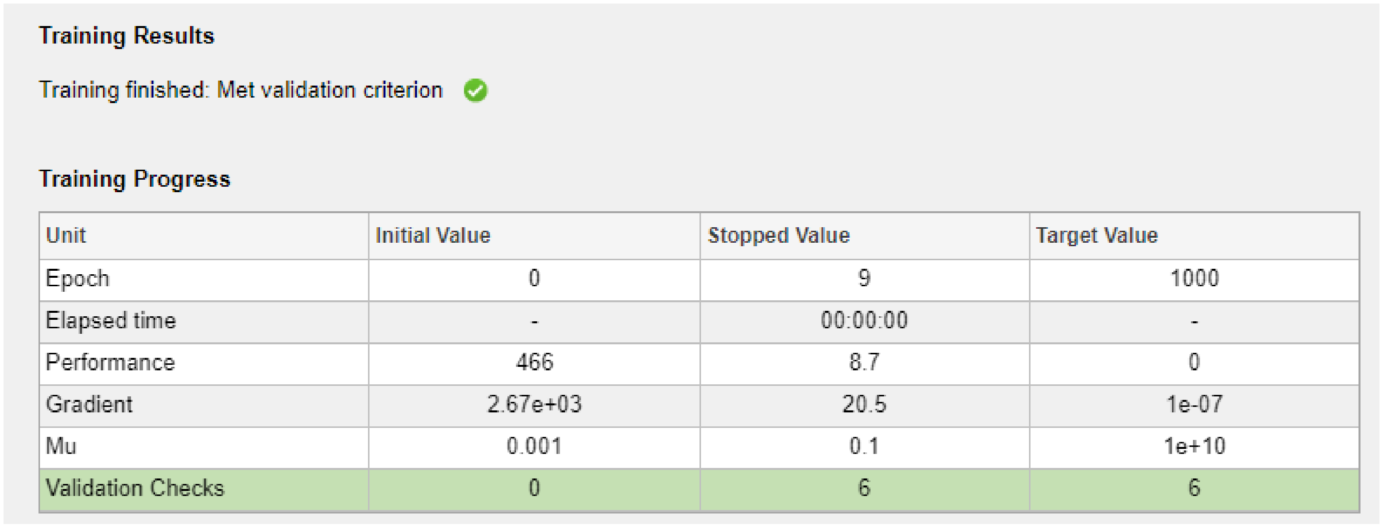
Train with Levenberg-Marquardt
Train a neural network with the Levenberg-Marquardt algorithm (fast)

Train with Bayesian Regularization
Train a neural network with the Bayesian regularization algorithm (slow, generalizes well)

Train with Scaled Conjugate Gradient
Train a neural network with the scaled conjugate gradient algorithm (memory efficient)

Training with Levenberg-Marquardt (`trainlm`) is recommended for most problems. For noisy or small problems, Bayesian Regularization (`trainbr`) can obtain a better solution, at the cost of taking longer. For large problems, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").



Training Results

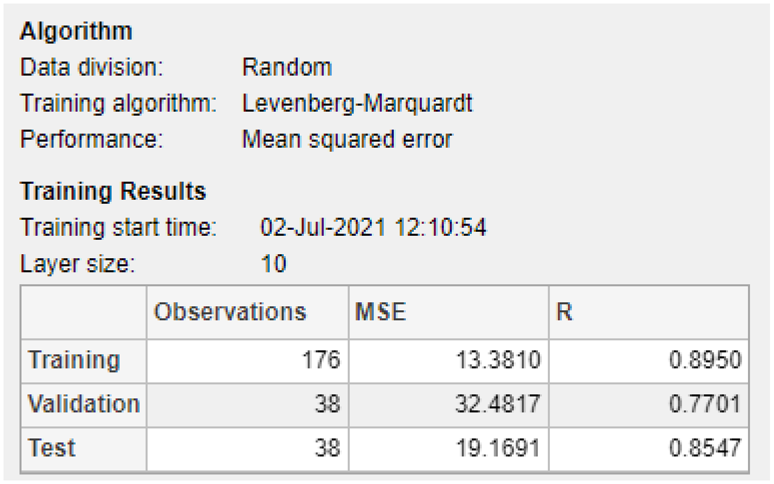
Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000
Elapsed time	-	00:00:00	-
Performance	466	8.7	0
Gradient	2.67e+03	20.5	1e-07
Mu	0.001	0.1	1e+10
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.



Algorithm

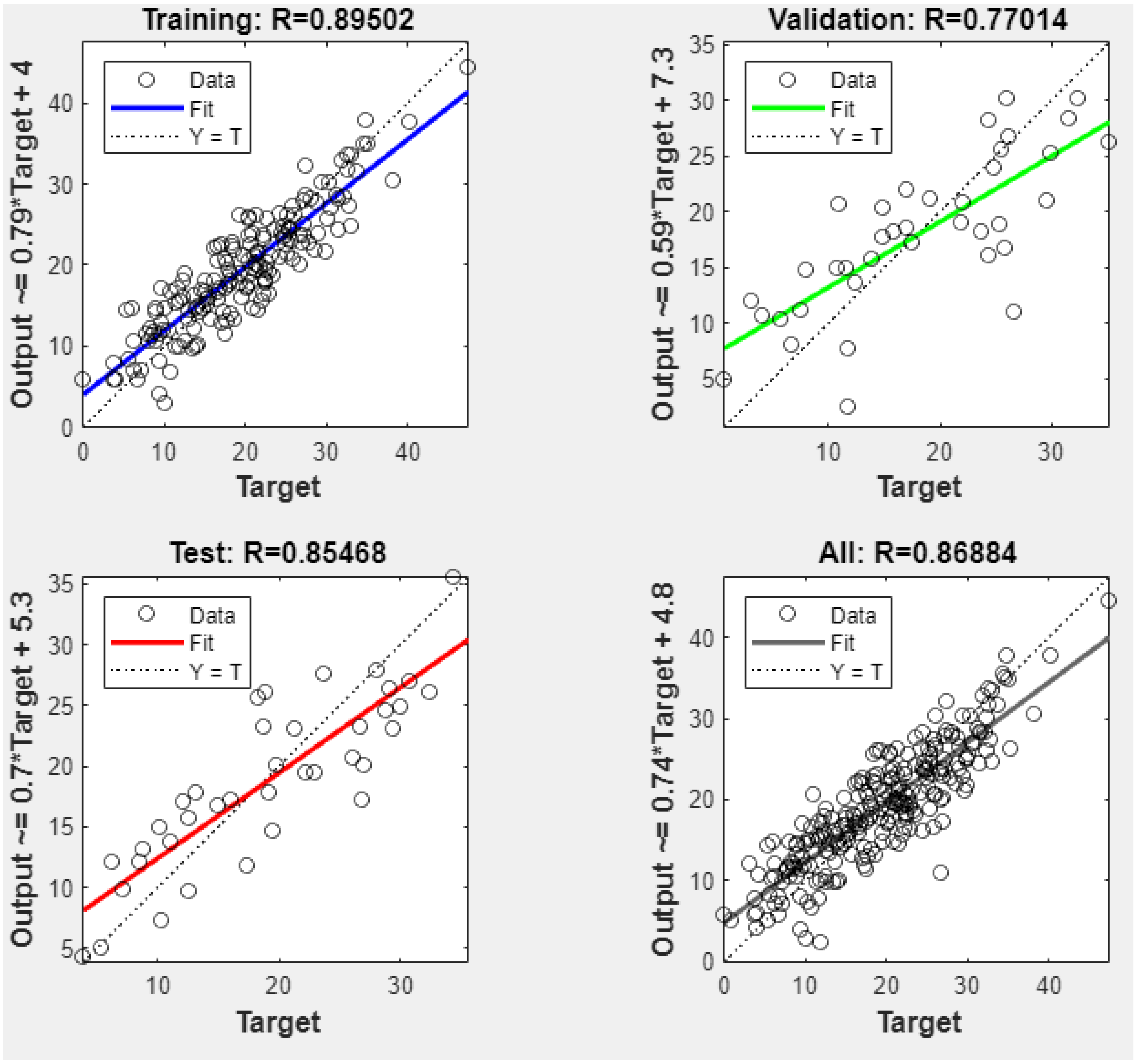
Data division:	Random
Training algorithm:	Levenberg-Marquardt
Performance:	Mean squared error

Training Results

Training start time:	02-Jul-2021 12:10:54
Layer size:	10

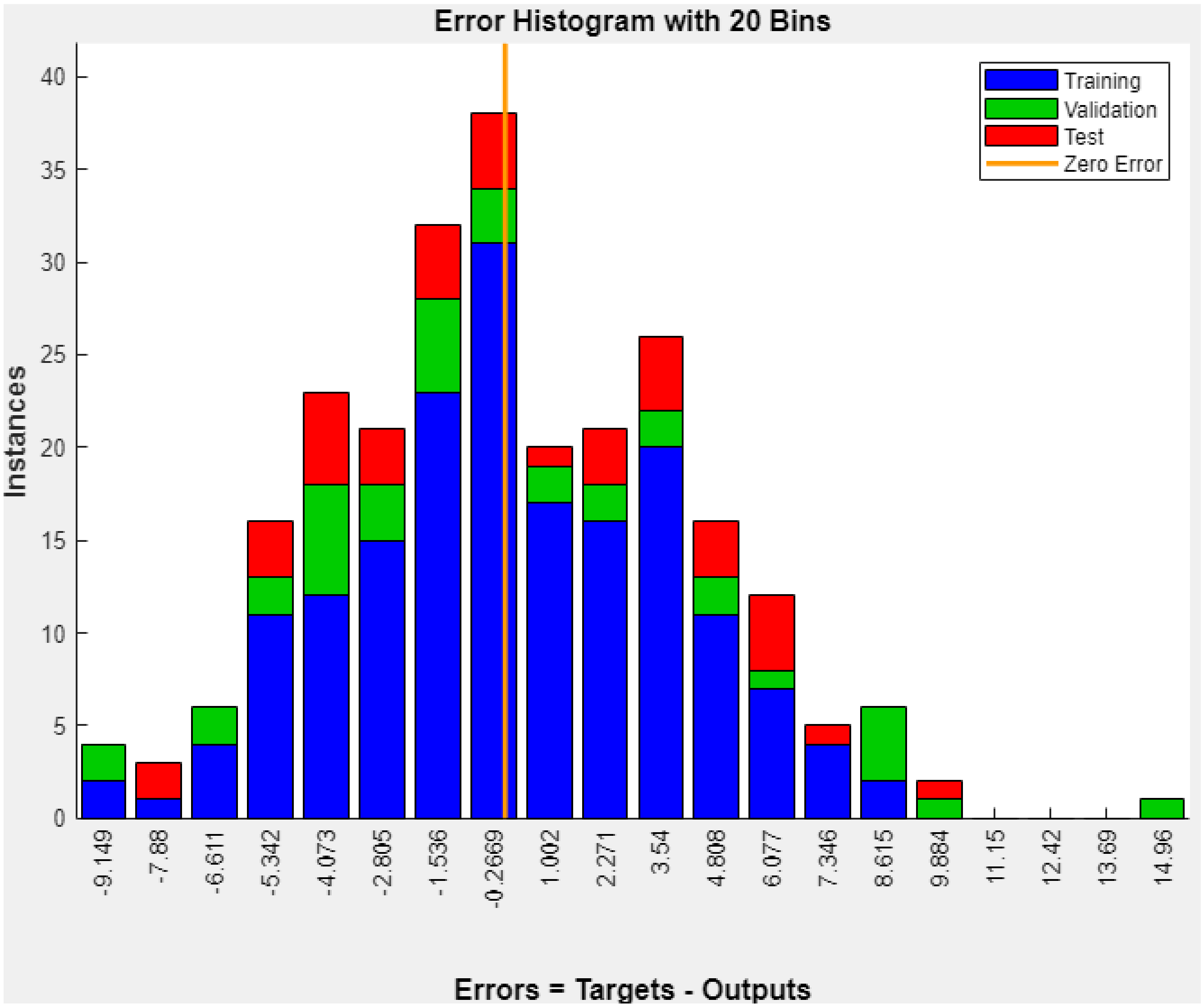
	Observations	MSE	R
Training	176	13.3810	0.8950
Validation	38	32.4817	0.7701
Test	38	19.1691	0.8547

You can further analyze the results by generating plots. To plot the linear regression, in the **Plots** section, click **Regression**. The regression plot displays the network predictions (output) with respect to responses (target) for the training, validation, and test sets.



For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the responses. For this problem, the fit is reasonably good for all of the data sets. If you require more accurate results, you can retrain the network by clicking **Train** again. Each training will have different initial weights and biases of the network, and can produce an improved network after retraining.

View the error histogram to obtain additional verification of network performance. In the **Plots** section, click **Error Histogram**.



The blue bars represent training data, the green bars represent validation data, and the red bars represent testing data. The histogram provides an indication of outliers, which are data points where the fit is significantly worse than most of the data. It is a good idea to check the outliers to determine if the data is poor, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points. You should collect more data that looks like the outlier points and retrain the network.

If you are unhappy with the network performance, you can do one of the following:

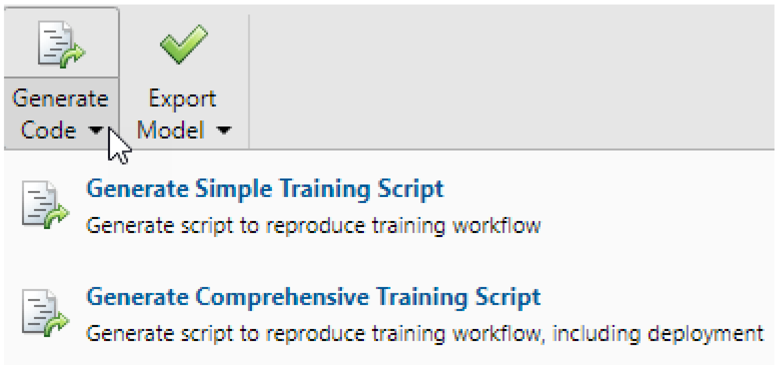
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Reducing the number of neurons can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test results. You can also generate plots to analyze the additional test data results.

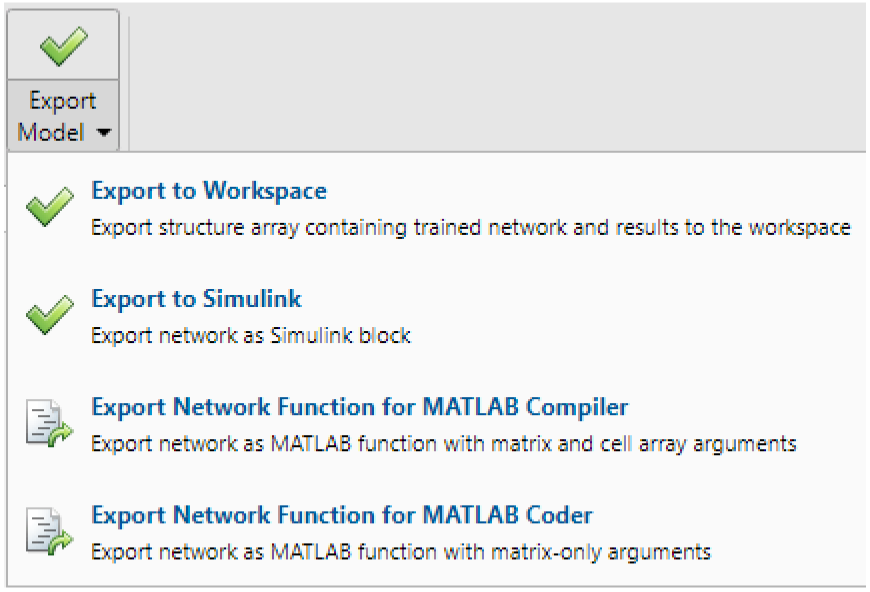
Generate Code

Select **Generate Code** > **Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command line functionality of the toolbox to customize the training process. In “Fit Data Using Command-Line Functions” on page 1-54, you will investigate the generated scripts in more detail.


Generate Code
Export Model
Generate Simple Training Script
Generate script to reproduce training workflow
Generate Comprehensive Training Script
Generate script to reproduce training workflow, including deployment

Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model** > **Export to Workspace**.



Export to Workspace
Export structure array containing trained network and results to the workspace

Export to Simulink
Export network as Simulink block

Export Network Function for MATLAB Compiler
Export network as MATLAB function with matrix and cell array arguments

Export Network Function for MATLAB Coder
Export network as MATLAB function with matrix-only arguments

Fit Data Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the apps, and then modify them to customize the network training. As an example, look at the simple script that was created in the previous section using the **Neural Net Fitting** app.

```
% Solve an Input-Output Fitting problem with a Neural Network
% Script generated by Neural Fitting app
% Created 15-Mar-2021 10:48:13
%
% This script assumes these variables are defined:
%
%   bodyfatInputs - input data.
%   bodyfatTargets - target data.

x = bodyfatInputs;
t = bodyfatTargets;

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
trainFcn = 'trainlm';  % Levenberg-Marquardt backpropagation.

% Create a Fitting Network
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize,trainFcn);

% Setup Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
```

```
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
%figure, plotregression(t,y)
%figure, plotfit(net,x,t)
```

You can save the script and then run it from the command line to reproduce the results of the previous training session. You can also edit the script to customize the training process. In this case, follow each step in the script.

Select Data

The script assumes that the predictor and response vectors are already loaded into the workspace. If the data is not loaded, you can load it as follows:

```
load bodyfat_dataset
```

This command loads the predictors `bodyfatInputs` and the responses `bodyfatTargets` into the workspace.

This data set is one of the sample data sets that is part of the toolbox. For information about the data sets available, see “Sample Data Sets for Shallow Neural Networks” on page 1-113. You can also see a list of all available data sets by entering the command `help nndatasets`. You can load the variables from any of these data sets using your own variable names. For example, the command

```
[x,t] = bodyfat_dataset;
```

will load the body fat predictors into the array `x` and the body fat responses into the array `t`.

Choose Training Algorithm

Choose training algorithm. The network uses the default Levenberg-Marquardt algorithm (`trainlm`) for training.

```
trainFcn = 'trainlm';  % Levenberg-Marquardt backpropagation.
```

For problems in which Levenberg-Marquardt does not produce as accurate results as desired, or for large data problems, consider setting the network training function to Bayesian Regularization (`trainbr`) or Scaled Conjugate Gradient (`trainscg`), respectively, with either

```
net.trainFcn = 'trainbr';
net.trainFcn = 'trainscg';
```

Create Network

Create a network. The default network for function fitting (or regression) problems, `fitnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. The network has a single hidden layer with ten neurons (default). The network has one output neuron because there is only one response value associated with each input vector.

```
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize,trainFcn);
```

Note More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `fitnet` command.

Divide Data

Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
```

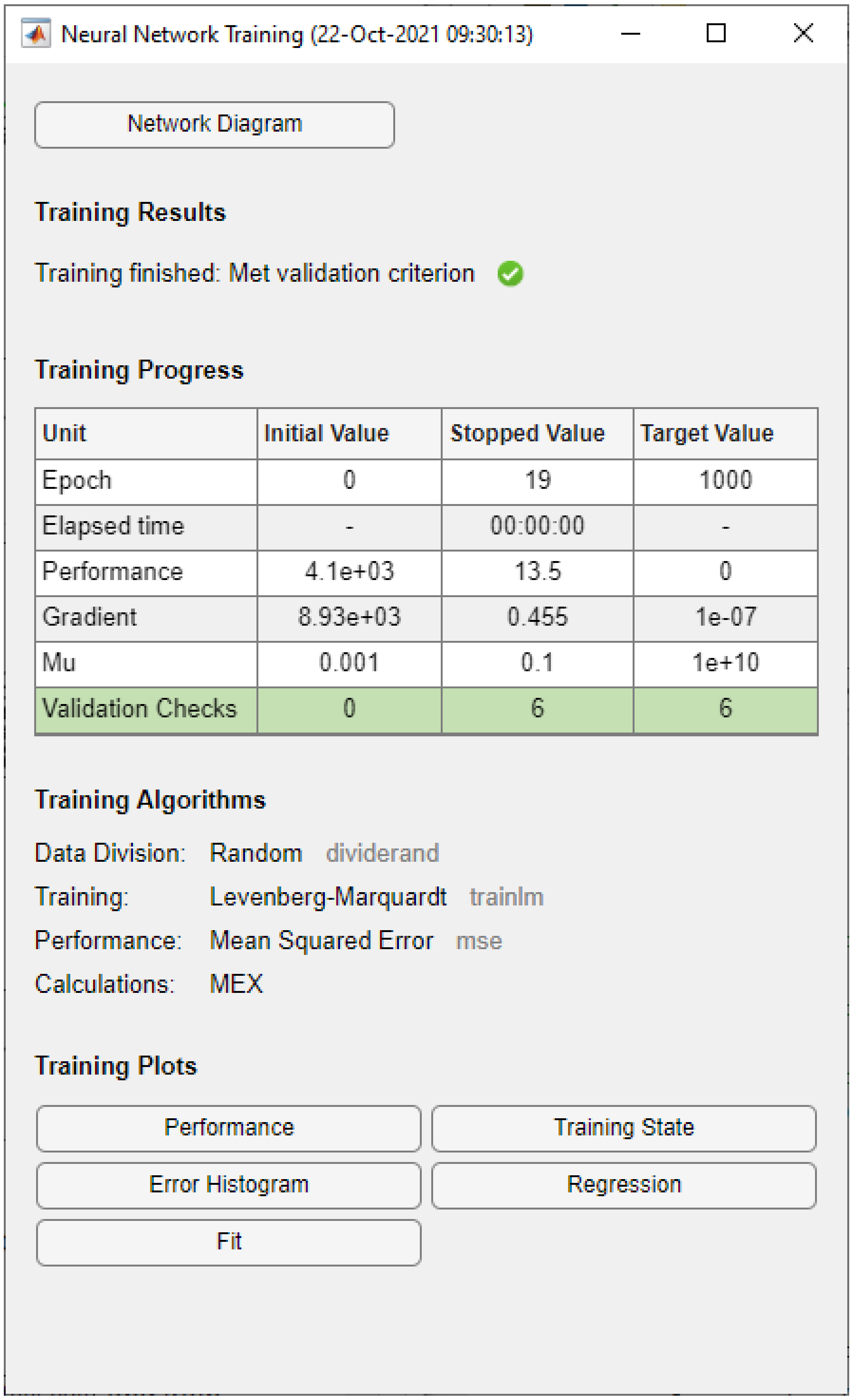
With these settings, the predictor vectors and response vectors are randomly divided, with 70% for training, 15% for validation, and 15% for testing. For more information about the data division process, see “Divide Data for Optimal Neural Network Training”.

Train Network

Train the network.

```
[net,tr] = train(net,x,t);
```

During training, the training progress window opens. You can interrupt training at any point by clicking the stop button ⏹.


Neural Network Training (22-Oct-2021 09:30:13)

Training Results

Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	19	1000
Elapsed time	-	00:00:00	-
Performance	4.1e+03	13.5	0
Gradient	8.93e+03	0.455	1e-07
Mu	0.001	0.1	1e+10
Validation Checks	0	6	6

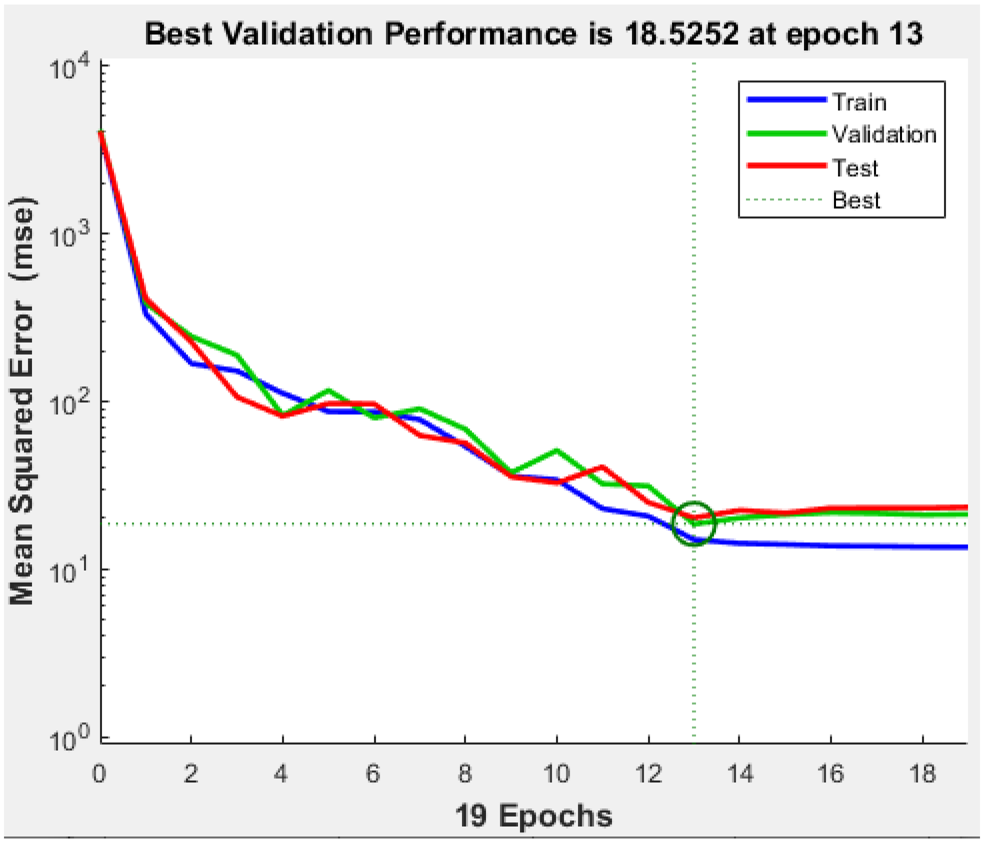
Training Algorithms

Data Division:	Random	dividerand
Training:	Levenberg-Marquardt	trainlm
Performance:	Mean Squared Error	mse
Calculations:	MEX	

Training Plots

Training finished when the validation error increased consecutively for six iterations. If you click **Performance** in the training window, a plot of the training errors, validation errors, and test errors appears, as shown in the following figure. In this example, the result is reasonable because of the following considerations:

- The final mean-square error is small.
- The test set error and the validation set error have similar characteristics.
- No significant overfitting has occurred by epoch 13 (where the best validation performance occurs).



Test Network

Test the network. After the network has trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors, and overall performance.

```
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)
```

```
performance =

   16.2815
```

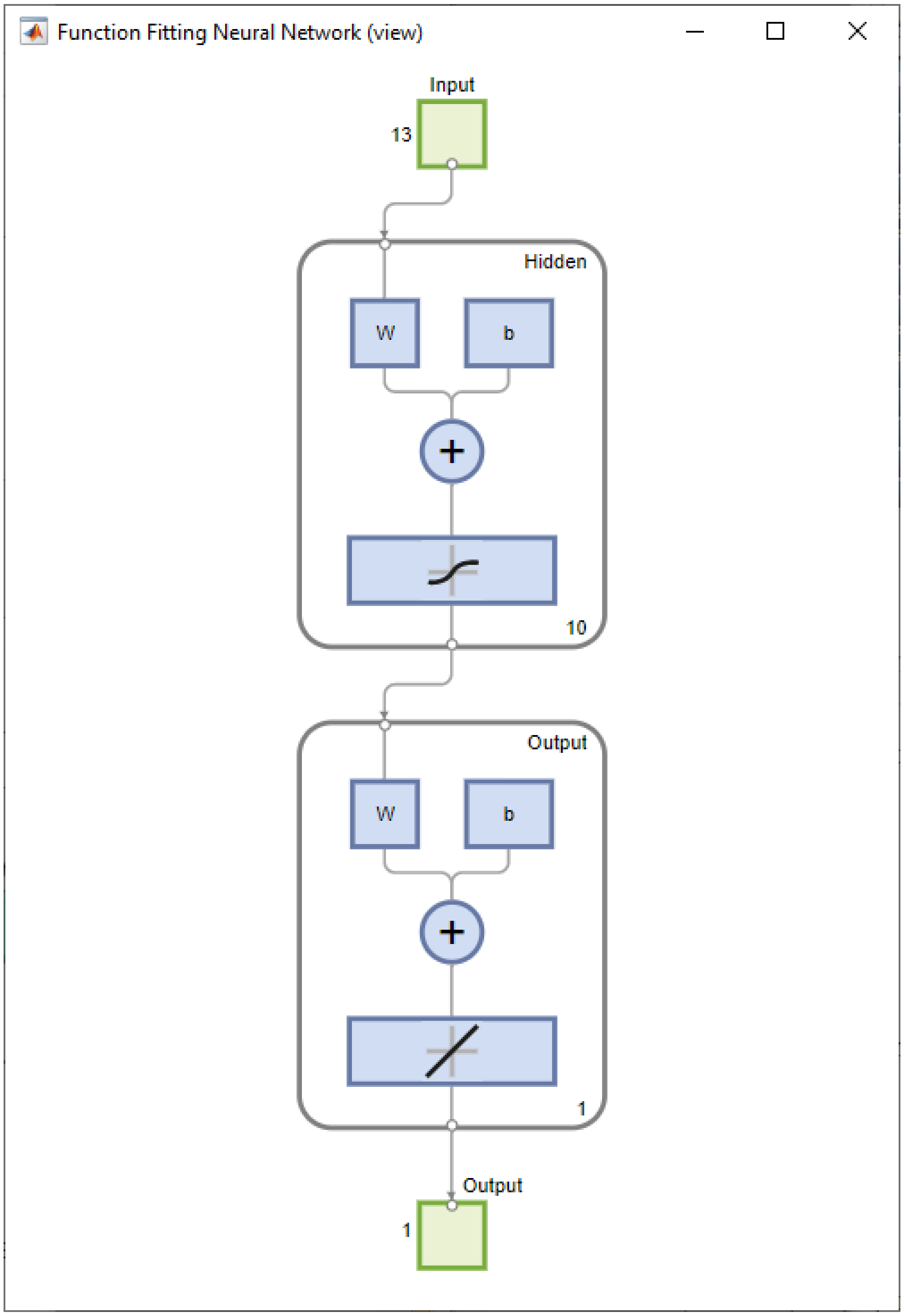
It is also possible to calculate the network performance only on the test set by using the testing indices, which are located in the training record. For more information, see “Analyze Shallow Neural Network Performance After Training”.

```
tInd = tr.testInd;
tstOutputs = net(x(:,tInd));
tstPerform = perform(net,t(tInd),tstOutputs)
tstPerform =

   20.1698
```

View Network

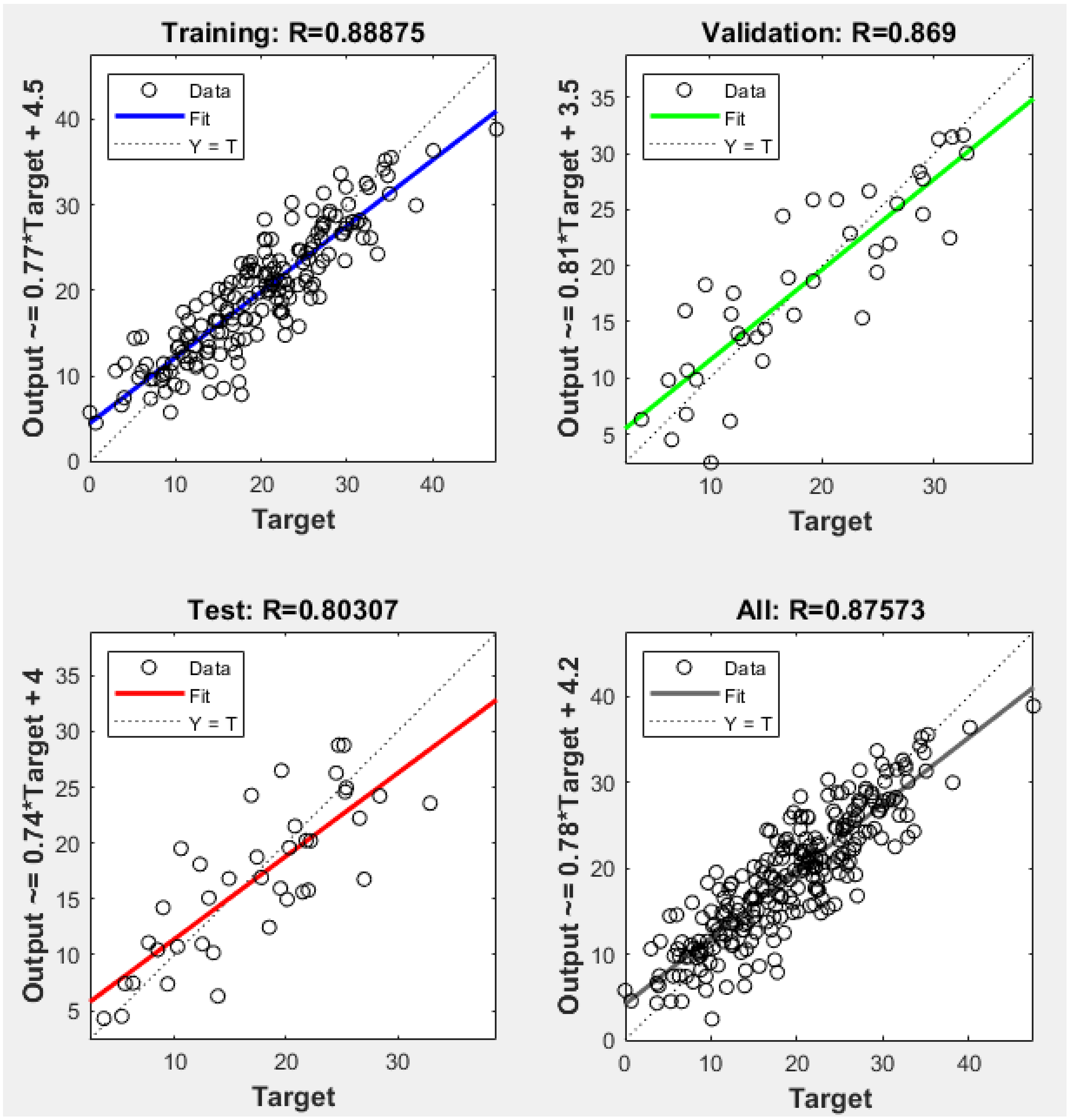
View the network diagram.

```
view(net)
```



Analyze Results

Analyze the results. To perform a linear regression between the network predictions (outputs) and the corresponding responses (targets), click **Regression** in the training window.



The output tracks the responses well for training, testing, and validation sets, and the R-value is over 0.87 for the total data set. If even more accurate results were required, you could try any of these approaches:

- Reset the initial network weights and biases to new values with `init` and train again.
- Increase the number of hidden neurons.
- Use a larger training data set.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Training Algorithms”).

In this case, the network response is satisfactory, and you can now put the network to use on new data.

Next Steps

To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the regression plot), and watch it animate.
- Plot from the command line with functions such as `plotfit`, `plotregression`, `plottrainstate` and `plotperform`.

Also, see the advanced script for more options, when training from the command line.

Each time a neural network is trained can result in a different solution due to random initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting”.

See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `trainlm` | `fitnet`

Related Examples

- “Classify Patterns with a Shallow Neural Network” on page 1-63
- “Cluster Data with a Self-Organizing Map” on page 1-77
- “Shallow Neural Network Time-Series Prediction and Modeling” on page 1-89

Classify Patterns with a Shallow Neural Network

In addition to function fitting, neural networks are also good at recognizing patterns.

For example, suppose you want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis, etc. You have 699 example cases for which you have 9 features and the correct classification as benign or malignant.

As with function fitting, there are two ways to solve this problem:

- Use the **Neural Net Pattern Recognition** app, as described in “Classify Patterns Using the Neural Net Pattern Recognition App” on page 1-64.
- Use command-line functions, as described in “Classify Patterns Using Command-Line Functions” on page 1-70.

It is generally best to start with the app, and then use the app to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each of the neural network apps has access to many sample data sets that you can use to experiment with the toolbox (see “Sample Data Sets for Shallow Neural Networks” on page 1-113). If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

Tip To interactively build and train deep networks, use **Deep Network Designer**.

Defining a Problem

To define a pattern recognition problem, arrange a set of input vectors (predictors) as columns in a matrix. Then arrange another set of response vectors indicating the classes to which the observations are assigned.

When there are only two classes, each response has two elements, 0 and 1, indicating which class the corresponding observation belongs to. For example, you can define a two-class classification problem as follows:

```
predictors = [7 10 3 1 6; 5 8 1 1 6; 6 7 1 1 6];
responses = [0 0 1 1 0; 1 1 0 0 1];
```

The data consists of five observations, each with three features, classified into one of two classes.

When predictors are to be classified into *N* different classes, the responses have *N* elements. For each response, one element is 1 and the others are 0. For example, the following lines show how to define a classification problem that divides the corners of a 5-by-5-by-5 cube into three classes:

- The origin (the first input vector) in one class
- The corner farthest from the origin (the last input vector) in a second class
- All other points in a third class

```
predictors  = [0 0 0 0 5 5 5 5; 0 0 5 5 0 0 5 5; 0 5 0 5 0 5 0 5];
responses = [1 0 0 0 0 0 0 0; 0 1 1 1 1 1 1 0; 0 0 0 0 0 0 0 1];
```

The data consists of eight observations, each with three features, classified into one of three classes.

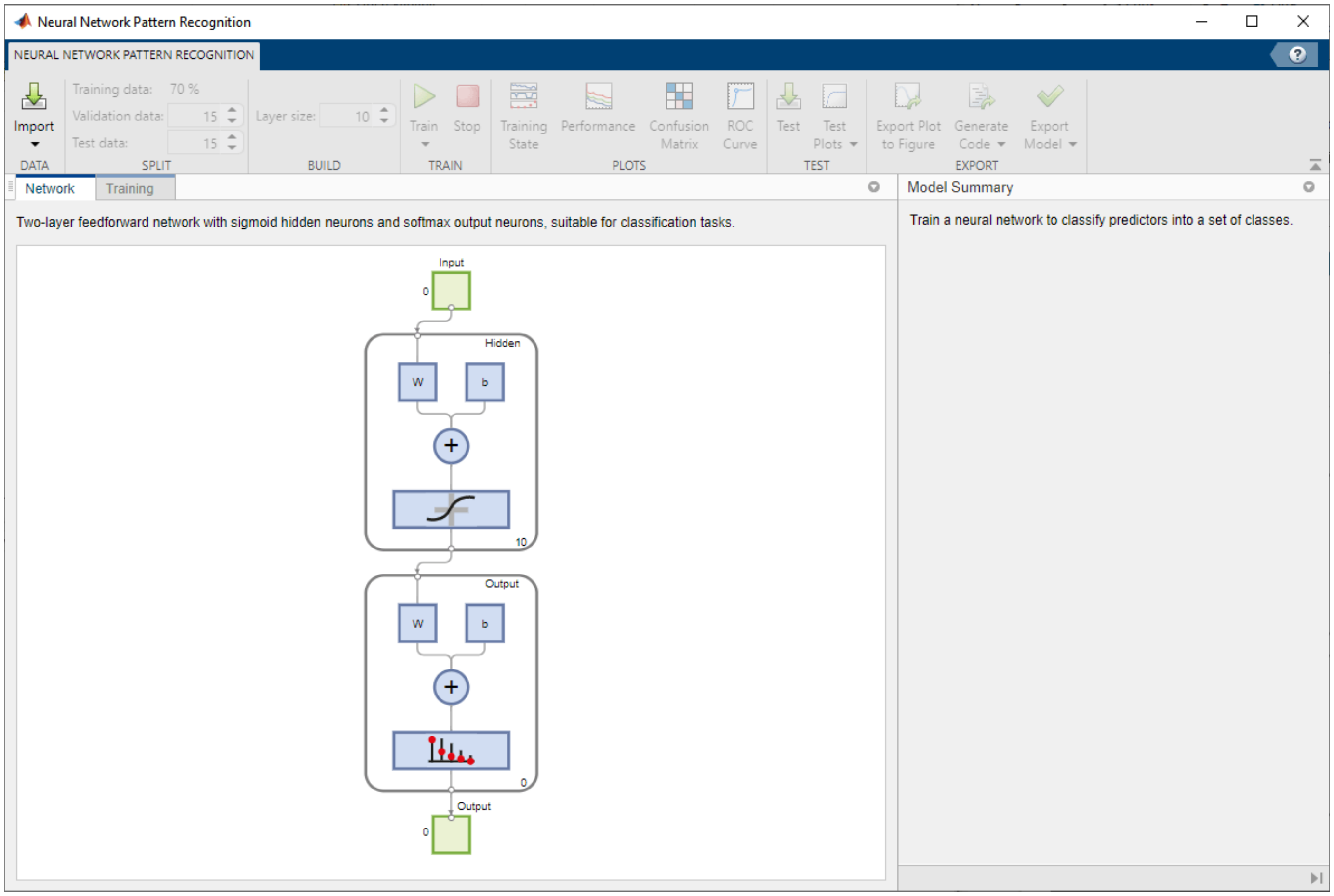
The next section shows how to train a network to recognize patterns, using the **Neural Net Pattern Recognition** app. This example uses an example data set provided with the toolbox.

Classify Patterns Using the Neural Net Pattern Recognition App

This example shows how to train a shallow neural network to classify patterns using the **Neural Net Pattern Recognition** app.

Open the **Neural Net Pattern Recognition** app using `nprtool`.

```
nprtool
```



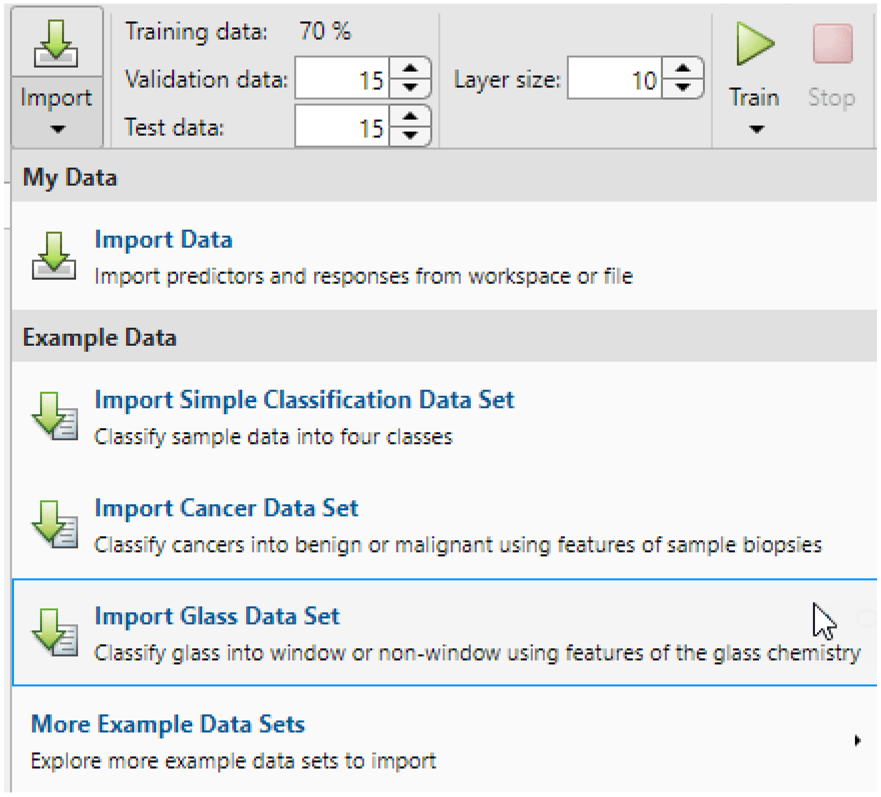
Two-layer feedforward network with sigmoid hidden neurons and softmax output neurons, suitable for classification tasks.

Train a neural network to classify predictors into a set of classes.

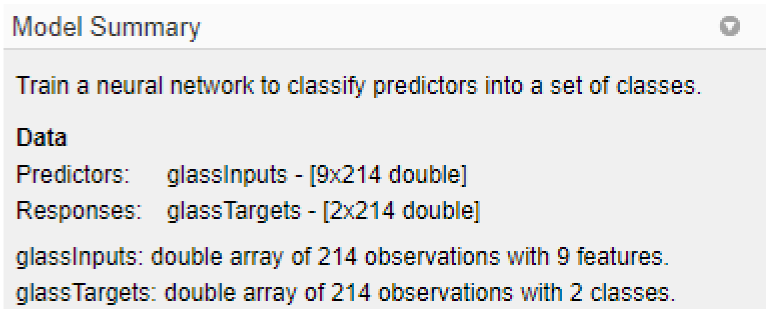
Select Data

The **Neural Net Pattern Recognition** app has example data to help you get started training a neural network.

To import example glass classification data, select **Import** > **Import Glass Data Set**. You can use this data set to train a neural network to classify glass as window or non-window, using properties of the glass chemistry. If you import your own data from file or the workspace, you must specify the predictors and responses, and whether the observations are in rows or columns.


Import
Training data: 70 %
Validation data: 15
Test data: 15
Layer size: 10
Train
Stop
My Data
Import Data
Import predictors and responses from workspace or file
Example Data
Import Simple Classification Data Set
Classify sample data into four classes
Import Cancer Data Set
Classify cancers into benign or malignant using features of sample biopsies
Import Glass Data Set
Classify glass into window or non-window using features of the glass chemistry
More Example Data Sets
Explore more example data sets to import

Information about the imported data appears in the **Model Summary**. This data set contains 214 observations, each with 9 features. Each observation is classified into one of two classes: window or non-window.


Model Summary

Train a neural network to classify predictors into a set of classes.

Data

Predictors: glassInputs - [9x214 double]

Responses: glassTargets - [2x214 double]

glassInputs: double array of 214 observations with 9 features.

glassTargets: double array of 214 observations with 2 classes.

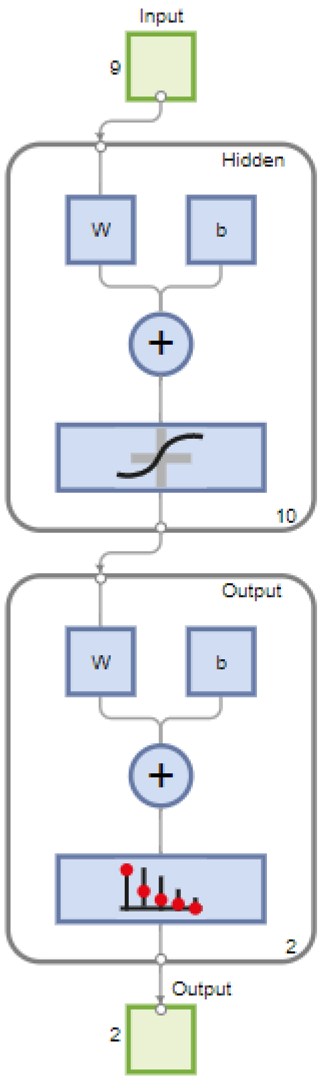
Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training”.

Create Network

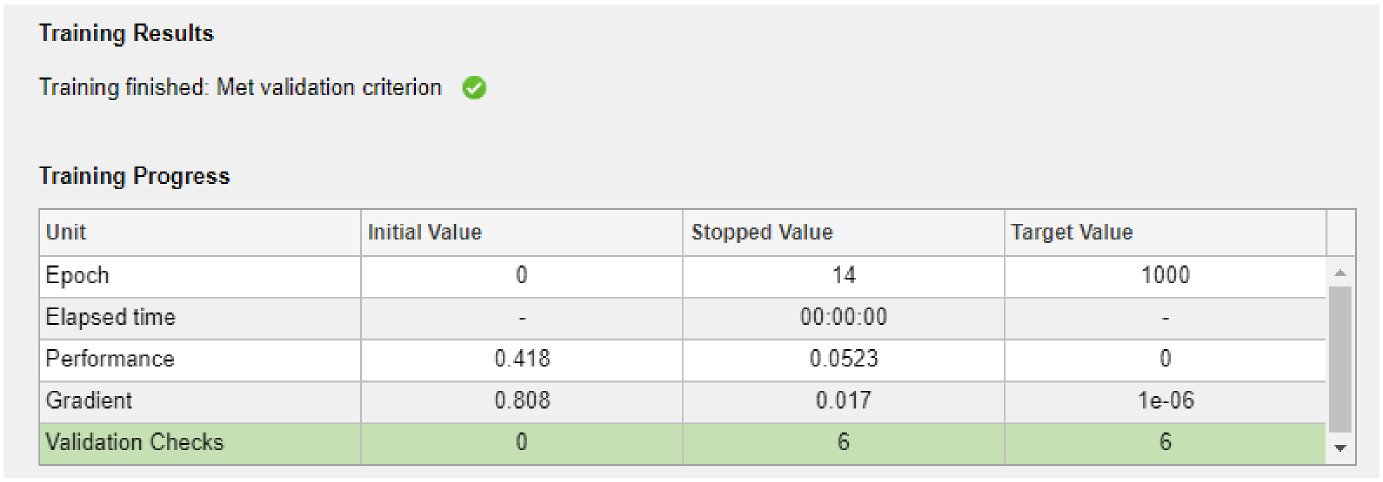
The network is a two-layer feedforward network with a sigmoid transfer function in the hidden layer and a softmax transfer function in the output layer. The size of the hidden layer corresponds to the number of hidden neurons. The default layer size is 10. You can see the network architecture in the **Network** pane. The number of output neurons is set to 2, which is equal to the number of classes specified by the response data.



Train Network

To train the network, click **Train**.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").



Training Results

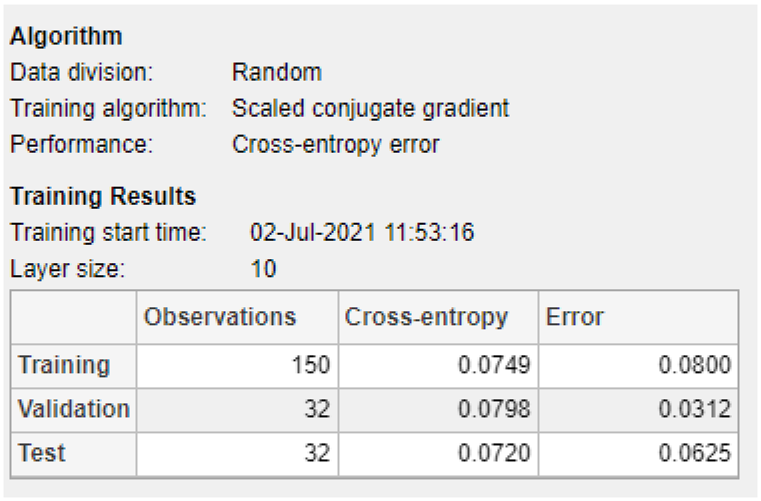
Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	14	1000
Elapsed time	-	00:00:00	-
Performance	0.418	0.0523	0
Gradient	0.808	0.017	1e-06
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.



Algorithm

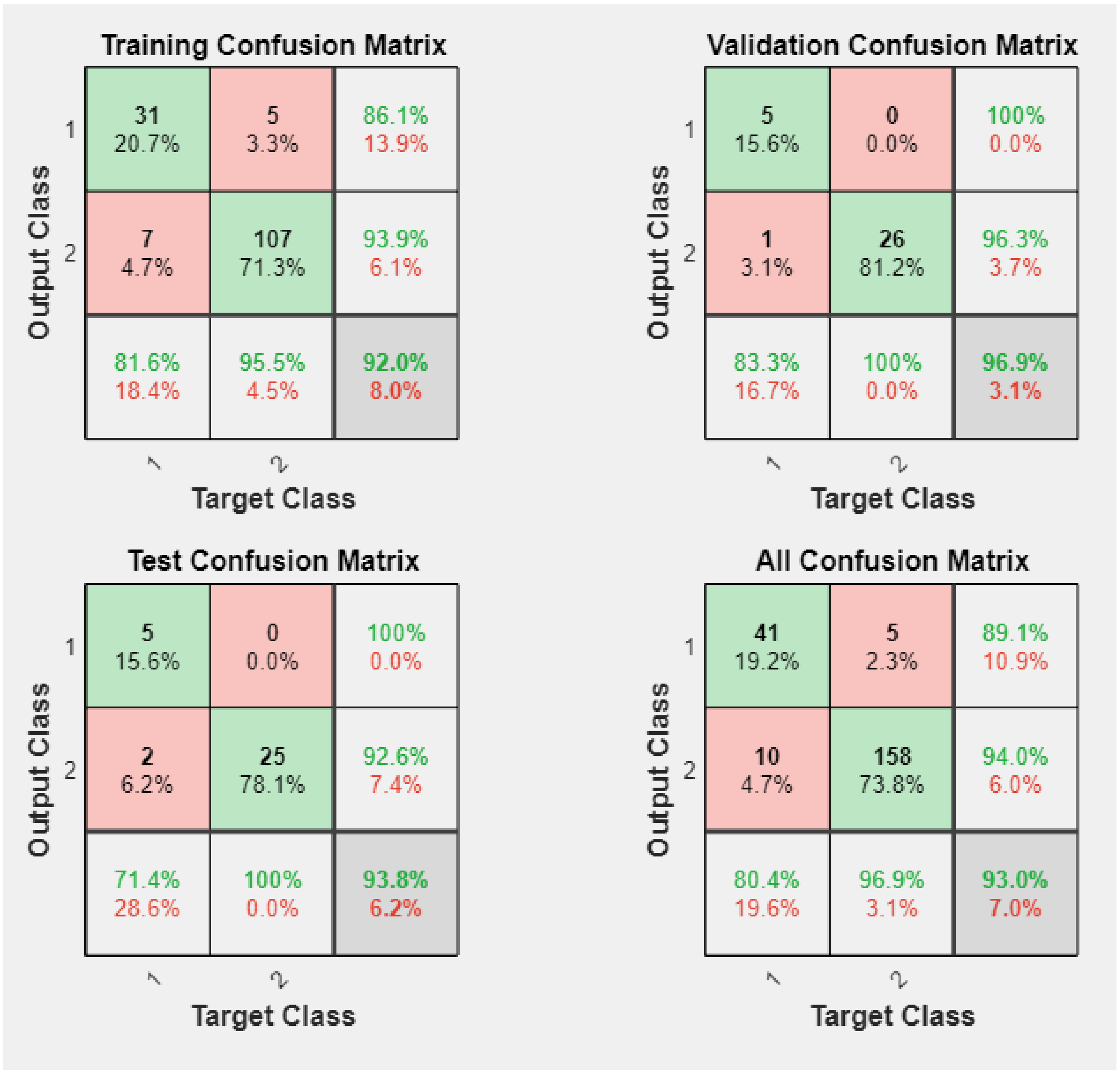
Data division:	Random
Training algorithm:	Scaled conjugate gradient
Performance:	Cross-entropy error

Training Results

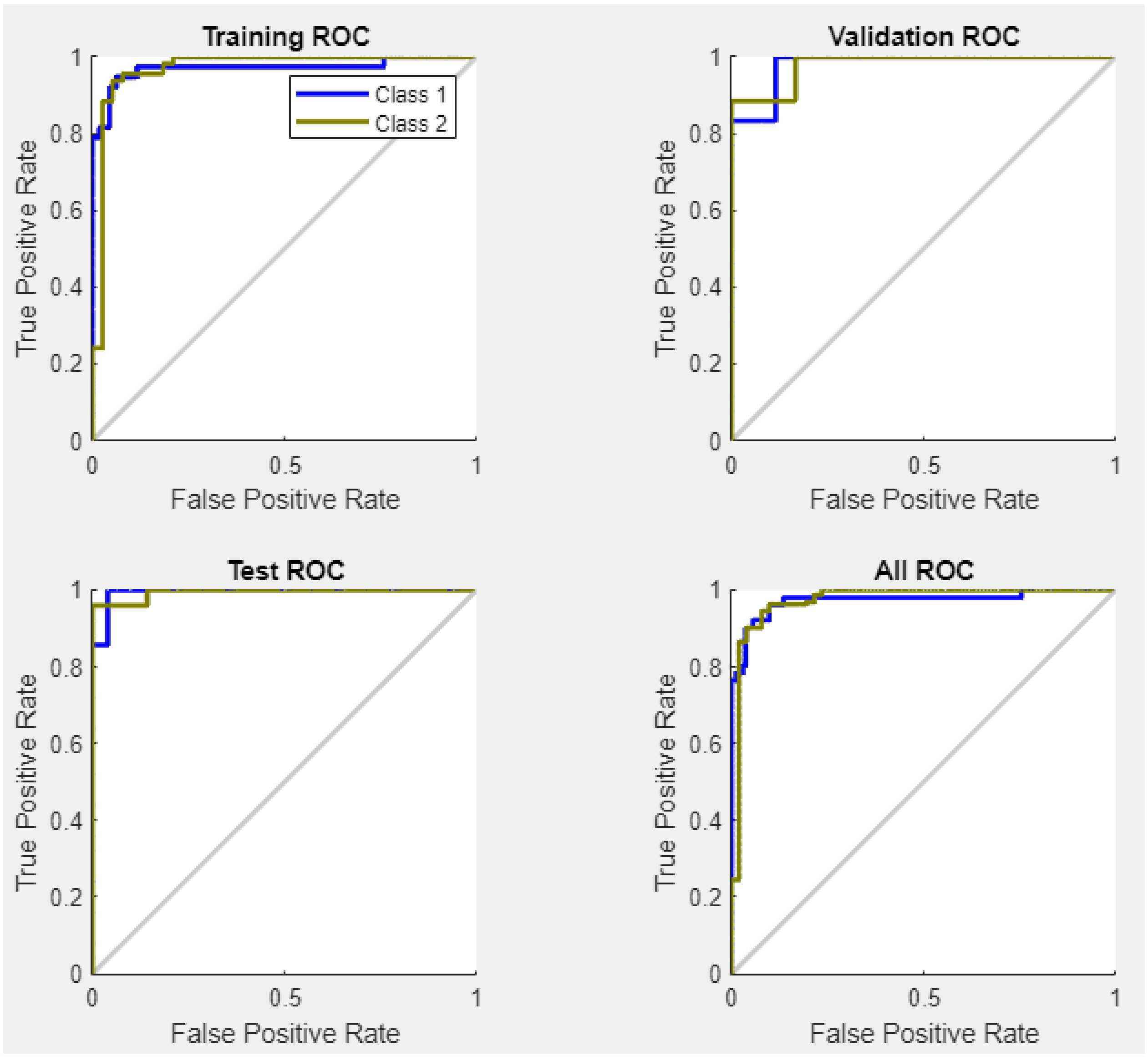
Training start time:	02-Jul-2021 11:53:16
Layer size:	10

	Observations	Cross-entropy	Error
Training	150	0.0749	0.0800
Validation	32	0.0798	0.0312
Test	32	0.0720	0.0625

You can further analyze the results by generating plots. To plot the confusion matrices, in the **Plots** section, click **Confusion Matrix**. The network outputs are very accurate, as you can see by the high numbers of correct classifications in the green squares (diagonal) and the low numbers of incorrect classifications in the red squares (off-diagonal).



View the ROC curve to obtain additional verification of network performance. In the **Plots** section, click **ROC Curve**.



The colored lines in each axis represent the ROC curves. The ROC curve is a plot of the true positive rate (sensitivity) versus the false positive rate (1 - specificity) as the threshold is varied. A perfect test would show points in the upper-left corner, with 100% sensitivity and 100% specificity. For this problem, the network performs very well.

If you are unhappy with the network performance, you can do one of the following:

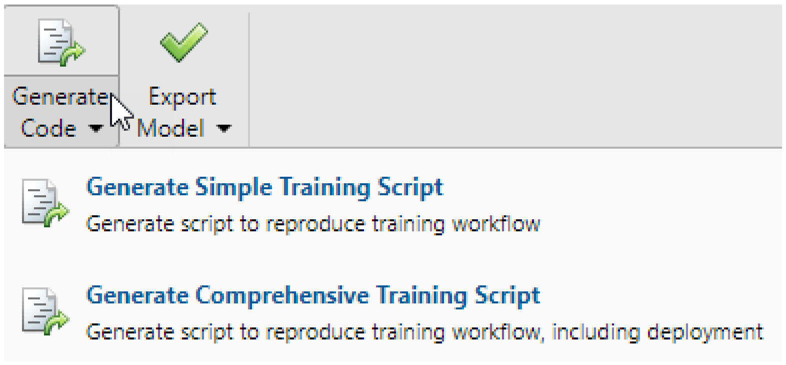
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Reducing the number of neurons can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test results. You can also generate plots to analyze the additional test results.

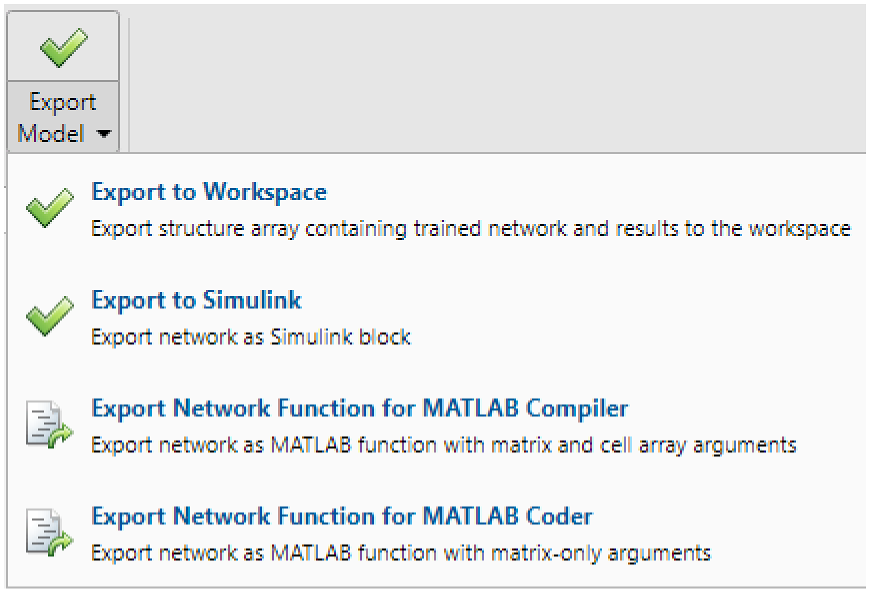
Generate Code

Select **Generate Code** > **Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command line functionality of the toolbox to customize the training process. In "Classify Patterns Using Command-Line Functions" on page 1-70, you will investigate the generated scripts in more detail.


Generate Code
Export Model
Generate Simple Training Script
Generate script to reproduce training workflow
Generate Comprehensive Training Script
Generate script to reproduce training workflow, including deployment

Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ and other MATLAB code generation tools. To export your trained network and results, select **Export Model** > **Export to Workspace**.


Export Model
Export to Workspace
Export structure array containing trained network and results to the workspace
Export to Simulink
Export network as Simulink block
Export Network Function for MATLAB Compiler
Export network as MATLAB function with matrix and cell array arguments
Export Network Function for MATLAB Coder
Export network as MATLAB function with matrix-only arguments

Classify Patterns Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the apps, and then modify them to customize the network training. As an example, look at the simple script that was created in the previous section using the **Neural Net Pattern Recognition** app.

```
% Solve a Pattern Recognition Problem with a Neural Network
% Script generated by Neural Pattern Recognition app
% Created 22-Mar-2021 16:50:20
%
% This script assumes these variables are defined:
%
%   glassInputs - input data.
%   glassTargets - target data.

x = glassInputs;
t = glassTargets;

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
trainFcn = 'trainscg';  % Scaled conjugate gradient backpropagation.

% Create a Pattern Recognition Network
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize, trainFcn);

% Setup Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)
tind = vec2ind(t);
yind = vec2ind(y);
percentErrors = sum(tind ~= yind)/numel(tind);

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
%figure, plotconfusion(t,y)
%figure, plotroc(t,y)
```

You can save the script and then run it from the command line to reproduce the results of the previous training session. You can also edit the script to customize the training process. In this case, follow each step in the script.

Select Data

The script assumes that the predictor and response vectors are already loaded into the workspace. If the data is not loaded, you can load it as follows:

```
load glass_dataset
```

This command loads the predictors `glassInputs` and the responses `glassTargets` into the workspace.

This data set is one of the sample data sets that is part of the toolbox. For information about the data sets available, see “Sample Data Sets for Shallow Neural Networks” on page 1-113. You can also see a list of all available data sets by entering the command `help nndatasets`. You can load the variables from any of these data sets using your own variable names. For example, the command

```
[x,t] = glass_dataset;
```

will load the glass predictors into the array `x` and the glass responses into the array `t`.

Choose Training Algorithm

Define training algorithm.

```
trainFcn = 'trainscg';  % Scaled conjugate gradient backpropagation.
```

Create Network

Create the network. The default network for pattern recognition (classification) problems, `patternnet`, is a feedforward network with the default sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer. The network has a single hidden layer with ten neurons (default).

The network has two output neurons, because there are two response values (classes) associated with each input vector. Each output neuron represents a class. When an input vector of the appropriate class is applied to the network, the corresponding neuron should produce a 1, and the other neurons should output a 0.

```
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize, trainFcn);
```

Note More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `patternnet` command.

Divide Data

Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
```

With these settings, the predictor vectors and response vectors are randomly divided, with 70% for training, 15% for validation, and 15% for testing. For more information about the data division process, see “Divide Data for Optimal Neural Network Training”.

Train Network

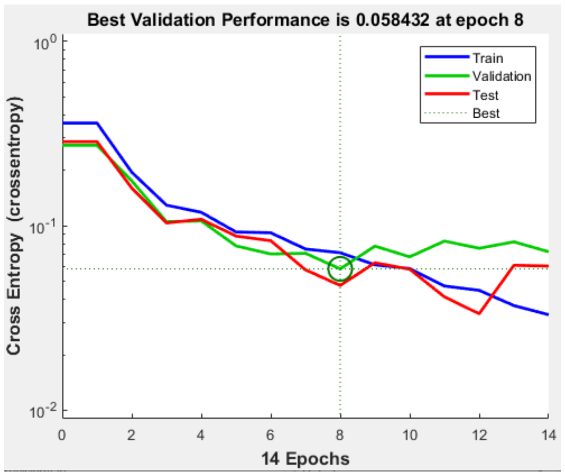
Train the network.

```
[net,tr] = train(net,x,t);
```

During training, the training progress window opens. You can interrupt training at any point by clicking the stop button ■.

Training finished when the validation error increased consecutively for six iterations, which occurred at iteration 14.

If you click **Performance** in the training window, a plot of the training errors, validation errors, and test errors appears, as shown in the following figure.



In this example, the result is reasonable as the final cross-entropy error is small.

Test Network

Test the network. After the network has trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors, and overall performance.

```
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)

performance =

    0.0659
```

You can also compute the fraction of misclassified observations. In this example, the model has a very low misclassification rate.

```
tind = vec2ind(t);
yind = vec2ind(y);
percentErrors = sum(tind ~= yind)/numel(tind)
```

```
percentErrors =

    0.0514
```

It is also possible to calculate the network performance only on the test set, by using the testing indices, which are located in the training record.

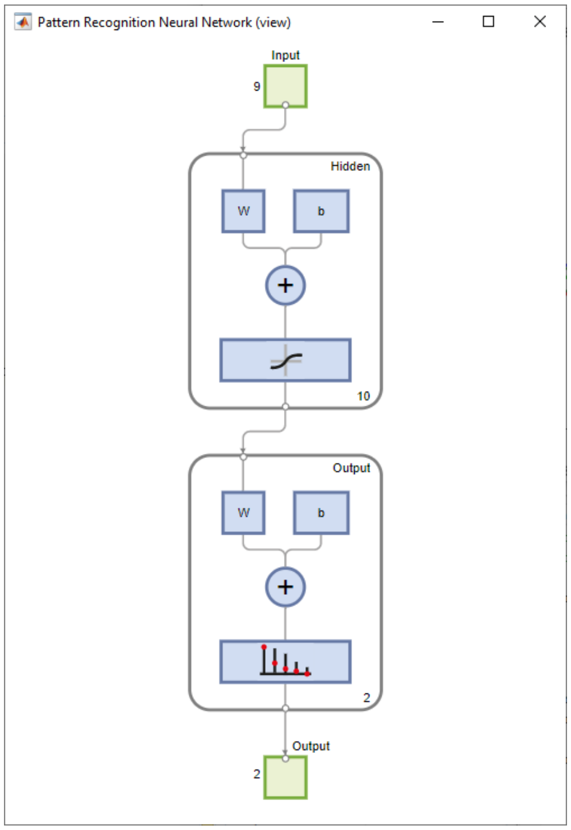
```
tInd = tr.testInd;
tstOutputs = net(x(:,tInd));
tstPerform = perform(net,t(tInd),tstOutputs)
```

```
tstPerform =

    2.0163
```

View Network

View the network diagram.

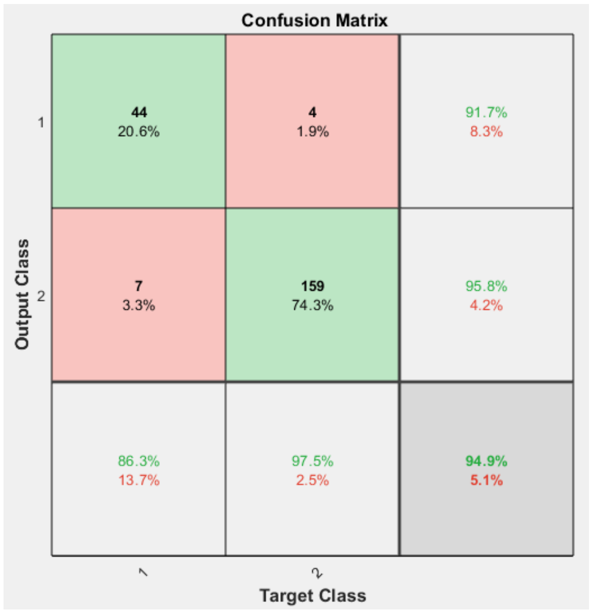
```
view(net)
```



Analyze Results

Use the `plotconfusion` function to plot the confusion matrix. You can also plot the confusion matrix for each of the data sets by clicking **Confusion** in the training window.

```
figure, plotconfusion(t,y)
```



The diagonal green cells show the number of cases that were correctly classified, and the off-diagonal red cells show the misclassified cases. The results show very good recognition. If you needed even more accurate results, you could try any of the following approaches:

- Reset the initial network weights and biases to new values with `init` and train again.
- Increase the number of hidden neurons.
- Use a larger training data set.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Training Algorithms”).

In this case, the network results are satisfactory, and you can now put the network to use on new input data.

Next Steps

To get more experience in command line operations, here are some tasks you can try:

- During training, open a plot window (such as the confusion plot), and watch it animate.
- Plot from the command line with functions such as `plotroc` and `plottrainstate`.

Each time a neural network is trained can result in a different solution due to random initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting”.

See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `trainscg`

Related Examples

- “Fit Data with a Shallow Neural Network” on page 1-45
- “Cluster Data with a Self-Organizing Map” on page 1-77
- “Shallow Neural Network Time-Series Prediction and Modeling” on page 1-89

Cluster Data with a Self-Organizing Map

Clustering data is another excellent application for neural networks. This process involves grouping data by similarity. For example, you might perform:

- Market segmentation by grouping people according to their buying patterns
- Data mining by partitioning data into related subsets
- Bioinformatic analysis by grouping genes with related expression patterns

Suppose that you want to cluster flower types according to petal length, petal width, sepal length, and sepal width. You have 150 example cases for which you have these four measurements.

As with function fitting and pattern recognition, there are two ways to solve this problem:

- Use the **Neural Net Clustering** app, as described in "Cluster Data Using the Neural Net Clustering App" on page 1-77.
- Use command-line functions, as described in "Cluster Data Using Command-Line Functions" on page 1-83.

It is generally best to start with the app, and then use the app to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each of the neural network apps has access to sample data sets that you can use to experiment with the toolbox (see "Sample Data Sets for Shallow Neural Networks" on page 1-113). If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

Defining a Problem

To define a clustering problem, arrange input vectors (predictors) to be clustered as columns in an input matrix. For instance, you might want to cluster this set of 10 two-element vectors:

```
predictors = [7 0 6 2 6 5 6 1 0 1; 6 2 5 0 7 5 5 1 2 2]
```

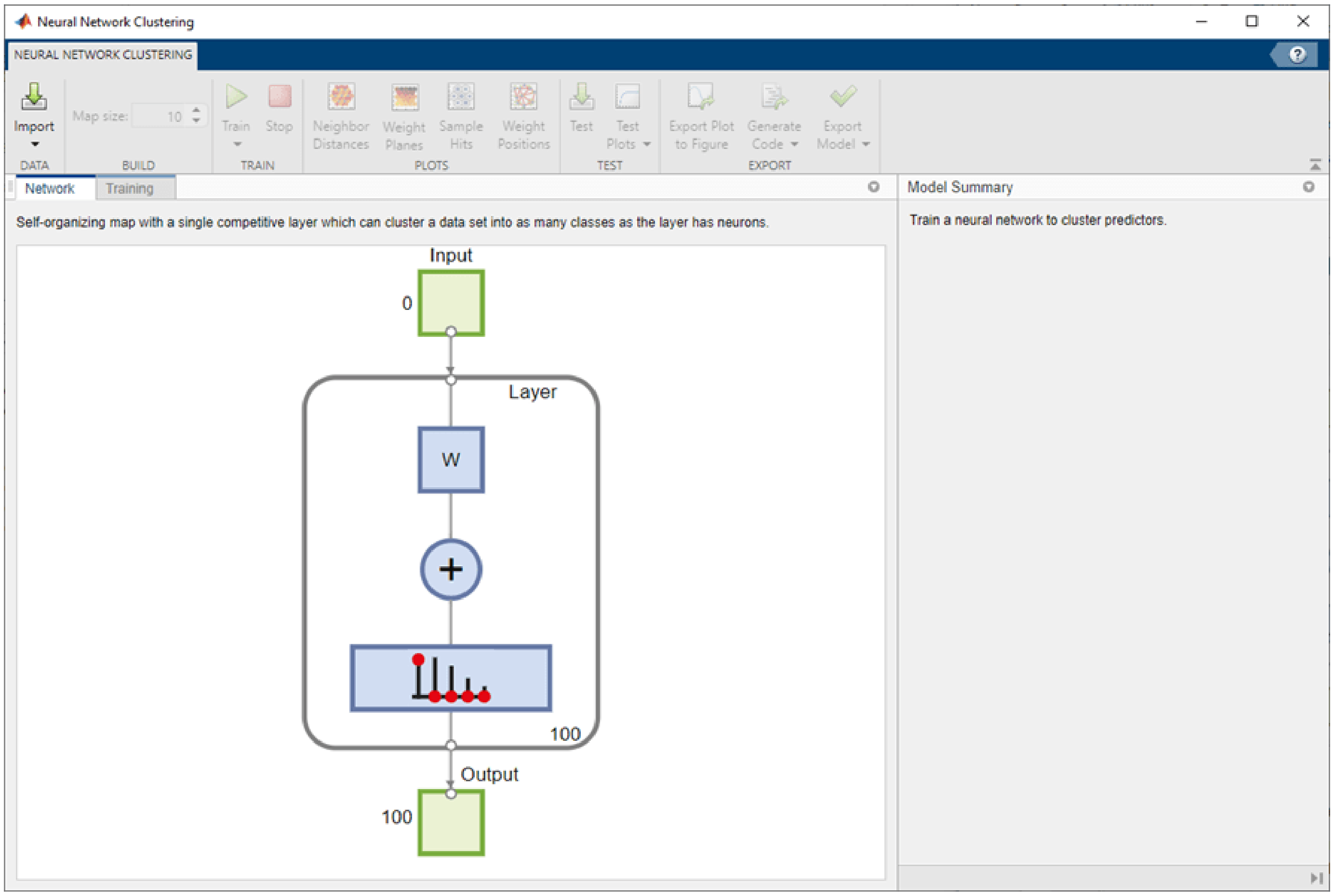
The next section shows how to train a network to cluster data, using the **Neural Net Clustering** app. This example uses an example data set provided with the toolbox.

Cluster Data Using the Neural Net Clustering App

This example shows how to train a shallow neural network to cluster data using the **Neural Net Clustering** app.

Open the **Neural Net Clustering** app using `nctool`.

```
nctool
```



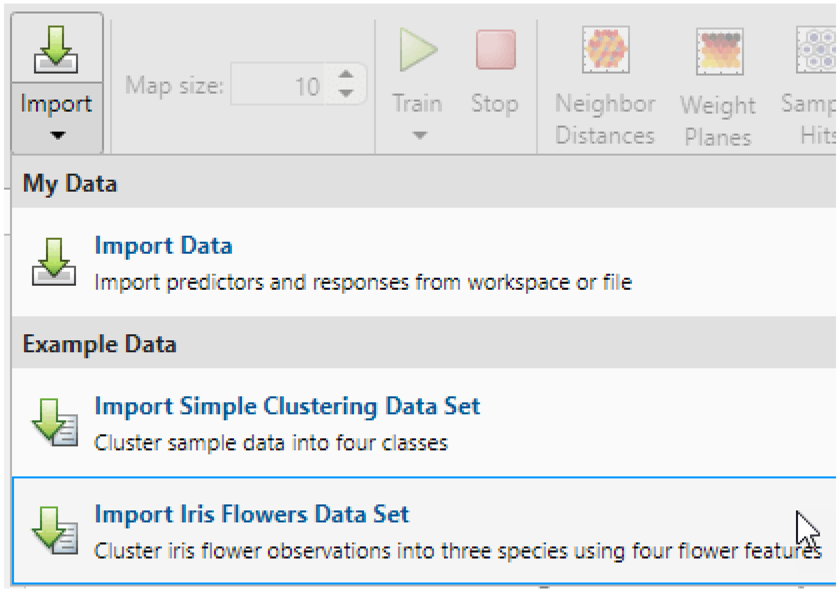
Self-organizing map with a single competitive layer which can cluster a data set into as many classes as the layer has neurons.

Train a neural network to cluster predictors.

Select Data

The **Neural Net Clustering** app has example data to help you get started training a neural network.

To import the example iris flower clustering data, select **Import** > **Import Iris Flowers Data Set**. If you import your own data from file or the workspace, you must specify the predictors and whether the observations are in rows or columns.



My Data

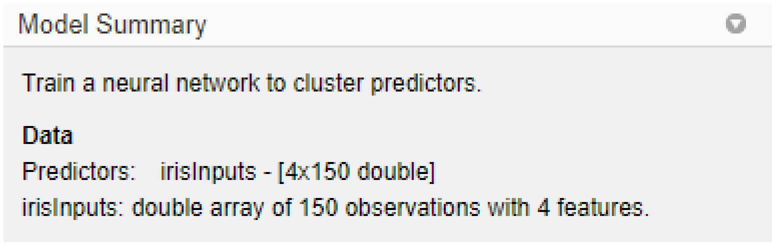
Import Data
Import predictors and responses from workspace or file

Example Data

Import Simple Clustering Data Set
Cluster sample data into four classes

Import Iris Flowers Data Set
Cluster iris flower observations into three species using four flower features

Information about the imported data appears in the **Model Summary**. This data set contains 150 observations, each with four features.


Model Summary

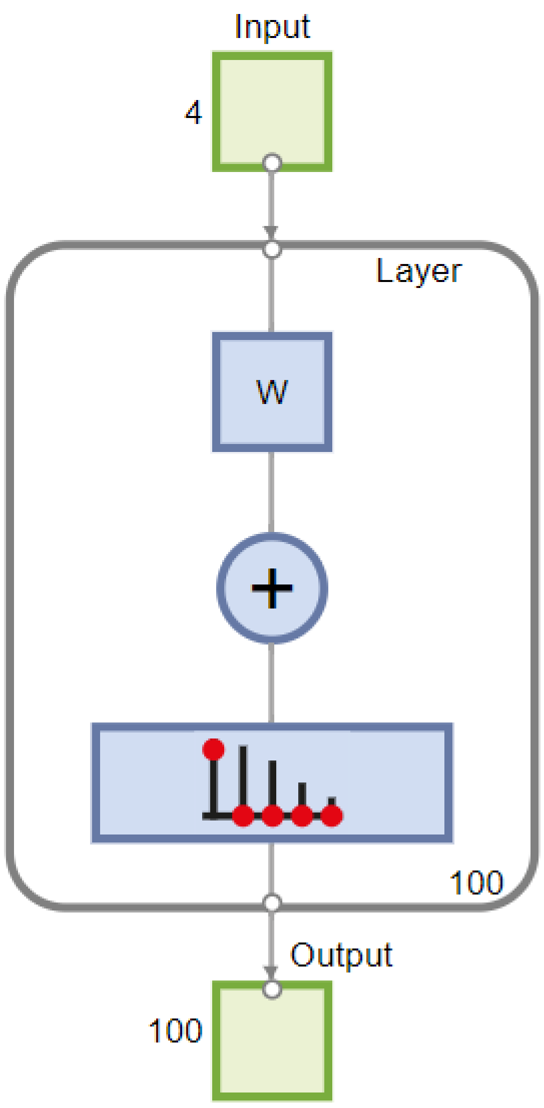
Train a neural network to cluster predictors.

Data
Predictors: irisInputs - [4x150 double]
irisInputs: double array of 150 observations with 4 features.

Create Network

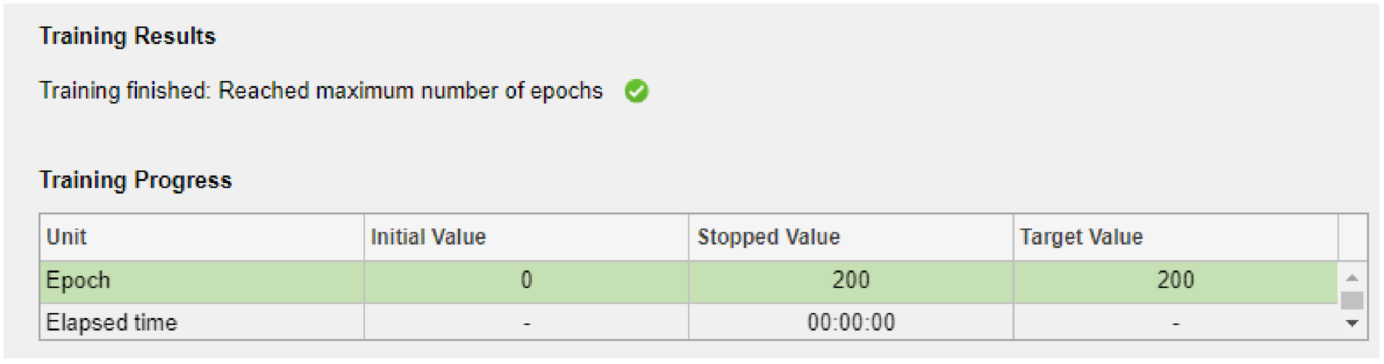
For clustering problems, the self-organizing feature map (SOM) is the most commonly used network. This network has one layer, with neurons organized in a grid. Self-organizing maps learn to cluster data based on similarity. For more information on the SOM, see “Cluster with Self-Organizing Map Neural Network”.

To create the network, specify the map size, this corresponds to the number of rows and columns in the grid. For this example, set the **Map size** value to 10, this corresponds to a grid with 10 rows and 10 columns. The total number of neurons is equal to the number of points in the grid, in this example, the map has 100 neurons. You can see the network architecture in the **Network** pane.



Train Network

To train the network, click **Train**. In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the maximum number of epochs is reached.



Training Results

Training finished: Reached maximum number of epochs ✓

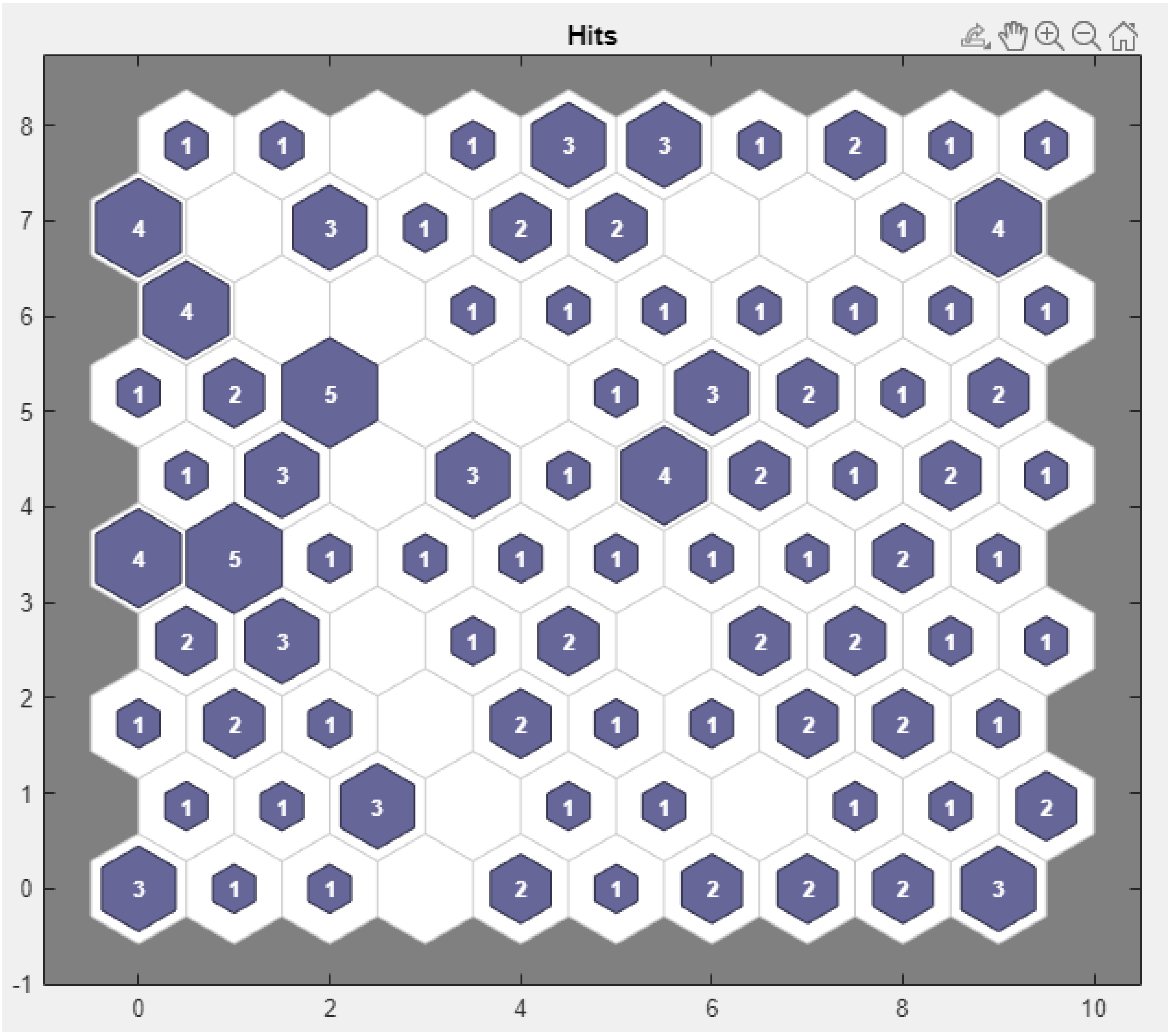
Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200
Elapsed time	-	00:00:00	-

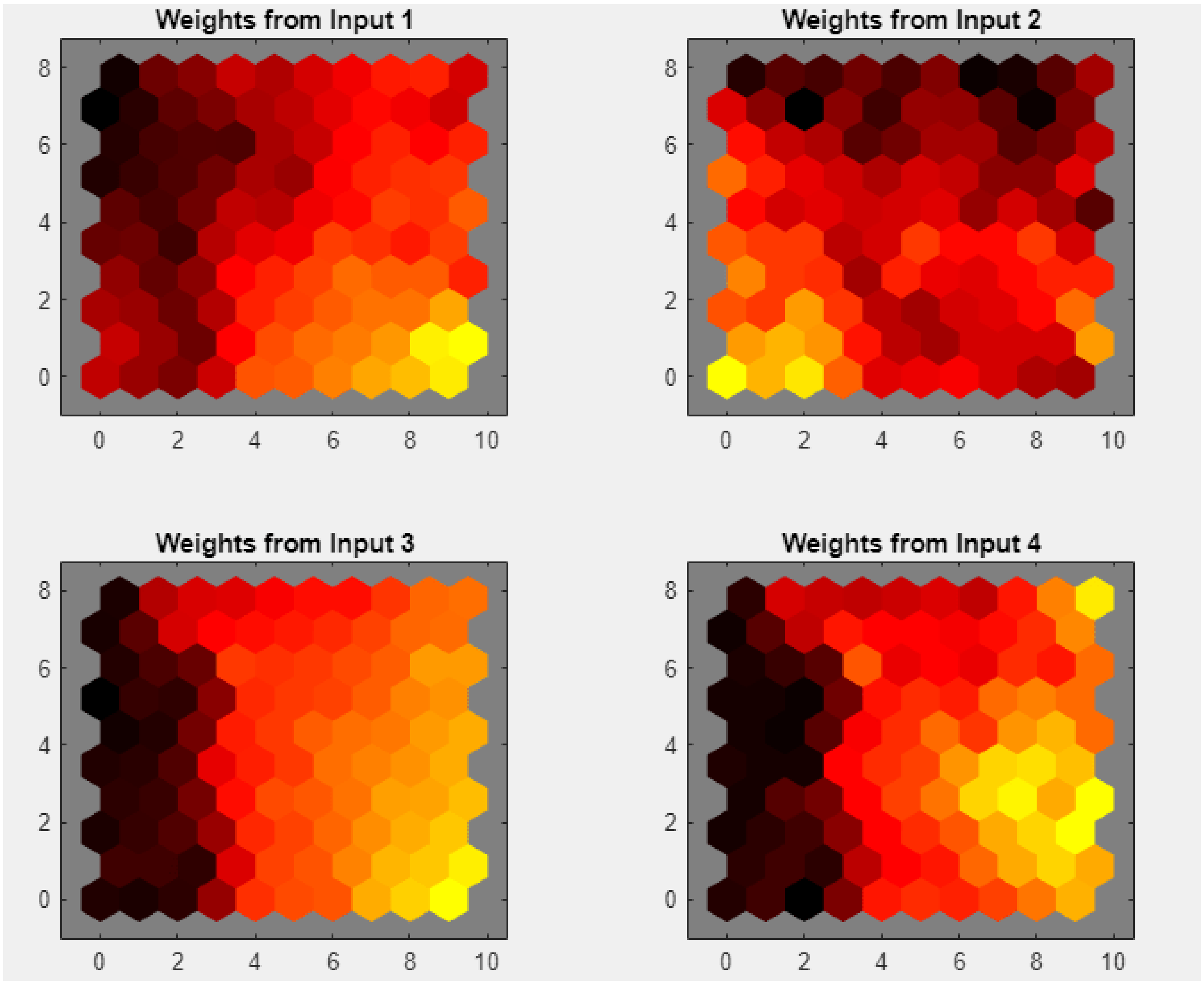
Analyze Results

To analyze the training results, generate plots. For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. The default topology of the SOM is hexagonal.

To plot the SOM Sample Hits, in the **Plots** section, click **Sample Hits**. This figure shows the neuron locations in the topology, and indicates how many of the observations are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated with any neuron is 5. Thus, there are 5 input vectors in that cluster.



Plot the weight planes (also referred to as *component planes*). In the **Plots** section, click **Weight Planes**. This figure shows a weight plane for each element of the input features (four, in this example). The plot shows the weights that connect each input to each of the neurons, with darker colors representing larger weights. If the connection patterns of two features are very similar, you can assume that the features are highly correlated.



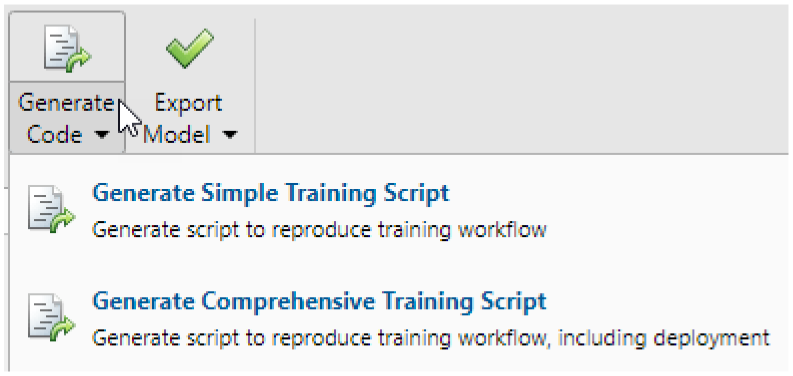
If you are unhappy with the network performance, you can do one of the following:

- Train the network again. Each training will have different initial weights and biases of the network, and can produce an improved network after retraining.
- Increase the number of neurons by increasing the map size.
- Use a larger training data set.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. Generate plots to analyze the additional test results.

Generate Code

Select **Generate Code** > **Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In "Cluster Data Using Command-Line Functions" on page 1-83, you will investigate the generated scripts in more detail.

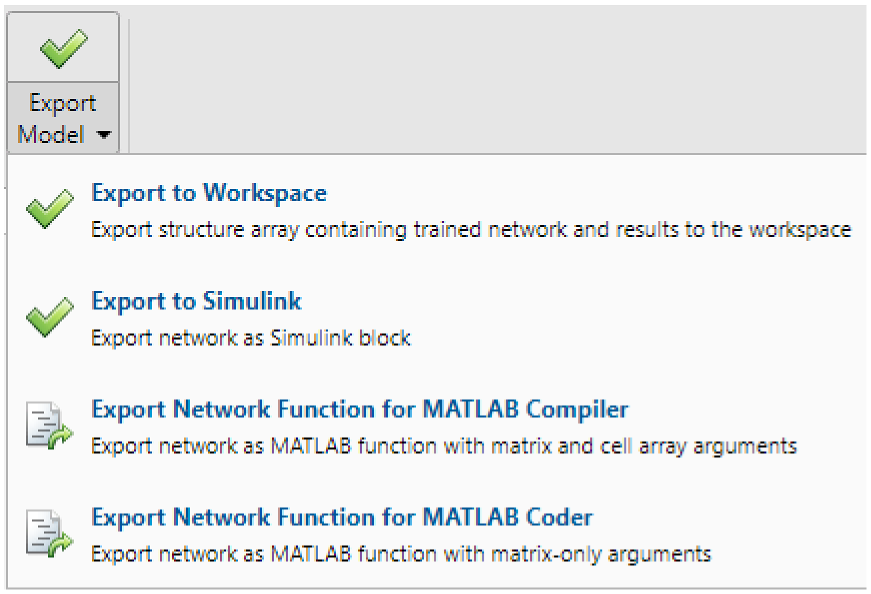


Generate Simple Training Script
Generate script to reproduce training workflow

Generate Comprehensive Training Script
Generate script to reproduce training workflow, including deployment

Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model** > **Export to Workspace**.



Export to Workspace
Export structure array containing trained network and results to the workspace

Export to Simulink
Export network as Simulink block

Export Network Function for MATLAB Compiler
Export network as MATLAB function with matrix and cell array arguments

Export Network Function for MATLAB Coder
Export network as MATLAB function with matrix-only arguments

Cluster Data Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the apps, and then modify them to customize the network training. As an example, look at the simple script that was created in the previous section using the **Neural Net Clustering** app.

```
% Solve a Clustering Problem with a Self-Organizing Map
% Script generated by Neural Clustering app
% Created 21-May-2021 10:15:01
%
% This script assumes these variables are defined:
```

```
%
%   irisInputs - input data.

x = irisInputs;

% Create a Self-Organizing Map
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1 dimension2]);

% Train the Network
[net,tr] = train(net,x);

% Test the Network
y = net(x);

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
%figure, plotsomtop(net)
%figure, plotsomnc(net)
%figure, plotsomnd(net)
%figure, plotsomplanes(net)
%figure, plotsomhits(net,x)
%figure, plotsompos(net,x)
```

You can save the script and then run it from the command line to reproduce the results of the previous training session. You can also edit the script to customize the training process. In this case, follow each step in the script.

Select Data

The script assumes that the predictors are already loaded into the workspace. If the data is not loaded, you can load it as follows:

```
load iris_dataset
```

This command loads the predictors `irisInputs` into the workspace.

This data set is one of the sample data sets that is part of the toolbox. For information about the data sets available, see “Sample Data Sets for Shallow Neural Networks” on page 1-113. You can also see a list of all available data sets by entering the command `help nndatasets`. You can load the variables from any of these data sets using your own variable names. For example, the command

```
x = irisInputs;
```

will load the iris flower predictors into the array `x`.

Create Network

Create a network. For this example, you use a self-organizing map (SOM). This network has one layer, with the neurons organized in a grid. For more information, see “Cluster with Self-Organizing Map Neural Network”. When creating the network with `selforgmap`, you specify the number of rows and columns in the grid.

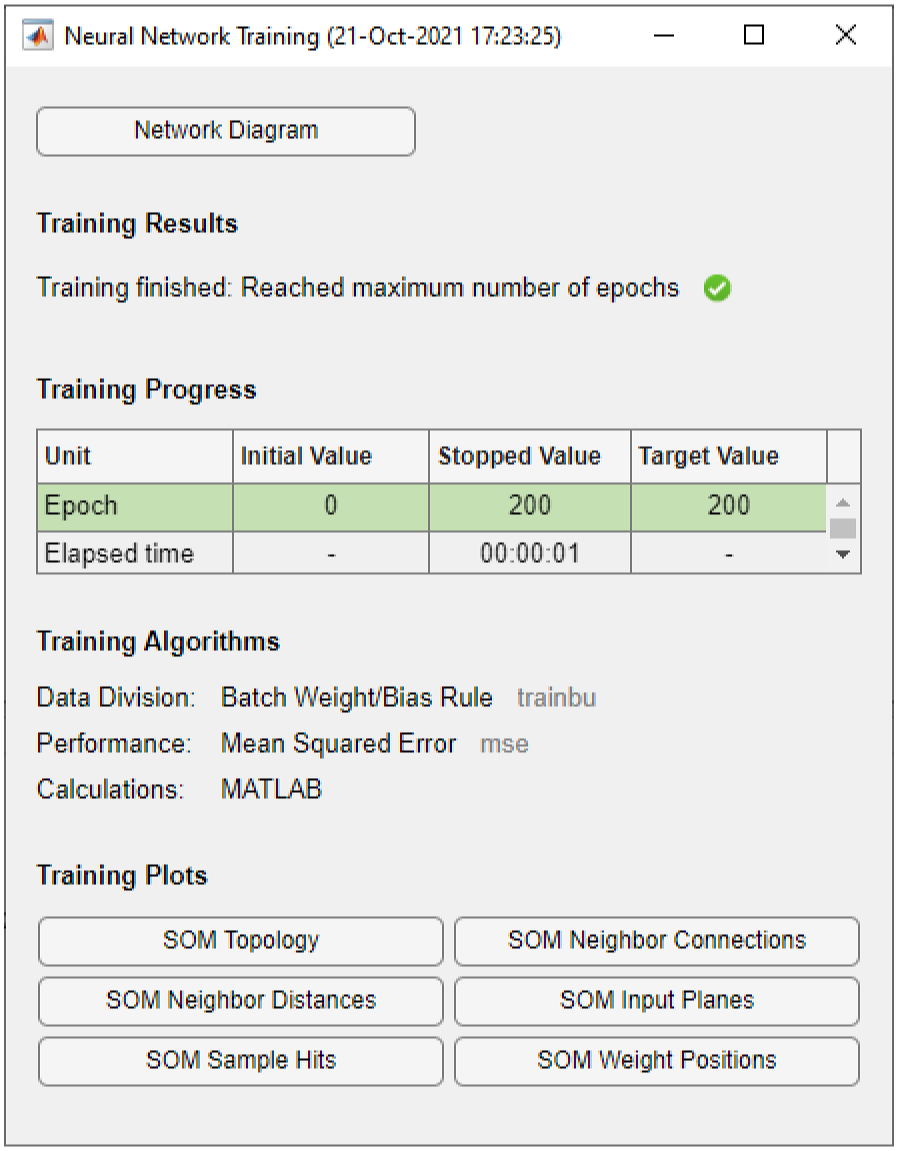
```
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1 dimension2]);
```

Train Network

Train the network. The SOM network uses the default batch SOM algorithm for training.

```
[net,tr] = train(net,x);
```

During training, the training window opens and displays the training progress. You can interrupt training at any point by clicking the stop button ■.


Neural Network Training (21-Oct-2021 17:23:25)

Training Results

Training finished: Reached maximum number of epochs ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200
Elapsed time	-	00:00:01	-

Training Algorithms

Data Division: Batch Weight/Bias Rule trainbu
Performance: Mean Squared Error mse
Calculations: MATLAB

Training Plots

Test Network

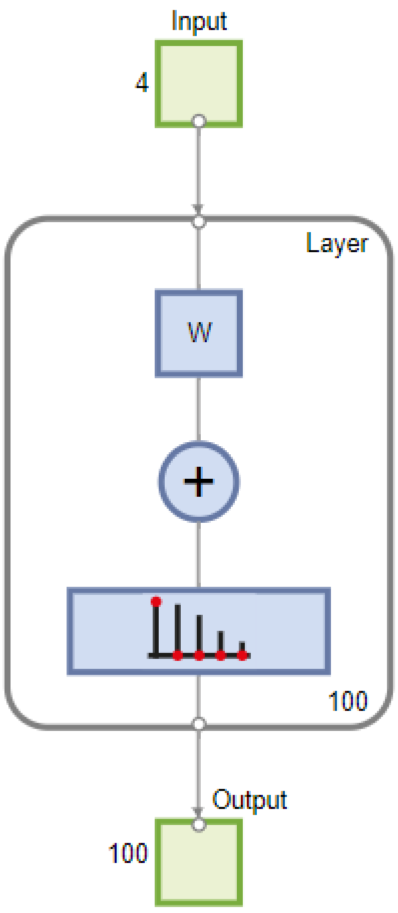
Test the network. After the network has been trained, you can use it to compute the network outputs.

```
y = net(x);
```

View Network

View the network diagram.

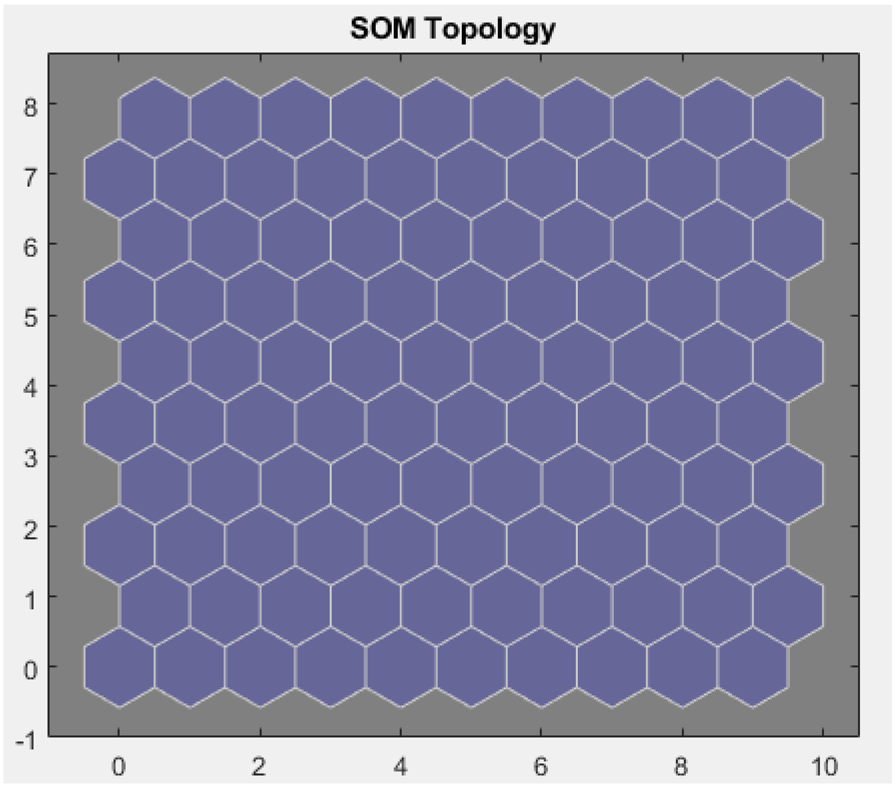
```
view(net)
```



Analyze Results

For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. The default SOM topology is hexagonal; to view it, enter the following commands.

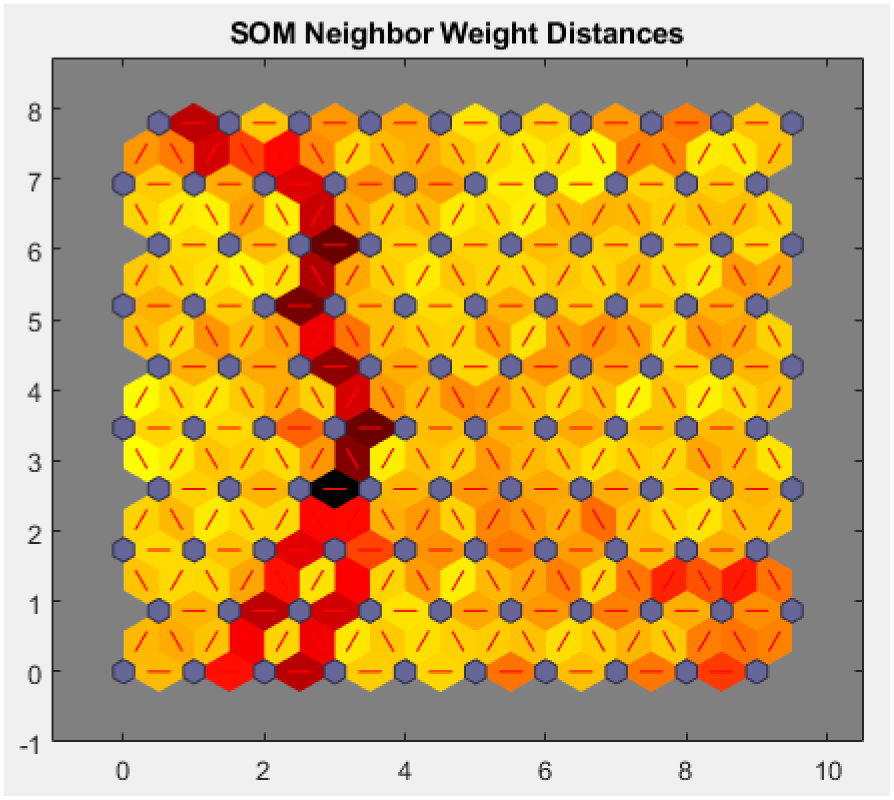
```
figure, plotsomtop(net)
```



In this figure, each of the hexagons represents a neuron. The grid is 10-by-10, so there are a total of 100 neurons in this network. There are four features in each input vector, so the input space is four-dimensional. The weight vectors (cluster centers) fall within this space.

Because this SOM has a two-dimensional topology, you can visualize in two dimensions the relationships among the four-dimensional cluster centers. One visualization tool for the SOM is the *weight distance matrix* (also called the *U-matrix*).

To view the U-matrix, click **SOM Neighbor Distances** in the training window.



In this figure, the blue hexagons represent the neurons. The red lines connect neighboring neurons. The colors in the regions containing the red lines indicate the distances between neurons. The darker colors represent larger distances, and the lighter colors represent smaller distances. A band of dark segments crosses the map. The SOM network appears to have clustered the flowers into two distinct groups.

Next Steps

To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the SOM weight position plot) and watch it animate.
- Plot from the command line with functions such as `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`.

See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `train`

Related Examples

- “Fit Data with a Shallow Neural Network” on page 1-45
- “Classify Patterns with a Shallow Neural Network” on page 1-63
- “Shallow Neural Network Time-Series Prediction and Modeling” on page 1-89

Shallow Neural Network Time-Series Prediction and Modeling

Dynamic neural networks are good at time-series prediction. To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction, see “Multistep Neural Network Prediction”.

Tip For deep learning with time series data, see instead “Sequence Classification Using Deep Learning”.

Suppose, for instance, that you have data from a pH neutralization process. You want to design a network that can predict the pH of a solution in a tank from past values of the pH and past values of the acid and base flow rate into the tank. You have a total of 2001 time steps for which you have those series.

You can solve this problem in two ways:

- Use the **Neural Net Time Series** app, as described in “Fit Time Series Data Using the Neural Net Time Series App” on page 1-90.
- Use command-line functions, as described in “Fit Time Series Data Using Command-Line Functions” on page 1-98.

It is generally best to start with the app, and then use the app to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each of the neural network apps has access to several sample data sets that you can use to experiment with the toolbox (see “Sample Data Sets for Shallow Neural Networks” on page 1-113). If you have a specific problem that you want to solve, you can load your own data into the workspace.

Time Series Networks

You can train a neural network to solve three types of time series problems.

NARX Network

In the first type of time series problem, you would like to predict future values of a time series $y(t)$ from past values of that time series and past values of a second time series $x(t)$. This form of prediction is called nonlinear autoregressive with exogenous (external) input, or NARX (see “Design Time Series NARX Feedback Neural Networks”), and can be written as follows:

$y(t) = f(y(t – 1), \ldots, y(t – d), x(t – 1), \ldots, (t – d))$

Use this model to predict future values of a stock or bond, based on such economic variables as unemployment rates, GDP, etc. You can also use this model for system identification, in which models are developed to represent dynamic systems, such as chemical processes, manufacturing systems, robotics, aerospace vehicles, etc.

NAR Network

In the second type of time series problem, there is only one series involved. The future values of a time series $y(t)$ are predicted only from past values of that series. This form of prediction is called nonlinear autoregressive, or NAR, and can be written as follows:

$y(t) = f(y(t – 1), \ldots, y(t – d))$

You can use this model to predict financial instruments, but without the use of a companion series.

Nonlinear Input-Output Network

The third time series problem is similar to the first type, in that two series are involved, an input series *x*(*t*) and an output series *y*(*t*). Here you want to predict values of *y*(*t*) from previous values of *x*(*t*), but without knowledge of previous values of *y*(*t*). This input/output model can be written as follows:

$y(t) = f(x(t - 1), \ldots, x(t - d))$

The NARX model will provide better predictions than this input-output model, because it uses the additional information contained in the previous values of *y*(*t*). However, there may be some applications in which the previous values of *y*(*t*) would not be available. Those are the only cases where you would want to use the input-output model instead of the NARX model.

Defining a Problem

To define a time series problem for the toolbox, arrange a set of time series predictor vectors as columns in a cell array. Then, arrange another set of time series response vectors (the correct response vectors for each of the predictor vectors) into a second cell array. Additionally, there are cases in which you only need to have a response data set. For example, you can define the following time series problem, in which you want to use previous values of a series to predict the next value:

```
responses = {1 2 3 4 5};
```

The next section shows how to train a network to fit a time series data set, using the **Neural Net Time Series** app. This example uses example data provided with the toolbox.

Fit Time Series Data Using the Neural Net Time Series App

This example shows how to train a shallow neural network to fit time series data using the **Neural Net Time Series** app.

Open the **Neural Net Time Series** app using `ntstool`.

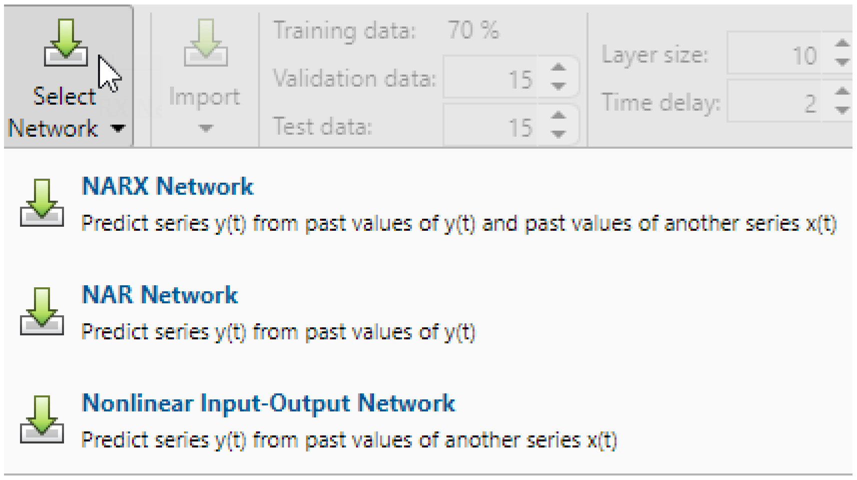
```
ntstool
```

Select Network

You can use the **Neural Net Time Series** app to solve three different kinds of time series problems.

- In the first type of time series problem, you would like to predict future values of a time series *y*(*t*) from past values of that time series and past values of a second time series *x*(*t*). This form of prediction is called nonlinear autoregressive network with exogenous (external) input, or NARX.
- In the second type of time series problem, there is only one series involved. The future values of a time series *y*(*t*) are predicted only from past values of that series. This form of prediction is called nonlinear autoregressive, or NAR.
- The third time series problem is similar to the first type, in that two series are involved, an input series (predictors) *x*(*t*) and an output series (responses) *y*(*t*). Here you want to predict values of *y*(*t*) from previous values of *x*(*t*), but without knowledge of previous values of *y*(*t*).

For this example, use a NARX network. Click **Select Network > NARX Network**.



NARX Network
Predict series y(t) from past values of y(t) and past values of another series x(t)

NAR Network
Predict series y(t) from past values of y(t)

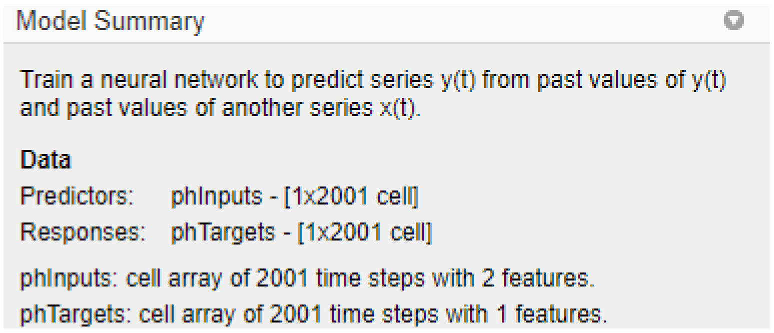
Nonlinear Input-Output Network
Predict series y(t) from past values of another series x(t)

Select Data

The **Neural Net Time Series** app has example data to help you get started training a neural network.

To import example pH neutralization process data, select **Import** > **More Example Data Sets** > **Import pH Neutralization Data Set**. You can use this data set to train a neural network to predict the pH of a solution using acid and base solution flow. If you import your own data from file or the workspace, you must specify the predictors and responses.

Information about the imported data appears in the **Model Summary**. This data set contains 2001 time steps. The predictors have two features (acid and base solution flow) and the responses have a single feature (solution pH).


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:	phInputs - [1x2001 cell]
Responses:	phTargets - [1x2001 cell]

phInputs: cell array of 2001 time steps with 2 features.
phTargets: cell array of 2001 time steps with 1 features.

Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training”.

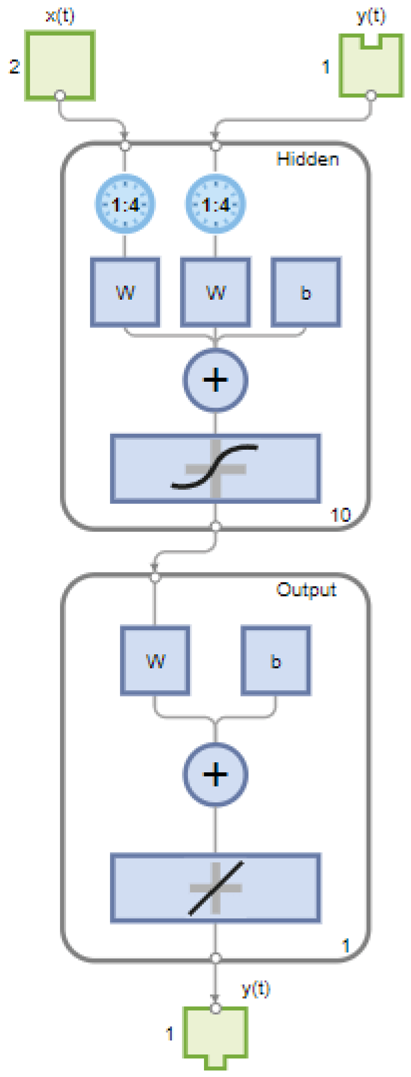
Create Network

The standard NARX network is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This network also uses tapped delay lines to store previous values of the *x*(*t*) and *y*(*t*) sequences. Note that the output of the NARX network, *y*(*t*), is fed back to the input of the network (through delays), since *y*(*t*) is a function of *y*(*t*–1), *y*(*t*–2), . . ., *y*(*t*–*d*). However, for efficient training this feedback loop can be opened.

Because the true output is available during the training of the network, you can use the open-loop architecture shown below, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and therefore a more efficient algorithm can be used for training. This network is discussed in more detail in “Design Time Series NARX Feedback Neural Networks”.

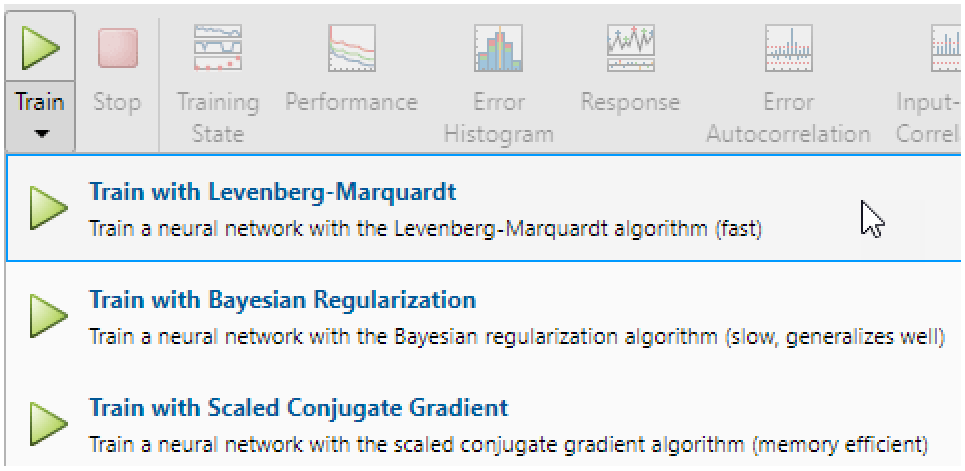
The **Layer size** value defines the number of hidden neurons. Keep the default layer size, `10`. Change the **Time delay** value to `4`. You might want to adjust these numbers if the network training performance is poor.

You can see the network architecture in the **Network** pane.



Train Network

To train the network, select **Train > Train with Levenberg-Marquardt**. This is the default training algorithm and the same as clicking **Train**.



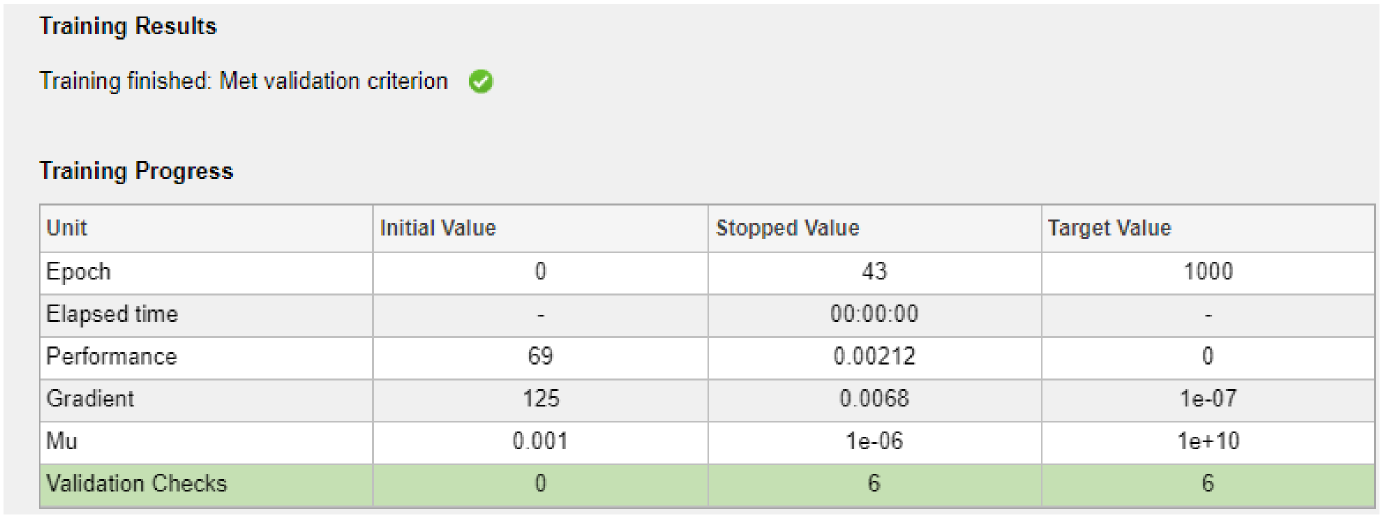
Train with Levenberg-Marquardt
Train a neural network with the Levenberg-Marquardt algorithm (fast)

Train with Bayesian Regularization
Train a neural network with the Bayesian regularization algorithm (slow, generalizes well)

Train with Scaled Conjugate Gradient
Train a neural network with the scaled conjugate gradient algorithm (memory efficient)

Training with Levenberg-Marquardt (`trainlm`) is recommended for most problems. For noisy or small problems, Bayesian Regularization (`trainbr`) can obtain a better solution, at the cost of taking longer. For large problems, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").



Training Results

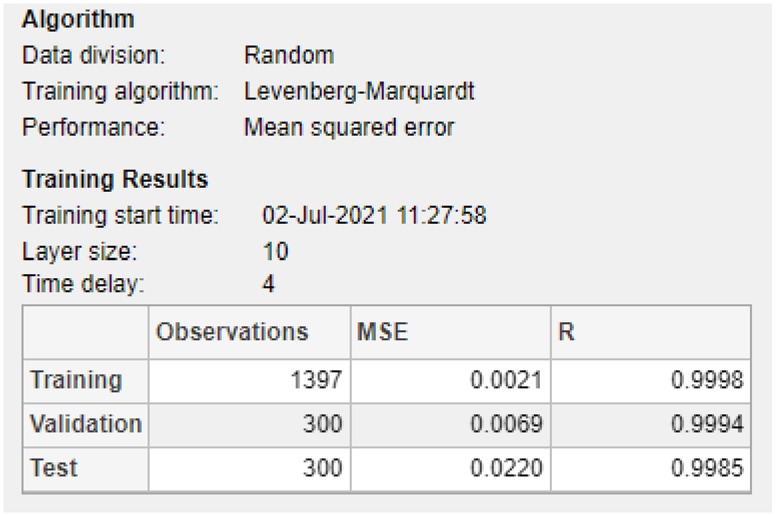
Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	43	1000
Elapsed time	-	00:00:00	-
Performance	69	0.00212	0
Gradient	125	0.0068	1e-07
Mu	0.001	1e-06	1e+10
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.



Algorithm

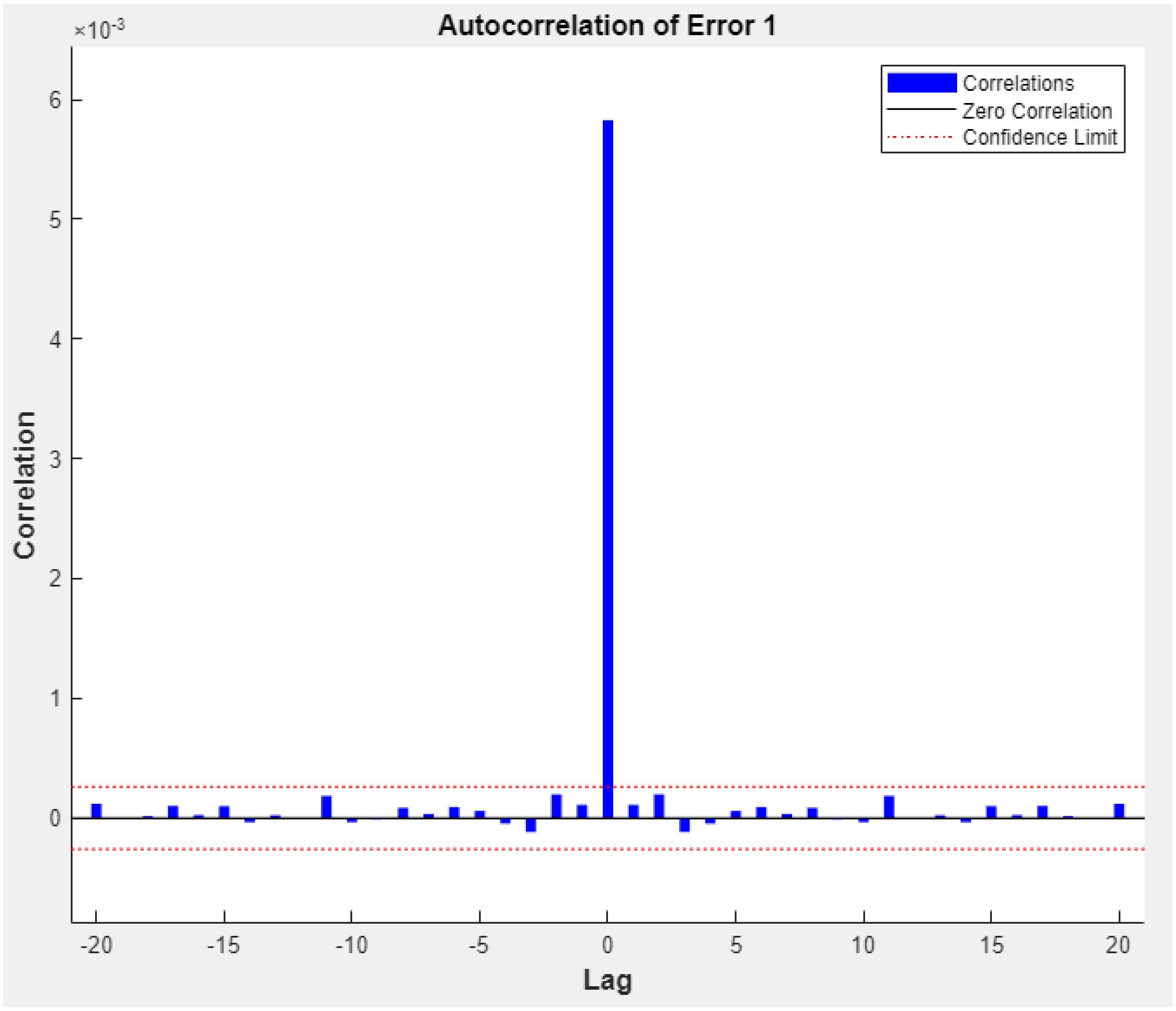
Data division:	Random
Training algorithm:	Levenberg-Marquardt
Performance:	Mean squared error

Training Results

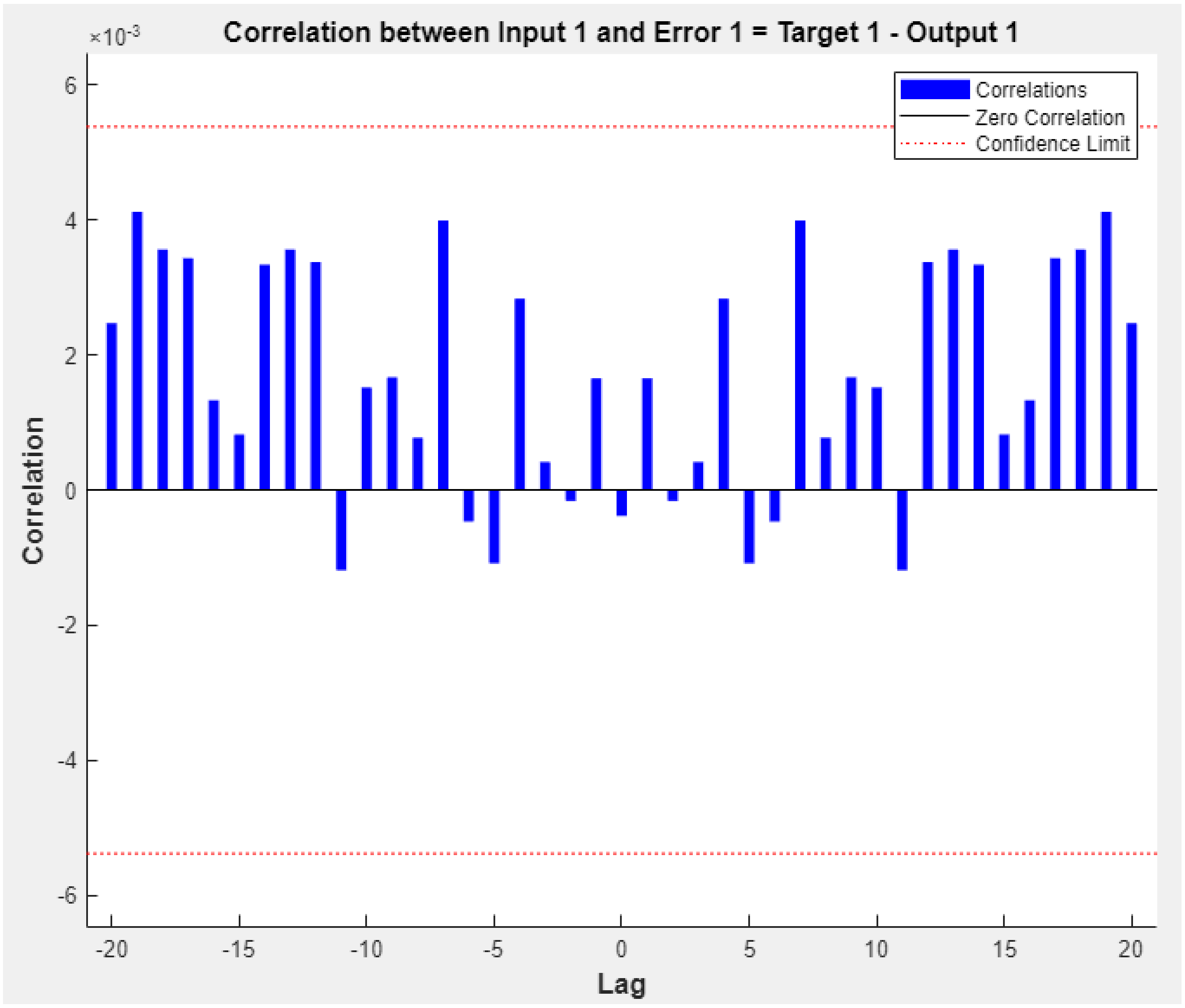
Training start time:	02-Jul-2021 11:27:58
Layer size:	10
Time delay:	4

	Observations	MSE	R
Training	1397	0.0021	0.9998
Validation	300	0.0069	0.9994
Test	300	0.0220	0.9985

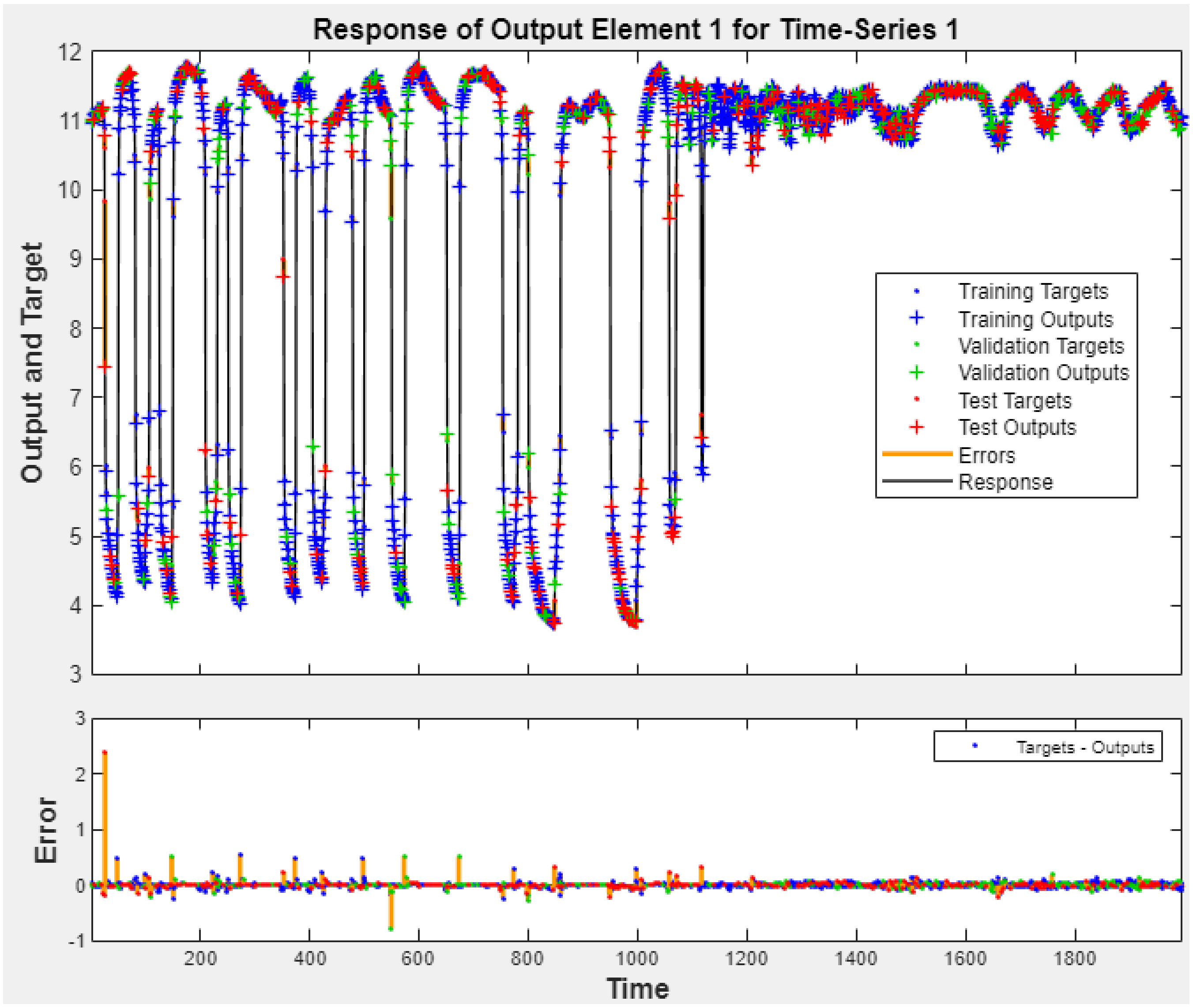
You can further analyze the results by generating plots. To plot the error autocorrelation, in the **Plots** section, click **Error Autocorrelation**. The autocorrelation plot describes how the prediction errors are related in time. For a perfect prediction model, there should only be one nonzero value of the autocorrelation function, and it should occur at zero lag (this is the mean square error). This would mean that the prediction errors were completely uncorrelated with each other (white noise). If there was significant correlation in the prediction errors, then it should be possible to improve the prediction - perhaps by increasing the number of delays in the tapped delay lines. In this case, the correlations, except for the one at zero lag, fall approximately within the 95% confidence limits around zero, so the model seems to be adequate. If even more accurate results were required, you could retrain the network. This will change the initial weights and biases of the network, and may produce an improved network after retraining.



View the input-error cross-correlation plot to obtain additional verification of network performance. In the **Plots** section, click **Input-Error Correlation**. The input-error cross-correlation plot illustrates how the errors are correlated with the input sequence $x(t)$. For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to improve the prediction, perhaps by increasing the number of delays in the tapped delay lines. In this case, most of the correlations fall within the confidence bounds around zero.



In the **Plots** section, click **Response**. This displays the outputs, responses (targets), and errors versus time. It also indicates which time points were selected for training, testing, and validation.



If you are unhappy with the network performance, you can do one of the following:

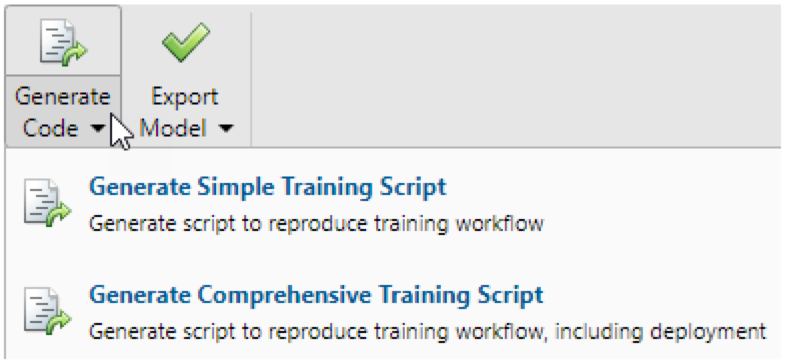
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Decreasing the layer size, and therefore decreasing the number of neurons, can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test data results. You can also generate plots to analyze the additional test data results.

Generate Code

Select **Generate Code** > **Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Fit Time Series Data Using Command-Line Functions” on page 1-98, you will investigate the generated scripts in more detail.

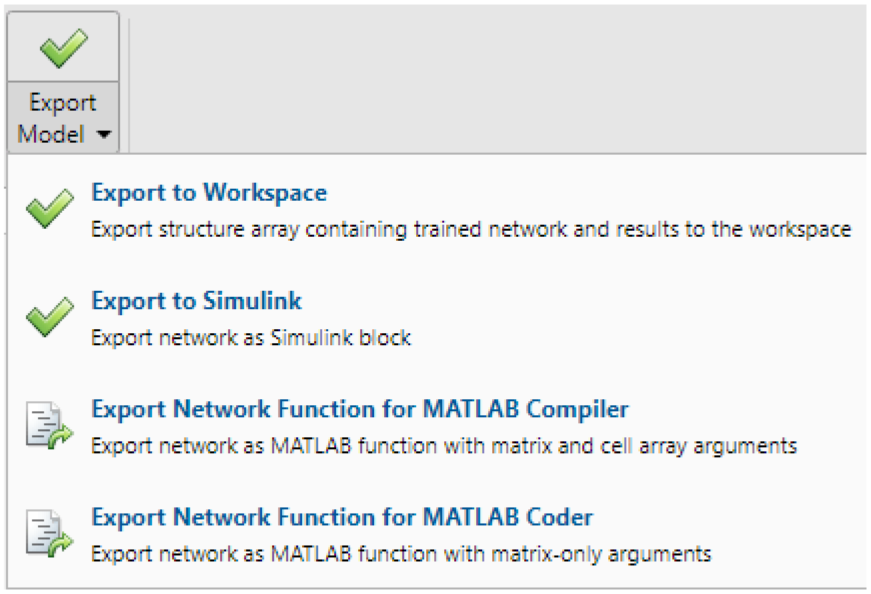


Generate Simple Training Script
Generate script to reproduce training workflow

Generate Comprehensive Training Script
Generate script to reproduce training workflow, including deployment

Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model** > **Export to Workspace**.



Export to Workspace
Export structure array containing trained network and results to the workspace

Export to Simulink
Export network as Simulink block

Export Network Function for MATLAB Compiler
Export network as MATLAB function with matrix and cell array arguments

Export Network Function for MATLAB Coder
Export network as MATLAB function with matrix-only arguments

Fit Time Series Data Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the apps, and then modify them to customize the network training. As an example, look at the simple script that was generated in the previous section using the **Neural Net Time Series** app.

```
% Solve an Autoregression Problem with External Input with a NARX Neural Network
% Script generated by Neural Time Series app
% Created 13-May-2021 17:34:27
%
% This script assumes these variables are defined:
%
%   phInputs - input time series.
%   phTargets - feedback time series.

X = phInputs;
T = phTargets;

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
trainFcn = 'trainlm';  % Levenberg-Marquardt backpropagation.

% Create a Nonlinear Autoregressive Network with External Input
inputDelays = 1:4;
feedbackDelays = 1:4;
hiddenLayerSize = 10;
net = narxnet(inputDelays,feedbackDelays,hiddenLayerSize,'open',trainFcn);

% Prepare the Data for Training and Simulation
% The function PREPARETS prepares timeseries data for a particular network,
% shifting time by the minimum amount to fill input states and layer
% states. Using PREPARETS allows you to keep your original time series data
% unchanged, while easily customizing it for networks with differing
% numbers of delays, with open loop or closed loop feedback modes.
[x,xi,ai,t] = preparets(net,X,{},T);

% Setup Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
[net,tr] = train(net,x,t,xi,ai);

% Test the Network
y = net(x,xi,ai);
e = gsubtract(t,y);
performance = perform(net,t,y)

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
%figure, plotregression(t,y)
%figure, plotresponse(t,y)
%figure, ploterrcorr(e)
%figure, plotinerrcorr(x,e)
```

```
% Closed Loop Network
% Use this network to do multi-step prediction.
% The function CLOSELOOP replaces the feedback input with a direct
% connection from the output layer.
netc = closeloop(net);
netc.name = [net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] = preparets(netc,X,{},T);
yc = netc(xc,xic,aic);
closedLoopPerformance = perform(net,tc,yc)

% Step-Ahead Prediction Network
% For some applications it helps to get the prediction a timestep early.
% The original network returns predicted y(t+1) at the same time it is
% given y(t+1). For some applications such as decision making, it would
% help to have predicted y(t+1) once y(t) is available, but before the
% actual y(t+1) occurs. The network can be made to return its output a
% timestep early by removing one delay so that its minimal tap delay is now
% 0 instead of 1. The new network returns the same outputs as the original
% network, but outputs are shifted left one timestep.
nets = removedelay(net);
nets.name = [net.name ' - Predict One Step Ahead'];
view(nets)
[xs,xis,ais,ts] = preparets(nets,X,{},T);
ys = nets(xs,xis,ais);
stepAheadPerformance = perform(nets,ts,ys)
```

You can save the script, and then run it from the command line to reproduce the results of the previous app session. You can also edit the script to customize the training process. In this case, follow each of the steps in the script.

Select Data

The script assumes that the predictor and response vectors are already loaded into the workspace. If the data is not loaded, you can load it as follows:

```
load ph_dataset
```

This command loads the predictors `pHInputs` and the responses `pHTargets` into the workspace.

This data set is one of the sample data sets that is part of the toolbox. For information about the data sets available, see “Sample Data Sets for Shallow Neural Networks” on page 1-113. You can also see a list of all available data sets by entering the command `help nndatasets`. You can load the variables from any of these data sets using your own variable names. For example, the command

```
[X,T] = ph_dataset;
```

will load the pH data set predictors into the cell array `X` and the pH data set responses into the cell array `T`.

Choose Training Algorithm

Define training algorithm. The network uses the default Levenberg-Marquardt algorithm (`trainlm`) for training.

```
trainFcn = 'trainlm';  % Levenberg-Marquardt backpropagation.
```

For problems in which Levenberg-Marquardt does not produce as accurate results as desired, or for large data problems, consider setting the network training function to Bayesian Regularization (`trainbr`) or Scaled Conjugate Gradient (`trainscg`), respectively, with either

```
net.trainFcn = 'trainbr';
net.trainFcn = 'trainscg';
```

Create Network

Create a network. The NARX network, `narxnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This network has two inputs. One is an external input, and the other is a feedback connection from the network output. After the network has been trained, this feedback connection can be closed, as you will see at a later step. For each of these inputs, there is a tapped delay line to store previous values. To assign the network architecture for a NARX network, you must select the delays associated with each tapped delay line, and also the number of hidden layer neurons. In the following steps, you assign the input delays and the feedback delays to range from 1 to 4 and the number of hidden neurons to be 10.

```
inputDelays = 1:4;
feedbackDelays = 1:4;
hiddenLayerSize = 10;
net = narxnet(inputDelays,feedbackDelays,hiddenLayerSize,'open',trainFcn);
```

Note Increasing the number of neurons and the number of delays requires more computation, and this has a tendency to overfit the data when the numbers are set too high, but it allows the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `narxnet` command.

Prepare Data for Training

Prepare the data for training. When training a network containing tapped delay lines, it is necessary to fill the delays with initial values of the predictors and responses of the network. There is a toolbox command that facilitates this process - `preparets`. This function has three input arguments: the network, the predictors, and the responses. The function returns the initial conditions that are needed to fill the tapped delay lines in the network, and modified predictor and response sequences, where the initial conditions have been removed. You can call the function as follows:

```
[x,xi,ai,t] = preparets(net,X,{},T);
```

Divide Data

Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
```

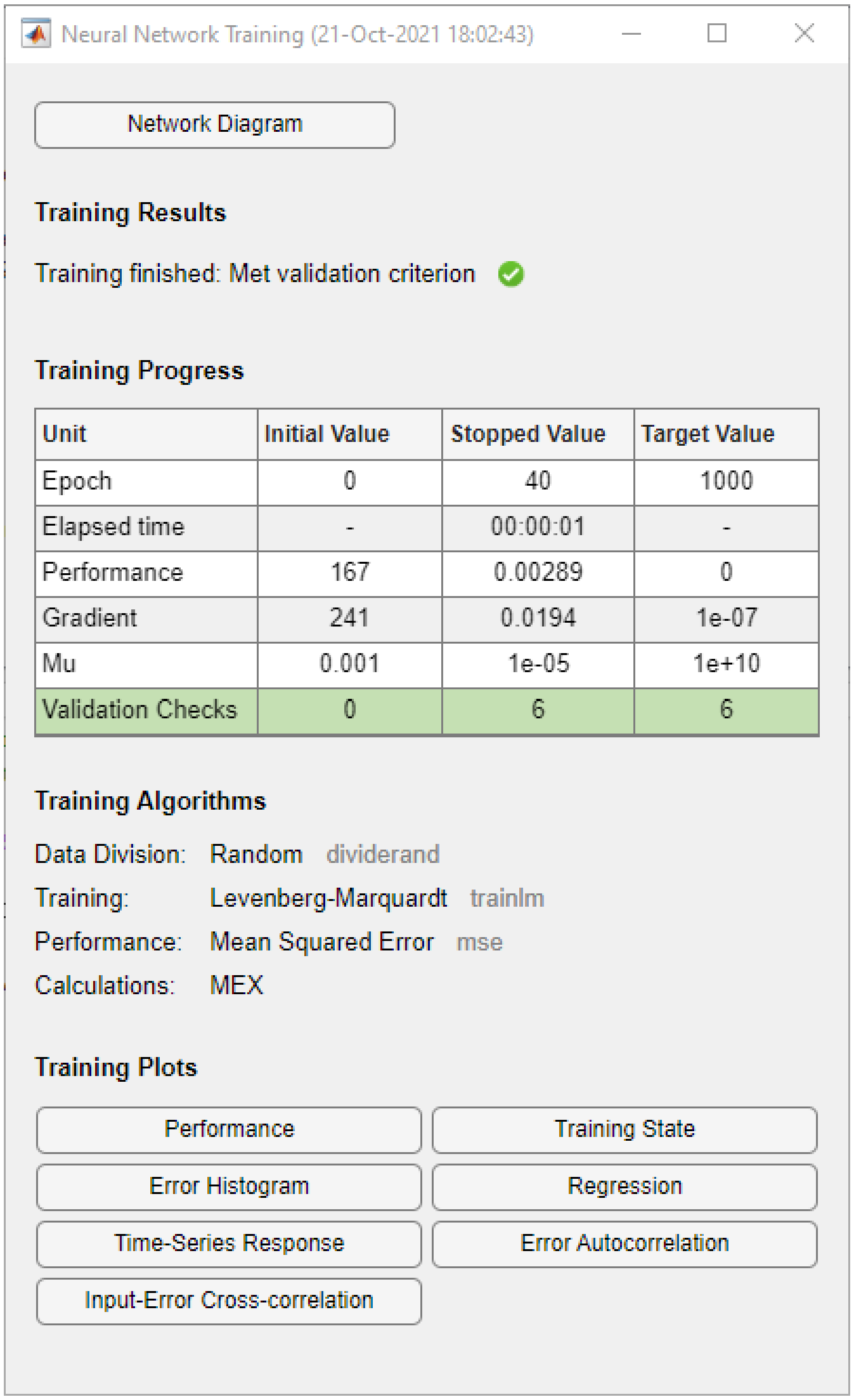
With these settings, the data will be randomly divided, with 70% used for training, 15% for validation, and 15% for testing.

Train Network

Train the network.

```
[net,tr] = train(net,x,t,xi,ai);
```

During training, the following training window opens. This window displays the training progress and allows you to interrupt training at any point by clicking the stop button ■.


Neural Network Training (21-Oct-2021 18:02:43)

Training Results

Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	40	1000
Elapsed time	-	00:00:01	-
Performance	167	0.00289	0
Gradient	241	0.0194	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

This training stopped when the validation error increased consecutively for six iterations.

Test Network

Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors, and overall performance. Note that to simulate a network with tapped delay lines, you need to assign the initial values for these delayed signals. This is done with the input states (`xi`) and the layer states (`ai`) provided by `preparets` at an earlier stage.

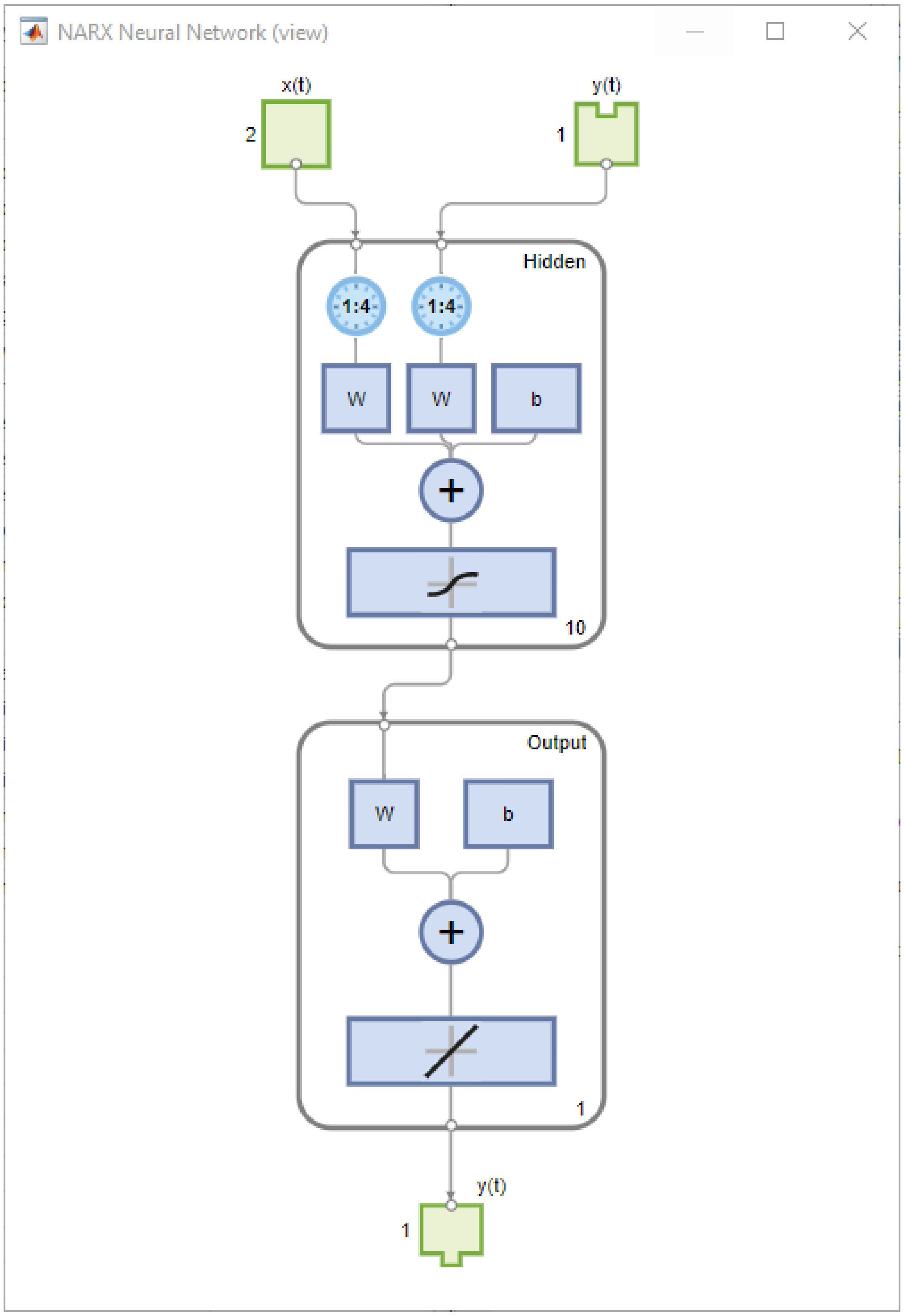
```
y = net(x,xi,ai);
e = gsubtract(t,y);
performance = perform(net,t,y)
```

```
performance =

    0.0042
```

View Network

View the network diagram.

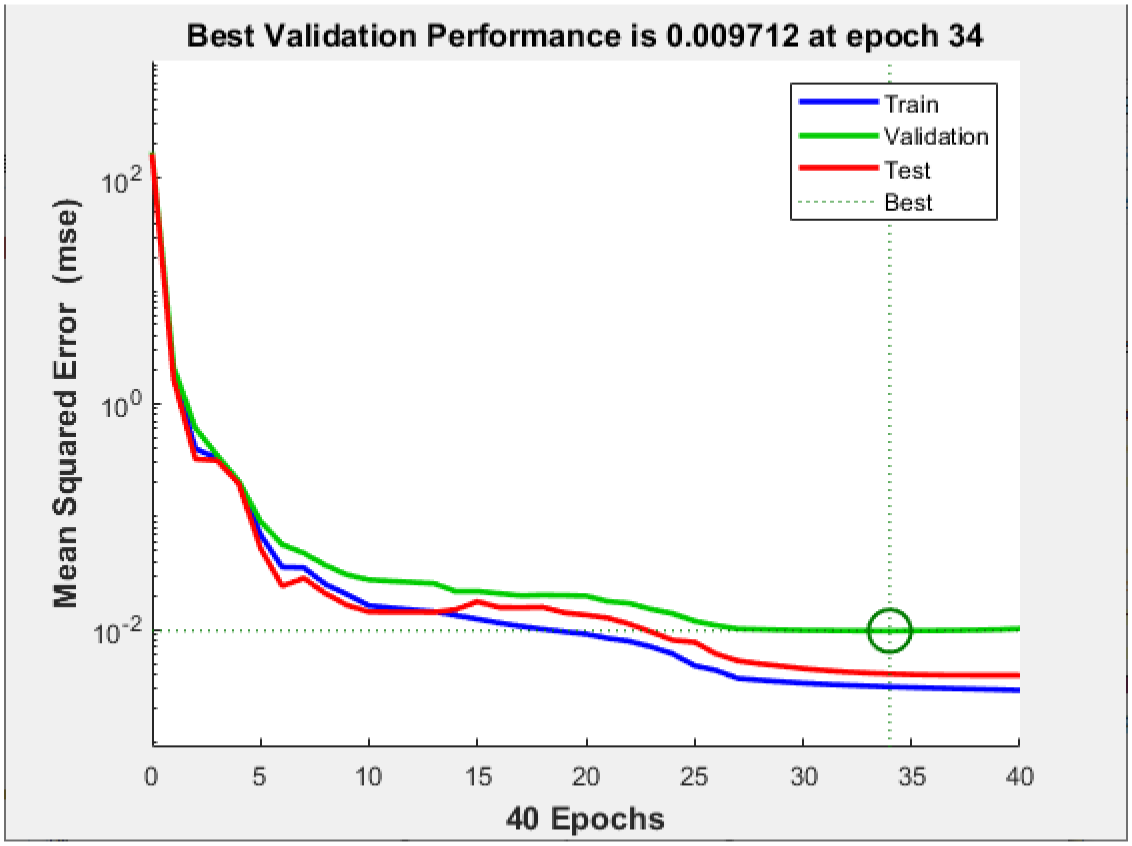
```
view(net)
```



Analyze Results

Plot the performance training record to check for potential overfitting.

```
figure, plotperform(tr)
```



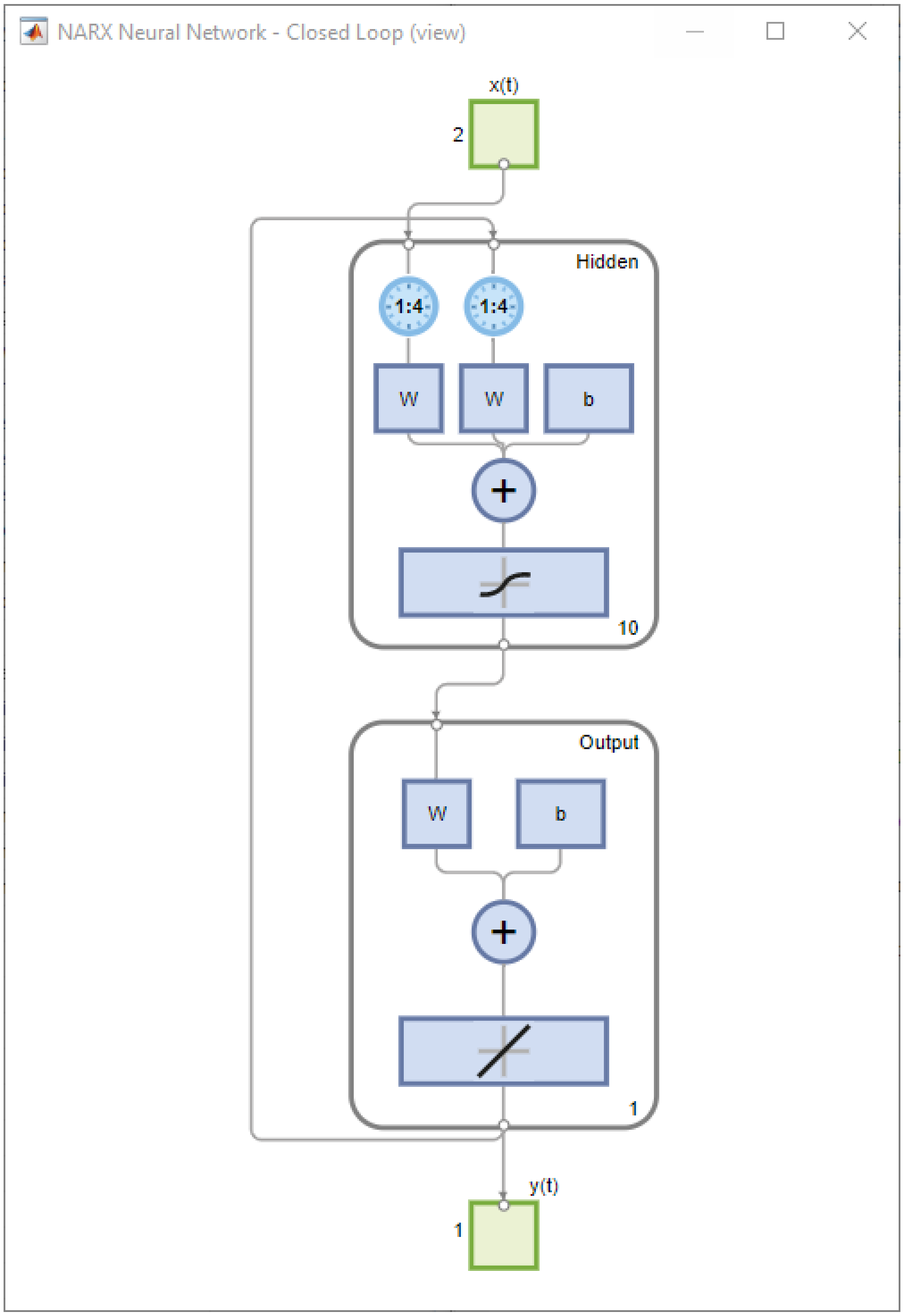
This figure shows that training and validation errors decrease until the highlighted epoch. It does not appear that any overfitting has occurred, because the validation error does not increase before this epoch.

All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) it is transformed to closed loop for multistep-ahead prediction. Likewise, the R values in the **Neural Net Times Series** app are computed based on the open-loop training results.

Closed Loop Network

Close the loop on the NARX network. When the feedback loop is open on the NARX network, it is performing a one-step-ahead prediction. It is predicting the next value of $y(t)$ from previous values of $y(t)$ and $x(t)$. With the feedback loop closed, it can be used to perform multi-step-ahead predictions. This is because predictions of $y(t)$ will be used in place of actual future values of $y(t)$. The following commands can be used to close the loop and calculate closed-loop performance

```
netc = closeloop(net);
netc.name = [net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] = preparets(netc,X,{},T);
yc = netc(xc,xic,aic);
closedLoopPerformance = perform(net,tc,yc)
closedLoopPerformance =

    0.4014
```

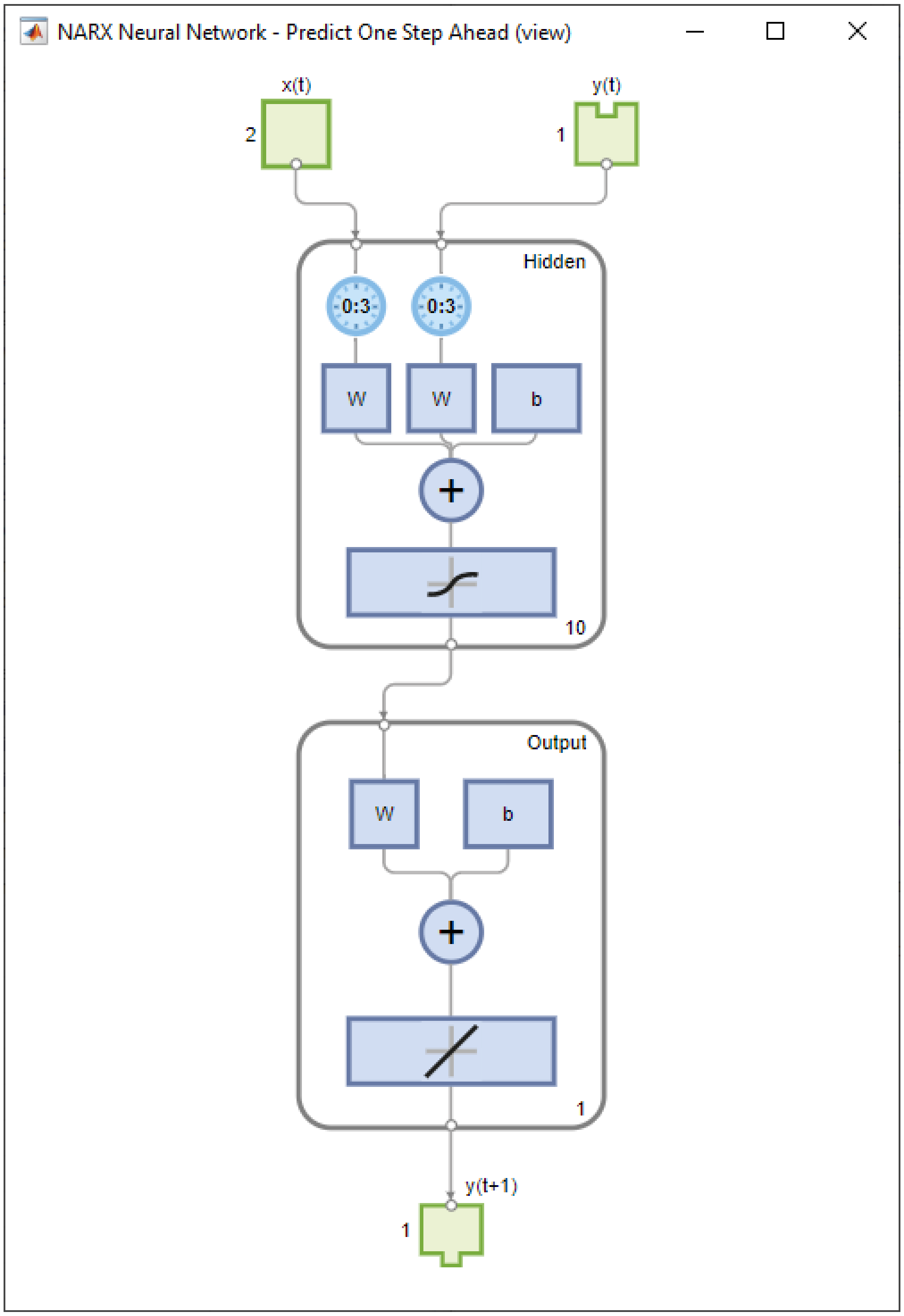


Step-Ahead Prediction Network

Remove a delay from the network, to get the prediction one time step early.

```
nets = removedelay(net);
nets.name = [net.name ' - Predict One Step Ahead'];
view(nets)
[xs,xis,ais,ts] = preparets(nets,X,{},T);
ys = nets(xs,xis,ais);
stepAheadPerformance = perform(nets,ts,ys)
```

```
stepAheadPerformance =

    0.0042
```



From this figure, you can see that the network is identical to the previous open-loop network, except that one delay has been removed from each of the tapped delay lines. The output of the network is then $y(t + 1)$ instead of $y(t)$. This may sometimes be helpful when a network is deployed for certain applications.

Next Steps

If the network performance is not satisfactory, you could try any of these approaches:

- Reset the initial network weights and biases to new values with `init` and train again.
- Increase the number of hidden neurons or the number of delays.
- Use a larger training data set.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Training Algorithms”).

To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the error correlation plot), and watch it animate.
- Plot from the command line with functions such as `plotresponse`, `ploterrcorr` and `plotperform`.

Each time a neural network is trained can result in a different solution due to random initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting”.

See Also
Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `train` | `preparets` | `narxnet` | `closeloop` | `perform` | `removedelay`

Related Examples
- “Fit Data with a Shallow Neural Network” on page 1-45
- “Classify Patterns with a Shallow Neural Network” on page 1-63
- “Cluster Data with a Self-Organizing Map” on page 1-77

Train Shallow Networks on CPUs and GPUs

In this section...

Parallel Computing Toolbox

Tip This topic describes shallow networks. For deep learning, see instead “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud”.

Neural network training and simulation involves many parallel calculations. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can all take advantage of parallel calculations.

Together, Deep Learning Toolbox and Parallel Computing Toolbox enable the multiple CPU cores and GPUs of a single computer to speed up training and simulation of large problems.

The following is a standard single-threaded training and simulation session. (While the benefits of parallelism are most visible for large problems, this example uses a small dataset that ships with Deep Learning Toolbox.)

```
[x, t] = bodyfat_dataset;
net1 = feedforwardnet(10);
net2 = train(net1, x, t);
y = net2(x);
```

Parallel CPU Workers

Intel® processors ship with as many as eight cores. Workstations with two processors can have as many as 16 cores, with even more possible in the future. Using multiple CPU cores in parallel can dramatically speed up calculations.

Start or get the current parallel pool and view the number of workers in the pool.

```
pool = gcp;
pool.NumWorkers
```

An error occurs if you do not have a license for Parallel Computing Toolbox.

When a parallel pool is open, set the `train` function’s `'useParallel'` option to `'yes'` to specify that training and simulation be performed across the pool.

```
net2 = train(net1,x,t,'useParallel','yes');
y = net2(x,'useParallel','yes');
```

GPU Computing

GPUs can have thousands of cores on a single card and are highly efficient on parallel algorithms like neural networks.

Use `gpuDeviceCount` to check whether a supported GPU card is available in your system. Use the function `gpuDevice` to review the currently selected GPU information or to select a different GPU.

```
gpuDeviceCount
gpuDevice
gpuDevice(2) % Select device 2, if available
```

An “Undefined function or variable” error appears if you do not have a license for Parallel Computing Toolbox.

When you have selected the GPU device, set the `train` or `sim` function’s `'useGPU'` option to `'yes'` to perform training and simulation on it.

```
net2 = train(net1,x,t,'useGPU','yes');
y = net2(x,'useGPU','yes');
```

Multiple GPU/CPU Computing

You can use multiple GPUs for higher levels of parallelism.

After opening a parallel pool, set both `'useParallel'` and `'useGPU'` to `'yes'` to harness all the GPUs and CPU cores on a single computer. Each worker associated with a unique GPU uses that GPU. The rest of the workers perform calculations on their CPU core.

```
net2 = train(net1,x,t,'useParallel','yes','useGPU','yes');
y = net2(x,'useParallel','yes','useGPU','yes');
```

For some problems, using GPUs and CPUs together can result in the highest computing speed. For other problems, the CPUs might not keep up with the GPUs, and so using only GPUs is faster. Set `'useGPU'` to `'only'`, to restrict the parallel computing to workers with unique GPUs.

```
net2 = train(net1,x,t,'useParallel','yes','useGPU','only');
y = net2(x,'useParallel','yes','useGPU','only');
```

Cluster Computing with MATLAB Parallel Server

MATLAB Parallel Server allows you to harness all the CPUs and GPUs on a network cluster of computers. To take advantage of a cluster, open a parallel pool with a cluster profile. Use the MATLAB **Home** tab **Environment** area **Parallel** menu to manage and select profiles.

After opening a parallel pool, train the network by calling `train` with the `'useParallel'` and `'useGPU'` options.

```
net2 = train(net1,x,t,'useParallel','yes');
y = net2(x,'useParallel','yes');

net2 = train(net1,x,t,'useParallel','yes','useGPU','only');
y = net2(x,'useParallel','yes','useGPU','only');
```

Load Balancing, Large Problems, and Beyond

For more information on parallel computing with Deep Learning Toolbox, see “Shallow Neural Networks with Parallel and GPU Computing”, which introduces other topics, such as how to manually distribute data sets across CPU and GPU workers to best take advantage of differences in machine speed and memory.

Distributing data manually also allows worker data to load sequentially, so that data sets are limited in size only by the total RAM of a cluster instead of the RAM of a single computer. This lets you apply neural networks to very large problems.

Sample Data Sets for Shallow Neural Networks

The Deep Learning Toolbox contains a number of sample data sets that you can use to experiment with shallow neural networks. To view the data sets that are available, use the following command:

```
help nndatasets
```

```
  Neural Network Datasets
  -----------------------

  Function Fitting, Function approximation and Curve fitting.

  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.
  Once the neural network has fit the data, it forms a generalization of
  the input-output relationship and can be used to generate outputs for
  inputs it was not trained on.

   simplefit_dataset     - Simple fitting dataset.
   abalone_dataset       - Abalone shell rings dataset.
   bodyfat_dataset       - Body fat percentage dataset.
   building_dataset      - Building energy dataset.
   chemical_dataset      - Chemical sensor dataset.
   cho_dataset           - Cholesterol dataset.
   engine_dataset        - Engine behavior dataset.
   vinyl_dataset         - Vinyl bromide dataset.

  ----------

  Pattern Recognition and Classification

  Pattern recognition is the process of training a neural network to assign
  the correct target classes to a set of input patterns.  Once trained the
  network can be used to classify patterns it has not seen before.

   simpleclass_dataset     - Simple pattern recognition dataset.
   cancer_dataset          - Breast cancer dataset.
   crab_dataset            - Crab gender dataset.
   glass_dataset           - Glass chemical dataset.
   iris_dataset            - Iris flower dataset.
   ovarian_dataset         - Ovarian cancer dataset.
   thyroid_dataset         - Thyroid function dataset.
   wine_dataset            - Italian wines dataset.
   digitTrain4DArrayData   - Synthetic handwritten digit dataset for
                             training in form of 4-D array.
   digitTrainCellArrayData - Synthetic handwritten digit dataset for
                             training in form of cell array.
   digitTest4DArrayData    - Synthetic handwritten digit dataset for
                             testing in form of 4-D array.
   digitTestCellArrayData  - Synthetic handwritten digit dataset for
                             testing in form of cell array.
   digitSmallCellArrayData - Subset of the synthetic handwritten digit
                             dataset for training in form of cell array.

  ----------

  Clustering, Feature extraction and Data dimension reduction
```

```
Clustering is the process of training a neural network on patterns
so that the network comes up with its own classifications according
to pattern similarity and relative topology.  This is useful for gaining
insight into data, or simplifying it before further processing.

 simplecluster_dataset - Simple clustering dataset.

The inputs of fitting or pattern recognition datasets may also clustered.

----------

Input-Output Time-Series Prediction, Forecasting, Dynamic modeling
Nonlinear autoregression, System identification and Filtering

Input-output time series problems consist of predicting the next value
of one time series given another time series. Past values of both series
(for best accuracy), or only one of the series (for a simpler system)
may be used to predict the target series.

 simpleseries_dataset  - Simple time series prediction dataset.
 simplenarx_dataset    - Simple time series prediction dataset.
 exchanger_dataset     - Heat exchanger dataset.
 maglev_dataset        - Magnetic levitation dataset.
 ph_dataset            - Solution PH dataset.
 pollution_dataset     - Pollution mortality dataset.
 refmodel_dataset      - Reference model dataset
 robotarm_dataset      - Robot arm dataset
 valve_dataset         - Valve fluid flow dataset.

----------

Single Time-Series Prediction, Forecasting, Dynamic modeling,
Nonlinear autoregression, System identification, and Filtering

Single time series prediction involves predicting the next value of
a time series given its past values.

 simplenar_dataset     - Simple single series prediction dataset.
 chickenpox_dataset    - Monthly chickenpox instances dataset.
 ice_dataset           - Global ice volume dataset.
 laser_dataset         - Chaotic far-infrared laser dataset.
 oil_dataset           - Monthly oil price dataset.
 river_dataset         - River flow dataset.
 solar_dataset         - Sunspot activity dataset
```

Notice that all of the data sets have file names of the form `name_dataset`. Inside these files will be the arrays `nameInputs` and `nameTargets`. You can load a data set into the workspace with a command such as

```
load simplefit_dataset
```

This will load `simplefitInputs` and `simplefitTargets` into the workspace. If you want to load the input and target arrays into different names, you can use a command such as

```
[x,t] = simplefit_dataset;
```

This will load the inputs and targets into the arrays `x` and `t`. You can get a description of a data set with a command such as

```
help maglev_dataset
```

See Also

Neural Net Fitting | **Neural Net Clustering** | **Neural Net Pattern Recognition** | **Neural Net Time Series**

Related Examples

- "Fit Data with a Shallow Neural Network" on page 1-45
- "Classify Patterns with a Shallow Neural Network" on page 1-63
- "Cluster Data with a Self-Organizing Map" on page 1-77
- "Shallow Neural Network Time-Series Prediction and Modeling" on page 1-89

Shallow Neural Networks Glossary

ADALINE	Acronym for a linear neuron: ADAptive LINear Element.
adaption	Training method that proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.
adaptive filter	Network that contains delays and whose weights are adjusted after each new input vector is presented. The network adapts to changes in the input signal properties if such occur. This kind of filter is used in long distance telephone lines to cancel echoes.
adaptive learning rate	Learning rate that is adjusted according to an algorithm during training to minimize training time.
architecture	Description of the number of the layers in a neural network, each layer's transfer function, the number of neurons per layer, and the connections between layers.
backpropagation learning rule	Learning rule in which weights and biases are adjusted by error-derivative (delta) vectors backpropagated through the network. Backpropagation is commonly applied to feedforward multilayer networks. Sometimes this rule is called the *generalized delta rule*.
backtracking search	Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in performance is obtained.
batch	Matrix of input (or target) vectors applied to the network simultaneously. Changes to the network weights and biases are made just once for the entire set of vectors in the input matrix. (The term *batch* is being replaced by the more descriptive expression “concurrent vectors.”)
batching	Process of presenting a set of input vectors for simultaneous calculation of a matrix of output vectors and/or new weights and biases.
Bayesian framework	Assumes that the weights and biases of the network are random variables with specified distributions.
BFGS quasi-Newton algorithm	Variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.
bias	Neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.
bias vector	Column vector of bias values for a layer of neurons.
Brent's search	Linear search that is a hybrid of the golden section search and a quadratic interpolation.

cascade-forward network	Layered network in which each layer only receives inputs from previous layers.
Charalambous' search	Hybrid line search that uses a cubic interpolation together with a type of sectioning.
classification	Association of an input vector with a particular target vector.
competitive layer	Layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector.
competitive learning	Unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.
competitive transfer function	Accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the winner, the neuron associated with the most positive element of the net input $\mathbf{n}$.
concurrent input vectors	Name given to a matrix of input vectors that are to be presented to a network simultaneously. All the vectors in the matrix are used in making just one set of changes in the weights and biases.
conjugate gradient algorithm	In the conjugate gradient algorithms, a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.
connection	One-way link between neurons in a network.
connection strength	Strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.
cycle	Single presentation of an input vector, calculation of output, and new weights and biases.
dead neuron	Competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.
decision boundary	Line, determined by the weight and bias vectors, for which the net input n is zero.
delta rule	See **Widrow-Hoff learning rule**.
delta vector	The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.
distance	Distance between neurons, calculated from their positions with a distance function.
distance function	Particular way of calculating distance, such as the Euclidean distance between two vectors.

early stopping	Technique based on dividing the data into three subsets. The first subset is the training set, used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design.
epoch	Presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.
error jumping	Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.
error ratio	Training parameter used with adaptive learning rate and momentum training of backpropagation networks.
error vector	Difference between a network's output vector in response to an input vector and an associated target output vector.
feedback network	Network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.
feedforward network	Layered network in which each layer only receives inputs from previous layers.
Fletcher-Reeves update	Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.
function approximation	Task performed by a network trained to respond to inputs with an approximation of a desired function.
generalization	Attribute of a network whose output for a new input vector tends to be close to outputs for similar input vectors in its training set.
generalized regression network	Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.
global minimum	Lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.
golden section search	Linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.
gradient descent	Process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

hard-limit transfer function	Transfer function that maps inputs greater than or equal to 0 to 1, and all other values to 0.
Hebb learning rule	Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and postweight neurons.
hidden layer	Layer of a network that is not connected to the network output (for instance, the first layer of a two-layer feedforward network).
home neuron	Neuron at the center of a neighborhood.
hybrid bisection-cubic search	Line search that combines bisection and cubic interpolation.
initialization	Process of setting the network weights and biases to their original values.
input layer	Layer of neurons receiving inputs directly from outside the network.
input space	Range of all possible input vectors.
input vector	Vector presented to the network.
input weight vector	Row vector of weights going to a neuron.
input weights	Weights connecting network inputs to layers.
Jacobian matrix	Contains the first derivatives of the network errors with respect to the weights and biases.
Kohonen learning rule	Learning rule that trains a selected neuron's weight vectors to take on the values of the current input vector.
layer	Group of neurons having connections to the same inputs and sending outputs to the same destinations.
layer diagram	Network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias, and weight matrices are shown. Individual neurons and connections are not shown.
layer weights	Weights connecting layers to other layers. Such weights need to have nonzero delays if they form a recurrent connection (i.e., a loop).
learning	Process by which weights and biases are adjusted to achieve some desired network behavior.
learning rate	Training parameter that controls the size of weight and bias changes during learning.
learning rule	Method of deriving the next changes that might be made in a network *or* a procedure for modifying the weights and biases of a network.

Levenberg-Marquardt	Algorithm that trains a neural network 10 to 100 times faster than the usual gradient descent backpropagation method. It always computes the approximate Hessian matrix, which has dimensions *n*-by-*n*.
line search function	Procedure for searching along a given search direction (line) to locate the minimum of the network performance.
linear transfer function	Transfer function that produces its input as its output.
link distance	Number of links, or steps, that must be taken to get to the neuron under consideration.
local minimum	Minimum of a function over a limited range of input values. A local minimum might not be the global minimum.
log-sigmoid transfer function	Squashing function of the form shown below that maps the input to the interval (0,1). (The toolbox function is `logsig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

Manhattan distance	The Manhattan distance between two vectors **x** and **y** is calculated as `D = sum(abs(x-y))`
maximum performance increase	Maximum amount by which the performance is allowed to increase in one iteration of the variable learning rate training algorithm.
maximum step size	Maximum step size allowed during a linear search. The magnitude of the weight vector is not allowed to increase by more than this maximum step size in one iteration of a training algorithm.
mean square error function	Performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.
momentum	Technique often used to make it less likely for a backpropagation network to get caught in a shallow minimum.
momentum constant	Training parameter that controls how much momentum is used.
mu parameter	Initial value for the scalar µ.
neighborhood	Group of neurons within a specified distance of a particular neuron. The neighborhood is specified by the indices for all the neurons that lie within a radius *d* of the winning neuron *i**: $Ni(d) = \{j, d_{ij} \leq d\}$
net input vector	Combination, in a layer, of all the layer's weighted input vectors with its bias.
neuron	Basic processing element of a neural network. Includes weights and bias, a summing junction, and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

neuron diagram	Network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.
ordering phase	Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.
output layer	Layer whose output is passed to the world outside the network.
output vector	Output of a neural network. Each element of the output vector is the output of a neuron.
output weight vector	Column vector of weights coming from a neuron or input. (See also **outstar learning rule**.)
outstar learning rule	Learning rule that trains a neuron's (or input's) output weight vector to take on the values of the current output vector of the postweight layer. Changes in the weights are proportional to the neuron's output.
overfitting	Case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.
pass	Each traverse through all the training input and target vectors.
pattern	A vector.
pattern association	Task performed by a network trained to respond with the correct output vector for each input vector presented.
pattern recognition	Task performed by a network trained to respond when an input vector close to a learned vector is presented. The network “recognizes” the input as one of the original target vectors.
perceptron	Single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.
perceptron learning rule	Learning rule for training single-layer hard-limit networks. It is guaranteed to result in a perfectly functioning network in finite time, given that the network is capable of doing so.
performance	Behavior of a network.
performance function	Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type `help nnperformance` for a list of performance functions.
Polak-Ribiére update	Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.
positive linear transfer function	Transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

postprocessing	Converts normalized outputs back into the same units that were used for the original targets.
Powell-Beale restarts	Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient.
preprocessing	Transformation of the input or target data before it is presented to the neural network.
principal component analysis	Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.
quasi-Newton algorithm	Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.
radial basis networks	Neural network that can be designed directly by fitting special response elements where they will do the most good.
radial basis transfer function	The transfer function for a radial basis neuron is

$$radbas(n) = e^{-n^2}$$

regularization	Modification of the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights.
resilient backpropagation	Training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid squashing transfer functions.
saturating linear transfer function	Function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is `satlin`.)
scaled conjugate gradient algorithm	Avoids the time-consuming line search of the standard conjugate gradient algorithm.
sequential input vectors	Set of vectors that are to be presented to a network one after the other. The network weights and biases are adjusted on the presentation of each input vector.
sigma parameter	Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.
sigmoid	Monotonic S-shaped function that maps numbers in the interval $(-\infty,\infty)$ to a finite interval such as (-1,+1) or (0,1).
simulation	Takes the network input **p**, and the network object `net`, and returns the network outputs **a**.
spread constant	Distance an input vector must be from a neuron's weight vector to produce an output of 0.5.

squashing function	Monotonically increasing function that takes input values between -∞ and +∞ and returns values in a finite interval.
star learning rule	Learning rule that trains a neuron's weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron's output.
sum-squared error	Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.
supervised learning	Learning process in which changes in a network's weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.
symmetric hard-limit transfer function	Transfer that maps inputs greater than or equal to 0 to +1, and all other values to -1.
symmetric saturating linear transfer function	Produces the input as its output as long as the input is in the range -1 to 1. Outside that range the output is -1 and +1, respectively.
tan-sigmoid transfer function	Squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is `tansig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

tapped delay line	Sequential set of delays with outputs available at each delay output.
target vector	Desired output vector for a given input vector.
test vectors	Set of input vectors (not used directly in training) that is used to test the trained network.
topology functions	Ways to arrange the neurons in a grid, box, hexagonal, or random topology.
training	Procedure whereby a network is adjusted to do a particular job. Commonly viewed as an offline job, as opposed to an adjustment made during each time interval, as is done in adaptive training.
training vector	Input and/or target vector used to train a network.
transfer function	Function that maps a neuron's (or layer's) net output **n** to its actual output.
tuning phase	Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.
underdetermined system	System that has more variables than constraints.
unsupervised learning	Learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly

	changes are a function of the current network input vectors, output vectors, and previous weights and biases.
update	Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.
validation vectors	Set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.
weight function	Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.
weight matrix	Matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix `W` refers to the connection strength from input `j` to neuron `i`.
weighted input vector	Result of applying a weight to a layer's input, whether it is a network input or the output of another layer.
Widrow-Hoff learning rule	Learning rule used to train single-layer linear networks. This rule is the predecessor of the backpropagation rule and is sometimes referred to as the delta rule.