

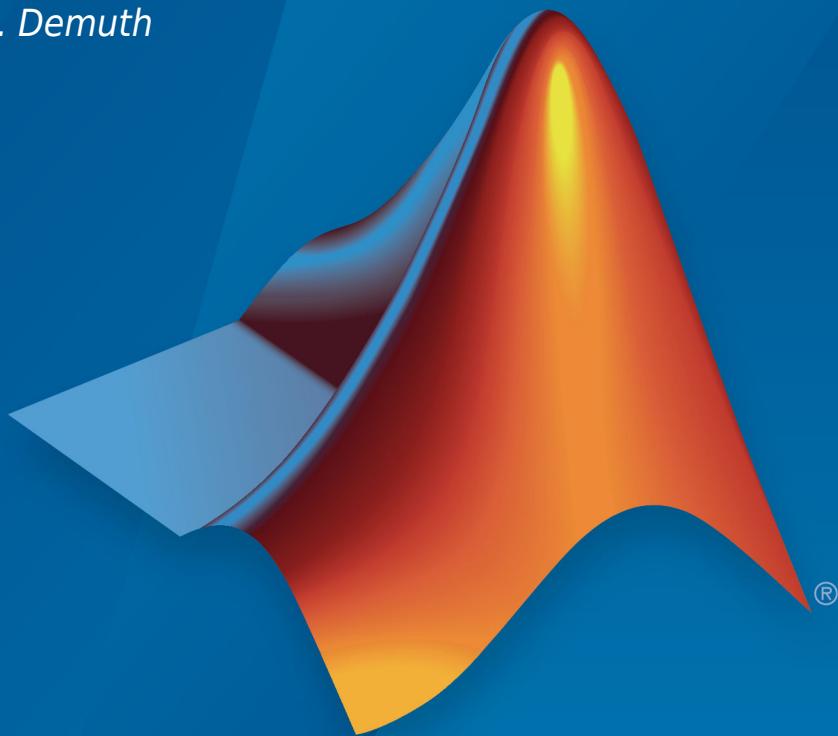
Deep Learning Toolbox™

User's Guide

Mark Hudson Beale

Martin T. Hagan

Howard B. Demuth



MATLAB®

R2019a

 MathWorks®

How to Contact MathWorks



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



Phone: 508-647-7000



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

Deep Learning Toolbox™ User's Guide

© COPYRIGHT 1992-2019 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

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

Deep Networks

1

Deep Learning in MATLAB	1-2
What Is Deep Learning?	1-2
Try Deep Learning in 10 Lines of MATLAB Code	1-5
Start Deep Learning Faster Using Transfer Learning	1-7
Train Classifiers Using Features Extracted from Pretrained Networks	1-8
Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud	1-8
Deep Learning with Big Data on GPUs and in Parallel	1-10
Training with Multiple GPUs	1-12
Deep Learning in the Cloud	1-13
Fetch and Preprocess Data in Background	1-14
Pretrained Deep Neural Networks	1-15
Load Pretrained Networks	1-16
Compare Pretrained Networks	1-17
Feature Extraction	1-19
Transfer Learning	1-20
Import and Export Networks	1-20
Learn About Convolutional Neural Networks	1-24
List of Deep Learning Layers	1-28
Layer Functions	1-28
Specify Layers of Convolutional Neural Network	1-37
Image Input Layer	1-38
Convolutional Layer	1-38
Batch Normalization Layer	1-43
ReLU Layer	1-44

Cross Channel Normalization (Local Response Normalization)	
Layer	1-45
Max and Average Pooling Layers	1-45
Dropout Layer	1-46
Fully Connected Layer	1-46
Output Layers	1-47
Set Up Parameters and Train Convolutional Neural Network	
.	1-52
Specify Solver and Maximum Number of Epochs	1-52
Specify and Modify Learning Rate	1-53
Specify Validation Data	1-54
Select Hardware Resource	1-54
Save Checkpoint Networks and Resume Training	1-55
Set Up Parameters in Convolutional and Fully Connected Layers	
.	1-55
Train Your Network	1-56
Deep Learning Tips and Tricks	1-57
Choose Network Architecture	1-57
Choose Training Options	1-59
Improve Training Accuracy	1-61
Fix Errors in Training	1-62
Prepare and Preprocess Data	1-64
Use Available Hardware	1-67
Fix Errors With Loading from MAT-Files	1-68
Resume Training from Checkpoint Network	1-70
Define Custom Deep Learning Layers	1-77
Layer Templates	1-78
Intermediate Layer Architecture	1-81
Check Validity of Layer	1-88
Include Layer in Network	1-89
Output Layer Architecture	1-90
Define Custom Deep Learning Layer with Learnable Parameters	1-97
Layer with Learnable Parameters Template	1-98
Name the Layer	1-99
Declare Properties and Learnable Parameters	1-100
Create Constructor Function	1-102
Create Forward Functions	1-104

Create Backward Function	1-107
Completed Layer	1-109
GPU Compatibility	1-110
Check Validity of Layer Using checkLayer	1-111
Include Custom Layer in Network	1-112
Define Custom Deep Learning Layer with Multiple Inputs	1-114
Layer with Learnable Parameters Template	1-114
Name the Layer	1-116
Declare Properties and Learnable Parameters	1-116
Create Constructor Function	1-118
Create Forward Functions	1-120
Create Backward Function	1-123
Completed Layer	1-126
Check Validity of Layer with Multiple Inputs	1-128
Use Custom Weighted Addition Layer in Network	1-128
Define Custom Regression Output Layer	1-131
Regression Output Layer Template	1-132
Name the Layer	1-132
Declare Layer Properties	1-133
Create Constructor Function	1-134
Create Forward Loss Function	1-135
Create Backward Loss Function	1-137
Completed Layer	1-137
GPU Compatibility	1-138
Check Output Layer Validity	1-139
Include Custom Regression Output Layer in Network	1-139
Define Custom Classification Output Layer	1-143
Classification Output Layer Template	1-144
Name the Layer	1-145
Declare Layer Properties	1-145
Create Constructor Function	1-146
Create Forward Loss Function	1-147
Create Backward Loss Function	1-148
Completed Layer	1-149
GPU Compatibility	1-150
Check Output Layer Validity	1-150
Include Custom Classification Output Layer in Network	1-151
Define Custom Weighted Classification Layer	1-154
Classification Output Layer Template	1-155
Name the Layer	1-156

Declare Layer Properties	1-156
Create Constructor Function	1-157
Create Forward Loss Function	1-158
Create Backward Loss Function	1-160
Completed Layer	1-161
GPU Compatibility	1-162
Check Output Layer Validity	1-163
Check Custom Layer Validity	1-165
Check Layer Validity	1-165
List of Tests	1-166
Generated Data	1-168
Diagnostics	1-169
Long Short-Term Memory Networks	1-181
LSTM Network Architecture	1-181
Layers	1-186
Classification, Prediction, and Forecasting	1-186
Sequence Padding, Truncation, and Splitting	1-187
Normalize Sequence Data	1-189
Out-of-Memory Data	1-190
LSTM Layer Architecture	1-190
Datastores for Deep Learning	1-194
Select Datastore	1-194
Input Datastore for Training, Validation, and Inference	1-194
Specify Read Size and Mini-Batch Size	1-196
Transform and Combine Datastores	1-196
Use Datastore for Parallel Training and Prefetch Read Optimization	1-199
Preprocess Images for Deep Learning	1-201
Resize Images	1-201
Augment Images for Training	1-202
Datastores for Advanced Image Preprocessing	1-203
Preprocess Volumes for Deep Learning	1-206
Read Volumetric Image Data	1-206
Read Volumetric Label Data for Semantic Segmentation	1-207
Associate Image and Label Data	1-208
Preprocess Volumetric Data	1-209

Develop Custom Mini-Batch Datastore	1-213
Overview	1-213
Implement MiniBatchable Datastore	1-213
Add Support for Shuffling	1-217
Validate Custom Mini-Batch Datastore	1-219

Deep Network Designer

2

Transfer Learning with Deep Network Designer	2-2
Choose a Pretrained Network	2-2
Import Network into Deep Network Designer	2-3
Edit Network for Transfer Learning	2-4
Check Network	2-9
Export Network for Training	2-10
Train Network Exported from Deep Network Designer	2-10
Build Networks with Deep Network Designer	2-15
Open the App and Import Networks	2-15
Create and Edit Networks	2-16
Check Network	2-17
Export Network for Training	2-18
Generate MATLAB Code from Deep Network Designer	2-20
Generate MATLAB Code and Recreate Network Layers	2-20
Train Network	2-20
Use Network for Prediction	2-22

Deep Learning in the Cloud

3

Scale Up Deep Learning in Parallel and in the Cloud	3-2
Deep Learning on Multiple GPUs	3-2
Deep Learning in the Cloud	3-4
Advanced Support for Fast Multi-Node GPU Communication	3-5

Deep Learning with MATLAB on Multiple GPUs	3-7
Select Particular GPUs to Use for Training	3-7
Train Network in the Cloud Using Automatic Parallel Support	3-8

Neural Network Design Book

Neural Network Objects, Data, and Training Styles	
4	
Workflow for Neural Network Design	4-2
Four Levels of Neural Network Design	4-4
Neuron Model	4-5
Simple Neuron	4-5
Transfer Functions	4-6
Neuron with Vector Input	4-7
Neural Network Architectures	4-11
One Layer of Neurons	4-11
Multiple Layers of Neurons	4-13
Input and Output Processing Functions	4-15
Create Neural Network Object	4-17
Configure Neural Network Inputs and Outputs	4-21
Understanding Deep Learning Toolbox Data Structures	4-23
Simulation with Concurrent Inputs in a Static Network	4-23
Simulation with Sequential Inputs in a Dynamic Network	4-24
Simulation with Concurrent Inputs in a Dynamic Network	4-26
Neural Network Training Concepts	4-28
Incremental Training with <code>adapt</code>	4-28
Batch Training	4-30

Multilayer Shallow Neural Networks and Backpropagation Training

5

Multilayer Shallow Neural Networks and Backpropagation Training	5-2
Multilayer Shallow Neural Network Architecture	5-4
Neuron Model (logsig, tansig, purelin)	5-4
Feedforward Neural Network	5-5
Prepare Data for Multilayer Shallow Neural Networks	5-8
Choose Neural Network Input-Output Processing Functions	5-9
Representing Unknown or Don't-Care Targets	5-11
Divide Data for Optimal Neural Network Training	5-12
Create, Configure, and Initialize Multilayer Shallow Neural Networks	5-14
Other Related Architectures	5-15
Initializing Weights (init)	5-15
Train and Apply Multilayer Shallow Neural Networks	5-17
Training Algorithms	5-18
Training Example	5-20
Use the Network	5-22
Analyze Shallow Neural Network Performance After Training	5-24
Improving Results	5-29
Limitations and Cautions	5-30

Introduction to Dynamic Neural Networks	6-2
How Dynamic Neural Networks Work	6-3
Feedforward and Recurrent Neural Networks	6-3
Applications of Dynamic Networks	6-10
Dynamic Network Structures	6-10
Dynamic Network Training	6-11
Design Time Series Time-Delay Neural Networks	6-14
Prepare Input and Layer Delay States	6-18
Design Time Series Distributed Delay Neural Networks	6-20
Design Time Series NARX Feedback Neural Networks	6-23
Multiple External Variables	6-30
Design Layer-Recurrent Neural Networks	6-31
Create Reference Model Controller with MATLAB Script	6-34
Multiple Sequences with Dynamic Neural Networks	6-41
Neural Network Time-Series Utilities	6-42
Train Neural Networks with Error Weights	6-44
Normalize Errors of Multiple Outputs	6-47
Multistep Neural Network Prediction	6-52
Set Up in Open-Loop Mode	6-52
Multistep Closed-Loop Prediction From Initial Conditions	6-53
Multistep Closed-Loop Prediction Following Known Sequence	6-53
Following Closed-Loop Simulation with Open-Loop Simulation	6-54

Introduction to Neural Network Control Systems	7-2
Design Neural Network Predictive Controller in Simulink	7-4
System Identification	7-4
Predictive Control	7-5
Use the Neural Network Predictive Controller Block	7-6
Design NARMA-L2 Neural Controller in Simulink	7-14
Identification of the NARMA-L2 Model	7-14
NARMA-L2 Controller	7-16
Use the NARMA-L2 Controller Block	7-18
Design Model-Reference Neural Controller in Simulink	7-23
Use the Model Reference Controller Block	7-24
Import-Export Neural Network Simulink Control Systems	7-31
Import and Export Networks	7-31
Import and Export Training Data	7-35

Radial Basis Neural Networks

Introduction to Radial Basis Neural Networks	8-2
Important Radial Basis Functions	8-2
Radial Basis Neural Networks	8-3
Neuron Model	8-3
Network Architecture	8-4
Exact Design (newrbe)	8-6
More Efficient Design (newrb)	8-7
Examples	8-8
Probabilistic Neural Networks	8-9
Network Architecture	8-9
Design (newpnn)	8-10

Generalized Regression Neural Networks	8-12
Network Architecture	8-12
Design (newgrnn)	8-14

Self-Organizing and Learning Vector Quantization Networks

9

Introduction to Self-Organizing and LVQ	9-2
Important Self-Organizing and LVQ Functions	9-2
Cluster with a Competitive Neural Network	9-3
Architecture	9-3
Create a Competitive Neural Network	9-4
Kohonen Learning Rule (learnk)	9-5
Bias Learning Rule (learncon)	9-5
Training	9-6
Graphical Example	9-8
Cluster with Self-Organizing Map Neural Network	9-9
Topologies (gridtop, hextop, randtop)	9-10
Distance Functions (dist, linkdist, mandist, boxdist)	9-14
Architecture	9-17
Create a Self-Organizing Map Neural Network (selforgmap)	
.	9-18
Training (learnsomb)	9-19
Examples	9-22
Learning Vector Quantization (LVQ) Neural Networks	9-34
Architecture	9-34
Creating an LVQ Network	9-35
LVQ1 Learning Rule (learnlv1)	9-38
Training	9-39
Supplemental LVQ2.1 Learning Rule (learnlv2)	9-41

Adaptive Filters and Adaptive Training

10

Adaptive Neural Network Filters	10-2
Adaptive Functions	10-2
Linear Neuron Model	10-3
Adaptive Linear Network Architecture	10-4
Least Mean Square Error	10-6
LMS Algorithm (learnwh)	10-7
Adaptive Filtering (adapt)	10-7

Advanced Topics

11

Neural Networks with Parallel and GPU Computing	11-2
Deep Learning	11-2
Modes of Parallelism	11-2
Distributed Computing	11-3
Single GPU Computing	11-5
Distributed GPU Computing	11-8
Parallel Time Series	11-10
Parallel Availability, Fallbacks, and Feedback	11-10
Optimize Neural Network Training Speed and Memory	11-12
Memory Reduction	11-12
Fast Elliot Sigmoid	11-12
Choose a Multilayer Neural Network Training Function	11-16
SIN Data Set	11-17
PARITY Data Set	11-19
ENGINE Data Set	11-22
CANCER Data Set	11-24
CHOLESTEROL Data Set	11-26
DIABETES Data Set	11-28
Summary	11-30
Improve Shallow Neural Network Generalization and Avoid Overfitting	11-32
Retraining Neural Networks	11-34
Multiple Neural Networks	11-35

Early Stopping	11-36
Index Data Division (divideind)	11-37
Random Data Division (dividerand)	11-37
Block Data Division (divideblock)	11-37
Interleaved Data Division (divideint)	11-38
Regularization	11-38
Summary and Discussion of Early Stopping and Regularization	11-41
Posttraining Analysis (regression)	11-43
Edit Shallow Neural Network Properties	11-46
Custom Network	11-46
Network Definition	11-47
Network Behavior	11-57
Custom Neural Network Helper Functions	11-60
Automatically Save Checkpoints During Neural Network Training	11-61
Deploy Shallow Neural Network Functions	11-63
Deployment Functions and Tools for Trained Networks	11-63
Generate Neural Network Functions for Application Deployment	11-64
.	
Generate Simulink Diagrams	11-67
Deploy Training of Shallow Neural Networks	11-68

Historical Neural Networks

12

Historical Neural Networks Overview	12-2
Perceptron Neural Networks	12-3
Neuron Model	12-3
Perceptron Architecture	12-5
Create a Perceptron	12-6
Perceptron Learning Rule (learnp)	12-8
Training (train)	12-10
Limitations and Cautions	12-15

Linear Neural Networks	12-18
Neuron Model	12-18
Network Architecture	12-19
Least Mean Square Error	12-22
Linear System Design (newlind)	12-23
Linear Networks with Delays	12-24
LMS Algorithm (learnwh)	12-26
Linear Classification (train)	12-28
Limitations and Cautions	12-30

Neural Network Object Reference

13

Neural Network Object Properties	13-2
General	13-2
Architecture	13-2
Subobject Structures	13-6
Functions	13-8
Weight and Bias Values	13-12
Neural Network Subobject Properties	13-14
Inputs	13-14
Layers	13-16
Outputs	13-22
Biases	13-24
Input Weights	13-25
Layer Weights	13-27

Shallow Neural Networks Bibliography

14

Shallow Neural Networks Bibliography	14-2
---	-------------

Mathematical Notation

A

Mathematics and Code Equivalents	A-2
Mathematics Notation to MATLAB Notation	A-2
Figure Notation	A-2

Neural Network Blocks for the Simulink Environment

B

Neural Network Simulink Block Library	B-2
Transfer Function Blocks	B-3
Net Input Blocks	B-3
Weight Blocks	B-3
Processing Blocks	B-4
Deploy Shallow Neural Network Simulink Diagrams	B-5
Example	B-5
Suggested Exercises	B-7
Generate Functions and Objects	B-8

Code Notes

C

Deep Learning Toolbox Data Conventions	C-2
Dimensions	C-2
Variables	C-2

Deep Networks

- “Deep Learning in MATLAB” on page 1-2
- “Deep Learning with Big Data on GPUs and in Parallel” on page 1-10
- “Pretrained Deep Neural Networks” on page 1-15
- “Learn About Convolutional Neural Networks” on page 1-24
- “List of Deep Learning Layers” on page 1-28
- “Specify Layers of Convolutional Neural Network” on page 1-37
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-52
- “Deep Learning Tips and Tricks” on page 1-57
- “Resume Training from Checkpoint Network” on page 1-70
- “Define Custom Deep Learning Layers” on page 1-77
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “Define Custom Regression Output Layer” on page 1-131
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Weighted Classification Layer” on page 1-154
- “Check Custom Layer Validity” on page 1-165
- “Long Short-Term Memory Networks” on page 1-181
- “Datastores for Deep Learning” on page 1-194
- “Preprocess Images for Deep Learning” on page 1-201
- “Preprocess Volumes for Deep Learning” on page 1-206
- “Develop Custom Mini-Batch Datastore” on page 1-213

Deep Learning in MATLAB

In this section...

- “What Is Deep Learning?” on page 1-2
- “Try Deep Learning in 10 Lines of MATLAB Code” on page 1-5
- “Start Deep Learning Faster Using Transfer Learning” on page 1-7
- “Train Classifiers Using Features Extracted from Pretrained Networks” on page 1-8
- “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud” on page 1-8

What Is Deep Learning?

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

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

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

What Do You Want to Do?	Learn More
Perform transfer learning to fine-tune a network with your data	<p>“Start Deep Learning Faster Using Transfer Learning” on page 1-7</p> <p>Tip Fine-tuning a pretrained network to learn a new task is typically much faster and easier than training a new network.</p>

What Do You Want to Do?	Learn More
Classify images with pretrained networks	“Pretrained Deep Neural Networks” on page 1-15
Create a new deep neural network for classification or regression	“Create Simple Deep Learning Network for Classification” “Train Convolutional Neural Network for Regression”
Resize, rotate, or preprocess images for training or prediction	“Preprocess Images for Deep Learning” on page 1-201
Label your image data automatically based on folder names, or interactively using an app	“Train Network for Image Classification” Image Labeler
Create deep learning networks for sequence and time series data.	“Sequence Classification Using Deep Learning” “Time Series Forecasting Using Deep Learning”
Classify each pixel of an image (for example, road, car, pedestrian)	“Semantic Segmentation Basics” (Computer Vision Toolbox)
Detect and recognize objects in images	“Deep Learning, Semantic Segmentation, and Detection” (Computer Vision Toolbox)
Classify text data	“Classify Text Data Using Deep Learning”
Classify audio data for speech recognition	“Speech Command Recognition Using Deep Learning”
Visualize what features networks have learned	“Deep Dream Images Using AlexNet” “Visualize Activations of a Convolutional Neural Network”
Train on CPU, GPU, multiple GPUs, in parallel on your desktop or on clusters in the cloud, and work with data sets too large to fit in memory	“Deep Learning with Big Data on GPUs and in Parallel” on page 1-10

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

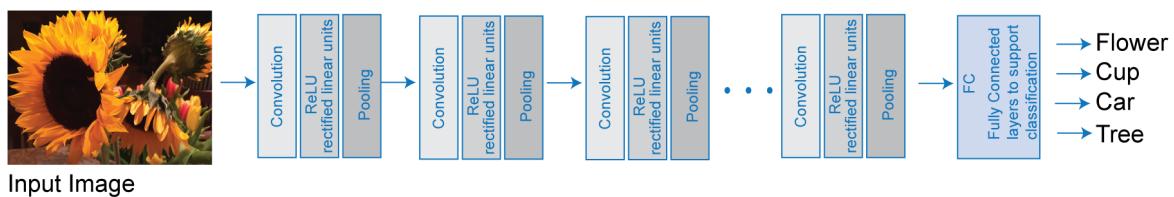
To choose whether to use a pretrained network or create a new deep network, consider the scenarios in this table.

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

For more information, see “Choose Network Architecture” on page 1-57.

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

You train models using a large set of labeled data and neural network architectures that contain many layers, usually including some convolutional layers. Training these models is computationally intensive and you can usually accelerate training by using a high performance GPU. This diagram shows how convolutional neural networks combine layers that automatically learn features from many images to classify new images.



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

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

Try Deep Learning in 10 Lines of MATLAB Code

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

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

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

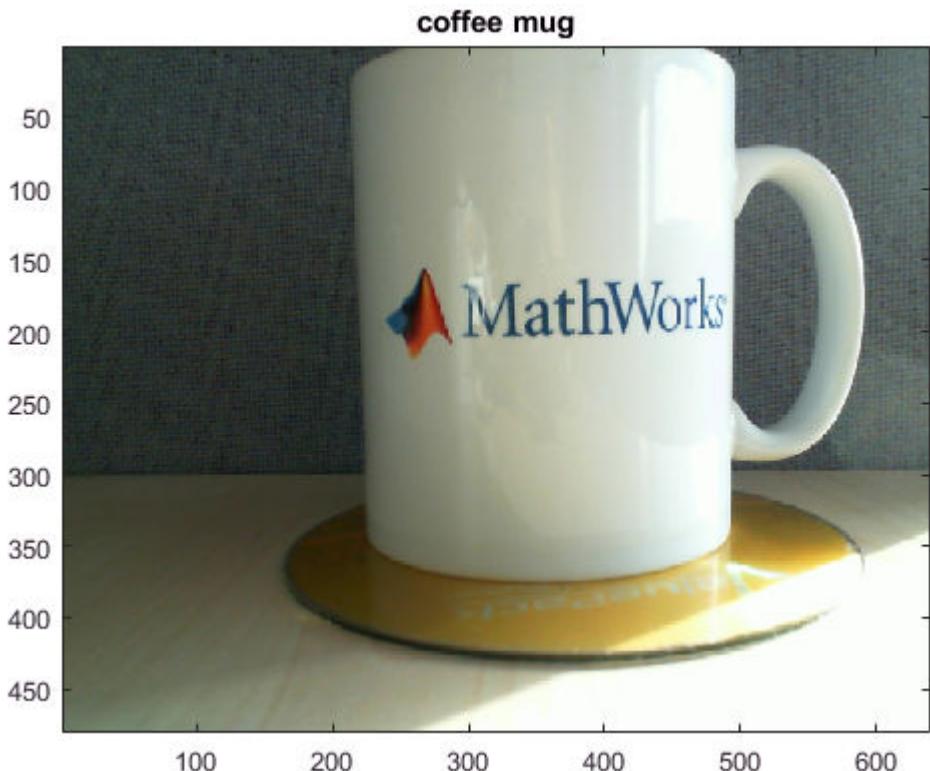
If you need to install the `webcam` and `alexnet` add-ons, a message from each function appears with a link to help you download the free add-ons using Add-On Explorer. Alternatively, see Deep Learning Toolbox Model for *AlexNet Network* and MATLAB Support Package for USB Webcams.

After you install Deep Learning Toolbox Model for *AlexNet Network*, you can use it to classify images. AlexNet is a pretrained convolutional neural network (CNN) that has been trained on more than a million images and can classify images into 1000 object categories (for example, keyboard, mouse, coffee mug, pencil, and many animals).

- 2 Run the following code to show and classify live images. Point the webcam at an object and the neural network reports what class of object it thinks the webcam is showing. It will keep classifying images until you press **Ctrl+C**. The code resizes the image for the network using `imresize`.

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

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



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

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

For next steps in deep learning, you can use the pretrained network for other tasks. Solve new classification problems on your image data with transfer learning or feature extraction. For examples, see “Start Deep Learning Faster Using Transfer Learning” on page 1-7 and “Train Classifiers Using Features Extracted from Pretrained Networks” on page 1-8. To try other pretrained networks, see “Pretrained Deep Neural Networks” on page 1-15.

Start Deep Learning Faster Using Transfer Learning

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

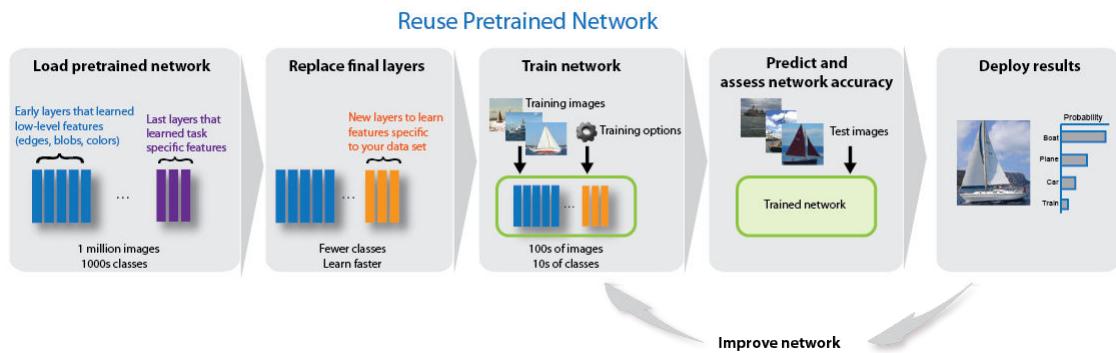
For example, if you take a network trained on thousands or millions of images, you can retrain it for new object detection using only hundreds of images. You can effectively fine-tune a pretrained network with much smaller data sets than the original training data. If you have a very large dataset, then transfer learning might not be faster than training a new network.

Transfer learning enables you to:

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

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

For programmatic examples, see “Get Started with Transfer Learning” and “Train Deep Learning Network to Classify New Images”.



Train Classifiers Using Features Extracted from Pretrained Networks

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

For an example, see “Extract Image Features Using Pretrained Network”.

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

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

Training deep networks is extremely computationally intensive and you can usually accelerate training by using a high performance GPU. If you do not have a suitable GPU, you can train on one or more CPU cores instead. You can train a convolutional neural network on a single GPU or CPU, or on multiple GPUs or CPU cores, or in parallel on a cluster. Using GPU or parallel options requires Parallel Computing Toolbox.

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

To learn more about deep learning hardware and memory settings, see “Deep Learning with Big Data on GPUs and in Parallel” on page 1-10.

See Also

Related Examples

- “Classify Webcam Images Using Deep Learning”
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Deep Learning Network to Classify New Images”
- “Pretrained Deep Neural Networks” on page 1-15
- “Create Simple Deep Learning Network for Classification”
- “Deep Learning with Big Data on GPUs and in Parallel” on page 1-10
- “Deep Learning, Semantic Segmentation, and Detection” (Computer Vision Toolbox)
- “Classify Text Data Using Deep Learning”
- “Deep Learning Tips and Tricks” on page 1-57

Deep Learning with Big Data on GPUs and in Parallel

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

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

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

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

Deep Learning Hardware and Memory Considerations	Recommendations	Required Products
Data too large to fit in memory	To import data from image collections that are too large to fit in memory, use the <code>augmentedImageDatastore</code> function. This function is designed to read batches of images for faster processing in machine learning and computer vision applications.	MATLAB Deep Learning Toolbox

Deep Learning Hardware and Memory Considerations	Recommendations	Required Products
CPU	If you do not have a suitable GPU, then you can train on a CPU instead. By default, the <code>trainNetwork</code> function uses the CPU if no GPU is available.	MATLAB Deep Learning Toolbox
GPU	By default, the <code>trainNetwork</code> function uses a GPU if available. Requires a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. Check your GPU using <code>gpuDevice</code> . Specify the execution environment using the <code>trainingOptions</code> function.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox
Parallel on your local machine using multiple GPUs or CPU cores	Take advantage of multiple workers by specifying the execution environment with the <code>trainingOptions</code> function. If you have more than one GPU on your machine, specify ' <code>multi-gpu</code> '. Otherwise, specify ' <code>parallel</code> '.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox

Deep Learning Hardware and Memory Considerations	Recommendations	Required Products
Parallel on a cluster or in the cloud	Scale up to use workers on clusters or in the cloud to accelerate your deep learning computations. Use <code>trainingOptions</code> and specify ' <code>parallel</code> ' to use a compute cluster. For more information, see "Deep Learning in the Cloud" on page 1-13.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox MATLAB Parallel Server™

Tip To learn more, see "Scale Up Deep Learning in Parallel and in the Cloud" on page 3-2.

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

Because single-precision and double-precision performance of GPUs can differ substantially, it is important to know in which precision computations are performed. If you only use a GPU for deep learning, then single-precision performance is one of the most important characteristics of a GPU. If you also use a GPU for other computations using Parallel Computing Toolbox, then high double-precision performance is important. This is because many functions in MATLAB use double-precision arithmetic by default. For more information, see "Improve Performance Using Single Precision Calculations" (Parallel Computing Toolbox).

Training with Multiple GPUs

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

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

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

The optimal batch size depends on your exact network, dataset, and GPU hardware. When training with multiple GPUs, each image batch is distributed between the GPUs. This effectively increases the total GPU memory available, allowing larger batch sizes. Because it improves the significance of each batch, you can increase the learning rate. A good general guideline is to increase the learning rate proportionally to the increase in batch size. Depending on your application, a larger batch size and learning rate can speed up training without a decrease in accuracy, up to some limit.

Using multiple GPUs can speed up training significantly. To decide if you expect multi-GPU training to deliver a performance gain, consider the following factors:

- How long is the iteration on each GPU? If each GPU iteration is short, then the added overhead of communication between GPUs can dominate. Try increasing the computation per iteration by using a larger batch size.
- Are all the GPUs on a single machine? Communication between GPUs on different machines introduces a significant communication delay. You can mitigate this if you have suitable hardware. For more information, see “Advanced Support for Fast Multi-Node GPU Communication” on page 3-5.

To learn more, see “Scale Up Deep Learning in Parallel and in the Cloud” on page 3-2 and “Select Particular GPUs to Use for Training” on page 3-7.

Deep Learning in the Cloud

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

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

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

Fetch and Preprocess Data in Background

When training a network in parallel, you can fetch and preprocess data in the background. To perform data dispatch in the background, enable background dispatch in the mini-batch datastore used by `trainNetwork`. You can use a built-in mini-batch datastore, such as `augmentedImage datastore`, `denoisingImage datastore`, or `pixelLabelImage datastore`. You can also use a custom mini-batch datastore with background dispatch enabled. For more information on creating custom mini-batch datastores, see “Develop Custom Mini-Batch Datastore” on page 1-213.

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

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

See Also

`trainNetwork` | `trainingOptions`

See Also

Related Examples

- “Scale Up Deep Learning in Parallel and in the Cloud” on page 3-2

Pretrained Deep Neural Networks

In this section...

- “Load Pretrained Networks” on page 1-16
- “Compare Pretrained Networks” on page 1-17
- “Feature Extraction” on page 1-19
- “Transfer Learning” on page 1-20
- “Import and Export Networks” on page 1-20

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

You can use previously trained networks for the following tasks:

Purpose	Description
Classification	Apply pretrained networks directly to classification problems. To classify a new image, use <code>classify</code> . For an example showing how to use a pretrained network for classification, see “Classify Image Using GoogLeNet”.
Feature Extraction	Use a pretrained network as a feature extractor by using the layer activations as features. You can use these activations as features to train another machine learning model, such as a support vector machine (SVM). For more information, see “Feature Extraction” on page 1-19. For an example, see “Extract Image Features Using Pretrained Network”.

Purpose	Description
Transfer Learning	Take layers from a network trained on a large data set and fine-tune on a new data set. For more information, see “Transfer Learning” on page 1-20. For a simple example, see “Get Started with Transfer Learning”. To try more pretrained networks, see “Train Deep Learning Network to Classify New Images”.

Load Pretrained Networks

Use functions such as `googlenet` to get links to download pretrained networks from the Add-On Explorer. The following table lists the available pretrained networks trained on ImageNet and some of their properties. The network depth is defined as the largest number of sequential convolutional or fully connected layers on a path from the input layer to the output layer. The inputs to all networks are RGB images.

Network	Depth	Size	Parameters (Millions)	Image Input Size
alexnet	8	227 MB	61.0	227-by-227
vgg16	16	515 MB	138	224-by-224
vgg19	19	535 MB	144	224-by-224
squeezezenet	18	4.6 MB	1.24	227-by-227
googlenet	22	27 MB	7.0	224-by-224
inceptionv3	48	89 MB	23.9	299-by-299
densenet201	201	77 MB	20.0	224-by-224
mobilenetv2	53	13 MB	3.5	224-by-224
resnet18	18	44 MB	11.7	224-by-224
resnet50	50	96 MB	25.6	224-by-224
resnet101	101	167 MB	44.6	224-by-224
xception	71	85 MB	22.9	299-by-299
inceptionresnetv2	164	209 MB	55.9	299-by-299

Network	Depth	Size	Parameters (Millions)	Image Input Size
shufflenet	50	6.3 MB	1.4	224-by-224
nasnetmobile	*	20 MB	5.3	224-by-224
nasnetlarge	*	360 MB	88.9	331-by-331

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

GoogLeNet Trained on Places365

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

Compare Pretrained Networks

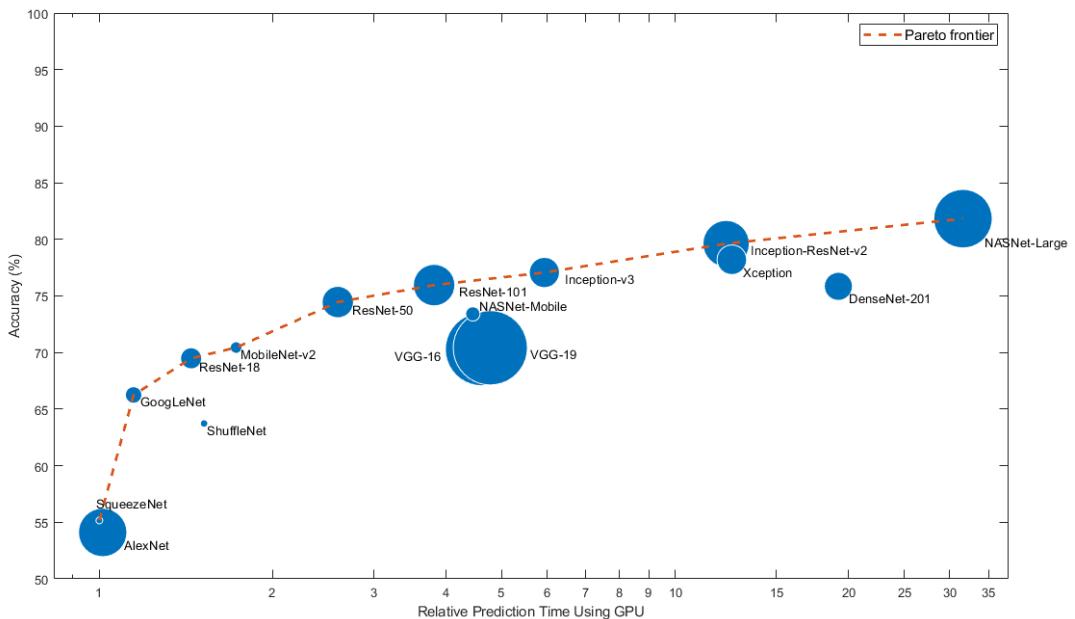
Pretrained networks have different characteristics that matter when choosing a network to apply to your problem. The most important characteristics are network accuracy, speed, and size. Choosing a network is generally a tradeoff between these characteristics.

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

Use the plot below to compare the ImageNet validation accuracy with the time required to make a prediction using the network. A good network has a high accuracy and is fast. The plot displays the classification accuracy versus the prediction time when using a modern GPU (an NVIDIA TITAN Xp) and a mini-batch size of 64. The prediction time is measured relative to the fastest network. The area of each marker is proportional to the size of the network on disk.

A network is *Pareto efficient* if there is no other network that is better on all the metrics being compared, in this case accuracy and prediction time. The set of all Pareto efficient networks is called the *Pareto frontier*. The Pareto frontier contains all the networks that are not worse than another network on both metrics. The plot connects the networks that are on the Pareto frontier in the plane of accuracy and prediction time. All networks except AlexNet, VGG-16, VGG-19, Xception, NASNet-Mobile, ShuffleNet, and DenseNet-201 are on the Pareto frontier.

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



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

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

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

Network Accuracy

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

Feature Extraction

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

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

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

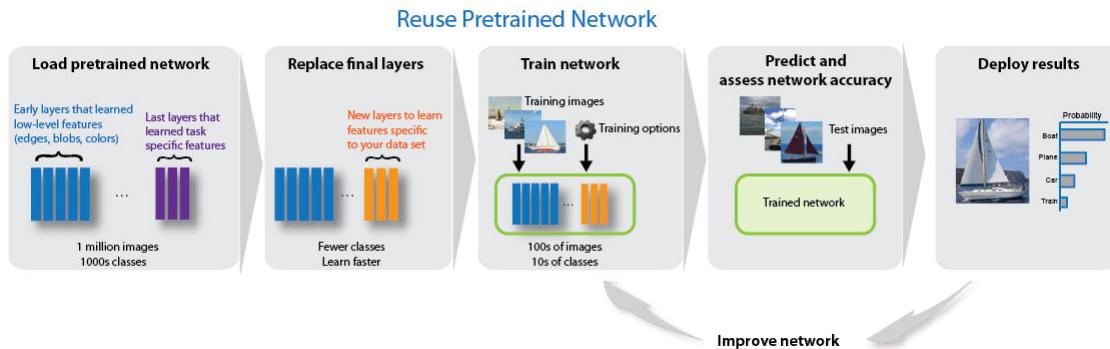
ResNets are often good feature extractors. For an example showing how to use a pretrained network for feature extraction, see “Extract Image Features Using Pretrained Network”.

Transfer Learning

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

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

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



Import and Export Networks

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

Import from Keras

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

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

Import from Caffe

Import pretrained networks from Caffe by using the `importCaffeNetwork` function. There are many pretrained networks available in Caffe Model Zoo [5]. Download the desired .prototxt and .caffemodel files and use `importCaffeNetwork` to import the pretrained network into MATLAB. For more information, see `importCaffeNetwork`.

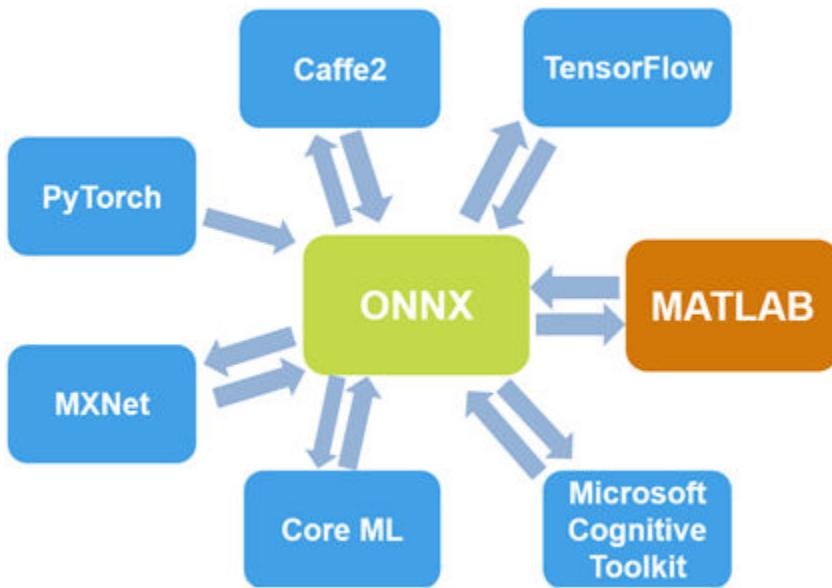
You can import network architectures of Caffe networks. Download the desired .prototxt file and use `importCaffeLayers` to import the network layers into MATLAB. For more information, see `importCaffeLayers`.

Export to and Import from ONNX

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

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

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



References

- [1] *ImageNet*. <http://www.image-net.org>
- [2] Russakovsky, O., Deng, J., Su, H., et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211-252
- [3] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. "Places: An image database for deep scene understanding." *arXiv preprint arXiv:1610.02055* (2016).
- [4] *Places*. <http://places2.csail.mit.edu/>
- [5] *Caffe Model Zoo*. http://caffe.berkeleyvision.org/model_zoo.html

See Also

`alexnet | densenet201 | exportONNXNetwork | googlenet | importCaffeLayers | importCaffeNetwork | importKerasLayers | importKerasNetwork |`

```
importONNXLayers | importONNXNetwork | inceptionresnetv2 | inceptionv3 |  
mobilenetv2 | nasnetlarge | nasnetmobile | resnet101 | resnet18 | resnet50 |  
shufflenet | squeezenet | vgg16 | vgg19 | xception
```

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Extract Image Features Using Pretrained Network”
- “Classify Image Using GoogLeNet”
- “Train Deep Learning Network to Classify New Images”
- “Visualize Features of a Convolutional Neural Network”
- “Visualize Activations of a Convolutional Neural Network”
- “Deep Dream Images Using AlexNet”

Learn About Convolutional Neural Networks

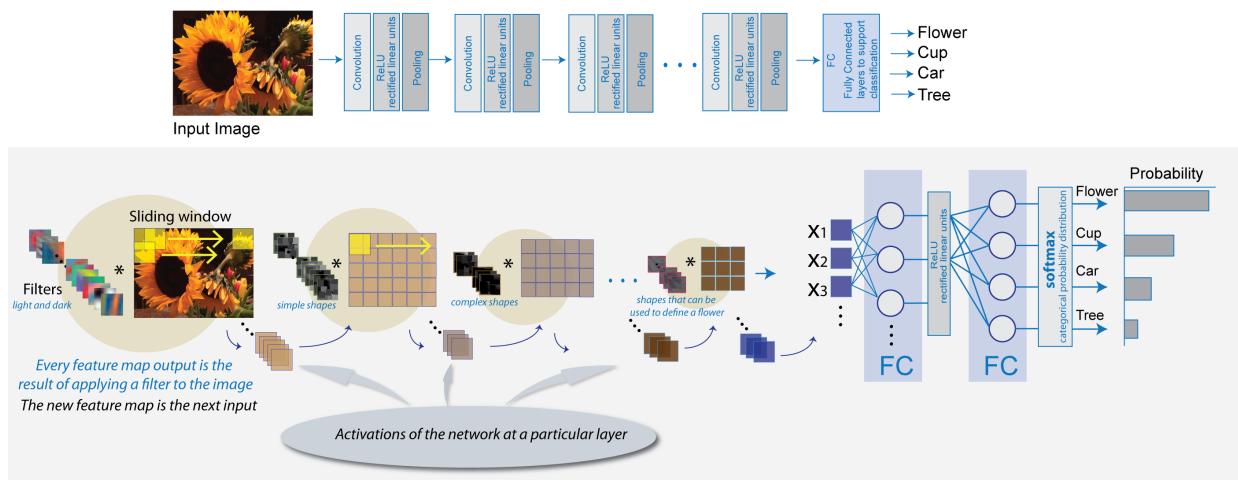
Convolutional neural networks (ConvNets) are widely used tools for deep learning. They are specifically suitable for images as inputs, although they are also used for other applications such as text, signals, and other continuous responses. They differ from other types of neural networks in a few ways:

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

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

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

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



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

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

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

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

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

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

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

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

For detailed discussion of layers of a ConvNet, see “Specify Layers of Convolutional Neural Network” on page 1-37. For setting up training parameters, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-52.

References

- [1] Hubel, H. D. and Wiesel, T. N. "Receptive Fields of Single neurones in the Cat's Striate Cortex." *Journal of Physiology*. Vol 148, pp. 574-591, 1959.
- [2] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.

See Also

`trainNetwork` | `trainingOptions`

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-37
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-52

- “Get Started with Transfer Learning”
- “Create Simple Deep Learning Network for Classification”
- “Pretrained Deep Neural Networks” on page 1-15

List of Deep Learning Layers

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

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

Task	Learn More
Create deep learning networks for image classification or regression.	“Create Simple Deep Learning Network for Classification” “Train Convolutional Neural Network for Regression” “Train Residual Network for Image Classification”
Create deep learning networks for sequence and time series data.	“Sequence Classification Using Deep Learning” “Time Series Forecasting Using Deep Learning”
Create deep learning network for audio data.	“Speech Command Recognition Using Deep Learning”
Create deep learning network for text data.	“Classify Text Data Using Deep Learning” “Generate Text Using Deep Learning”

Layer Functions

Use the following functions to create different layer types. Alternatively, you can define your own custom layers. To learn how to define your own custom layers, see “Define Custom Deep Learning Layers” on page 1-77.

Input Layers

Function	Description
 <code>imageInputLayer</code>	An image input layer inputs 2-D images to a network and applies data normalization.
 <code>image3dInputLayer</code>	A 3-D image input layer inputs 3-D images or volumes to a network and applies data normalization.
 <code>sequenceInputLayer</code>	A sequence input layer inputs sequence data to a network.
 <code>roiInputLayer</code> (Computer Vision Toolbox™)	An ROI input layer inputs images to a Fast R-CNN object detection network.

Convolution and Fully Connected Layers

Function	Description
 <code>convolution2dLayer</code>	A 2-D convolutional layer applies sliding convolutional filters to the input.
 <code>convolution3dLayer</code>	A 3-D convolutional layer applies sliding cuboidal convolution filters to three-dimensional input.
 <code>groupedConvolution2dLayer</code>	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.
 <code>transposedConv2dLayer</code>	A transposed 2-D convolution layer upsamples feature maps.
 <code>transposedConv3dLayer</code>	A transposed 3-D convolution layer upsamples three-dimensional feature maps.

Function	Description
 <code>fullyConnectedLayer</code>	A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.

Sequence Layers

Function	Description
 <code>sequenceInputLayer</code>	A sequence input layer inputs sequence data to a network.
 <code>lstmLayer</code>	An LSTM layer learns long-term dependencies between time steps in time series and sequence data.
 <code>bilstmLayer</code>	A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.
 <code>sequenceFoldingLayer</code>	A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.
 <code>sequenceUnfoldingLayer</code>	A sequence unfolding layer restores the sequence structure of the input data after sequence folding.
 <code>flattenLayer</code>	A flatten layer collapses the spatial dimensions of the input into the channel dimension.
 <code>wordEmbeddingLayer</code> (Text Analytics Toolbox™)	A word embedding layer maps word indices to vectors.

Activation Layers

Function	Description
 <code>reluLayer</code>	A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.
 <code>leakyReluLayer</code>	A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.
 <code>clippedReluLayer</code>	A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the <i>clipping ceiling</i> is set to that clipping ceiling.
 <code>eluLayer</code>	An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.
 <code>tanhLayer</code>	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.
 <code>preluLayer</code> on page 1-97 (Custom layer example)	A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar learned at training time.

Normalization, Dropout, and Cropping Layers

Function	Description
 <code>batchNormalizationLayer</code>	A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

Function	Description
 crossChannelNormalizationLayer	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.
 dropoutLayer	A dropout layer randomly sets input elements to zero with a given probability.
 crop2dLayer (Computer Vision Toolbox)	A 2-D crop layer applies 2-D cropping to the input.

Pooling and Unpooling Layers

Function	Description
 averagePooling2dLayer	An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.
 averagePooling3dLayer	A 3-D average pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions and computing the average values of each region.
 maxPooling2dLayer	A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.
 maxPooling3dLayer	A 3-D max pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions, and computing the maximum of each region.
 maxUnpooling2dLayer	A max unpooling layer un pools the output of a max pooling layer.

Combination Layers

Function	Description
 <code>additionLayer</code>	An addition layer adds inputs from multiple neural network layers element-wise.
 <code>depthConcatenationLayer</code>	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).
 <code>concatenationLayer</code>	A concatenation layer takes inputs and concatenates them along a specified dimension. The inputs must have the same size in all dimensions except the concatenation dimension.
 <code>weightedAdditionLayer</code> on page 1-114 (Custom layer example)	A weighted addition layer scales and adds inputs from multiple neural network layers element-wise.

Object Detection Layers

Function	Description
 <code>roiInputLayer</code> (Computer Vision Toolbox)	An ROI input layer inputs images to a Fast R-CNN object detection network.
 <code>roiMaxPooling2dLayer</code> (Computer Vision Toolbox)	An ROI max pooling layer outputs fixed size feature maps for every rectangular ROI within the input feature map. Use this layer to create a Fast or Faster R-CNN object detection network.
 <code>regionProposalLayer</code> (Computer Vision Toolbox)	A region proposal layer outputs bounding boxes around potential objects in an image as part of the region proposal network (RPN) within Faster R-CNN.

Function	Description
 <code>rpnSoftmaxLayer</code> (Computer Vision Toolbox)	A region proposal network (RPN) softmax layer applies a softmax activation function to the input. Use this layer to create a Faster R-CNN object detection network.
 <code>rpnClassificationLayer</code> (Computer Vision Toolbox)	A region proposal network (RPN) classification layer classifies image regions as either <i>object</i> or <i>background</i> by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.
 <code>rcnnBoxRegressionLayer</code> (Computer Vision Toolbox)	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.

Output Layers

Function	Description
 <code>softmaxLayer</code>	A softmax layer applies a softmax function to the input.
 <code>classificationLayer</code>	A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.
 <code>regressionLayer</code>	A regression layer computes the half-mean-squared-error loss for regression problems.
 <code>pixelClassificationLayer</code> (Computer Vision Toolbox)	A pixel classification layer provides a categorical label for each image pixel or voxel.
 <code>rpnSoftmaxLayer</code> (Computer Vision Toolbox)	A region proposal network (RPN) softmax layer applies a softmax activation function to the input. Use this layer to create a Faster R-CNN object detection network.

Function	Description
 rpnClassificationLayer (Computer Vision Toolbox)	A region proposal network (RPN) classification layer classifies image regions as either <i>object</i> or <i>background</i> by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.
 rcnnBoxRegressionLayer (Computer Vision Toolbox)	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.
 weightedClassificationLayer on page 1-154 (Custom layer example)	A weighted classification layer computes the weighted cross entropy loss for classification problems.
 dicePixelClassificationLayer (Custom layer example)	A Dice pixel classification layer computes the Dice loss for semantic segmentation problems.
 sseClassificationLayer on page 1-143 (Custom layer example)	A classification SSE layer computes the sum of squares error loss for classification problems.
 maeRegressionLayer on page 1-131 (Custom layer example)	A regression MAE layer computes the mean absolute error loss for regression problems.

See Also

[trainNetwork](#) | [trainingOptions](#)

More About

- “Specify Layers of Convolutional Neural Network” on page 1-37
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-52
- “Define Custom Deep Learning Layers” on page 1-77
- “Create Simple Deep Learning Network for Classification”

- “Sequence Classification Using Deep Learning”
- “Pretrained Deep Neural Networks” on page 1-15
- “Deep Learning Tips and Tricks” on page 1-57

Specify Layers of Convolutional Neural Network

In this section...

- “Image Input Layer” on page 1-38
- “Convolutional Layer” on page 1-38
- “Batch Normalization Layer” on page 1-43
- “ReLU Layer” on page 1-44
- “Cross Channel Normalization (Local Response Normalization) Layer” on page 1-45
- “Max and Average Pooling Layers” on page 1-45
- “Dropout Layer” on page 1-46
- “Fully Connected Layer” on page 1-46
- “Output Layers” on page 1-47

The first step of creating and training a new convolutional neural network (ConvNet) is to define the network architecture. This topic explains the details of ConvNet layers, and the order they appear in a ConvNet. For a complete list of deep learning layers and how to create them, see “List of Deep Learning Layers” on page 1-28. To learn about LSTM networks for sequence classification and regression, see “Long Short-Term Memory Networks” on page 1-181. To learn how to create your own custom layers, see “Define Custom Deep Learning Layers” on page 1-77.

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

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

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

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

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

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

Image Input Layer

Create an image input layer using `imageInputLayer`.

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

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

Convolutional Layer

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

The convolutional layer consists of various components.¹

Filters and Stride

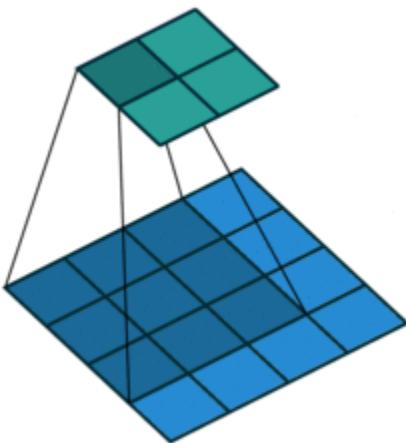
A convolutional layer consists of neurons that connect to subregions of the input images or the outputs of the previous layer. The layer learns the features localized by these regions while scanning through an image. When creating a layer using the

1. Image credit: Convolution arithmetic (License)

`convolution2dLayer` function, you can specify the size of these regions using the `filterSize` input argument.

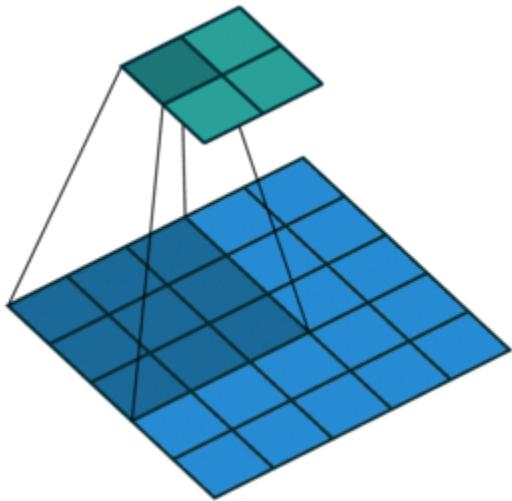
For each region, the `trainNetwork` function computes a dot product of the weights and the input, and then adds a bias term. A set of weights that is applied to a region in the image is called a *filter*. The filter moves along the input image vertically and horizontally, repeating the same computation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

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



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

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



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

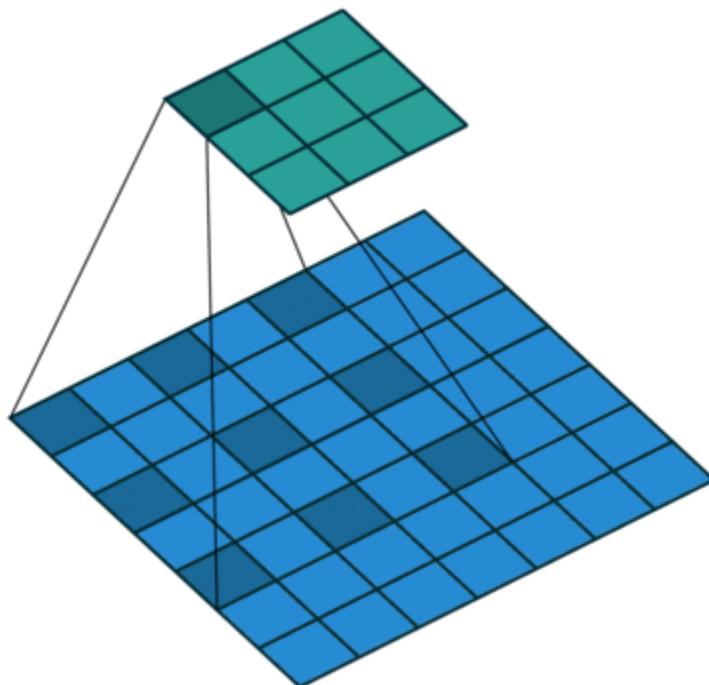
Dilated Convolution

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

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

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

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



Feature Maps

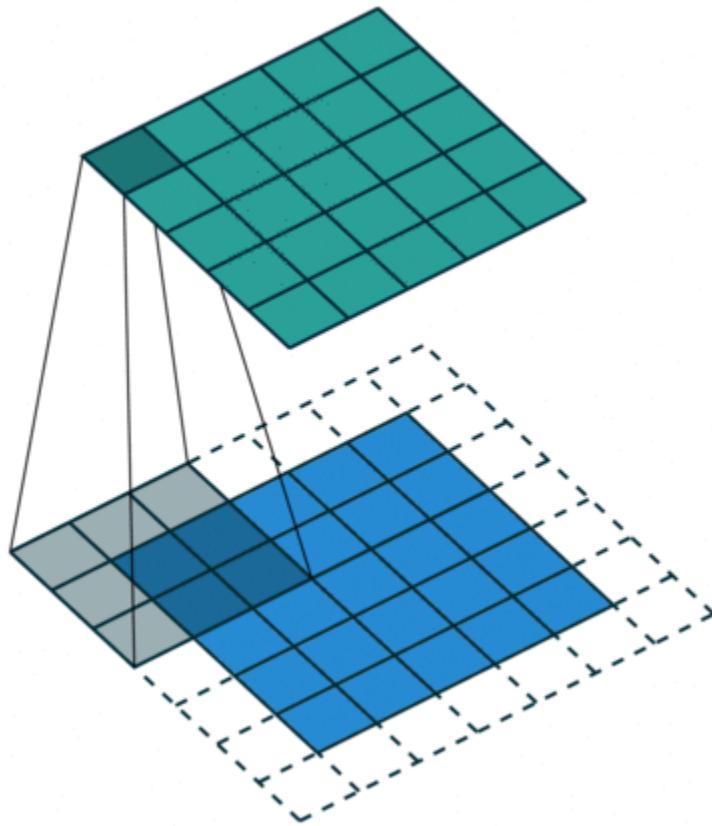
As a filter moves along the input, it uses the same set of weights and the same bias for the convolution, forming a *feature map*. Each feature map is the result of a convolution using a different set of weights and a different bias. Hence, the number of feature maps is equal to the number of filters.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

Zero Padding

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

the borders of an image input. By adjusting the padding, you can control the output size of the layer.

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



Output Size

The output height and width of a convolutional layer is $(Input\ Size - ((Filter\ Size - 1)*Dilation\ Factor + 1) + 2*Padding)/Stride + 1$. This value must be an integer for the whole image to be fully covered. If the combination of these options does not lead to an integer result, the convolution will be truncated or zero-padded to ensure the output size is an integer.

image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edges in the convolution.

Number of Neurons

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

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

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

Learning Parameters

You can adjust the learning rates and regularization options for the layer using name-value pair arguments while defining the convolutional layer. If you choose not to specify these options, then `trainNetwork` uses the global training options defined with the `trainingOptions` function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-52.

Number of Layers

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

Batch Normalization Layer

Create a batch normalization layer using `batchNormalizationLayer`.

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

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

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

ReLU Layer

Create a ReLU layer using `reluLayer`.

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

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

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

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

There are other nonlinear activation layers that perform different operations and can improve the network accuracy for some applications. For a list of activation layers, see "Activation Layers" on page 1-31.

Cross Channel Normalization (Local Response Normalization) Layer

Create a cross channel normalization layer using `crossChannelNormalizationLayer`.

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

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

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

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

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

Max and Average Pooling Layers

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

An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region. Create an average pooling layer using `averagePooling2dLayer`.

Pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers. They do not perform any learning

themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting.

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

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

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

Dropout Layer

Create a dropout layer using `dropoutLayer`.

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

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

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

Fully Connected Layer

Create a fully connected layer using `fullyConnectedLayer`.

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

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

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

You can also adjust the learning rate and the regularization parameters for this layer using the related name-value pair arguments when creating the fully connected layer. If you choose not to adjust them, then `trainNetwork` uses the global training parameters defined by the `trainingOptions` function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-52.

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

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

Output Layers

Softmax and Classification Layers

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

A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes. Create a classification layer using `classificationLayer`.

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

The output unit activation function is the softmax function:

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

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

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

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

where $0 \leq P(c_r|x, \theta) \leq 1$ and $\sum_{j=1}^k P(c_j|x, \theta) = 1$. Moreover, $a_r = \ln(P(x, \theta|c_r)P(c_r))$, $P(x, \theta|c_r)$

is the conditional probability of the sample given class r , and $P(c_r)$ is the class prior probability.

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

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

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

where N is the number of samples, K is the number of classes, t_{ij} is the indicator that the i th sample belongs to the j th class, and y_{ij} is the output for sample i for class j , which in

this case, is the value from the softmax function. That is, it is the probability that the network associates the i th input with class j .

Regression Layer

Create a regression layer using `regressionLayer`.

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

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

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

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

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

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

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

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

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

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

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

where S is the sequence length.

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

References

- [1] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.
- [2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol 25, 2012.
- [3] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D., et al. "Handwritten Digit Recognition with a Back-propagation Network." In *Advances of Neural Information Processing Systems*, 1990.
- [4] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based Learning Applied to Document Recognition." *Proceedings of the IEEE*. Vol 86, pp. 2278-2324, 1998.
- [5] Nair, V. and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines." In Proc. 27th International Conference on Machine Learning, 2010.
- [6] Nagi, J., F. Ducatelle, G. A. Di Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, L. M. Gambardella. "Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition". *IEEE International Conference on Signal and Image Processing Applications (ICSIPA2011)*, 2011.
- [7] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.
- [8] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

[9] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *preprint, arXiv:1502.03167* (2015).

See Also

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

More About

- “List of Deep Learning Layers” on page 1-28
- “Learn About Convolutional Neural Networks” on page 1-24
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-52
- “Resume Training from Checkpoint Network” on page 1-70
- “Create Simple Deep Learning Network for Classification”
- “Pretrained Deep Neural Networks” on page 1-15
- “Deep Learning in MATLAB” on page 1-2

Set Up Parameters and Train Convolutional Neural Network

In this section...

- “Specify Solver and Maximum Number of Epochs” on page 1-52
- “Specify and Modify Learning Rate” on page 1-53
- “Specify Validation Data” on page 1-54
- “Select Hardware Resource” on page 1-54
- “Save Checkpoint Networks and Resume Training” on page 1-55
- “Set Up Parameters in Convolutional and Fully Connected Layers” on page 1-55
- “Train Your Network” on page 1-56

After you define the layers of your neural network as described in “Specify Layers of Convolutional Neural Network” on page 1-37, the next step is to set up the training options for the network. Use the `trainingOptions` function to define the global training parameters. To train a network, use the object returned by `trainingOptions` as an input argument to the `trainNetwork` function. For example:

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

Layers with learnable parameters also have options for adjusting the learning parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers” on page 1-55.

Specify Solver and Maximum Number of Epochs

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

The ‘`adam`’ (derived from *adaptive moment estimation*) solver is often a good optimizer to try first. You can also try the ‘`rmsprop`’ (root mean square propagation) and ‘`sgdm`’ (stochastic gradient descent with momentum) optimizers and see if this improves

training. Different solvers work better for different problems. For more information about the different solvers, see “Stochastic Gradient Descent”.

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

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

Specify and Modify Learning Rate

You can specify the global learning rate by using the '`InitialLearnRate`' name-value pair argument of `trainingOptions`. By default, `trainNetwork` uses this value throughout the entire training process. You can choose to modify the learning rate every certain number of epochs by multiplying the learning rate with a factor. Instead of using a small, fixed learning rate throughout the training process, you can choose a larger learning rate in the beginning of training and gradually reduce this value during optimization. Doing so can shorten the training time, while enabling smaller steps towards the minimum of the loss as training progresses.

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

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

Specify Validation Data

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

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

To reduce overfitting, you can try adding data augmentation. Use an `augmentedImageDatastore` to perform random transformations on your input images. This helps to prevent the network from memorizing the exact position and orientation of objects. You can also try increasing the L₂ regularization using the '`L2Regularization`' name-value pair argument, using batch normalization layers after convolutional layers, and adding dropout layers.

Select Hardware Resource

If a GPU is available, then `trainNetwork` uses it for training, by default. Otherwise, `trainNetwork` uses a CPU. Alternatively, you can specify the execution environment you want using the '`ExecutionEnvironment`' name-value pair argument. You can specify a single CPU ('`cpu`'), a single GPU ('`gpu`'), multiple GPUs ('`multi-gpu`'), or a local parallel pool or compute cluster ('`parallel`'). All options other than '`cpu`' require Parallel Computing Toolbox. Training on a GPU requires a CUDA enabled GPU with compute capability 3.0 or higher.

Save Checkpoint Networks and Resume Training

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

`trainNetwork` automatically assigns unique names to checkpoint network files. In the example name, `net_checkpoint_351_2018_04_12_18_09_52.mat`, 351 is the iteration number, 2018_04_12 is the date, and 18_09_52 is the time at which `trainNetwork` saves the network. You can load a checkpoint network file by double-clicking it or using the `load` command at the command line. For example:

```
load net_checkpoint_351_2018_04_12_18_09_52.mat
```

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

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

You must manually specify the training options and the input data, because the checkpoint network does not contain this information. For an example, see "Resume Training from Checkpoint Network" on page 1-70.

Set Up Parameters in Convolutional and Fully Connected Layers

You can set the learning parameters to be different from the global values specified by `trainingOptions` in layers with learnable parameters, such as convolutional and fully connected layers. For example, to adjust the learning rate for the biases or weights, you can specify a value for the `BiasLearnRateFactor` or `WeightLearnRateFactor` properties of the layer, respectively. The `trainNetwork` function multiplies the learning rate that you specify by using `trainingOptions` with these factors. Similarly, you can also specify the L₂ regularization factors for the weights and biases in these layers by specifying the `BiasL2Factor` and `WeightL2Factor` properties, respectively. `trainNetwork` then multiplies the L₂ regularization factors that you specify by using `trainingOptions` with these factors.

Initialize Weights in Convolutional and Fully Connected Layers

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

Train Your Network

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

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

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

See Also

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

More About

- “Learn About Convolutional Neural Networks” on page 1-24
- “Specify Layers of Convolutional Neural Network” on page 1-37
- “Create Simple Deep Learning Network for Classification”
- “Resume Training from Checkpoint Network” on page 1-70

Deep Learning Tips and Tricks

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

Choose Network Architecture

The appropriate network architecture depends on the task and the data available.

Consider these suggestions when deciding which architecture to use and whether to use a pretrained network or to train from scratch.

Data	Description of Task	Learn More
Images	Classification of natural images	Try different pretrained networks. For a list of pretrained deep learning networks, see “Pretrained Deep Neural Networks” on page 1-15. To learn how to interactively prepare a network for transfer learning using Deep Network Designer, see “Transfer Learning with Deep Network Designer” on page 2-2.
	Regression of natural images	Try different pretrained networks. For an example showing how to convert a pretrained classification network into a regression network, see “Convert Classification Network into Regression Network”.

Data	Description of Task	Learn More
	Classification and regression of non-natural images (for example, tiny images and spectrograms)	<p>For an example showing how to classify tiny images, see “Train Residual Network for Image Classification”.</p> <p>For an example showing how to classify spectrograms, see “Speech Command Recognition Using Deep Learning”.</p>
	Semantic segmentation	Computer Vision Toolbox provides tools to create deep learning networks for semantic segmentation. For more information, see “Semantic Segmentation Basics” (Computer Vision Toolbox).
Sequences, time series, and signals	Sequence-to-label classification	For an example, see “Sequence Classification Using Deep Learning”.
	Sequence-to-sequence classification and regression	To learn more, see “Sequence-to-Sequence Classification Using Deep Learning” and “Sequence-to-Sequence Regression Using Deep Learning”.
	Time series forecasting	For an example, see “Time Series Forecasting Using Deep Learning”.
Text	Classification and regression	Text Analytics Toolbox provides tools to create deep learning networks for text data. For an example, see “Classify Text Data Using Deep Learning”.

Data	Description of Task	Learn More
	Text generation	For an example, see "Generate Text Using Deep Learning".
Audio	Audio classification and regression	For an example, see "Speech Command Recognition Using Deep Learning".

Choose Training Options

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

Tip	More Information
Monitor training progress	To turn on the training progress plot, set the 'Plots' option in <code>trainingOptions</code> to 'training-progress'.
Use validation data	<p>To specify validation data, use the 'ValidationData' option in <code>trainingOptions</code>.</p> <p>Note If your validation data set is too small and does not sufficiently represent the data, then the reported metrics might not help you. Using a too large validation data set can result in slower training.</p>

Tip	More Information
For transfer learning, speed up the learning of new layers and slow down the learning in the transferred layers	<p>Specify higher learning rate factors for new layers by using, for example, the <code>WeightLearnRateFactor</code> property of <code>convolution2dLayer</code>.</p> <p>Decrease the initial learning rate using the '<code>InitialLearnRate</code>' option of <code>trainingOptions</code>.</p> <p>When transfer learning, you do not need to train for as many epochs. Decrease the number of epochs using the '<code>MaxEpochs</code>' option in <code>trainingOptions</code>.</p> <p>To learn how to interactively prepare a network for transfer learning using Deep Network Designer, see "Transfer Learning with Deep Network Designer" on page 2-2.</p>
Shuffle your data every epoch	<p>To shuffle your data every epoch (one full pass of the data), set the '<code>Shuffle</code>' option in <code>trainingOptions</code> to '<code>every-epoch</code>'.</p> <p>Note For sequence data, shuffling can have a negative impact on the accuracy as it can increase the amount of padding or truncated data. If you have sequence data, then sorting the data by sequence length can help. To learn more, see "Sequence Padding, Truncation, and Splitting" on page 1-187.</p>
Try different optimizers	To specify different optimizers, use the <code>solverName</code> argument in <code>trainingOptions</code> .

For more information, see "Set Up Parameters and Train Convolutional Neural Network" on page 1-52.

Improve Training Accuracy

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

Problem	Possible Solution
NaNs or large spikes in the loss	Decrease the initial learning rate using the ' <code>InitialLearnRate</code> ' option of <code>trainingOptions</code> . If decreasing the learning rate does not help, then try using gradient clipping. To set the gradient threshold, use the ' <code>GradientThreshold</code> ' option in <code>trainingOptions</code> .
Loss is still decreasing at the end of training	Train for longer by increasing the number of epochs using the ' <code>MaxEpochs</code> ' option in <code>trainingOptions</code> .
Loss plateaus	If the loss plateaus at an unexpectedly high value, then drop the learning rate at the plateau. To change the learning rate schedule, use the ' <code>LearnRateSchedule</code> ' option in <code>trainingOptions</code> . If dropping the learning rate does not help, then the model might be underfitting. Try increasing the number of parameters or layers. You can check if the model is underfitting by monitoring the validation loss.

Problem	Possible Solution
Validation loss is much higher than the training loss	<p>To prevent overfitting, try one or more of the following:</p> <ul style="list-style-type: none">• Use data augmentation. For more information, see “Train Network with Augmented Images”.• Use dropout layers. For more information, see <code>dropoutLayer</code>.• Increase the global L2 regularization factor using the ‘L2Regularization’ option in <code>trainingOptions</code>.
Loss decreases very slowly	<p>Increase the initial learning rate using the ‘InitialLearnRate’ option of <code>trainingOptions</code>.</p> <p>For image data, try including batch normalization layers in your network. For more information, see <code>batchNormalizationLayer</code>.</p>

For more information, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-52.

Fix Errors in Training

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

Error	Description	Possible Solution
Out-of-memory error when training	The available hardware is unable to store the current mini-batch, the network weights, and the computed activations.	Try reducing the mini-batch size using the 'MiniBatchSize' option of <code>trainingOptions</code> . If reducing the mini-batch size does not work, then try using a smaller network, reducing the number of layers, or reducing the number of parameters or filters in the layers.
Custom layer errors	There could be an issue with the implementation of the custom layer.	Check the validity of the custom layer and find potential issues using <code>checkLayer</code> . If a test fails when you use <code>checkLayer</code> , then the function provides a test diagnostic and a framework diagnostic. The test diagnostic highlights any layer issues, whereas the framework diagnostic provides more detailed information. To learn more about the test diagnostics and get suggestions for possible solutions, see "Diagnostics" on page 1-169.

Error	Description	Possible Solution
Training throws the error 'CUDA_ERROR_UNKNOWN'	Sometimes, the GPU throws this error when it is being used for both compute and display requests from the OS.	Try reducing the mini-batch size using the 'MiniBatchSize' option of <code>trainingOptions</code> . If reducing the mini-batch size does not work, then in Windows®, try adjusting the Timeout Detection and Recovery (TDR) settings. For example, change the <code>TdrDelay</code> from 2 seconds (default) to 4 seconds (requires registry edit).

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

Prepare and Preprocess Data

You can improve the accuracy by preprocessing your data.

Weight or Balance Classes

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

For semantic segmentation tasks, you can specify class weights in `pixelClassificationLayer` using the `ClassWeights` property. For image

classification tasks, you can use the example custom classification layer provided in “Define Custom Weighted Classification Layer” on page 1-154.

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

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

Preprocess Image Data

For more information about preprocessing image data, see “Preprocess Images for Deep Learning” on page 1-201.

Task	More Information
Resize images	<p>To use a pretrained network, you must resize images to the input size of the network. To resize images, use <code>augmentedImageDatastore</code>. For example, this syntax resizes images in the image datastore <code>imds</code>:</p> <pre>auimds = augmentedImageDatastore(inputSize,imds)</pre> <p>Tip Use <code>augmentedImageDatastore</code> for efficient preprocessing of images for deep learning including image resizing.</p> <p>Do not use the <code>readFcn</code> option of <code>imageDatastore</code> as this option is usually significantly slower.</p>
Image augmentation	To avoid overfitting, use image transformation. To learn more, see “Train Network with Augmented Images”.

Task	More Information
Normalize regression targets	Normalize the predictors before you input them to the network. If you normalize the responses before training, then you must transform the predictions of the trained network to obtain the predictions of the original responses. For more information, see “Train Convolutional Neural Network for Regression”.

Preprocess Sequence Data

For more information about working with LSTM networks, see “Long Short-Term Memory Networks” on page 1-181.

Task	More Information
Normalize sequence data	To normalize sequence data, first calculate the per-feature mean and standard deviation for all the sequences. Then, for each training observation, subtract the mean value and divide by the standard deviation. To learn more, see “Normalize Sequence Data” on page 1-189.
Reduce sequence padding and truncation	To reduce the amount of padding or discarded data when padding or truncating sequences, try sorting your data by sequence length. To learn more, see “Sequence Padding, Truncation, and Splitting” on page 1-187.

Task	More Information
Specify mini-batch size and padding options for prediction	<p>When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network.</p> <p>To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options of the <code>classify</code>, <code>predict</code>, <code>classifyAndUpdateState</code>, and <code>predictAndUpdateState</code> functions.</p>

Use Available Hardware

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

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

Problem	More Information
Software does not use all available GPUs	If you have access to a machine with multiple GPUs, simply set the 'ExecutionEnvironment' option in <code>trainingOptions</code> to 'multi-gpu'. For more information, see "Deep Learning on Multiple GPUs" on page 3-2.

For more information, see "Scale Up Deep Learning in Parallel and in the Cloud" on page 3-2.

Fix Errors With Loading from MAT-Files

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

Warning: Unable to load instances of class `layerType` into a heterogeneous array. The definition of `layerType` could be missing or contain an error. Default objects will be substituted.

Warning: While loading an object of class 'SeriesNetwork': Error using 'forward' in Layer `nnet.cnn.layer.MissingLayer`. The function threw an error and could not be executed.

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

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

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

See Also

[Deep Network Designer](#) | [analyzeNetwork](#) | [checkLayer](#) | [trainingOptions](#)

More About

- “Pretrained Deep Neural Networks” on page 1-15
- “Preprocess Images for Deep Learning” on page 1-201
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Deep Learning Network to Classify New Images”
- “Convert Classification Network into Regression Network”

Resume Training from Checkpoint Network

This example shows how to save checkpoint networks while training a deep learning network and resume training from a previously saved network.

Load Sample Data

Load the sample data as a 4-D array. `digitTrain4DArrayData` loads the digit training set as 4-D array data. `XTrain` is a 28-by-28-by-1-by-5000 array, where 28 is the height and 28 is the width of the images. 1 is the number of channels and 5000 is the number of synthetic images of handwritten digits. `YTrain` is a categorical vector containing the labels for each observation.

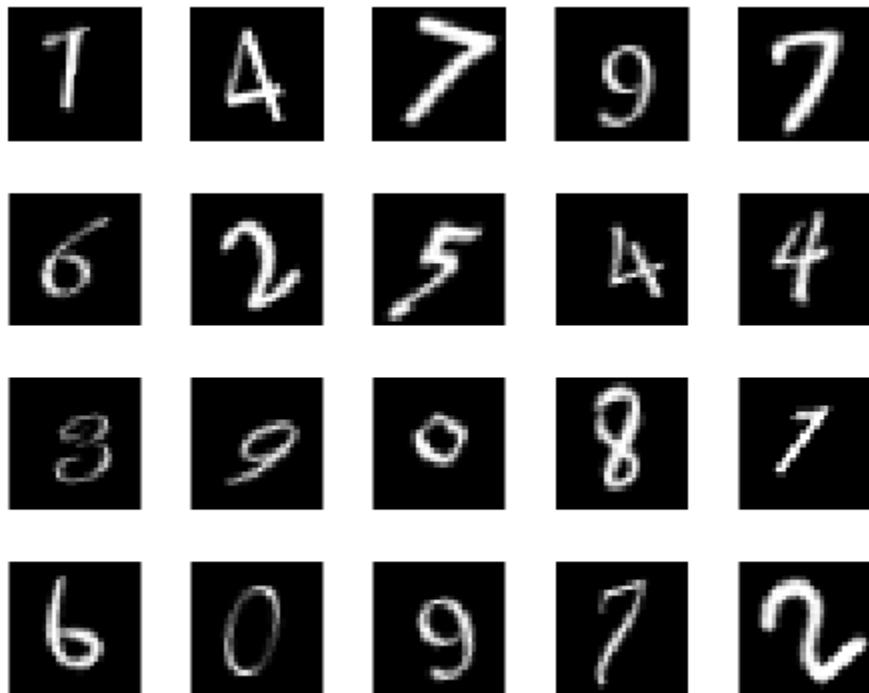
```
[XTrain,YTrain] = digitTrain4DArrayData;
size(XTrain)
```

```
ans = 1x4
```

```
28           28           1           5000
```

Display some of the images in `XTrain`.

```
figure;
perm = randperm(size(XTrain,4),20);
for i = 1:20
    subplot(4,5,i);
    imshow(XTrain(:,:,:,:perm(i)));
end
```



Define Network Architecture

Define the neural network architecture.

```
layers = [  
    imageInputLayer([28 28 1])  
  
    convolution2dLayer(3,8,'Padding','same')  
    batchNormalizationLayer  
    reluLayer  
    maxPooling2dLayer(2,'Stride',2)  
  
    convolution2dLayer(3,16,'Padding','same')  
    batchNormalizationLayer
```

```
reluLayer  
maxPooling2dLayer(2,'Stride',2)  
  
convolution2dLayer(3,32,'Padding','same')  
batchNormalizationLayer  
reluLayer  
averagePooling2dLayer(7)  
  
fullyConnectedLayer(10)  
softmaxLayer  
classificationLayer];
```

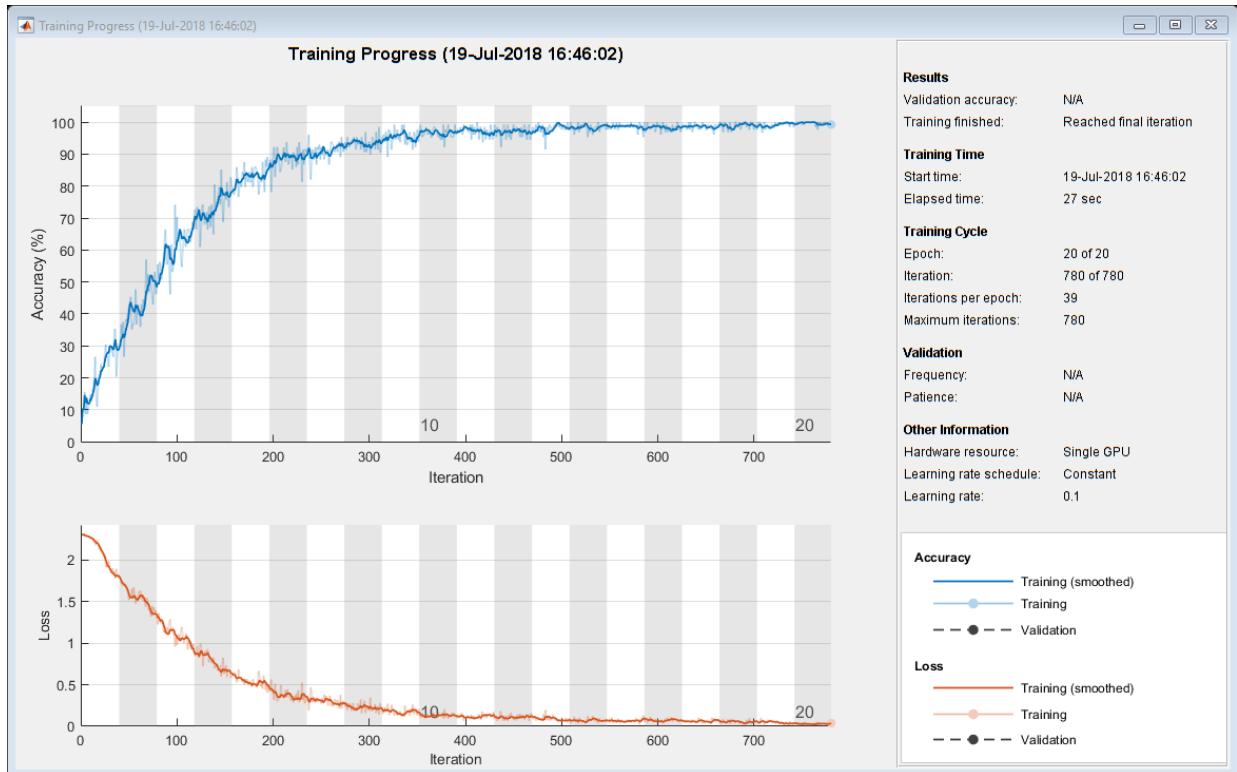
Specify Training Options and Train Network

Specify training options for stochastic gradient descent with momentum (SGDM) and specify the path for saving the checkpoint networks.

```
checkpointPath = pwd;  
options = trainingOptions('sgdm', ...  
    'InitialLearnRate',0.1, ...  
    'MaxEpochs',20, ...  
    'Verbose',false, ...  
    'Plots','training-progress', ...  
    'Shuffle','every-epoch', ...  
    'CheckpointPath',checkpointPath);
```

Train the network. `trainNetwork` uses a GPU if there is one available. If there is no available GPU, then it uses CPU. `trainNetwork` saves one checkpoint network each epoch and automatically assigns unique names to the checkpoint files.

```
net1 = trainNetwork(XTrain,YTrain,layers,options);
```



Load Checkpoint Network and Resume Training

Suppose that training was interrupted and did not complete. Rather than restarting the training from the beginning, you can load the last checkpoint network and resume training from that point. `trainNetwork` saves the checkpoint files with file names on the form `net_checkpoint_195_2018_07_13_11_59_10.mat`, where 195 is the iteration number, 2018_07_13 is the date, and 11_59_10 is the time `trainNetwork` saved the network. The checkpoint network has the variable name `net`.

Load the checkpoint network into the workspace.

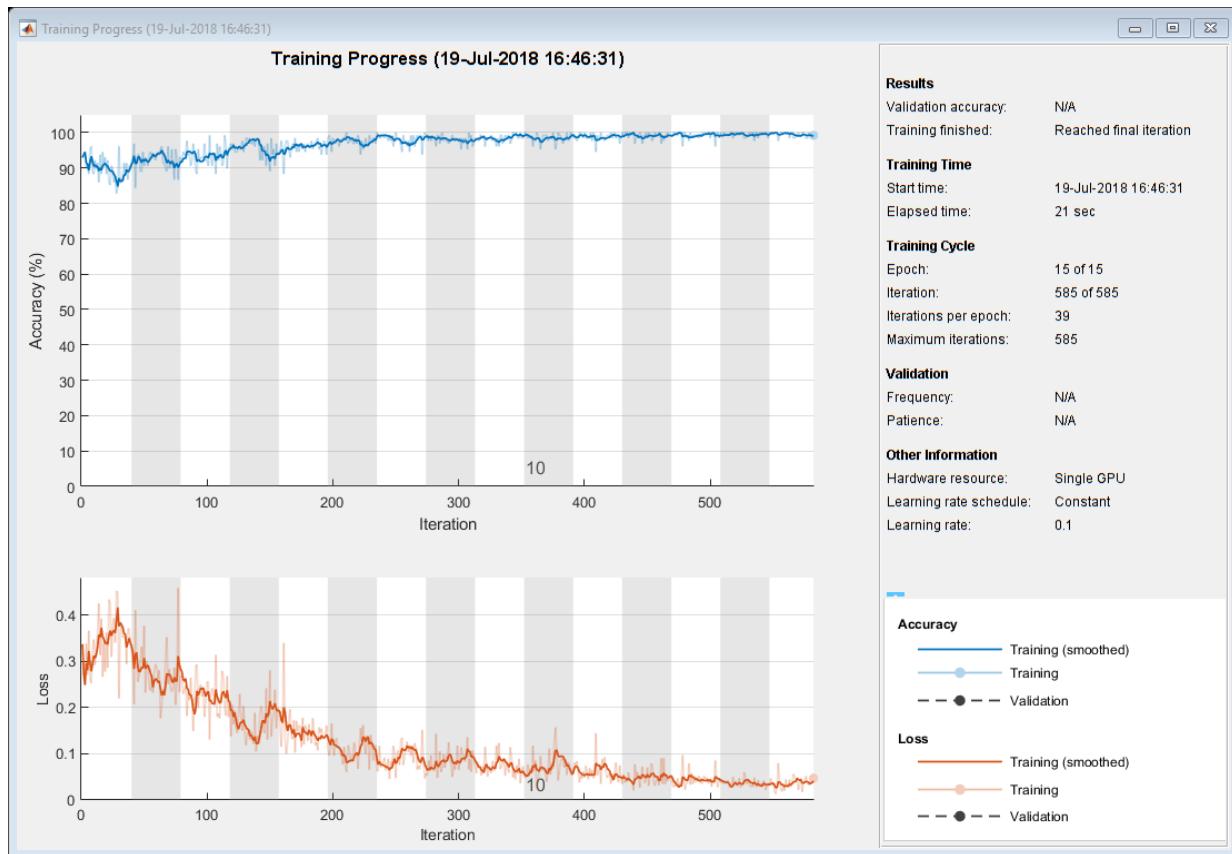
```
load('net_checkpoint_195_2018_07_13_11_59_10.mat','net')
```

Specify the training options and reduce the maximum number of epochs. You can also adjust other training options, such as the initial learning rate.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.1, ...
    'MaxEpochs',15, ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch', ...
    'CheckpointPath',checkpointPath);
```

Resume training using the layers of the checkpoint network you loaded with the new training options. If the checkpoint network is a DAG network, then use `layerGraph(net)` as the argument instead of `net.Layers`.

```
net2 = trainNetwork(XTrain,YTrain,net.Layers,options);
```



See Also

[trainNetwork](#) | [trainingOptions](#)

Related Examples

- “Create Simple Deep Learning Network for Classification”

More About

- “Learn About Convolutional Neural Networks” on page 1-24

- “Specify Layers of Convolutional Neural Network” on page 1-37
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-52

Define Custom Deep Learning Layers

Tip This topic explains how to define custom deep learning layers for your problems. For a list of built-in layers in Deep Learning Toolbox, see “List of Deep Learning Layers” on page 1-28.

This topic explains the architecture of deep learning layers and how to define custom layers to use for your problems.

Type	Description
Layer	<p>Define a custom deep learning layer and specify optional learnable parameters, forward functions, and a backward function.</p> <p>For an example showing how to define a custom layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97. For an example showing how to define a custom layer with multiple inputs, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114.</p>
Classification Output Layer	<p>Define a custom classification output layer and specify a loss function.</p> <p>For an example showing how to define a custom classification output layer and specify a loss function, see “Define Custom Classification Output Layer” on page 1-143.</p>
Regression Output Layer	<p>Define a custom regression output layer and specify a loss function.</p> <p>For an example showing how to define a custom regression output layer and specify a loss function, see “Define Custom Regression Output Layer” on page 1-131.</p>

Layer Templates

You can use the following templates to define new layers.

Intermediate Layer Template

This template outlines the structure of an intermediate layer with learnable parameters. If the layer does not have learnable parameters, then you can omit the **properties (learnable)** section. For an example showing how to define a layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97.

```
classdef myLayer < nnet.layer.Layer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Layer learnable parameters go here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Layer constructor function goes here.
        end

        function [Z1, ..., Zm] = predict(layer, X1, ..., Xn)
            % Forward input data through the layer at prediction time and
            % output the result.
            %
            % Inputs:
            %     layer      - Layer to forward propagate through
            %     X1, ..., Xn - Input data
            % Outputs:
            %     Z1, ..., Zm - Outputs of layer forward function

            % Layer forward function for prediction goes here.
        end

        function [Z1, ..., Zm, memory] = forward(layer, X1, ..., Xn)
            % (Optional) Forward input data through the layer at training
            % time and output the result and a memory value.
            %

            % Inputs:
```

```

%           layer      - Layer to forward propagate through
%           X1, ..., Xn - Input data
% Outputs:
%           Z1, ..., Zm - Outputs of layer forward function
%           memory      - Memory value for backward propagation

    % Layer forward function for training goes here.
end

function [dLdX1, ..., dLdXn, dLdW1, ..., dLdWk] = ...
    backward(layer, X1, ..., Xn, Z1, ..., Zm, dLdZ1, ..., dLdZm, memory)
    % Backward propagate the derivative of the loss function through
    % the layer.
    %
% Inputs:
%           layer      - Layer to backward propagate through
%           X1, ..., Xn - Input data
%           Z1, ..., Zm - Outputs of layer forward function
%           dLdZ1, ..., dLdZm - Gradients propagated from the next layers
%           memory      - Memory value from forward function
% Outputs:
%           dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
%                               inputs
%           dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
%                               learnable parameter

    % Layer backward function goes here.
end
end
end

```

Classification Output Layer Template

This template outlines the structure of a classification output layer with a loss function. For an example showing how to define a classification output layer and specify a loss function, see “Define Custom Classification Output Layer” on page 1-143.

```

classdef myClassificationLayer < nnet.layer.ClassificationLayer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

methods
    function layer = myClassificationLayer()
        % (Optional) Create a myClassificationLayer.

        % Layer constructor function goes here.
    end

    function loss = forwardLoss(layer, Y, T)
        % Return the loss between the predictions Y and the

```

```
% training targets T.  
%  
% Inputs:  
%     layer - Output layer  
%     Y      - Predictions made by network  
%     T      - Training targets  
%  
% Output:  
%     loss  - Loss between Y and T  
  
% Layer forward loss function goes here.  
end  
  
function dLdY = backwardLoss(layer, Y, T)  
    % Backward propagate the derivative of the loss function.  
    %  
    % Inputs:  
    %     layer - Output layer  
    %     Y      - Predictions made by network  
    %     T      - Training targets  
    %  
    % Output:  
    %     dLdY - Derivative of the loss with respect to the predictions Y  
  
    % Layer backward loss function goes here.  
end  
end  
end
```

Regression Output Layer Template

This template outlines the structure of a regression output layer with a loss function. For an example showing how to define a regression output layer and specify a loss function, see “Define Custom Regression Output Layer” on page 1-131.

```
classdef myRegressionLayer < nnet.layer.RegistrationLayer  
  
properties  
    % (Optional) Layer properties.  
  
    % Layer properties go here.  
end  
  
methods  
    function layer = myRegressionLayer()  
        % (Optional) Create a myRegressionLayer.  
  
        % Layer constructor function goes here.  
    end  
  
    function loss = forwardLoss(layer, Y, T)  
        % Return the loss between the predictions Y and the  
        % training targets T.  
        %  
        % Inputs:  
        %     layer - Output layer  
        %     Y      - Predictions made by network
```

```

%
%      T      - Training targets
%
% Output:
%      loss - Loss between Y and T

% Layer forward loss function goes here.
end

function dLdY = backwardLoss(layer, Y, T)
    % Backward propagate the derivative of the loss function.
%
% Inputs:
%      layer - Output layer
%      Y     - Predictions made by network
%      T     - Training targets
%
% Output:
%      dLdY - Derivative of the loss with respect to the predictions Y

% Layer backward loss function goes here.
end
end
end

```

Intermediate Layer Architecture

During training, the software iteratively performs forward and backward passes through the network.

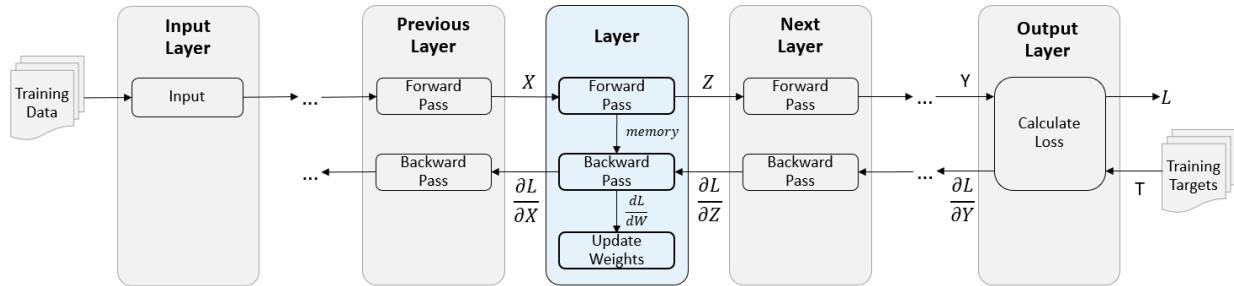
When making a forward pass through the network, each layer takes the outputs of the previous layers, applies a function, and then outputs (forward propagates) the results to the next layers.

Layers can have multiple inputs or outputs. For example, a layer can take X_1, \dots, X_n from multiple previous layers and forward propagate the outputs Z_1, \dots, Z_m to the next layers.

At the end of a forward pass of the network, the output layer calculates the loss L between the predictions Y and the true targets T .

During the backward pass of a network, each layer takes the derivatives of the loss with respect to the outputs of the layer, computes the derivatives of the loss L with respect to the inputs, and then backward propagates the results. If the layer has learnable parameters, then the layer also computes the derivatives of the layer weights (learnable parameters). The layer uses the derivatives of the weights to update the learnable parameters.

The following figure describes the flow of data through a deep neural network and highlights the data flow through a layer with a single input X , a single output Z , and a learnable parameter W .



Intermediate Layer Properties

Declare the layer properties in the **properties** section of the class definition.

By default, custom intermediate layers have these properties:

Property	Description
Name	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.
Description	One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer description, then the software displays the layer class name.
Type	Type of the layer, specified as a character vector or a string scalar. The value of Type appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer type, then the software displays the layer class name.

Property	Description
NumInputs	Number of inputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets NumInputs to the number of names in InputNames . The default value is 1.
InputNames	The input names of the layer specified as a cell array of character vectors. If you do not specify this value and NumInputs is greater than 1, then the software automatically sets InputNames to <code>{'in1', ..., 'inN'}</code> , where N is equal to NumInputs . The default value is <code>{'in'}</code> .
NumOutputs	Number of outputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets NumOutputs to the number of names in OutputNames . The default value is 1.
OutputNames	The output names of the layer specified as a cell array of character vectors. If you do not specify this value and NumOutputs is greater than 1, then the software automatically sets OutputNames to <code>{'out1', ..., 'outM'}</code> , where M is equal to NumOutputs . The default value is <code>{'out'}</code> .

If the layer has no other properties, then you can omit the **properties** section.

Tip If you are creating a layer with multiple inputs, then you must set either the **NumInputs** or **InputNames** in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the **NumOutputs** or **OutputNames** in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114.

Learnable Parameters

Declare the layer learnable parameters in the `properties (Learnable)` section of the class definition. If the layer has no learnable parameters, then you can omit the `properties (Learnable)` section.

Optionally, you can specify the learning rate factor and the L2 factor of the learnable parameters. By default, each learnable parameter has its learning rate factor and L2 factor set to 1.

For both built-in and custom layers, you can set and get the learn rate factors and L2 regularization factors using the following functions.

Function	Description
<code>setLearnRateFactor</code>	Set the learn rate factor of a learnable parameter.
<code>setL2Factor</code>	Set the L2 regularization factor of a learnable parameter.
<code>getLearnRateFactor</code>	Get the learn rate factor of a learnable parameter.
<code>getL2Factor</code>	Get the L2 regularization factor of a learnable parameter.

To specify the learning rate factor and the L2 factor of a learnable parameter, use the syntaxes `layer = setLearnRateFactor(layer, 'MyParameterName', value)` and `layer = setL2Factor(layer, 'MyParameterName', value)`, respectively.

To get the value of the learning rate factor and the L2 factor of a learnable parameter, use the syntaxes `getLearnRateFactor(layer, 'MyParameterName')` and `getL2Factor(layer, 'MyParameterName')` respectively.

For example, this syntax sets the learn rate factor of the learnable parameter with the name 'Alpha' to 0.1.

```
layer = setLearnRateFactor(layer, 'Alpha', 0.1);
```

Forward Functions

A layer uses one of two functions to perform a forward pass: `predict` or `forward`. If the forward pass is at prediction time, then the layer uses the `predict` function. If the

forward pass is at training time, then the layer uses the `forward` function. The `forward` function has an additional output argument `memory`, which you can use during backward propagation.

If you do not require two different functions for prediction time and training time, then you can omit the `forward` function. In this case, the layer uses `predict` at training time.

The syntax for `predict` is

```
[Z1,...,Zm] = predict(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs and Z_1, \dots, Z_m are the m layer outputs. The values n and m must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` can vary, then use `varargin` instead of X_1, \dots, X_n . In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i . If the number of outputs can vary, then use `varargout` instead of Z_1, \dots, Z_m . In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to Z_j .

The syntax for `forward` is

```
[Z1,...,Zm,mem] = forward(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m layer outputs, and `memory` is the memory of the layer.

Tip If the number of inputs to `forward` can vary, then use `varargin` instead of X_1, \dots, X_n . In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i . If the number of outputs can vary, then use `varargout` instead of Z_1, \dots, Z_m . In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to Z_j for $j=1, \dots, \text{NumOutputs}$ and `varargout{NumOutputs+1}` corresponds to `memory`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
2-D images	$h\text{-by-}w\text{-by-}c\text{-by-}N$, where h , w , and c correspond to the height, width, and number of channels of the images respectively, and N is the number of observations.	4
3-D images	$h\text{-by-}w\text{-by-}D\text{-by-}c\text{-by-}N$, where h , w , D , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, and N is the number of observations.	5
Vector sequences	$c\text{-by-}N\text{-by-}S$, where c is the number of features of the sequences, N is the number of observations, and S is the sequence length.	2
2-D image sequences	$h\text{-by-}w\text{-by-}c\text{-by-}N\text{-by-}S$, where h , w , and c correspond to the height, width, and number of channels of the images respectively, N is the number of observations, and S is the sequence length.	4
3-D image sequences	$h\text{-by-}w\text{-by-}d\text{-by-}c\text{-by-}N\text{-by-}S$, where h , w , d , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, N is the number of observations, and S is the sequence length.	5

Backward Function

The layer uses one function for a backward pass: `backward`. The `backward` function computes the derivatives of the loss with respect to the input data and then outputs (backward propagates) results to the previous layer. If the layer has learnable parameters, then `backward` also computes the derivatives of the layer weights (learnable parameters). During the backward pass, the layer automatically updates the learnable parameters using these derivatives.

To calculate the derivatives of the loss, you can use the chain rule:

$$\frac{\partial L}{\partial X^{(i)}} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial X^{(i)}}$$

$$\frac{\partial L}{\partial W_i} = \sum_j \frac{\partial L}{\partial Z_j} \frac{\partial Z_j}{\partial W_i}$$

The syntax for `backward` is

```
[dLdX1, ..., dLdXn, dLdW1, ..., dLdWk] = backward(layer, X1, ..., Xn, Z1, ..., Zm, dLdZ1, ..., dLdZm, memory)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m outputs of `forward`, $dLdZ_1, \dots, dLdZ_m$ are the gradients backward propagated from the next layer, and `memory` is the memory output of `forward`. For the outputs, $dLdX_1, \dots, dLdX_n$ are the derivatives of the loss with respect to the layer inputs and $dLdW_1, \dots, dLdW_k$ are the derivatives of the loss with respect to the k learnable parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i for $i=1, \dots, \text{NumInputs}$, `varargin{NumInputs+j}` and `varargin{NumInputs+NumOutputs+j}` correspond to Z_j and $dLdZ_j$, respectively, for $j=1, \dots, \text{NumOutputs}$, and `varargin{end}` corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where `varargout{i}` corresponds to $dLdX_i$ for $i=1, \dots, \text{NumInputs}$ and `varargout{NumInputs+t}` corresponds to $dLdW_t$ for $t=1, \dots, k$, where k is the number of learnable parameters.

The values of X_1, \dots, X_n and Z_1, \dots, Z_m are the same as in the forward functions. The dimensions of $dLdZ_1, \dots, dLdZ_m$ are the same as the dimensions of Z_1, \dots, Z_m , respectively.

The dimensions and data type of $dLdX_1, \dots, dLdX_n$ are the same as the dimensions and data type of X_1, \dots, X_n , respectively. The dimensions and data types of $dLdW_1, \dots, dLdW_k$ are the same as the dimensions and data types of W_1, \dots, W_k , respectively.

During the backward pass, the layer automatically updates the learnable parameters using the derivatives $dLdW_1, \dots, dLdW_k$.

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Check Validity of Layer

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, and correctly defined gradients. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer, and `dim` specifies the dimension of the observations in the layer input data. For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

For more information, see “Check Custom Layer Validity” on page 1-165.

Check Validity of Layer Using `checkLayer`

Check the layer validity of the custom layer `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set `'ObservationDimension'` to 4.

```
layer = preluLayer(20,'prelu');
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

Skipping GPU tests. No compatible GPU device found.

Running `nnet.checklayer.TestCase`

.....
Done `nnet.checklayer.TestCase`

```
Test Summary:
 18 Passed, 0 Failed, 0 Incomplete, 6 Skipped.
 Time elapsed: 169.4468 seconds.
```

Here, the function does not detect any issues with the layer.

Include Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create a layer array that includes the custom layer `preluLayer`.

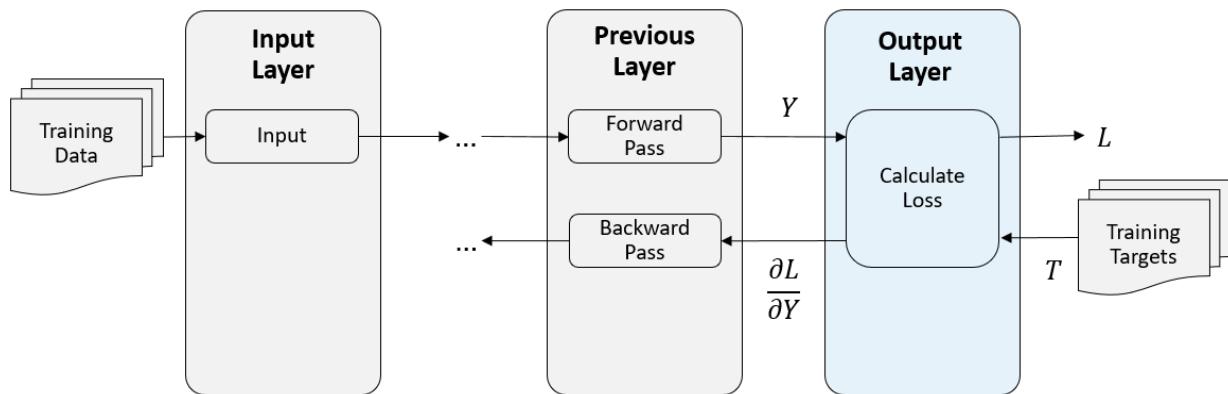
```
layers = [
  imageInputLayer([28 28 1])
  convolution2dLayer(5,20)
  batchNormalizationLayer
```

```
preluLayer(20, 'prelu')
fullyConnectedLayer(10)
softmaxLayer
classificationLayer];
```

Output Layer Architecture

At the end of a forward pass at training time, an output layer takes the predictions (outputs) y of the previous layer and calculates the loss L between these predictions and the training targets. The output layer computes the derivatives of the loss L with respect to the predictions y and outputs (backward propagates) results to the previous layer.

The following figure describes the flow of data through a convolutional neural network and an output layer.



Output Layer Properties

Declare the layer properties in the `properties` section of the class definition.

By default, custom output layers have the following properties:

- **Name** – Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
- **Description** – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If

you do not specify a layer description, then the software displays "Classification Output" or "Regression Output".

- **Type** - Type of the layer, specified as a character vector or a string scalar. The value of **Type** appears when the layer is displayed in a **Layer** array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- **Classes** - Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If **Classes** is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors **str**, then the software sets the classes of the output layer to **categorical(str,str)**. The default value is 'auto'.

Custom regression layers also have the following property:

- **ResponseNames** - Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

If the layer has no other properties, then you can omit the **properties** section.

Loss Functions

The output layer uses two functions to compute the loss and the derivatives: **forwardLoss** and **backwardLoss**. The **forwardLoss** function computes the loss **L**. The **backwardLoss** function computes the derivatives of the loss with respect to the predictions.

The syntax for **forwardLoss** is **loss = forwardLoss(layer, Y, T)**. The input **Y** corresponds to the predictions made by the network. These predictions are the output of the previous layer. The input **T** corresponds to the training targets. The output **loss** is the loss between **Y** and **T** according to the specified loss function. The output **loss** must be scalar.

The syntax for **backwardLoss** is **dLdY = backwardLoss(layer, Y, T)**. The inputs **Y** are the predictions made by the network and **T** are the training targets. The output **dLdY** is the derivative of the loss with respect to the predictions **Y**. The output **dLdY** must be the same size as the layer input **Y**.

For classification problems, the dimensions of **T** depend on the type of problem.

Classification Task	Input Size	Observation Dimension
2-D image classification	1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	4
3-D image classification	1-by-1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	5
Sequence-to-label classification	K -by- N , where K is the number of classes and N is the number of observations.	2
Sequence-to-sequence classification	K -by- N -by- S , where K is the number of classes, N is the number of observations, and S is the sequence length.	2

The size of Y depends on the output of the previous layer. To ensure that Y is the same size as T , you must include a layer that outputs the correct size before the output layer. For example, to ensure that Y is a 4-D array of prediction scores for K classes, you can include a fully connected layer of size K followed by a softmax layer before the output layer.

For regression problems, the dimensions of T also depend on the type of problem.

Regression Task	Input Size	Observation Dimension
2-D image regression	1-by-1-by- R -by- N , where R is the number of responses and N is the number of observations.	4
2-D Image-to-image regression	h -by- w -by- c -by- N , where h , w , and c are the height, width, and number of channels of the output respectively, and N is the number of observations.	4

Regression Task	Input Size	Observation Dimension
3-D image regression	1-by-1-by-1-by- R -by- N , where R is the number of responses and N is the number of observations.	5
3-D Image-to-image regression	h -by- w -by- d -by- c -by- N , where h , w , d , and c are the height, width, depth, and number of channels of the output respectively, and N is the number of observations.	5
Sequence-to-one regression	R -by- N , where R is the number of responses and N is the number of observations.	2
Sequence-to-sequence regression	R -by- N -by- S , where R is the number of responses, N is the number of observations, and S is the sequence length.	2

For example, if the network defines an image regression network with one response and has mini-batches of size 50, then T is a 4-D array of size 1-by-1-by-1-by-50.

The size of Y depends on the output of the previous layer. To ensure that Y is the same size as T , you must include a layer that outputs the correct size before the output layer. For example, for image regression with R responses, to ensure that Y is a 4-D array of the correct size, you can include a fully connected layer of size R before the output layer.

The `forwardLoss` and `backwardLoss` functions have the following output arguments.

Function	Output Argument	Description
<code>forwardLoss</code>	<code>loss</code>	Calculated loss between the predictions Y and the true target T .
<code>backwardLoss</code>	<code>dLdY</code>	Derivative of the loss with respect to the predictions Y .

If you want to include a custom output layer after a built-in layer, then `backwardLoss` must output `dLdY` with the size expected by the previous layer. Built-in layers expect `dLdY` to be the same size as `Y`.

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Include Custom Regression Output Layer in Network

You can use a custom output layer in the same way as any other output layer in Deep Learning Toolbox. This section shows how to create and train a network for regression using a custom output layer.

The example constructs a convolutional neural network architecture, trains a network, and uses the trained network to predict angles of rotated, handwritten digits. These predictions are useful for optical character recognition.

Define a custom mean absolute error regression layer. To create this layer, save the file `maeRegressionLayer.m` in the current folder.

Load the example training data.

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

Create a layer array and include the custom regression output layer `maeRegressionLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(1)
    maeRegressionLayer('mae')]
```

```
layers =
6x1 Layer array with layers:

1    ''      Image Input          28x28x1 images with 'zerocenter' normalization
2    ''      Convolution        20 5x5 convolutions with stride [1 1] and paddi
3    ''      Batch Normalization  Batch normalization
4    ''      ReLU                ReLU
5    ''      Fully Connected     1 fully connected layer
6    'mae'   Regression Output  Mean absolute error
```

Set the training options and train the network.

```
options = trainingOptions('sgdm','Verbose',false);
net = trainNetwork(XTrain,YTrain,layers,options);
```

Evaluate the network performance by calculating the prediction error between the predicted and actual angles of rotation.

```
[XTest,~,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);
predictionError = YTest - YPred;
```

Calculate the number of predictions within an acceptable error margin from the true angles. Set the threshold to 10 degrees and calculate the percentage of predictions within this threshold.

```
thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numTestImages = size(XTest,4);
accuracy = numCorrect/numTestImages

accuracy = 0.7872
```

See Also

[assembleNetwork](#) | [checkLayer](#) | [getL2Factor](#) | [getLearnRateFactor](#) |
[setL2Factor](#) | [setLearnRateFactor](#)

More About

- “Check Custom Layer Validity” on page 1-165
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97

- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Regression Output Layer” on page 1-131
- “Define Custom Weighted Classification Layer” on page 1-154
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Define Custom Deep Learning Layer with Learnable Parameters

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-28.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

- 1 Name the layer – give the layer a name so that it can be used in MATLAB.
- 2 Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
- 3 Create a constructor function (optional) – specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
- 4 Create forward functions – specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
- 5 Create a backward function – specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation).

This example shows how to create a PReLU layer, which is a layer with a learnable parameter and use it in a convolutional neural network. A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar learned at training time.[1] For values less than zero, a PReLU layer applies scaling coefficients α_i to each channel of the input. These coefficients form a learnable parameter, which the layer learns during training.

This figure from [1] compares the ReLU and PReLU layer functions.

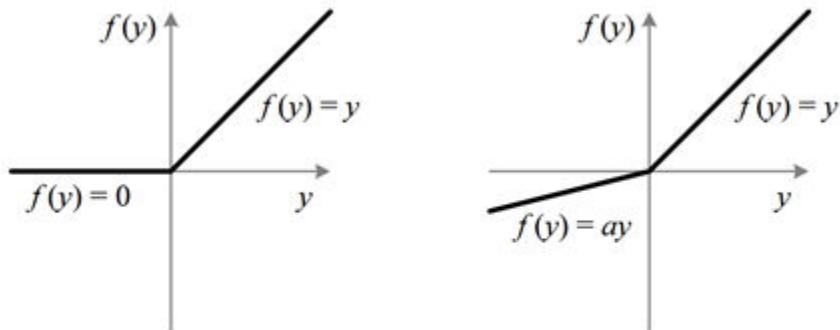


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Layer with Learnable Parameters Template

Copy the layer with learnable parameters template into a new file in MATLAB. This template outlines the structure of a layer with learnable parameters and includes the functions that define the layer behavior.

```
classdef myLayer < nnet.layer.Layer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

properties (Learnable)
    % (Optional) Layer learnable parameters.

    % Layer learnable parameters go here.
end

methods
    function layer = myLayer()
        % (Optional) Create a myLayer.
        % This function must have the same name as the class.

        % Layer constructor function goes here.
    end

    function [Z1, ..., Zm] = predict(layer, X1, ..., Xn)
        % Forward input data through the layer at prediction time and
        % output the result.
    end

```

```

%
% Inputs:
%     layer      - Layer to forward propagate through
%     X1, ..., Xn - Input data
% Outputs:
%     Z1, ..., Zm - Outputs of layer forward function

% Layer forward function for prediction goes here.
end

function [Z1, ..., Zm, memory] = forward(layer, X1, ..., Xn)
    % (Optional) Forward input data through the layer at training
    % time and output the result and a memory value.
    %
% Inputs:
%     layer      - Layer to forward propagate through
%     X1, ..., Xn - Input data
% Outputs:
%     Z1, ..., Zm - Outputs of layer forward function
%     memory      - Memory value for backward propagation

% Layer forward function for training goes here.
end

function [dLdX1, ..., dLdXn, dLdW1, ..., dLdWk] = ...
    backward(layer, X1, ..., Xn, Z1, ..., Zm, dLdZ1, ..., dLdZm, memory)
    % Backward propagate the derivative of the loss function through
    % the layer.
    %
% Inputs:
%     layer      - Layer to backward propagate through
%     X1, ..., Xn - Input data
%     Z1, ..., Zm - Outputs of layer forward function
%     dLdZ1, ..., dLdZm - Gradients propagated from the next layers
%     memory      - Memory value from forward function
% Outputs:
%     dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
%                         inputs
%     dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
%                         learnable parameter

% Layer backward function goes here.
end
end
end

```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `preluLayer`.

```
classdef preluLayer < nnet.layer.Layer
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
methods
    function layer = preluLayer()
        ...
    end

    ...
end
```

Save the Layer

Save the layer class file in a new file named `preluLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties:

Property	Description
Name	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and <code>Name</code> is set to ' ', then the software automatically assigns a name to the layer at training time.

Property	Description
Description	One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer description, then the software displays the layer class name.
Type	Type of the layer, specified as a character vector or a string scalar. The value of <code>Type</code> appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer type, then the software displays the layer class name.
NumInputs	Number of inputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets <code>NumInputs</code> to the number of names in <code>InputNames</code> . The default value is 1.
InputNames	The input names of the layer specified as a cell array of character vectors. If you do not specify this value and <code>NumInputs</code> is greater than 1, then the software automatically sets <code>InputNames</code> to <code>{'in1', ..., 'inN'}</code> , where <code>N</code> is equal to <code>NumInputs</code> . The default value is <code>{'in'}</code> .
NumOutputs	Number of outputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets <code>NumOutputs</code> to the number of names in <code>OutputNames</code> . The default value is 1.

Property	Description
OutputNames	The output names of the layer specified as a cell array of character vectors. If you do not specify this value and NumOutputs is greater than 1, then the software automatically sets OutputNames to <code>{'out1', ..., 'outM'}</code> , where M is equal to NumOutputs. The default value is <code>{'out'}</code> .

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114.

A PReLU layer does not require any additional properties, so you can remove the `properties` section.

A PReLU layer has only one learnable parameter, the scaling coefficient a . Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Alpha`.

```
properties (Learnable)
    % Layer learnable parameters

    % Scaling coefficient
    Alpha
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The PReLU layer constructor function requires two inputs arguments: the number of channels of the expected input data and the layer name. The number of channels specifies

the size of the learnable parameter Alpha. Specify two input arguments named numChannels and name in the preluLayer function. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = preluLayer(numChannels, name)
    % layer = preluLayer(numChannels) creates a PReLU layer with
    % numChannels channels and specifies the layer name.

    ...
end ...
```

Initialize Layer Properties

Initialize the layer properties, including learnable parameters in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the Name property to the input argument name.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the Description property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "PReLU with " + numChannels + " channels";
```

For a PReLU layer, when the input values are negative, the layer multiplies each channel of the input by the corresponding channel of Alpha. Initialize the learnable parameter Alpha to be a random vector of size 1-by-1-by-numChannels. With the third dimension specified as size numChannels, the layer can use element-wise multiplication of the input in the forward function. Alpha is a property of the layer object, so you must assign the vector to layer.Alpha.

```
% Initialize scaling coefficient.
layer.Alpha = rand([1 1 numChannels]);
```

View the completed constructor function.

```
function layer = preluLayer(numChannels, name)
    % layer = preluLayer(numChannels, name) creates a PReLU layer
    % with numChannels channels and specifies the layer name.
```

```
% Set layer name.  
layer.Name = name;  
  
% Set layer description.  
layer.Description = "PReLU with " + numChannels + " channels";  
  
% Initialize scaling coefficient.  
layer.Alpha = rand([1 1 numChannels]);  
end
```

With this constructor function, the command `preluLayer(3, 'prelu')` creates a PReLU layer with three channels and the name '`prelu`'.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The syntax for `predict` is

```
[Z1,...,Zm] = predict(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs and Z_1, \dots, Z_m are the m layer outputs. The values n and m must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` can vary, then use `varargin` instead of X_1, \dots, X_n . In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i . If the number of outputs can vary, then use `varargout` instead of Z_1, \dots, Z_m . In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to Z_j .

Because a PReLU layer has only one input and one output, the syntax for `predict` for a PReLU layer is `Z = predict(layer,X)`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for the backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
2-D images	h -by- w -by- c -by- N , where h , w , and c correspond to the height, width, and number of channels of the images respectively, and N is the number of observations.	4
3-D images	h -by- w -by- D -by- c -by- N , where h , w , D , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, and N is the number of observations.	5
Vector sequences	c -by- N -by- S , where c is the number of features of the sequences, N is the number of observations, and S is the sequence length.	2
2-D image sequences	h -by- w -by- c -by- N -by- S , where h , w , and c correspond to the height, width, and number of channels of the images respectively, N is the number of observations, and S is the sequence length.	4
3-D image sequences	h -by- w -by- d -by- c -by- N -by- S , where h , w , d , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, N is the number of observations, and S is the sequence length.	5

The **forward** function propagates the data forward through the layer at *training time* and also outputs a memory value.

The syntax for **forward** is

```
[Z1,...,Zm,memory] = forward(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m layer outputs, and **memory** is the memory of the layer.

Tip If the number of inputs to **forward** can vary, then use **varargin** instead of X_1, \dots, X_n . In this case, **varargin** is a cell array of the inputs, where **varargin**{ i } corresponds to X_i . If the number of outputs can vary, then use **varargout** instead of Z_1, \dots, Z_m . In this case, **varargout** is a cell array of the outputs, where **varargout**{ j } corresponds to Z_j for $j=1, \dots, \text{NumOutputs}$ and **varargout**{ $\text{NumOutputs}+1$ } corresponds to **memory**.

The PReLU operation is given by

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ \alpha_i x_i & \text{if } x_i \leq 0 \end{cases}$$

where x_i is the input of the nonlinear activation f on channel i , and α_i is the coefficient controlling the slope of the negative part. The subscript i in α_i indicates that the nonlinear activation can vary on different channels.

Implement this operation in **predict**. In **predict**, the input X corresponds to x in the equation. The output Z corresponds to $f(x_i)$. The PReLU layer does not require memory or a different forward function for training, so you can remove the **forward** function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

```
function Z = predict(layer, X)
    % Z = predict(layer, X) forwards the input data X through the
    % layer and outputs the result Z.

    Z = max(0, X) + layer.Alpha .* min(0, X);
end
```

Create Backward Function

Implement the derivatives of the loss with respect to the input data and the learnable parameters in the `backward` function.

The syntax for `backward` is

```
[dLdX1,...,dLdXn,dLdW1,...,dLdWk] = backward(layer,X1,...,Xn,Z1,...,Zm,dLdZ1,...,dLdZm,memory)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m outputs of `forward`, $dLdZ_1, \dots, dLdZ_m$ are the gradients backward propagated from the next layer, and `memory` is the memory output of `forward`. For the outputs, $dLdX_1, \dots, dLdX_n$ are the derivatives of the loss with respect to the layer inputs and $dLdW_1, \dots, dLdW_k$ are the derivatives of the loss with respect to the k learnable parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i for $i=1, \dots, \text{NumInputs}$, `varargin{NumInputs+j}` and `varargin{NumInputs+NumOutputs+j}` correspond to Z_j and $dLdZ_j$, respectively, for $j=1, \dots, \text{NumOutputs}$, and `varargin{end}` corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where `varargout{i}` corresponds to $dLdX_i$ for $i=1, \dots, \text{NumInputs}$ and `varargout{NumInputs+t}` corresponds to $dLdW_t$ for $t=1, \dots, k$, where k is the number of learnable parameters.

Because a PReLU layer has only one input, one output, and one learnable parameter, the syntax for `backward` for a PReLU layer is `[dLdX,dLdAlpha] = backward(layer,X,Z,dLdZ,memory)`. The dimensions of X and Z are the same as in the forward functions. The dimensions of $dLdZ$ are the same as the dimensions of Z . The dimensions and data type of $dLdX$ are the same as the dimensions and data type of X . The dimension and data type of $dLdAlpha$ is the same as the dimension and data type of the learnable parameter `Alpha`.

During the backward pass, the layer automatically updates the learnable parameters using the corresponding derivatives.

To include a custom layer in a network, the layer forward functions must accept the outputs of the previous layer and forward propagate arrays with the size expected by the

next layer. Similarly, `backward` must accept inputs with the same size as the corresponding output of the forward function and backward propagate derivatives with the same size.

The derivative of the loss with respect to the input data is

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial f(x_i)} \frac{\partial f(x_i)}{\partial x_i}$$

where $\partial L / \partial f(x_i)$ is the gradient propagated from the next layer, and the derivative of the activation is

$$\frac{\partial f(x_i)}{\partial x_i} = \begin{cases} 1 & \text{if } x_i \geq 0 \\ \alpha_i & \text{if } x_i < 0 \end{cases}.$$

The derivative of the loss with respect to the learnable parameters is

$$\frac{\partial L}{\partial \alpha_i} = \sum_j \frac{\partial L}{\partial f(x_{ij})} \frac{\partial f(x_{ij})}{\partial \alpha_i}$$

where i indexes the channels, j indexes the elements over height, width, and observations, and $\partial L / \partial f(x_i)$ is the gradient propagated from the deeper layer, and the gradient of the activation is

$$\frac{\partial f(x_i)}{\partial \alpha_i} = \begin{cases} 0 & \text{if } x_i \geq 0 \\ x_i & \text{if } x_i < 0 \end{cases}.$$

In `backward` of the layer template, replace the output `dLdW` with the output `dLdAlpha`, where `dLdAlpha` corresponds to $\partial L / \partial \alpha_i$. In `backward`, the input `X` corresponds to `x`. The input `Z` corresponds to $f(x_i)$. The input `dLdZ` corresponds to $\partial L / \partial f(x_i)$. The output `dLdX` corresponds to $\partial L / \partial x_i$.

Add a comment to the top of the function that explains the syntaxes of the function. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`. Because the layer function does not require the input arguments `Z` and `memory`, replace these arguments with `~`.

```
function [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
    % [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
```

```
% backward propagates the derivative of the loss function
% through the layer.
% Inputs:
%     layer    - Layer to backward propagate through
%     X        - Input data
%     dLdZ    - Gradient propagated from the deeper layer
% Outputs:
%     dLdX    - Derivative of the loss with respect to the
%               input data
%     dLdAlpha - Derivative of the loss with respect to the
%               learnable parameter Alpha

dLdX = layer.Alpha .* dLdZ;
dLdX(X>0) = dLdZ(X>0);
dLdAlpha = min(0,X) .* dLdZ;
dLdAlpha = sum(sum(dLdAlpha,1),2);

% Sum over all observations in mini-batch.
dLdAlpha = sum(dLdAlpha,4);
end
```

Completed Layer

View the completed layer class file.

```
classdef preluLayer < nnet.layer.Layer
    % Example custom PReLU layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficient
        Alpha
    end

    methods
        function layer = preluLayer(numChannels, name)
            % layer = preluLayer(numChannels, name) creates a PReLU layer
            % with numChannels channels and specifies the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
```

```
layer.Description = "PReLU with " + numChannels + " channels";

    % Initialize scaling coefficient.
    layer.Alpha = rand([1 1 numChannels]);
end

function Z = predict(layer, X)
    % Z = predict(layer, X) forwards the input data X through the
    % layer and outputs the result Z.

    Z = max(0, X) + layer.Alpha .* min(0, X);
end

function [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
    % [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
    % backward propagates the derivative of the loss function
    % through the layer.
    % Inputs:
    %         layer      - Layer to backward propagate through
    %         X         - Input data
    %         dLdZ     - Gradient propagated from the deeper layer
    % Outputs:
    %         dLdX      - Derivative of the loss with respect to the
    %                     input data
    %         dLdAlpha - Derivative of the loss with respect to the
    %                     learnable parameter Alpha

    dLdX = layer.Alpha .* dLdZ;
    dLdX(X>0) = dLdZ(X>0);
    dLdAlpha = min(0,X) .* dLdZ;
    dLdAlpha = sum(sum(dLdAlpha,1),2);

    % Sum over all observations in mini-batch.
    dLdAlpha = sum(dLdAlpha,4);
end
end
end
```

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at

least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `predict`, `forward`, and `backward` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Validity of Layer Using `checkLayer`

Check the layer validity of the custom layer `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set `'ObservationDimension'` to 4.

```
layer = preluLayer(20,'prelu');
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

Skipping GPU tests. No compatible GPU device found.

```
Running nnet.checklayer.TestCase
.....
Done nnet.checklayer.TestCase
```

```
Test Summary:
18 Passed, 0 Failed, 0 Incomplete, 6 Skipped.
Time elapsed: 169.4468 seconds.
```

Here, the function does not detect any issues with the layer.

Include Custom Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox. This section shows how to create and train a network for digit classification using the PReLU layer you created earlier.

Load the example training data.

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

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder. Create a layer array including the custom layer `preluLayer`.

```
layers = [  
    imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    batchNormalizationLayer  
    preluLayer(20,'prelu')  
    fullyConnectedLayer(10)  
    softmaxLayer  
    classificationLayer];
```

Set the training options and train the network.

```
options = trainingOptions('adam','MaxEpochs',10);  
net = trainNetwork(XTrain,YTrain,layers,options);
```

Training on single CPU.

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	7.03%	3.3828	0.00
2	50	00:00:12	74.22%	0.7206	0.00
3	100	00:00:24	89.84%	0.3583	0.00
4	150	00:00:35	88.28%	0.4036	0.00
6	200	00:00:48	96.88%	0.2033	0.00
7	250	00:00:59	96.88%	0.1368	0.00
8	300	00:01:10	100.00%	0.0608	0.00
9	350	00:01:22	100.00%	0.0533	0.00
10	390	00:01:31	99.22%	0.0528	0.00

Evaluate the network performance by predicting on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
accuracy = sum(YTest==YPred)/numel(YTest)

accuracy = 0.9194
```

References

- [1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification." *In Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

See Also

[assembleNetwork](#) | [checkLayer](#)

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Check Custom Layer Validity” on page 1-165
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Weighted Classification Layer” on page 1-154
- “Define Custom Regression Output Layer” on page 1-131
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Define Custom Deep Learning Layer with Multiple Inputs

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-28.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

- 1** Name the layer – give the layer a name so that it can be used in MATLAB.
- 2** Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
- 3** Create a constructor function (optional) – specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
- 4** Create forward functions – specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
- 5** Create a backward function – specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation).

This example shows how to create a weighted addition layer, which is a layer with multiple inputs and learnable parameter, and use it in a convolutional neural network. A weighted addition layer scales and adds inputs from multiple neural network layers element-wise.

Layer with Learnable Parameters Template

Copy the layer with learnable parameters template into a new file in MATLAB. This template outlines the structure of a layer with learnable parameters and includes the functions that define the layer behavior.

```
classdef myLayer < nnet.layer.Layer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

properties (Learnable)
    % (Optional) Layer learnable parameters.
```

```

    % Layer learnable parameters go here.
end

methods
    function layer = myLayer()
        % (Optional) Create a myLayer.
        % This function must have the same name as the class.

        % Layer constructor function goes here.
    end

    function [Z1, ..., Zm] = predict(layer, X1, ..., Xn)
        % Forward input data through the layer at prediction time and
        % output the result.
        %
        % Inputs:
        %     layer      - Layer to forward propagate through
        %     X1, ..., Xn - Input data
        % Outputs:
        %     Z1, ..., Zm - Outputs of layer forward function

        % Layer forward function for prediction goes here.
    end

    function [Z1, ..., Zm, memory] = forward(layer, X1, ..., Xn)
        % (Optional) Forward input data through the layer at training
        % time and output the result and a memory value.
        %
        % Inputs:
        %     layer      - Layer to forward propagate through
        %     X1, ..., Xn - Input data
        % Outputs:
        %     Z1, ..., Zm - Outputs of layer forward function
        %     memory      - Memory value for backward propagation

        % Layer forward function for training goes here.
    end

    function [dLdX1, ..., dLdXn, dLdW1, ..., dLdWk] = ...
        backward(layer, X1, ..., Xn, Z1, ..., Zm, dLdZ1, ..., dLdZm, memory)
        % Backward propagate the derivative of the loss function through
        % the layer.
        %
        % Inputs:
        %     layer      - Layer to backward propagate through
        %     X1, ..., Xn - Input data
        %     Z1, ..., Zm - Outputs of layer forward function
        %     dLdZ1, ..., dLdZm - Gradients propagated from the next layers
        %     memory      - Memory value from forward function
        % Outputs:
        %     dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
        %                         inputs
        %     dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
        %                         learnable parameter

```

```
        % Layer backward function goes here.  
    end  
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `weightedAdditionLayer`.

```
classdef weightedAdditionLayer < nnet.layer.Layer  
    ...  
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
methods  
    function layer = weightedAdditionLayer()  
        ...  
    end  
  
    ...  
end
```

Save the Layer

Save the layer class file in a new file named `weightedAdditionLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties:

Property	Description
Name	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.
Description	One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer description, then the software displays the layer class name.
Type	Type of the layer, specified as a character vector or a string scalar. The value of Type appears when the layer is displayed in a <code>Layer</code> array. If you do not specify a layer type, then the software displays the layer class name.
NumInputs	Number of inputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets <code>NumInputs</code> to the number of names in <code>InputNames</code> . The default value is 1.
InputNames	The input names of the layer specified as a cell array of character vectors. If you do not specify this value and <code>NumInputs</code> is greater than 1, then the software automatically sets <code>InputNames</code> to <code>{'in1', ..., 'inN'}</code> , where N is equal to <code>NumInputs</code> . The default value is <code>{'in'}</code> .
NumOutputs	Number of outputs of the layer specified as a positive integer. If you do not specify this value, then the software automatically sets <code>NumOutputs</code> to the number of names in <code>OutputNames</code> . The default value is 1.

Property	Description
OutputNames	The output names of the layer specified as a cell array of character vectors. If you do not specify this value and NumOutputs is greater than 1, then the software automatically sets OutputNames to <code>{'out1', ..., 'outM'}</code> , where M is equal to NumOutputs. The default value is <code>{'out'}</code> .

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` in the layer constructor.

A weighted addition layer does not require any additional properties, so you can remove the `properties` section.

A weighted addition layer has only one learnable parameter, the weights. Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Weights`.

```
properties (Learnable)
    % Layer learnable parameters

    % Scaling coefficients
    Weights
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The weighted addition layer constructor function requires two inputs: the number of inputs to the layer and the layer name. This number of inputs to the layer specifies the size of the learnable parameter `Weights`. Specify two input arguments named

`numInputs` and `name` in the `weightedAdditionLayer` function. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = weightedAdditionLayer(numInputs, name)
    % layer = weightedAdditionLayer(numInputs, name) creates a
    % weighted addition layer and specifies the number of inputs
    % and the layer name.

    ...
end
```

Initialize Layer Properties

Initialize the layer properties, including learnable parameters, in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the `NumInputs` property to the input argument `numInputs`.

```
% Set number of inputs.
layer.NumInputs = numInputs;
```

Set the `Name` property to the input argument `name`.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "Weighted addition of " + numInputs + ...
    " inputs";
```

A weighted addition layer multiplies each layer input by the corresponding coefficient in `Weights` and adds the resulting values together. Initialize the learnable parameter `Weights` to be a random vector of size 1-by-`numInputs`. `Weights` is a property of the layer object, so you must assign the vector to `layer.Weights`.

```
% Initialize layer weights
layer.Weights = rand(1,numInputs);
```

View the completed constructor function.

```
function layer = weightedAdditionLayer(numInputs,name)
    % layer = weightedAdditionLayer(numInputs,name) creates a
    % weighted addition layer and specifies the number of inputs
    % and the layer name.

    % Set number of inputs.
    layer.NumInputs = numInputs;

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    layer.Description = "Weighted addition of " + numInputs + ...
        " inputs";

    % Initialize layer weights.
    layer.Weights = rand(1,numInputs);
end
```

With this constructor function, the command `weightedAdditionLayer(3, 'add')` creates a weighted addition layer with three inputs and the name 'add'.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The syntax for `predict` is

```
[Z1,...,Zm] = predict(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs and Z_1, \dots, Z_m are the m layer outputs. The values n and m must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` can vary, then use `varargin` instead of X_1, \dots, X_n . In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i . If the number of outputs can vary, then use `varargout` instead of Z_1, \dots, Z_m . In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to Z_j .

Because a weighted addition layer has only one output and a variable number of inputs, the syntax for `predict` for a weighted addition layer is `Z = predict(layer, varargin)`, where `varargin{i}` corresponds to X_i for positive integers i less than or equal to `NumInputs`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for the backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
2-D images	h -by- w -by- c -by- N , where h , w , and c correspond to the height, width, and number of channels of the images respectively, and N is the number of observations.	4
3-D images	h -by- w -by- D -by- c -by- N , where h , w , D , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, and N is the number of observations.	5
Vector sequences	c -by- N -by- S , where c is the number of features of the sequences, N is the number of observations, and S is the sequence length.	2

Layer Input	Input Size	Observation Dimension
2-D image sequences	h -by- w -by- c -by- N -by- S , where h , w , and c correspond to the height, width, and number of channels of the images respectively, N is the number of observations, and S is the sequence length.	4
3-D image sequences	h -by- w -by- d -by- c -by- N -by- S , where h , w , d , and c correspond to the height, width, depth, and number of channels of the 3-D images respectively, N is the number of observations, and S is the sequence length.	5

The **forward** function propagates the data forward through the layer at *training time* and also outputs a memory value.

The syntax for **forward** is

```
[Z1,...,Zm,memory] = forward(layer,X1,...,Xn)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m layer outputs, and **memory** is the memory of the layer.

Tip If the number of inputs to **forward** can vary, then use **varargin** instead of X_1, \dots, X_n . In this case, **varargin** is a cell array of the inputs, where **varargin{i}** corresponds to X_i . If the number of outputs can vary, then use **varargout** instead of Z_1, \dots, Z_m . In this case, **varargout** is a cell array of the outputs, where **varargout{j}** corresponds to Z_j for $j=1,\dots,\text{NumOutputs}$ and **varargout{NumOutputs+1}** corresponds to **memory**.

The forward function of a weighted addition layer is

$$f(X^{(1)}, \dots, X^{(n)}) = \sum_{i=1}^n W_i X^{(i)}$$

where $X^{(1)}, \dots, X^{(n)}$ correspond to the layer inputs and W_1, \dots, W_n are the layer weights.

Implement the forward function in `predict`. In `predict`, the output Z corresponds to $f(X^{(1)}, \dots, X^{(n)})$. The weighted addition layer does not require memory or a different forward function for training, so you can remove the `forward` function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

Tip If you preallocate arrays using functions like `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type of another array, use the `'like'` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz, 'like', X)`.

```
function Z = predict(layer, varargin)
    % Z = predict(layer, X1, ..., Xn) forwards the input data X1,
    % ..., Xn through the layer and outputs the result Z.

    X = varargin;
    W = layer.Weights;

    % Initialize output
    X1 = X{1};
    sz = size(X1);
    Z = zeros(sz, 'like', X1);

    % Weighted addition
    for i = 1:layer.NumInputs
        Z = Z + W(i)*X{i};
    end
end
```

Create Backward Function

Implement the derivatives of the loss with respect to the input data and the learnable parameters in the `backward` function.

The syntax for `backward` is

```
[dLdX1,...,dLdXn,dLdW1,...,dLdWk] = backward(layer,X1,...,Xn,Z1,...,Zm,dLdZ1,...,dLdZm,memory)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m outputs of `forward`, $dLdZ_1, \dots, dLdZ_m$ are the gradients backward propagated from the next layer, and `memory` is the memory output of `forward`. For the outputs, $dLdX_1, \dots, dLdX_n$ are the derivatives of the loss with respect to the layer inputs and $dLdW_1, \dots, dLdW_k$ are the derivatives of the loss with respect to the k learnable parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i for $i=1, \dots, \text{NumInputs}$, `varargin{NumInputs+j}` and `varargin{NumInputs+NumOutputs+j}` correspond to Z_j and $dLdZ_j$, respectively, for $j=1, \dots, \text{NumOutputs}$, and `varargin{end}` corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where `varargout{i}` corresponds to $dLdX_i$ for $i=1, \dots, \text{NumInputs}$ and `varargout{NumInputs+t}` corresponds to $dLdW_t$ for $t=1, \dots, k$, where k is the number of learnable parameters.

Because a weighted addition layer has one output, one learnable parameter, and a variable number of inputs, the syntax for `backward` for a weighted addition layer is `varargout = backward(layer, varargin)`. In this case, `varargin{i}` corresponds to X_i for positive integers i less than or equal to `NumInputs`, `varargin{NumInputs+1}` corresponds to Z , and `varargin{NumInputs+2}` corresponds to $dLdZ$. For the outputs, `varargout{i}` corresponds to $dLdX_i$ for positive integers i less than or equal to `NumInputs` and `varargout{NumInputs+1}` corresponds to `dLdW`.

The dimensions of X_1, \dots, X_n and Z are the same as in the forward functions. The dimensions of $dLdZ$ are the same as the dimensions of Z . The dimensions and data types of $dLdX_1, \dots, dLdX_n$ are the same as the dimensions and data type of X_1, \dots, X_n . The dimension and data type of $dLdW$ is the same as the dimension and data type of the learnable parameter W .

During the backward pass, the layer automatically updates the learnable parameters using the corresponding derivatives.

To include a custom layer in a network, the layer forward functions must accept the outputs of the previous layer and forward propagate arrays with the size expected by the

next layer. Similarly, `backward` must accept inputs with the same size as the corresponding output of the forward function and backward propagate derivatives with the same size.

For a weighted addition layer, the derivatives of the loss with respect to each input are

$$\frac{\partial L}{\partial X_k^{(i)}} = \sum_j \frac{\partial L}{\partial Z_j} \frac{\partial Z_j}{\partial X_k^{(i)}},$$

where k indexes into each $X^{(i)}$ linearly, $Z = f(X^{(1)}, \dots, X^{(n)})$, $\partial L / \partial Z$ is the gradient propagated from the next layer, j indexes into Z linearly, and the derivative of the activation is

$$\frac{\partial Z}{\partial X^{(i)}} = W_i.$$

The derivative of the loss with respect to each learnable parameter W_i is

$$\frac{\partial L}{\partial W_i} = \sum_j \frac{\partial L}{\partial Z_j} \frac{\partial Z_j}{\partial W_i},$$

where j indexes the elements of Z linearly and the gradient of the activation is

$$\frac{\partial Z}{\partial W_i} = X^{(i)}.$$

Implement the backward function in `backward` and add a comment to the top of the function that explains the syntaxes of the function.

```
function varargout = backward(layer, varargin)
    % [dLdX1,...,dLdXn,dLdW] = backward(layer,X1,...,Xn,Z,dLdZ,~)
    % backward propagates the derivative of the loss function
    % through the layer.

    numInputs = layer.NumInputs;
    W = layer.Weights;
    X = varargin(1:numInputs);
    dLdZ = varargin{numInputs+2};

    % Calculate derivatives
    dLdX = cell(1,numInputs);
```

```
dLdW = zeros(1,numInputs,'like',W);
for i = 1:numInputs
    dLdX{i} = dLdZ * W(i);
    dLdW(i) = sum(dLdZ .* X{i}, 'all');
end

% Pack output arguments.
varargout(1:numInputs) = dLdX;
varargout{numInputs+1} = dLdW;
end
```

Completed Layer

View the completed layer class file.

```
classdef weightedAdditionLayer < nnet.layer.Layer
    % Example custom weighted addition layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficients
        Weights
    end

    methods
        function layer = weightedAdditionLayer(numInputs,name)
            % layer = weightedAdditionLayer(numInputs,name) creates a
            % weighted addition layer and specifies the number of inputs
            % and the layer name.

            % Set number of inputs.
            layer.NumInputs = numInputs;

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = "Weighted addition of " + numInputs + ...
                " inputs";

            % Initialize layer weights.
            layer.Weights = rand(1,numInputs);
    end
```

```
function Z = predict(layer, varargin)
    % Z = predict(layer, X1, ..., Xn) forwards the input data X1,
    % ..., Xn through the layer and outputs the result Z.

    X = varargin;
    W = layer.Weights;

    % Initialize output
    X1 = X{1};
    sz = size(X1);
    Z = zeros(sz,'like',X1);

    % Weighted addition
    for i = 1:layer.NumInputs
        Z = Z + W(i)*X{i};
    end
end

function varargout = backward(layer, varargin)
    % [dLdX1,...,dLdXn,dLdW] = backward(layer,X1,...,Xn,Z,dLdZ,~)
    % backward propagates the derivative of the loss function
    % through the layer.

    numInputs = layer.NumInputs;
    W = layer.Weights;
    X = varargin(1:numInputs);
    dLdZ = varargin{numInputs+2};

    % Calculate derivatives
    dLdX = cell(1,numInputs);
    dLdW = zeros(1,numInputs, 'like',W);
    for i = 1:numInputs
        dLdX{i} = dLdZ * W(i);
        dLdW(i) = sum(dLdZ .* X{i}, 'all');
    end

    % Pack output arguments.
    varargout(1:numInputs) = dLdX;
    varargout{numInputs+1} = dLdW;
end
end
```

Check Validity of Layer with Multiple Inputs

Check the layer validity of the custom layer `weightedAdditionLayer`.

Define a custom weighted addition layer. To create this layer, save the file `weightedAdditionLayer.m` in the current folder.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input sizes to be the typical sizes of a single observation for each input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set `'ObservationDimension'` to 4.

```
layer = weightedAdditionLayer(2, 'add');
validInputSize = {[24 24 20], [24 24 20]};
checkLayer(layer, validInputSize, 'ObservationDimension', 4)
```

Skipping GPU tests. No compatible GPU device found.

```
Running nnet.checklayer.TestCase
.....
Done nnet.checklayer.TestCase
```

```
Test Summary:
 18 Passed, 0 Failed, 0 Incomplete, 6 Skipped.
 Time elapsed: 324.3603 seconds.
```

Here, the function does not detect any issues with the layer.

Use Custom Weighted Addition Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox. This section shows how to create and train a network for digit classification using the weighted addition layer you created earlier.

Load the example training data.

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

Define a custom weighted addition layer. To create this layer, save the file `weightedAdditionLayer.m` in the current folder.

Create a layer graph including the custom layer `weightedAdditionLayer`.

```
layers = [
    imageInputLayer([28 28 1], 'Name', 'in')
    convolution2dLayer(5,20, 'Name', 'conv1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20, 'Padding', 1, 'Name', 'conv2')
    reluLayer('Name', 'relu2')
    convolution2dLayer(3,20, 'Padding', 1, 'Name', 'conv3')
    reluLayer('Name', 'relu3')
    weightedAdditionLayer(2, 'add')
    fullyConnectedLayer(10, 'Name', 'fc')
    softmaxLayer('Name', 'softmax')
    classificationLayer('Name', 'classoutput')];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph, 'relu1', 'add/in2');
```

Set the training options and train the network.

```
options = trainingOptions('adam', 'MaxEpochs', 10);
net = trainNetwork(XTrain, YTrain, lgraph, options);
```

Training on single CPU.

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	7.81%	2.3117	0.0000
2	50	00:00:32	79.69%	0.6953	0.0000
3	100	00:01:03	91.41%	0.2488	0.0000
4	150	00:01:32	96.09%	0.0991	0.0000
6	200	00:02:01	99.22%	0.0299	0.0000
7	250	00:02:29	98.44%	0.0609	0.0000
8	300	00:03:00	100.00%	0.0186	0.0000
9	350	00:03:30	100.00%	0.0155	0.0000
10	390	00:03:55	100.00%	0.0143	0.0000

View the weights learned by the weighted addition layer.

```
net.Layers(8).Weights  
ans = 1x2 single row vector  
1.0092    0.9914
```

Evaluate the network performance by predicting on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;  
YPred = classify(net,XTest);  
accuracy = sum(YTest==YPred)/numel(YTest)  
  
accuracy = 0.9824
```

See Also

[analyzeNetwork](#) | [checkLayer](#) | [trainNetwork](#)

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Check Custom Layer Validity” on page 1-165
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Weighted Classification Layer” on page 1-154
- “Define Custom Regression Output Layer” on page 1-131
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Define Custom Regression Output Layer

Tip To create a regression output layer with mean squared error loss, use `regressionLayer`. If you want to use a different loss function for your regression problems, then you can define a custom regression output layer using this example as a guide.

This example shows how to create a custom regression output layer with the mean absolute error (MAE) loss.

To define a custom regression output layer, you can use the template provided in this example, which takes you through the following steps:

- 1 Name the layer - Give the layer a name so it can be used in MATLAB.
- 2 Declare the layer properties - Specify the properties of the layer.
- 3 Create a constructor function - Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with '' at creation.
- 4 Create a forward loss function - Specify the loss between the predictions and the training targets.
- 5 Create a backward loss function - Specify the derivative of the loss with respect to the predictions.

A regression MAE layer computes the mean absolute error loss for regression problems. MAE loss is an error measure between two continuous random variables. For predictions Y and training targets T , the MAE loss between Y and T is given by

$$L = \frac{1}{N} \sum_{n=1}^N \left(\frac{1}{R} \sum_{i=1}^R |Y_{ni} - T_{ni}| \right),$$

where N is the number of observations and R is the number of responses.

Regression Output Layer Template

Copy the regression output layer template into a new file in MATLAB. This template outlines the structure of a regression output layer and includes the functions that define the layer behavior.

```
classdef myRegressionLayer < nnet.layer.RegressionLayer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

methods
    function layer = myRegressionLayer()
        % (Optional) Create a myRegressionLayer.

        % Layer constructor function goes here.
    end

    function loss = forwardLoss(layer, Y, T)
        % Return the loss between the predictions Y and the
        % training targets T.
        %
        % Inputs:
        %     layer - Output layer
        %     Y     - Predictions made by network
        %     T     - Training targets
        %
        % Output:
        %     loss  - Loss between Y and T

        % Layer forward loss function goes here.
    end

    function dLdY = backwardLoss(layer, Y, T)
        % Backward propagate the derivative of the loss function.
        %
        % Inputs:
        %     layer - Output layer
        %     Y     - Predictions made by network
        %     T     - Training targets
        %
        % Output:
        %     dLdY - Derivative of the loss with respect to the predictions Y

        % Layer backward loss function goes here.
    end
end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myRegressionLayer` with `maeRegressionLayer`.

```
classdef maeRegressionLayer < nnet.layer.RegressionLayer
    ...
end
```

Next, rename the `myRegressionLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
methods
    function layer = maeRegressionLayer()
        ...
    end
    ...
end
```

Save the Layer

Save the layer class file in a new file named `maeRegressionLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Layer Properties

Declare the layer properties in the `properties` section.

By default, custom output layers have the following properties:

- **Name** – Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
- **Description** – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays "Classification Output" or "Regression Output".
- **Type** – Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- **Classes** – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If **Classes** is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors **str**, then the software sets the classes of the output layer to **categorical(str,str)**. The default value is 'auto'.

Custom regression layers also have the following property:

- **ResponseNames** – Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

If the layer has no other properties, then you can omit the **properties** section.

The layer does not require any additional properties, so you can remove the **properties** section.

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

To initialize the **Name** property at creation, specify the input argument **name**. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = maeRegressionLayer(name)
    % layer = maeRegressionLayer(name) creates a
    % mean-absolute-error regression layer and specifies the layer
    % name.

    ...
end
```

Initialize Layer Properties

Replace the comment **% Layer constructor function goes here** with code that initializes the layer properties.

Give the layer a one-line description by setting the **Description** property of the layer. Set the **Name** property to the input argument **name**. Set the description to describe the type of layer and its size.

```
function layer = maeRegressionLayer(name)
    % layer = maeRegressionLayer(name) creates a
    % mean-absolute-error regression layer and specifies the layer
```

```
% name.

% Set layer name.
layer.Name = name;

% Set layer description.
layer.Description = 'Mean absolute error';
end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the MAE loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` contains the training targets.

For regression problems, the dimensions of `T` also depend on the type of problem.

Regression Task	Input Size	Observation Dimension
2-D image regression	1-by-1-by- R -by- N , where R is the number of responses and N is the number of observations.	4
2-D Image-to-image regression	h -by- w -by- c -by- N , where h , w , and c are the height, width, and number of channels of the output respectively, and N is the number of observations.	4
3-D image regression	1-by-1-by-1-by- R -by- N , where R is the number of responses and N is the number of observations.	5
3-D Image-to-image regression	h -by- w -by- d -by- c -by- N , where h , w , d , and c are the height, width, depth, and number of channels of the output respectively, and N is the number of observations.	5

Regression Task	Input Size	Observation Dimension
Sequence-to-one regression	R -by- N , where R is the number of responses and N is the number of observations.	2
Sequence-to-sequence regression	R -by- N -by- S , where R is the number of responses, N is the number of observations, and S is the sequence length.	2

For example, if the network defines an image regression network with one response and has mini-batches of size 50, then T is a 4-D array of size 1-by-1-by-1-by-50.

The size of Y depends on the output of the previous layer. To ensure that Y is the same size as T , you must include a layer that outputs the correct size before the output layer. For example, for image regression with R responses, to ensure that Y is a 4-D array of the correct size, you can include a fully connected layer of size R before the output layer.

A regression MAE layer computes the mean absolute error loss for regression problems. MAE loss is an error measure between two continuous random variables. For predictions Y and training targets T , the MAE loss between Y and T is given by

$$L = \frac{1}{N} \sum_{n=1}^N \left(\frac{1}{R} \sum_{i=1}^R |Y_{ni} - T_{ni}| \right)$$

where N is the number of observations and R is the number of responses.

The inputs Y and T correspond to Y and T in the equation, respectively. The output loss corresponds to L . To ensure that loss is scalar, output the mean loss over the mini-batch. Add a comment to the top of the function that explains the syntaxes of the function.

```
function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the MAE loss between
    % the predictions Y and the training targets T.

    % Calculate MAE.
    R = size(Y,3);
    meanAbsoluteError = sum(abs(Y-T),3)/R;
```

```
% Take mean over mini-batch.
N = size(Y,4);
loss = sum(meanAbsoluteError)/N;
end
```

Create Backward Loss Function

Create the backward loss function.

Create a function named `backwardLoss` that returns the derivatives of the MAE loss with respect to the predictions `Y`. The syntax for `backwardLoss` is `loss = backwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` contains the training targets.

The dimensions of `Y` and `T` are the same as the inputs in `forwardLoss`.

The derivative of the MAE loss with respect to the predictions `Y` is given by

$$\frac{\partial L}{\partial Y_i} = \frac{1}{NR} \text{sign}(Y_i - T_i),$$

where N is the number of observations and R is the number of responses. Add a comment to the top of the function that explains the syntaxes of the function.

```
function dLdY = backwardLoss(layer, Y, T)
    % Returns the derivatives of the MAE loss with respect to the predictions

    R = size(Y,3);
    N = size(Y,4);
    dLdY = sign(Y-T)/(N*R);
end
```

Completed Layer

View the completed regression output layer class file.

```
classdef maeRegressionLayer < nnet.layer.RegressionLayer
    % Example custom regression layer with mean-absolute-error loss.

    methods
        function layer = maeRegressionLayer(name)
```

```
% layer = maeRegressionLayer(name) creates a
% mean-absolute-error regression layer and specifies the layer
% name.

% Set layer name.
layer.Name = name;

% Set layer description.
layer.Description = 'Mean absolute error';
end

function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the MAE loss between
    % the predictions Y and the training targets T.

    % Calculate MAE.
    R = size(Y,3);
    meanAbsoluteError = sum(abs(Y-T),3)/R;

    % Take mean over mini-batch.
    N = size(Y,4);
    loss = sum(meanAbsoluteError)/N;
end

function dLdY = backwardLoss(layer, Y, T)
    % Returns the derivatives of the MAE loss with respect to the predictions Y.

    R = size(Y,3);
    N = size(Y,4);
    dLdY = sign(Y-T)/(N*R);
end
end
end
```

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For

more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss`, and `backwardLoss` in `maeRegressionLayer` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Output Layer Validity

Check the layer validity of the custom classification output layer `maeRegressionLayer`.

Define a custom mean absolute error regression layer. To create this layer, save the file `maeRegressionLayer.m` in the current folder. Create an instance of the layer.

```
layer = maeRegressionLayer('mae');
```

Check the layer is valid using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a 1-by-1-by-R-by-N array inputs, where R is the number of responses, and N is the number of observations in the mini-batch.

```
validInputSize = [1 1 10];
checkLayer(layer,validInputSize,'ObservationDimension',4);
```

Skipping GPU tests. No compatible GPU device found.

```
Running nnet.checklayer.OutputLayerTestCase
.....
Done nnet.checklayer.OutputLayerTestCase
```

```
Test Summary:
 13 Passed, 0 Failed, 0 Incomplete, 4 Skipped.
 Time elapsed: 0.24092 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Include Custom Regression Output Layer in Network

You can use a custom output layer in the same way as any other output layer in Deep Learning Toolbox. This section shows how to create and train a network for regression using the custom output layer you created earlier.

The example constructs a convolutional neural network architecture, trains a network, and uses the trained network to predict angles of rotated, handwritten digits. These predictions are useful for optical character recognition.

Load the example training data.

```
[trainImages,~,trainAngles] = digitTrain4DArrayData;
```

Create a layer array including the regression output layer `maeRegressionLayer`.

```
layers = [  
    imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    batchNormalizationLayer  
    reluLayer  
    fullyConnectedLayer(1)  
    maeRegressionLayer('mae')]
```

```
layers =  
6x1 Layer array with layers:
```

1	''	Image Input	28x28x1 images with 'zerocenter' normalization
2	''	Convolution	20 5x5 convolutions with stride [1 1] and padd
3	''	Batch Normalization	Batch normalization
4	''	ReLU	ReLU
5	''	Fully Connected	1 fully connected layer
6	'mae'	Regression Output	Mean absolute error

Set the training options and train the network.

```
options = trainingOptions('sgdm');  
net = trainNetwork(trainImages,trainAngles,layers,options);
```

Training on single CPU.

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch RMSE	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	28.21	25.1	0.01
2	50	00:00:08	14.46	11.4	0.01
3	100	00:00:17	13.04	10.0	0.01
4	150	00:00:26	10.21	7.9	0.01
6	200	00:00:35	10.61	8.1	0.01
7	250	00:00:44	9.83	7.3	0.01

8	300	00:00:53	8.85	7.2	0.01
9	350	00:01:02	9.21	7.3	0.01
11	400	00:01:11	10.07	8.3	0.01
12	450	00:01:20	8.63	6.4	0.01
13	500	00:01:29	9.45	6.2	0.01
15	550	00:01:38	9.10	6.8	0.01
16	600	00:01:47	8.46	6.3	0.01
17	650	00:01:56	8.38	6.3	0.01
18	700	00:02:06	8.37	6.4	0.01
20	750	00:02:15	7.06	5.2	0.01
21	800	00:02:24	7.35	5.5	0.01
22	850	00:02:33	6.43	5.0	0.01
24	900	00:02:42	6.78	4.7	0.01
25	950	00:02:52	6.54	4.4	0.01
26	1000	00:03:01	8.16	6.1	0.01
27	1050	00:03:10	6.96	5.0	0.01
29	1100	00:03:18	6.21	4.6	0.01
30	1150	00:03:27	6.38	4.9	0.01
30	1170	00:03:30	6.14	4.6	0.01

Evaluate the network performance by calculating the prediction error between the predicted and actual angles of rotation.

```
[testImages,~,testAngles] = digitTest4DArrayData;
predictedTestAngles = predict(net,testImages);
predictionError = testAngles - predictedTestAngles;
```

Calculate the number of predictions within an acceptable error margin from the true angles. Set the threshold to be 10 degrees and calculate the percentage of predictions within this threshold.

```
thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numTestImages = size(testImages,4);
accuracy = numCorrect/numTestImages

accuracy = 0.7872
```

See Also

[assembleNetwork](#) | [checkLayer](#) | [regressionLayer](#)

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Check Layer Validity” on page 1-165
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Weighted Classification Layer” on page 1-154
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Define Custom Classification Output Layer

Tip To construct a classification output layer with cross entropy loss for k mutually exclusive classes, use `classificationLayer`. If you want to use a different loss function for your classification problems, then you can define a custom classification output layer using this example as a guide.

This example shows how to define a custom classification output layer with the sum of squares error (SSE) loss and use it in a convolutional neural network.

To define a custom classification output layer, you can use the template provided in this example, which takes you through the following steps:

- 1 Name the layer - Give the layer a name so it can be used in MATLAB.
- 2 Declare the layer properties - Specify the properties of the layer.
- 3 Create a constructor function - Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with '' at creation.
- 4 Create a forward loss function - Specify the loss between the predictions and the training targets.
- 5 Create a backward loss function - Specify the derivative of the loss with respect to the predictions.

A classification SSE layer computes the sum of squares error loss for classification problems. SSE is an error measure between two continuous random variables. For predictions Y and training targets T , the SSE loss between Y and T is given by

$$L = \frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K (Y_{ni} - T_{ni})^2,$$

where N is the number of observations and K is the number of classes.

Classification Output Layer Template

Copy the classification output layer template into a new file in MATLAB. This template outlines the structure of a classification output layer and includes the functions that define the layer behavior.

```
classdef myClassificationLayer < nnet.layer.ClassificationLayer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

methods
    function layer = myClassificationLayer()
        % (Optional) Create a myClassificationLayer.

        % Layer constructor function goes here.
    end

    function loss = forwardLoss(layer, Y, T)
        % Return the loss between the predictions Y and the
        % training targets T.
        %
        % Inputs:
        %       layer - Output layer
        %       Y     - Predictions made by network
        %       T     - Training targets
        %
        % Output:
        %       loss - Loss between Y and T

        % Layer forward loss function goes here.
    end

    function dLdY = backwardLoss(layer, Y, T)
        % Backward propagate the derivative of the loss function.
        %
        % Inputs:
        %       layer - Output layer
        %       Y     - Predictions made by network
        %       T     - Training targets
        %
        % Output:
        %       dLdY - Derivative of the loss with respect to the predictions Y

        % Layer backward loss function goes here.
    end
end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myClassificationLayer` with `sseClassificationLayer`.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    ...
end
```

Next, rename the `myClassificationLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
methods
    function layer = sseClassificationLayer()
        ...
    end
    ...
end
```

Save the Layer

Save the layer class file in a new file named `sseClassificationLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Layer Properties

Declare the layer properties in the `properties` section.

By default, custom output layers have the following properties:

- **Name** – Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
- **Description** – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays "Classification Output" or "Regression Output".

- **Type** – Type of the layer, specified as a character vector or a string scalar. The value of **Type** appears when the layer is displayed in a **Layer** array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- **Classes** – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If **Classes** is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors **str**, then the software sets the classes of the output layer to **categorical(str,str)**. The default value is 'auto'.

Custom regression layers also have the following property:

- **ResponseNames** – Names of the responses, specified as a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

If the layer has no other properties, then you can omit the **properties** section.

In this example, the layer does not require any additional properties, so you can remove the **properties** section.

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

Specify the input argument **name** to assign to the **Name** property at creation. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = sseClassificationLayer(name)
    % layer = sseClassificationLayer(name) creates a sum of squares
    % error classification layer and specifies the layer name.

    ...
end
```

Initialize Layer Properties

Replace the comment **% Layer constructor function goes here** with code that initializes the layer properties.

Give the layer a one-line description by setting the `Description` property of the layer. Set the `Name` property to the input argument name.

```
function layer = sseClassificationLayer(name)
    % layer = sseClassificationLayer(name) creates a sum of squares
    % error classification layer and specifies the layer name.

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    layer.Description = 'Sum of squares error';
end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the SSE loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

For classification problems, the dimensions of `T` depend on the type of problem.

Classification Task	Input Size	Observation Dimension
2-D image classification	1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	4
3-D image classification	1-by-1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	5
Sequence-to-label classification	K -by- N , where K is the number of classes and N is the number of observations.	2
Sequence-to-sequence classification	K -by- N -by- S , where K is the number of classes, N is the number of observations, and S is the sequence length.	2

The size of Y depends on the output of the previous layer. To ensure that Y is the same size as T , you must include a layer that outputs the correct size before the output layer. For example, to ensure that Y is a 4-D array of prediction scores for K classes, you can include a fully connected layer of size K followed by a softmax layer before the output layer.

A classification SSE layer computes the sum of squares error loss for classification problems. SSE is an error measure between two continuous random variables. For predictions Y and training targets T , the SSE loss between Y and T is given by

$$L = \frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K (Y_{ni} - T_{ni})^2,$$

where N is the number of observations and K is the number of classes.

The inputs Y and T correspond to y and t in the equation, respectively. The output loss corresponds to L . Add a comment to the top of the function that explains the syntaxes of the function.

```
function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the SSE loss between
    % the predictions Y and the training targets T.

    % Calculate sum of squares.
    sumSquares = sum((Y-T).^2);

    % Take mean over mini-batch.
    N = size(Y,4);
    loss = sum(sumSquares)/N;
end
```

Create Backward Loss Function

Create the backward loss function.

Create a function named `backwardLoss` that returns the derivatives of the SSE loss with respect to the predictions Y . The syntax for `backwardLoss` is `loss = backwardLoss(layer, Y, T)`, where Y is the output of the previous layer and T represents the training targets.

The dimensions of Y and T are the same as the inputs in `forwardLoss`.

The derivative of the SSE loss with respect to the predictions Y is given by

$$\frac{\delta L}{\delta Y_i} = \frac{2}{N}(Y_i - T_i)$$

where N is the number of observations. Add a comment to the top of the function that explains the syntaxes of the function.

```
function dLdY = backwardLoss(layer, Y, T)
    % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
    % the SSE loss with respect to the predictions Y.

    N = size(Y,4);
    dLdY = 2*(Y-T)/N;
end
```

Completed Layer

View the completed classification output layer class file.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    % Example custom classification layer with sum of squares error loss.

    methods
        function layer = sseClassificationLayer(name)
            % layer = sseClassificationLayer(name) creates a sum of squares
            % error classification layer and specifies the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = 'Sum of squares error';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the SSE loss between
            % the predictions Y and the training targets T.

            % Calculate sum of squares.
            sumSquares = sum((Y-T).^2);

            % Take mean over mini-batch.
            N = size(Y,4);
```

```
    loss = sum(sumSquares)/N;
end

function dLdY = backwardLoss(layer, Y, T)
    % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
    % the SSE loss with respect to the predictions Y.

    N = size(Y,4);
    dLdY = 2*(Y-T)/N;
end
end
end
```

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss`, and `backwardLoss` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Output Layer Validity

Check the layer validity of the custom classification output layer `sseClassificationLayer`.

Define a custom sum-of-squares error classification layer. To create this layer, save the file `sseClassificationLayer.m` in the current folder. Create an instance of the layer.

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

Check the layer is valid using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a 1-by-1-by- K -by- N array inputs, where K is the number of classes, and N is the number of observations in the mini-batch.

```
validInputSize = [1 1 10];
checkLayer(layer,validInputSize,'ObservationDimension',4);

Skipping GPU tests. No compatible GPU device found.

Running nnet.checklayer.OutputLayerTestCase
.....
Done nnet.checklayer.OutputLayerTestCase

Test Summary:
13 Passed, 0 Failed, 0 Incomplete, 4 Skipped.
Time elapsed: 0.44462 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Include Custom Classification Output Layer in Network

You can use a custom output layer in the same way as any other output layer in Deep Learning Toolbox. This section shows how to create and train a network for classification using the custom classification output layer that you created earlier.

Load the example training data.

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

Define a custom sum-of-squares error classification layer. To create this layer, save the file `sseClassificationLayer.m` in the current folder. Create an instance of the layer. Create a layer array including the custom classification output layer `sseClassificationLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    sseClassificationLayer('sse')]

layers =
7x1 Layer array with layers:
```

```

1   ''      Image Input          28x28x1 images with 'zerocenter' normalization
2   ''      Convolution        20 5x5 convolutions with stride [1 1] and pad
3   ''      Batch Normalization  Batch normalization
4   ''      ReLU                ReLU
5   ''      Fully Connected    10 fully connected layer
6   ''      Softmax             softmax
7 'sse'  Classification Output Sum of squares error

```

Set the training options and train the network.

```

options = trainingOptions('sgdm');
net = trainNetwork(XTrain,YTrain,layers,options);

```

Training on single CPU.

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	14.84%	0.9414	0.0100
2	50	00:00:09	77.34%	0.3519	0.0100
3	100	00:00:17	90.63%	0.1375	0.0100
4	150	00:00:25	97.66%	0.0495	0.0100
6	200	00:00:33	96.09%	0.0660	0.0100
7	250	00:00:42	96.88%	0.0489	0.0100
8	300	00:00:50	99.22%	0.0172	0.0100
9	350	00:00:59	98.44%	0.0284	0.0100
11	400	00:01:07	100.00%	0.0064	0.0100
12	450	00:01:16	100.00%	0.0050	0.0100
13	500	00:01:24	100.00%	0.0064	0.0100
15	550	00:01:33	100.00%	0.0063	0.0100
16	600	00:01:41	100.00%	0.0018	0.0100
17	650	00:01:50	100.00%	0.0031	0.0100
18	700	00:01:59	100.00%	0.0021	0.0100
20	750	00:02:07	100.00%	0.0023	0.0100
21	800	00:02:16	100.00%	0.0017	0.0100
22	850	00:02:25	100.00%	0.0014	0.0100
24	900	00:02:34	100.00%	0.0016	0.0100
25	950	00:02:43	100.00%	0.0010	0.0100
26	1000	00:02:52	100.00%	0.0011	0.0100
27	1050	00:03:03	100.00%	0.0051	0.0100
29	1100	00:03:14	100.00%	0.0014	0.0100
30	1150	00:03:24	100.00%	0.0012	0.0100
30	1170	00:03:28	100.00%	0.0014	0.0100

Evaluate the network performance by making predictions on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net, XTest);
accuracy = mean(YTest == YPred)

accuracy = 0.9878
```

See Also

[assembleNetwork](#) | [checkLayer](#) | [classificationLayer](#)

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Check Custom Layer Validity” on page 1-165
- “Define Custom Weighted Classification Layer” on page 1-154
- “Define Custom Regression Output Layer” on page 1-131
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Define Custom Weighted Classification Layer

Tip To construct a classification output layer with cross entropy loss for k mutually exclusive classes, use `classificationLayer`. If you want to use a different loss function for your classification problems, then you can define a custom classification output layer using this example as a guide.

This example shows how to define and create a custom weighted classification output layer with weighted cross entropy loss. Use a weighted classification layer for classification problems with an imbalanced distribution of classes. For an example showing how to use a weighted classification layer in a network, see “Speech Command Recognition Using Deep Learning”.

To define a custom classification output layer, you can use the template provided in this example, which takes you through the following steps:

- 1 Name the layer - Give the layer a name so it can be used in MATLAB.
- 2 Declare the layer properties - Specify the properties of the layer.
- 3 Create a constructor function - Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with '' at creation.
- 4 Create a forward loss function - Specify the loss between the predictions and the training targets.
- 5 Create a backward loss function - Specify the derivative of the loss with respect to the predictions.

A weighted classification layer computes the weighted cross entropy loss for classification problems. Weighted cross entropy is an error measure between two continuous random variables. For prediction scores Y and training targets T , the weighted cross entropy loss between Y and T is given by

$$L = -\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K w_i T_{ni} \log(Y_{ni}),$$

where N is the number of observations, K is the number of classes, and w is a vector of weights for each class.

Classification Output Layer Template

Copy the classification output layer template into a new file in MATLAB. This template outlines the structure of a classification output layer and includes the functions that define the layer behavior.

```
classdef myClassificationLayer < nnet.layer.ClassificationLayer

properties
    % (Optional) Layer properties.

    % Layer properties go here.
end

methods
    function layer = myClassificationLayer()
        % (Optional) Create a myClassificationLayer.

        % Layer constructor function goes here.
    end

    function loss = forwardLoss(layer, Y, T)
        % Return the loss between the predictions Y and the
        % training targets T.
        %
        % Inputs:
        %     layer - Output layer
        %     Y     - Predictions made by network
        %     T     - Training targets
        %
        % Output:
        %     loss - Loss between Y and T

        % Layer forward loss function goes here.
    end

    function dLdY = backwardLoss(layer, Y, T)
        % Backward propagate the derivative of the loss function.
        %
        % Inputs:
        %     layer - Output layer
        %     Y     - Predictions made by network
        %     T     - Training targets
        %
        % Output:
        %     dLdY - Derivative of the loss with respect to the predictions Y

        % Layer backward loss function goes here.
    end
end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myClassificationLayer` with `weightedClassificationLayer`.

```
classdef weightedClassificationLayer < nnet.layer.ClassificationLayer
    ...
end
```

Next, rename the `myClassificationLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
methods
    function layer = weightedClassificationLayer()
        ...
    end
    ...
end
```

Save the Layer

Save the layer class file in a new file named `weightedClassificationLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Layer Properties

Declare the layer properties in the `properties` section.

By default, custom output layers have the following properties:

- **Name** – Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
- **Description** – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays "Classification Output" or "Regression Output".

- **Type** – Type of the layer, specified as a character vector or a string scalar. The value of **Type** appears when the layer is displayed in a **Layer** array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- **Classes** – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If **Classes** is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors **str**, then the software sets the classes of the output layer to **categorical(str,str)**. The default value is 'auto'.

Custom regression layers also have the following property:

- **ResponseNames** – Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

If the layer has no other properties, then you can omit the **properties** section.

In this example, the layer requires an additional property to save the class weights. Specify the property **ClassWeights** in the **properties** section.

```
properties
    % Vector of weights corresponding to the classes in the training
    % data
    ClassWeights
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

Specify input argument **classWeights** to assign to the **ClassWeights** property. Also specify an optional input argument **name** to assign to the **Name** property at creation. Add a comment to the top of the function that explains the syntaxes of the function.

```
function layer = weightedClassificationLayer(classWeights, name)
    % layer = weightedClassificationLayer(classWeights) creates a
    % weighted cross entropy loss layer. classWeights is a row
    % vector of weights corresponding to the classes in the order
    % that they appear in the training data.
    %
    % layer = weightedClassificationLayer(classWeights, name)
```

```
% additionally specifies the layer name.  
end ...
```

Initialize Layer Properties

Replace the comment % Layer constructor function goes here with code that initializes the layer properties.

Give the layer a one-line description by setting the `Description` property of the layer. Set the `Name` property to the optional input argument `name`.

```
function layer = weightedClassificationLayer(classWeights, name)  
    % layer = weightedClassificationLayer(classWeights) creates a  
    % weighted cross entropy loss layer. classWeights is a row  
    % vector of weights corresponding to the classes in the order  
    % that they appear in the training data.  
    %  
    % layer = weightedClassificationLayer(classWeights, name)  
    % additionally specifies the layer name.  
  
    % Set class weights  
    layer.ClassWeights = classWeights;  
  
    % Set layer name  
    if nargin == 2  
        layer.Name = name;  
    end  
  
    % Set layer description  
    layer.Description = 'Weighted cross entropy';  
end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the weighted cross entropy loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

For classification problems, the dimensions of `T` depend on the type of problem.

Classification Task	Input Size	Observation Dimension
2-D image classification	1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	4
3-D image classification	1-by-1-by-1-by- K -by- N , where K is the number of classes and N is the number of observations.	5
Sequence-to-label classification	K -by- N , where K is the number of classes and N is the number of observations.	2
Sequence-to-sequence classification	K -by- N -by- S , where K is the number of classes, N is the number of observations, and S is the sequence length.	2

The size of Y depends on the output of the previous layer. To ensure that Y is the same size as T , you must include a layer that outputs the correct size before the output layer. For example, to ensure that Y is a 4-D array of prediction scores for K classes, you can include a fully connected layer of size K followed by a softmax layer before the output layer.

A weighted classification layer computes the weighted cross entropy loss for classification problems. Weighted cross entropy is an error measure between two continuous random variables. For prediction scores Y and training targets T , the weighted cross entropy loss between Y and T is given by

$$L = -\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^K w_i T_{ni} \log(Y_{ni}),$$

where N is the number of observations, K is the number of classes, and w is a vector of weights for each class.

The inputs Y and T correspond to Y and T in the equation, respectively. The output loss corresponds to L . Add a comment to the top of the function that explains the syntaxes of the function.

```
function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the weighted cross
    % entropy loss between the predictions Y and the training
    % targets T.

    N = size(Y,4);
    Y = squeeze(Y);
    T = squeeze(T);
    W = layer.ClassWeights;

    loss = -sum(W*(T.*log(Y)))/N;
end
```

Create Backward Loss Function

Create the backward loss function.

Create a function named `backwardLoss` that returns the derivatives of the weighted cross entropy loss with respect to the predictions `Y`. The syntax for `backwardLoss` is `loss = backwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

The dimensions of `Y` and `T` are the same as the inputs in `forwardLoss`.

The derivative of the weighted cross entropy loss with respect to the predictions `Y` is given by

$$\frac{\delta L}{\delta Y_i} = -\frac{1}{N} \frac{w_i T_i}{Y_i},$$

where `N` is the number of observations and `w` is a vector of weights for each class. Add a comment to the top of the function that explains the syntaxes of the function.

```
function dLdY = backwardLoss(layer, Y, T)
    % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
    % the weighted cross entropy loss with respect to the
    % predictions Y.

    [~,~,K,N] = size(Y);
    Y = squeeze(Y);
    T = squeeze(T);
    W = layer.ClassWeights;
```

```

dLdY = -(W' .* T ./ Y) / N;
dLdY = reshape(dLdY,[1 1 K N]);
end

```

Completed Layer

View the completed classification output layer class file.

```

classdef weightedClassificationLayer < nnet.layer.ClassificationLayer

properties
    % Vector of weights corresponding to the classes in the training
    % data
    ClassWeights
end

methods
    function layer = weightedClassificationLayer(classWeights, name)
        % layer = weightedClassificationLayer(classWeights) creates a
        % weighted cross entropy loss layer. classWeights is a row
        % vector of weights corresponding to the classes in the order
        % that they appear in the training data.
        %
        % layer = weightedClassificationLayer(classWeights, name)
        % additionally specifies the layer name.

        % Set class weights
        layer.ClassWeights = classWeights;

        % Set layer name
        if nargin == 2
            layer.Name = name;
        end

        % Set layer description
        layer.Description = 'Weighted cross entropy';
    end

    function loss = forwardLoss(layer, Y, T)
        % loss = forwardLoss(layer, Y, T) returns the weighted cross
        % entropy loss between the predictions Y and the training
        % targets T.

        N = size(Y,4);
    end
end

```

```
Y = squeeze(Y);
T = squeeze(T);
W = layer.ClassWeights;

    loss = -sum(W.*log(Y))/N;
end

function dLdY = backwardLoss(layer, Y, T)
% dLdY = backwardLoss(layer, Y, T) returns the derivatives of
% the weighted cross entropy loss with respect to the
% predictions Y.

[~,~,K,N] = size(Y);
Y = squeeze(Y);
T = squeeze(T);
W = layer.ClassWeights;

dLdY = -(W' .*T./Y)/N;
dLdY = reshape(dLdY,[1 1 K N]);
end
end
end
```

GPU Compatibility

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss` and `backwardLoss` in `weightedClassificationLayer` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Output Layer Validity

Check the validity of the custom classification output layer `weightedClassificationLayer`.

Define a custom weighted classification layer. To create this layer, save the file `weightedClassificationLayer.m` in the current folder.

Create an instance of the layer. Specify the class weights as a vector with three elements corresponding to three classes.

```
classWeights = [0.1 0.7 0.2];
layer = weightedClassificationLayer(classWeights);
```

Check that the layer is valid using `checkLayer`. Set the valid input size to the typical size of a single observation input to the layer. The layer expects a 1-by-1-by- K -by- N array input, where K is the number of classes and N is the number of observations in the mini-batch.

```
numClasses = numel(classWeights);
validInputSize = [1 1 numClasses];
checkLayer(layer,validInputSize,'ObservationDimension',4);
```

Skipping GPU tests. No compatible GPU device found.

```
Running nnet.checklayer.OutputLayerTestCase
.
.
.
Done nnet.checklayer.OutputLayerTestCase
```

```
Test Summary:
 13 Passed, 0 Failed, 0 Incomplete, 4 Skipped.
 Time elapsed: 0.49523 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

See Also

`assembleNetwork` | `checkLayer` | `classificationLayer`

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Check Layer Validity” on page 1-165
- “Define Custom Regression Output Layer” on page 1-131
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Check Custom Layer Validity

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, and correctly defined gradients. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer, and `dim` specifies the dimension of the observations in the layer input data. For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Check Layer Validity

Check the validity of the example custom layer `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create an instance of the layer and check that it is valid using `checkLayer`. Set the valid input size to the typical size of a single observation input to the layer. For a single input, the layer expects observations of size h -by- w -by- c , where h , w , and c are the height, width, and number of channels of the previous layer output, respectively.

Specify `validInputSize` as the typical size of an input array.

```
layer = preluLayer(20,'prelu');
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

Skipping multi-observation tests. To enable tests with multiple observations, specify `'ObservationDimension'`.
For 2-D image data, set `'ObservationDimension'` to 4.
For 3-D image data, set `'ObservationDimension'` to 5.
For sequence data, set `'ObservationDimension'` to 2.

Skipping GPU tests. No compatible GPU device found.

```
Running nnet.checklayer.TestCase
.....
Done nnet.checklayer.TestCase
```

```
Test Summary:  
13 Passed, 0 Failed, 0 Incomplete, 11 Skipped.  
Time elapsed: 3.9634 seconds.
```

The results show the number of passed, failed, and skipped tests. If you do not specify the '`ObservationsDimension`' option, or do not have a GPU, then the function skips the corresponding tests.

Check Multiple Observations

For multi-observation input, the layer expects an array of observations of size h -by- w -by- c -by- N , where h , w , and c are the height, width, and number of channels, respectively, and N is the number of observations.

To check the layer validity for multiple observations, specify the typical size of an observation and set '`ObservationDimension`' to 4.

```
layer = preluLayer(20,'prelu');  
validInputSize = [5 5 20];  
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

Skipping GPU tests. No compatible GPU device found.

Running `nnet.checklayer.TestCase`

.....
Done `nnet.checklayer.TestCase`

```
Test Summary:  
18 Passed, 0 Failed, 0 Incomplete, 6 Skipped.  
Time elapsed: 5.4552 seconds.
```

In this case, the function does not detect any issues with the layer.

List of Tests

The `checkLayer` function checks the validity of a custom layer by performing a series of tests.

Intermediate Layers

The `checkLayer` function uses these tests to check the validity of custom intermediate layers (layers of type `nnet.layer.Layer`).

Test	Description
<code>methodSignaturesAreCorrect</code>	The syntaxes of the layer functions are correctly defined.
<code>predictDoesNotError</code>	<code>predict</code> does not error.
<code>forwardDoesNotError</code>	<code>forward</code> does not error.
<code>forwardPredictAreConsistentInSize</code>	<code>forward</code> and <code>predict</code> output values of the same size.
<code>backwardDoesNotError</code>	<code>backward</code> does not error.
<code>backwardIsConsistentInSize</code>	The outputs of <code>backward</code> are consistent in size: <ul style="list-style-type: none"> The derivatives with respect to each input are the same size as the corresponding input. The derivatives with respect to each learnable parameter are the same size as the corresponding learnable parameter.
<code>predictIsConsistentInType</code>	The outputs of <code>predict</code> are consistent in type with the inputs.
<code>forwardIsConsistentInType</code>	The outputs of <code>forward</code> are consistent in type with the inputs.
<code>backwardIsConsistentInType</code>	The outputs of <code>backward</code> are consistent in type with the inputs.
<code>gradientsAreNumericallyCorrect</code>	The gradients computed in <code>backward</code> are consistent with the numerical gradients.

The tests `predictIsConsistentInType`, `forwardIsConsistentInType`, and `backwardIsConsistentInType` also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

If you have not implemented `forward`, then `checkLayer` does not run the `forwardDoesNotError`, `forwardPredictAreConsistentInSize`, and `forwardIsConsistentInType` tests.

Output Layers

The `checkLayer` function uses these tests to check the validity of custom output layers (layers of type `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`).

Test	Description
<code>forwardLossDoesNotError</code>	<code>forwardLoss</code> does not error.
<code>backwardLossDoesNotError</code>	<code>backwardLoss</code> does not error.
<code>forwardLossIsScalar</code>	The output of <code>forwardLoss</code> is scalar.
<code>backwardLossIsConsistentInSize</code>	The output of <code>backwardLoss</code> is consistent in size: <code>dLdY</code> is the same size as the predictions <code>Y</code> .
<code>forwardLossIsConsistentInType</code>	The output of <code>forwardLoss</code> is consistent in type: <code>loss</code> is the same type as the predictions <code>Y</code> .
<code>backwardLossIsConsistentInType</code>	The output of <code>backwardLoss</code> is consistent in type: <code>dLdY</code> must be the same type as the predictions <code>Y</code> .
<code>gradientsAreNumericallyCorrect</code>	The gradients computed in <code>backwardLoss</code> are numerically correct.

The `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` tests also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

Generated Data

To check the layer validity, the `checkLayer` function generates data depending on the type of layer:

Layer Type	Description of Generated Data
Intermediate	Values in the range [-1,1]
Regression output	Predictions and targets with values in the range [-1,1]
Classification output	<p>Predictions with values in the range [0,1].</p> <p>If you specify the '<code>'ObservationDimension'</code>' option, then the targets are one-hot encoded vectors (vectors containing a single 1, and 0 elsewhere).</p> <p>If you do not specify the '<code>'ObservationDimension'</code>' option, then the targets are values in the range [0,1].</p>

To check for multiple observations, specify the observation dimension using the '`'ObservationDimension'`' name-value pair. If you specify the observation dimension, then the `checkLayer` function checks that the layer functions are valid using generated data with mini-batches of size 1 and 2. If you do not specify this name-value pair, then the function skips the tests that check that the layer functions are valid for multiple observations.

Diagnostics

If a test fails when you use `checkLayer`, then the function provides a test diagnostic and a framework diagnostic. The test diagnostic highlights any issues found with the layer. The framework diagnostic provides more detailed information.

Function Syntaxes

The test method `methodSignaturesAreCorrect` checks that the layer functions have correctly defined syntaxes.

Test Diagnostic	Description	Possible Solution
Incorrect number of input arguments for ' <code>'predict'</code> ' in Layer.	The syntax for <code>predict</code> is not consistent with the number of layer inputs.	Specify the correct number of input and output arguments in <code>predict</code> .

Test Diagnostic	Description	Possible Solution
Incorrect number of output arguments for 'predict' in Layer	The syntax for predict is not consistent with the number of layer outputs.	<p>The syntax for predict is <code>[Z₁, ..., Z_m] = predict(layer, X₁, ..., X_n)</code> where X₁, ..., X_n are the n layer inputs and Z₁, ..., Z_m are the m layer outputs. The values n and m must correspond to the NumInputs and NumOutputs properties of the layer.</p> <p>Tip If the number of inputs to predict can vary, then use varargin instead of X₁, ..., X_n. In this case, varargin is a cell array of the inputs, where varargin{i} corresponds to X_i. If the number of outputs can vary, then use varargout instead of Z₁, ..., Z_m. In this case, varargout is a cell array of the outputs, where varargout{j} corresponds to Z_j.</p>
Incorrect number of input arguments for 'forward' in Layer	The syntax for forward is not consistent with the number of layer inputs.	<p>Specify the correct number of input and output arguments in forward.</p> <p>The syntax for forward is <code>[Z₁, ..., Z_m, memory] = forward(layer, X₁, ..., X_n)</code> where X₁, ..., X_n are the n layer inputs, Z₁, ..., Z_m are the m layer outputs, and</p>

Test Diagnostic	Description	Possible Solution
Incorrect number of output arguments for 'forward' in Layer	The syntax for <code>forward</code> is not consistent with the number of layer outputs.	<p><code>memory</code> is the memory of the layer.</p> <p>Tip If the number of inputs to <code>forward</code> can vary, then use <code>varargin</code> instead of <code>X1, ..., Xn</code>. In this case, <code>varargin</code> is a cell array of the inputs, where <code>varargin{i}</code> corresponds to <code>Xi</code>. If the number of outputs can vary, then use <code>varargout</code> instead of <code>Z1, ..., Zm</code>. In this case, <code>varargout</code> is a cell array of the outputs, where <code>varargout{j}</code> corresponds to <code>Zj</code> for $j=1, \dots, \text{NumOutputs}$ and <code>varargout{NumOutputs +1}</code> corresponds to <code>memory</code>.</p>
Incorrect number of input arguments for 'backward' in Layer	The syntax for <code>backward</code> is not consistent with the number of layer inputs and outputs.	<p>Specify the correct number of input and output arguments in <code>backward</code>.</p> <p>The syntax for <code>backward</code> is <code>[dLdX1, ..., dLdXn, dLdW1, ..., dLdWk] = backward(layer, X1, ..., Xn, Z1, ..., Zm, dLdZ1, ..., dLdZm)</code>, where <code>X1, ..., Xn</code> are the n layer inputs, <code>Z1, ..., Zm</code> are the m outputs of <code>forward</code>, <code>dLdZ1, ..., dLdZm</code> are the gradients backward propagated from the next layer, and <code>memory</code> is the memory output of <code>forward</code>. For the outputs, <code>dLdX1, ..., dLdXn</code> are the derivatives</p>

Test Diagnostic	Description	Possible Solution
Incorrect number of output arguments for 'backward' in Layer	The syntax for <code>backward</code> is not consistent with the number of layer outputs.	<p>of the loss with respect to the layer inputs and $dLdW_1, \dots, dLdW_k$ are the derivatives of the loss with respect to the k learnable parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with <code>~</code>.</p> <p>Tip If the number of inputs to <code>backward</code> can vary, then use <code>varargin</code> instead of the input arguments after <code>layer</code>. In this case, <code>varargin</code> is a cell array of the inputs, where <code>varargin{i}</code> corresponds to X_i for $i=1, \dots, \text{NumInputs}$, <code>varargin{NumInputs+j}</code> and <code>varargin{NumInputs+NumOutputs+j}</code> correspond to Z_j and $dLdZ_j$, respectively, for $j=1, \dots, \text{NumOutputs}$, and <code>varargin{end}</code> corresponds to <code>memory</code>.</p> <p>If the number of outputs can vary, then use <code>varargout</code> instead of the output arguments. In this case, <code>varargout</code> is a cell array of the outputs, where <code>varargout{i}</code> corresponds</p>

Test Diagnostic	Description	Possible Solution
		to $dLdXi$ for $i=1, \dots, \text{NumInputs}$ and $\text{varargout}\{\text{NumInputs}+t\}$ corresponds to $dLdWt$ for $t=1, \dots, k$, where k is the number of learnable parameters.

For layers with multiple inputs or outputs, you must set the values of the layer properties `NumInputs` (or alternatively, `InputNames`) and `NumOutputs` (or alternatively, `OutputNames`) in the layer constructor function, respectively.

Multiple Observations

The `checkLayer` function checks that the layer functions are valid for single and multiple observations. To check for multiple observations, specify the observation dimension using the `'ObservationDimension'` name-value pair. If you specify the observation dimension, then the `checkLayer` function checks that the layer functions are valid using generated data with mini-batches of size 1 and 2. If you do not specify this name-value pair, then the function skips the tests that check that the layer functions are valid for multiple observations.

Test Diagnostic	Description	Possible Solution
Skipping multi-observation tests. To enable checks with multiple observations, specify the <code>'ObservationDimension'</code> parameter in <code>checkLayer</code> .	If you do not specify the <code>'ObservationDimension'</code> parameter in <code>checkLayer</code> , then the function skips the tests that check data with multiple observations.	Use the command <code>checkLayer(layer, validInputSize, 'ObservationDimension', dim)</code> , where <code>layer</code> is an instance of the custom layer, <code>validInputSize</code> is a vector specifying the valid input size to the layer, and <code>dim</code> specifies the dimension of the observations in the layer input. For more information, see "Layer Input Sizes".

Functions Do Not Error

These tests check that the layers do not error when passed input data of valid size.

Intermediate Layers

The tests `predictDoesNotError`, `forwardDoesNotError`, and `backwardDoesNotError` check that the layer functions do not error when passed inputs of valid size. If you specify an observation dimension, then the function checks the layer for both a single observation and multiple observations.

Test Diagnostic	Description	Possible Solution
The function ' <code>predict</code> ' threw an error:	The <code>predict</code> function errors when passed data of size <code>validInputSize</code> .	Address the error described in the Framework Diagnostic section.
The function ' <code>forward</code> ' threw an error:	The <code>forward</code> function errors when passed data of size <code>validInputSize</code> .	
The function ' <code>backward</code> ' threw an error:	The <code>backward</code> function errors when passed the output of <code>predict</code> .	

Output Layers

The tests `forwardLossDoesNotError` and `backwardLossDoesNotError` check that the layer functions do not error when passed inputs of valid size. If you specify an observation dimension, then the function checks the layer for both a single observation and multiple observations.

Test Diagnostic	Description	Possible Solution
The function ' <code>forwardLoss</code> ' threw an error:	The <code>forwardLoss</code> function errors when passed data of size <code>validInputSize</code> .	Address the error described in the Framework Diagnostic section.
The function ' <code>backwardLoss</code> ' threw an error:	The <code>backwardLoss</code> function errors when passed data of size <code>validInputSize</code> .	

Outputs Are Consistent in Size

These tests check that the layer function outputs are consistent in size.

Intermediate Layers

The test `backwardIsConsistentInSize` checks that the `backward` function outputs derivatives of the correct size.

The syntax for `backward` is

```
[dLdX1,...,dLdXn,dLdW1,...,dLdWk] = backward(layer,X1,...,Xn,Z1,...,Zm,dLdZ1,...,dLdZm,memory)
```

where X_1, \dots, X_n are the n layer inputs, Z_1, \dots, Z_m are the m outputs of `forward`, $dLdZ_1, \dots, dLdZ_m$ are the gradients backward propagated from the next layer, and `memory` is the memory output of `forward`. For the outputs, $dLdX_1, \dots, dLdX_n$ are the derivatives of the loss with respect to the layer inputs and $dLdW_1, \dots, dLdW_k$ are the derivatives of the loss with respect to the k learnable parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to X_i for $i=1, \dots, \text{NumInputs}$, `varargin{NumInputs+j}` and `varargin{NumInputs+NumOutputs+j}` correspond to Z_j and $dLdZ_j$, respectively, for $j=1, \dots, \text{NumOutputs}$, and `varargin{end}` corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where `varargout{i}` corresponds to $dLdX_i$ for $i=1, \dots, \text{NumInputs}$ and `varargout{NumInputs+t}` corresponds to $dLdW_t$ for $t=1, \dots, k$, where k is the number of learnable parameters.

The derivatives $dLdX_1, \dots, dLdX_n$ must be the same size as the corresponding layer inputs, and $dLdW_1, \dots, dLdW_k$ must be the same size as the corresponding learnable parameters. The sizes must be consistent for input data with single and multiple observations.

Test Diagnostic	Description	Possible Solution
Incorrect size of ' <code>dLdX</code> ' for ' <code>backward</code> '.	The derivatives of the loss with respect to the layer	Return the derivatives <code>dLdX_1, \dots, dLdX_n</code> with the

Test Diagnostic	Description	Possible Solution
Incorrect size of the derivative of the loss with respect to the input 'in1' for 'backward'	inputs must be the same size as the corresponding layer input.	same size as the corresponding layer inputs X_1, \dots, X_n .
The size of 'Z' returned from 'forward' must be the same as for 'predict'.	The outputs of predict must be the same size as the corresponding outputs of forward.	Return the outputs Z_1, \dots, Z_m of predict with the same size as the corresponding outputs Z_1, \dots, Z_m of forward.
Incorrect size of the derivative of the loss with respect to 'W' for 'backward'.	The derivatives of the loss with respect to the learnable parameters must be the same size as the corresponding learnable parameters.	Return the derivatives $dLdW_1, \dots, dLdW_k$ with the same size as the corresponding learnable parameters W_1, \dots, W_k .

Output Layers

The tests `forwardLossIsScalar` and `backwardLossIsConsistentInSize` check that the outputs of `forwardLoss` and `backwardLoss` are of the correct size.

The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`. The input `Y` corresponds to the predictions made by the network. These predictions are the output of the previous layer. The input `T` corresponds to the training targets. The output `loss` is the loss between `Y` and `T` according to the specified loss function. The output `loss` must be scalar.

The syntax for `backwardLoss` is `dLdY = backwardLoss(layer, Y, T)`. The inputs `Y` are the predictions made by the network and `T` are the training targets. The output `dLdY` is the derivative of the loss with respect to the predictions `Y`. The output `dLdY` must be the same size as the layer input `Y`.

Test Diagnostic	Description	Possible Solution
Incorrect size of 'loss' for 'forwardLoss'.	The output loss of forwardLoss must be a scalar.	Return the output loss as a scalar. For example, if you have multiple values of the loss, then you can use mean or sum.
Incorrect size of the derivative of loss 'dLdY' for 'backwardLoss'.	The derivatives of the loss with respect to the layer input must be the same size as the layer input Y.	Return derivative dLdY with the same size as the layer input Y.

Consistent Data Types and GPU Compatibility

These tests check that the layer function outputs are consistent in type and that the layer functions are GPU compatible.

For GPU compatibility, the layer functions must support inputs and return outputs of type `gpuArray`. Any other functions the layer uses must do the same. Many MATLAB built-in functions support `gpuArray` input arguments. If you call any of these functions with at least one `gpuArray` input, then the function executes on the GPU and returns a `gpuArray` output. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Intermediate Layers

The tests `predictIsConsistentInType`, `forwardIsConsistentInType`, and `backwardIsConsistentInType` check that the layer functions output variables of the correct data type. The tests check that the layer functions return consistent data types when given inputs of the data types `single`, `double`, and `gpuArray` with the underlying types `single` or `double`.

Tip If you preallocate arrays using functions like `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type of another array, use the '`like`' option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz, 'like', X)`.

Test Diagnostic	Description	Possible Solution
Incorrect type of 'Z' for 'predict'.	The types of the outputs Z_1, \dots, Z_m of <code>predict</code> must be consistent with the inputs X_1, \dots, X_n .	Return the outputs Z_1, \dots, Z_m with the same type as the inputs X_1, \dots, X_n .
Incorrect type of output 'out1' for 'predict'.		
Incorrect type of 'Z' for 'forward'.	The types of the outputs Z_1, \dots, Z_m of <code>forward</code> must be consistent with the inputs X_1, \dots, X_n .	
Incorrect type of output 'out1' for 'forward'.		
Incorrect type of 'dLdX' for 'backward'.	The types of the derivatives $dLdX_1, \dots, dLdX_n$ of <code>backward</code> must be consistent with the inputs X_1, \dots, X_n .	Return the derivatives $dLdX_1, \dots, dLdX_n$ with the same type as the inputs X_1, \dots, X_n .
Incorrect type of the derivative of the loss with respect to the input 'in1' for 'backward'.		
Incorrect type of the derivative of loss with respect to 'W' for 'backward'.	The type of the derivative of the loss of the learnable parameters must be consistent with the corresponding learnable parameters.	For each learnable parameter, return the derivative with the same type as the corresponding learnable parameter.

Output Layers

The tests `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` check that the layer functions output variables of the correct data type. The tests check that the layers return consistent data types when given inputs of the data types `single`, `double`, and `gpuArray` with the underlying types `single` or `double`.

Test Diagnostic	Description	Possible Solution
Incorrect type of 'loss' for 'forwardLoss'.	The type of the output <code>loss</code> of <code>forwardLoss</code> must be consistent with the input <code>Y</code> .	Return <code>loss</code> with the same type as the input <code>Y</code> .

Test Diagnostic	Description	Possible Solution
Incorrect type of the derivative of loss 'dLdY' for 'backwardLoss'.	The type of the output dLdY of backwardLoss must be consistent with the input Y.	Return dLdY with the same type as the input Y.

Gradients Are Numerically Correct

The test gradientsAreNumericallyCorrect checks that the gradients computed by the layer functions are numerically correct.

Intermediate Layers

The test gradientsAreNumericallyCorrect tests that the gradients computed in backward are numerically correct.

Test Diagnostic	Description	Possible Solution
The derivative 'dLdX' for 'backward' is inconsistent with the numerical gradient.	One or more of the following: <ul style="list-style-type: none"> The specified derivative is incorrectly computed Function is non-differentiable at some input points Error tolerance is too small 	Check that the derivatives in backward are correctly computed. If the derivatives are correctly computed, then in the Framework Diagnostic section, manually check the absolute and relative error between the actual and expected values of the derivative.
The derivative of the loss with respect to the input 'inl' for 'backward' is inconsistent with the numerical gradient.		
The derivative of loss with respect to 'W' for 'backward' is inconsistent with the numerical gradient.		If the absolute and relative errors are within an acceptable margin of the tolerance, then you can ignore this test diagnostic.

Output Layers

The test gradientsAreNumericallyCorrect checks that the gradients computed in backwardLoss are numerically correct.

Test Diagnostic	Description	Possible Solution
The derivative 'dLdY' for 'backwardLoss' is inconsistent with the numerical gradient.	<p>One or more of the following:</p> <ul style="list-style-type: none">• The derivative with respect to the predictions Y is incorrectly computed• Function is non-differentiable at some input points• Error tolerance is too small	<p>Check that the derivatives in <code>backwardLoss</code> are correctly computed.</p> <p>If the derivatives are correctly computed, then in the <code>Framework Diagnostic</code> section, manually check the absolute and relative error between the actual and expected values of the derivative.</p> <p>If the absolute and relative errors are within an acceptable margin of the tolerance, then you can ignore this test diagnostic.</p>

See Also

`analyzeNetwork` | `checkLayer`

More About

- “Define Custom Deep Learning Layers” on page 1-77
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 1-97
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 1-114
- “Define Custom Classification Output Layer” on page 1-143
- “Define Custom Weighted Classification Layer” on page 1-154
- “Define Custom Regression Output Layer” on page 1-131
- “List of Deep Learning Layers” on page 1-28
- “Deep Learning Tips and Tricks” on page 1-57

Long Short-Term Memory Networks

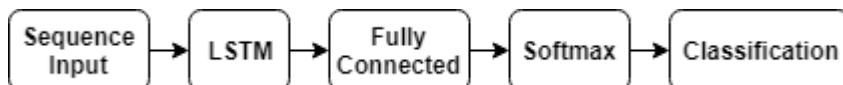
This topic explains how to work with sequence and time series data for classification and regression tasks using long short-term memory (LSTM) networks. For an example showing how to classify sequence data using an LSTM network, see “Sequence Classification Using Deep Learning”.

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

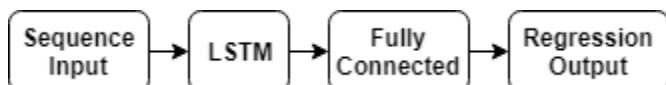
LSTM Network Architecture

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

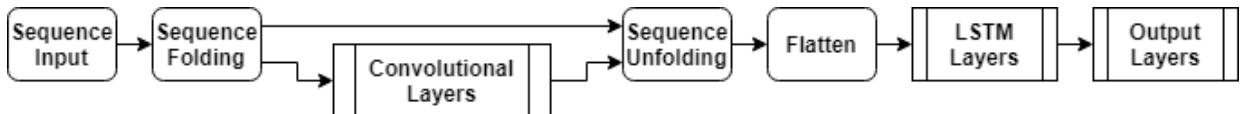
This diagram illustrates the architecture of a simple LSTM network for classification. The network starts with a sequence input layer followed by an LSTM layer. To predict class labels, the network ends with a fully connected layer, a softmax layer, and a classification output layer.



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



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



Classification LSTM Networks

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

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

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

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

For an example showing how to train an LSTM network for sequence-to-label classification and classify new data, see "Sequence Classification Using Deep Learning".

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

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

Regression LSTM Networks

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

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

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

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

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

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

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

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

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

Video Classification Network

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

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

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

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

    sequenceFoldingLayer('Name', 'fold')

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

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

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

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

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

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

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

Deeper LSTM Networks

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

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

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

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

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

Layers

Function	Description
 <code>sequenceInputLayer</code>	A sequence input layer inputs sequence data to a network.
 <code>lstmLayer</code>	An LSTM layer learns long-term dependencies between time steps in time series and sequence data.
 <code>bilstmLayer</code>	A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.
 <code>sequenceFoldingLayer</code>	A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.
 <code>sequenceUnfoldingLayer</code>	A sequence unfolding layer restores the sequence structure of the input data after sequence folding.
 <code>flattenLayer</code>	A flatten layer collapses the spatial dimensions of the input into the channel dimension.
 <code>wordEmbeddingLayer</code> (Text Analytics Toolbox)	A word embedding layer maps word indices to vectors.

Classification, Prediction, and Forecasting

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

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

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

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

Sequence Padding, Truncation, and Splitting

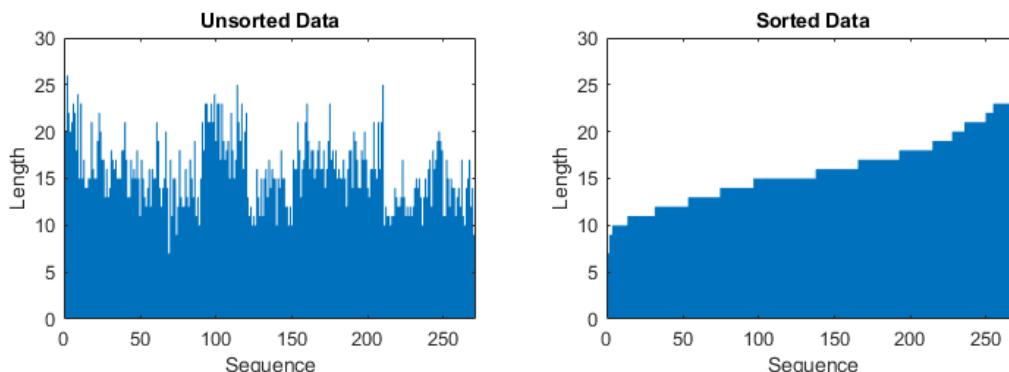
LSTM networks support input data with varying sequence lengths. When passing data through the network, the software pads, truncates, or splits sequences so that all the sequences in each mini-batch have the specified length. You can specify the sequence lengths and the value used to pad the sequences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

Sort Sequences by Length

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

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

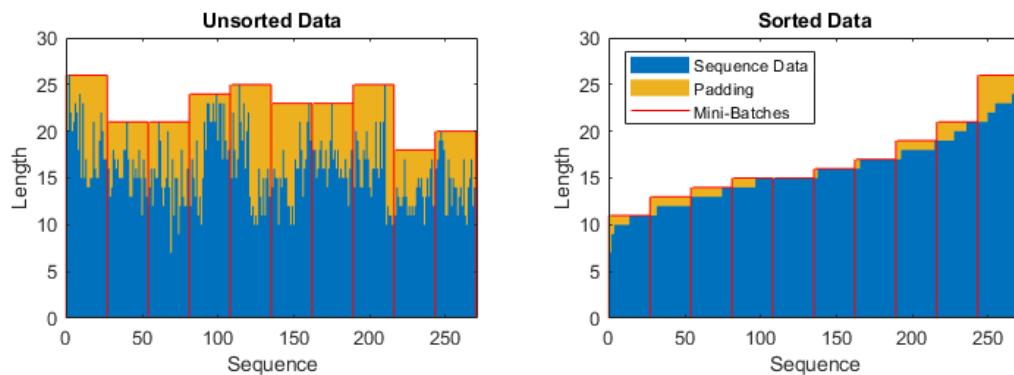
The following figures show the sequence lengths of the sorted and unsorted data in bar charts.



Pad Sequences

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

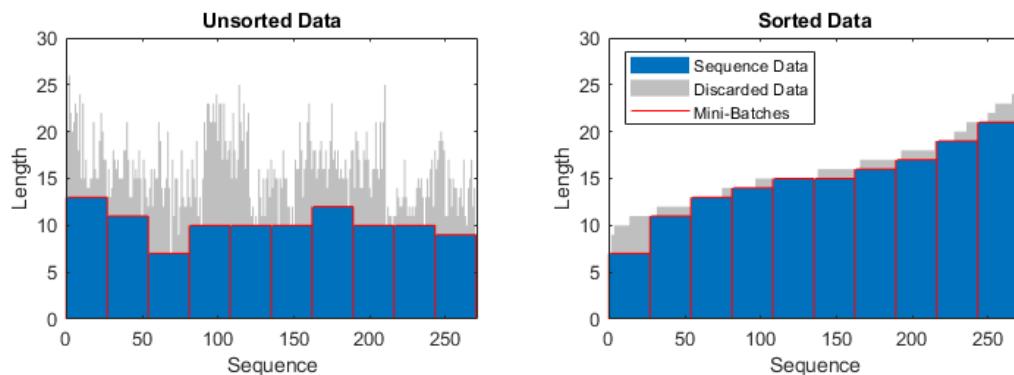
The following figures illustrate the effect of setting 'SequenceLength' to 'longest'.



Truncate Sequences

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

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



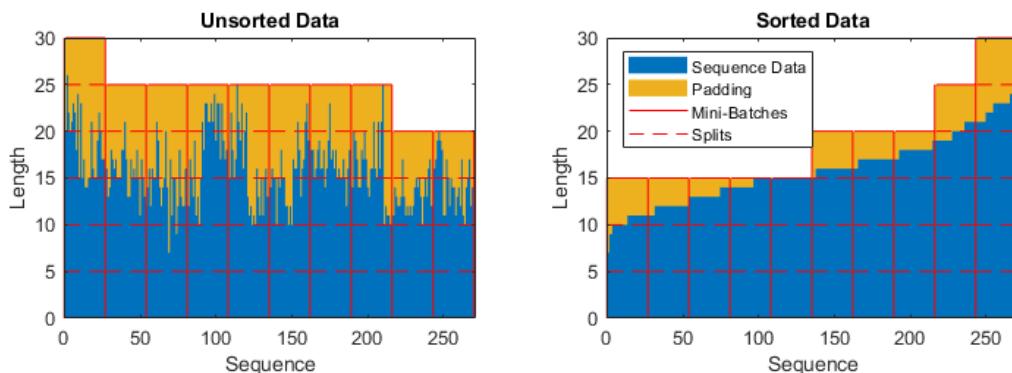
Split Sequences

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

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

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

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



Normalize Sequence Data

To recenter training data automatically at training time using zero-center normalization, set the `Normalization` property of `sequenceInputLayer` to '`'zerocenter'`'.

Alternatively, you can normalize sequence data by first calculating the per-feature mean and standard deviation of all the sequences. Then, for each training observation, subtract the mean value and divide by the standard deviation.

```

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

```

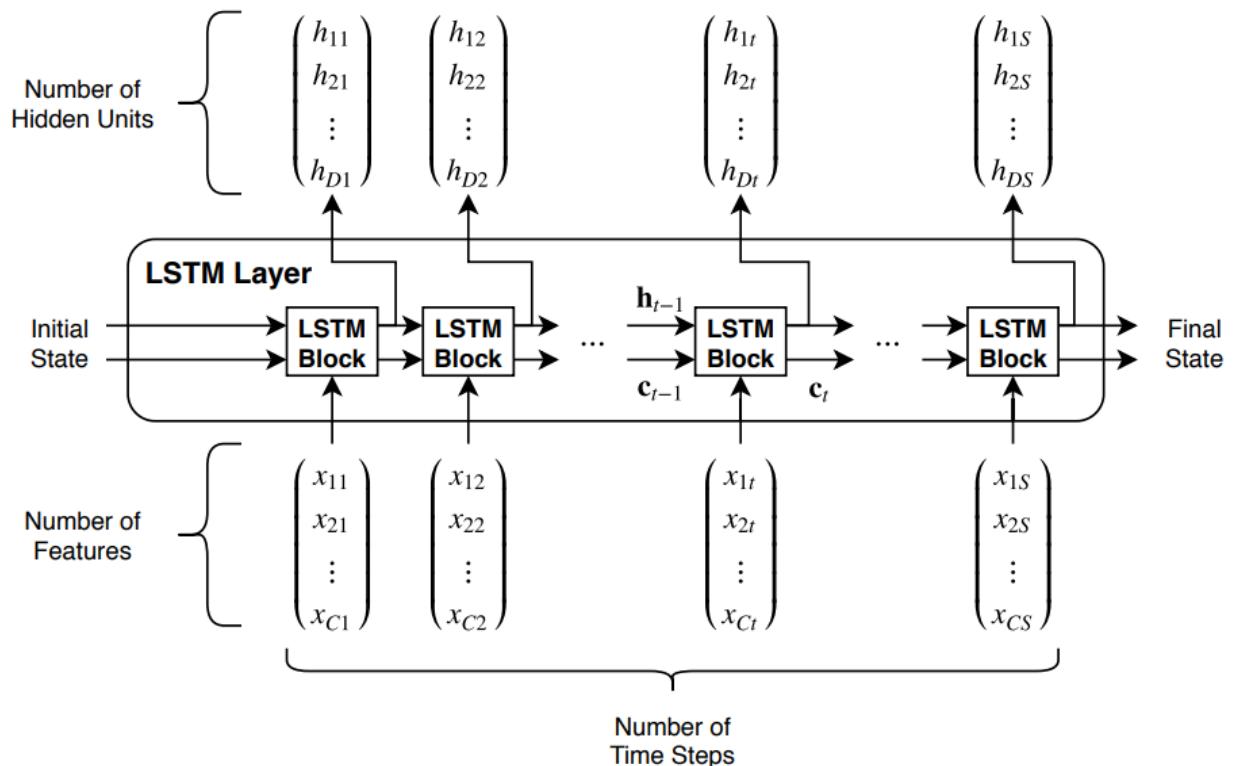
Out-of-Memory Data

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

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

LSTM Layer Architecture

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



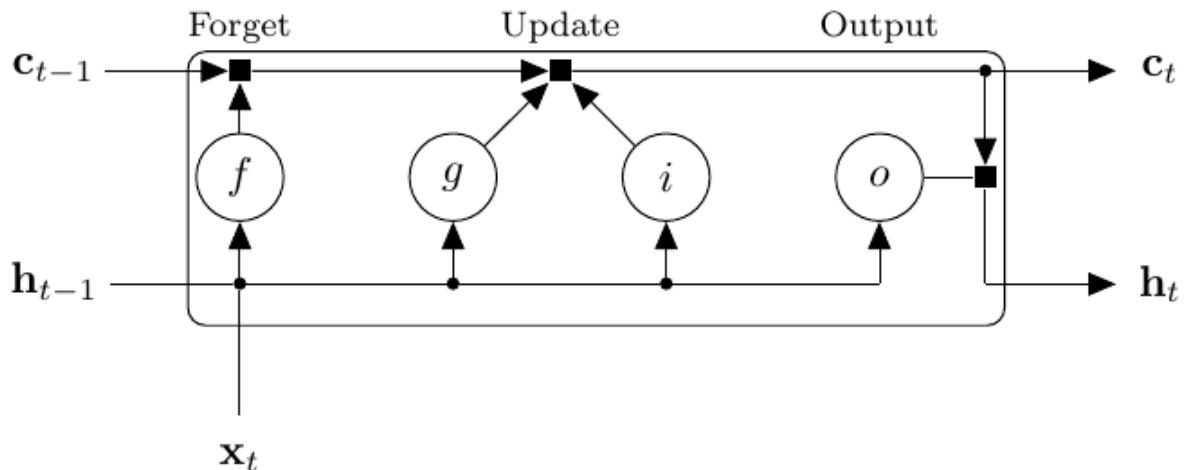
The first LSTM block uses the initial state of the network and the first time step of the sequence to compute the first output and the updated cell state. At time step t , the block uses the current state of the network ($\mathbf{c}_{t-1}, \mathbf{h}_{t-1}$) and the next time step of the sequence to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

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

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

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

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



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

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

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

The cell state at time step t is given by

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

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

The hidden state at time step t is given by

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

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

The following formulas describe the components at time step t .

Component	Formula
Input gate	$i_t = \sigma_g(W_i \mathbf{x}_t + R_i \mathbf{h}_{t-1} + b_i)$
Forget gate	$f_t = \sigma_g(W_f \mathbf{x}_t + R_f \mathbf{h}_{t-1} + b_f)$
Cell candidate	$g_t = \sigma_c(W_g \mathbf{x}_t + R_g \mathbf{h}_{t-1} + b_g)$
Output gate	$o_t = \sigma_g(W_o \mathbf{x}_t + R_o \mathbf{h}_{t-1} + b_o)$

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

References

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

See Also

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

Related Examples

- “Sequence Classification Using Deep Learning”
- “Time Series Forecasting Using Deep Learning”
- “Sequence-to-Sequence Classification Using Deep Learning”
- “Sequence-to-Sequence Regression Using Deep Learning”
- “Classify Videos Using Deep Learning”
- “Develop Custom Mini-Batch Datastore” on page 1-213
- “Deep Learning in MATLAB” on page 1-2

Datastores for Deep Learning

Datastores in MATLAB are a convenient way of working with and representing collections of data that are too large to fit in memory at one time. Because deep learning often requires large amounts of data, datastores are an important part of the deep learning workflow in MATLAB.

Select Datastore

For many applications, the easiest approach is to start with a built-in datastore. For more information about the available built-in datastores, see “Select Datastore for File Format or Application” (MATLAB). However, only some types of built-in datastores can be used directly as input for network training, validation, and inference. These datastores are:

- `ImageDatastore`
- `AugmentedImageDatastore`
- `RandomPatchExtractionDatastore` (requires Image Processing Toolbox™)
- `PixelLabelImageDatastore` (requires Computer Vision Toolbox)
- `DenoisingImageDatastore` (requires Image Processing Toolbox)

Other built-in datastores can be used as input for deep learning, but the data read from these datastores must be preprocessed into a format required by a deep learning network. For more information on the required format of read data, see “Input Datastore for Training, Validation, and Inference” on page 1-194. For more information on how to preprocess data read from datastores, see “Transform and Combine Datastores” on page 1-196.

For some applications, there may not be a built-in datastore type that fits your data well. For these problems, you can create a custom datastore. For more information, see “Develop Custom Datastore” (MATLAB). All custom datastores are valid inputs to deep learning interfaces as long as the `read` function of the custom datastore returns data in the required two-column form.

Input Datastore for Training, Validation, and Inference

Datastores are valid inputs in Deep Learning Toolbox for training, validation, and inference.

Training and Validation

To use an image datastore as a source of training data, use the `imds` argument of `trainNetwork`. To use all other types of datastore as a source of training data, use the `ds` argument of `trainNetwork`. To use a datastore for validation, use the '`ValidationData`' name-value pair argument in `trainingOptions`.

To be a valid input for training or validation, the `read` function of a datastore (with the exception of `ImageDatastore`) must return data as either a two-column cell array or a two-column table. The first column of data represents inputs to the network and the second column of data represents responses. Each row of data represents a separate observation. For `ImageDatastore` only, `trainNetwork` and `trainingOptions` support data returned as integer arrays and single-column cell array of integer arrays.

The table shows sample output of calling the `read` function for datastore `ds`.

Format of Read Data	Sample Output
Two-column cell array	<pre>data = read(ds) data = 4×2 cell array {28×28 double} {[7]} {28×28 double} {[7]} {28×28 double} {[9]} {28×28 double} {[9]}</pre>
Two-column table	<pre>data = read(ds) data = 4×2 table input response _____ _____ {28×28 double} 7 {28×28 double} 7 {28×28 double} 9 {28×28 double} 9</pre>

Inference

For inference using `predict`, `classify`, and `activations`, a datastore is only required to yield one column. The inference functions ignore additional columns of data beyond the first.

Specify Read Size and Mini-Batch Size

A datastore may return any number of rows (observations) for each call to `read`. Functions such as `trainNetwork`, `predict`, `classify`, and `activations` that accept datastores and support specifying a '`MiniBatchSize`' call `read` as many times as is necessary to form complete mini-batches of data. As these functions form mini-batches, they use internal queues in memory to store read data. For example, if a datastore consistently returns 64 rows per call to `read` and `MiniBatchSize` is 128, then to form each mini-batch of data requires two calls to `read`.

For best runtime performance, it is recommended to configure datastores such that the number of observations returned by `read` is equal to the '`MiniBatchSize`'. For datastores that have a '`ReadSize`' property, set the '`ReadSize`' to change the number of observations returned by the datastore for each call to `read`.

Transform and Combine Datastores

Deep learning frequently requires the data to be preprocessed and augmented before data is in an appropriate form to input to a network. The `transform` and `combine` functions of datastore are useful in preparing data to be fed into a network.

Transform Datastores

The `transform` function creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore.

- For complex transformations involving several preprocessing operations, define the complete set of transformations in your own function. Then, specify a handle to your function as the `@fcn` argument of `transform`. For more information, see "Create Functions in Files" (MATLAB).
- For simple transformations that can be expressed in one line of code, you can specify a handle to an anonymous function as the `@fcn` argument of `transform`. For more information, see "Anonymous Functions" (MATLAB).

The function handle provided to `transform` must accept input data in the same format as returned by the `read` function of the underlying datastore.

Example: Transform Image Datastore to Train Digit Classification Network

This example uses the `transform` function to create a training set in which randomized 90 degree rotation is added to each image within an image datastore. Pass the resulting `TransformedDatastore` to `trainNetwork` to train a simple digit classification network.

Create an image datastore containing digit images.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Set the mini-batch size equal to the `ReadSize` of the image datastore.

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;
```

Transform images in the image datastore by adding randomized 90 degree rotation. The transformation function, `preprocessForTraining`, is defined at the end of this example.

```
dsTrain = transform(imds,@preprocessForTraining,'IncludeInfo',true)
dsTrain =
```

TransformedDatastore with properties:

```
UnderlyingDatastore: [1x1 matlab.io.datastore.ImageDatastore]
    Transforms: {@preprocessForTraining}
    IncludeInfo: 1
```

Specify layers of the network and training options, then train the network using the transformed datastore `dsTrain` as a source of data.

```
layers = [
    ...
    imageInputLayer([28 28 1],'Normalization','none')
    convolution2dLayer(5,20)
    reluLayer()
```

```
maxPooling2dLayer(2,'Stride',2)
fullyConnectedLayer(10);
softmaxLayer()
classificationLayer());
```



```
options = trainingOptions('adam', ...
    'Plots','training-progress', ...
    'MiniBatchSize',miniBatchSize);
```



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

Define a function that performs the desired transformations of data, `data`, read from the underlying datastore. The function loops through each read image and performs randomized rotation, then returns the transformed image and corresponding label as a two-column cell array as expected by `trainNetwork`.

```
function [dataOut,info] = preprocessForTraining(data,info)
```



```
numRows = size(data,1);
dataOut = cell(numRows,2);
```



```
for idx = 1:numRows
```



```
% Randomized 90 degree rotation
imgOut = rot90(data{idx,1},randi(4)-1);
```



```
% Return the label from info struct as the
% second column in dataOut.
dataOut(idx,:) = {imgOut,info.Label(idx)};
```



```
end
end
```

Combine Datastores

The `combine` function associates two datastores of the same length to create the two-column format expected of training and validation data. Combining datastores maintains the parity between the datastores. Each call to the `read` function of the resulting `CombinedDatastore` returns data from corresponding parts of the underlying datastores.

For example, if you are training an image in, image out regression network, then you can create the training data set by combining two image datastores. This sample code demonstrates combining two image datastores named `imdsX` and `imdsY`. Image

datastores return data as a cell array, therefore the combined datastore `imdsTrain` returns data as a two-column cell array.

```
imdsX = imageDatastore(__);
imdsY = imageDatastore(__);
imdsTrain = combine(imdsX,imdsY)

imdsTrain =
    CombinedDatastore with properties:
        UnderlyingDatastores: {1×2 cell}
```

Use Datastore for Parallel Training and Prefetch Read Optimization

Datastores used for parallel training or multi-GPU training must be partitionable. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

Many built-in datastores are already partitionable because they support the `partition` function. Transformed datastores are partitionable if their underlying datastore is partitionable. Using the `transform` function with built-in datastores frequently maintains support for parallel and multi-GPU training.

If you need to create a custom datastore that supports parallel or multi-GPU training, then your datastore must implement the `matlab.io.datastore.Partitionable` class.

Some partitionable datastores support reading training data using asynchronous prefetch reading, which queues data in memory while the GPU is working. Specify prefetch reading using the `'DispatchInBackground'` name-value pair argument of `trainingOptions`. Prefetch reading requires Parallel Computing Toolbox.

There are some limitations when using datastores with parallel training, multi-GPU training, and prefetch read optimization.

- Datastores do not support specifying the `'Shuffle'` name-value pair argument of `trainingOptions` as `'none'`.
- Combined datastores are not partitionable and therefore do not support parallel training, multi-GPU training, or prefetch reading.

See Also

`combine | read | trainNetwork | trainingOptions | transform`

Related Examples

- “Prepare Datastore for Image-to-Image Regression”
- “Classify Text Data Using Convolutional Neural Network”

More About

- “Getting Started with Datastore” (MATLAB)
- “Select Datastore for File Format or Application” (MATLAB)
- “Develop Custom Datastore” (MATLAB)

Preprocess Images for Deep Learning

Training a network and making predictions on new data require images that match the input size of the network. Depending on the format of your data, you can use `imresize` or `augmentedImage datastore` to resize images to the required size.

You can apply affine geometric transformations to images to augment training, validation, test, and prediction data sets. Augmenting training images helps to prevent the network from overfitting and memorizing the exact details of the training images.

For more advanced preprocessing, you can start with a built-in datastore that performs specific image preprocessing operations suitable for common applications. You can also preprocess images according to your own pipeline by using the `transform` and `combine` functions. For more information, see “Datastores for Deep Learning” on page 1-194.

Resize Images

You can store image data as a numeric array, `ImageDatastore`, or table. An `ImageDatastore` enables you to import data from image collections that are too large to fit in memory. This function is designed to read batches of images for faster processing in machine learning and computer vision applications. You can use an augmented image datastore or a resized 4-D array for training, prediction, and classification. You can use a resized 3-D array for prediction and classification only.

The method to resize images depends on the image data type.

Data Type	Resizing Function	Sample Code
3-D array representing a single color image, a single multispectral image, or a stack of grayscale images	<code>imresize</code>	To resize images in the 3-D array <code>im3d</code> : <code>im = imresize(im3d, inputSize);</code>
4-D array representing a stack of images	<code>imresize</code>	To resize images in the 4-D array <code>im4d</code> : <code>im = imresize(im4d, inputSize);</code>
	<code>augmentedImage datastore</code>	To rescale images in the 4-D array <code>im4d</code> : <code>auimds = augmentedImage datastore(inputSize, im4d)</code>

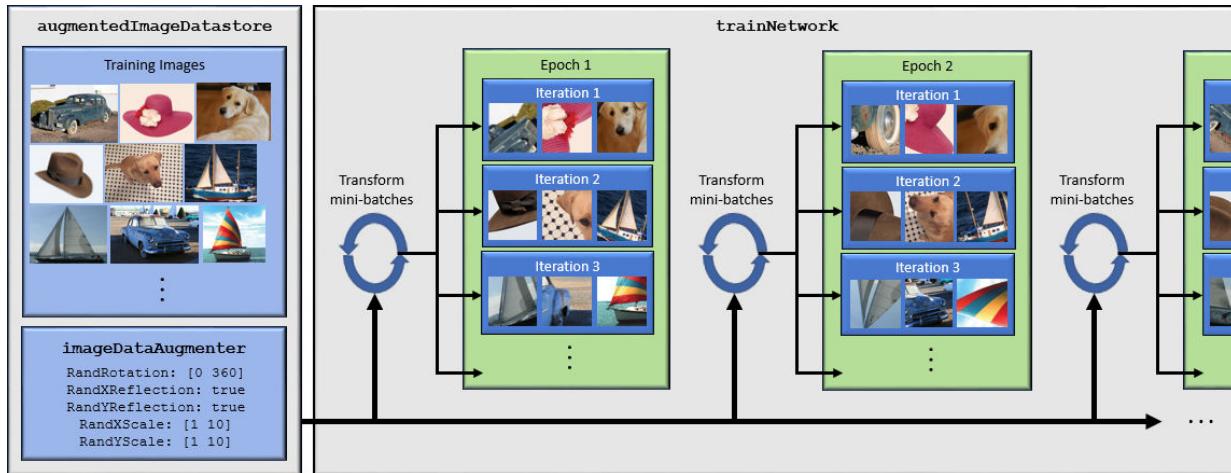
Data Type	Resizing Function	Sample Code
ImageDatastore	augmentedImageDatastore	To rescale images in the image datastore <code>imds</code> : <code>auimds = augmentedImageDatastore(inputSize,imds)</code> For a more complete example, see “Train Deep Learning Network to Classify New Images”.
table	augmentedImageDatastore	To rescale images in the table <code>tbl</code> : <code>auimds = augmentedImageDatastore(inputSize,tbl);</code>

By default, `augmentedImageDatastore` rescales images to the desired size. If instead you want to crop images from the center or from random positions in the image, you can use the `'OutputSizeMode'` name-value pair argument. For example, this code shows how to crop images in image datastore `imds` from the center of each image:

```
auimds = augmentedImageDatastore(inputSize,imds,'OutputSizeMode','centercrop');
```

Augment Images for Training

In addition to resizing images, an `augmentedImageDatastore` enables you to preprocess images with a combination of rotation, reflection, shear, and translation transformations. The diagram shows how `trainNetwork` uses an augmented image datastore to transform training data for each epoch. For an example of the workflow, see “Train Network with Augmented Images”.



- 1 Specify your training images.
- 2 Configure image transformation options, such as the range of rotation angles and whether to apply reflection at random, by creating an `imageDataAugmenter`.

Tip To preview the transformations applied to sample images, use the `augment` function.

- 3 Create an `augmentedImageDatastore`. Specify the training images, the size of output images, and the `imageDataAugmenter`. The size of output images must be compatible with the size of the `imageInputLayer` of the network.
- 4 Train the network, specifying the augmented image datastore as the data source for `trainNetwork`. For each iteration of training, the augmented image datastore applies a random combination of transformations to images in the mini-batch of training data.

When you use an augmented image datastore as a source of training images, the datastore randomly perturbs the training data for each epoch, so that each epoch uses a slightly different data set. The actual number of training images at each epoch does not change. The transformed images are not stored in memory.

Datastores for Advanced Image Preprocessing

Some datastores perform specific image preprocessing operations when they read a batch of data. These application-specific datastores are listed in the table. You can use these

datastores as a source of training, validation, and test data sets for deep learning applications that use Deep Learning Toolbox. All of these datastores return data in a format supported by `trainNetwork`.

Datastore	Description
<code>augmentedImageDatastore</code>	Apply random affine geometric transformations, including resizing, rotation, reflection, shear, and translation, for training deep neural networks. For an example, see “Transfer Learning Using AlexNet”.
<code>pixelLabelImageDatastore</code>	Apply identical affine geometric transformations to images and corresponding ground truth labels for training semantic segmentation networks (requires Computer Vision Toolbox). For an example, see “Semantic Segmentation Using Deep Learning”.
<code>randomPatchExtractionDatastore</code>	Extract multiple pairs of random patches from images or pixel label images (requires Image Processing Toolbox). You optionally can apply identical random affine geometric transformations to the pairs of patches. For an example, see “Single Image Super-Resolution Using Deep Learning”.
<code>denoisingImageDatastore</code>	Apply randomly generated Gaussian noise for training denoising networks (requires Image Processing Toolbox).

To perform more general and complex image preprocessing operations than offered by the application-specific datastores, you can use the `transform` and `combine` functions. The `transform` function creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore according to a transformation function that you define. The `combine` function concatenates the data read from multiple datastores to the two-column table or two-column cell array format required by `trainNetwork`. The `combine` function maintains parity between the underlying datastores.

Function	Resulting Datastore	Description
<code>transform</code>	<code>TransformedDatastore</code>	Transform batches of read data from an underlying datastore according to your own preprocessing pipeline.
<code>combine</code>	<code>CombinedDatastore</code>	Horizontally concatenate the data read from two or more underlying datastores.

The custom transformation function must accept data in the format returned by the `read` function of the underlying datastore. For image data, the format depends on the `ReadSize` property of the underlying `ImageDatastore`.

- When `ReadSize` is 1, the transformation function must accept an integer array. The size of the array is consistent with the type of images in the `ImageDatastore`. For example, a grayscale image has dimensions m -by- n , a truecolor image has dimensions m -by- n -by-3, and a multispectral image with c channels has dimensions m -by- n -by- c .
- When `ReadSize` is greater than 1, the transformation function must accept a cell array of image data corresponding to each image in the batch.

The `transform` function must return data that matches the input size of the network. The `transform` function does not support one-to-many observation mappings.

Tip The `transform` function supports prefetching when the underlying `ImageDatastore` reads a batch of JPG or PNG image files. For these image types, do not use the `readFcn` argument of `ImageDatastore` to apply image preprocessing, as this option is usually significantly slower. If you use a custom read function, then `ImageDatastore` does not prefetch.

See Also

`ImageDatastore` | `combine` | `imresize` | `trainNetwork` | `transform`

Related Examples

- “Train Network with Augmented Images”
- “Train Deep Learning Network to Classify New Images”
- “Prepare Datastore for Image-to-Image Regression”

More About

- “Datastores for Deep Learning” on page 1-194
- “Deep Learning in MATLAB” on page 1-2

Preprocess Volumes for Deep Learning

Read Volumetric Image Data

Supported file formats for volumetric image data include MAT-files, Digital Imaging and Communications in Medicine (DICOM) files, and Neuroimaging Informatics Technology Initiative (NIfTI) files.

You can read volumetric image data into an `ImageDatastore`. When you create the image datastore, specify the `'FileExtensions'` argument as the file extensions of your data. Specify the `ReadFcn` property as a function handle that reads data of the file format. For more information, see “Datastores for Deep Learning” on page 1-194.

The table shows how to create an image datastore for each of the supported file formats. The `filepath` argument specifies the path to the files or folder containing image data.

Image File Format	Image Datastore Creation
MAT	<pre>volds = imageDatastore(filepath, ... 'FileExtensions', '.mat', 'ReadFcn', @(x) matRead(x));</pre> <p>You must create the <code>matRead</code> function to read data from a MAT-file. Save the function in a file called <code>matRead.m</code>.</p> <pre>function data = matRead(filename) % data = matRead(filename) reads the image data in the MAT-file filename</pre> <pre>inp = load(filename); f = fields(inp); data = inp.(f{1}); end</pre>
DICOM	<pre>volds = imageDatastore(filepath, ... 'FileExtensions', '.dcm', 'ReadFcn', @(x) dicomread(x));</pre> <p>Using DICOM files for 3-D deep learning requires the volume data be stored in a single file. You cannot volume data that is divided into multiple image files.</p> <p>For more information about the DICOM file format, see <code>dicomread</code>.</p>

Image File Format	Image Datastore Creation
NIfTI	<pre>volds = imageDatastore(filepath, ... 'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));</pre> <p>For more information about the NIfTI file format, see niftiread.</p>

Read Volumetric Label Data for Semantic Segmentation

To perform semantic segmentation of volumetric data, you must have labels corresponding to the image data.

You can read volumetric label data into a `PixelLabelDatastore`. When you create the pixel label datastore, specify the `'FileExtensions'` argument as the file extensions of your data. Specify the `ReadFcn` property as a function handle that reads data of the file format.

The table shows how to create a pixel label datastore for each of the supported file formats. The `filepath` argument specifies the path to the files or folder containing label data. The `classNames` and `pixelLabelID` arguments specify the mapping of voxel label values to class names.

Image File Format	Pixel Label Datastore Creation
MAT	<pre>pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ... 'FileExtensions','.mat','ReadFcn',@(x) matRead(x));</pre> <p>You must create the <code>matRead</code> function to read data from a MAT-file. Save the function in a file called <code>matRead.m</code>.</p> <pre>function data = matRead(filename) % data = matRead(filename) reads the label data in the MAT-file filename inp = load(filename); f = fields(inp); data = inp.(f{1}); end</pre>

Image File Format	Pixel Label Datastore Creation
DICOM	<pre>pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ... 'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));</pre> <p>Using DICOM files for 3-D deep learning requires the volume data be stored in a single file. You cannot volume data that is divided into multiple image files.</p> <p>For more information about the DICOM file format, see dicomread.</p>
NIFTI	<pre>pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ... 'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));</pre> <p>For more information about the NIFTI file format, see niftiread.</p>

Associate Image and Label Data

To associate volumetric image and label data for semantic segmentation, or two volumetric image datastores for regression, use a `randomPatchExtractionDatastore`. A random patch extraction datastore extracts corresponding randomly-positioned patches from two datastores. Patching is a common technique to prevent running out of memory when training with arbitrarily large volumes. Specify a patch size that matches the input size of the network and, for memory efficiency, is smaller than the full size of the volume, such as 64-by-64-by-64 voxels.

You can also use the `combine` function to associate two datastores. However, associating two datastores using a `randomPatchExtractionDatastore` has several benefits over `combine`.

- `randomPatchExtractionDatastore` supports parallel training, multi-GPU training, and prefetch reading. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Specify prefetch reading using the `'DispatchInBackground'` name-value pair argument of `trainingOptions`. Prefetch reading requires Parallel Computing Toolbox.
- `randomPatchExtractionDatastore` inherently supports patch extraction. In contrast, to extract patches from a `CombinedDatastore`, you must define your own function that crops images into patches, and then use the `transform` function to apply the cropping operations.

- `randomPatchExtractionDatastore` can generate several image patches from one test image. One-to-many patch extraction effectively increases the amount of available training data.

Preprocess Volumetric Data

Deep learning frequently requires the data to be preprocessed and augmented. For example, you may want to normalize image intensities, enhance image contrast, or add randomized affine transformations to prevent overfitting.

To preprocess volumetric data, use the `transform` function. `transform` creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore according to the set of operations you define in a custom function. Image Processing Toolbox provides several functions that accept volumetric input. For a full list of functions, see [3-D Volumetric Image Processing \(Image Processing Toolbox\)](#). You can also preprocess volumetric images using functions in MATLAB that work on multidimensional arrays.

The custom transformation function must accept data in the format returned by the `read` function of the underlying datastore.

Underlying Datastore	Format of Input to Custom Transformation Function
Image datastore	<p>The input to the custom transformation function depends on the <code>ReadSize</code> property.</p> <ul style="list-style-type: none"> • When <code>ReadSize</code> is 1, the transformation function must accept an integer array. The size of the array is consistent with the type of images in the <code>ImageDatastore</code>. For example, a grayscale image has size m-by-n, a truecolor image has size m-by-n-by-3, and a multispectral image with c channels has size m-by-n-by-c. • When <code>ReadSize</code> is greater than 1, the transformation function must accept a cell array of image data corresponding to each image in the batch. <p>For more information, see the <code>read</code> function of <code>ImageDatastore</code>.</p>

Underlying Datastore	Format of Input to Custom Transformation Function
PixelLabelDatastore	<p>The input to the custom transformation function depends on the <code>ReadSize</code> property.</p> <ul style="list-style-type: none">• When <code>ReadSize</code> is 1, the transformation function must accept a categorical matrix.• When <code>ReadSize</code> is greater than 1, the transformation function must accept a cell array of categorical matrices. <p>For more information, see the <code>read</code> function of <code>PixelLabelDatastore</code>.</p>
randomPatchExtractionDatastore	<p>The input to the custom transformation function must be a table with two columns.</p> <p>For more information, see the <code>read</code> function of <code>randomPatchExtractionDatastore</code>.</p>

`RandomPatchExtractionDatastore` does not support the `DataAugmentation` property for volumetric data. To apply random affine transformations to volumetric data, you must use `transform`.

The `transform` function must return data that matches the input size of the network. The `transform` function does not support one-to-many observation mappings.

Example: Transform Volumetric Data in Image Datastore

This sample code shows how to transform volumetric data in image datastore `volds` using an arbitrary preprocessing pipeline defined in the function `preprocessVolumetricIMDS`. The example assumes that the `ReadSize` of `volds` is greater than 1.

```
dsTrain = transform(volds,@(x) preprocessVolumetricIMDS(x,inputSize));
```

Define the `preprocessVolumetricIMDS` function that performs the desired transformations of data read from the underlying datastore. The function must accept a cell array of image data. The function loops through each read image and transforms the data according to this preprocessing pipeline:

- Randomly rotate the image about the *z*-axis.

- Resize the volume to the size expected by the network.
- Create a noisy version of the image with Gaussian noise.
- Return the image in a cell array.

```
function dataOut = preprocessVolumetricIMDS(data,inputSize)

numRows = size(data,1);
dataOut = cell(numRows,1);

for idx = 1:numRows

    % Perform randomized 90 degree rotation about the z-axis
    data = imrotate3(data{idx,1},90*(randi(4)-1),[0 0 1]);

    % Resize the volume to the size expected by the network
    dataClean = imresize(data,inputSize);

    % Add zero-mean Gaussian noise with a normalized variance of 0.01
    dataNoisy = imnoise(dataClean,'gaussian',0.01);

    % Return the preprocessed data
    dataOut(idx) = dataNoisy;

end
end
```

Example: Transform Volumetric Data in Random Patch Extraction Datastore

This sample code shows how to transform volumetric data in random patch extraction datastore `volds` using an arbitrary preprocessing pipeline defined in the function `preprocessVolumetricPatchDS`. The example assumes that the `ReadSize` of `volds` is 1.

```
dsTrain = transform(volds,@preprocessVolumetricPatchDS);
```

Define the `preprocessVolumetricPatchDS` function that performs the desired transformations of data read from the underlying datastore. The function must accept a table. The function transforms the data according to this preprocessing pipeline:

- Randomly select one of five augmentations.
- Apply the same augmentation to the data in both columns of the table.
- Return the augmented image pair in a table.

```
function dataOut = preprocessVolumetricPatchDS(data)

img = data(1);
resp = data(2);

% 5 augmentations: nil,rot90,fliplr,flipud,rot90(fliplr)
augType = {@(x) x,@rot90,@fliplr,@flipud,@(x) rot90(fliplr(x))};

rndIdx = randi(5,1);
imgOut = augType{rndIdx}(img);
respOut = augType{rndIdx}(resp);

% Return the preprocessed data
dataOut = table(imgOut,respOut);

end
```

See Also

[imageDatastore](#) | [pixelLabelDatastore](#) | [randomPatchExtractionDatastore](#) |
[trainNetwork](#) | [transform](#)

Related Examples

- “3-D Brain Tumor Segmentation Using Deep Learning”

More About

- “Datastores for Deep Learning” on page 1-194
- “Deep Learning in MATLAB” on page 1-2
- “Create Functions in Files” (MATLAB)

Develop Custom Mini-Batch Datastore

A *mini-batch datastore* is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications that use Deep Learning Toolbox.

To preprocess sequence, time series, or text data, build your own mini-batch datastore using the framework described here. For an example showing how to use a custom mini-batch datastore, see “Train Network Using Custom Mini-Batch Datastore for Sequence Data”.

Overview

Build your custom datastore interface using the custom datastore classes and objects. Then, use the custom datastore to bring your data into MATLAB.

Designing your custom mini-batch datastore involves inheriting from the `matlab.io.Datastore` and `matlab.io.datastore.MiniBatchable` classes, and implementing the required properties and methods. You optionally can add support for shuffling during training.

Processing Needs	Classes
Mini-batch datastore for training, validation, test, and prediction data sets in Deep Learning Toolbox	<code>matlab.io.Datastore</code> and <code>matlab.io.datastore.MiniBatchable</code> See “Implement MiniBatchable Datastore” on page 1-213.
Mini-batch datastore with support for shuffling during training	<code>matlab.io.Datastore</code> , <code>matlab.io.datastore.MiniBatchable</code> , and <code>matlab.io.datastore.Shuffleable</code> See “Add Support for Shuffling” on page 1-217.

Implement MiniBatchable Datastore

To implement a custom mini-batch datastore named `MyDatastore`, create a script `MyDatastore.m`. The script must be on the MATLAB path and should contain code that

inherits from the appropriate class and defines the required methods. The code for creating a mini-batch datastore for training, validation, test, and prediction data sets in Deep Learning Toolbox must:

- Inherit from the classes `matlab.io.Datastore` and `matlab.io.datastore.MiniBatchable`.
- Define these properties: `MiniBatchSize` and `NumObservations`.
- Define these methods: `hasdata`, `read`, `reset`, and `progress`.

In addition to these steps, you can define any other properties or methods that you need to process and analyze your data.

Note If you are training a network and `trainingOptions` specifies 'Shuffle' as 'once' or 'every-epoch', then you must also inherit from the `matlab.io.datastore.Shuffleable` class. For more information, see "Add Support for Shuffling" on page 1-217.

This example shows how to create a custom mini-batch datastore for processing sequence data. Save the script in a file called `MySequenceDatastore.m`.

Steps	Implementation
<p>1 Begin defining your class. Inherit from the base class <code>matlab.io.Datastore</code> and the <code>matlab.io.datastore.MiniBatchable</code> class.</p>	<pre><code>classdef MySequenceDatastore < matlab.io.Datastore & ... matlab.io.datastore.MiniBatchable properties Datastore Labels NumClasses SequenceDimension MiniBatchSize end properties(SetAccess = protected) NumObservations end properties(Access = private) % This property is inherited from Datastore CurrentFileIndex end</code></pre>
<p>2 Define properties.</p> <ul style="list-style-type: none"> Redefine the <code>MiniBatchSize</code> and <code>NumObservations</code> properties. You optionally can assign additional property attributes to either property. For more information, see “Property Attributes” (MATLAB). You can also define properties unique to your custom mini-batch datastore. 	<pre><code>methods function ds = MySequenceDatastore(folder) % Construct a MySequenceDatastore object % Create a file datastore. The readSequence function is % defined following the class definition. fds = fileDatastore(folder, ... 'ReadFcn',@readSequence, ... 'IncludeSubfolders',true); ds.Datastore = fds; % Read labels from folder names numObservations = numel(fds.Files); for i = 1:numObservations file = fds.Files{i}; filepath = fileparts(file); [~,label] = fileparts(filepath); labels{i,1} = label; end ds.Labels = categorical(labels); ds.NumClasses = numel(unique(labels)); % Determine sequence dimension. When you define the LSTM % network architecture, you can use this property to % specify the input size of the sequenceInputLayer. X = preview(fds); ds.SequenceDimension = size(X,1); % Initialize datastore properties. ds.MiniBatchSize = 128; ds.NumObservations = numObservations;</code></pre>

Steps	Implementation
<p>3 Define methods.</p> <ul style="list-style-type: none"> • Implement the custom mini-batch datastore constructor. • Implement the <code>hasdata</code> method. • Implement the <code>read</code> method, which must return data as a table with the predictors in the first column and responses in the second column. <p>For sequence data, the sequences must be matrices of size D-by-S, where D is the number of features and S is sequence length. The value of S can vary between mini-batches.</p> <ul style="list-style-type: none"> • Implement the <code>reset</code> method. 	<pre> ds.CurrentFileIndex = 1; end function tf = hasdata(ds) % Return true if more data is available tf = ds.CurrentFileIndex + ds.MiniBatchSize - 1 ... <= ds.NumObservations; end function [data,info] = read(ds) % Read one mini-batch batch of data miniBatchSize = ds.MiniBatchSize; info = struct; for i = 1:miniBatchSize predictors{i,1} = read(ds.Datastore); responses(i,1) = ds.Labels(ds.CurrentFileIndex); ds.CurrentFileIndex = ds.CurrentFileIndex + 1; end data = preprocessData(ds,predictors,responses); end function data = preprocessData(ds,predictors,responses) % data = preprocessData(ds,predictors,responses) preprocesses % the data in predictors and responses and returns the table % data miniBatchSize = ds.MiniBatchSize; % Pad data to length of longest sequence. sequenceLengths = cellfun(@(X) size(X,2),predictors); maxSequenceLength = max(sequenceLengths); for i = 1:miniBatchSize X = predictors{i}; % Pad sequence with zeros. if size(X,2) < maxSequenceLength X(:,maxSequenceLength) = 0; end predictors{i} = X; end % Return data as a table. data = table(predictors,responses); end function reset(ds) % Reset to the start of the data reset(ds.Datastore); ds.CurrentFileIndex = 1; end </pre>

Steps	Implementation
<ul style="list-style-type: none"> • Implement the <code>progress</code> method. • You can also define methods unique to your custom mini-batch datastore. 	<pre> end methods (Hidden = true) function frac = progress(ds) % Determine percentage of data read from datastore frac = (ds.CurrentPageIndex - 1) / ds.NumObservations; end end % end class definition </pre> <p>The implementation of the read method of your custom datastore uses a function called <code>readSequence</code>. You must create this function to read sequence data from a MAT-file.</p>
4 End the <code>classdef</code> section.	<pre> function data = readSequence(filename) % data = readSequence(filename) reads the sequence X from the MAT- % filename S = load(filename); data = S.X; end </pre>

Add Support for Shuffling

To add support for shuffling, first follow the instructions in “Implement MiniBatchable Datastore” on page 1-213 and then update your implementation code in `MySequenceDatastore.m` to:

- Inherit from an additional class `matlab.io.datastore.Shuffleable`.
- Define the additional method `shuffle`.

This example code adds shuffling support to the `MySequenceDatastore` class. Vertical ellipses indicate where you should copy code from the `MySequenceDatastore` implementation.

Steps	Implementation
1 Update the class definition to also inherit from the <code>matlab.io.datastore.Shuffleable</code> class.	<pre>classdef MySequenceDatastore < matlab.io.Datastore & ... matlab.io.datastore.MiniBatchable & ... matlab.io.datastore.Shuffleable % previously defined properties % % % methods</pre>
2 Add the definition for <code>shuffle</code> to the existing <code>methods</code> section.	<pre> % previously defined methods % % function dsNew = shuffle(ds) % dsNew = shuffle(ds) shuffles the files and the % corresponding labels in the datastore. % Create a copy of datastore dsNew = copy(ds); dsNew.Datastore = copy(ds.Datastore); fds = dsNew.Datastore; % Shuffle files and corresponding labels numObservations = dsNew.NumObservations; idx = randperm(numObservations); fds.Files = fds.Files(idx); dsNew.Labels = dsNew.Labels(idx); end end end</pre>

Validate Custom Mini-Batch Datastore

If you have followed all the instructions presented here, then the implementation of your custom mini-batch datastore is complete. Before using this datastore, qualify it using the guidelines presented in “Testing Guidelines for Custom Datastores” (MATLAB).

See Also

`trainNetwork`

Related Examples

- “Train Network Using Custom Mini-Batch Datastore for Sequence Data”

More About

- “Getting Started with Datastore” (MATLAB)
- “Develop Custom Datastore” (MATLAB)
- “Developing Classes — Typical Workflow” (MATLAB)
- “Testing Guidelines for Custom Datastores” (MATLAB)
- “Deep Learning in MATLAB” on page 1-2

Deep Network Designer

- “Transfer Learning with Deep Network Designer” on page 2-2
- “Build Networks with Deep Network Designer” on page 2-15
- “Generate MATLAB Code from Deep Network Designer” on page 2-20

Transfer Learning with Deep Network Designer

In this section...

["Choose a Pretrained Network" on page 2-2](#)

["Import Network into Deep Network Designer" on page 2-3](#)

["Edit Network for Transfer Learning" on page 2-4](#)

["Check Network" on page 2-9](#)

["Export Network for Training" on page 2-10](#)

["Train Network Exported from Deep Network Designer" on page 2-10](#)

This example shows how to interactively prepare a network for transfer learning using the Deep Network Designer app. Transfer learning is the process of taking a pretrained deep learning network and fine-tuning it to learn a new task. Using transfer learning is usually much faster and easier than training a network from scratch because you can quickly transfer learned features to a new task using a smaller number of training images.

Perform transfer learning by following these steps:

- 1** Choose a pretrained network and import it into the app.
- 2** Replace the final layers with new layers adapted to the new data set:
 - a** Specify the new number of classes in your training images.
 - b** Set learning rates to learn faster in the new layers than in the transferred layers.
- 3** Export the network for training at the command line.

Choose a Pretrained Network

Deep Learning Toolbox provides a selection of pretrained image classification networks that have learned rich feature representations suitable for a wide range of images.

Transfer learning works best if your images are similar to the images originally used to train the network. If your training images are natural images like those in the ImageNet database, then any of the pretrained networks is suitable. To try a faster network first, use `googlenet` or `squeezezenet`. For a list of available networks and how to compare them, see ["Pretrained Deep Neural Networks" on page 1-15](#).

If your data is very different from the ImageNet data, it might be better to train a new network. For example, if you have tiny images, spectrograms, or nonimage data, then see instead “Build Networks with Deep Network Designer” on page 2-15.

Load a pretrained GoogLeNet network. If you need to download the network, then the function provides a link to Add-On Explorer.

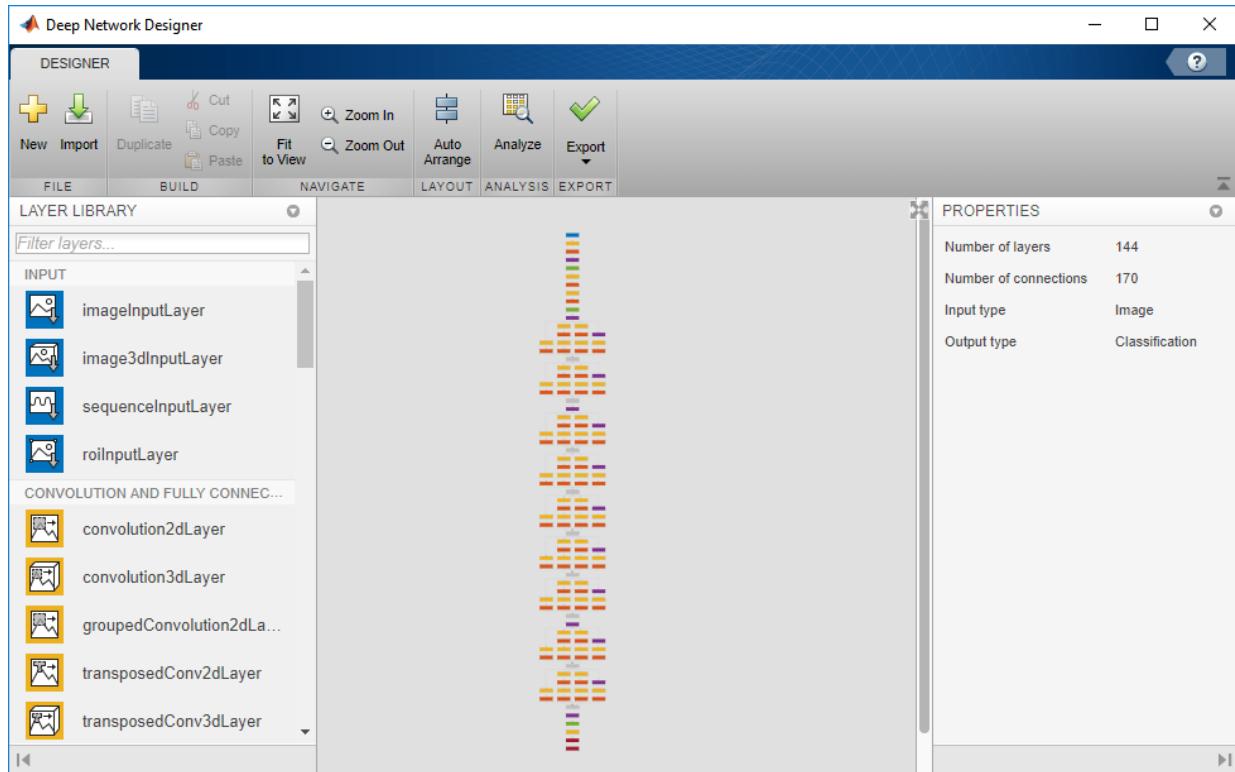
```
net = googlenet;
```

Import Network into Deep Network Designer

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

```
deepNetworkDesigner
```

Click **Import** and select the network to load from the workspace. 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**. 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.

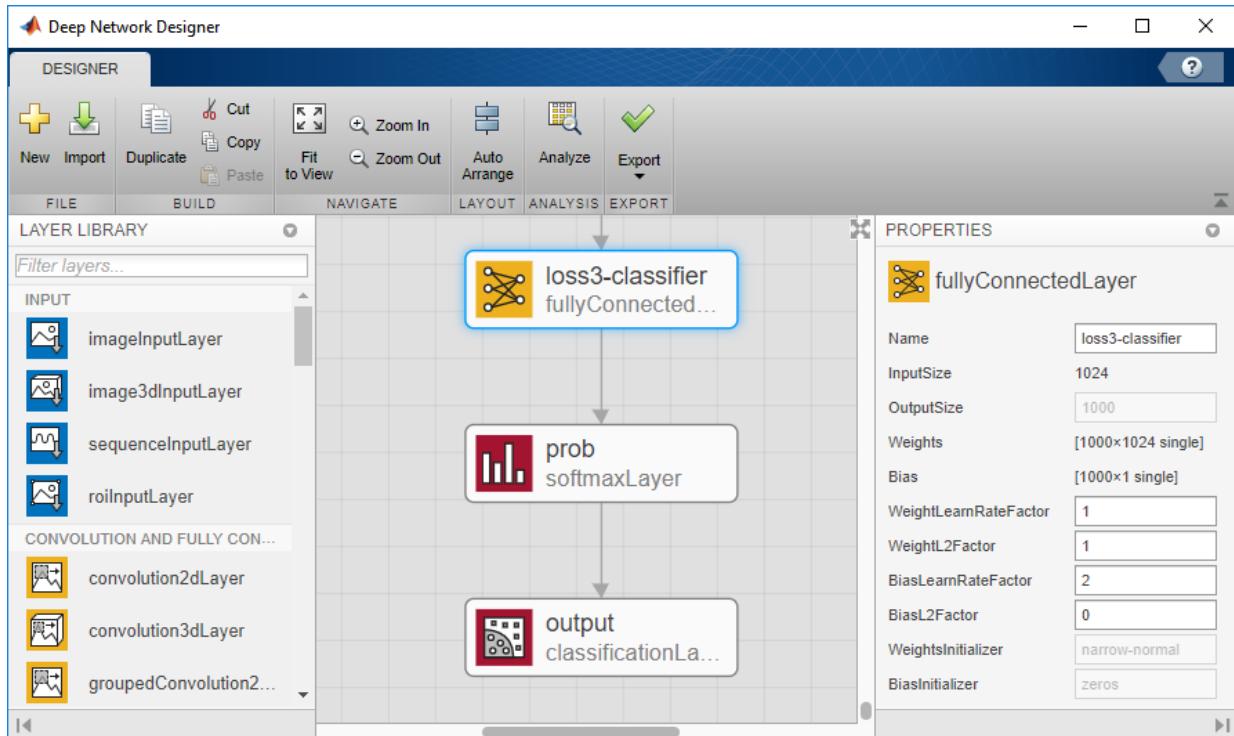
Edit Network for Transfer Learning

The network classifies input images using the last learnable layer and the final classification layer. To retrain a pretrained network to classify new images, replace these final layers with new layers adapted to the new data set.

Change Number of Classes

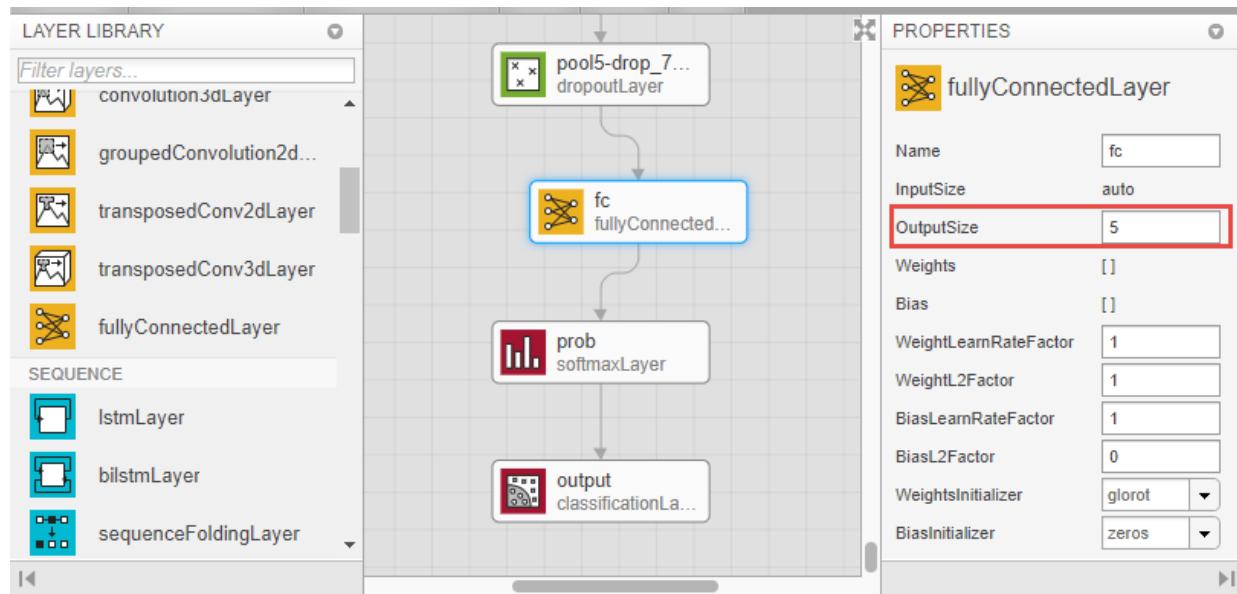
To use a pretrained network for transfer learning, you must change the number of classes to match your new data set. First, find the last learnable layer in the network. For

GoogLeNet, and most pretrained networks, the last learnable layer is a fully connected layer. Click the layer **loss3-classifier** and view its properties.

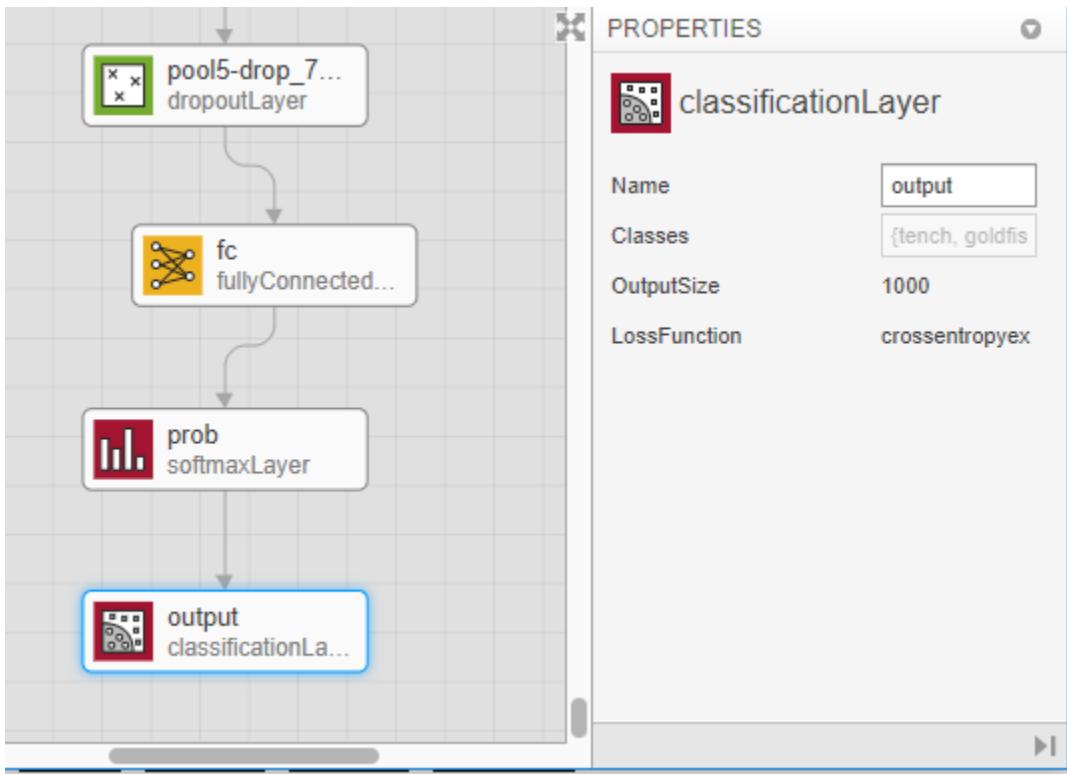


The **OutputSize** property defines the number of classes for classification problems. The **Properties** pane indicates that the pretrained network can classify images into 1000 classes. You cannot edit **OutputSize**.

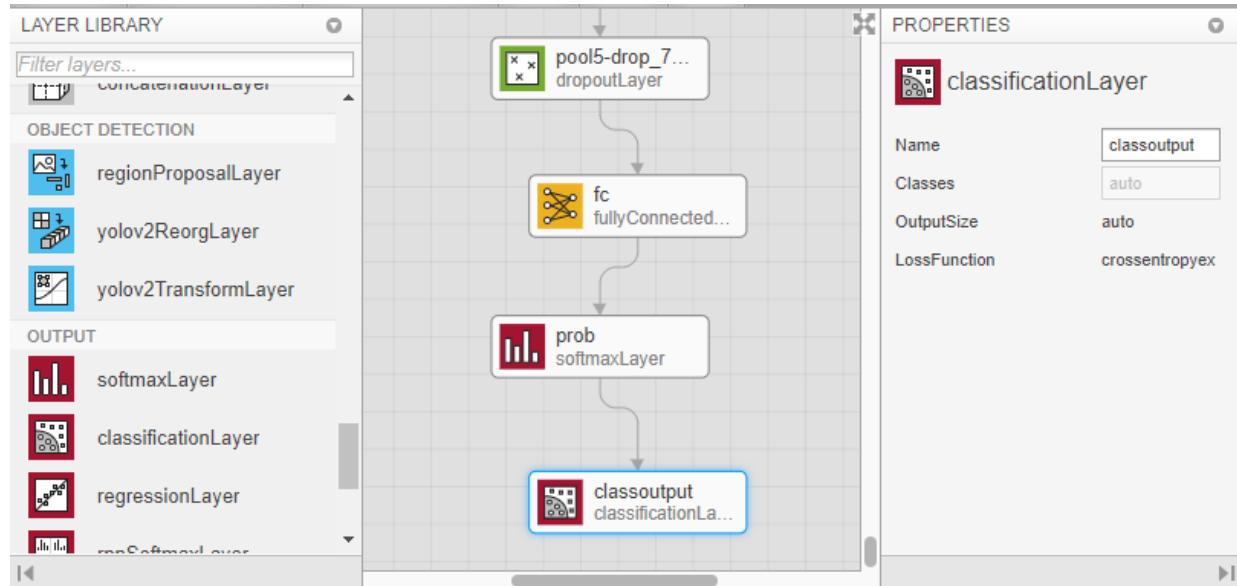
To change the number of classes, drag a new **fullyConnectedLayer** from the **Layer Library** onto the canvas. Edit the **OutputSize** property to the number of classes in your data. For this example, enter 5. Delete the original **loss3-classifier** layer and connect your new layer in its place.



Select the last layer, the classification layer. In the Properties pane, the layer property **OutputSize** shows 1000 classes and the first few class names.

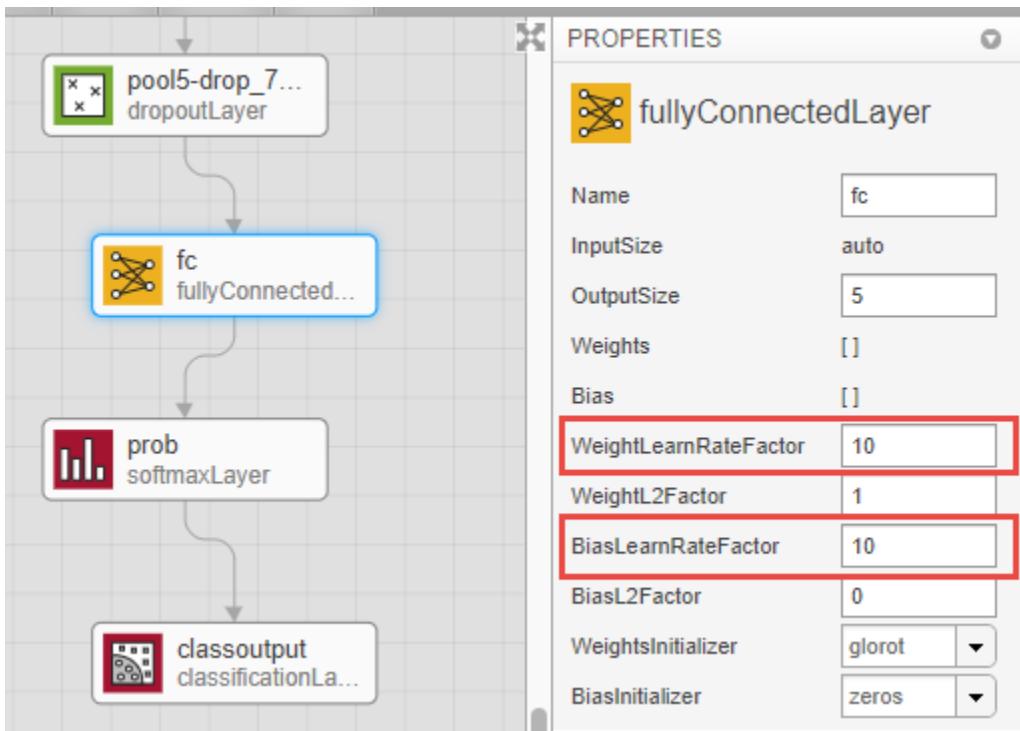


For transfer learning, you need to replace the output layer. Scroll to the end of the **Layer Library** and drag a new **classificationLayer** onto the canvas. Delete the original output layer and connect your new layer in its place. For a new output layer, you do not need to set the **OutputSize**. At training time, `trainNetwork` automatically sets the output classes of the layer from the data.



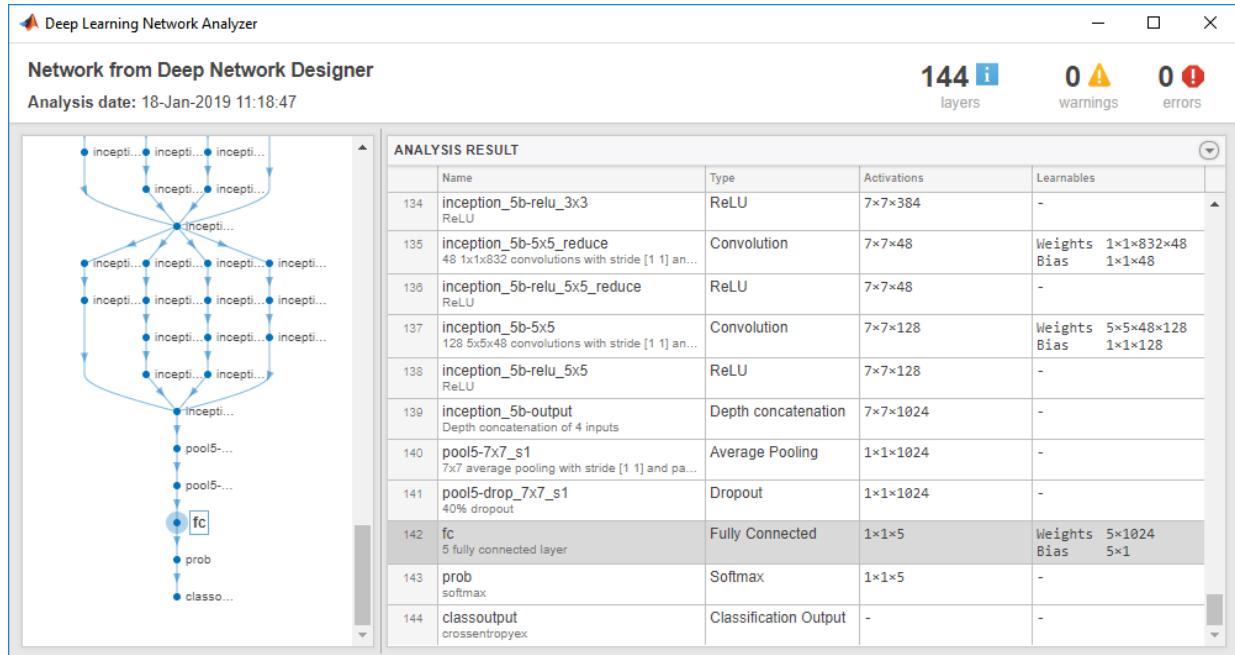
Make New Layers Learn Faster

Edit learning rates to learn faster in the new layer than in the transferred layers. On your new `fullyConnectedLayer` layer, set `WeightLearnRateFactor` and `BiasLearnRateFactor` to 10.



Check Network

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



Export Network for Training

To export the network to the workspace, return to the Deep Network Designer and click **Export**. The Deep Network Designer exports the network to a new variable containing the edited network layers, called `lgraph_1`. After exporting, you can supply the layer variable to the `trainNetwork` function. You can also generate MATLAB code that recreates the network architecture and returns it as a variable in the workspace. For more information, see “Generate MATLAB Code from Deep Network Designer” on page 2-20.

Train Network Exported from Deep Network Designer

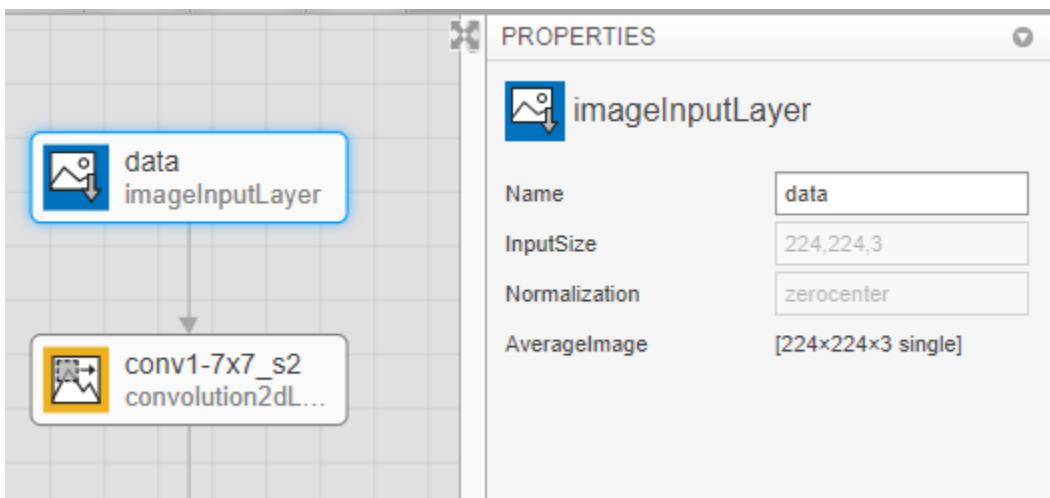
This example shows how to use a network exported from Deep Network Designer for transfer learning. After preparing the network in the app, you need to:

- Resize images.
- Specify training options.

- Train the network.

Resize Images for Transfer Learning

For transfer learning, resize your images to match the input size of the pretrained network. To find the image input size of the network, in Deep Network Designer, examine the **imageInputLayer**. For GoogLeNet, the InputSize is 224x224.



Unzip and load the images as an image datastore. This very small data set contains only 75 images in 5 classes. Divide the data into 70% for training and 30% for validation.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7);
```

If your training images are in a folder with subfolders for each class, you can create a datastore for your data by replacing **MerchData** with the folder location. Check the number of classes - you must prepare the network for transfer learning with the number of classes to match your data.

Resize images in the image datastores to match the pretrained network GoogLeNet.

```
augimdsTrain = augmentedImageDatastore([224 224],imdsTrain);
augimdsValidation = augmentedImageDatastore([224 224],imdsValidation);
```

You can also apply transformations to the images to help prevent the network from overfitting. For details, see `imageDataAugmenter`.

Set Training Options for Transfer Learning

Before training, specify training options.

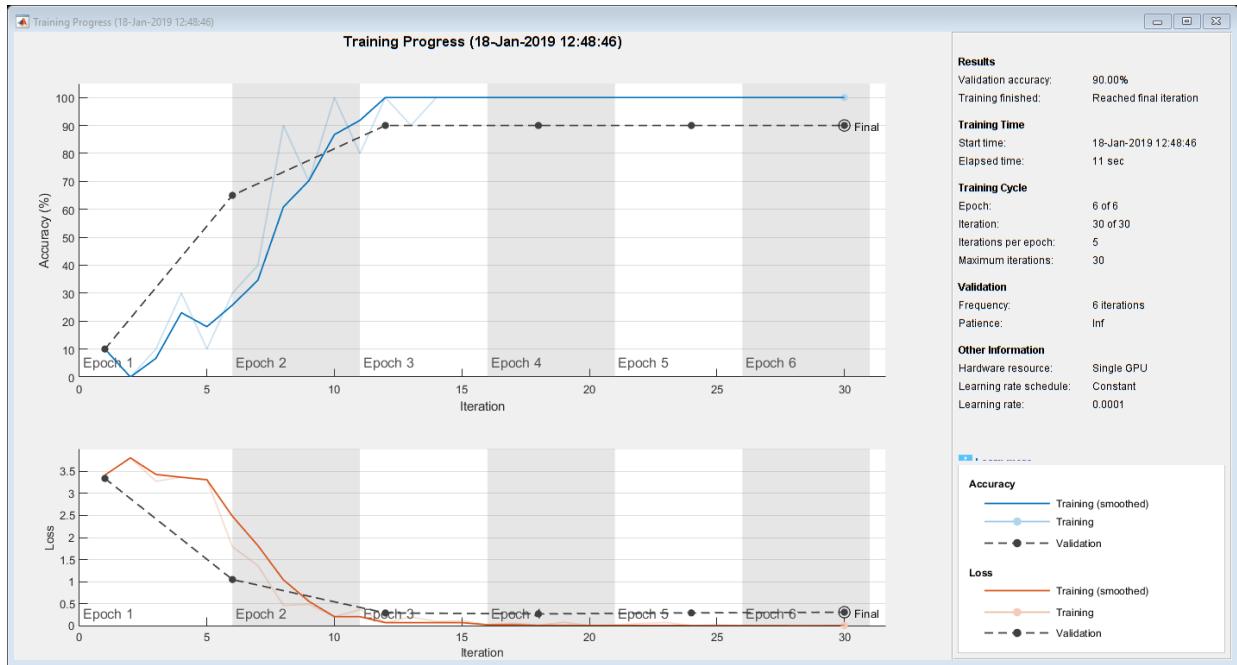
- For transfer learning, set `InitialLearnRate` to a small value to slow down learning in the transferred layers. In the app, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data set. For transfer learning, you do not need to train for as many epochs. Shuffle the data every epoch.
- Specify the mini-batch size, that is, how many images to use in each iteration.
- Specify validation data and validation frequency.
- Turn on the training plot to monitor progress while you train.

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

Train Network

To train the network, supply the layers you exported from the app, here named `lgraph_1`, your resized images, and training options, to the `trainNetwork` function. By default, `trainNetwork` uses a GPU if available (requires Parallel Computing Toolbox™). Otherwise, it uses a CPU. Training is fast because the data set is so small.

```
net = trainNetwork(augimdsTrain,lgraph_1,options);
```



Test Trained Network by Classifying Validation Images

Use the fine-tuned network to classify the validation images, and calculate the classification accuracy.

```
[YPred,probs] = classify(net,augimdsValidation);
accuracy = mean(YPred == imdsValidation.Labels)

accuracy = 0.9000
```

Display four sample validation images with predicted labels and predicted probabilities.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label) + ", " + num2str(100*max(probs(idx(i),:)),3) + "%");
end
```

MathWorks Torch, 99.3%



MathWorks Cap, 98.6%



MathWorks Cube, 41.1%



MathWorks Playing Cards, 100%



See Also

[Deep Network Designer](#)

Related Examples

- “Build Networks with Deep Network Designer” on page 2-15
- “Generate MATLAB Code from Deep Network Designer” on page 2-20
- “Deep Learning Tips and Tricks” on page 1-57
- “List of Deep Learning Layers” on page 1-28

Build Networks with Deep Network Designer

Build and edit deep learning networks interactively using the Deep Network Designer app. Using this app, you can:

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

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

Open the App and Import Networks

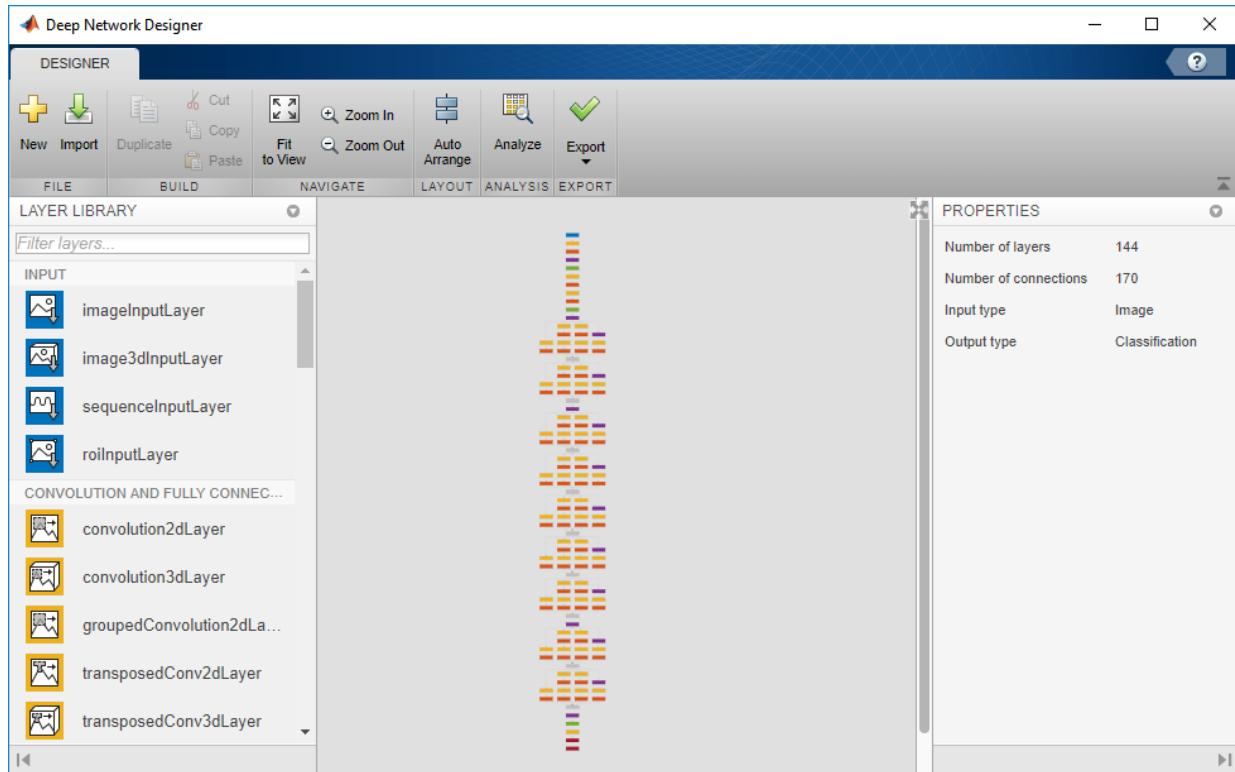
To open Deep Network Designer, on the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon. Alternatively, you can open the app from the command line:

```
deepNetworkDesigner
```

If you want to modify or copy an existing network, you can import it into the app from the workspace. To try editing a pretrained network, enter:

```
net = googlenet;
```

Click **Import** and choose the network to load from the workspace. Deep Network Designer displays a zoomed-out view of the whole network.



In the app, you can use any of the built-in layers to build a network. In addition, you can work with custom layers by creating them at the command line and then importing the network into the app. For a list of available layers and examples of custom layers, see “List of Deep Learning Layers” on page 1-28.

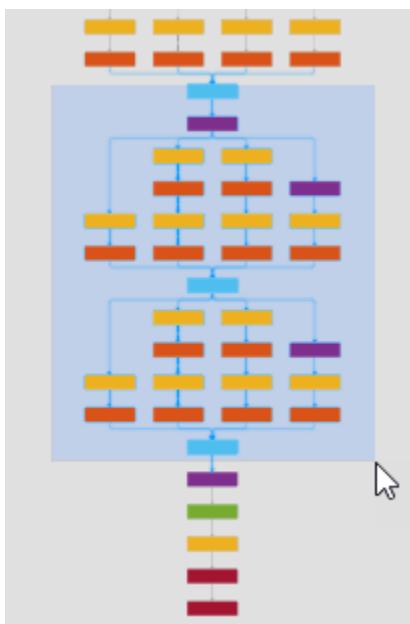
Create and Edit Networks

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

To view and edit layer properties, select a layer. For information on all layer properties, click the layer name in the table on the “List of Deep Learning Layers” on page 1-28 page.

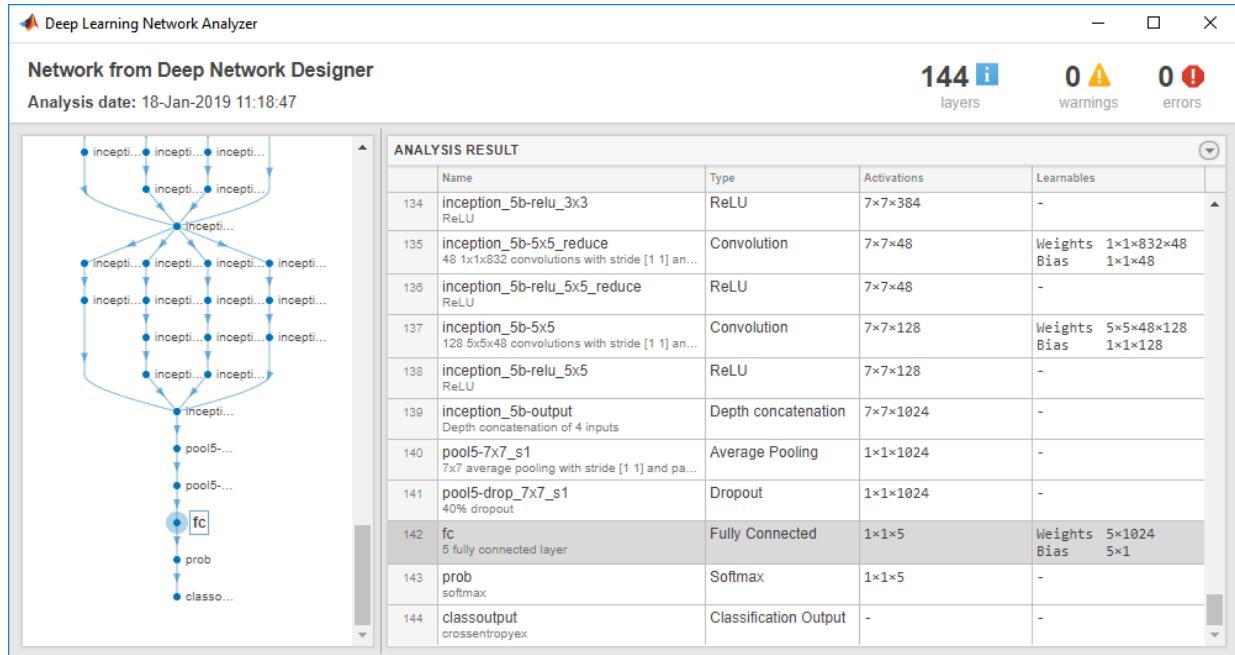
For tips on selecting a suitable network architecture, see “Deep Learning Tips and Tricks” on page 1-57.

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



Check Network

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



Export Network for Training

To export the network to the workspace, return to the Deep Network Designer and click **Export**. The Deep Network Designer exports the network to a new variable containing the edited network layers. After exporting, you can supply the layer variable to the `trainNetwork` function.

Train the network. For this example, assume that the layers exported from the app are named `lgraph_1`, and that your images are in an augmented image datastore called `images`.

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

For information on resizing and processing images, see “Preprocess Images for Deep Learning” on page 1-201.

For an example script showing how to train a network after editing it in the app, see “Train Network Exported from Deep Network Designer” on page 2-10.

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

For an example showing how to generate MATLAB code that recreates the network architecture and returns it as a variable in the workspace, see “Generate MATLAB Code from Deep Network Designer” on page 2-20.

See Also

Deep Network Designer

Related Examples

- “List of Deep Learning Layers” on page 1-28
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-20
- “Deep Learning Tips and Tricks” on page 1-57

Generate MATLAB Code from Deep Network Designer

The Deep Network Designer app enables you to generate MATLAB code for a network that you create or edit in the app. After generating a script, you can:

- Run the script to recreate the network layers created in the app.
- To train the network, run the script and then supply the layers to the `trainNetwork` function.
- Examine the code to learn how to create and connect layers programmatically.
- To modify the layers, edit the code, or run the script and import the network back into the app for editing.

Generate MATLAB Code and Recreate Network Layers

To generate MATLAB code in Deep Network Designer, choose one of these options:

- To generate a script to recreate the layers in your network, select **Export > Generate Code**.
- To generate a script to recreate your network including any learnable parameters, select **Export > Generate Code with Pretrained Parameters**. The app creates a script and a MAT-file containing the learnable parameters (weights and biases) from your network. Run the script to recreate the network layers including the learnable parameters from the MAT-file. Use this option to preserve the weights if you want to perform transfer learning.

Running the generated script returns the network architecture as a variable in the workspace. Depending on the network architecture, the variable is a layer graph named *lgraph* or a layer array named *layers*.

Train Network

If the layers require training, supply the layer graph or layer array to the `trainNetwork` function.

```
net = trainNetwork(data,lgraph,options);
```

Before training, you must define the data and training options. This example defines data and options suitable for training a GoogLeNet network prepared for transfer learning, as shown in “Transfer Learning with Deep Network Designer” on page 2-2.

- 1 Define the data. For this example, use an image datastore with 5 classes split into training and validation sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomize');
```

You usually need to resize the images to match the input size of the network. Resize at training time to 224-by-224 to match the pretrained network GoogLeNet.

```
augimdsTrain = augmentedImageDatastore([224 224],imdsTrain);
augimdsValidation = augmentedImageDatastore([224 224],imdsValidation);
```

- 2 Define the training options. For example, turn on the progress plot, specify the validation data, specify the number of images to use in each iteration (`MiniBatchSize`) and the number of training cycles to perform on the entire data set (`MaxEpochs`). For transfer learning, set `InitialLearnRate` to a small value to slow down learning in the transferred layers.

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

- 3 To recreate the network layers, run the generated script.
- 4 To train the network, supply the layer graph or layer array to the `trainNetwork` function, using the specified data and training options.

```
net = trainNetwork(augimdsTrain,lgraph,options);
```

For an example script that sets training options for transfer learning on a network prepared in Deep Network Designer, see “Train Network Exported from Deep Network Designer” on page 2-10.

Use Network for Prediction

To use the trained network for prediction, use the `predict` function. For example, use the network to predict the class of `peppers.png`.

```
img = imread("peppers.png");
img = imresize(img, [128, 128]);
label = predict(net, img);
imshow(img);
title(label);
```

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

See Also

[Deep Network Designer](#) | [trainNetwork](#) | [trainingOptions](#)

Related Examples

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

Deep Learning in the Cloud

- “Scale Up Deep Learning in Parallel and in the Cloud” on page 3-2
- “Deep Learning with MATLAB on Multiple GPUs” on page 3-7

Scale Up Deep Learning in Parallel and in the Cloud

In this section...

"Deep Learning on Multiple GPUs" on page 3-2

"Deep Learning in the Cloud" on page 3-4

"Advanced Support for Fast Multi-Node GPU Communication" on page 3-5

Deep Learning on Multiple GPUs

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

Training deep networks is extremely computationally intensive and you can usually accelerate training by using a high performance GPU. If you do not have a suitable GPU, you can train on one or more CPU cores instead, or rent GPUs in the cloud. You can train a convolutional neural network on a single GPU or CPU, or on multiple GPUs or CPU cores, or in parallel on a cluster. Using GPU or any parallel option requires Parallel Computing Toolbox.

Tip GPU support is automatic. By default, the `trainNetwork` function uses a GPU if available.

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

If you want to use more resources, you can scale up deep learning training to the cloud.

Deep Learning Built-In Parallel Support

Training Resource	Settings	Learn More
Single GPU on local machine	Automatic. By default, the <code>trainNetwork</code> function uses a GPU if available.	"ExecutionEnvironment" "Create Simple Deep Learning Network for Classification"
Multiple GPUs on local machine	Specify <code>'ExecutionEnvironment', 'multi-gpu'</code> with the <code>trainingOptions</code> function.	"ExecutionEnvironment" "Select Particular GPUs to Use for Training" on page 3-7
Multiple CPU cores on local machine	Specify <code>'ExecutionEnvironment', 'parallel'</code> . With default settings, <code>'parallel'</code> uses the local cluster profile. Only use CPUs if you do not have a GPU, because CPUs are generally far slower than GPUs for training.	"ExecutionEnvironment"
Cluster or in the cloud	After setting a default cluster, specify <code>'ExecutionEnvironment', 'parallel'</code> with the <code>trainingOptions</code> function. Training executes on the cluster and returns the built-in progress plot to your local MATLAB.	"Train Network in the Cloud Using Automatic Parallel Support"

Train Multiple Deep Networks in Parallel

Training Scenario	Recommendations	Learn More
Interactively on your local machine or in the cloud	Use a <code>parfor</code> loop to train multiple networks, and plot results using the <code>OutputFcn</code> . Runs locally by default, or choose a different cluster profile.	"Use <code>parfor</code> to Train Multiple Deep Learning Networks"
In the background on your local machine or in the cloud	Use <code>parfeval</code> to train without blocking your local MATLAB, and plot results using the <code>OutputFcn</code> . Runs locally by default, or choose a different cluster profile.	"Run Multiple Deep Learning Experiments" "Use <code>parfeval</code> to Train Multiple Deep Learning Networks"
On a cluster, and turn off your local machine	Use the <code>batch</code> function to send training code to the cluster. You can close MATLAB and fetch results later.	"Send Deep Learning Batch Job to Cluster"

Deep Learning in the Cloud

If your deep learning training takes hours or days, you can rent high performance GPUs in the cloud to accelerate training. Working in the cloud requires some initial setup, but after the initial setup using the cloud can reduce training time, or allow you to train more networks in the same time. To try deep learning in the cloud, you can follow example steps to set up your accounts, copy your data into the cloud, and create a cluster. After this initial setup, you can run your training code with minimal changes to run in the cloud. After setting up your default cluster, simply specify the training option `'ExecutionEnvironment', 'parallel'` to train networks on your cloud cluster on multiple GPUs.

Configure Deep Learning in the Cloud	Notes	Learn More
Set up MathWorks Cloud Center and Amazon accounts	One-time setup.	Getting Started with Cloud Center

Configure Deep Learning in the Cloud	Notes	Learn More
Create a cluster	Use Cloud Center to set up and run clusters in the Amazon cloud. For deep learning, choose a machine type with GPUs such as the P2 or G3 instances.	Create a Cloud Cluster
Upload data to the cloud	To work with data in the cloud, upload to Amazon S3. Use datastores to access the data in S3 from your desktop client MATLAB, or from your cluster workers, without changing your code.	"Upload Deep Learning Data to the Cloud"

Advanced Support for Fast Multi-Node GPU Communication

If you are using a Linux compute cluster with fast interconnects between machines such as Infiniband, or fast interconnects between GPUs on different machines, such as GPUDirect RDMA, you might be able to take advantage of fast multi-node support in MATLAB. Enable this support on all the workers in your pool by setting the environment variable `PARALLEL_SERVER_FAST_MULTINODE_GPU_COMMUNICATION` to 1. Set this environment variable in the Cluster Profile Manager.

This feature is part of the NVIDIA NCCL library for GPU communication. To configure it, you must set additional environment variables to define the network interface protocol, especially `NCCL_SOCKET_IFNAME`. For more information, see the NCCL documentation and in particular the section on NCCL Knobs.

See Also

More About

- “Deep Learning with MATLAB on Multiple GPUs” on page 3-7
- “Run Multiple Deep Learning Experiments”

- “Send Deep Learning Batch Job to Cluster”
- “Use parfeval to Train Multiple Deep Learning Networks”
- “Use parfor to Train Multiple Deep Learning Networks”
- “Upload Deep Learning Data to the Cloud”

Deep Learning with MATLAB on Multiple GPUs

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

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

If you want to use more resources, you can scale up deep learning training to clusters or the cloud. To learn more about parallel options, see "Scale Up Deep Learning in Parallel and in the Cloud" on page 3-2. To try an example, see "Train Network in the Cloud Using Automatic Parallel Support" on page 3-8.

Select Particular GPUs to Use for Training

To use all available GPUs on your machine, simply specify the training option '`'ExecutionEnvironment', 'multi-gpu'`'.

To select one of multiple GPUs to use to train a single model, use:

```
gpuDevice(index)
```

If you want to train a single model using multiple GPUs, and do not want to use all your GPUs, open the parallel pool in advance, and select the GPUs manually. To select particular GPUs, use the following code, where `gpuIndices` are the indices of the GPUs:

```
parpool('local', numel(gpuIndices));
spmd, gpuDevice(gpuIndices(labindex)); end
```

When you run `trainNetwork` with the '`'multi-gpu'`' `ExecutionEnvironment` (or '`'parallel'` for the same result), the training function will use this pool and not open a new one.

Another option is to select workers using the '`WorkerLoad`' option in `trainingOptions`. For example:

```
parpool('local', 5);
opts = trainingOptions('sgdm', 'WorkerLoad', [1 1 1 0 1], ...)
```

In this case, the 4th worker is part of the pool but idle, which is not an ideal use of the parallel resources. It is more efficient to specify GPUs with `gpuDevice`.

If you want to train multiple models with one GPU each, start a MATLAB session for each and select a device using `gpuDevice`.

Alternatively, use a `parfor` loop:

```
parfor i=1:gpuDeviceCount
    trainNetwork(...);
end
```

Train Network in the Cloud Using Automatic Parallel Support

This example shows how to train a convolutional neural network using MATLAB automatic support for parallel training. Deep learning training often takes hours or days. With parallel computing, you can speed up training using multiple graphical processing units (GPUs) locally or in a cluster in the cloud. If you have access to a machine with multiple GPUs, then you can complete this example on a local copy of the data. If you want to use more resources, then you can scale up deep learning training to the cloud. To learn more about your options for parallel training, see “Scale Up Deep Learning in Parallel and in the Cloud” on page 3-2. This example guides you through the steps to train a deep learning network in a cluster in the cloud using MATLAB automatic parallel support.

Requirements

Before you can run the example, you need to configure a cluster and upload data to the cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. After that, upload your data to an Amazon S3 bucket and access it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see “Upload Deep Learning Data to the Cloud”.

Set Up Parallel Pool

Start a parallel pool in the cluster and set the number of workers to the number of GPUs in your cluster. If you specify more workers than GPUs, then the remaining workers are

idle. This example assumes that the cluster you are using is set as the default cluster profile. Check the default cluster profile on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**.

```
numberOfWorkers = 8;
parpool(numberOfWorkers);
```

```
Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
connected to 8 workers.
```

Load Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. In this example, you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See “Upload Deep Learning Data to the Cloud”.

```
imdsTrain = imageDatastore('s3://cifar10cloud/cifar10/train', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Train the network with augmented image data by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture and Training Options

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
blockDepth = 4; % blockDepth controls the depth of a convolutional block
netWidth = 32; % netWidth controls the number of filters in a convolutional block

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,blockDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];
```

Define the training options. Train the network in parallel using the current cluster, by setting the execution environment to `parallel`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU constant. Scale the learning rate according to the mini-batch size. Use a learning rate schedule to drop the learning rate as the training progresses. Turn on the training progress plot to obtain visual feedback during training.

```
miniBatchSize = 256 * numberOfWorkers;
initialLearnRate = 1e-1 * miniBatchSize/256;

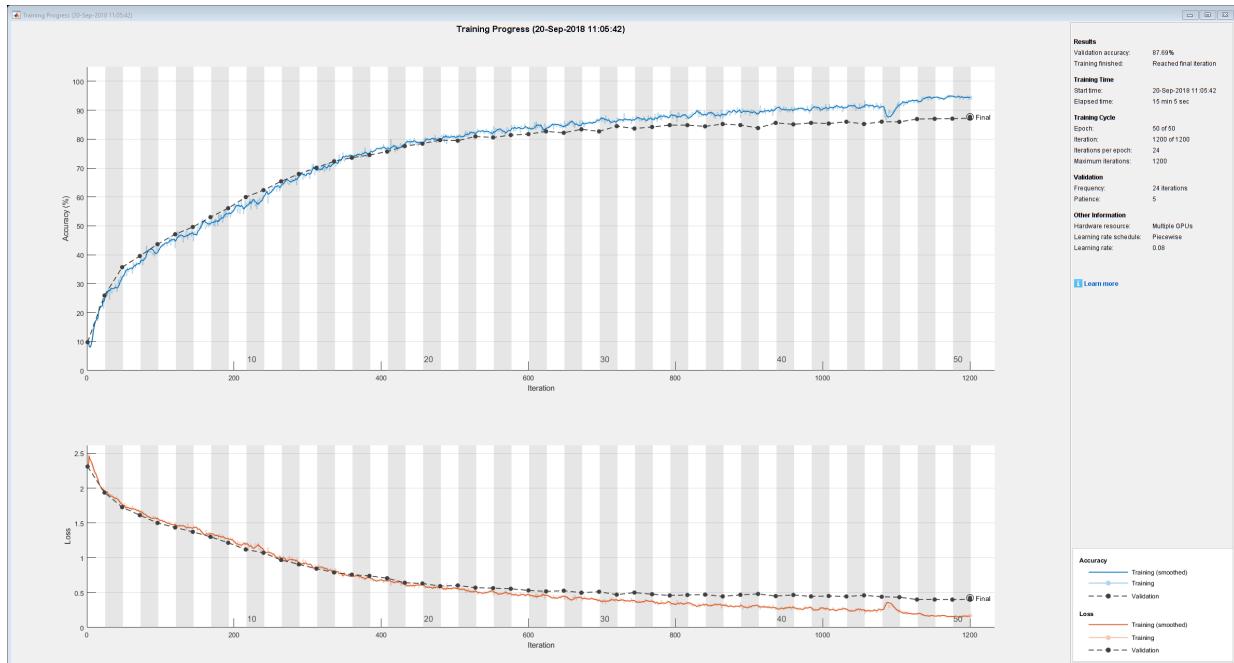
options = trainingOptions('sgdm',...
    'ExecutionEnvironment','parallel',...
    % Turn on automatic parallel support.
    'InitialLearnRate',initialLearnRate,... % Set the initial learning rate.
    'MiniBatchSize',miniBatchSize,... % Set the MiniBatchSize.
    'Verbose',false,... % Do not send command line output.
    'Plots','training-progress',... % Turn on the training progress plot.
    'L2Regularization',1e-10,...%
    'MaxEpochs',50,...%
    'Shuffle','every-epoch',...%
```

```
'ValidationData',imdsTest, ...
'ValidationFrequency',floor(numel(imdsTrain.Files)/miniBatchSize), ...
'LearnRateSchedule','piecewise', ...
'LearnRateDropFactor',0.1, ...
'LearnRateDropPeriod',45);
```

Train Network and Use for Classification

Train the network in the cluster. During training, the plot displays the progress.

```
net = trainNetwork(augmentedImdsTrain,layers,options)
```



```
net =
SeriesNetwork with properties:
```

```
Layers: [43x1 nnet.cnn.layer.Layer]
```

Determine the accuracy of the network, by using the trained network to classify the test images on your local machine. Then compare the predicted labels to the actual labels.

```
YPredicted = classify(net,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

Define Helper Function

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same')
        batchNormalizationLayer
        reluLayer
    ];
    layers = repmat(layers,numConvLayers,1);
end
```

See Also

[gpuDevice](#) | [imageDatastore](#) | [spmd](#) | [trainNetwork](#) | [trainingOptions](#)

Related Examples

- “Run Multiple Deep Learning Experiments”
- “Upload Deep Learning Data to the Cloud”
- “Use parfor to Train Multiple Deep Learning Networks”
- “Scale Up Deep Learning in Parallel and in the Cloud” on page 3-2

Neural Network Design Book

The developers of the Deep Learning Toolbox software have written a textbook, *Neural Network Design* (Hagan, Demuth, and Beale, ISBN 0-9717321-0-8). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of the MATLAB environment and Deep Learning Toolbox software. Example programs from the book are used in various sections of this documentation. (You can find all the book example programs in the Deep Learning Toolbox software by typing `nnd`.)

Obtain this book from John Stovall at (303) 492-3648, or by email at John.Stovall@colorado.edu.

The *Neural Network Design* textbook includes:

- An Instructor's Manual for those who adopt the book for a class
- Transparency Masters for class use

If you are teaching a class and want an Instructor's Manual (with solutions to the book exercises), contact John Stovall at (303) 492-3648, or by email at John.Stovall@colorado.edu

To look at sample chapters of the book and to obtain Transparency Masters, go directly to the Neural Network Design page at:

<http://hagan.okstate.edu/nnd.html>

From this link, you can obtain sample book chapters in PDF format and you can download the Transparency Masters by clicking **Transparency Masters (3.6MB)**.

You can get the Transparency Masters in PowerPoint or PDF format.

Neural Network Objects, Data, and Training Styles

- “Workflow for Neural Network Design” on page 4-2
- “Four Levels of Neural Network Design” on page 4-4
- “Neuron Model” on page 4-5
- “Neural Network Architectures” on page 4-11
- “Create Neural Network Object” on page 4-17
- “Configure Neural Network Inputs and Outputs” on page 4-21
- “Understanding Deep Learning Toolbox Data Structures” on page 4-23
- “Neural Network Training Concepts” on page 4-28

Workflow for Neural Network Design

The work flow for the neural network design process has seven primary steps. Referenced topics discuss the basic ideas behind steps 2, 3, and 5.

- 1** Collect data
- 2** Create the network — “Create Neural Network Object” on page 4-17
- 3** Configure the network — “Configure Neural Network Inputs and Outputs” on page 4-21
- 4** Initialize the weights and biases
- 5** Train the network — “Neural Network Training Concepts” on page 4-28
- 6** Validate the network
- 7** Use the network

Data collection in step 1 generally occurs outside the framework of Deep Learning Toolbox software, but it is discussed in general terms in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2. Details of the other steps and discussions of steps 4, 6, and 7, are discussed in topics specific to the type of network.

The Deep Learning Toolbox software uses the network object to store all of the information that defines a neural network. This topic describes the basic components of a neural network and shows how they are created and stored in the network object.

After a neural network has been created, it needs to be configured and then trained. Configuration involves arranging the network so that it is compatible with the problem you want to solve, as defined by sample data. After the network has been configured, the adjustable network parameters (called weights and biases) need to be tuned, so that the network performance is optimized. This tuning process is referred to as training the network. Configuration and training require that the network be provided with example data. This topic shows how to format the data for presentation to the network. It also explains network configuration and the two forms of network training: incremental training and batch training.

See Also

More About

- “Four Levels of Neural Network Design” on page 4-4
- “Neuron Model” on page 4-5
- “Neural Network Architectures” on page 4-11
- “Understanding Deep Learning Toolbox Data Structures” on page 4-23

Four Levels of Neural Network Design

There are four different levels at which the neural network software can be used. The first level is represented by the GUIs that are described in “Getting Started with Deep Learning Toolbox”. These provide a quick way to access the power of the toolbox for many problems of function fitting, pattern recognition, clustering and time series analysis.

The second level of toolbox use is through basic command-line operations. The command-line functions use simple argument lists with intelligent default settings for function parameters. (You can override all of the default settings, for increased functionality.) This topic, and the ones that follow, concentrate on command-line operations.

The GUIs described in Getting Started can automatically generate MATLAB code files with the command-line implementation of the GUI operations. This provides a nice introduction to the use of the command-line functionality.

A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox.

The fourth level of toolbox usage is the ability to modify any of the code files contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

The first level of toolbox use (through the GUIs) is described in Getting Started which also introduces command-line operations. The following topics will discuss the command-line operations in more detail. The customization of the toolbox is described in “Define Shallow Neural Network Architectures”.

See Also

More About

- “Workflow for Neural Network Design” on page 4-2

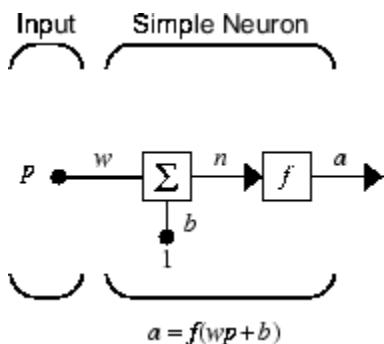
Neuron Model

In this section...

- “Simple Neuron” on page 4-5
- “Transfer Functions” on page 4-6
- “Neuron with Vector Input” on page 4-7

Simple Neuron

The fundamental building block for neural networks is the single-input neuron, such as this example.



There are three distinct functional operations that take place in this example neuron. First, the scalar input p is multiplied by the scalar weight w to form the product wp , again a scalar. Second, the weighted input wp is added to the scalar bias b to form the net input n . (In this case, you can view the bias as shifting the function f to the left by an amount b .) The bias is much like a weight, except that it has a constant input of 1.) Finally, the net input is passed through the transfer function f , which produces the scalar output a . The names given to these three processes are: the weight function, the net input function and the transfer function.

For many types of neural networks, the weight function is a product of a weight times the input, but other weight functions (e.g., the distance between the weight and the input, $|w - p|$) are sometimes used. (For a list of weight functions, type `help nnweight`.) The most common net input function is the summation of the weighted inputs with the bias, but other operations, such as multiplication, can be used. (For a list of net input functions, type `help nnnnetinput`.) “Introduction to Radial Basis Neural Networks” on page 8-2

discusses how distance can be used as the weight function and multiplication can be used as the net input function. There are also many types of transfer functions. Examples of various transfer functions are in “Transfer Functions” on page 4-6. (For a list of transfer functions, type `help nntransfer`.)

Note that w and b are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, you can train the network to do a particular job by adjusting the weight or bias parameters.

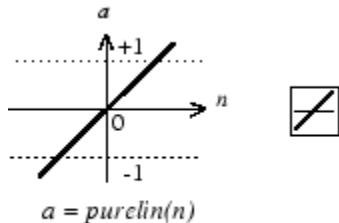
All the neurons in the Deep Learning Toolbox software have provision for a bias, and a bias is used in many of the examples and is assumed in most of this toolbox. However, you can omit a bias in a neuron if you want.

Transfer Functions

Many transfer functions are included in the Deep Learning Toolbox software.

Two of the most commonly used functions are shown below.

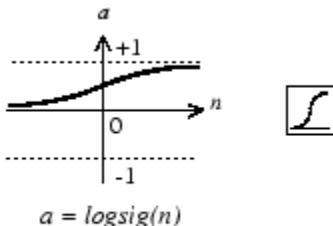
The following figure illustrates the linear transfer function.



Linear Transfer Function

Neurons of this type are used in the final layer of multilayer networks that are used as function approximators. This is shown in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

The sigmoid transfer function shown below takes the input, which can have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



Log-Sigmoid Transfer Function

This transfer function is commonly used in the hidden layers of multilayer networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons replace the general f in the network diagram blocks to show the particular transfer function being used.

For a complete list of transfer functions, type `help nntransfer`. You can also specify your own transfer functions.

You can experiment with a simple neuron and various transfer functions by running the example program `nnd2n1`.

Neuron with Vector Input

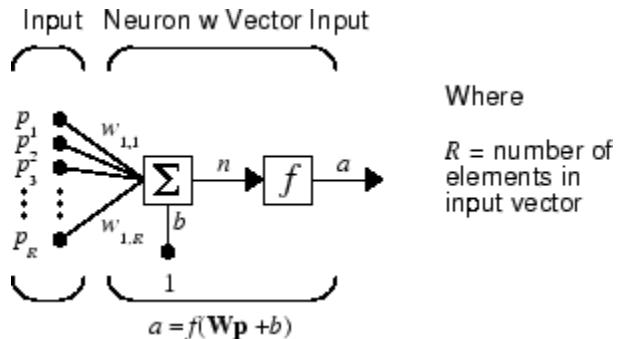
The simple neuron can be extended to handle inputs that are vectors. A neuron with a single R -element input vector is shown below. Here the individual input elements

$$p_1, p_2, \dots, p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \dots, w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply $\mathbf{W}\mathbf{p}$, the dot product of the (single row) matrix \mathbf{W} and the vector \mathbf{p} . (There are other weight functions, in addition to the dot product, such as the distance between the row of the weight matrix and the input vector, as in “Introduction to Radial Basis Neural Networks” on page 8-2.)



Where

R = number of elements in input vector

The neuron has a bias b , which is summed with the weighted inputs to form the net input n . (In addition to the summation, other net input functions can be used, such as the multiplication that is used in “Introduction to Radial Basis Neural Networks” on page 8-2.) The net input n is the argument of the transfer function f .

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

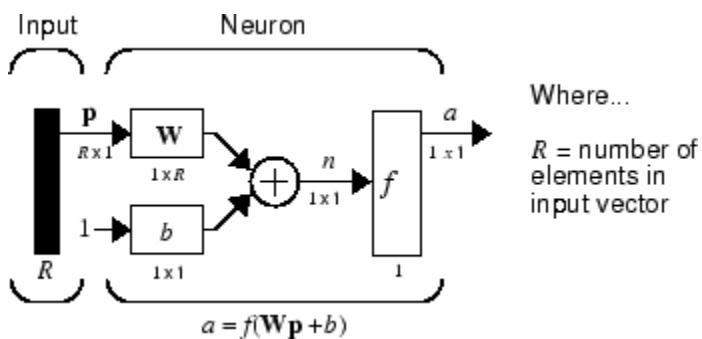
This expression can, of course, be written in MATLAB code as

$$\mathbf{n} = \mathbf{W} * \mathbf{p} + \mathbf{b}$$

However, you will seldom be writing code at this level, for such code is already built into functions to define and simulate entire networks.

Abbreviated Notation

The figure of a single neuron shown above contains a lot of detail. When you consider networks with many neurons, and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which is used later in circuits of multiple neurons, is shown here.



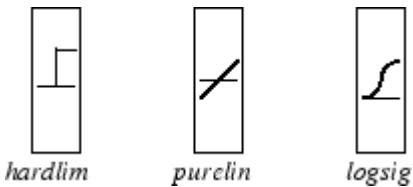
Here the input vector \mathbf{p} is represented by the solid dark vertical bar at the left. The dimensions of \mathbf{p} are shown below the symbol \mathbf{p} in the figure as $R \times 1$. (Note that a capital letter, such as R in the previous sentence, is used when referring to the size of a vector.) Thus, \mathbf{p} is a vector of R input elements. These inputs postmultiply the single-row, R -column matrix \mathbf{W} . As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias b . The net input to the transfer function f is n , the sum of the bias b and the product \mathbf{Wp} . This sum is passed to the transfer function f to get the neuron's output a , which in this case is a scalar. Note that if there were more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the previous figure. A layer includes the weights, the multiplication and summing operations (here realized as a vector product \mathbf{Wp}), the bias b , and the transfer function f . The array of inputs, vector \mathbf{p} , is not included in or called a layer.

As with the “Simple Neuron” on page 4-5, there are three operations that take place in the layer: the weight function (matrix multiplication, or dot product, in this case), the net input function (summation, in this case), and the transfer function.

Each time this abbreviated network notation is used, the sizes of the matrices are shown just below their matrix variable names. This notation will allow you to understand the architectures and follow the matrix mathematics associated with them.

As discussed in “Transfer Functions” on page 4-6, when a specific transfer function is to be used in a figure, the symbol for that transfer function replaces the f shown above. Here are some examples.



You can experiment with a two-element neuron by running the example program `nnd2n2`.

See Also

More About

- “Neural Network Architectures” on page 4-11
- “Workflow for Neural Network Design” on page 4-2

Neural Network Architectures

In this section...

"One Layer of Neurons" on page 4-11

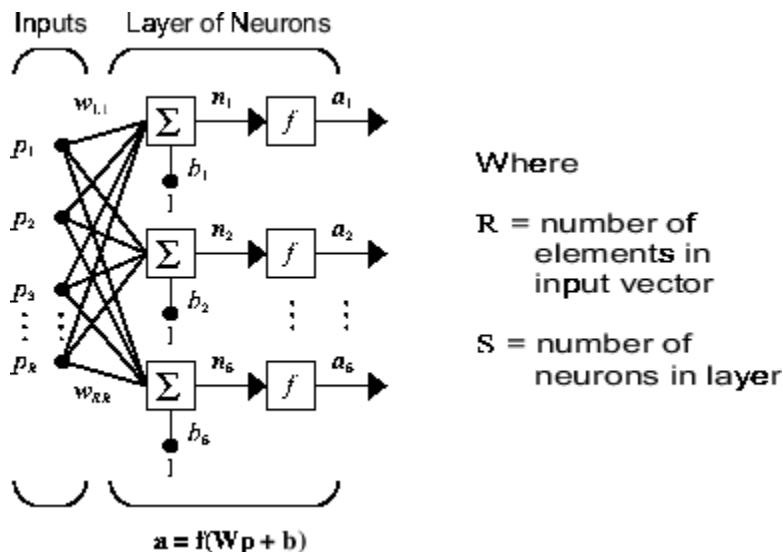
"Multiple Layers of Neurons" on page 4-13

"Input and Output Processing Functions" on page 4-15

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

One Layer of Neurons

A one-layer network with R input elements and S neurons follows.



In this network, each element of the input vector \mathbf{p} is connected to each neuron input through the weight matrix \mathbf{W} . The i th neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$. The various $n(i)$ taken together form an S -element net input vector \mathbf{n} . Finally, the neuron layer outputs form a column vector \mathbf{a} . The expression for \mathbf{a} is shown at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., R is not necessarily equal to S). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

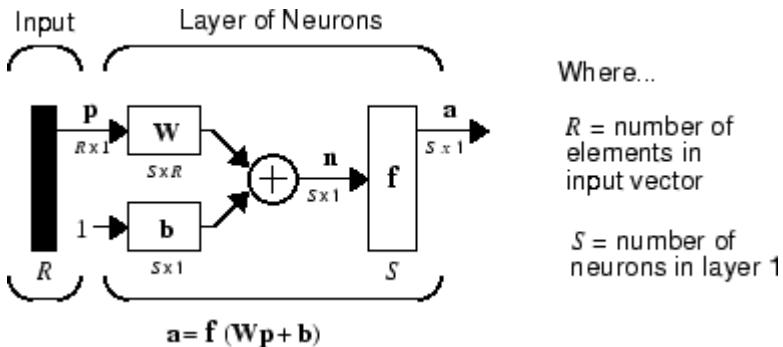
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix \mathbf{W} .

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix \mathbf{W} indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R -input one-layer network also can be drawn in abbreviated notation.



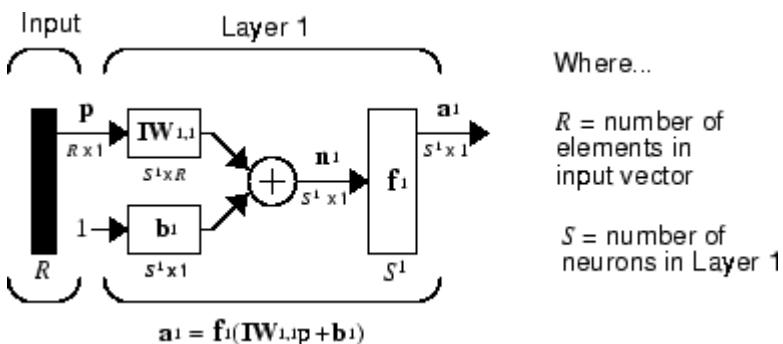
Here \mathbf{p} is an R -length input vector, \mathbf{W} is an $S \times R$ matrix, \mathbf{a} and \mathbf{b} are S -length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector \mathbf{b} , the summer, and the transfer function blocks.

Inputs and Layers

To describe networks having multiple layers, the notation must be extended. Specifically, it needs to make a distinction between weight matrices that are connected to inputs and

weight matrices that are connected between layers. It also needs to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs *input weights*; we will call weight matrices connected to layer outputs *layer weights*. Further, superscripts are used to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, the one-layer multiple input network shown earlier is redrawn in abbreviated form here.

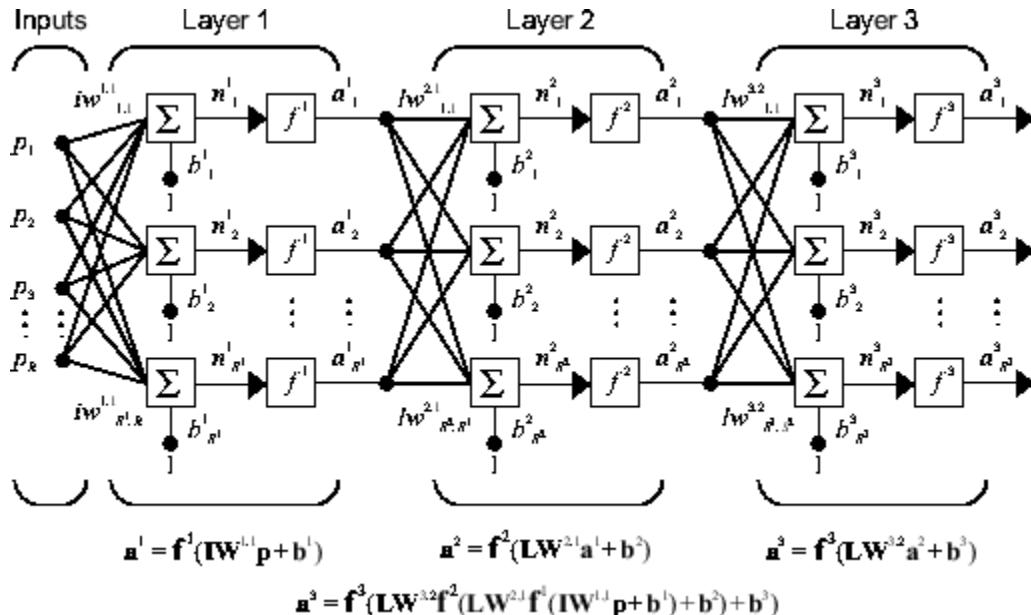


As you can see, the weight matrix connected to the input vector p is labeled as an input weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

"Multiple Layers of Neurons" on page 4-13 uses layer weight (**LW**) matrices as well as input weight (**IW**) matrices.

Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in the figures, the number of the layer is appended as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown next, and in the equations at the bottom of the figure.



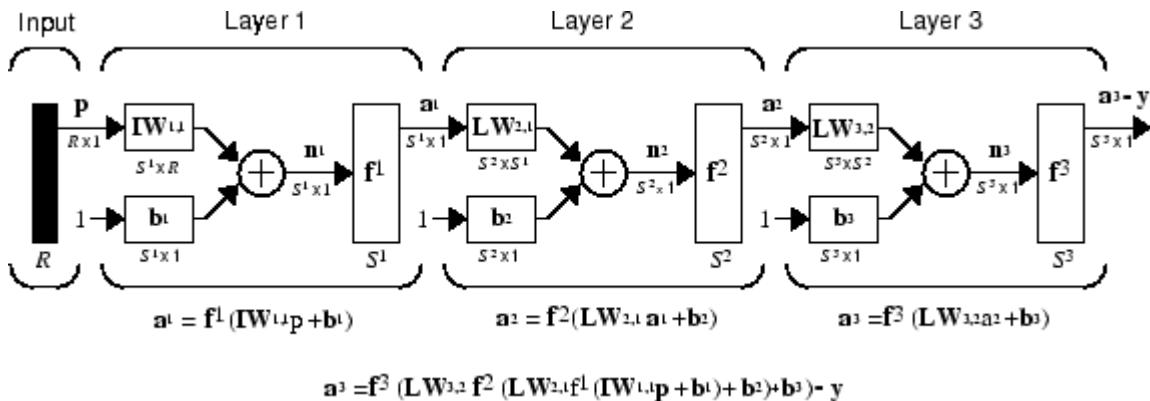
The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an $S^2 \times S^1$ weight matrix \mathbf{W}^2 . The input to layer 2 is \mathbf{a}^1 ; the output is \mathbf{a}^2 . Now that all the vectors and matrices of layer 2 have been identified, it can be treated as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. This toolbox does not use that designation.

The architecture of a multilayer network with a single input vector can be specified with the notation $R - S^1 - S^2 - \dots - S^M$, where the number of elements of the input vector and the number of neurons in each layer are specified.

The same three-layer network can also be drawn using abbreviated notation.



Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

Here it is assumed that the output of the third layer, a^3 , is the network output of interest, and this output is labeled as y . This notation is used to specify the output of multilayer networks.

Input and Output Processing Functions

Network inputs might have associated processing functions. Processing functions transform user input data to a form that is easier or more efficient for a network.

For instance, `mapminmax` transforms input data so that all values fall into the interval $[-1, 1]$. This can speed up learning for many networks. `removeconstantrows` removes the rows of the input vector that correspond to input elements that always have the same value, because these input elements are not providing any useful information to the network. The third common processing function is `fixunknowns`, which recodes unknown data (represented in the user's data with NaN values) into a numerical form for the network. `fixunknowns` preserves information about which values are known and which are unknown.

Similarly, network outputs can also have associated processing functions. Output processing functions are used to transform user-provided target vectors for network use.

Then, network outputs are reverse-processed using the same functions to produce output data with the same characteristics as the original user-provided targets.

Both `mapminmax` and `removeconstantrows` are often associated with network outputs. However, `fixunknowns` is not. Unknown values in targets (represented by `NaN` values) do not need to be altered for network use.

Processing functions are described in more detail in “Choose Neural Network Input-Output Processing Functions” on page 5-9.

See Also

More About

- “Neuron Model” on page 4-5
- “Workflow for Neural Network Design” on page 4-2

Create Neural Network Object

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 4-2.

The easiest way to create a neural network is to use one of the network creation functions. To investigate how this is done, you can create a simple, two-layer feedforward network, using the command `feedforwardnet`:

```
net = feedforwardnet  
net =  
  
Neural Network  
  
    name: 'Feed-Forward Neural Network'  
    userdata: (your custom info)  
  
dimensions:  
  
    numInputs: 1  
    numLayers: 2  
    numOutputs: 1  
    numInputDelays: 0  
    numLayerDelays: 0  
    numFeedbackDelays: 0  
    numWeightElements: 10  
    sampleTime: 1  
  
connections:  
  
    biasConnect: [1; 1]  
    inputConnect: [1; 0]  
    layerConnect: [0 0; 1 0]  
    outputConnect: [0 1]  
  
subobjects:  
  
    inputs: {1x1 cell array of 1 input}  
    layers: {2x1 cell array of 2 layers}  
    outputs: {1x2 cell array of 1 output}  
    biases: {2x1 cell array of 2 biases}  
    inputWeights: {2x1 cell array of 1 weight}  
    layerWeights: {2x2 cell array of 1 weight}
```

functions:

```
adaptFcn: 'adaptwb'  
adaptParam: (none)  
derivFcn: 'defaultderiv'  
divideFcn: 'dividerand'  
divideParam: .trainRatio, .valRatio, .testRatio  
divideMode: 'sample'  
initFcn: 'initlay'  
performFcn: 'mse'  
performParam: .regularization, .normalization  
plotFcns: {'plotperform', plottrainstate, ploterrhist,  
plotregression}  
plotParams: {1x4 cell array of 4 params}  
trainFcn: 'trainlm'  
trainParam: .showWindow, .showCommandLine, .show, .epochs,  
.time, .goal, .min_grad, .max_fail, .mu, .mu_dec,  
.mu_inc, .mu_max
```

weight and bias values:

```
IW: {2x1 cell} containing 1 input weight matrix  
LW: {2x2 cell} containing 1 layer weight matrix  
b: {2x1 cell} containing 2 bias vectors
```

methods:

```
adapt: Learn while in continuous use  
configure: Configure inputs & outputs  
gensim: Generate Simulink model  
init: Initialize weights & biases  
perform: Calculate performance  
sim: Evaluate network outputs given inputs  
train: Train network with examples  
view: View diagram  
unconfigure: Unconfigure inputs & outputs  
  
evaluate: outputs = net(inputs)
```

This display is an overview of the network object, which is used to store all of the information that defines a neural network. There is a lot of detail here, but there are a few key sections that can help you to see how the network object is organized.

The dimensions section stores the overall structure of the network. Here you can see that there is one input to the network (although the one input can be a vector containing many elements), one network output, and two layers.

The connections section stores the connections between components of the network. For example, there is a bias connected to each layer, the input is connected to layer 1, and the output comes from layer 2. You can also see that layer 1 is connected to layer 2. (The rows of `net.layerConnect` represent the destination layer, and the columns represent the source layer. A one in this matrix indicates a connection, and a zero indicates no connection. For this example, there is a single one in element 2,1 of the matrix.)

The key subobjects of the network object are `inputs`, `layers`, `outputs`, `biases`, `inputWeights`, and `layerWeights`. View the `layers` subobject for the first layer with the command

```
net.layers{1}
```

```
Neural Network Layer
```

```
    name: 'Hidden'
    dimensions: 10
    distanceFcn: (none)
    distanceParam: (none)
    distances: []
        initFcn: 'initnw'
    netInputFcn: 'netsum'
    netInputParam: (none)
    positions: []
        range: [10x2 double]
        size: 10
    topologyFcn: (none)
    transferFcn: 'tansig'
    transferParam: (none)
    userdata: (your custom info)
```

The number of neurons in a layer is given by its `size` property. In this case, the layer has 10 neurons, which is the default size for the `feedforwardnet` command. The net input function is `netsum` (summation) and the transfer function is the `tansig`. If you wanted to change the transfer function to `logsig`, for example, you could execute the command:

```
net.layers{1}.transferFcn = 'logsig';
```

To view the `layerWeights` subobject for the weight between layer 1 and layer 2, use the command:

```
net.layerWeights{2,1}
```

Neural Network Weight

```
    delays: 0
    initFcn: (none)
    initConfig: .inputSize
        learn: true
        learnFcn: 'learngdm'
    learnParam: .lr, .mc
        size: [0 10]
    weightFcn: 'dotprod'
    weightParam: (none)
        userdata: (your custom info)
```

The weight function is `dotprod`, which represents standard matrix multiplication (dot product). Note that the size of this layer weight is 0-by-10. The reason that we have zero rows is because the network has not yet been configured for a particular data set. The number of output neurons is equal to the number of rows in your target vector. During the configuration process, you will provide the network with example inputs and targets, and then the number of output neurons can be assigned.

This gives you some idea of how the network object is organized. For many applications, you will not need to be concerned about making changes directly to the network object, since that is taken care of by the network creation functions. It is usually only when you want to override the system defaults that it is necessary to access the network object directly. Other topics will show how this is done for particular networks and training methods.

To investigate the network object in more detail, you might find that the object listings, such as the one shown above, contain links to help on each subobject. Click the links, and you can selectively investigate those parts of the object that are of interest to you.

Configure Neural Network Inputs and Outputs

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 4-2.

After a neural network has been created, it must be configured. The configuration step consists of examining input and target data, setting the network's input and output sizes to match the data, and choosing settings for processing inputs and outputs that will enable best network performance. The configuration step is normally done automatically, when the training function is called. However, it can be done manually, by using the configuration function. For example, to configure the network you created previously to approximate a sine function, issue the following commands:

```
p = -2:.1:2;
t = sin(pi*p/2);
net1 = configure(net,p,t);
```

You have provided the network with an example set of inputs and targets (desired network outputs). With this information, the `configure` function can set the network input and output sizes to match the data.

After the configuration, if you look again at the weight between layer 1 and layer 2, you can see that the dimension of the weight is 1 by 20. This is because the target for this network is a scalar.

```
net1.layerWeights{2,1}
```

```
Neural Network Weight

    delays: 0
    initFcn: (none)
initConfig: .inputSize
    learn: true
    learnFcn: 'learngdm'
    learnParam: .lr, .mc
    size: [1 10]
    weightFcn: 'dotprod'
    weightParam: (none)
    userdata: (your custom info)
```

In addition to setting the appropriate dimensions for the weights, the configuration step also defines the settings for the processing of inputs and outputs. The input processing can be located in the `inputs` subobject:

```
net1.inputs{1}

    Neural Network Input

        name: 'Input'
        feedbackOutput: []
        processFcns: {'removeconstantrows', mapminmax}
        processParams: {1x2 cell array of 2 params}
        processSettings: {1x2 cell array of 2 settings}
        processedRange: [1x2 double]
        processedSize: 1
        range: [1x2 double]
        size: 1
        userdata: (your custom info)
```

Before the input is applied to the network, it will be processed by two functions: `removeconstantrows` and `mapminmax`. These are discussed fully in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2 so we won’t address the particulars here. These processing functions may have some processing parameters, which are contained in the subobject `net1.inputs{1}.processParam`. These have default values that you can override. The processing functions can also have configuration settings that are dependent on the sample data. These are contained in `net1.inputs{1}.processSettings` and are set during the configuration process. For example, the `mapminmax` processing function normalizes the data so that all inputs fall in the range $[-1, 1]$. Its configuration settings include the minimum and maximum values in the sample data, which it needs to perform the correct normalization. This will be discussed in much more depth in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

As a general rule, we use the term “parameter,” as in process parameters, training parameters, etc., to denote constants that have default values that are assigned by the software when the network is created (and which you can override). We use the term “configuration setting,” as in process configuration setting, to denote constants that are assigned by the software from an analysis of sample data. These settings do not have default values, and should not generally be overridden.

For more information, see also “Understanding Deep Learning Toolbox Data Structures” on page 4-23.

Understanding Deep Learning Toolbox Data Structures

In this section...

["Simulation with Concurrent Inputs in a Static Network" on page 4-23](#)

["Simulation with Sequential Inputs in a Dynamic Network" on page 4-24](#)

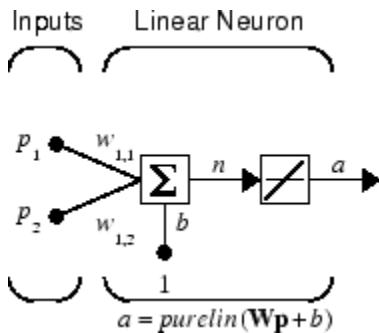
["Simulation with Concurrent Inputs in a Dynamic Network" on page 4-26](#)

This topic discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks. The following section describes how the format of the data structures affects network training.

There are two basic types of input vectors: those that occur *concurrently* (at the same time, or in no particular time sequence), and those that occur *sequentially* in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, you could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

Simulation with Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, you need not be concerned about whether or not the input vectors occur in a particular time sequence, so you can treat the inputs as concurrent. In addition, the problem is made even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this linear feedforward network, use the following commands:

```
net = linearlayer;
net.inputs{1}.size = 2;
net.layers{1}.dimensions = 1;
```

For simplicity, assign the weight matrix and bias to be $\mathbf{W} = [1 \ 2]$ and $b = [0]$.

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

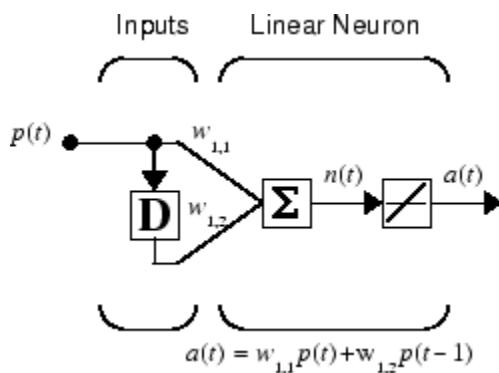
You can now simulate the network:

```
A = net(P)
A =
    5      4      8      5
```

A single matrix of concurrent vectors is presented to the network, and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important, because they do not interact with each other.

Simulation with Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, the next figure shows a simple network that contains one delay.



The following commands create this network:

```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
```

Assign the weight matrix to be $\mathbf{W} = [1 \ 2]$.

The command is:

```
net.IW{1,1} = [1 2];
```

Suppose that the input sequence is:

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

You can now simulate the network:

```
A = net(P)
A =
    [1]      [4]      [7]      [10]
```

You input a cell array containing a sequence of inputs, and the network produces a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying

the current input by 1 and the preceding input by 2 and summing the result. If you were to change the order of the inputs, the numbers obtained in the output would change.

Simulation with Concurrent Inputs in a Dynamic Network

If you were to apply the same inputs as a set of concurrent inputs instead of a sequence of inputs, you would obtain a completely different response. (However, it is not clear why you would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, “Simulation with Sequential Inputs in a Dynamic Network” on page 4-24, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When you simulate with concurrent inputs, you obtain

```
A = net(P)
A =
    1      2      3      4
```

The result is the same as if you had concurrently applied each one of the inputs to a separate network and computed one output. Note that because you did not assign any initial conditions to the network delays, they were assumed to be 0. For this case the output is simply 1 times the input, because the weight that multiplies the current input is 1.

In certain special cases, you might want to simulate the network response to several different sequences at the same time. In this case, you would want to present the network with a concurrent set of sequences. For example, suppose you wanted to present the following two sequences to the network:

$$\mathbf{p}_1(1) = [1], \mathbf{p}_1(2) = [2], \mathbf{p}_1(3) = [3], \mathbf{p}_1(4) = [4]$$

$$\mathbf{p}_2(1) = [4], \mathbf{p}_2(2) = [3], \mathbf{p}_2(3) = [2], \mathbf{p}_2(4) = [1]$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

```
P = {[1 4] [2 3] [3 2] [4 1]};
```

You can now simulate the network:

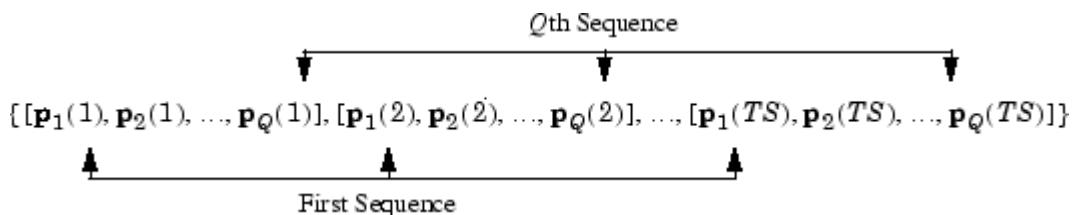
```
A = net(P);
```

The resulting network output would be

```
A = {[1 4] [4 11] [7 8] [10 5]}
```

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the network input P when there are Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



In this topic, you apply sequential and concurrent inputs to dynamic networks. In “Simulation with Concurrent Inputs in a Static Network” on page 4-23, you applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It does not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in “Neural Network Training Concepts” on page 4-28.

See also “Configure Neural Network Inputs and Outputs” on page 4-21.

Neural Network Training Concepts

In this section...

["Incremental Training with adapt" on page 4-28](#)

["Batch Training" on page 4-30](#)

["Training Feedback" on page 4-33](#)

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 4-2.

This topic describes two different styles of training. In *incremental* training the weights and biases of the network are updated each time an input is presented to the network. In *batch* training the weights and biases are only updated after all the inputs are presented. The batch training methods are generally more efficient in the MATLAB environment, and they are emphasized in the Deep Learning Toolbox software, but there are some applications where incremental training can be useful, so that paradigm is implemented as well.

Incremental Training with adapt

Incremental training can be applied to both static and dynamic networks, although it is more commonly used with dynamic networks, such as adaptive filters. This section illustrates how incremental training is performed on both static and dynamic networks.

Incremental Training of Static Networks

Consider again the static network used for the first example. You want to train it incrementally, so that the weights and biases are updated after each input is presented. In this case you use the function `adapt`, and the inputs and targets are presented as sequences.

Suppose you want to train the network to create the linear function:

$$t = 2p_1 + p_2$$

Then for the previous inputs,

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

the targets would be

$$\mathbf{t}_1 = [4], \mathbf{t}_2 = [5], \mathbf{t}_3 = [7], \mathbf{t}_4 = [7]$$

For incremental training, you present the inputs and targets as sequences:

```
P = {[1;2] [2;1] [2;3] [3;1]};
T = {4 5 7 7};
```

First, set up the network with zero initial weights and biases. Also, set the initial learning rate to zero to show the effect of incremental training.

```
net = linearlayer(0,0);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Recall from “Simulation with Concurrent Inputs in a Static Network” on page 4-23 that, for a static network, the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. However, this is not true when training the network. When you use the `adapt` function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As shown in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

You are now ready to train the network incrementally.

```
[net,a,e,pe] = adapt(net,P,T);
```

The network outputs remain zero, because the learning rate is zero, and the weights are not updated. The errors are equal to the targets:

```
a = [0]      [0]      [0]      [0]
e = [4]      [5]      [7]      [7]
```

If you now set the learning rate to 0.1 you can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1,1}.learnParam.lr = 0.1;
[net,a,e,pe] = adapt(net,P,T);
a = [0]      [2]      [6]      [5.8]
e = [4]      [3]      [1]      [1.2]
```

The first output is the same as it was with zero learning rate, because no update is made until the first input is presented. The second output is different, because the weights have

been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error is eventually driven to zero.

Incremental Training with Dynamic Networks

You can also train dynamic networks incrementally. In fact, this would be the most common situation.

To train the network incrementally, present the inputs and targets as elements of cell arrays. Here are the initial input P_i and the inputs P and targets T as elements of cell arrays.

```
Pi = {1};  
P = {2 3 4};  
T = {3 5 7};
```

Take the linear network with one delay at the input, as used in a previous example. Initialize the weights to zero and set the learning rate to 0.1.

```
net = linearlayer([0 1],0.1);  
net = configure(net,P,T);  
net.IW{1,1} = [0 0];  
net.biasConnect = 0;
```

You want to train the network to create the current output by summing the current and the previous inputs. This is the same input sequence you used in the previous example with the exception that you assign the first term in the sequence as the initial condition for the delay. You can now sequentially train the network using `adapt`.

```
[net,a,e,pe] = adapt(net,P,T,Pi);  
a = [0] [2.4] [7.98]  
e = [3] [2.6] [-0.98]
```

The first output is zero, because the weights have not yet been updated. The weights change at each subsequent time step.

Batch Training

Batch training, in which weights and biases are only updated after all the inputs and targets are presented, can be applied to both static and dynamic networks. Both types of networks are discussed in this section.

Batch Training with Static Networks

Batch training can be done using either `adapt` or `train`, although `train` is generally the best option, because it typically has access to more efficient training algorithms.

Incremental training is usually done with `adapt`; batch training is usually done with `train`.

For batch training of a static network with `adapt`, the input vectors must be placed in one matrix of concurrent vectors.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

Begin with the static network used in previous examples. The learning rate is set to 0.01.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

When you call `adapt`, it invokes `trains` (the default adaption function for the linear network) and `learnwh` (the default learning function for the weights and biases). `trains` uses Widrow-Hoff learning.

```
[net,a,e,pe] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all the training set has been presented. If you display the weights, you find

```
net.IW{1,1}
ans = 0.4900 0.4100
net.b{1}
ans =
0.2300
```

This is different from the result after one pass of `adapt` with incremental updating.

Now perform the same batch training using `train`. Because the Widrow-Hoff rule can be used in incremental or batch mode, it can be invoked by `adapt` or `train`. (There are several algorithms that can only be used in batch mode (e.g., Levenberg-Marquardt), so these algorithms can only be invoked by `train`.)

For this case, the input vectors can be in a matrix of concurrent vectors or in a cell array of sequential vectors. Because the network is static and because `train` always operates in batch mode, `train` converts any cell array of sequential vectors to a matrix of concurrent vectors. Concurrent mode operation is used whenever possible because it has a more efficient implementation in MATLAB code:

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

The network is set up in the same way.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Now you are ready to train the network. Train it for only one epoch, because you used only one pass of `adapt`. The default training function for the linear network is `trainb`, and the default learning function for the weights and biases is `learnwh`, so you should get the same results obtained using `adapt` in the previous example, where the default adaption function was `trains`.

```
net.trainParam.epochs = 1;
net = train(net,P,T);
```

If you display the weights after one epoch of training, you find

```
net.IW{1,1}
ans = 0.4900 0.4100
net.b{1}
ans =
0.2300
```

This is the same result as the batch mode training in `adapt`. With static networks, the `adapt` function can implement incremental or batch training, depending on the format of the input data. If the data is presented as a matrix of concurrent vectors, batch training occurs. If the data is presented as a sequence, incremental training occurs. This is not true for `train`, which always performs batch training, regardless of the format of the input.

Batch Training with Dynamic Networks

Training static networks is relatively straightforward. If you use `train` the network is trained in batch mode and the inputs are converted to concurrent vectors (columns of a

matrix), even if they are originally passed as a sequence (elements of a cell array). If you use `adapt`, the format of the input determines the method of training. If the inputs are passed as a sequence, then the network is trained in incremental mode. If the inputs are passed as concurrent vectors, then batch mode training is used.

With dynamic networks, batch mode training is typically done with `train` only, especially if only one training sequence exists. To illustrate this, consider again the linear network with a delay. Use a learning rate of 0.02 for the training. (When using a gradient descent algorithm, you typically use a smaller learning rate for batch mode training than incremental training, because all the individual gradients are summed before determining the step change to the weights.)

```
net = linearlayer([0 1],0.02);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
Pi = {1};
P = {2 3 4};
T = {3 5 6};
```

You want to train the network with the same sequence used for the incremental training earlier, but this time you want to update the weights only after all the inputs are applied (batch mode). The network is simulated in sequential mode, because the input is a sequence, but the weights are updated in batch mode.

```
net = train(net,P,T,Pi);
```

The weights after one epoch of training are

```
net.IW{1,1}
ans = 0.9000    0.6200
```

These are different weights than you would obtain using incremental training, where the weights would be updated three times during one pass through the training set. For batch training the weights are only updated once in each epoch.

Training Feedback

The `showWindow` parameter allows you to specify whether a training window is visible when you train. The training window appears by default. Two other parameters,

`showCommandLine` and `show`, determine whether command-line output is generated and the number of epochs between command-line feedback during training. For instance, this code turns off the training window and gives you training status information every 35 epochs when the network is later trained with `train`:

```
net.trainParam.showWindow = false;  
net.trainParam.showCommandLine = true;  
net.trainParam.show= 35;
```

Sometimes it is convenient to disable all training displays. To do that, turn off both the training window and command-line feedback:

```
net.trainParam.showWindow = false;  
net.trainParam.showCommandLine = false;
```

The training window appears automatically when you train. Use the `nntraintool` function to manually open and close the training window.

```
nntraintool  
nntraintool('close')
```

Multilayer Shallow Neural Networks and Backpropagation Training

- “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2
- “Multilayer Shallow Neural Network Architecture” on page 5-4
- “Prepare Data for Multilayer Shallow Neural Networks” on page 5-8
- “Choose Neural Network Input-Output Processing Functions” on page 5-9
- “Divide Data for Optimal Neural Network Training” on page 5-12
- “Create, Configure, and Initialize Multilayer Shallow Neural Networks” on page 5-14
- “Train and Apply Multilayer Shallow Neural Networks” on page 5-17
- “Analyze Shallow Neural Network Performance After Training” on page 5-24
- “Limitations and Cautions” on page 5-30

Multilayer Shallow Neural Networks and Backpropagation Training

The shallow multilayer feedforward neural network can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems, as discussed in “Design Time Series Time-Delay Neural Networks” on page 6-14. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network.

Note The training functions described in this topic are not limited to multilayer networks. They can be used to train arbitrary architectures (even custom networks), as long as their components are differentiable.

The work flow for the general neural network design process has seven primary steps:

- 1** Collect data
- 2** Create the network
- 3** Configure the network
- 4** Initialize the weights and biases
- 5** Train the network
- 6** Validate the network (post-training analysis)
- 7** Use the network

Step 1 might happen outside the framework of Deep Learning Toolbox software, but this step is critical to the success of the design process.

Details of this workflow are discussed in these sections:

- “Multilayer Shallow Neural Network Architecture” on page 5-4
- “Prepare Data for Multilayer Shallow Neural Networks” on page 5-8
- “Create, Configure, and Initialize Multilayer Shallow Neural Networks” on page 5-14
- “Train and Apply Multilayer Shallow Neural Networks” on page 5-17
- “Analyze Shallow Neural Network Performance After Training” on page 5-24
- “Use the Network” on page 5-22

- “Limitations and Cautions” on page 5-30

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 5-9
- “Divide Data for Optimal Neural Network Training” on page 5-12
- “Neural Networks with Parallel and GPU Computing” on page 11-2

For time series, dynamic modeling, and prediction, see this section:

- “How Dynamic Neural Networks Work” on page 6-3

Multilayer Shallow Neural Network Architecture

In this section...

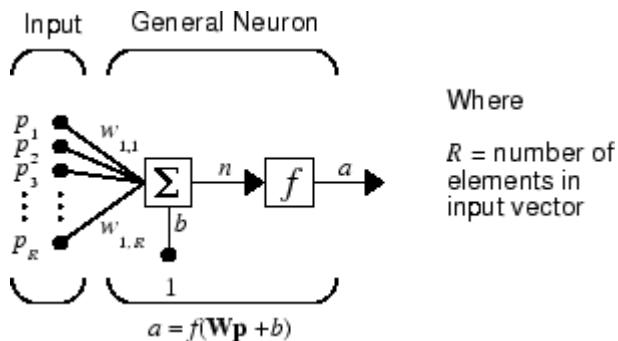
[“Neuron Model \(logsig, tansig, purelin\)” on page 5-4](#)

[“Feedforward Neural Network” on page 5-5](#)

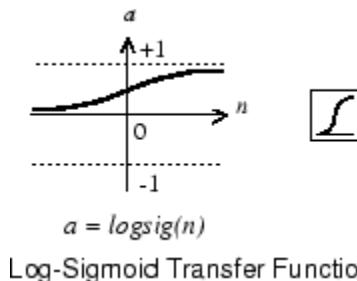
This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

Neuron Model (logsig, tansig, purelin)

An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w . The sum of the weighted inputs and the bias forms the input to the transfer function f . Neurons can use any differentiable transfer function f to generate their output.

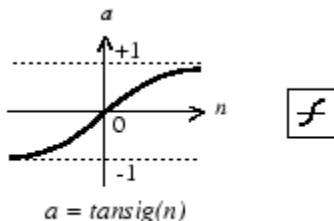


Multilayer networks often use the log-sigmoid transfer function `logsig`.



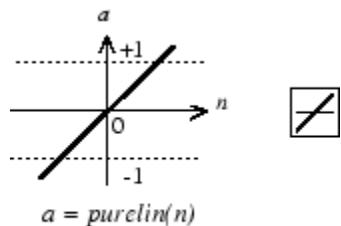
The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks can use the tan-sigmoid transfer function `tansig`.



Tan-Sigmoid Transfer Function

Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function `purelin` is shown below.

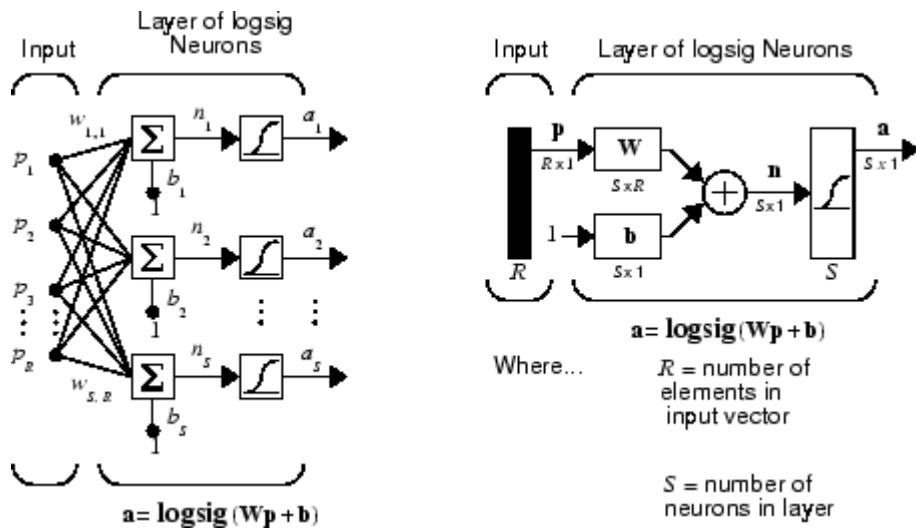


Linear Transfer Function

The three transfer functions described here are the most commonly used transfer functions for multilayer networks, but other differentiable transfer functions can be created and used if desired.

Feedforward Neural Network

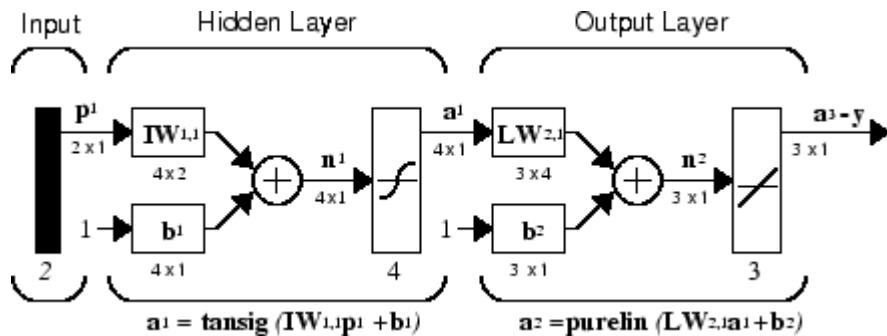
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as `logsig`). This is the case when the network is used for pattern recognition problems (in which a decision is being made by the network).

For multiple-layer networks the layer number determines the superscript on the weight matrix. The appropriate notation is used in the two-layer `tansig/purelin` network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer.

Now that the architecture of the multilayer network has been defined, the design process is described in the following sections.

Prepare Data for Multilayer Shallow Neural Networks

Tip To learn how to prepare image data for deep learning networks, see “Preprocess Images for Deep Learning” on page 1-201.

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

Before beginning the network design process, you first collect and prepare sample data. It is generally difficult to incorporate prior knowledge into a neural network, therefore the network can only be as accurate as the data that are used to train the network.

It is important that the data cover the range of inputs for which the network will be used. Multilayer networks can be trained to generalize well within the range of inputs for which they have been trained. However, they do not have the ability to accurately extrapolate beyond this range, so it is important that the training data span the full range of the input space.

After the data have been collected, there are two steps that need to be performed before the data are used to train the network: the data need to be preprocessed, and they need to be divided into subsets.

Choose Neural Network Input-Output Processing Functions

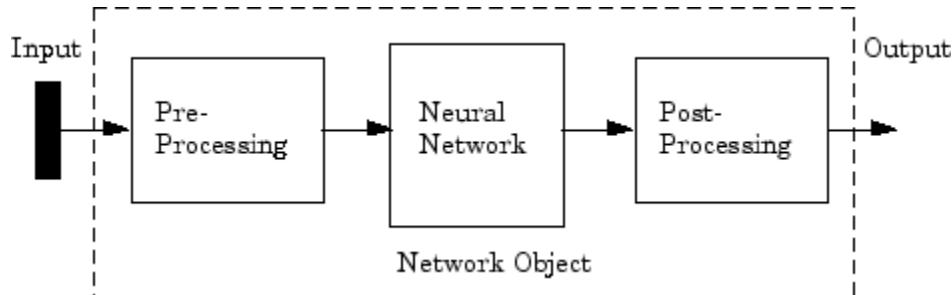
This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

Neural network training can be more efficient if you perform certain preprocessing steps on the network inputs and targets. This section describes several preprocessing routines that you can use. (The most common of these are provided automatically when you create a network, and they become part of the network object, so that whenever the network is used, the data coming into the network is preprocessed in the same way.)

For example, in multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three ($\exp(-3) \approx 0.05$). If this happens at the beginning of the training process, the gradients will be very small, and the network training will be very slow. In the first layer of the network, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be very small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network.

Generally, the normalization step is applied to both the input vectors and the target vectors in the data set. In this way, the network output always falls into a normalized range. The network output can then be reverse transformed back into the units of the original target data when the network is put to use in the field.

It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and the output, as shown in the following figure.



Most of the network creation functions in the toolbox, including the multilayer network creation functions, such as `feedforwardnet`, automatically assign processing functions to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for network training.

You can override the default input and output processing functions by adjusting network properties after you create the network.

To see a cell array list of processing functions assigned to the input of a network, access this property:

```
net.inputs{1}.processFcns
```

where the index 1 refers to the first input vector. (There is only one input vector for the feedforward network.) To view the processing functions returned by the output of a two-layer network, access this network property:

```
net.outputs{2}.processFcns
```

where the index 2 refers to the output vector coming from the second layer. (For the feedforward network, there is only one output vector, and it comes from the final layer.) You can use these properties to change the processing functions that you want your network to apply to the inputs and outputs. However, the defaults usually provide excellent performance.

Several processing functions have parameters that customize their operation. You can access or change the parameters of the i^{th} input processing function for the network input as follows:

```
net.inputs{1}.processParams{i}
```

You can access or change the parameters of the i^{th} output processing function for the network output associated with the second layer, as follows:

```
net.outputs{2}.processParams{i}
```

For multilayer network creation functions, such as `feedforwardnet`, the default input processing functions are `removeconstantrows` and `mapminmax`. For outputs, the default processing functions are also `removeconstantrows` and `mapminmax`.

The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

Function	Algorithm
<code>mapminmax</code>	Normalize inputs/targets to fall in the range $[-1, 1]$
<code>mapstd</code>	Normalize inputs/targets to have zero mean and unity variance
<code>processpca</code>	Extract principal components from the input vector
<code>fixunknowns</code>	Process unknown inputs
<code>removeconstantrows</code>	Remove inputs/targets that are constant

Representing Unknown or Don't-Care Targets

Unknown or “don’t care” targets can be represented with `NaN` values. We do not want unknown target values to have an impact on training, but if a network has several outputs, some elements of any target vector may be known while others are unknown. One solution would be to remove the partially unknown target vector and its associated input vector from the training set, but that involves the loss of the good target values. A better solution is to represent those unknown targets with `NaN` values. All the performance functions of the toolbox will ignore those targets for purposes of calculating performance and derivatives of performance.

Divide Data for Optimal Neural Network Training

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

When training multilayer networks, the general practice is to first divide the data into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error. This technique is discussed in more detail in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. The data division is normally performed automatically when you train the network.

Function	Algorithm
<code>dividerand</code>	Divide the data randomly (default)
<code>divideblock</code>	Divide the data into contiguous blocks
<code>divideint</code>	Divide the data using an interleaved selection
<code>divideind</code>	Divide the data by index

You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of the division functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

The divide function is accessed automatically whenever the network is trained, and is used to divide the data into training, validation and testing subsets. If `net.divideFcn` is set to '`'dividerand'`' (the default), then the data is randomly divided into the three subsets using the division parameters `net.divideParam.trainRatio`, `net.divideParam.valRatio`, and `net.divideParam.testRatio`. The fraction of data that is placed in the training set is `trainRatio/(trainRatio+valRatio +testRatio)`, with a similar formula for the other two sets. The default ratios for training, testing and validation are 0.7, 0.15 and 0.15, respectively.

If `net.divideFcn` is set to '`'divideblock'`', then the data is divided into three subsets using three contiguous blocks of the original data set (training taking the first block, validation the second and testing the third). The fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

If `net.divideFcn` is set to '`'divideint'`', then the data is divided by an interleaved method, as in dealing a deck of cards. It is done so that different percentages of data go into the three subsets. The fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

When `net.divideFcn` is set to '`'divideind'`', the data is divided by index. The indices for the three subsets are defined by the division parameters `net.divideParam.trainInd`, `net.divideParam.valInd` and `net.divideParam.testInd`. The default assignment for these indices is the null array, so you must set the indices when using this option.

Create, Configure, and Initialize Multilayer Shallow Neural Networks

In this section...

[“Other Related Architectures” on page 5-15](#)

[“Initializing Weights \(init\)” on page 5-15](#)

This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

After the data has been collected, the next step in training a network is to create the network object. The function `feedforwardnet` creates a multilayer feedforward network. If this function is invoked with no input arguments, then a default network object is created that has not been configured. The resulting network can then be configured with the `configure` command.

As an example, the file `bodyfat_dataset.mat` contains a predefined set of input and target vectors. The input vectors define data regarding physical attributes of people and the target values define percentage body fat of the people. Load the data using the following command:

```
load bodyfat_dataset
```

Loading this file creates two variables. The input matrix `bodyfatInputs` consists of 252 column vectors of 13 physical attribute variables for 252 different people. The target matrix `bodyfatTargets` consists of the corresponding 252 body fat percentages.

The next step is to create the network. The following call to `feedforwardnet` creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.)

```
net = feedforwardnet;
net = configure(net, bodyfatInputs, bodyfatTargets);
```

Optional arguments can be provided to `feedforwardnet`. For instance, the first argument is an array containing the number of neurons in each hidden layer. (The default setting is 10, which means one hidden layer with 10 neurons. One hidden layer generally produces excellent results, but you may want to try two hidden layers, if the results with

one are not adequate. Increasing the number of neurons in the hidden layer increases the power of the network, but requires more computation and is more likely to produce overfitting.) The second argument contains the name of the training function to be used. If no arguments are supplied, the default number of layers is 2, the default number of neurons in the hidden layer is 10, and the default training function is `trainlm`. The default transfer function for hidden layers is `tansig` and the default for the output layer is `purelin`.

The `configure` command configures the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you might want to reinitialize the weights, or to perform a custom initialization. “Initializing Weights (`init`)” on page 5-15 explains the details of the initialization process. You can also skip the configuration step and go directly to training the network. The `train` command will automatically configure the network and initialize the weights.

Other Related Architectures

While two-layer feedforward networks can potentially learn virtually any input-output relationship, feedforward networks with more layers might learn complex relationships more quickly. For most problems, it is best to start with two layers, and then increase to three layers, if the performance with two layers is not satisfactory.

The function `cascadeforwardnet` creates cascade-forward networks. These are similar to feedforward networks, but include a weight connection from the input to each layer, and from each layer to the successive layers. For example, a three-layer network has connections from layer 1 to layer 2, layer 2 to layer 3, and layer 1 to layer 3. The three-layer network also has connections from the input to all three layers. The additional connections might improve the speed at which the network learns the desired relationship.

The function `patternnet` creates a network that is very similar to `feedforwardnet`, except that it uses the `tansig` transfer function in the last layer. This network is generally used for pattern recognition. Other networks can learn dynamic or time-series relationships.

Initializing Weights (`init`)

Before training a feedforward network, you must initialize the weights and biases. The `configure` command automatically initializes the weights, but you might want to reinitialize them. You do this with the `init` command. This function takes a network

object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized):

```
net = init(net);
```

Train and Apply Multilayer Shallow Neural Networks

In this section...

- “Training Algorithms” on page 5-18
- “Training Example” on page 5-20
- “Use the Network” on page 5-22

Tip To train a deep learning network, use `trainNetwork`.

This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

When the network weights and biases are initialized, the network is ready for training. The multilayer feedforward network can be trained for function approximation (nonlinear regression) or pattern recognition. The training process requires a set of examples of proper network behavior—network inputs p and target outputs t .

The process of training a neural network involves tuning the values of the weights and biases of the network to optimize network performance, as defined by the network performance function `net.performFcn`. The default performance function for feedforward networks is mean square error `mse`—the average squared error between the network outputs a and the target outputs t . It is defined as follows:

$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2$$

(Individual squared errors can also be weighted. See “Train Neural Networks with Error Weights” on page 6-44.) There are two different ways in which training can be implemented: incremental mode and batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs in the training set are applied to the network before the weights are updated. This topic describes batch mode training with the `train` command. Incremental training with the `adapt` command is discussed in “Incremental Training with `adapt`” on page 4-28. For most problems, when using the Deep Learning Toolbox software, batch training is significantly faster and produces smaller errors than incremental training.

For training multilayer feedforward networks, any standard numerical optimization algorithm can be used to optimize the performance function, but there are a few key ones that have shown excellent performance for neural network training. These optimization methods use either the gradient of the network performance with respect to the network weights, or the Jacobian of the network errors with respect to the weights.

The gradient and the Jacobian are calculated using a technique called the *backpropagation* algorithm, which involves performing computations backward through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapters 11 (for the gradient) and 12 (for the Jacobian) of [HDB96 on page 14-2].

Training Algorithms

As an illustration of how the training works, consider the simplest optimization algorithm — gradient descent. It updates the network weights and biases in the direction in which the performance function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where \mathbf{x}_k is a vector of current weights and biases, \mathbf{g}_k is the current gradient, and α_k is the learning rate. This equation is iterated until the network converges.

A list of the training algorithms that are available in the Deep Learning Toolbox software and that use gradient- or Jacobian-based methods, is shown in the following table.

For a detailed description of several of these techniques, see also Hagan, M.T., H.B. Demuth, and M.H. Beale, Neural Network Design, Boston, MA: PWS Publishing, 1996, Chapters 11 and 12.

Function	Algorithm
<code>trainlm</code>	Levenberg-Marquardt
<code>trainbr</code>	Bayesian Regularization
<code>trainbfg</code>	BFGS Quasi-Newton
<code>trainrp</code>	Resilient Backpropagation
<code>trainscg</code>	Scaled Conjugate Gradient
<code>traincgb</code>	Conjugate Gradient with Powell/Beale Restarts

Function	Algorithm
<code>traincgf</code>	Fletcher-Powell Conjugate Gradient
<code>traincgp</code>	Polak-Ribière Conjugate Gradient
<code>trainoss</code>	One Step Secant
<code>traingdx</code>	Variable Learning Rate Gradient Descent
<code>traingdm</code>	Gradient Descent with Momentum
<code>traingd</code>	Gradient Descent

The fastest training function is generally `trainlm`, and it is the default training function for `feedforwardnet`. The quasi-Newton method, `trainbfg`, is also quite fast. Both of these methods tend to be less efficient for large networks (with thousands of weights), since they require more memory and more computation time for these cases. Also, `trainlm` performs better on function fitting (nonlinear regression) problems than on pattern recognition problems.

When training large networks, and when training pattern recognition networks, `trainscg` and `trainrp` are good choices. Their memory requirements are relatively small, and yet they are much faster than standard gradient descent algorithms.

See “Choose a Multilayer Neural Network Training Function” on page 11-16 for a full comparison of the performances of the training algorithms shown in the table above.

As a note on terminology, the term “backpropagation” is sometimes used to refer specifically to the gradient descent algorithm, when applied to neural network training. That terminology is not used here, since the process of computing the gradient and Jacobian by performing calculations backward through the network is applied in all of the training functions listed above. It is clearer to use the name of the specific optimization algorithm that is being used, rather than to use the term backpropagation alone.

Also, the multilayer network is sometimes referred to as a backpropagation network. However, the backpropagation technique that is used to compute gradients and Jacobians in a multilayer network can also be applied to many different network architectures. In fact, the gradients and Jacobians for any network that has differentiable transfer functions, weight functions and net input functions can be computed using the Deep Learning Toolbox software through a backpropagation process. You can even create your own custom networks and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you.

Training Example

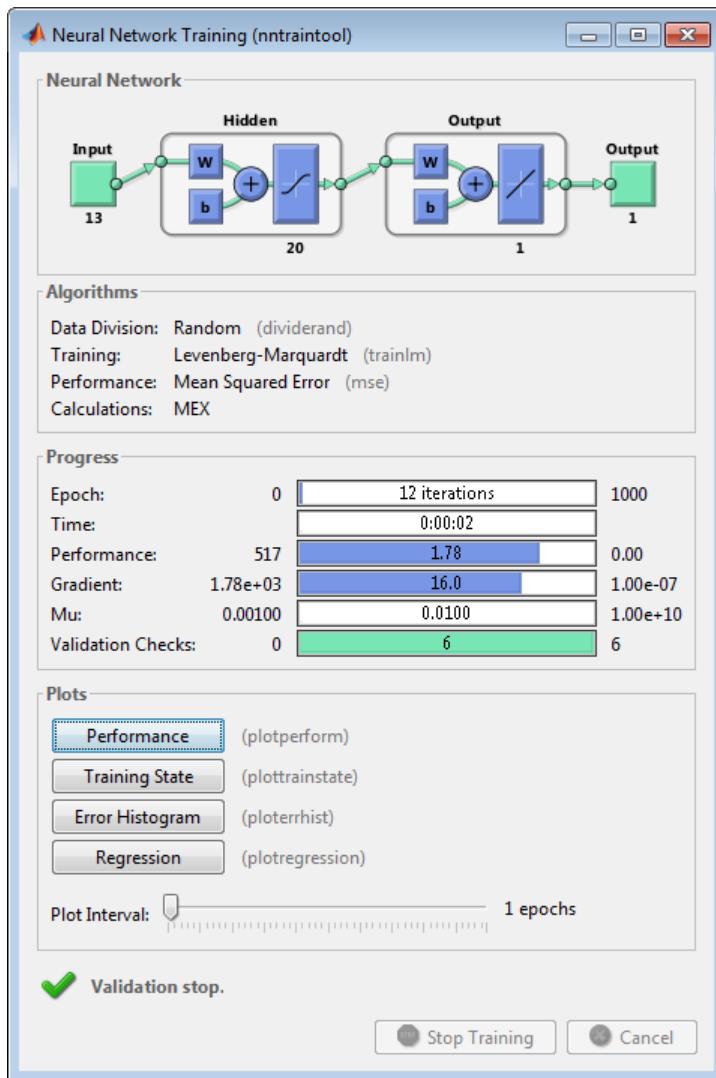
To illustrate the training process, execute the following commands:

```
load bodyfat_dataset  
net = feedforwardnet(20);  
[net,tr] = train(net,bodyfatInputs,bodyfatTargets);
```

Notice that you did not need to issue the `configure` command, because the configuration is done automatically by the `train` function. The training window will appear during training, as shown in the following figure. (If you do not want to have this window displayed during training, you can set the parameter `net.trainParam.showWindow` to `false`. If you want training information displayed in the command line, you can set the parameter `net.trainParam.showCommandLine` to `true`.)

This window shows that the data has been divided using the `dividerand` function, and the Levenberg-Marquardt (`trainlm`) training method has been used with the mean square error performance function. Recall that these are the default settings for `feedforwardnet`.

During training, the progress is constantly updated in the training window. Of most interest are the performance, the magnitude of the gradient of performance and the number of validation checks. The magnitude of the gradient and the number of validation checks are used to terminate the training. The gradient will become very small as the training reaches a minimum of the performance. If the magnitude of the gradient is less than `1e-5`, the training will stop. This limit can be adjusted by setting the parameter `net.trainParam.min_grad`. The number of validation checks represents the number of successive iterations that the validation performance fails to decrease. If this number reaches 6 (the default value), the training will stop. In this run, you can see that the training did stop because of the number of validation checks. You can change this criterion by setting the parameter `net.trainParam.max_fail`. (Note that your results may be different than those shown in the following figure, because of the random setting of the initial weights and biases.)



There are other criteria that can be used to stop network training. They are listed in the following table.

Parameter	Stopping Criteria
min_grad	Minimum Gradient Magnitude
max_fail	Maximum Number of Validation Increases
time	Maximum Training Time
goal	Minimum Performance Value
epochs	Maximum Number of Training Epochs (Iterations)

The training will also stop if you click the **Stop Training** button in the training window. You might want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by reissuing the `train` command shown above. It will continue to train the network from the completion of the previous run.

From the training window, you can access four plots: performance, training state, error histogram, and regression. The performance plot shows the value of the performance function versus the iteration number. It plots training, validation, and test performances. The training state plot shows the progress of other training variables, such as the gradient magnitude, the number of validation checks, etc. The error histogram plot shows the distribution of the network errors. The regression plot shows a regression between network outputs and network targets. You can use the histogram and regression plots to validate network performance, as is discussed in “Analyze Shallow Neural Network Performance After Training” on page 5-24.

Use the Network

After the network is trained and validated, the network object can be used to calculate the network response to any input. For example, if you want to find the network response to the fifth input vector in the building data set, you can use the following

```
a = net(bodyfatInputs(:,5))
```

```
a =
```

```
27.3740
```

If you try this command, your output might be different, depending on the state of your random number generator when the network was initialized. Below, the network object is called to calculate the outputs for a concurrent set of all the input vectors in the body fat data set. This is the batch mode form of simulation, in which all the input vectors are

placed in one matrix. This is much more efficient than presenting the vectors one at a time.

```
a = net(bodyfatInputs);
```

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

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32.

Analyze Shallow Neural Network Performance After Training

This topic presents part of a typical shallow neural network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2. To learn about how to monitor deep learning training progress, see “Monitor Deep Learning Training Progress”.

When the training in “Train and Apply Multilayer Shallow Neural Networks” on page 5-17 is complete, you can check the network performance and determine if any changes need to be made to the training process, the network architecture, or the data sets. First check the training record, `tr`, which was the second argument returned from the training function.

```
tr

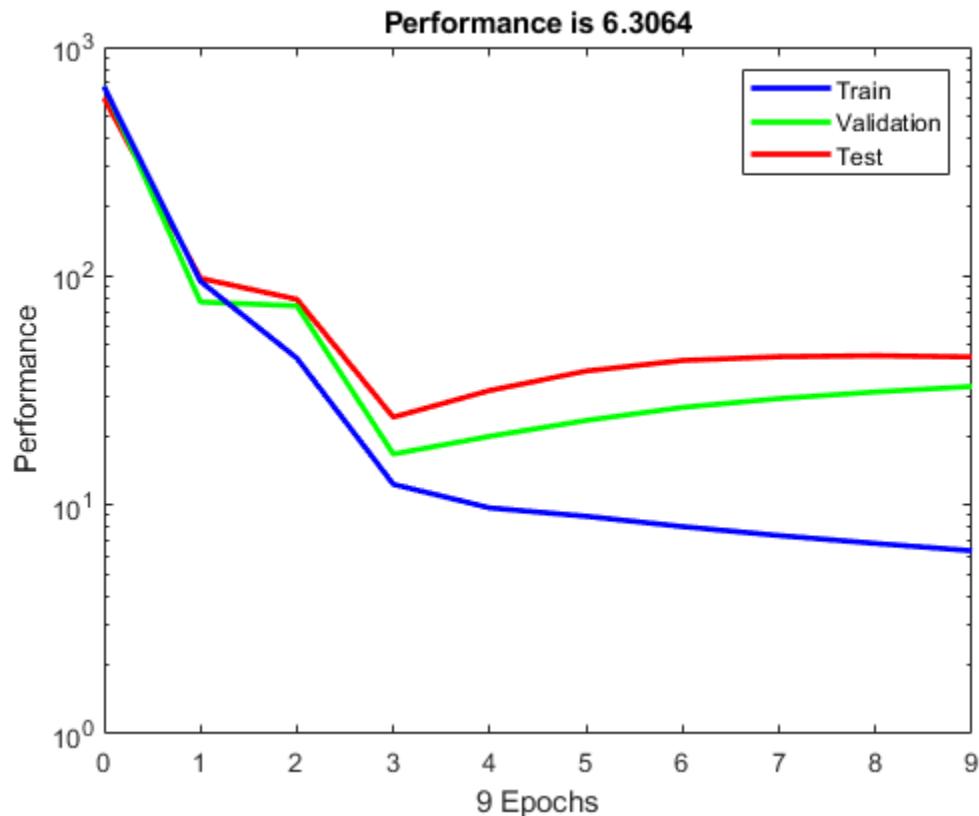
tr = struct with fields:
    trainFcn: 'trainlm'
    trainParam: [1x1 struct]
    performFcn: 'mse'
    performParam: [1x1 struct]
        derivFcn: 'defaultderiv'
        divideFcn: 'dividerand'
        divideMode: 'sample'
        divideParam: [1x1 struct]
            trainInd: [1x176 double]
            valInd: [1x38 double]
            testInd: [1x38 double]
            stop: 'Validation stop.'
    num_epochs: 9
    trainMask: {[1x252 double]}
    valMask: {[1x252 double]}
    testMask: {[1x252 double]}
    best_epoch: 3
        goal: 0
        states: {1x8 cell}
        epoch: [0 1 2 3 4 5 6 7 8 9]
        time: [1x10 double]
        perf: [1x10 double]
        vperf: [1x10 double]
        tperf: [1x10 double]
        mu: [1x10 double]
    gradient: [1x10 double]
```

```
val_fail: [0 0 0 0 1 2 3 4 5 6]
best_perf: 12.3078
best_vperf: 16.6857
best_tperf: 24.1796
```

This structure contains all of the information concerning the training of the network. For example, `tr.trainInd`, `tr.valInd` and `tr.testInd` contain the indices of the data points that were used in the training, validation and test sets, respectively. If you want to retrain the network using the same division of data, you can set `net.divideFcn` to '`'divideInd'`', `net.divideParam.trainInd` to `tr.trainInd`, `net.divideParam.valInd` to `tr.valInd`, `net.divideParam.testInd` to `tr.testInd`.

The `tr` structure also keeps track of several variables during the course of training, such as the value of the performance function, the magnitude of the gradient, etc. You can use the training record to plot the performance progress by using the `plotperf` command:

```
plotperf(tr)
```



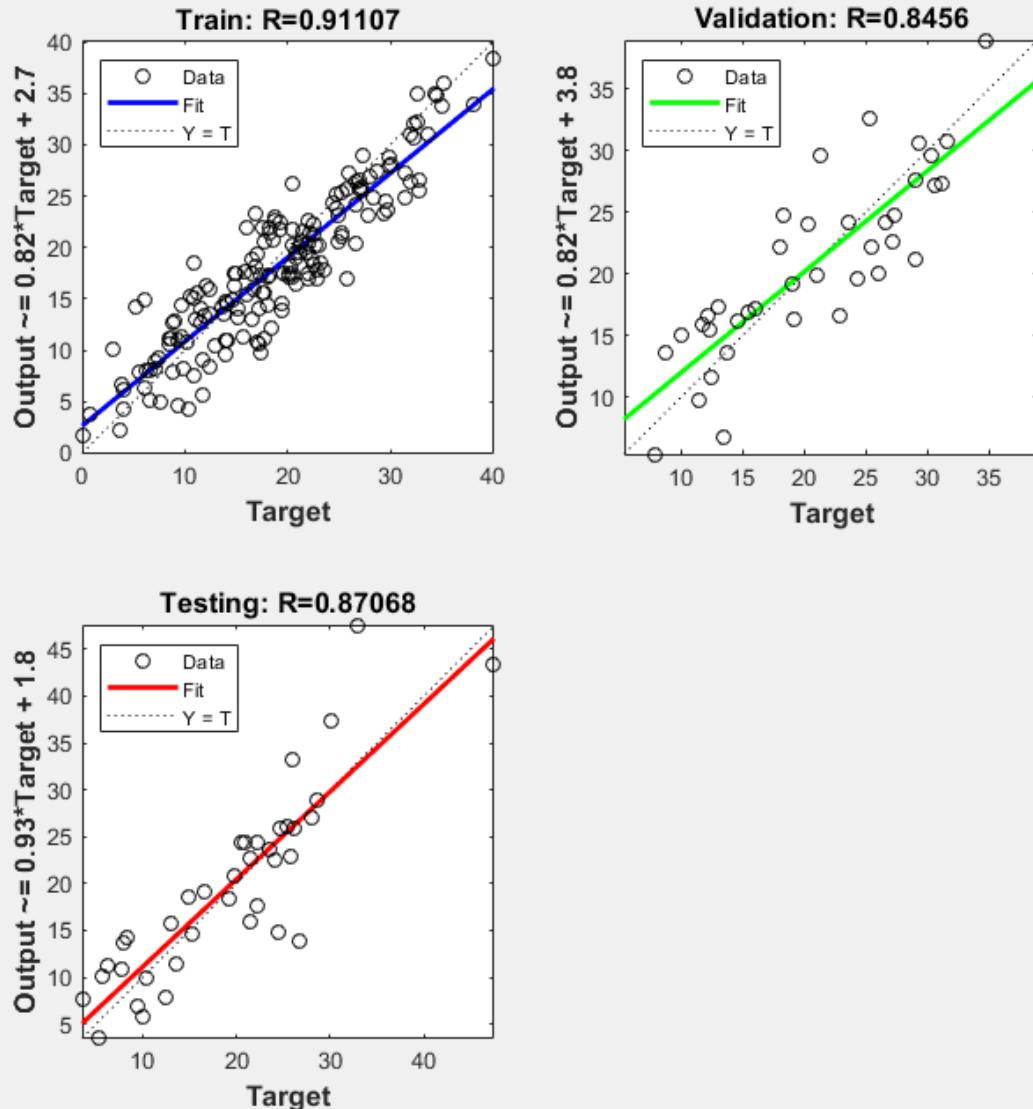
The property `tr.best_epoch` indicates the iteration at which the validation performance reached a minimum. The training continued for 6 more iterations before the training stopped.

This figure does not indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred.

The next step in validating the network is to create a regression plot, which shows the relationship between the outputs of the network and the targets. If the training were perfect, the network outputs and the targets would be exactly equal, but the relationship is rarely perfect in practice. For the body fat example, we can create a regression plot with the following commands. The first command calculates the trained network response

to all of the inputs in the data set. The following six commands extract the outputs and targets that belong to the training, validation and test subsets. The final command creates three regression plots for training, testing and validation.

```
bodyfatOutputs = net(bodyfatInputs);
trOut = bodyfatOutputs(tr.trainInd);
vOut = bodyfatOutputs(tr.valInd);
tsOut = bodyfatOutputs(tr.testInd);
trTarg = bodyfatTargets(tr.trainInd);
vTarg = bodyfatTargets(tr.valInd);
tsTarg = bodyfatTargets(tr.testInd);
plotregression(trTarg, trOut, 'Train', vTarg, vOut, 'Validation', tsTarg, tsOut, 'Test')
```



The three plots represent the training, validation, and testing data. The dashed line in each plot represents the perfect result – outputs = targets. The solid line represents the best fit linear regression line between outputs and targets. The R value is an indication of the relationship between the outputs and targets. If R = 1, this indicates that there is an exact linear relationship between outputs and targets. If R is close to zero, then there is no linear relationship between outputs and targets.

For this example, the training data indicates a good fit. The validation and test results also show large R values. The scatter plot is helpful in showing that certain data points have poor fits. For example, there is a data point in the test set whose network output is close to 35, while the corresponding target value is about 12. The next step would be to investigate this data point to determine if it represents extrapolation (i.e., is it outside of the training data set). If so, then it should be included in the training set, and additional data should be collected to be used in the test set.

Improving Results

If the network is not sufficiently accurate, you can try initializing the network and the training again. Each time you initialize a feedforward network, the network parameters are different and might produce different solutions.

```
net = init(net);
net = train(net, bodyfatInputs, bodyfatTargets);
```

As a second approach, you can increase the number of hidden neurons above 20. Larger numbers of neurons in the hidden layer give the network more flexibility because the network has more parameters it can optimize. (Increase the layer size gradually. If you make the hidden layer too large, you might cause the problem to be under-characterized and the network must optimize more parameters than there are data vectors to constrain these parameters.)

A third option is to try a different training function. Bayesian regularization training with `trainbr`, for example, can sometimes produce better generalization capability than using early stopping.

Finally, try using additional training data. Providing additional data for the network is more likely to produce a network that generalizes well to new data.

Limitations and Cautions

You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrnp`.

Multilayer networks are capable of performing just about any linear or nonlinear computation, and they can approximate any reasonable function arbitrarily well. However, while the network being trained might theoretically be capable of performing correctly, backpropagation and its variations might not always find a solution. See page 12-8 of [HDB96 on page 14-2] for a discussion of convergence to local minimum points.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity, see the figures on pages 12-5 to 12-7 of [HDB96 on page 14-2], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface, depending on the initial starting conditions, it is possible for the network solution to become trapped in one of these local minima. Settling in a local minimum can be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation does not always find the correct weights for the optimum solution. You might want to reinitialize the network and retrain several times to guarantee that you have the best solution.

Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fitted, but the fitting curve oscillates wildly between these points. Ways of dealing with various of these issues are discussed in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32. This topic is also discussed starting on page 11-21 of [HDB96 on page 14-2].

For more information about the workflow with multilayer networks, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.

Dynamic Neural Networks

- “Introduction to Dynamic Neural Networks” on page 6-2
- “How Dynamic Neural Networks Work” on page 6-3
- “Design Time Series Time-Delay Neural Networks” on page 6-14
- “Design Time Series Distributed Delay Neural Networks” on page 6-20
- “Design Time Series NARX Feedback Neural Networks” on page 6-23
- “Design Layer-Recurrent Neural Networks” on page 6-31
- “Create Reference Model Controller with MATLAB Script” on page 6-34
- “Multiple Sequences with Dynamic Neural Networks” on page 6-41
- “Neural Network Time-Series Utilities” on page 6-42
- “Train Neural Networks with Error Weights” on page 6-44
- “Normalize Errors of Multiple Outputs” on page 6-47
- “Multistep Neural Network Prediction” on page 6-52

Introduction to Dynamic Neural Networks

Neural networks can be classified into dynamic and static categories. Static (feedforward) networks have no feedback elements and contain no delays; the output is calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs, or states of the network.

Details of this workflow are discussed in the following sections:

- “Design Time Series Time-Delay Neural Networks” on page 6-14
- “Prepare Input and Layer Delay States” on page 6-18
- “Design Time Series Distributed Delay Neural Networks” on page 6-20
- “Design Time Series NARX Feedback Neural Networks” on page 6-23
- “Design Layer-Recurrent Neural Networks” on page 6-31

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 5-9
- “Divide Data for Optimal Neural Network Training” on page 5-12
- “Train Neural Networks with Error Weights” on page 6-44

How Dynamic Neural Networks Work

In this section...

["Feedforward and Recurrent Neural Networks" on page 6-3](#)

["Applications of Dynamic Networks" on page 6-10](#)

["Dynamic Network Structures" on page 6-10](#)

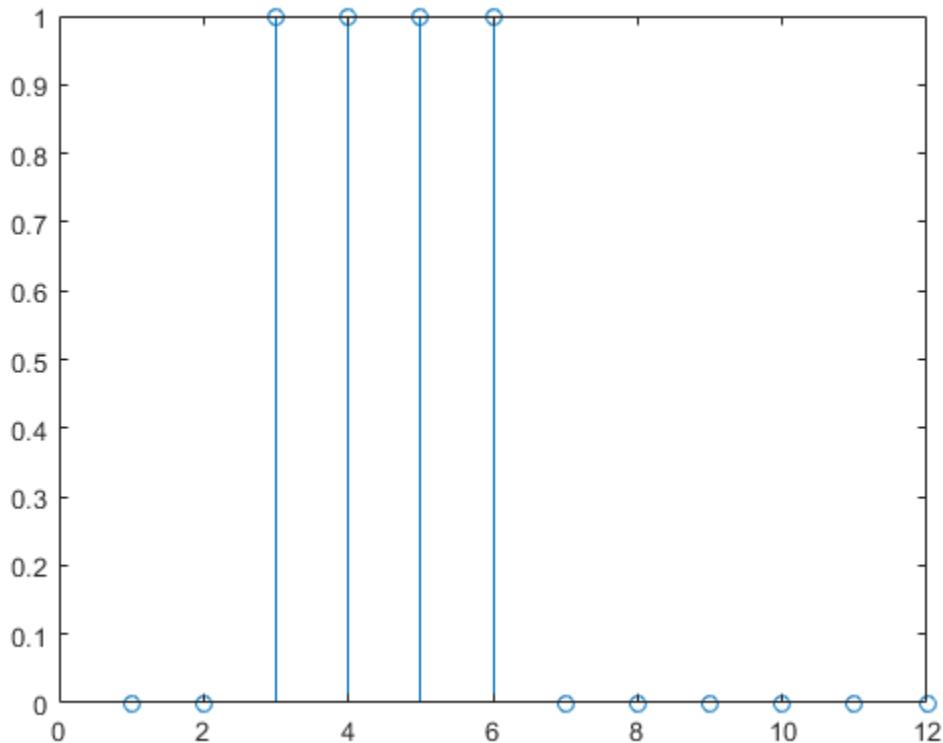
["Dynamic Network Training" on page 6-11](#)

Feedforward and Recurrent Neural Networks

Dynamic networks can be divided into two categories: those that have only feedforward connections, and those that have feedback, or recurrent, connections. To understand the differences between static, feedforward-dynamic, and recurrent-dynamic networks, create some networks and see how they respond to an input sequence. (First, you might want to review "Simulation with Sequential Inputs in a Dynamic Network" on page 4-24.)

The following commands create a pulse input sequence and plot it:

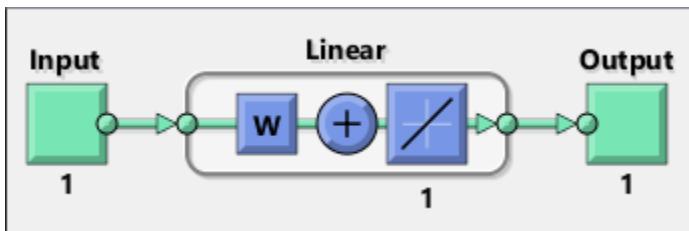
```
p = {0 0 1 1 1 0 0 0 0 0};  
stem(cell2mat(p))
```



Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

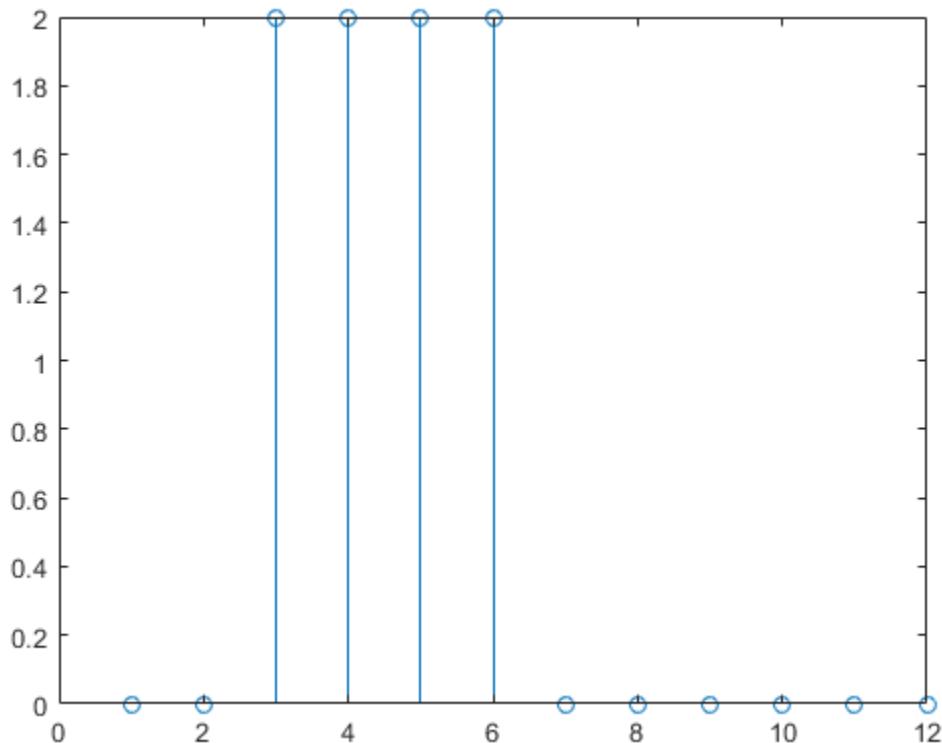
```
net = linearlayer;
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = 2;

view(net)
```



You can now simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```

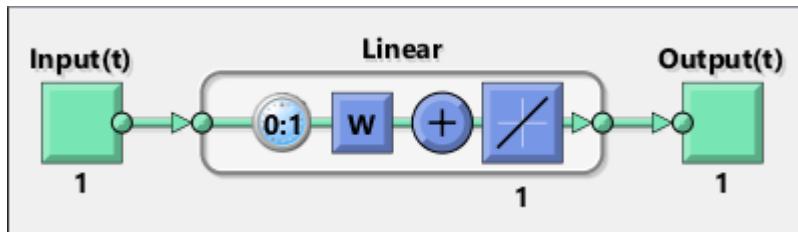


Note that the response of the static network lasts just as long as the input pulse. The response of the static network at any time point depends only on the value of the input sequence at that same time point.

Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used in “Simulation with Concurrent Inputs in a Dynamic Network” on page 4-26, which was a linear network with a tapped delay line on the input:

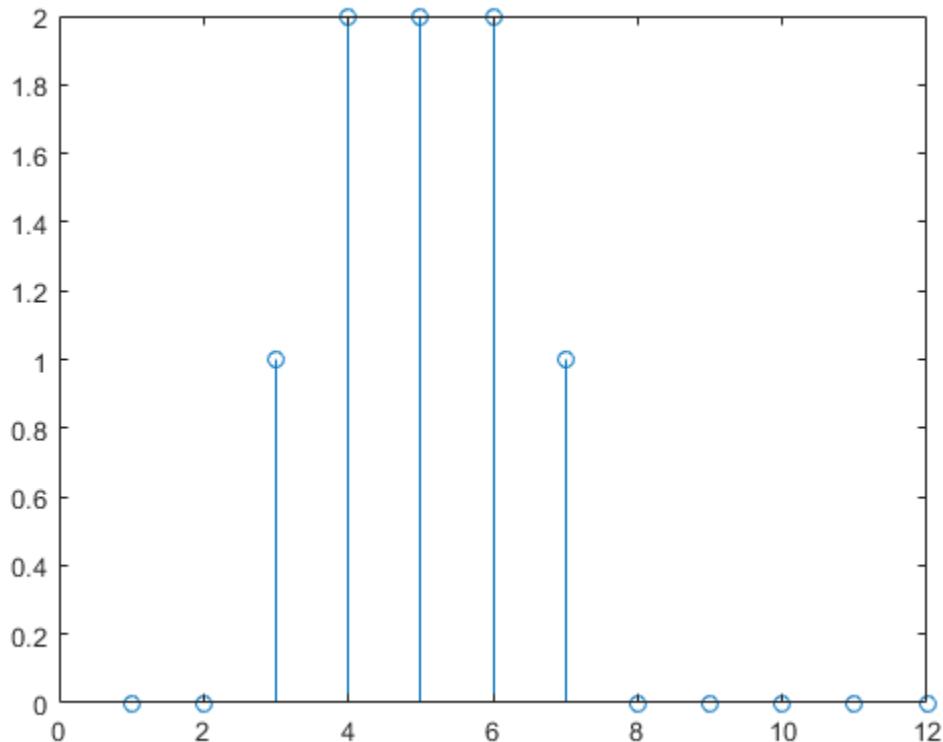
```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = [1 1];

view(net)
```



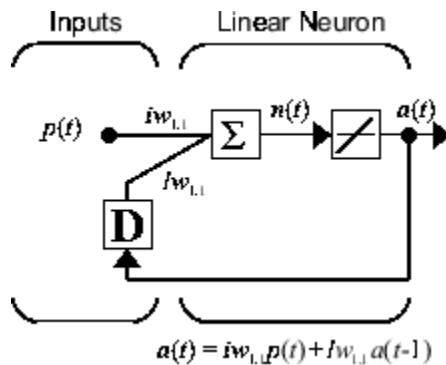
You can again simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```



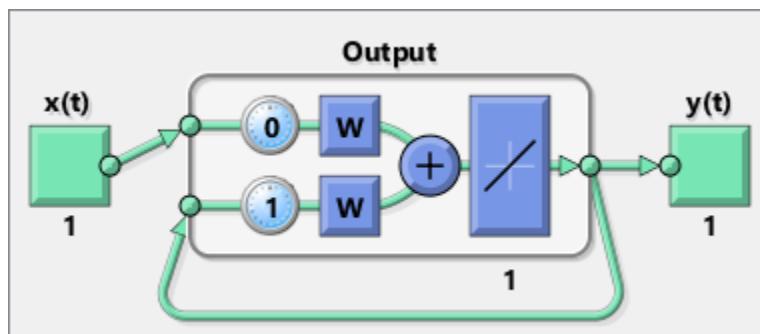
The response of the dynamic network lasts longer than the input pulse. The dynamic network has memory. Its response at any given time depends not only on the current input, but on the history of the input sequence. If the network does not have any feedback connections, then only a finite amount of history will affect the response. In this figure you can see that the response to the pulse lasts one time step beyond the pulse duration. That is because the tapped delay line on the input has a maximum delay of 1.

Now consider a simple recurrent-dynamic network, shown in the following figure.



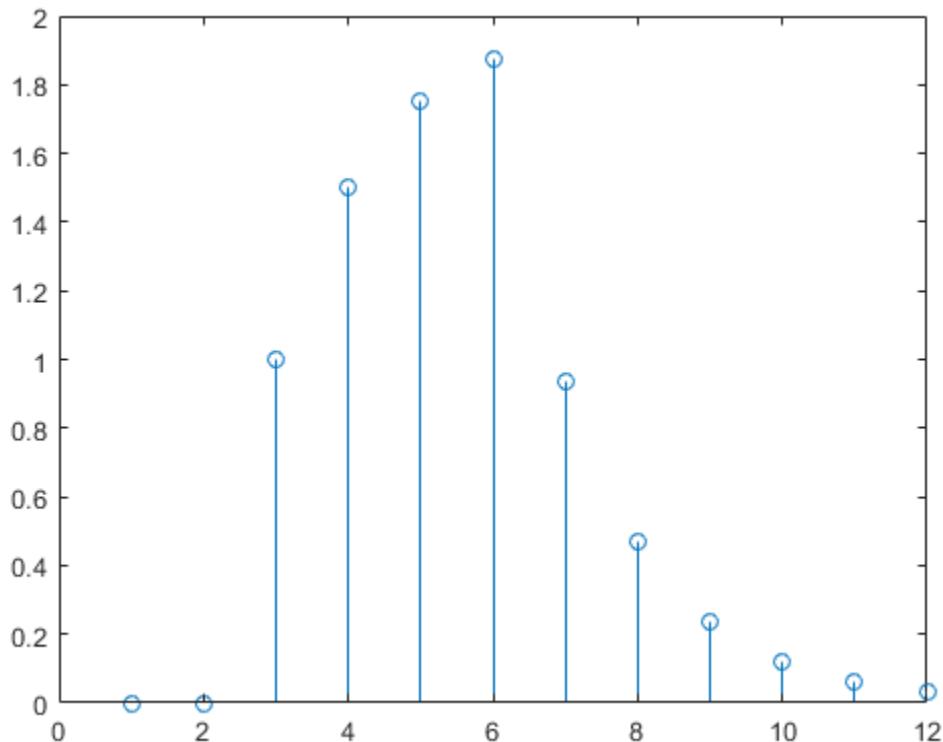
You can create the network, view it and simulate it with the following commands. The `narxnet` command is discussed in “Design Time Series NARX Feedback Neural Networks” on page 6-23.

```
net = narxnet(0,1,[],'closed');
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.LW{1} = .5;
net.IW{1} = 1;
view(net)
```



The following commands plot the network response.

```
a = net(p);
stem(cell2mat(a))
```



Notice that recurrent-dynamic networks typically have a longer response than feedforward-dynamic networks. For linear networks, feedforward-dynamic networks are called finite impulse response (FIR), because the response to an impulse input will become zero after a finite amount of time. Linear recurrent-dynamic networks are called infinite impulse response (IIR), because the response to an impulse can decay to zero (for a stable network), but it will never become exactly equal to zero. An impulse response for a nonlinear network cannot be defined, but the ideas of finite and infinite responses do carry over.

Applications of Dynamic Networks

Dynamic networks are generally more powerful than static networks (although somewhat more difficult to train). Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. This has applications in such disparate areas as prediction in financial markets [RoJa96 on page 14-2], channel equalization in communication systems [FeTs03 on page 14-2], phase detection in power systems [KaGr96 on page 14-2], sorting [JaRa04 on page 14-2], fault detection [ChDa99 on page 14-2], speech recognition [Robin94 on page 14-2], and even the prediction of protein structure in genetics [GiPr02 on page 14-2]. You can find a discussion of many more dynamic network applications in [MeJa00 on page 14-2].

One principal application of dynamic neural networks is in control systems. This application is discussed in detail in “Neural Network Control Systems”. Dynamic networks are also well suited for filtering. You will see the use of some linear dynamic networks for filtering in and some of those ideas are extended in this topic, using nonlinear dynamic networks.

Dynamic Network Structures

The Deep Learning Toolbox software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

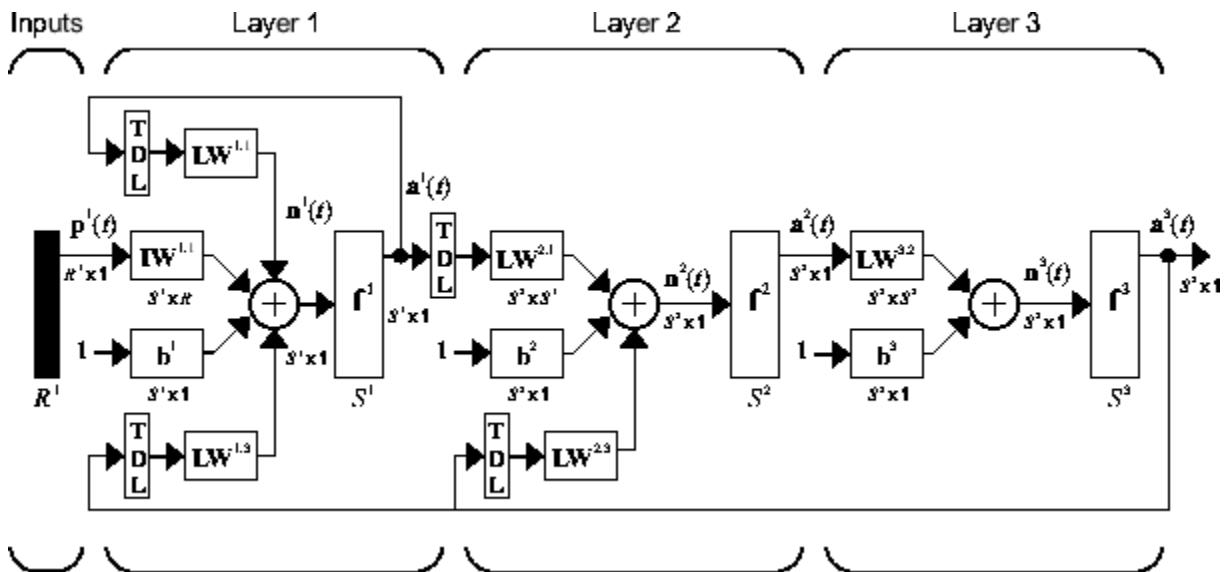
Each layer in the LDDN is made up of the following parts:

- Set of weight matrices that come into that layer (which can connect from other layers or from external inputs), associated weight function rule used to combine the weight matrix with its input (normally standard matrix multiplication, `dotprod`), and associated tapped delay line
- Bias vector
- Net input function rule that is used to combine the outputs of the various weight functions with the bias to produce the net input (normally a summing junction, `netprod`)
- Transfer function

The network has inputs that are connected to special weights, called input weights, and denoted by \mathbf{IW}^{ij} (`net.IW{i, j}` in the code), where j denotes the number of the input vector that enters the weight, and i denotes the number of the layer to which the weight

is connected. The weights connecting one layer to another are called layer weights and are denoted by $\text{LW}^{i,j}$ (`net.LW{i, j}` in the code), where j denotes the number of the layer coming into the weight and i denotes the number of the layer at the output of the weight.

The following figure is an example of a three-layer LDDN. The first layer has three weights associated with it: one input weight, a layer weight from layer 1, and a layer weight from layer 3. The two layer weights have tapped delay lines associated with them.



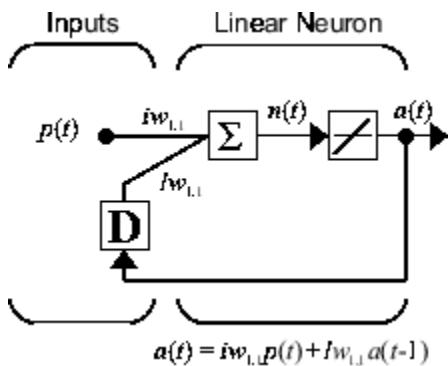
The Deep Learning Toolbox software can be used to train any LDDN, so long as the weight functions, net input functions, and transfer functions have derivatives. Most well-known dynamic network architectures can be represented in LDDN form. In the remainder of this topic you will see how to use some simple commands to create and train several very powerful dynamic networks. Other LDDN networks not covered in this topic can be created using the generic network command, as explained in “Define Shallow Neural Network Architectures”.

Dynamic Network Training

Dynamic networks are trained in the Deep Learning Toolbox software using the same gradient-based algorithms that were described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2. You can select from any of the training

functions that were presented in that topic. Examples are provided in the following sections.

Although dynamic networks can be trained using the same gradient-based algorithms that are used for static networks, the performance of the algorithms on dynamic networks can be quite different, and the gradient must be computed in a more complex way. Consider again the simple recurrent network shown in this figure.



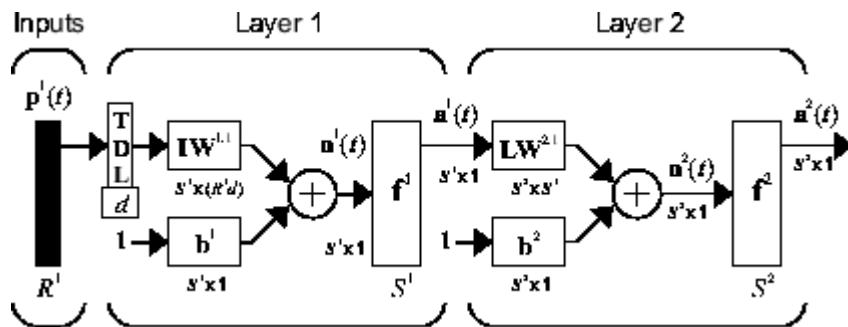
The weights have two different effects on the network output. The first is the direct effect, because a change in the weight causes an immediate change in the output at the current time step. (This first effect can be computed using standard backpropagation.) The second is an indirect effect, because some of the inputs to the layer, such as $a(t - 1)$, are also functions of the weights. To account for this indirect effect, you must use dynamic backpropagation to compute the gradients, which is more computationally intensive. (See [DeHa01a on page 14-2], [DeHa01b on page 14-2] and [DeHa07 on page 14-2].) Expect dynamic backpropagation to take more time to train, in part for this reason. In addition, the error surfaces for dynamic networks can be more complex than those for static networks. Training is more likely to be trapped in local minima. This suggests that you might need to train the network several times to achieve an optimal result. See [DHH01 on page 14-2] and [HDH09 on page 14-2] for some discussion on the training of dynamic networks.

The remaining sections of this topic show how to create, train, and apply certain dynamic networks to modeling, detection, and forecasting problems. Some of the networks require dynamic backpropagation for computing the gradients and others do not. As a user, you do not need to decide whether or not dynamic backpropagation is needed. This is determined automatically by the software, which also decides on the best form of dynamic

backpropagation to use. You just need to create the network and then invoke the standard **train** command.

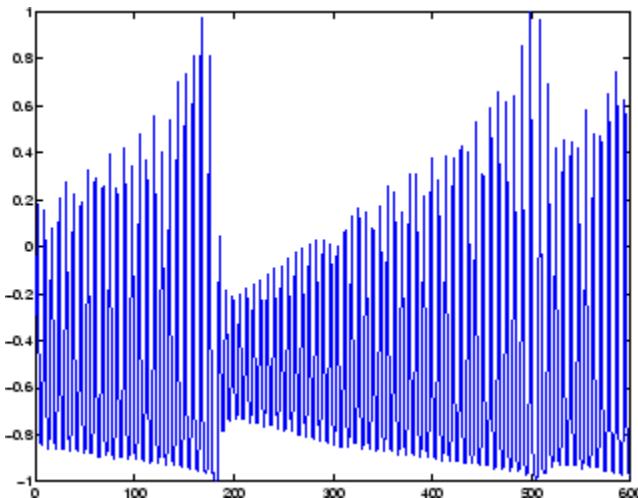
Design Time Series Time-Delay Neural Networks

Begin with the most straightforward dynamic network, which consists of a feedforward network with a tapped delay line at the input. This is called the focused time-delay neural network (FTDNN). This is part of a general class of dynamic networks, called focused networks, in which the dynamics appear only at the input layer of a static multilayer feedforward network. The following figure illustrates a two-layer FTDNN.



This network is well suited to time-series prediction. The following example shows the use of the FTDNN for predicting a classic time series.

The following figure is a plot of normalized intensity data recorded from a Far-Infrared-Laser in a chaotic state. This is a part of one of several sets of data used for the Santa Fe Time Series Competition [WeGe94 on page 14-2]. In the competition, the objective was to use the first 1000 points of the time series to predict the next 100 points. Because our objective is simply to illustrate how to use the FTDNN for prediction, the network is trained here to perform one-step-ahead predictions. (You can use the resulting network for multistep-ahead predictions by feeding the predictions back to the input of the network and continuing to iterate.)



The first step is to load the data, normalize it, and convert it to a time sequence (represented by a cell array):

```
y = laser_dataset;
y = y(1:600);
```

Now create the FTDNN network, using the **timedelaynet** command. This command is similar to the **feedforwardnet** command, with the additional input of the tapped delay line vector (the first input). For this example, use a tapped delay line with delays from 1 to 8, and use ten neurons in the hidden layer:

```
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = '';
```

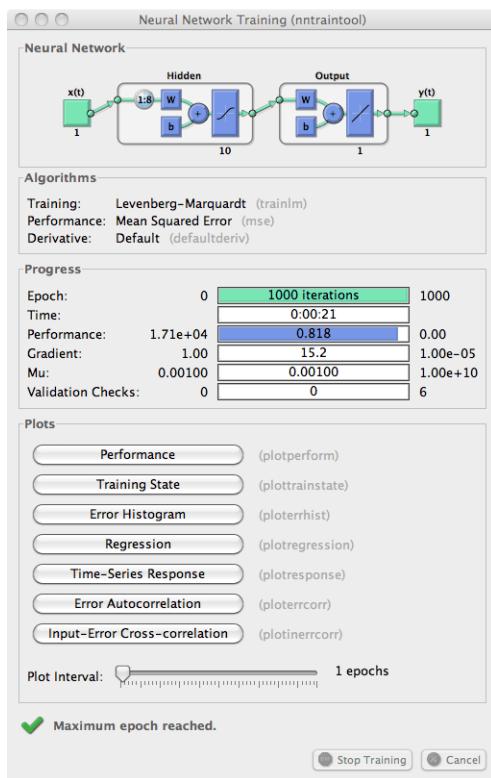
Arrange the network inputs and targets for training. Because the network has a tapped delay line with a maximum delay of 8, begin by predicting the ninth value of the time series. You also need to load the tapped delay line with the eight initial values of the time series (contained in the variable **Pi**):

```
p = y(9:end);
t = y(9:end);
Pi=y(1:8);
ftdnn_net = train(ftdnn_net,p,t,Pi);
```

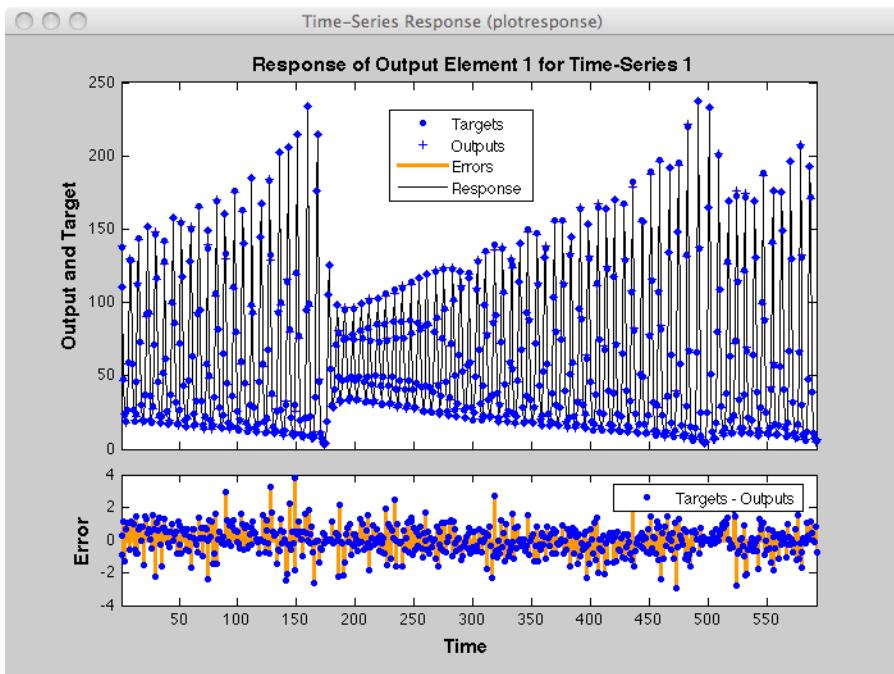
6 Dynamic Neural Networks

Notice that the input to the network is the same as the target. Because the network has a minimum delay of one time step, this means that you are performing a one-step-ahead prediction.

During training, the following training window appears.



Training stopped because the maximum epoch was reached. From this window, you can display the response of the network by clicking **Time-Series Response**. The following figure appears.



Now simulate the network and determine the prediction error.

```
yp = ftdnn_net(p,Pi);
e = gsubtract(yp,t);
rmse = sqrt(mse(e))

rmse =
    0.9740
```

(Note that `gsubtract` is a general subtraction function that can operate on cell arrays.) This result is much better than you could have obtained using a linear predictor. You can verify this with the following commands, which design a linear filter with the same tapped delay line input as the previous FTDNN.

```
lin_net = linearlayer([1:8]);
lin_net.trainFcn='trainlm';
[lin_net,tr] = train(lin_net,p,t,Pi);
lin_yp = lin_net(p,Pi);
lin_e = gsubtract(lin_yp,t);
lin_rmse = sqrt(mse(lin_e))
```

```
lin_rmse =  
21.1386
```

The `rms` error is 21.1386 for the linear predictor, but 0.9740 for the nonlinear FTDNN predictor.

One nice feature of the FTDNN is that it does not require dynamic backpropagation to compute the network gradient. This is because the tapped delay line appears only at the input of the network, and contains no feedback loops or adjustable parameters. For this reason, you will find that this network trains faster than other dynamic networks.

If you have an application for a dynamic network, try the linear network first (`linearlayer`) and then the FTDNN (`timedelaynet`). If neither network is satisfactory, try one of the more complex dynamic networks discussed in the remainder of this topic.

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

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32.

Prepare Input and Layer Delay States

You will notice in the last section that for dynamic networks there is a significant amount of data preparation that is required before training or simulating the network. This is because the tapped delay lines in the network need to be filled with initial conditions, which requires that part of the original data set be removed and shifted. There is a toolbox function that facilitates the data preparation for dynamic (time series) networks - `preprets`. For example, the following lines:

```
p = y(9:end);  
t = y(9:end);  
Pi = y(1:8);
```

can be replaced with

```
[p,Pi,Ai,t] = preprets(ftdnn_net,y,y);
```

The **prepares** function uses the network object to determine how to fill the tapped delay lines with initial conditions, and how to shift the data to create the correct inputs and targets to use in training or simulating the network. The general form for invoking **prepares** is

```
[X,Xi,Ai,T,EW,shift] = prepares(net,inputs,targets,feedback,EW)
```

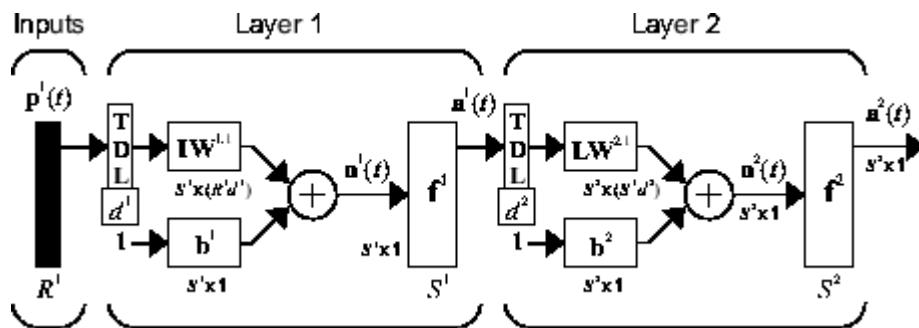
The input arguments for **prepares** are the network object (**net**), the external (non-feedback) input to the network (**inputs**), the non-feedback target (**targets**), the feedback target (**feedback**), and the error weights (**EW**) (see “Train Neural Networks with Error Weights” on page 6-44). The difference between external and feedback signals will become clearer when the NARX network is described in “Design Time Series NARX Feedback Neural Networks” on page 6-23. For the FTDNN network, there is no feedback signal.

The return arguments for **prepares** are the time shift between network inputs and outputs (**shift**), the network input for training and simulation (**X**), the initial inputs (**Xi**) for loading the tapped delay lines for input weights, the initial layer outputs (**Ai**) for loading the tapped delay lines for layer weights, the training targets (**T**), and the error weights (**EW**).

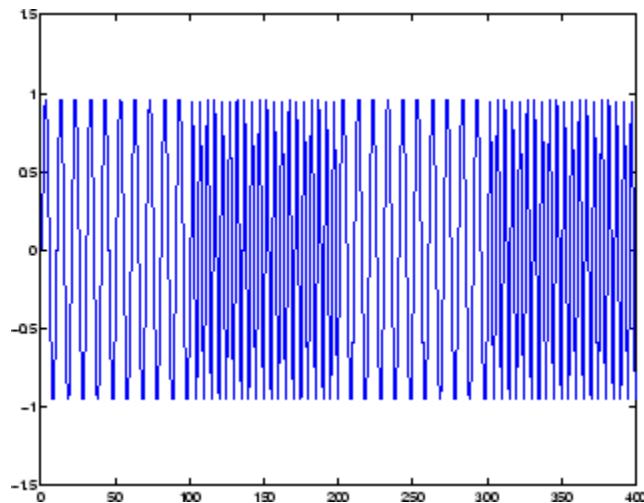
Using **prepares** eliminates the need to manually shift inputs and targets and load tapped delay lines. This is especially useful for more complex networks.

Design Time Series Distributed Delay Neural Networks

The FTDNN had the tapped delay line memory only at the input to the first layer of the static feedforward network. You can also distribute the tapped delay lines throughout the network. The distributed TDNN was first introduced in [WaHa89 on page 14-2] for phoneme recognition. The original architecture was very specialized for that particular problem. The following figure shows a general two-layer distributed TDNN.



This network can be used for a simplified problem that is similar to phoneme recognition. The network will attempt to recognize the frequency content of an input signal. The following figure shows a signal in which one of two frequencies is present at any given time.

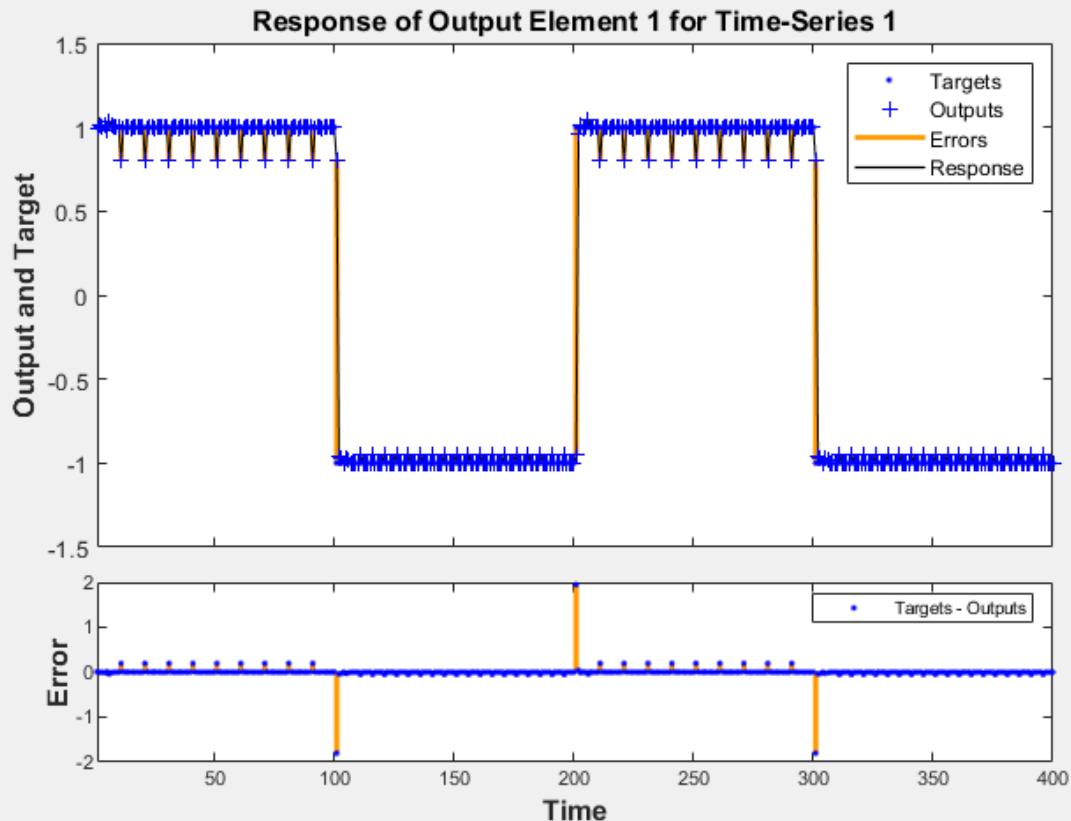


The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
time = 0:99;
y1 = sin(2*pi*time/10);
y2 = sin(2*pi*time/5);
y = [y1 y2 y1 y2];
t1 = ones(1,100);
t2 = -ones(1,100);
t = [t1 t2 t1 t2];
```

Now create the distributed TDNN network with the `distdelaynet` function. The only difference between the `distdelaynet` function and the `timedelaynet` function is that the first input argument is a cell array that contains the tapped delays to be used in each layer. In the next example, delays of zero to four are used in layer 1 and zero to three are used in layer 2. (To add some variety, the training function `trainbr` is used in this example instead of the default, which is `trainlm`. You can use any training function discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.)

```
d1 = 0:4;
d2 = 0:3;
p = con2seq(y);
t = con2seq(t);
dtdnn_net = distdelaynet({d1,d2},5);
dtdnn_net.trainFcn = 'trainbr';
dtdnn_net.divideFcn = '';
dtdnn_net.trainParam.epochs = 100;
dtdnn_net = train(dtdnn_net,p,t);
yp = sim(dtdnn_net,p);
plotresponse(t,yp)
```



The network is able to accurately distinguish the two “phonemes.”

You will notice that the training is generally slower for the distributed TDNN network than for the FTDNN. This is because the distributed TDNN must use dynamic backpropagation.

Design Time Series NARX Feedback Neural Networks

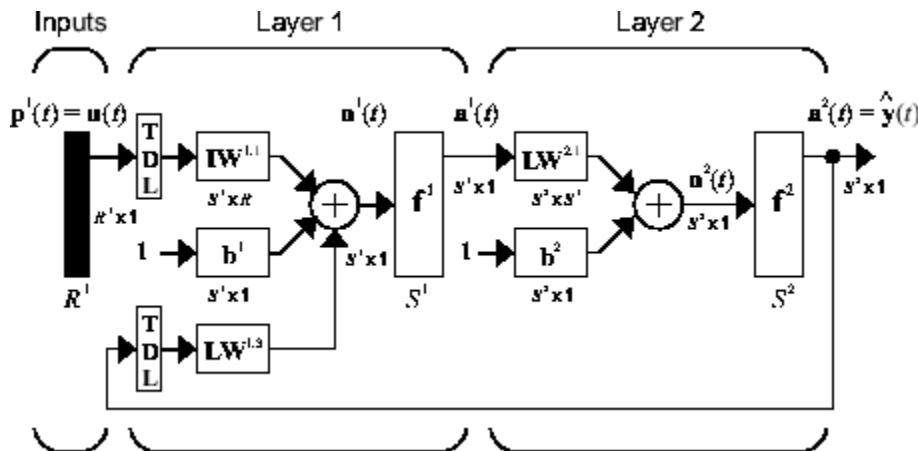
To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 6-52.

All the specific dynamic networks discussed so far have either been focused networks, with the dynamics only at the input layer, or feedforward networks. The nonlinear autoregressive network with exogenous inputs (NARX) is a recurrent dynamic network, with feedback connections enclosing several layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time-series modeling.

The defining equation for the NARX model is

$$y(t) = f(y(t - 1), y(t - 2), \dots, y(t - n_y), u(t - 1), u(t - 2), \dots, u(t - n_u))$$

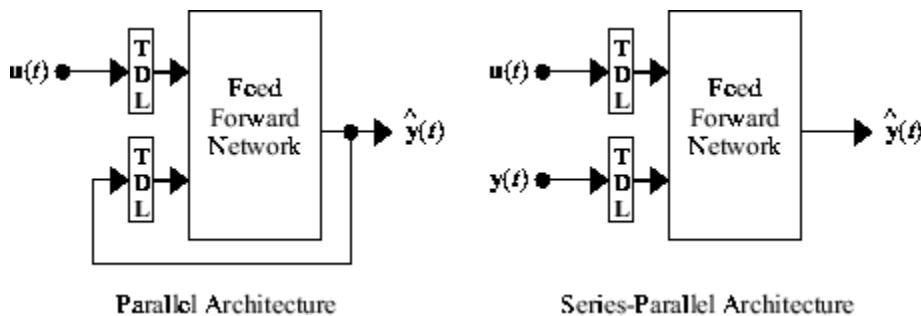
where the next value of the dependent output signal $y(t)$ is regressed on previous values of the output signal and previous values of an independent (exogenous) input signal. You can implement the NARX model by using a feedforward neural network to approximate the function f . A diagram of the resulting network is shown below, where a two-layer feedforward network is used for the approximation. This implementation also allows for a vector ARX model, where the input and output can be multidimensional.



There are many applications for the NARX network. It can be used as a predictor, to predict the next value of the input signal. It can also be used for nonlinear filtering, in which the target output is a noise-free version of the input signal. The use of the NARX

network is shown in another important application, the modeling of nonlinear dynamic systems.

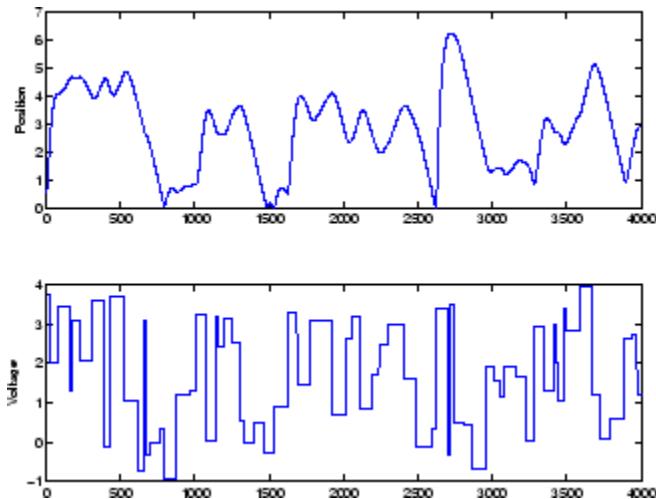
Before showing the training of the NARX network, an important configuration that is useful in training needs explanation. You can consider the output of the NARX network to be an estimate of the output of some nonlinear dynamic system that you are trying to model. The output is fed back to the input of the feedforward neural network as part of the standard NARX architecture, as shown in the left figure below. Because the true output is available during the training of the network, you could create a series-parallel architecture (see [NaPa91 on page 14-2]), in which the true output is used instead of feeding back the estimated output, as shown in the right figure below. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training.



The following shows the use of the series-parallel architecture for training a NARX network to model a dynamic system.

The example of the NARX network is the magnetic levitation system described beginning in “Use the NARMA-L2 Controller Block” on page 7-18. The bottom graph in the following figure shows the voltage applied to the electromagnet, and the top graph shows the position of the permanent magnet. The data was collected at a sampling interval of 0.01 seconds to form two time series.

The goal is to develop a NARX model for this magnetic levitation system.



First, load the training data. Use tapped delay lines with two delays for both the input and the output, so training begins with the third data point. There are two inputs to the series-parallel network, the $u(t)$ sequence and the $y(t)$ sequence.

```
load magdata
y = con2seq(y);
u = con2seq(u);
```

Create the series-parallel NARX network using the function `narxnet`. Use 10 neurons in the hidden layer and use `trainlm` for the training function, and then prepare the data with `preperts`:

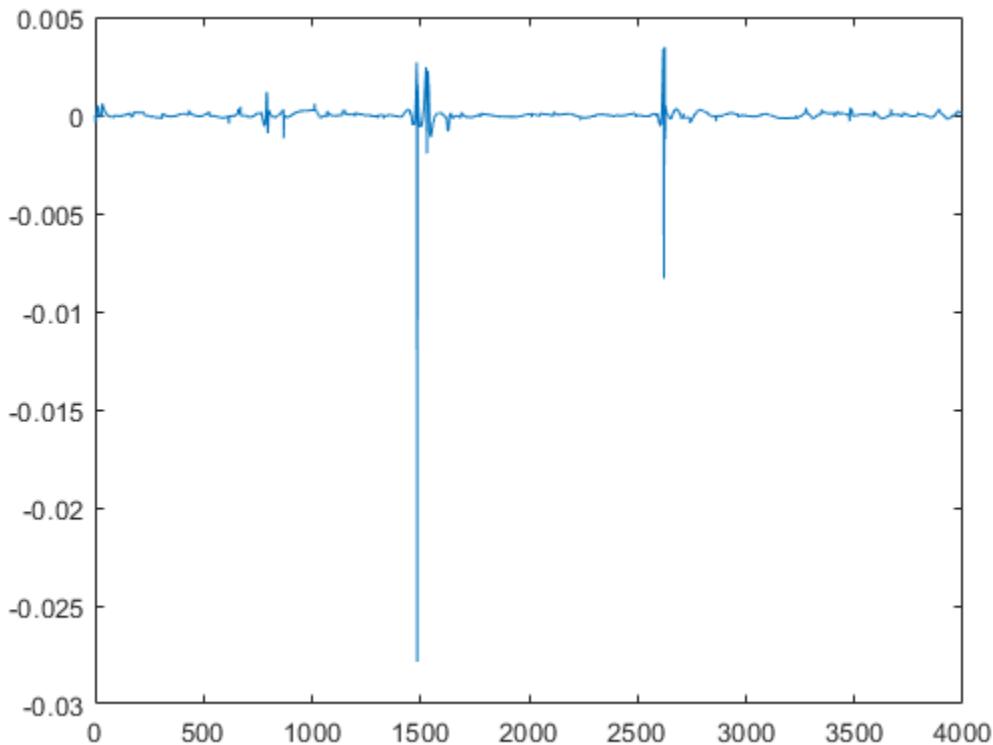
```
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preperts(narx_net,u,[],y);
```

(Notice that the y sequence is considered a feedback signal, which is an input that is also an output (target). Later, when you close the loop, the appropriate output will be connected to the appropriate input.) Now you are ready to train the network.

```
narx_net = train(narx_net,p,t,Pi);
```

You can now simulate the network and plot the resulting errors for the series-parallel implementation.

```
yp = sim(narx_net,p,Pi);
e = cell2mat(yp)-cell2mat(t);
plot(e)
```



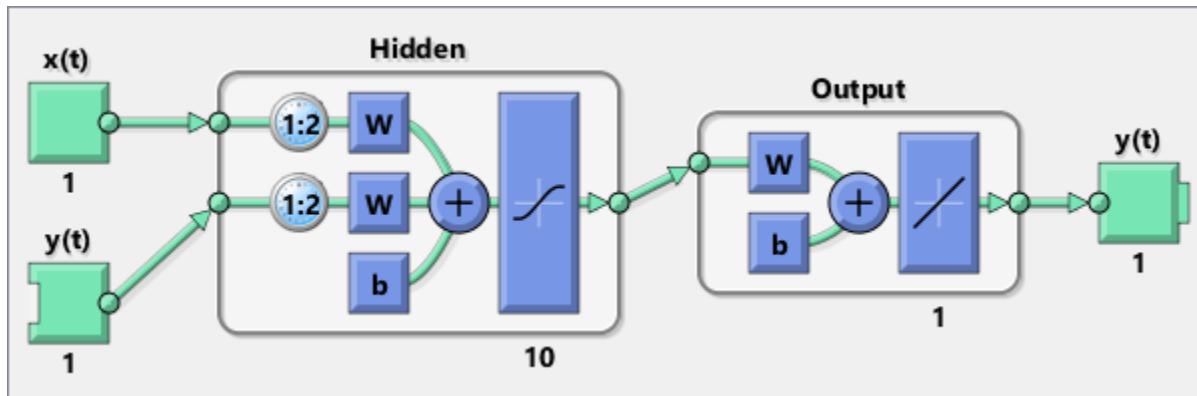
You can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form (closed loop) and then to perform an iterated prediction over many time steps. Now the parallel operation is shown.

There is a toolbox function (`closeloop`) for converting NARX (and other) networks from the series-parallel configuration (open loop), which is useful for training, to the parallel configuration (closed loop), which is useful for multi-step-ahead prediction. The following command illustrates how to convert the network that you just trained to parallel form:

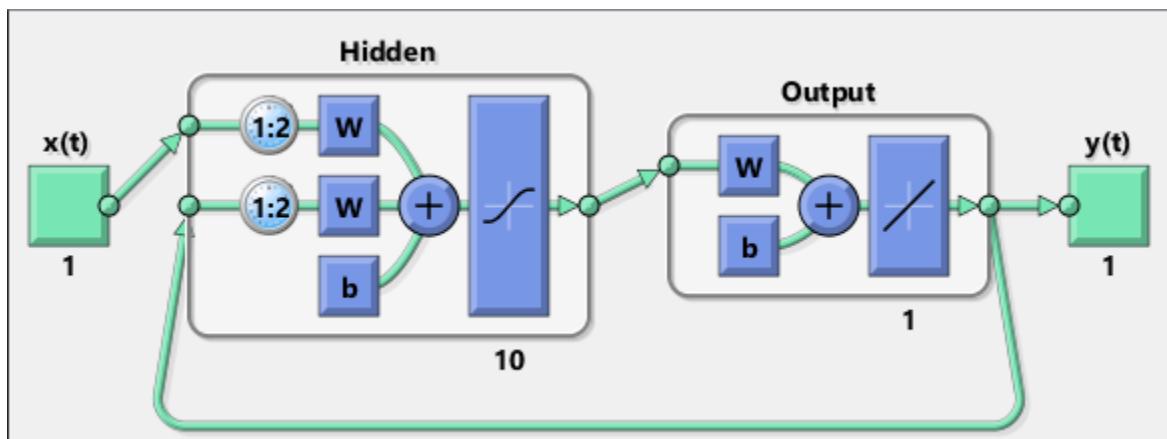
```
narx_net_closed = closeloop(narx_net);
```

To see the differences between the two networks, you can use the `view` command:

```
view(narx_net)
```



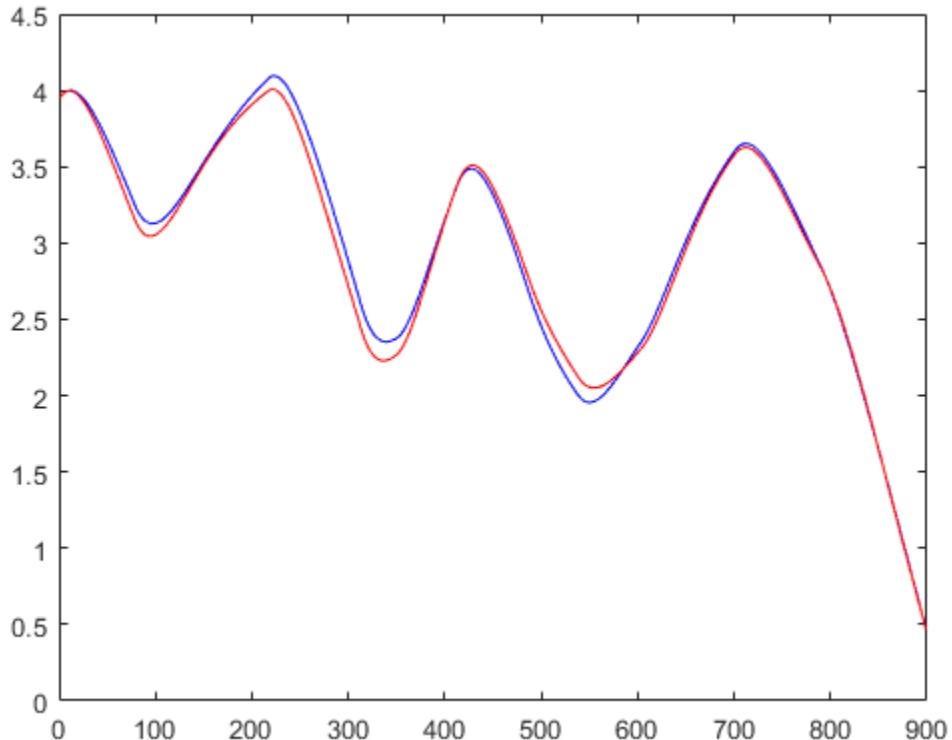
```
view(narx_net_closed)
```



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

You can now use the closed-loop (parallel) configuration to perform an iterated prediction of 900 time steps. In this network you need to load the two initial inputs and the two initial outputs as initial conditions. You can use the `prepares` function to prepare the data. It will use the network structure to determine how to divide and shift the data appropriately.

```
y1 = y(1700:2600);
u1 = u(1700:2600);
[p1,Pi1,Ai1,t1] = prepares(narx_net_closed,u1,[],y1);
yp1 = narx_net_closed(p1,Pi1,Ai1);
TS = size(t1,2);
plot(1:TS,cell2mat(t1),'b',1:TS,cell2mat(yp1),'r')
```



The figure illustrates the iterated prediction. The blue line is the actual position of the magnet, and the red line is the position predicted by the NARX neural network. Even though the network is predicting 900 time steps ahead, the prediction is very accurate.

In order for the parallel response (iterated prediction) to be accurate, it is important that the network be trained so that the errors in the series-parallel configuration (one-step-ahead prediction) are very small.

You can also create a parallel (closed loop) NARX network, using the `narxnet` command with the fourth input argument set to '`'closed'`', and train that network directly. Generally, the training takes longer, and the resulting performance is not as good as that obtained with series-parallel training.

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

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32.

Multiple External Variables

The maglev example showed how to model a time series with a single external input value over time. But the NARX network will work for problems with multiple external input elements and predict series with multiple elements. In these cases, the input and target consist of row cell arrays representing time, but with each cell element being an N-by-1 vector for the N elements of the input or target signal.

For example, here is a dataset which consists of 2-element external variables predicting a 1-element series.

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

The external inputs X are formatted as a row cell array of 2-element vectors, with each vector representing acid and base solution flow. The targets represent the resulting pH of the solution over time.

You can reformat your own multi-element series data from matrix form to neural network time-series form with the function `con2seq`.

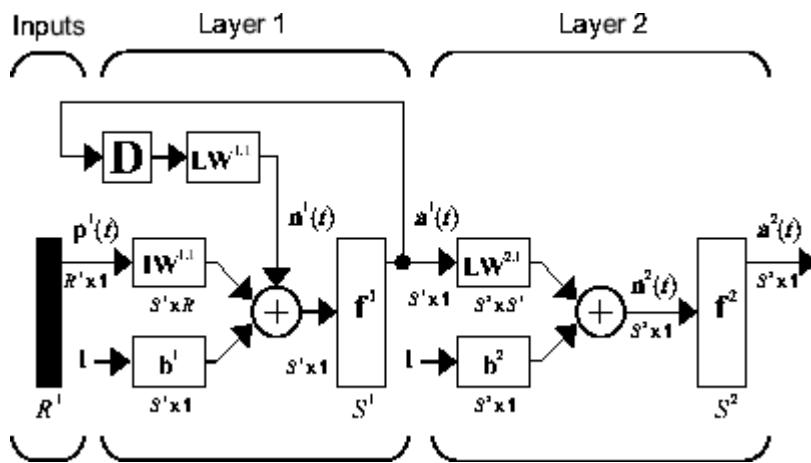
The process for training a network proceeds as it did above for the maglev problem.

```
net = narxnet(10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
e = gsubtract(t,y);
```

To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 6-52.

Design Layer-Recurrent Neural Networks

The next dynamic network to be introduced is the Layer-Recurrent Network (LRN). An earlier simplified version of this network was introduced by Elman [Elma90 on page 14-2]. In the LRN, there is a feedback loop, with a single delay, around each layer of the network except for the last layer. The original Elman network had only two layers, and used a `tansig` transfer function for the hidden layer and a `purelin` transfer function for the output layer. The original Elman network was trained using an approximation to the backpropagation algorithm. The `layrecnet` command generalizes the Elman network to have an arbitrary number of layers and to have arbitrary transfer functions in each layer. The toolbox trains the LRN using exact versions of the gradient-based algorithms discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2. The following figure illustrates a two-layer LRN.



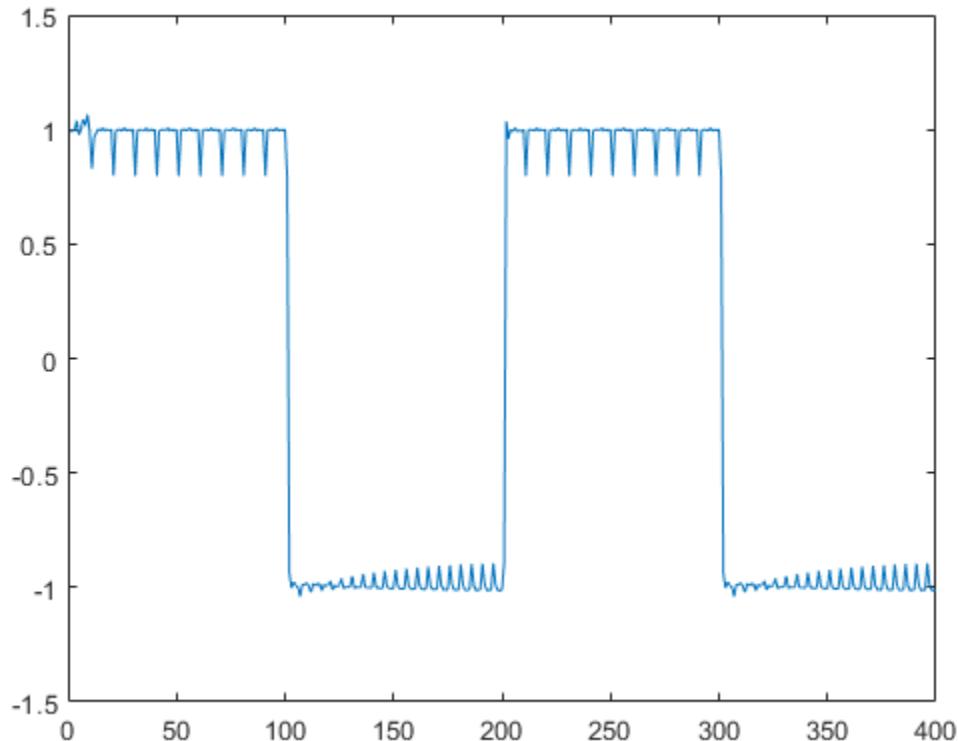
The LRN configurations are used in many filtering and modeling applications discussed already. To show its operation, this example uses the “phoneme” detection problem discussed in “Design Time Series Distributed Delay Neural Networks” on page 6-20. Here is the code to load the data and to create and train the network:

```
load phoneme
p = con2seq(y);
t = con2seq(t);
lrn_net = layrecnet(1,8);
lrn_net.trainFcn = 'trainbr';
lrn_net.trainParam.show = 5;
```

```
lrn_net.trainParam.epochs = 50;  
lrn_net = train(lrn_net,p,t);
```

After training, you can plot the response using the following code:

```
y = lrn_net(p);  
plot(cell2mat(y))
```



The plot shows that the network was able to detect the “phonemes.” The response is very similar to the one obtained using the TDNN.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give

different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

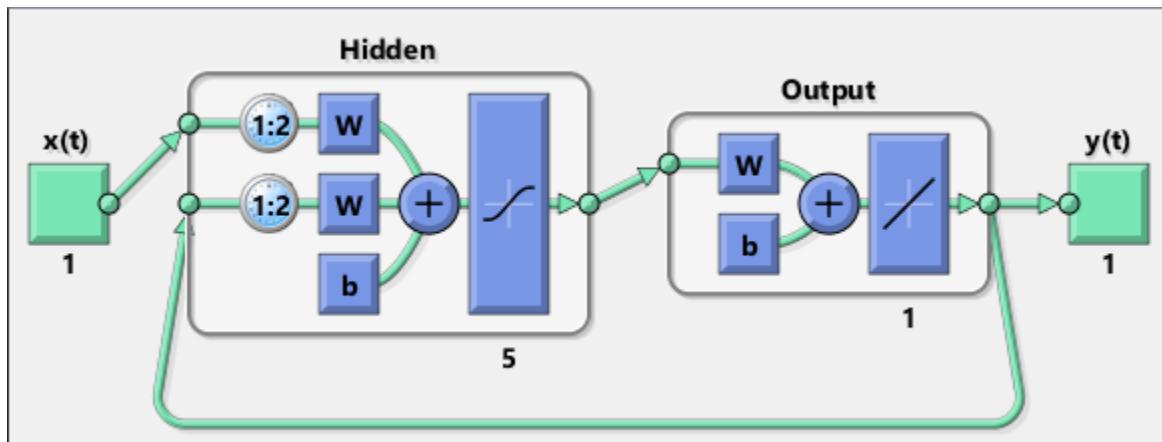
There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32.

Create Reference Model Controller with MATLAB Script

So far, this topic has described the training procedures for several specific dynamic network architectures. However, *any* network that can be created in the toolbox can be trained using the training functions described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2 so long as the components of the network are differentiable. This section gives an example of how to create and train a custom architecture. The custom architecture you will use is the model reference adaptive control (MRAC) system that is described in detail in “Design Model-Reference Neural Controller in Simulink” on page 7-23.

As you can see in “Design Model-Reference Neural Controller in Simulink” on page 7-23, the model reference control architecture has two subnetworks. One subnetwork is the model of the plant that you want to control. The other subnetwork is the controller. You will begin by training a NARX network that will become the plant model subnetwork. For this example, you will use the robot arm to represent the plant, as described in “Design Model-Reference Neural Controller in Simulink” on page 7-23. The following code will load data collected from the robot arm and create and train a NARX network. For this simple problem, you do not need to preprocess the data, and all of the data can be used for training, so no data division is needed.

```
[u,y] = robotarm_dataset;
d1 = [1:2];
d2 = [1:2];
S1 = 5;
narx_net = narxnet(d1,d2,S1);
narx_net.divideFcn = 'none';
narx_net.inputs{1}.processFcns = {};
narx_net.inputs{2}.processFcns = {};
narx_net.outputs{2}.processFcns = {};
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,{},y);
narx_net = train(narx_net,p,t,Pi);
narx_net_closed = closeloop(narx_net);
view(narx_net_closed)
```



The resulting network is shown in the figure.

Now that the NARX plant model is trained, you can create the total MRAC system and insert the NARX model inside. Begin with a feedforward network, and then add the feedback connections. Also, turn off learning in the plant model subnetwork, since it has already been trained. The next stage of training will train only the controller subnetwork.

```
mrac_net = feedforwardnet([S1 1 S1]);
mrac_net.layerConnect = [0 1 0 1;1 0 0 0;0 1 0 1;0 0 1 0];
mrac_net.outputs{4}.feedbackMode = 'closed';
mrac_net.layers{2}.transferFcn = 'purelin';
mrac_net.layerWeights{3,4}.delays = 1:2;
mrac_net.layerWeights{3,2}.delays = 1:2;
mrac_net.layerWeights{3,2}.learn = 0;
mrac_net.layerWeights{3,4}.learn = 0;
mrac_net.layerWeights{4,3}.learn = 0;
mrac_net.biases{3}.learn = 0;
mrac_net.biases{4}.learn = 0;
```

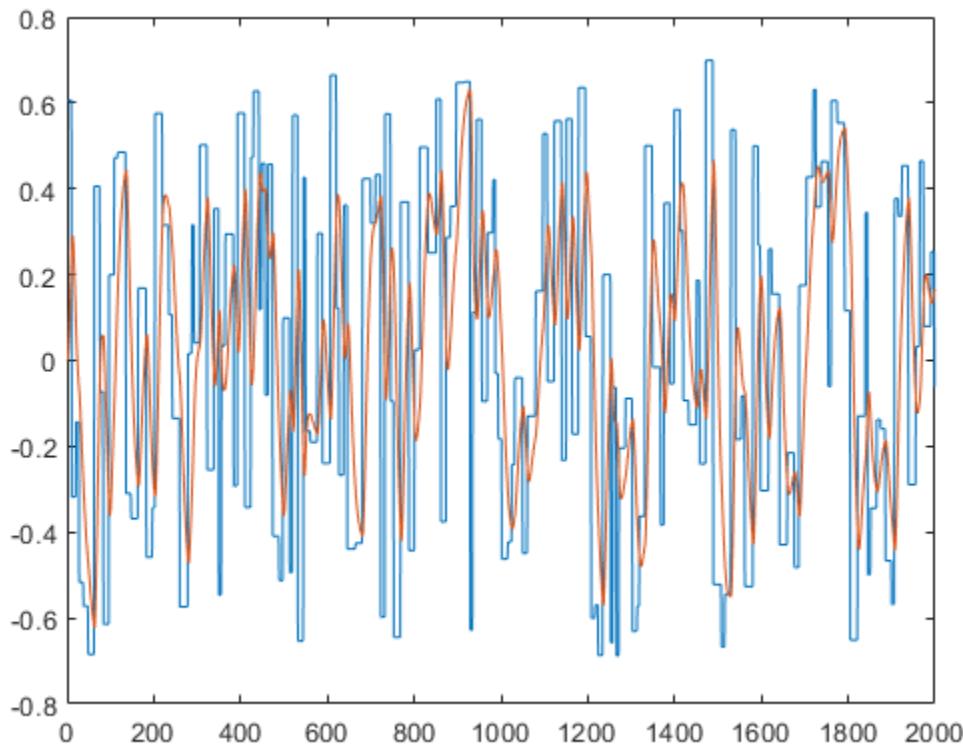
The following code turns off data division and preprocessing, which are not needed for this example problem. It also sets the delays needed for certain layers and names the network.

```
mrac_net.divideFcn = '';
mrac_net.inputs{1}.processFcns = {};
mrac_net.outputs{4}.processFcns = {};
mrac_net.name = 'Model Reference Adaptive Control Network';
mrac_net.layerWeights{1,2}.delays = 1:2;
```

```
mrac_net.layerWeights{1,4}.delays = 1:2;  
mrac_net.inputWeights{1}.delays = 1:2;
```

To configure the network, you need some sample training data. The following code loads and plots the training data, and configures the network:

```
[refin,refout] = refmodel_dataset;  
ind = 1:length(refin);  
plot(ind,cell2mat(refin),ind,cell2mat(refout))  
mrac_net = configure(mrac_net,refin,refout);
```



You want the closed-loop MRAC system to respond in the same way as the reference model that was used to generate this data. (See “Use the Model Reference Controller Block” on page 7-24 for a description of the reference model.)

Now insert the weights from the trained plant model network into the appropriate location of the MRAC system.

```
mrac_net.LW{3,2} = narx_net_closed.IW{1};
mrac_net.LW{3,4} = narx_net_closed.LW{1,2};
mrac_net.b{3} = narx_net_closed.b{1};
mrac_net.LW{4,3} = narx_net_closed.LW{2,1};
mrac_net.b{4} = narx_net_closed.b{2};
```

You can set the output weights of the controller network to zero, which will give the plant an initial input of zero.

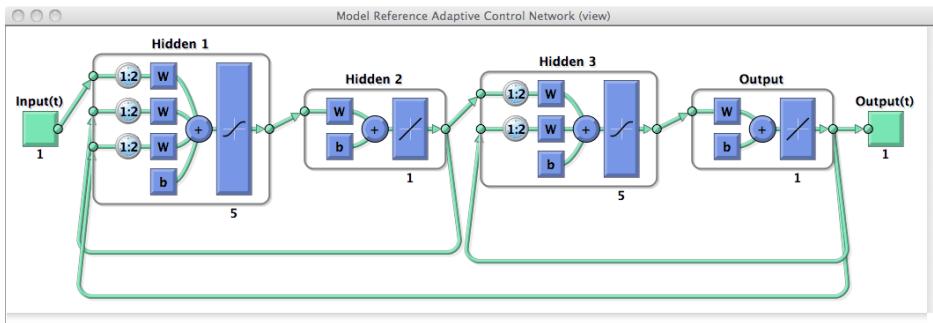
```
mrac_net.LW{2,1} = zeros(size(mrac_net.LW{2,1}));
mrac_net.b{2} = 0;
```

You can also associate any plots and training function that you desire to the network.

```
mrac_net.plotFcns = {'plotperform', 'plottrainstate',...
    'ploterrhist', 'plotregression', 'plotresponse'};
mrac_net.trainFcn = 'trainlm';
```

The final MRAC network can be viewed with the following command:

```
view(mrac_net)
```



Layer 3 and layer 4 (output) make up the plant model subnetwork. Layer 1 and layer 2 make up the controller.

You can now prepare the training data and train the network.

```
[x_tot,xi_tot,ai_tot,t_tot] = ...
    prepares(mrac_net,refin,[],refout);
mrac_net.trainParam.epochs = 50;
```

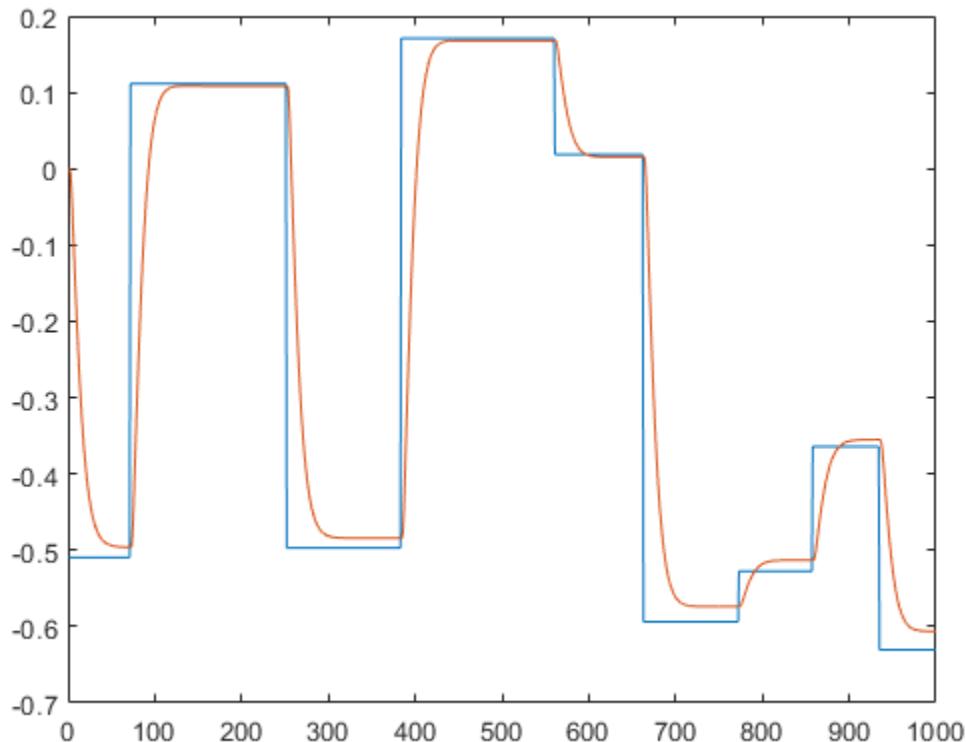
```
mrac_net.trainParam.min_grad = 1e-10;
[mrac_net,tr] = train(mrac_net,x_tot,t_tot,xi_tot,ai_tot);
```

Note Notice that you are using the `trainlm` training function here, but any of the training functions discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2 could be used as well. Any network that you can create in the toolbox can be trained with any of those training functions. The only limitation is that all of the parts of the network must be differentiable.

You will find that the training of the MRAC system takes much longer than the training of the NARX plant model. This is because the network is recurrent and dynamic backpropagation must be used. This is determined automatically by the toolbox software and does not require any user intervention. There are several implementations of dynamic backpropagation (see [DeHa07 on page 14-2]), and the toolbox software automatically determines the most efficient one for the selected network architecture and training algorithm.

After the network has been trained, you can test the operation by applying a test input to the MRAC network. The following code creates a `skyline` input function, which is a series of steps of random height and width, and applies it to the trained MRAC network.

```
testin = skyline(1000,50,200,-.7,.7);
testinseq = con2seq(testin);
testoutseq = mrac_net(testinseq);
testout = cell2mat(testoutseq);
figure
plot([testin' testout'])
```



From the figure, you can see that the plant model output does follow the reference input with the correct critically damped response, even though the input sequence was not the same as the input sequence in the training data. The steady state response is not perfect for each step, but this could be improved with a larger training set and perhaps more hidden neurons.

The purpose of this example was to show that you can create your own custom dynamic network and train it using the standard toolbox training functions without any modifications. Any network that you can create in the toolbox can be trained with the standard training functions, as long as each component of the network has a defined derivative.

It should be noted that recurrent networks are generally more difficult to train than feedforward networks. See [HDH09 on page 14-2] for some discussion of these training difficulties.

Multiple Sequences with Dynamic Neural Networks

There are times when time-series data is not available in one long sequence, but rather as several shorter sequences. When dealing with static networks and concurrent batches of static data, you can simply append data sets together to form one large concurrent batch. However, you would not generally want to append time sequences together, since that would cause a discontinuity in the sequence. For these cases, you can create a concurrent set of sequences, as described in “Understanding Deep Learning Toolbox Data Structures” on page 4-23.

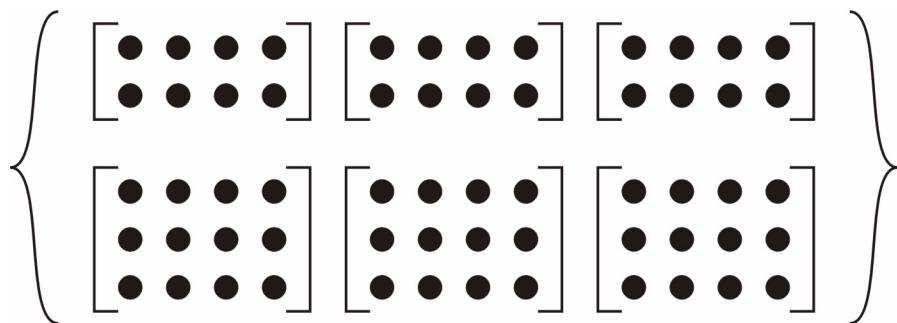
When training a network with a concurrent set of sequences, it is required that each sequence be of the same length. If this is not the case, then the shorter sequence inputs and targets should be padded with NaNs, in order to make all sequences the same length. The targets that are assigned values of NaN will be ignored during the calculation of network performance.

The following code illustrates the use of the function `catsamples` to combine several sequences together to form a concurrent set of sequences, while at the same time padding the shorter sequences.

```
load magmulseq
y_mul = catsamples(y1,y2,y3,'pad');
u_mul = catsamples(u1,u2,u3,'pad');
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u_mul,[],y_mul);
narx_net = train(narx_net,p,t,Pi);
```

Neural Network Time-Series Utilities

There are other utility functions that are useful when manipulating neural network data, which can consist of time sequences, concurrent batches or combinations of both. It can also include multiple signals (as in multiple input, output or target vectors). The following diagram illustrates the structure of a general neural network data object. For this example there are three time steps of a batch of four samples (four sequences) of two signals. One signal has two elements, and the other signal has three elements.



The following table lists some of the more useful toolbox utility functions for neural network data. They allow you to do things like add, subtract, multiply, divide, etc. (Addition and subtraction of cell arrays do not have standard definitions, but for neural network data these operations are well defined and are implemented in the following functions.)

Function	Operation
gadd	Add neural network (nn) data.
gdivide	Divide nn data.
getelements	Select indicated elements from nn data.
getsamples	Select indicated samples from nn data.
getsignals	Select indicated signals from nn data.
gettimesteps	Select indicated time steps from nn data.
gmultiply	Multiply nn data.
gnegate	Take the negative of nn data.
gsubtract	Subtract nn data.

Function	Operation
<code>nndata</code>	Create an nn data object of specified size, where values are assigned randomly or to a constant.
<code>nnsize</code>	Return number of elements, samples, time steps and signals in an nn data object.
<code>numelements</code>	Return the number of elements in nn data.
<code>numsamples</code>	Return the number of samples in nn data.
<code>numsignals</code>	Return the number of signals in nn data.
<code>numtimesteps</code>	Return the number of time steps in nn data.
<code>setelements</code>	Set specified elements of nn data.
<code>setsamples</code>	Set specified samples of nn data.
<code>setsignals</code>	Set specified signals of nn data.
<code>settimesteps</code>	Set specified time steps of nn data.

There are also some useful plotting and analysis functions for dynamic networks that are listed in the following table. There are examples of using these functions in the “Getting Started with Deep Learning Toolbox”.

Function	Operation
<code>ploterrcorr</code>	Plot the autocorrelation function of the error.
<code>plotinerrcorr</code>	Plot the crosscorrelation between the error and the input.
<code>plotresponse</code>	Plot network output and target versus time.

Train Neural Networks with Error Weights

In the default mean square error performance function (see “Train and Apply Multilayer Shallow Neural Networks” on page 5-17), each squared error contributes the same amount to the performance function as follows:

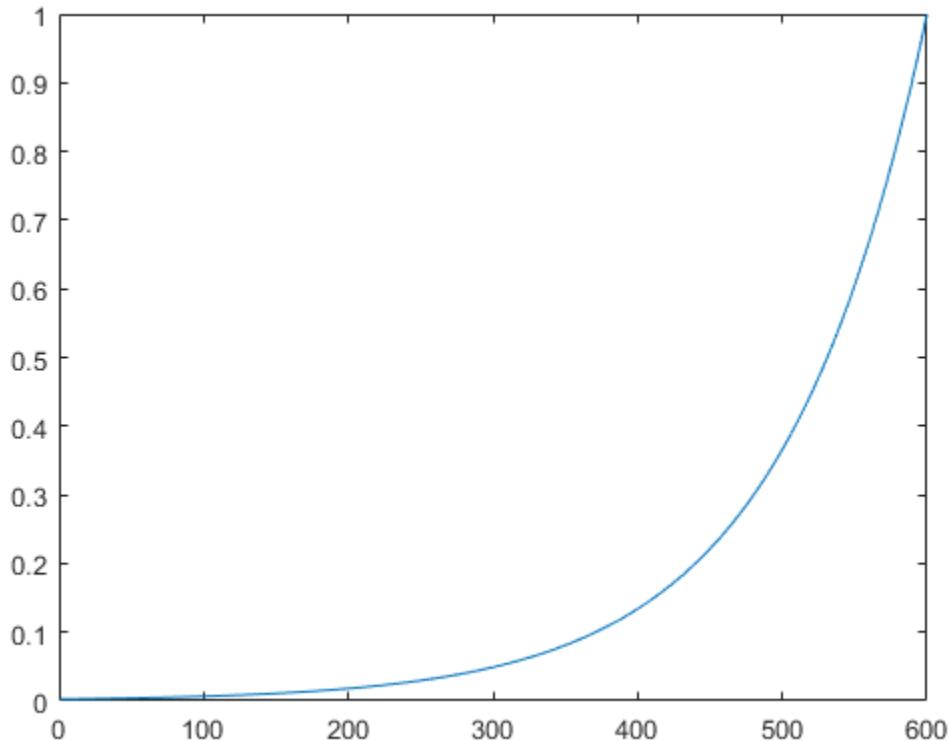
$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2$$

However, the toolbox allows you to weight each squared error individually as follows:

$$F = mse = \frac{1}{N} \sum_{i=1}^N w_i^e (e_i)^2 = \frac{1}{N} \sum_{i=1}^N w_i^e (t_i - a_i)^2$$

The error weighting object needs to have the same dimensions as the target data. In this way, errors can be weighted according to time step, sample number, signal number or element number. The following is an example of weighting the errors at the end of a time sequence more heavily than errors at the beginning of a time sequence. The error weighting object is passed as the last argument in the call to `train`.

```
y = laser_dataset;
y = y(1:600);
ind = 1:600;
ew = 0.99.^ (600-ind);
figure
plot(ew)
```

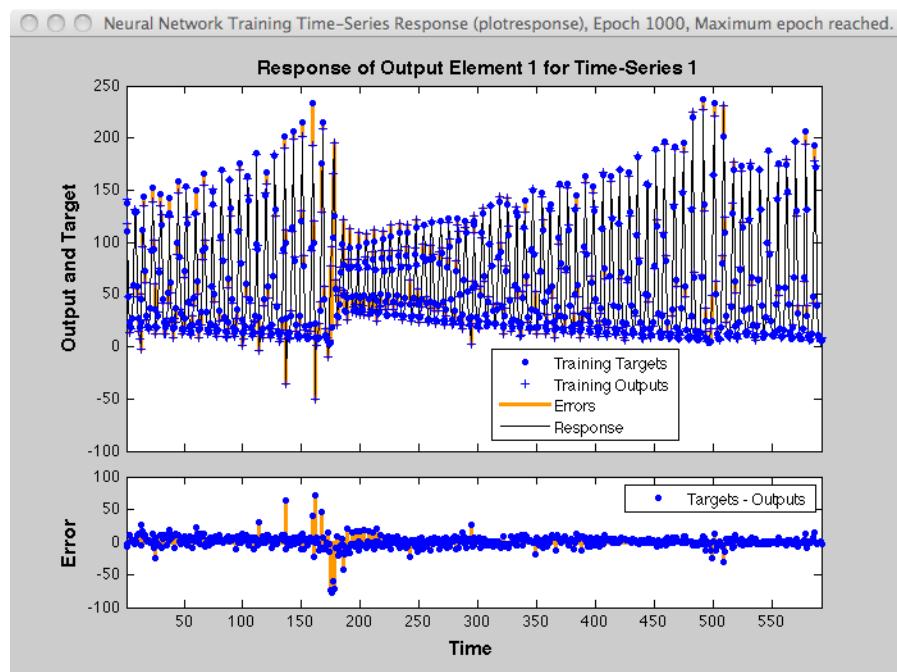


```
ew = con2seq(ew);
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = ' ';
[p,Pi,Ai,t,ew1] = preparets(ftdnn_net,y,y,[],ew);
[ftdnn_net1,tr] = train(ftdnn_net,p,t,Pi,Ai,ew1);
```

The figure illustrates the error weighting for this example. There are 600 time steps in the training data, and the errors are weighted exponentially, with the last squared error having a weight of 1, and the squared error at the first time step having a weighting of 0.0024.

The response of the trained network is shown in the following figure. If you compare this response to the response of the network that was trained without exponential weighting

on the squared errors, as shown in “Design Time Series Time-Delay Neural Networks” on page 6-14, you can see that the errors late in the sequence are smaller than the errors earlier in the sequence. The errors that occurred later are smaller because they contributed more to the weighted performance index than earlier errors.



Normalize Errors of Multiple Outputs

The most common performance function used to train neural networks is mean squared error (mse). However, with multiple outputs that have different ranges of values, training with mean squared error tends to optimize accuracy on the output element with the wider range of values relative to the output element with a smaller range.

For instance, here two target elements have very different ranges:

```
x = -1:0.01:1;
t1 = 100*sin(x);
t2 = 0.01*cos(x);
t = [t1; t2];
```

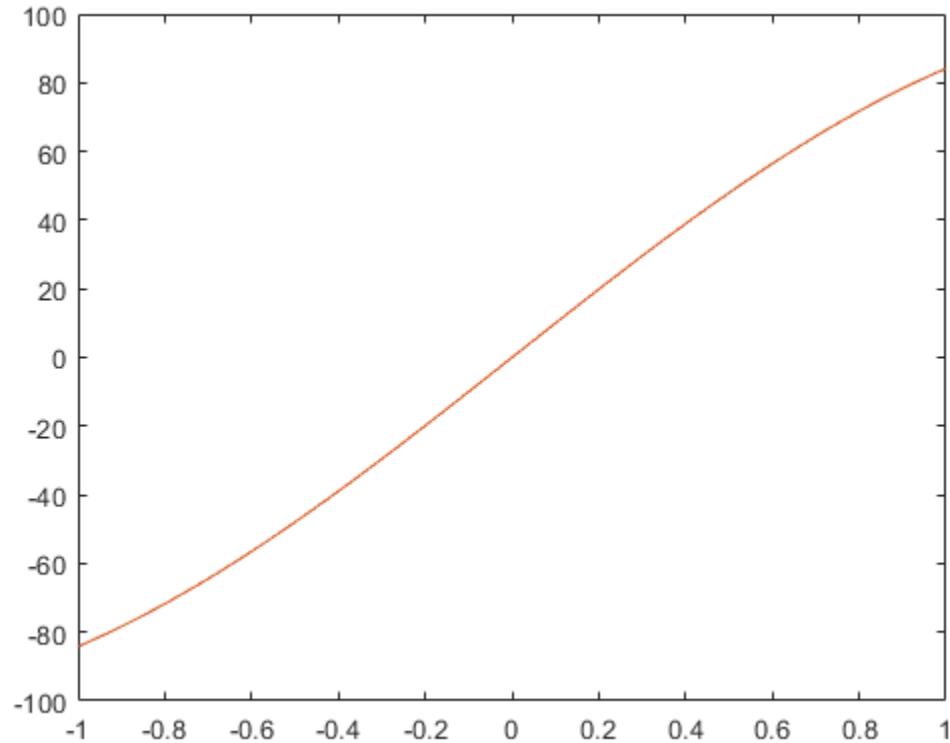
The range of t_1 is 200 (from a minimum of -100 to a maximum of 100), while the range of t_2 is only 0.02 (from -0.01 to 0.01). The range of t_1 is 10,000 times greater than the range of t_2 .

If you create and train a neural network on this to minimize mean squared error, training favors the relative accuracy of the first output element over the second.

```
net = feedforwardnet(5);
net1 = train(net,x,t);
y = net1(x);
```

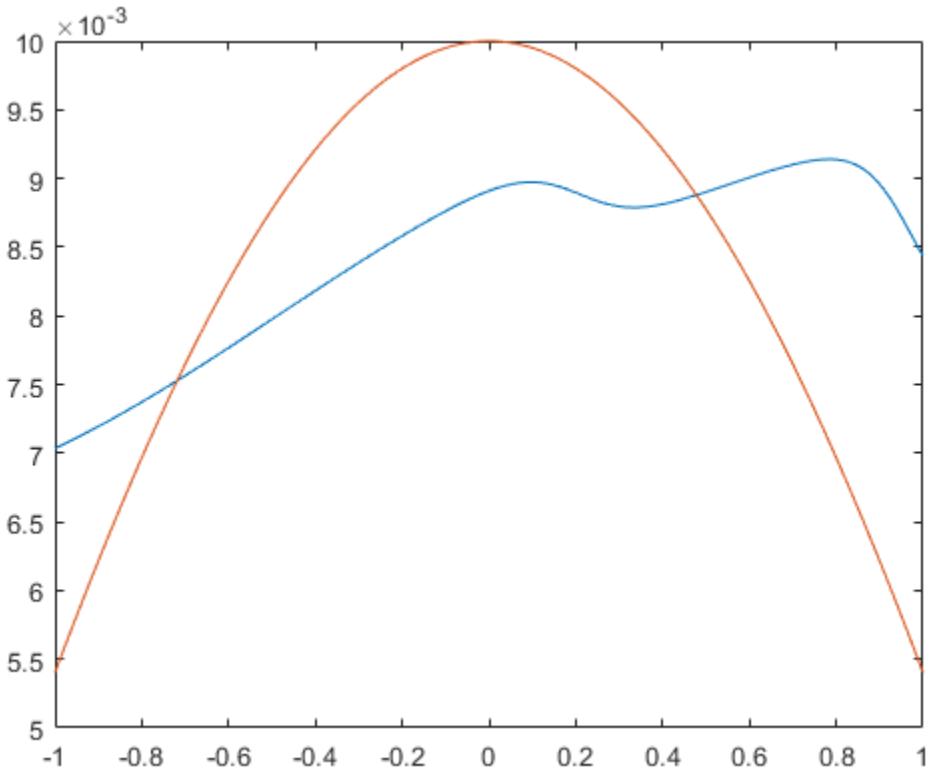
Here you can see that the network has learned to fit the first output element very well.

```
figure(1)
plot(x,y(1,:),x,t(1,:))
```



However, the second element's function is not fit nearly as well.

```
figure(2)
plot(x,y(2,:),x,t(2,:))
```

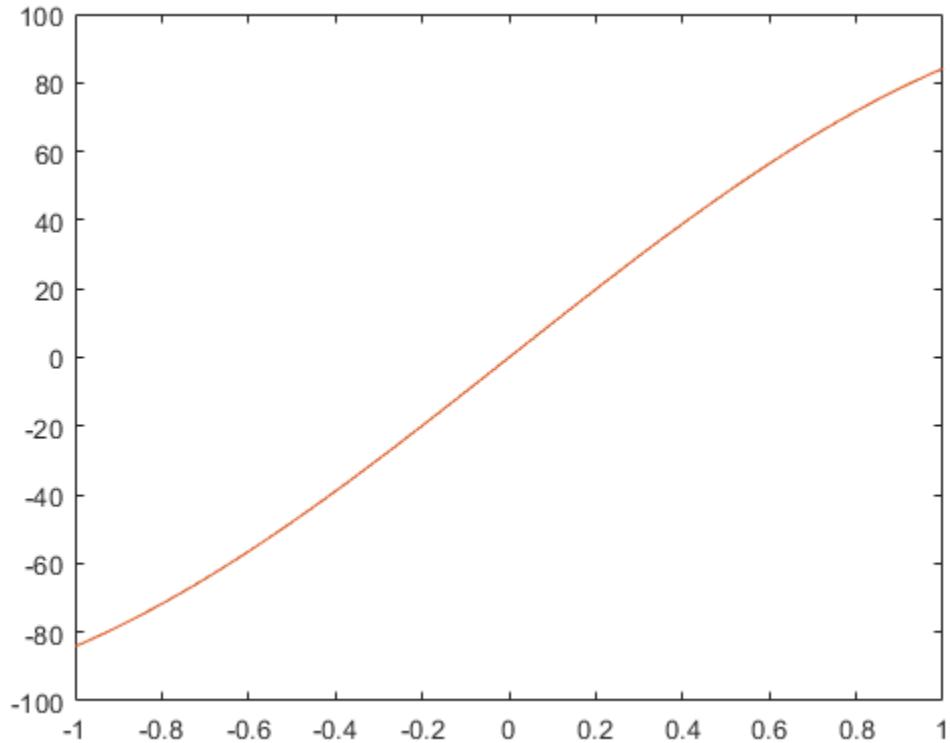


To fit both output elements equally well in a relative sense, set the normalization performance parameter to 'standard'. This then calculates errors for performance measures as if each output element has a range of 2 (i.e., as if each output element's values range from -1 to 1, instead of their differing ranges).

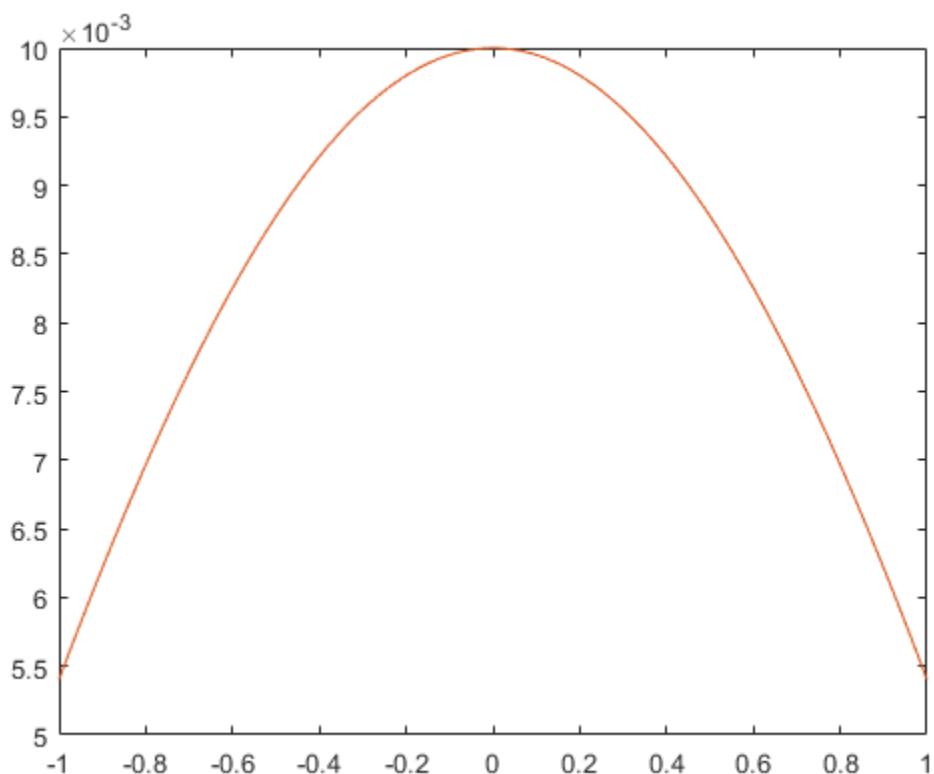
```
net.performParam.normalization = 'standard';
net2 = train(net,x,t);
y = net2(x);
```

Now the two output elements both fit well.

```
figure(3)
plot(x,y(1,:),x,t(1,:))
```



```
figure(4)
plot(x,y(2,:),x,t(2,:))
```



Multistep Neural Network Prediction

In this section...

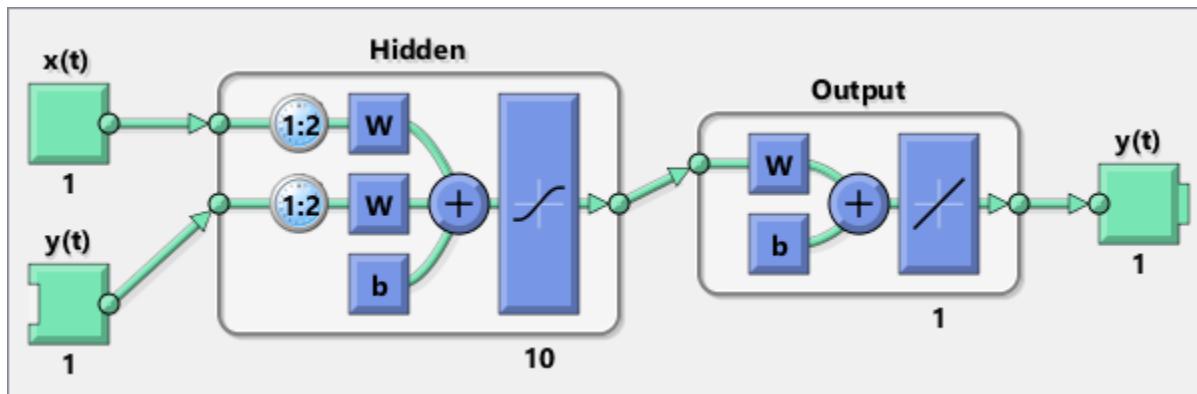
- “Set Up in Open-Loop Mode” on page 6-52
- “Multistep Closed-Loop Prediction From Initial Conditions” on page 6-53
- “Multistep Closed-Loop Prediction Following Known Sequence” on page 6-53
- “Following Closed-Loop Simulation with Open-Loop Simulation” on page 6-54

Set Up in Open-Loop Mode

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words they continue to predict when external feedback is missing, by using internal feedback.

Here a neural network is trained to model the magnetic levitation system and simulated in the default open-loop mode.

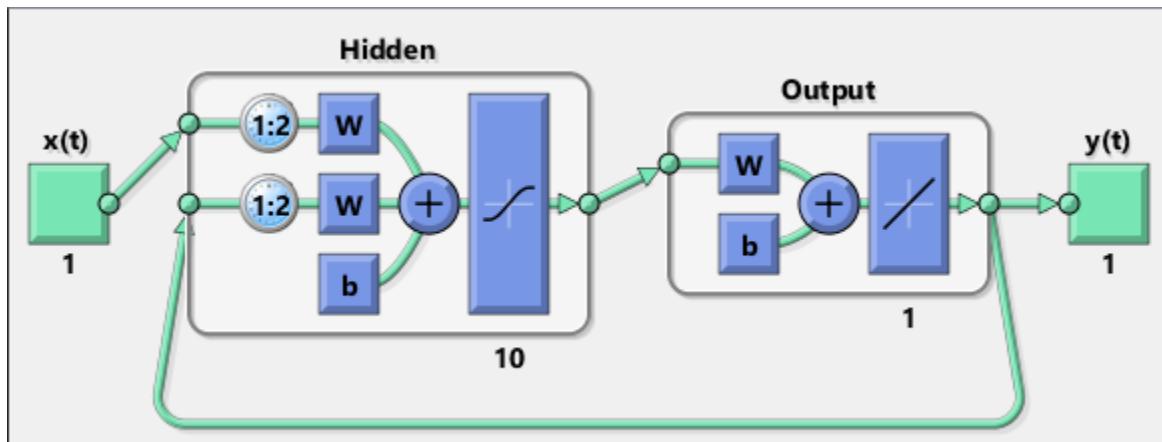
```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
view(net)
```



Multistep Closed-Loop Prediction From Initial Conditions

A neural network can also be simulated only in closed-loop form, so that given an external input series and initial conditions, the neural network performs as many predictions as the input series has time steps.

```
netc = closeloop(net);
view(netc)
```



Here the training data is used to define the inputs x , and the initial input and layer delay states, x_i and a_i , but they can be defined to make multiple predictions for any input series and initial states.

```
[x,xi,ai,t] = preparets(netc,X,[],T);
yc = netc(x,xi,ai);
```

Multistep Closed-Loop Prediction Following Known Sequence

It can also be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired.

Just as `openloop` and `closeloop` can be used to transform between open- and closed-loop neural networks, they can convert the state of open- and closed-loop networks. Here are the full interfaces for these functions.

```
[open_net,open_xi,open_ai] = openloop(closed_net,closed_xi,closed_ai);  
[closed_net,closed_xi,closed_ai] = closeloop(open_net,open_xi,open_ai);
```

Consider the case where you might have a record of the Maglev's behavior for 20 time steps, and you want to predict ahead for 20 more time steps.

First, define the first 20 steps of inputs and targets, representing the 20 time steps where the known output is defined by the targets t . With the next 20 time steps of the input are defined, use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);  
t1 = t(1:20);  
x2 = x(21:40);
```

The open-loop neural network is then simulated on this data.

```
[x,xi,ai,t] = prepares(net,x1,[],t1);  
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states xf and layer states af of the open-loop network become the initial input states xi and layer states ai of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically use `prepares` to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `prepares` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that you can set $x2$ to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs are used:

```
x2 = num2cell(rand(1,10));  
[y2,xf,af] = netc(x2,xi,ai);
```

Following Closed-Loop Simulation with Open-Loop Simulation

If after simulating the network in closed-loop form, you can continue the simulation from there in open-loop form. Here the closed-loop state is converted back to open-loop state.

(You do not have to convert the network back to open-loop form as you already have the original open-loop network.)

```
[~,xi,ai] = openloop(netc,xf,af);
```

Now you can define continuations of the external input and open-loop feedback, and simulate the open-loop network.

```
x3 = num2cell(rand(2,10));  
y3 = net(x3,xi,ai);
```

In this way, you can switch simulation between open-loop and closed-loop manners. One application for this is making time-series predictions of a sensor, where the last sensor value is usually known, allowing open-loop prediction of the next step. But on some occasions the sensor reading is not available, or known to be erroneous, requiring a closed-loop prediction step. The predictions can alternate between open-loop and closed-loop form, depending on the availability of the last step's sensor reading.

Control Systems

- “Introduction to Neural Network Control Systems” on page 7-2
- “Design Neural Network Predictive Controller in Simulink” on page 7-4
- “Design NARMA-L2 Neural Controller in Simulink” on page 7-14
- “Design Model-Reference Neural Controller in Simulink” on page 7-23
- “Import-Export Neural Network Simulink Control Systems” on page 7-31

Introduction to Neural Network Control Systems

Neural networks have been applied successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99 on page 14-2]. This topic introduces three popular neural network architectures for prediction and control that have been implemented in the Deep Learning Toolbox software, and presents brief descriptions of each of these architectures and shows how you can use them:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

There are typically two steps involved when using neural networks for control:

- 1 System identification
- 2 Control design

In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this topic, the system identification stage is identical. The control design stage, however, is different for each architecture:

- For model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance.
- For NARMA-L2 control, the controller is simply a rearrangement of the plant model.
- For model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training.

The next three sections discuss model predictive control, NARMA-L2 control, and model reference control. Each section consists of a brief description of the control concept, followed by an example of the use of the appropriate Deep Learning Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Deep Learning Toolbox blockset.

To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application.

- **Model Predictive Control** — This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form. (This is true for all three control architectures.) The controller, however, requires a significant amount of online computation, because an optimization algorithm is performed at each sample time to compute the optimal control input.
- **NARMA-L2 Control** — This controller requires the least computation of these three architectures. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (“Identification of the NARMA-L2 Model” on page 7-14 describes the companion form model.)
- **Model Reference Control** — The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained offline, in addition to the neural network plant model. The controller training is computationally expensive, because it requires the use of dynamic backpropagation [HaJe99 on page 14-2]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

Design Neural Network Predictive Controller in Simulink

In this section...

[“System Identification” on page 7-4](#)

[“Predictive Control” on page 7-5](#)

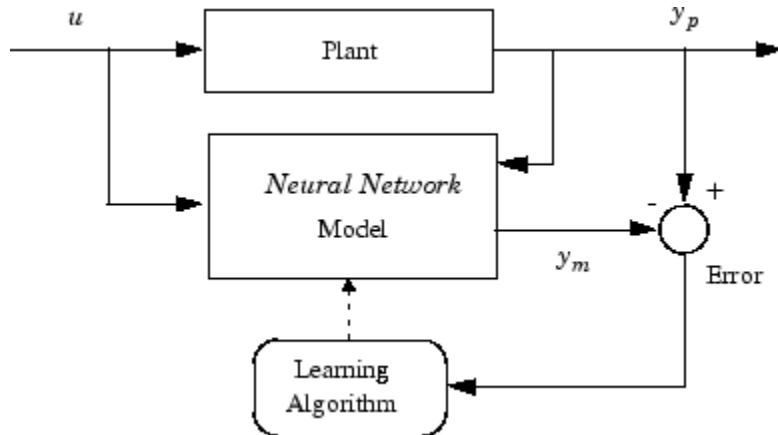
[“Use the Neural Network Predictive Controller Block” on page 7-6](#)

The neural network predictive controller that is implemented in the Deep Learning Toolbox software uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance. (See the Model Predictive Control Toolbox™ documentation for complete coverage of the application of various model predictive control strategies to linear systems.)

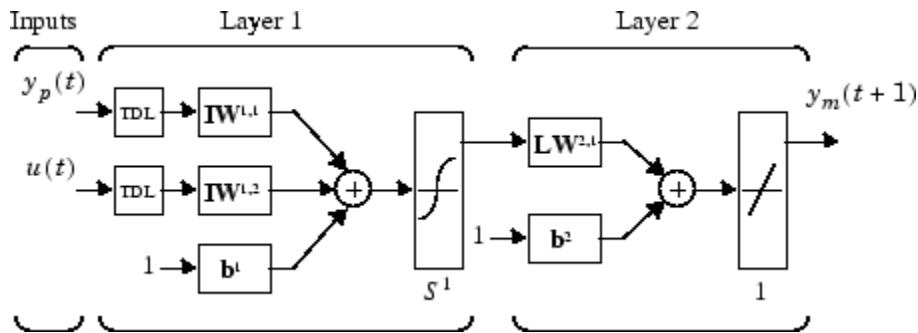
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink environment.

System Identification

The first stage of model predictive control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by the following figure:



The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.



This network can be trained offline in batch mode, using data collected from the operation of the plant. You can use any of the training algorithms discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2 for network training. This process is discussed in more detail in following sections.

Predictive Control

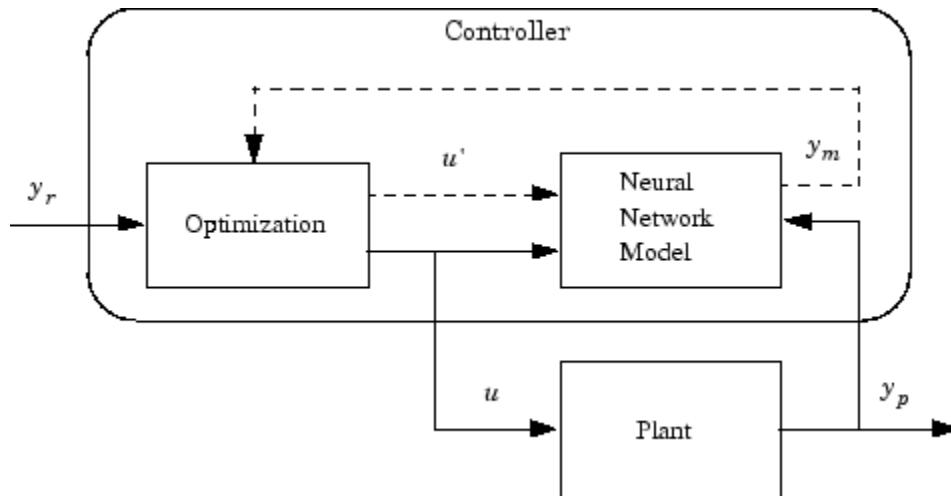
The model predictive control method is based on the receding horizon technique [SoHa96 on page 14-2]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine

the control signal that minimizes the following performance criterion over the specified horizon

$$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$

where N_1 , N_2 , and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response, and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index.

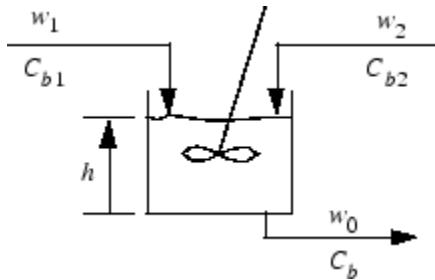
The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J , and then the optimal u is input to the plant. The controller block is implemented in Simulink, as described in the following section.



Use the Neural Network Predictive Controller Block

This section shows how the NN Predictive Controller block is used. The first step is to copy the NN Predictive Controller block from the Deep Learning Toolbox block library to the Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the predictive controller. This example uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.



The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

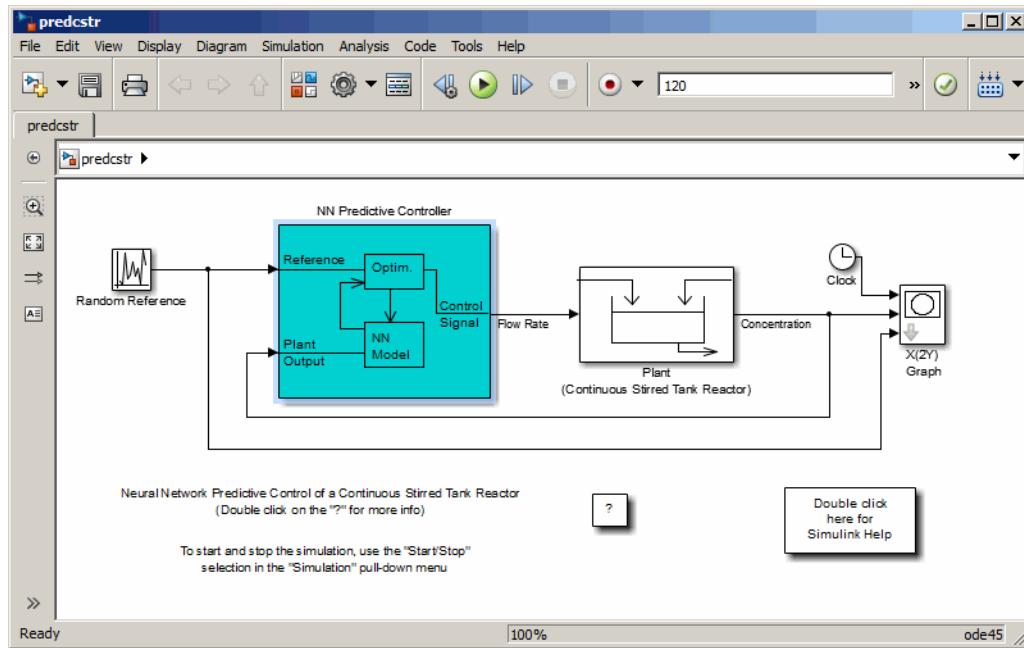
$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1} , and $w_2(t)$ is the flow rate of the diluted feed C_{b2} . The input concentrations are set to $C_{b1} = 24.9$ and $C_{b2} = 0.1$. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$.

The objective of the controller is to maintain the product concentration by adjusting the flow $w_1(t)$. To simplify the example, set $w_2(t) = 0.1$. The level of the tank $h(t)$ is not controlled for this experiment.

To run this example:

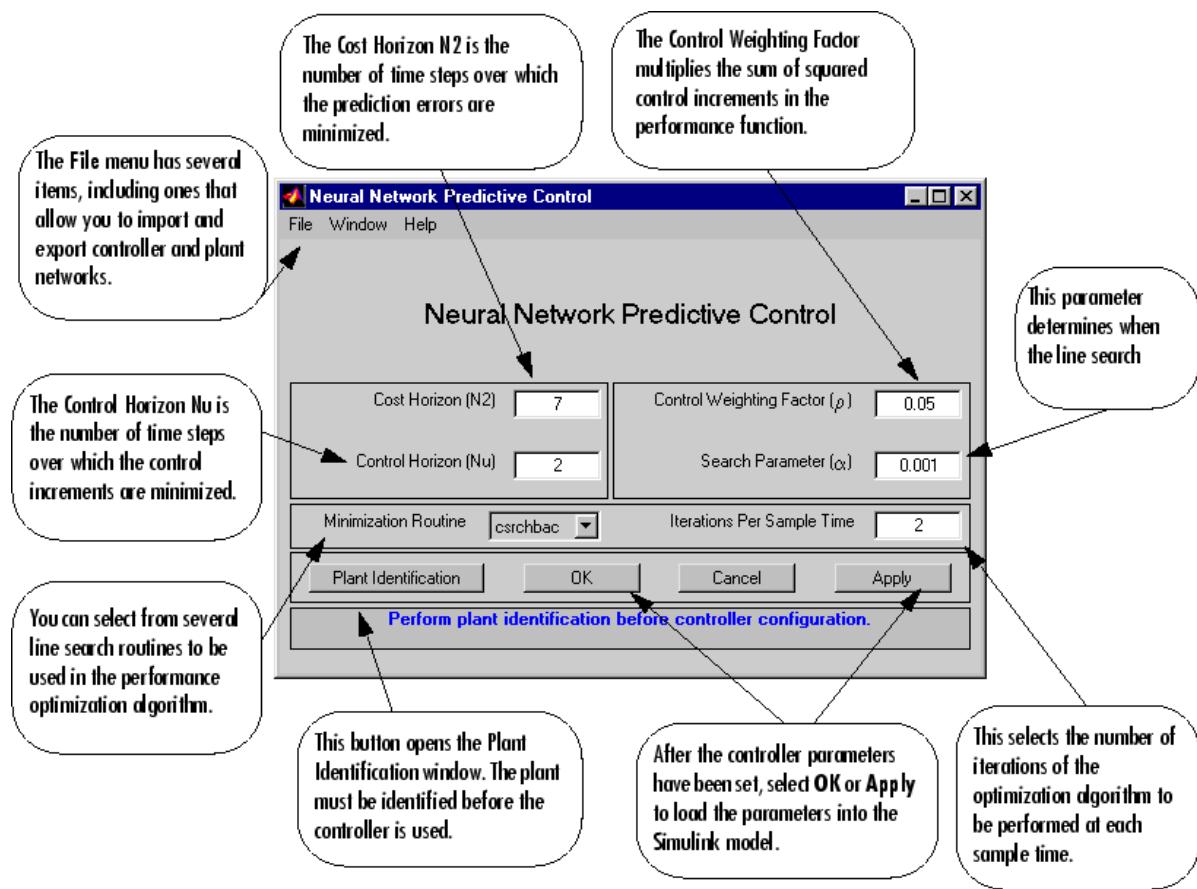
- 1** Start MATLAB.
- 2** Type `predcstr` in the MATLAB Command Window. This command opens the Simulink Editor with the following model.



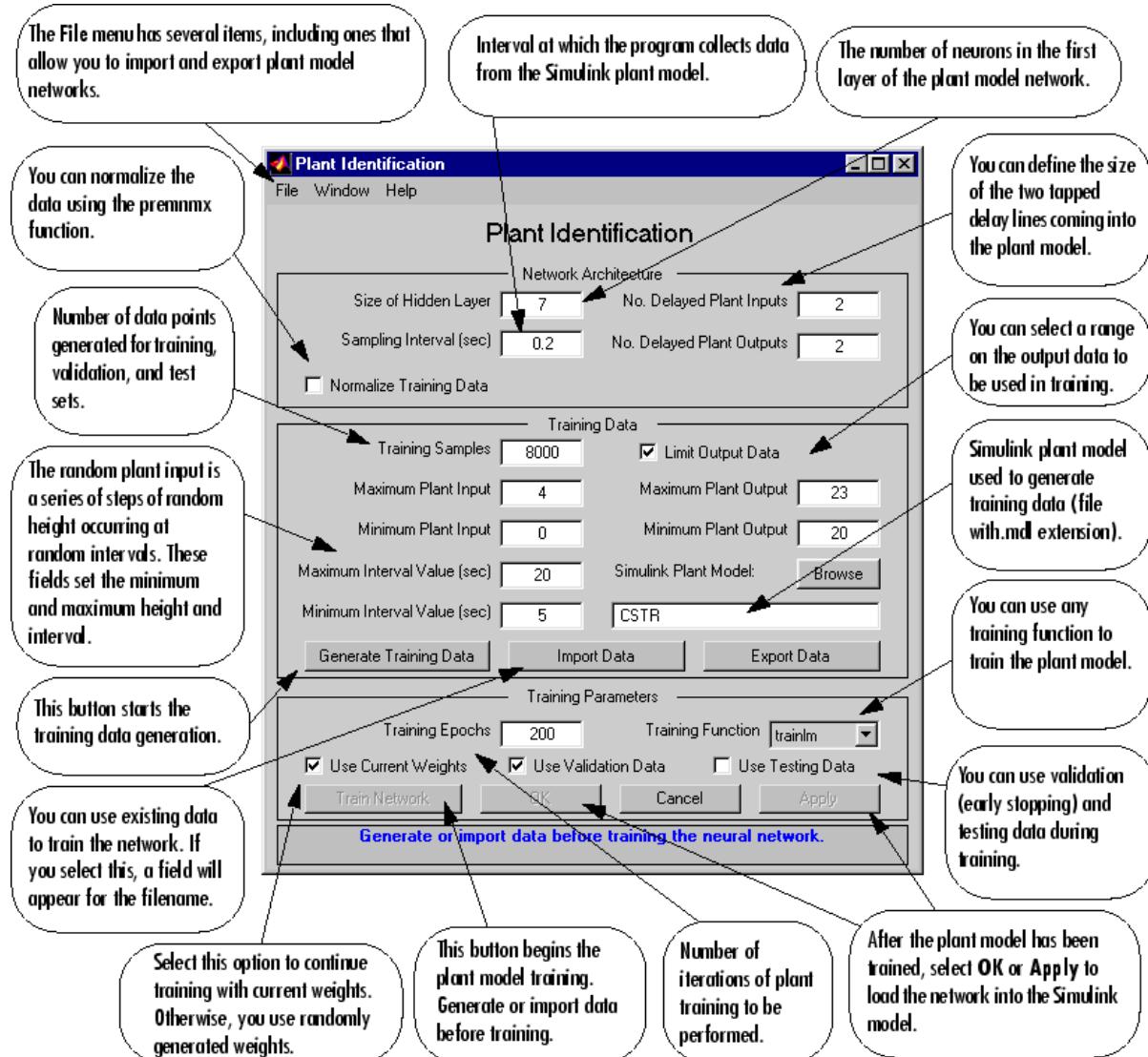
The Plant block contains the Simulink CSTR plant model. The NN Predictive Controller block signals are connected as follows:

- Control Signal is connected to the input of the Plant model.
- The Plant Output signal is connected to the Plant block output.
- The Reference is connected to the Random Reference signal.

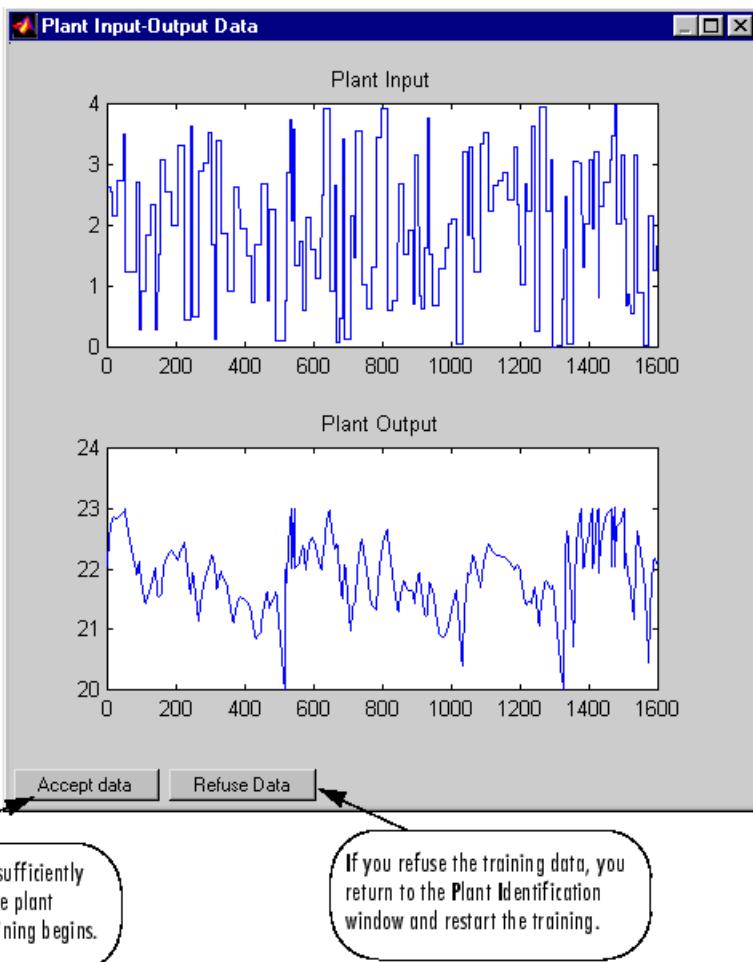
- 3** Double-click the NN Predictive Controller block. This opens the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u . (N_1 is fixed at 1.) The weighting parameter ρ , described earlier, is also defined in this window. The parameter α is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2.



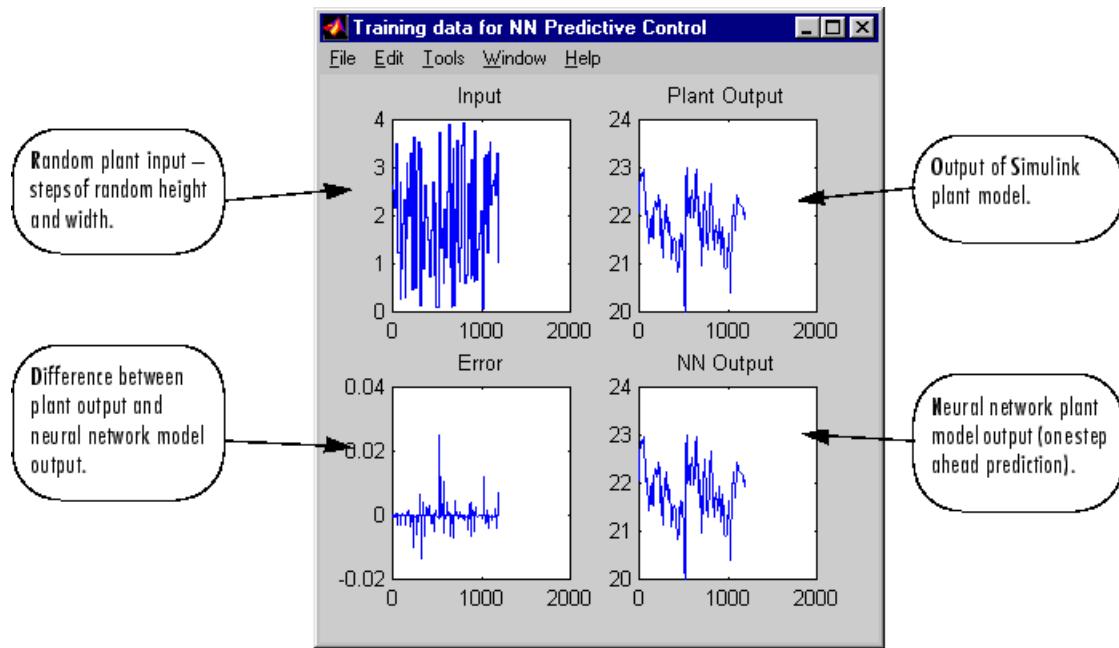
- 4 Select **Plant Identification**. This opens the following window. You must develop the neural network plant model before you can use the controller. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. You select the size of that layer, the number of delayed inputs and delayed outputs, and the training function in this window. You can select any of the training functions described in "Multilayer Shallow Neural Networks and Backpropagation Training" on page 5-2 to train the neural network plant model.



- 5 Click **Generate Training Data**. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.

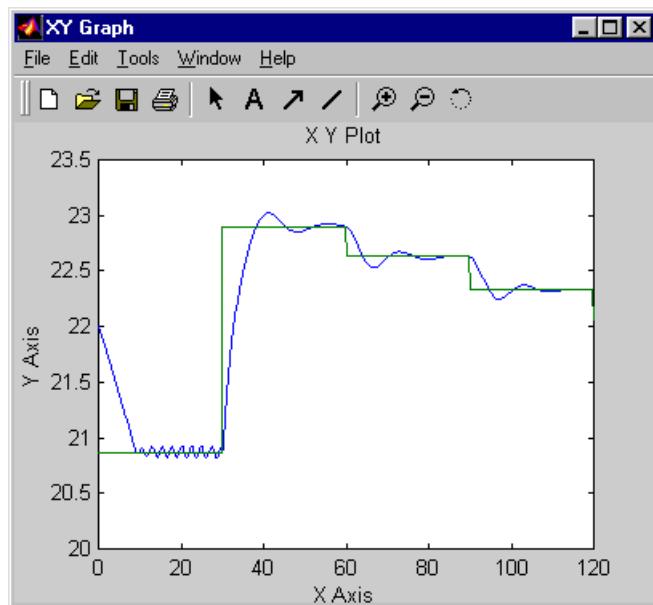


- 6 Click **Accept Data**, and then click **Train Network** in the Plant Identification window. Plant model training begins. The training proceeds according to the training algorithm (`trainlm` in this case) you selected. This is a straightforward application of batch training, as described in "Multilayer Shallow Neural Networks and Backpropagation Training" on page 5-2. After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.)



You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this example, begin the simulation, as shown in the following steps.

- 7 Select **OK** in the Plant Identification window. This loads the trained neural network plant model into the NN Predictive Controller block.
- 8 Select **OK** in the Neural Network Predictive Control window. This loads the controller parameters into the NN Predictive Controller block.
- 9 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



Design NARMA-L2 Neural Controller in Simulink

In this section...

["Identification of the NARMA-L2 Model" on page 7-14](#)

["NARMA-L2 Controller" on page 7-16](#)

["Use the NARMA-L2 Controller Block" on page 7-18](#)

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form). It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by canceling the nonlinearities. This section begins by presenting the companion form system model and showing how you can use a neural network to identify this model. Then it describes how the identified neural network model can be used to develop a controller. This is followed by an example of how to use the NARMA-L2 Control block, which is contained in the Deep Learning Toolbox blockset.

Identification of the NARMA-L2 Model

As with model predictive control, the first step in using feedback linearization (or NARMA-L2) control is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that is used to represent general discrete-time nonlinear systems is the nonlinear autoregressive-moving average (NARMA) model:

$$y(k + d) = N[y(k), y(k - 1), \dots, y(k - n + 1), u(k), u(k - 1), \dots, u(k - n + 1)]$$

where $u(k)$ is the system input, and $y(k)$ is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function N . This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory $y_r(k + d) = y_r(k + d)$, the next step is to develop a nonlinear controller of the form:

$$u(k) = G[y(k), y(k - 1), \dots, y(k - n + 1), y_r(k + d), u(k - 1), \dots, u(k - m + 1)]$$

The problem with using this controller is that if you want to train a neural network to create the function G to minimize mean square error, you need to use dynamic

backpropagation ([NaPa91 on page 14-2] or [HaJe99 on page 14-2]). This can be quite slow. One solution, proposed by Narendra and Mukhopadhyay [NaMu97 on page 14-2], is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\begin{aligned}\hat{y}(k+d) = & f[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)] \\ & + g[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)] \cdot u(k)\end{aligned}$$

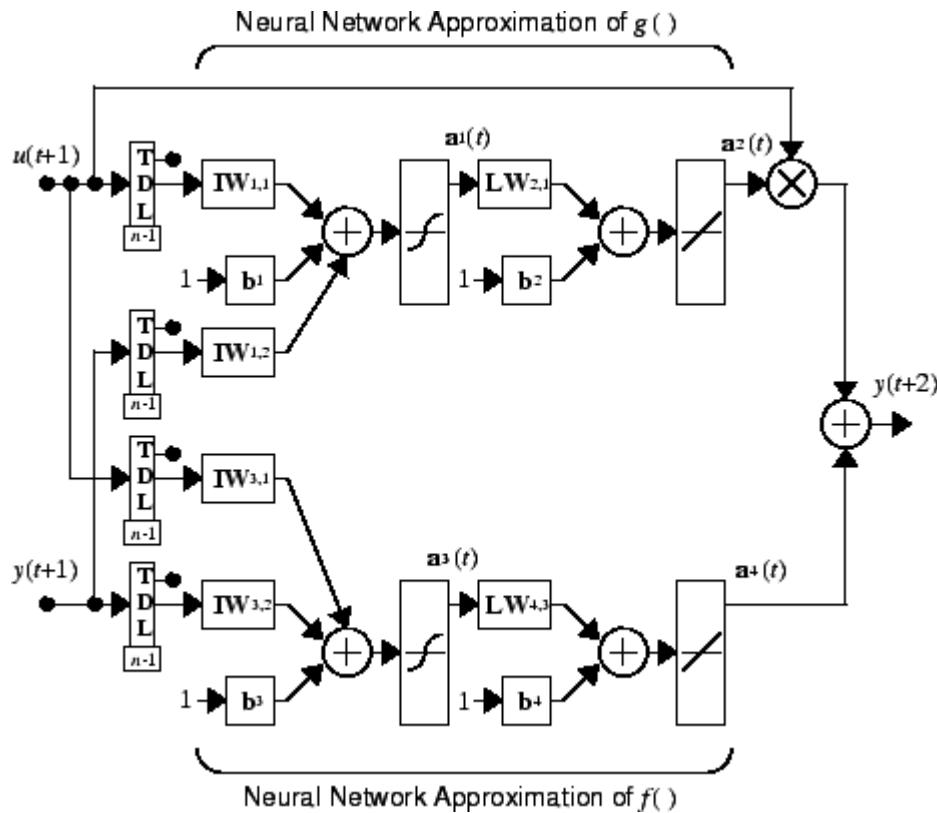
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)]}{g[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input $u(k)$ based on the output at the same time, $y(k)$. So, instead, use the model

$$\begin{aligned}y(k+d) = & f[y(k), y(k-1), \dots, y(k-n+1), u(k), u(k-1), \dots, u(k-m+1)] \\ & + g[y(k), \dots, y(k-n+1), u(k), \dots, u(k-m+1)] \cdot u(k+1)\end{aligned}$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

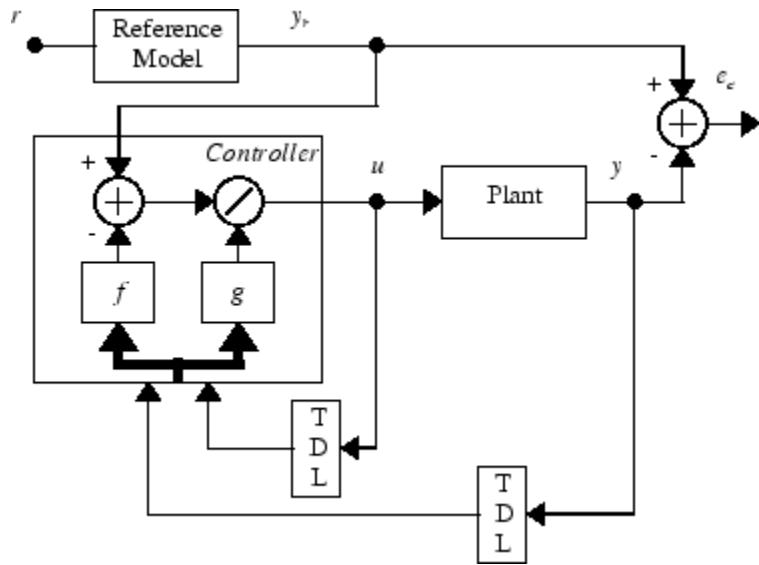


NARMA-L2 Controller

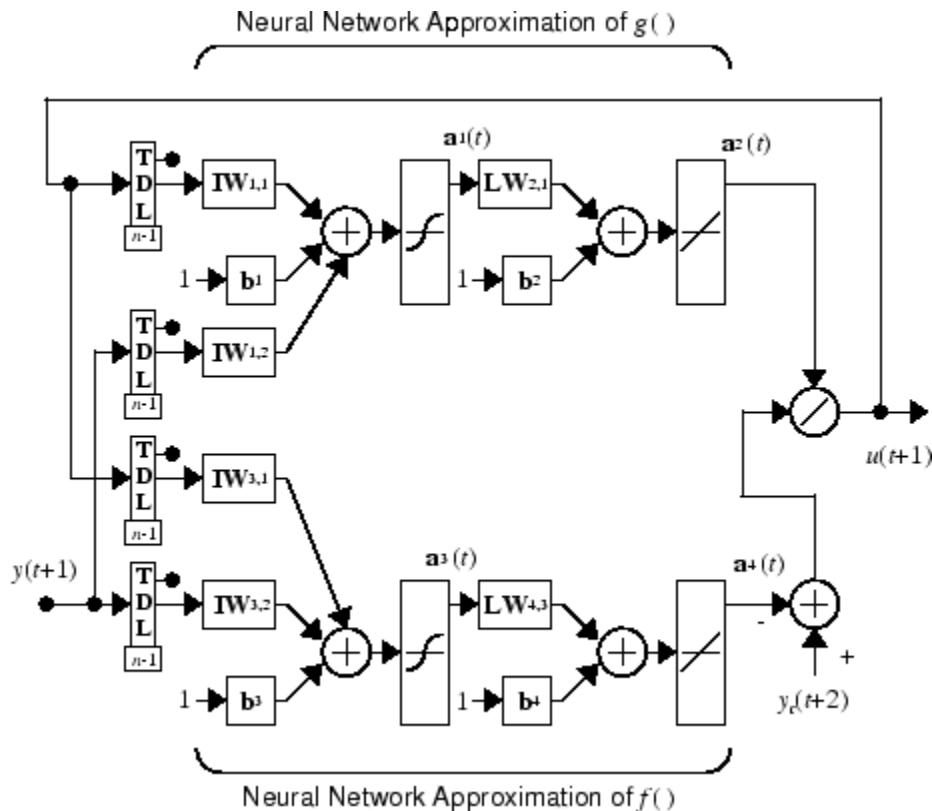
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \dots, y(k-n+1), u(k), \dots, u(k-n+1)]}{g[y(k), \dots, y(k-n+1), u(k), \dots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



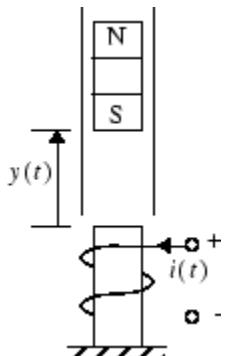
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



Use the NARMA-L2 Controller Block

This section shows how the NARMA-L2 controller is trained. The first step is to copy the NARMA-L2 Controller block from the Deep Learning Toolbox block library to the Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the NARMA-L2 controller. In this example, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as in the following figure.



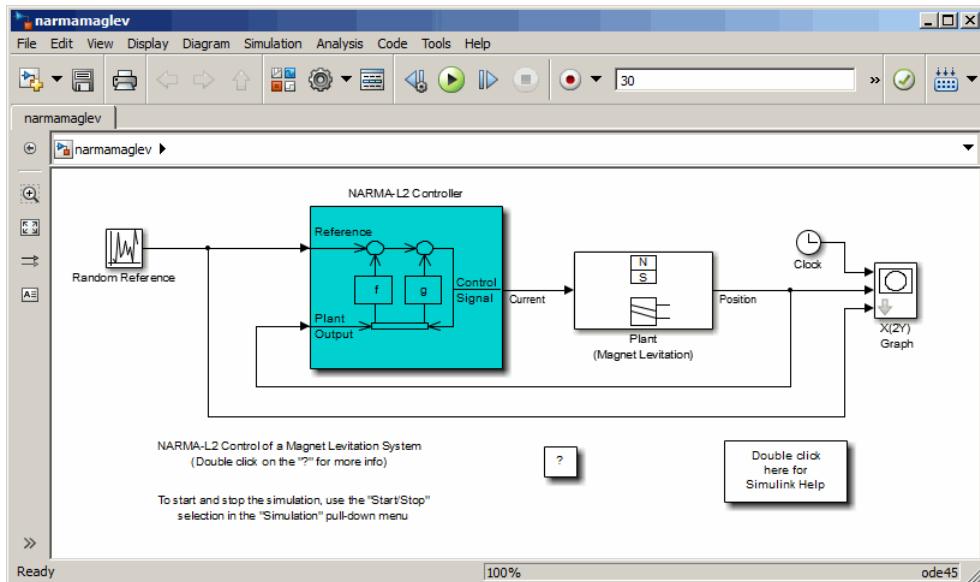
The equation of motion for this system is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M} \frac{i^2(t)}{y(t)} - \frac{\beta}{M} \frac{dy(t)}{dt}$$

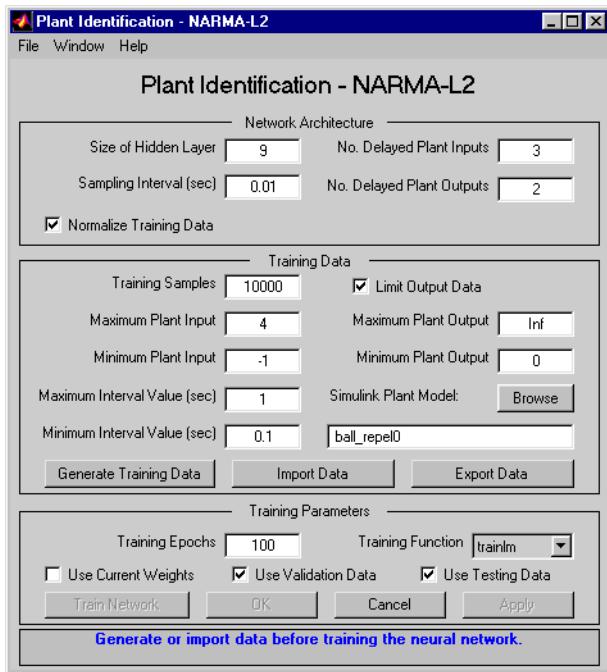
where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.

To run this example:

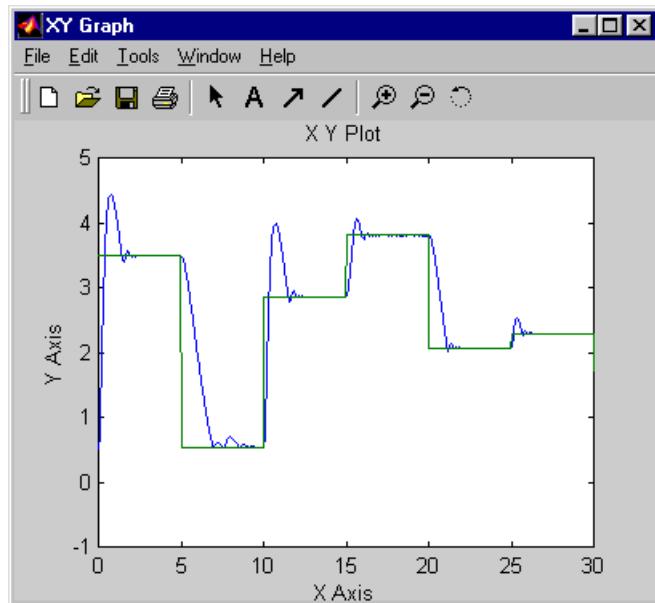
- 1** Start MATLAB.
- 2** Type `narmamaglev` in the MATLAB Command Window. This command opens the Simulink Editor with the following model. The NARMA-L2 Control block is already in the model.



- 3** Double-click the NARMA-L2 Controller block. This opens the following window. This window enables you to train the NARMA-L2 model. There is no separate window for the controller, because the controller is determined directly from the model, unlike the model predictive controller.

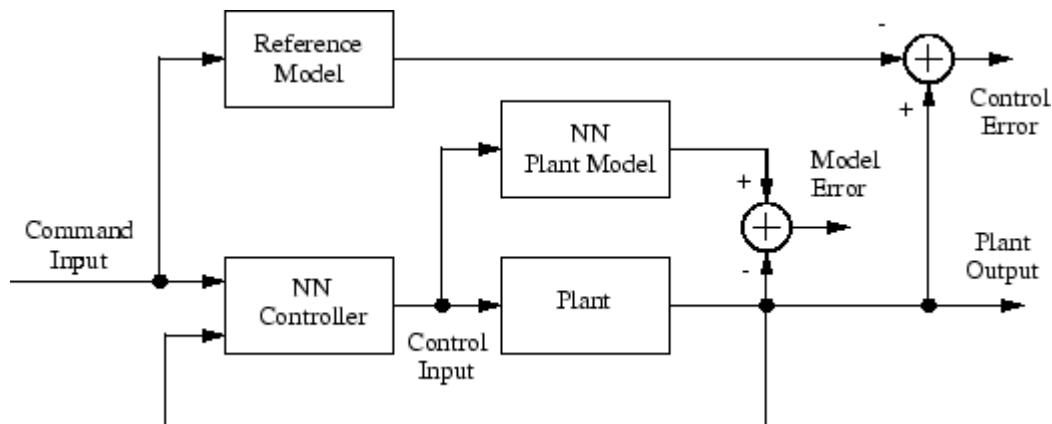


- 4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.
- 5 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



Design Model-Reference Neural Controller in Simulink

The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.



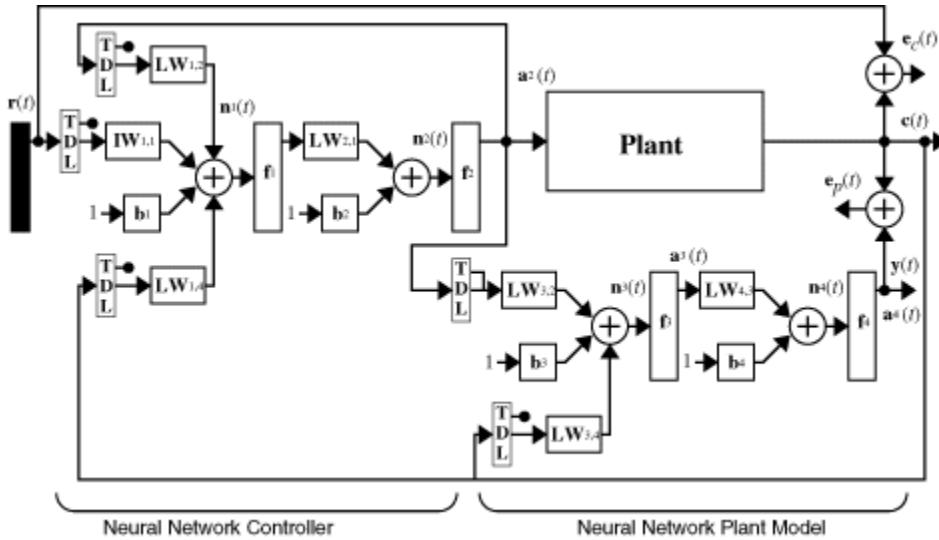
The following figure shows the details of the neural network plant model and the neural network controller as they are implemented in the Deep Learning Toolbox software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs:

- Delayed reference inputs
- Delayed controller outputs
- Delayed plant outputs

For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model:

- Delayed controller outputs
- Delayed plant outputs

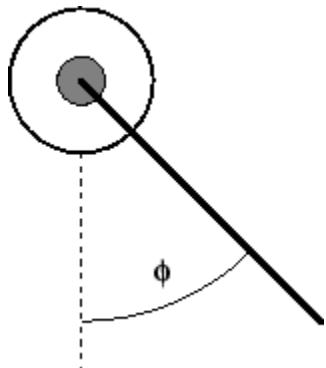
As with the controller, you can set the number of delays. The next section shows how you can set the parameters.



Use the Model Reference Controller Block

This section shows how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Deep Learning Toolbox blockset to Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the model reference controller. In this example, the objective is to control the movement of a simple, single-link robot arm, as shown in the following figure:



The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10\sin\phi - 2\frac{d\phi}{dt} + u$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor.

The objective is to train the controller so that the arm tracks the reference model

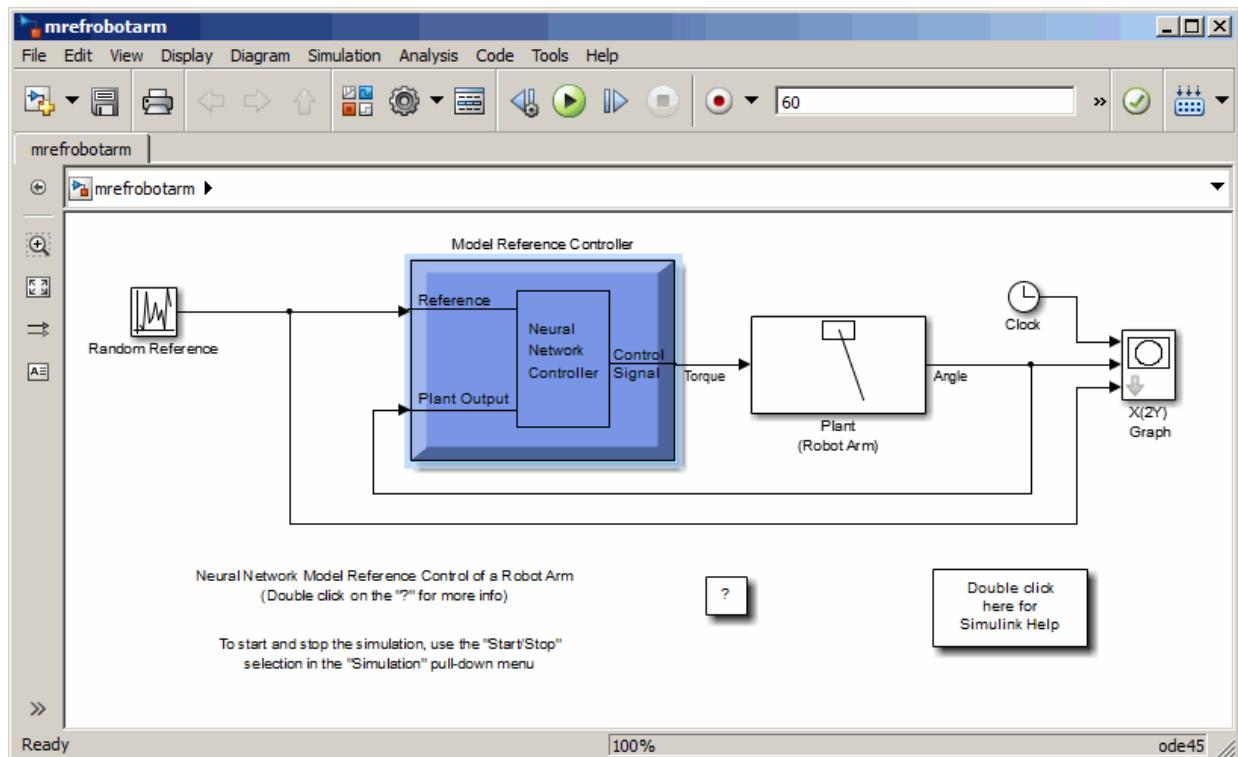
$$\frac{d^2y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

where y_r is the output of the reference model, and r is the input reference signal.

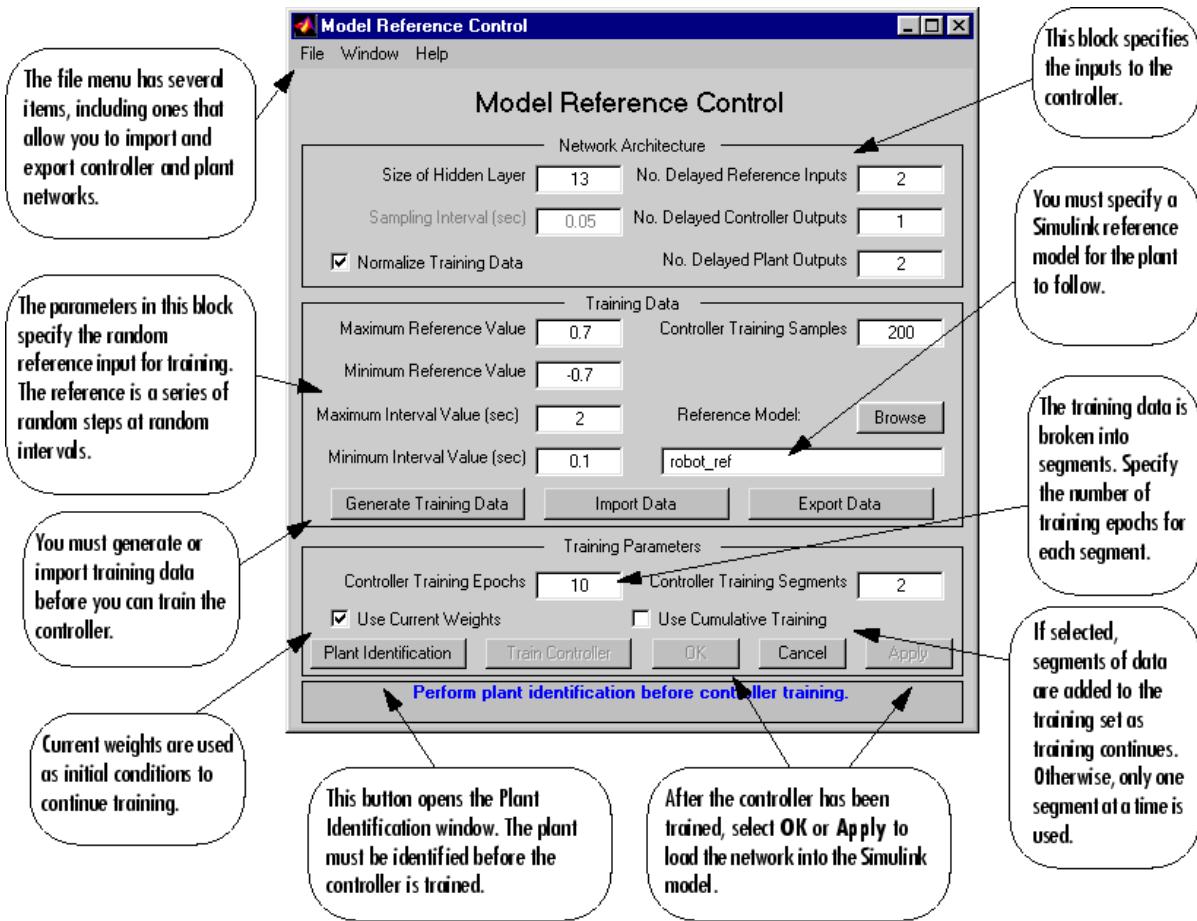
This example uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

To run this example:

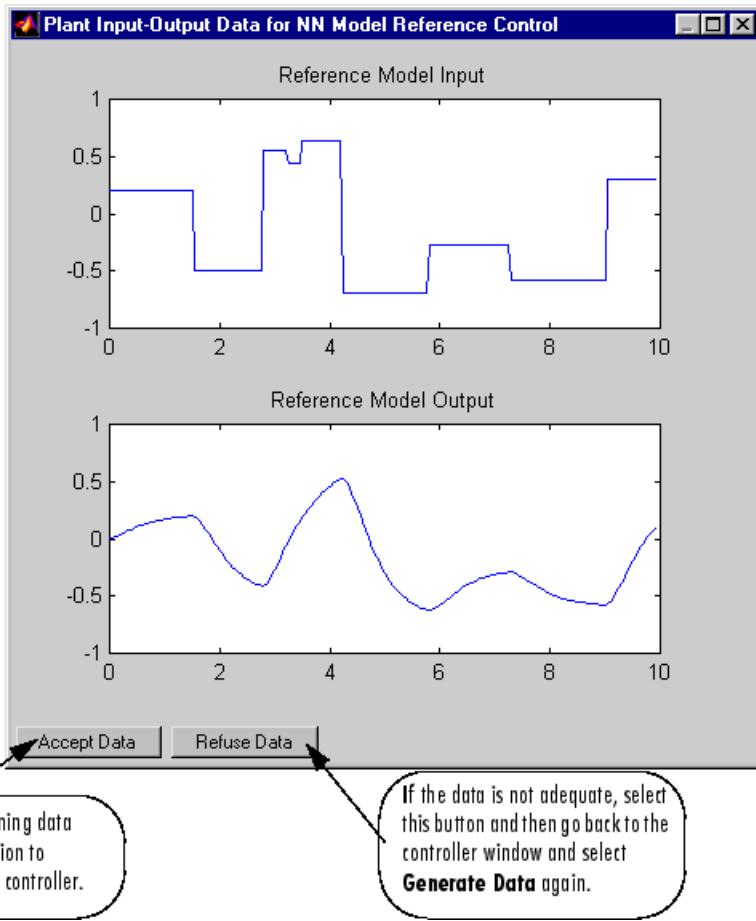
- 1** Start MATLAB.
- 2** Type `mrefrobotarm` in the MATLAB Command Window. This command opens the Simulink Editor with the Model Reference Control block already in the model.



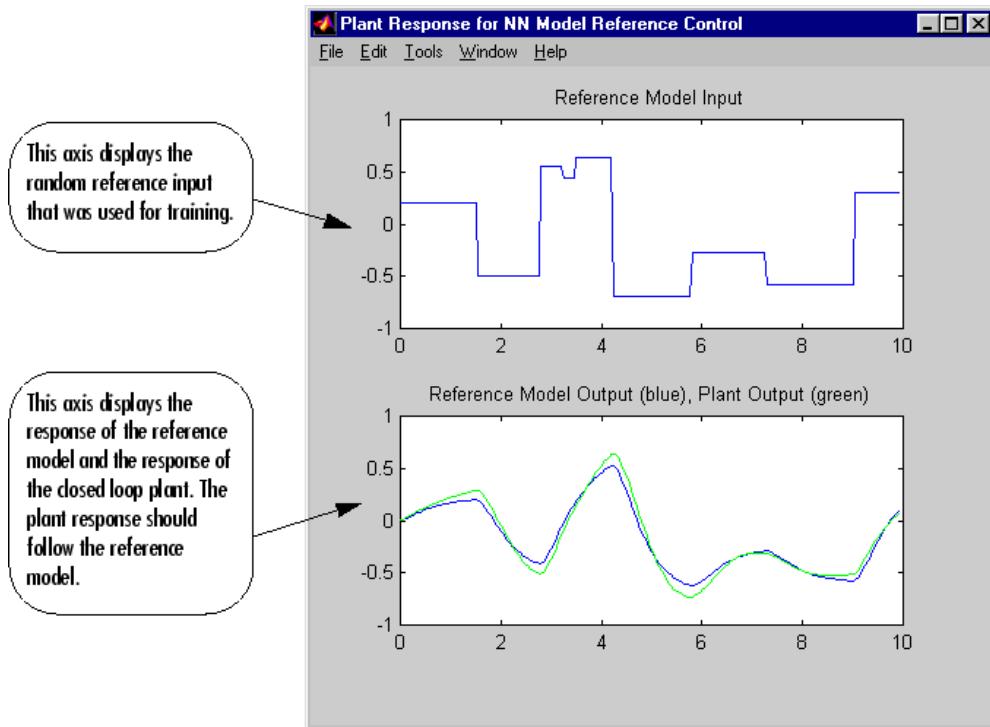
- 3 Double-click the Model Reference Control block. This opens the following window for training the model reference controller.



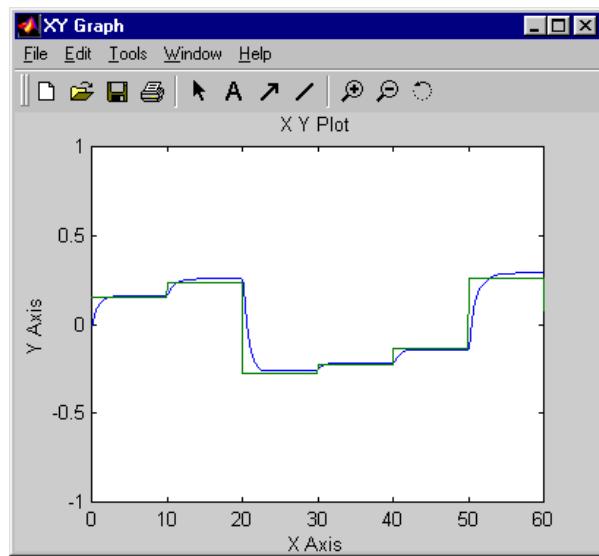
- 4 The next step would normally be to click **Plant Identification**, which opens the Plant Identification window. You would then train the plant model. Because the Plant Identification window is identical to the one used with the previous controllers, that process is omitted here.
- 5 Click **Generate Training Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.



- 6** Click **Accept Data**. Return to the Model Reference Control window and click **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues, one segment at a time, until the entire training set has been presented to the network. Controller training can be significantly more time consuming than plant model training. This is because the controller must be trained using *dynamic* backpropagation (see [HaJe99 on page 14-2]). After the training is complete, the response of the resulting closed loop system is displayed, as in the following figure.



- 7 Go back to the Model Reference Control window. If the performance of the controller is not accurate, then you can select **Train Controller** again, which continues the controller training with the same data set. If you would like to use a new data set to continue training, select **Generate Data** or **Import Data** before you select **Train Controller**. (Be sure that **Use Current Weights** is selected if you want to continue training with the same weights.) It might also be necessary to retrain the plant model. If the plant model is not accurate, it can affect the controller training. For this example, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.
- 8 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



Import-Export Neural Network Simulink Control Systems

In this section...

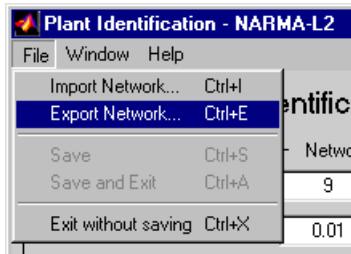
["Import and Export Networks" on page 7-31](#)

["Import and Export Training Data" on page 7-35](#)

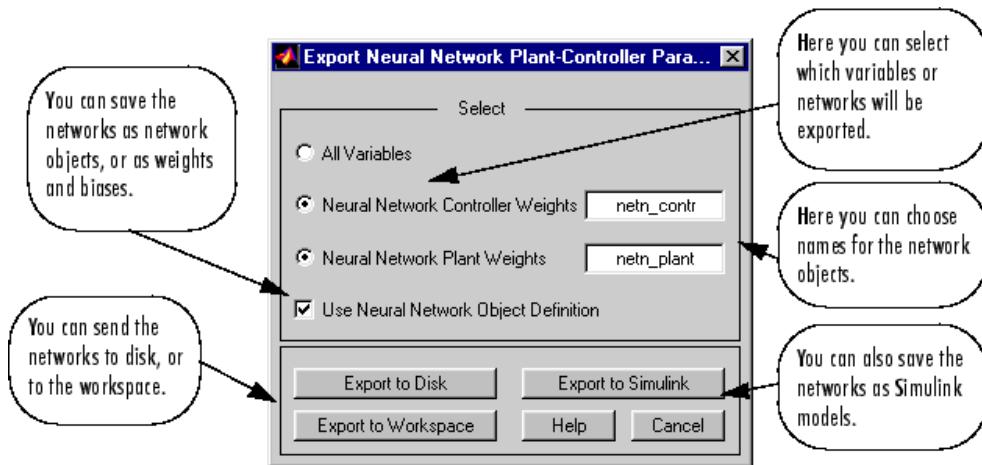
Import and Export Networks

The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you might want to transfer the networks into other applications, or you might want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following example leads you through the export and import processes. (The NARMA-L2 window is used for this example, but the same procedure applies to all the controllers.)

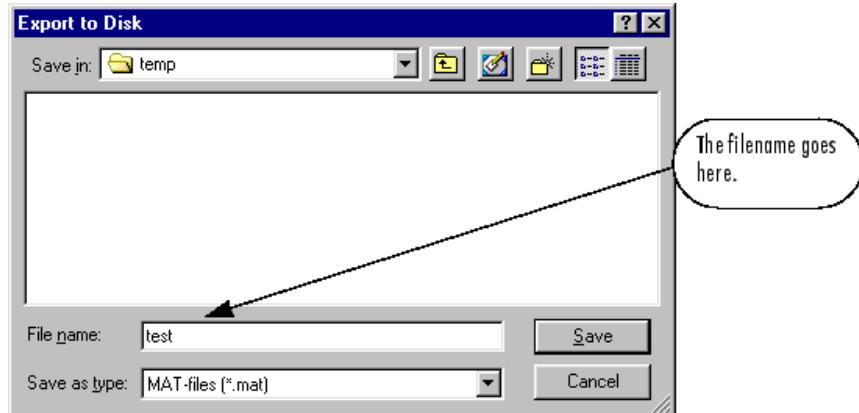
- 1 Repeat the first three steps of the NARMA-L2 example in “Use the NARMA-L2 Controller Block” on page 7-18. The NARMA-L2 Plant Identification window should now be open.
- 2 Select **File > Export Network**, as shown below.



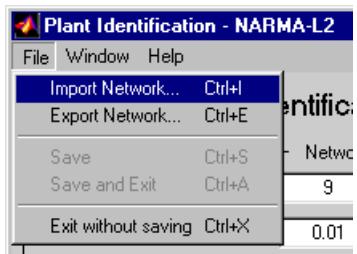
This opens the following window.



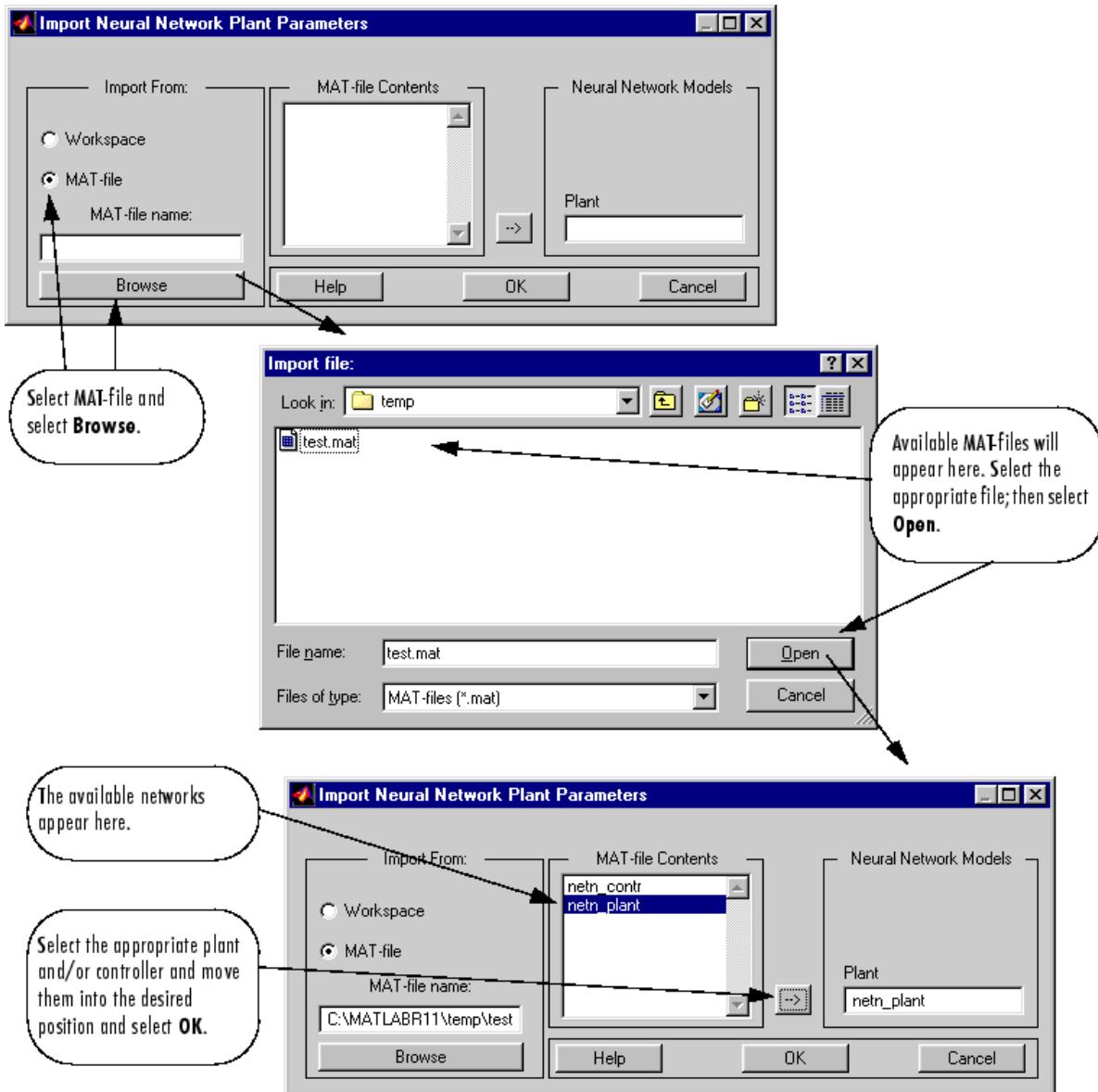
- 3 Select **Export to Disk**. The following window opens. Enter the file name **test** in the box, and select **Save**. This saves the controller and plant networks to disk.



- 4 Retrieve that data with the **Import** menu option. Select **File > Import Network**, as in the following figure.



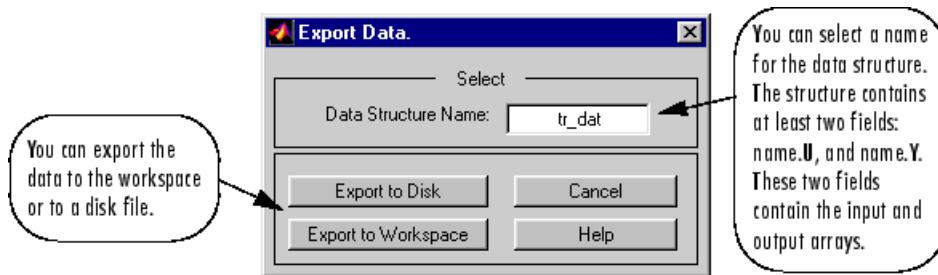
This causes the following window to appear. Follow the steps indicated to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by clicking **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you do not need to import both networks.



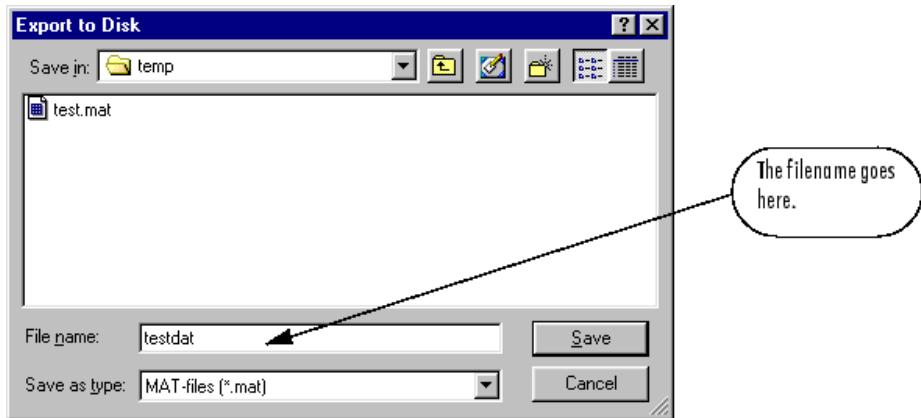
Import and Export Training Data

The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You might want to save the training data to the workspace or to a disk file so that you can load it again at a later time. You might also want to combine data sets manually and then load them back into the training window. You can do this by using the **Import** and **Export** buttons. The following example leads you through the import and export processes. (The NN Predictive Control window is used for this example, but the same procedure applies to all the controllers.)

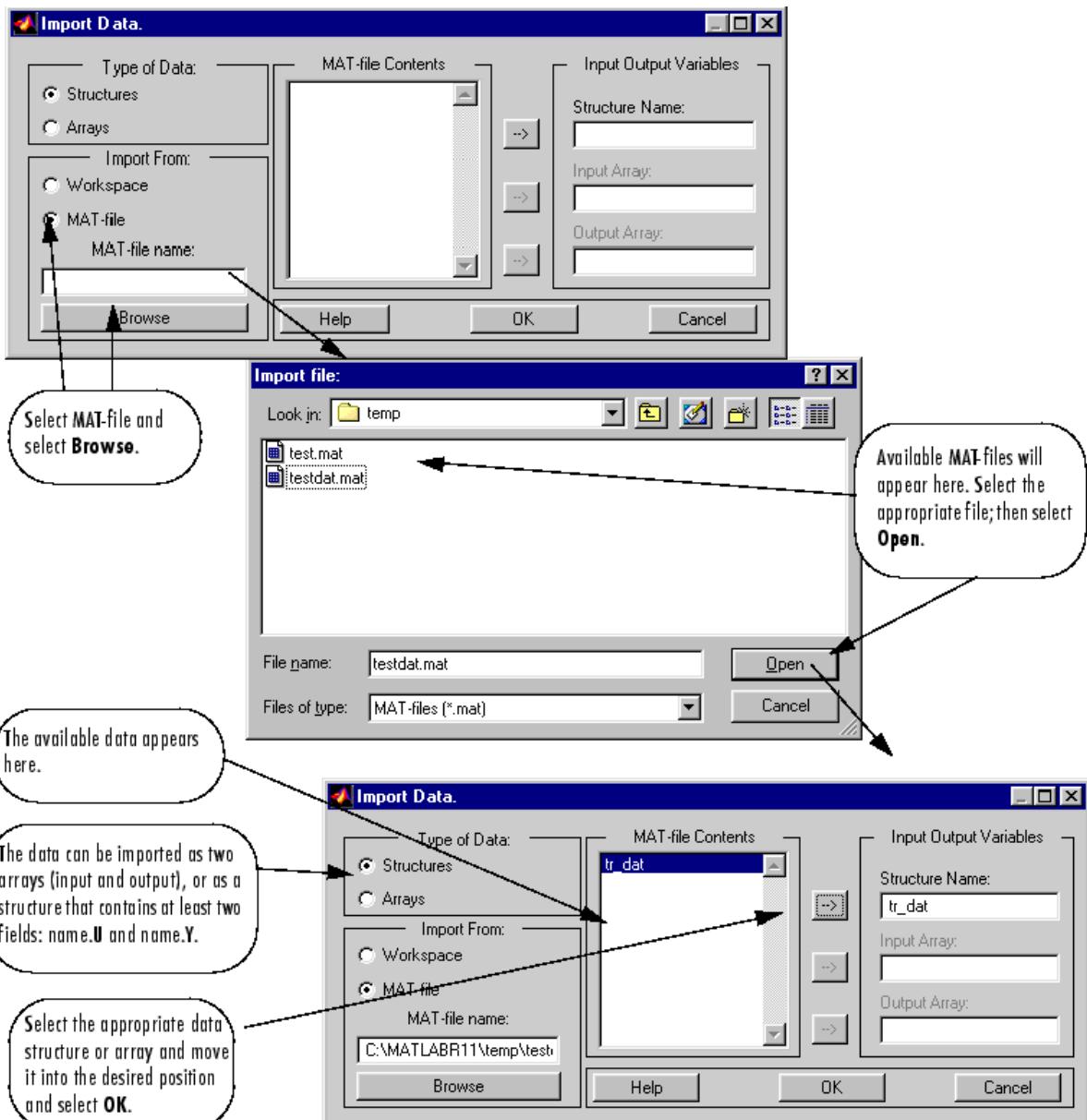
- 1 Repeat the first five steps of the NN Predictive Control example in “Use the Neural Network Predictive Controller Block” on page 7-6. Then select **Accept Data**. The Plant Identification window should then be open, and the **Import** and **Export** buttons should be active.
- 2 Click **Export** to open the following window.



- 3 Click **Export to Disk**. The following window opens. Enter the filename `testdat` in the box, and select **Save**. This saves the training data structure to disk.



- 4 Now retrieve the data with the import command. Click **Import** in the Plant Identification window to open the following window. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.



Radial Basis Neural Networks

- “Introduction to Radial Basis Neural Networks” on page 8-2
- “Radial Basis Neural Networks” on page 8-3
- “Probabilistic Neural Networks” on page 8-9
- “Generalized Regression Neural Networks” on page 8-12

Introduction to Radial Basis Neural Networks

Radial basis networks can require more neurons than standard feedforward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feedforward networks. They work best when many training vectors are available.

You might want to consult the following paper on this subject: Chen, S., C.F.N. Cowan, and P.M. Grant, "Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, March 1991, pp. 302-309.

This topic discusses two variants of radial basis networks, generalized regression networks (GRNN) and probabilistic neural networks (PNN). You can read about them in P.D. Wasserman, *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993, on pp. 155-61 and pp. 35-55, respectively.

Important Radial Basis Functions

Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

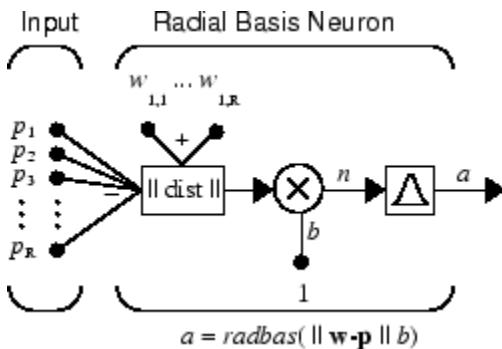
Radial Basis Neural Networks

In this section...

- “Neuron Model” on page 8-3
- “Network Architecture” on page 8-4
- “Exact Design (newrbe)” on page 8-6
- “More Efficient Design (newrb)” on page 8-7
- “Examples” on page 8-8

Neuron Model

Here is a radial basis network with R inputs.

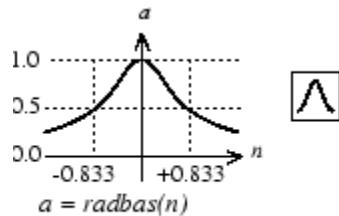


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector w and the input vector p , multiplied by the bias b . (The $\| \text{dist} \|$ box in this figure accepts the input vector p and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is

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

Here is a plot of the `radbas` transfer function.



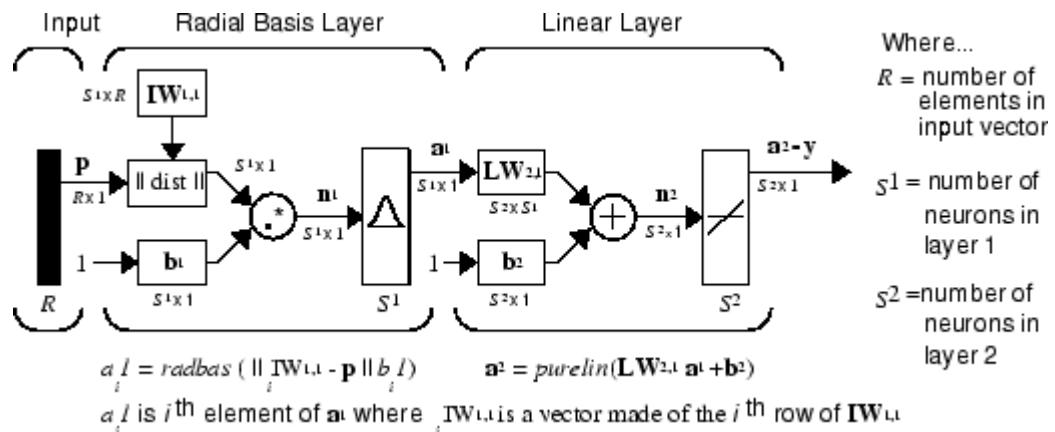
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between \mathbf{w} and \mathbf{p} decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input \mathbf{p} is identical to its weight vector \mathbf{w} .

The bias b allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector \mathbf{p} at vector distance of 8.326 ($0.8326/b$) from its weight vector \mathbf{w} .

Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S^1 neurons, and an output linear layer of S^2 neurons.



The $\| \text{dist} \|$ box in this figure accepts the input vector \mathbf{p} and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the distances between the input vector and vectors $i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector **b**¹ and the output of `|| dist ||` are combined with the MATLAB operation `.*`, which does element-by-element multiplication.

The output of the first layer for a feedforward network `net` can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All the details of designing this network are built into design functions `newrbe` and `newrb`, and you can obtain their outputs with `sim`.

You can understand how this network behaves by following an input vector **p** through the network to the output **a**². If you present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1, its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0s (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with `netprod`. Each neuron's output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input is `spread`, its net input is `sqrt(-log(.5))` (or 0.8326), therefore its output is 0.5.

Exact Design (newrbe)

You can design radial basis networks with the function `newrbe`. This function can produce a network with zero error on training vectors. It is called in the following way:

```
net = newrbe(P,T,SPREAD)
```

The function `newrbe` takes matrices of input vectors P and target vectors T , and a spread constant $SPREAD$ for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly T when the inputs are P .

This function `newrbe` creates as many `radbas` neurons as there are input vectors in P , and sets the first-layer weights to P' . Thus, there is a layer of `radbas` neurons in which each neuron acts as a detector for a different input vector. If there are Q input vectors, then there will be Q neurons.

Each bias in the first layer is set to $0.8326/SPREAD$. This gives radial basis functions that cross 0.5 at weighted inputs of $+/- SPREAD$. This determines the width of an area in the input space to which each neuron responds. If $SPREAD$ is 4, then each `radbas` neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. $SPREAD$ should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights $IW^{2,1}$ (or in code, `IW{2,1}`) and biases b^2 (or in code, `b{2}`) are found by simulating the first-layer outputs a^1 (`A{1}`), and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones(1,Q)] = T
```

You know the inputs to the second layer (`A{1}`) and the target (T), and the layer is linear. You can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

```
Wb = T/[A{1}; ones(1,Q)]
```

Here `Wb` contains both weights and biases, with the biases in the last column. The sum-squared error is always 0, as explained below.

There is a problem with C constraints (input/target pairs) and each neuron has $C + 1$ variables (the C weights from the C `radbas` neurons, and a bias). A linear problem with C constraints and more than C variables has an infinite number of zero error solutions.

Thus, `newrbe` creates a network with zero error on training vectors. The only condition required is to make sure that `SPREAD` is large enough that the active input regions of the `radbas` neurons overlap enough so that several `radbas` neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in the design. (However, `SPREAD` should not be so large that each neuron is effectively responding in the same large area of the input space.)

The drawback to `newrbe` is that it produces a network with as many hidden neurons as there are input vectors. For this reason, `newrbe` does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

More Efficient Design (`newrb`)

The function `newrb` iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is

```
net = newrb(P,T,GOAL,SPREAD)
```

The function `newrb` takes matrices of input and target vectors `P` and `T`, and design parameters `GOAL` and `SPREAD`, and returns the desired network.

The design method of `newrb` is similar to that of `newrbe`. The difference is that `newrb` creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most is used to create a `radbas` neuron. The error of the new network is checked, and if low enough `newrb` is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met or the maximum number of neurons is reached.

As with `newrbe`, it is important that the spread parameter be large enough that the `radbas` neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feedforward network? Radial basis networks, even when designed efficiently with `newrbe`, tend to have many times more neurons than a comparable feedforward network with `tansig` or `logsig` neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while `radbas` neurons only respond to relatively small regions of the input space. The

result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more `radbas` neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons' being used, as can be seen in the next example.

Examples

The example `demorb1` shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Examples `demorb3` and `demorb4` examine how the spread constant affects the design process for radial basis networks.

In `demorb3`, a radial basis network is designed to solve the same problem as in `demorb1`. However, this time the spread constant used is 0.01. Thus, each radial basis neuron returns 0.5 or lower for any input vector with a distance of 0.01 or more from its weight vector.

Because the training inputs occur at intervals of 0.1, no two radial basis neurons have a strong output for any given input.

`demorb3` showed that having too small a spread constant can result in a solution that does not generalize from the input/target vectors used in the design. Example `demorb4` shows the opposite problem. If the spread constant is large enough, the radial basis neurons will output large values (near 1.0) for all the inputs used to design the network.

If all the radial basis neurons always output 1, any information presented to the network becomes lost. No matter what the input, the second layer outputs 1's. The function `newrb` will attempt to find a network, but cannot because of numerical problems that arise in this situation.

The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the leftmost and rightmost inputs.

Probabilistic Neural Networks

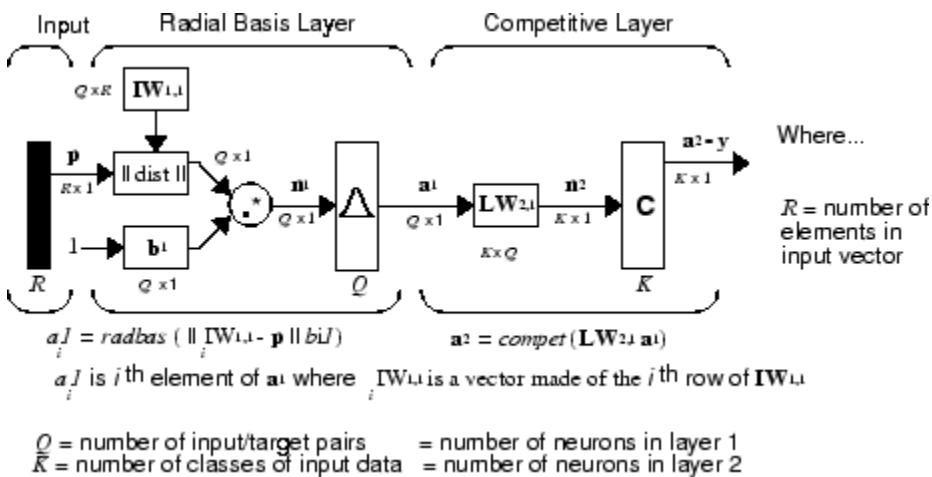
In this section...

["Network Architecture" on page 8-9](#)

["Design \(newpnn\)" on page 8-10](#)

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these elements is 1 and the rest are 0. Thus, each input vector is associated with one of K classes.

The first-layer input weights, $\mathbf{IW}^{1,1}$ (`net.IW{1,1}`), are set to the transpose of the matrix formed from the Q training pairs, \mathbf{P}' . When an input is presented, the $\|\mathbf{dist}\|$ box

produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent to the `radbas` transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector \mathbf{a}^1 . If an input is close to several training vectors of a single class, it is represented by several elements of \mathbf{a}^1 that are close to 1.

The second-layer weights, $LW^{1,2}$ (`net.LW{2,1}`), are set to the matrix \mathbf{T} of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0s elsewhere. (Use function `ind2vec` to create the proper vectors.) The multiplication $\mathbf{T}\mathbf{a}^1$ sums the elements of \mathbf{a}^1 due to each of the K input classes. Finally, the second-layer transfer function, `compet`, produces a 1 corresponding to the largest element of \mathbf{n}^2 , and 0s elsewhere. Thus, the network classifies the input vector into a specific K class because that class has the maximum probability of being correct.

Design (newpnn)

You can use the function `newpnn` to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

```
P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]'
```

which yields

```
P =
    0      1      0      1      3      4      4
    0      1      3      4      1      1      3
Tc = [1 1 2 2 3 3 3]
```

which yields

```
Tc =
    1      1      2      2      3      3      3
```

You need a target matrix with 1s in the right places. You can get it with the function `ind2vec`. It gives a matrix with 0s except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
(1,1)      1
(1,2)      1
```

(2,3)	1
(2,4)	1
(3,5)	1
(3,6)	1
(3,7)	1

Now you can create a network and simulate it, using the input P to make sure that it does produce the correct classifications. Use the function `vec2ind` to convert the output Y into a row Yc to make the classifications clear.

```
net = newpnn(P,T);
Y = sim(net,P);
Yc = vec2ind(Y)
```

This produces

```
Yc =
    1     1     2     2     3     3     3
```

You might try classifying vectors other than those that were used to design the network. Try to classify the vectors shown below in $P2$.

```
P2 = [1 4;0 1;5 2]';
P2 =
    1     0     5
    4     1     2
```

Can you guess how these vectors will be classified? If you run the simulation and plot the vectors as before, you get

```
Yc =
    2     1     3
```

These results look good, for these test vectors were quite close to members of classes 2, 1, and 3, respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

You might want to try `demopnn1`. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

Generalized Regression Neural Networks

In this section...

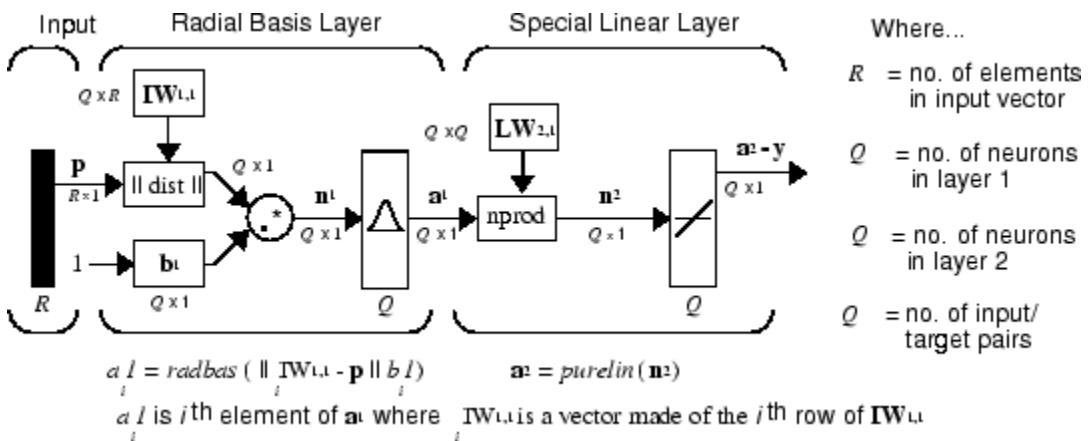
"Network Architecture" on page 8-12

"Design (newgrnn)" on page 8-14

Network Architecture

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



Here the **nprod** box shown above (code function `normprod`) produces S^2 elements in vector \mathbf{n}^2 . Each element is the dot product of a row of $\mathbf{LW}_{2,1}$ and the input vector \mathbf{a}^1 , all normalized by the sum of the elements of \mathbf{a}^1 . For instance, suppose that

```
LW{2,1}=[1 -2;3 4;5 6];
a{1}=[0.7;0.3];
```

Then

```
aout = normprod(LW{2,1},a{1})
aout =
```

```

0.1000
3.3000
5.3000

```

The first layer is just like that for newrbe networks. It has as many neurons as there are input/ target vectors in \mathbf{P} . Specifically, the first-layer weights are set to \mathbf{P}' . The bias \mathbf{b}^1 is set to a column vector of $0.8326/\text{SPREAD}$. The user chooses SPREAD, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the newrbe radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the product of its weighted input with its bias, calculated with `netprod`. Each neuron's output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input will be `spread`, and its net input will be $\sqrt{-\log(0.5)}$ (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here `LW{2,1}` is set to `T`.

Suppose you have an input vector \mathbf{p} close to \mathbf{p}_i , one of the input vectors among the input vector/target pairs used in designing layer 1 weights. This input \mathbf{p} produces a layer 1 \mathbf{a}^i output close to 1. This leads to a layer 2 output close to \mathbf{t}_i , one of the targets used to form layer 2 weights.

A larger `spread` leads to a large area around the input vector where layer 1 neurons will respond with significant outputs. Therefore if `spread` is small the radial basis function is very steep, so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network tends to respond with the target vector associated with the nearest design input vector.

As `spread` becomes larger the radial basis function's slope becomes smoother and several neurons can respond to an input vector. The network then acts as if it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As `spread` becomes larger more and more neurons contribute to the average, with the result that the network function becomes smoother.

Design (newgrnn)

You can use the function `newgrnn` to create a GRNN. For instance, suppose that three input and three target vectors are defined as

```
P = [4 5 6];
T = [1.5 3.6 6.7];
```

You can now obtain a GRNN with

```
net = newgrnn(P,T);
```

and simulate it with

```
P = 4.5;
v = sim(net,P);
```

You might want to try `demogrnn1`. It shows how to approximate a function with a GRNN.

Function	Description
<code>compet</code>	Competitive transfer function.
<code>dist</code>	Euclidean distance weight function.
<code>dotprod</code>	Dot product weight function.
<code>ind2vec</code>	Convert indices to vectors.
<code>negdist</code>	Negative Euclidean distance weight function.
<code>netprod</code>	Product net input function.
<code>newgrnn</code>	Design a generalized regression neural network.
<code>newpnn</code>	Design a probabilistic neural network.
<code>newrb</code>	Design a radial basis network.
<code>newrbe</code>	Design an exact radial basis network.
<code>normprod</code>	Normalized dot product weight function.
<code>radbas</code>	Radial basis transfer function.
<code>vec2ind</code>	Convert vectors to indices.

Self-Organizing and Learning Vector Quantization Networks

- “Introduction to Self-Organizing and LVQ” on page 9-2
- “Cluster with a Competitive Neural Network” on page 9-3
- “Cluster with Self-Organizing Map Neural Network” on page 9-9
- “Learning Vector Quantization (LVQ) Neural Networks” on page 9-34

Introduction to Self-Organizing and LVQ

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors. Self-organizing maps do not have target vectors, since their purpose is to divide the input vectors into clusters of similar vectors. There is no desired output for these types of networks.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner (with target outputs). A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

LVQ networks, on the other hand, learn to classify input vectors into target classes chosen by the user.

You might consult the following reference: Kohonen, T., *Self-Organization and Associative Memory*, 2nd Edition, Berlin: Springer-Verlag, 1987.

Important Self-Organizing and LVQ Functions

You can create competitive layers and self-organizing maps with `competlayer` and `selforgmap`, respectively.

You can create an LVQ network with the function `lvqnet`.

Cluster with a Competitive Neural Network

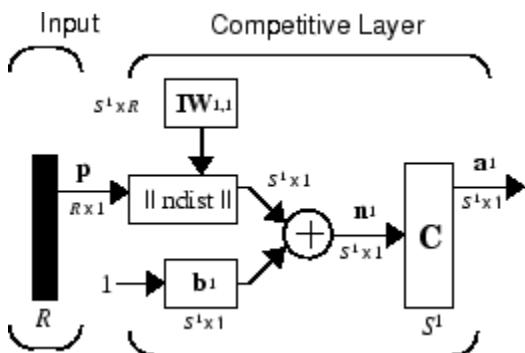
In this section...

- “Architecture” on page 9-3
- “Create a Competitive Neural Network” on page 9-4
- “Kohonen Learning Rule (learnk)” on page 9-5
- “Bias Learning Rule (learncon)” on page 9-5
- “Training” on page 9-6
- “Graphical Example” on page 9-8

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

Architecture

The architecture for a competitive network is shown below.



The $\|\mathbf{dist}\|$ box in this figure accepts the input vector \mathbf{p} and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the negative of the distances between the input vector and vectors $\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

Compute the net input \mathbf{n}^1 of a competitive layer by finding the negative distance between input vector \mathbf{p} and the weight vectors and adding the biases \mathbf{b} . If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector \mathbf{p} equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input \mathbf{n}^1 . The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in “Bias Learning Rule (learncon)” on page 9-5.

Create a Competitive Neural Network

You can create a competitive neural network with the function `competlayer`. A simple example shows how this works.

Suppose you want to divide the following four two-element vectors into two classes.

```
p = [.1 .8 .1 .9; .2 .9 .1 .8]
```

```
p =
    0.1000    0.8000    0.1000    0.9000
    0.2000    0.9000    0.1000    0.8000
```

There are two vectors near the origin and two vectors near (1,1).

First, create a two-neuron competitive layer.:

```
net = competlayer(2);
```

Now you have a network, but you need to train it to do the classification job.

The first time the network is trained, its weights will initialized to the centers of the input ranges with the function `midpoint`. You can check see these initial values using the number of neurons and the input data:

```
wts = midpoint(2,p)
```

```
wts =
    0.5000    0.5000
    0.5000    0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs.

The initial biases are computed by `initcon`, which gives

```

biases = initcon(2)

biases =
5.4366
5.4366

```

Recall that each neuron competes to respond to an input vector \mathbf{p} . If the biases are all 0, the neuron whose weight vector is closest to \mathbf{p} gets the highest net input and, therefore, wins the competition, and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

Kohonen Learning Rule (`learnk`)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the i th neuron wins, the elements of the i th row of the input weight matrix are adjusted as shown below.

$${}_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1))$$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

Bias Learning Rule (`learncon`)

One of the limitations of competitive networks is that some neurons might not always be *allocated*. In other words, some neuron weight vectors might start out far from any input vectors and never win the competition, no matter how long the training is continued. The

result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this, use biases to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently active neurons become smaller, and biases of infrequently active neurons become larger.

As the biases of infrequently active neurons increase, the input space to which those neurons respond increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually, the neuron responds to the same number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias eventually becomes large enough so that it can win. When this happens, it moves toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk` to make sure that the running average is accurate.

Training

Now train the network for 500 epochs. You can use either `train` or `adapt`.

```
net.trainParam.epochs = 500;  
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainru`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

```
ans =
trainru
```

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated after each individual presentation.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

```
a = sim(net,p);
ac = vec2ind(a)

ac =
    1     2     1     2
```

You see that the network is trained to classify the input vectors into two groups, those near the origin, class 1, and those near (1,1), class 2.

It might be interesting to look at the final weights and biases.

```
net.IW{1,1}

ans =
    0.1000    0.1500
    0.8500    0.8500

net.b{1}

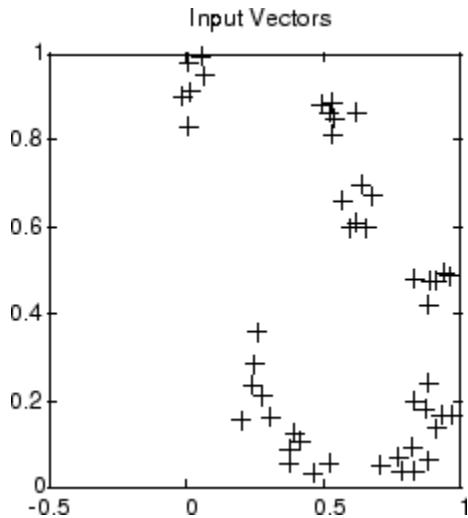
ans =
    5.4367
    5.4365
```

(You might get different answers when you run this problem, because a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to the origin, while the vector formed from the second row of the weight matrix is close to the input vectors near (1,1). Thus, the network has been trained—just by exposing it to the inputs—to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

Graphical Example

Competitive layers can be understood better when their weight vectors and input vectors are shown graphically. The diagram below shows 48 two-element input vectors represented with + markers.



The input vectors above appear to fall into clusters. You can use a competitive network of eight neurons to classify the vectors into such clusters.

Try `democ1` to see a dynamic example of competitive learning.

Cluster with Self-Organizing Map Neural Network

In this section...

- “Topologies (gridtop, hextop, randtop)” on page 9-10
- “Distance Functions (dist, linkdist, mandist, boxdist)” on page 9-14
- “Architecture” on page 9-17
- “Create a Self-Organizing Map Neural Network (selforgmap)” on page 9-18
- “Training (learnsomb)” on page 9-19
- “Examples” on page 9-22

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The function `gridtop`, `hextop`, or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, `dist`, `boxdist`, `linkdist`, and `mandist`. Link distance is the most common. These topology and distance functions are described in “Topologies (gridtop, hextop, randtop)” on page 9-10 and “Distance Functions (dist, linkdist, mandist, boxdist)” on page 9-14.

Here a self-organizing feature map network identifies a winning neuron i^* using the same procedure as employed by a competitive layer. However, instead of updating only the winning neuron, all neurons within a certain neighborhood $N_{i^*}(d)$ of the winning neuron are updated, using the Kohonen rule. Specifically, all such neurons $i \in N_{i^*}(d)$ are adjusted as follows:

$$_i\mathbf{w}(q) = _i\mathbf{w}(q - 1) + \alpha(\mathbf{p}(q) - _i\mathbf{w}(q - 1))$$

or

$$_i\mathbf{w}(q) = (1 - \alpha)_i\mathbf{w}(q - 1) + \alpha\mathbf{p}(q)$$

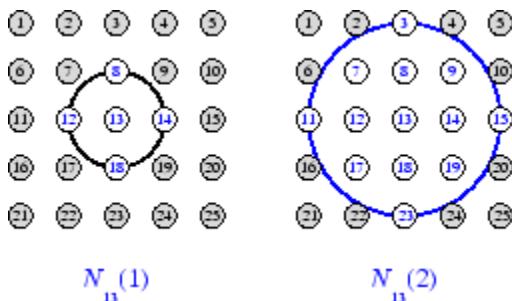
Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^* .

$$N_i(d) = \{j, d_{ij} \leq d\}$$

Thus, when a vector \mathbf{p} is presented, the weights of the winning neuron *and* its close neighbors move toward \mathbf{p} . Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.



These neighborhoods could be written as $N_{13}(1) = \{8, 12, 13, 14, 18\}$ and $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$.

The neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

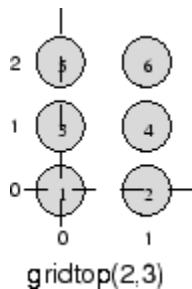
Topologies (gridtop, hextop, randtop)

You can specify different topologies for the original neuron locations with the functions `gridtop`, `hextop`, and `randtop`.

The `gridtop` topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2-by-3 array of six neurons. You can get this with

```
pos = gridtop([2, 3])
pos =
    0      1      0      1      0      1
    0      0      1      1      2      2
```

Here neuron 1 has the position (0,0), neuron 2 has the position (1,0), and neuron 3 has the position (0,1), etc.

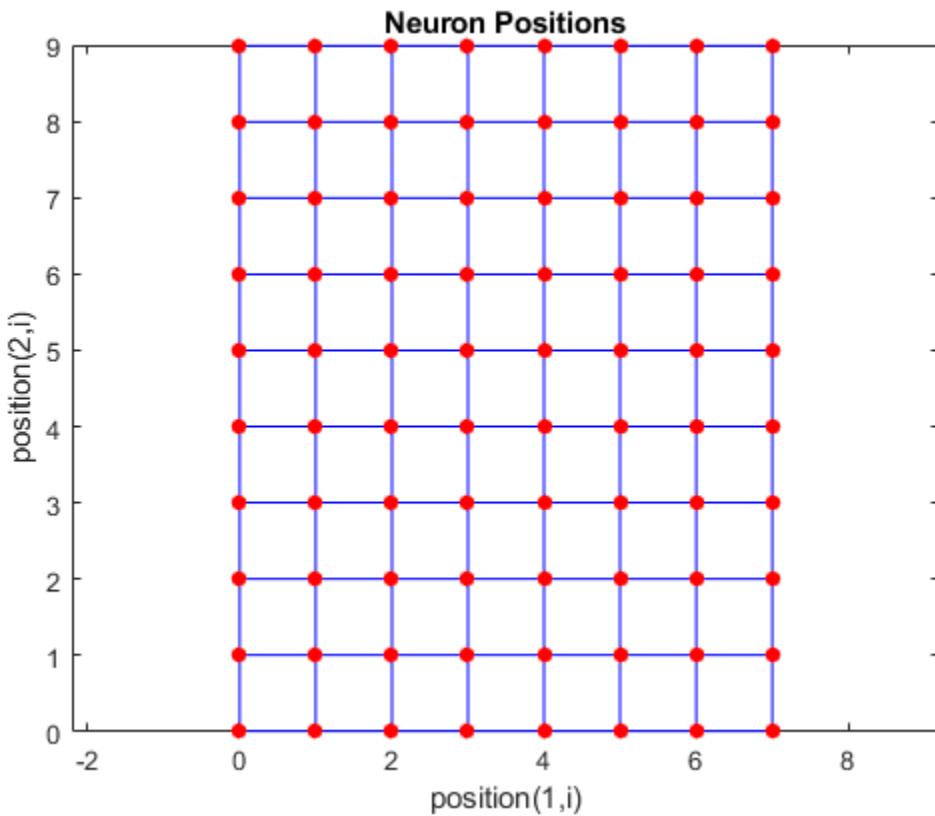


Note that had you asked for a `gridtop` with the dimension sizes reversed, you would have gotten a slightly different arrangement:

```
pos = gridtop([3, 2])
pos =
    0      1      2      0      1      2
    0      0      0      1      1      1
```

You can create an 8-by-10 set of neurons in a `gridtop` topology with the following code:

```
pos = gridtop([8 10]);
plotsom(pos)
```



As shown, the neurons in the `gridtop` topology do indeed lie on a grid.

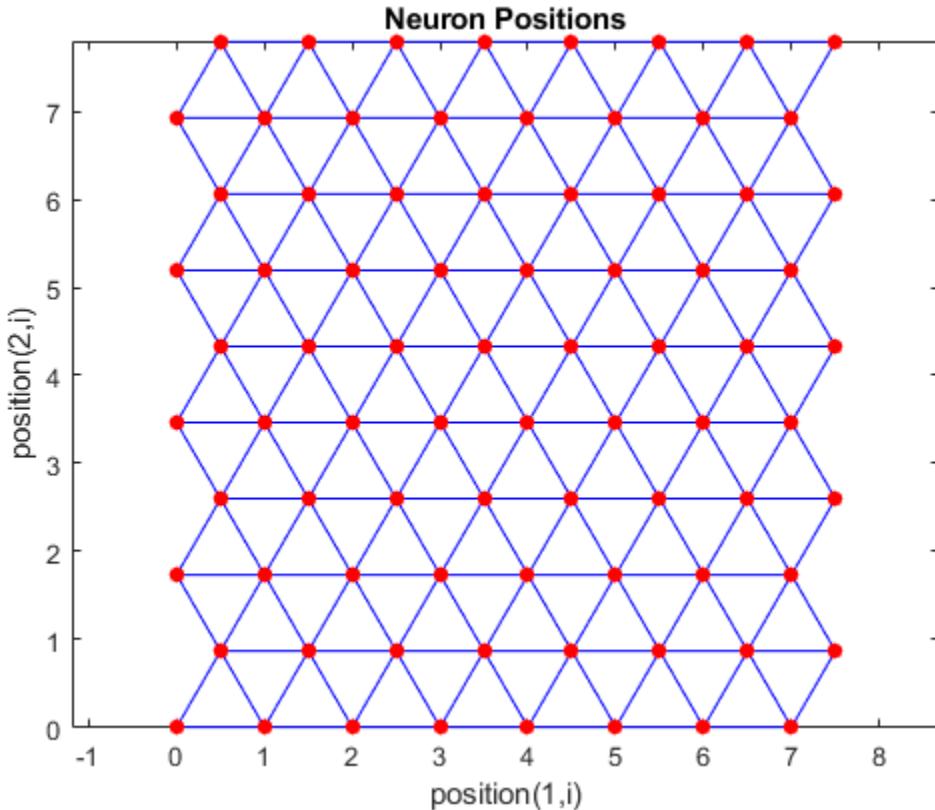
The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
pos = hextop([2, 3])
pos =
    0    1.0000    0.5000    1.5000      0    1.0000
    0        0    0.8660    0.8660    1.7321    1.7321
```

Note that `hextop` is the default pattern for SOM networks generated with `selforgmap`.

You can create and plot an 8-by-10 set of neurons in a `hextop` topology with the following code:

```
pos = hextop([8 10]);
plotsom(pos)
```



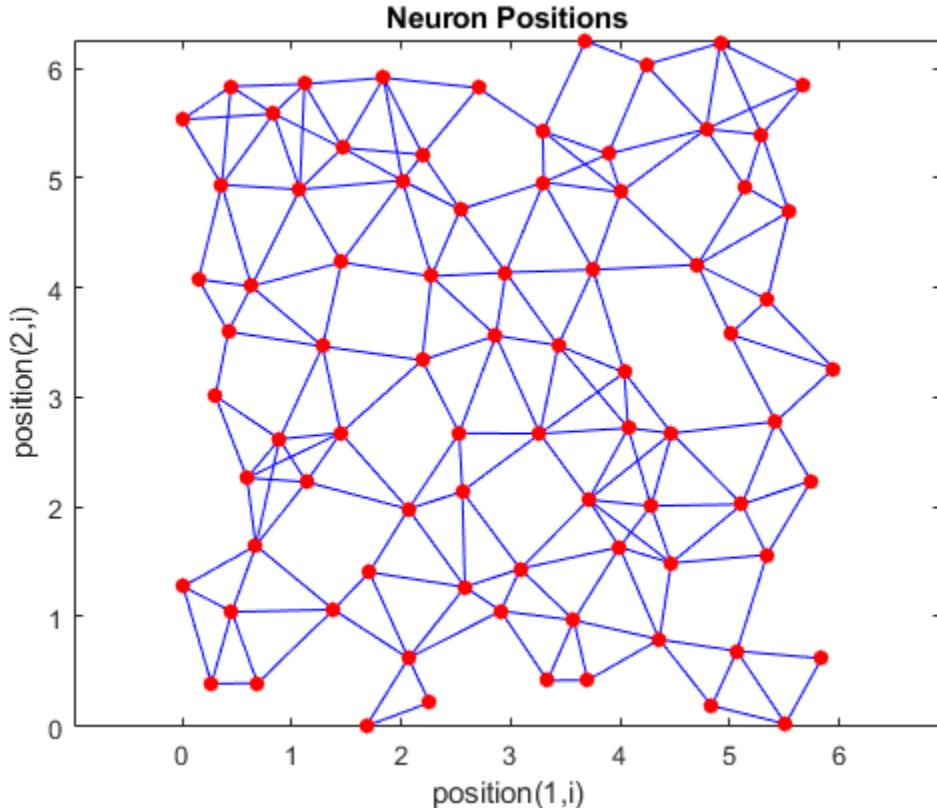
Note the positions of the neurons in a hexagonal arrangement.

Finally, the `randtop` function creates neurons in an N-dimensional random pattern. The following code generates a random pattern of neurons.

```
pos = randtop([2, 3])
pos =
    0      0.7620      0.6268      1.4218      0.0663      0.7862
  0.0925          0      0.4984      0.6007      1.1222      1.4228
```

You can create and plot an 8-by-10 set of neurons in a `randtop` topology with the following code:

```
pos = randtop([8 10]);  
plotsom(pos)
```



For examples, see the help for these topology functions.

Distance Functions (`dist`, `linkdist`, `mandist`, `boxdist`)

In this toolbox, there are four ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

The `dist` function calculates the Euclidean distances from a *home* neuron to other neurons. Suppose you have three neurons:

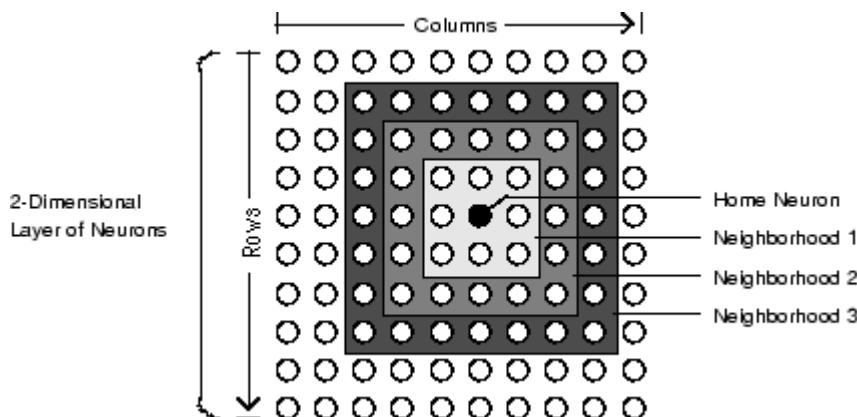
```
pos2 = [0 1 2; 0 1 2]
pos2 =
    0     1     2
    0     1     2
```

You find the distance from each neuron to the other with

```
D2 = dist(pos2)
D2 =
    0     1.4142   2.8284
  1.4142      0     1.4142
  2.8284   1.4142      0
```

Thus, the distance from neuron 1 to itself is 0, the distance from neuron 1 to neuron 2 is 1.4142, etc.

The graph below shows a home neuron in a two-dimensional (`gridtop`) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1 includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the `dist` function, all the neighborhoods for an S -neuron layer map are represented by an S -by- S matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function `boxdist`. Suppose that you have six neurons in a `gridtop` configuration.

```
pos = gridtop([2, 3])
pos =
    0      1      0      1      0      1
    0      0      1      1      2      2
```

Then the box distances are

```
d = boxdist(pos)
d =
    0      1      1      1      2      2
    1      0      1      1      2      2
    1      1      0      1      1      1
    1      1      1      0      1      1
    2      2      1      1      0      1
    2      2      1      1      1      0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if you calculate the distances from the same set of neurons with `linkdist`, you get

```
dlink =
    0      1      1      2      2      3
    1      0      2      1      3      2
    1      2      0      1      1      2
    2      1      1      0      2      1
    2      3      1      2      0      1
    3      2      2      1      1      0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

Thus if you have

```
W1 = [1 2; 3 4; 5 6]
W1 =
    1      2
    3      4
    5      6
```

and

```
P1 = [1;1]
P1 =
    1
    1
```

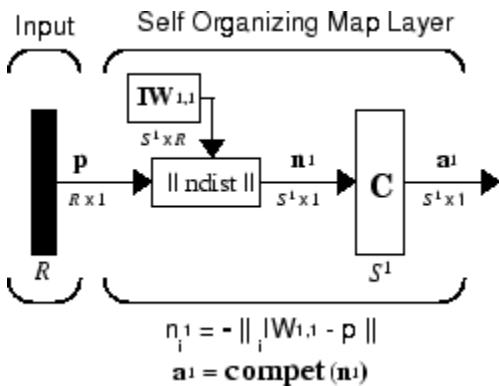
then you get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
    1
    5
    9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element $a_{1,i}^1$ corresponding to i^* , the winning neuron. All other output elements in a^1 are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. You can choose from various topologies of neurons. Similarly, you can choose from various distance expressions to calculate neurons that are close to the winning neuron.

Create a Self-Organizing Map Neural Network (**selforgmap**)

You can create a new SOM network with the function **selforgmap**. This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that you want to create a network having input vectors with two elements, and that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is:

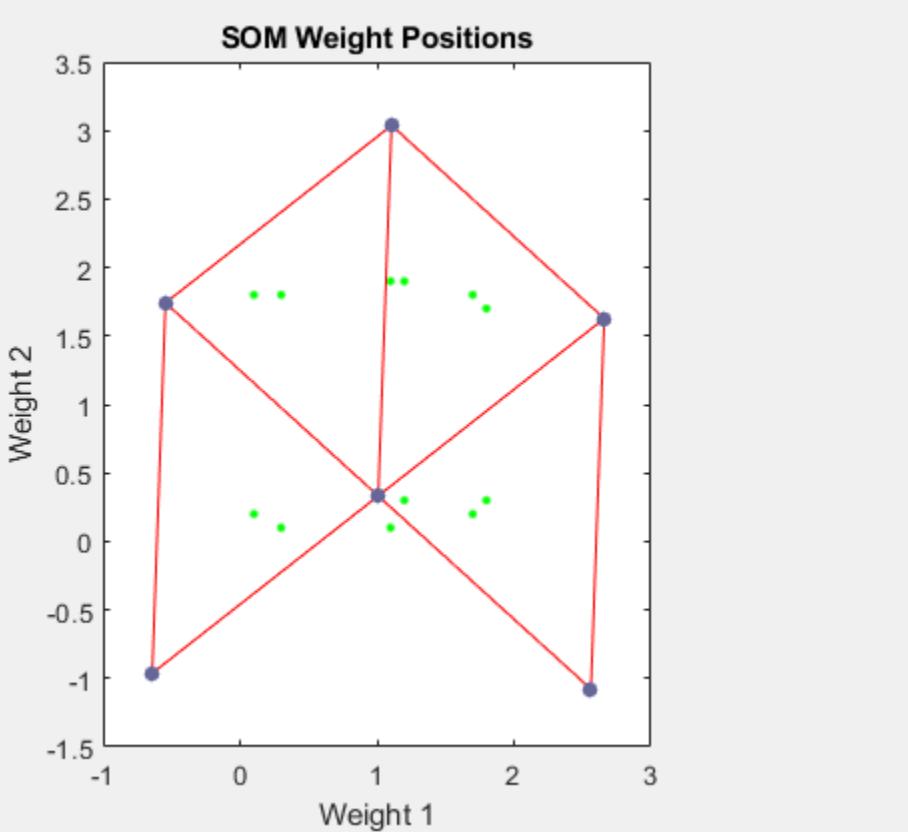
```
net = selforgmap([2 3]);
```

Suppose that the vectors to train on are:

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1 1.8 1.7; ...  
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9 1.7 1.8];
```

You can configure the network to input the data and plot all of this with:

```
net = configure(net,P);  
plotsompos(net,P)
```



The green spots are the training vectors. The initialization for `selforgmap` spreads the initial weights across the input space. Note that they are initially some distance from the training vectors.

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated (`negdist`) to get the weighted inputs. The weighted inputs are also the net inputs (`netsum`). The net inputs compete (`compet`) so that only the neuron with the most positive net input will output a 1.

Training (`learnsomb`)

The default learning in a self-organizing feature map occurs in the batch mode (`trainbu`). The weight learning function for the self-organizing map is `learnsomb`.

First, the network identifies the winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner or for which it is in the neighborhood of a winner. The distance that defines the size of the neighborhood is altered during training through two phases.

Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts at a given initial distance, and decreases to the tuning neighborhood distance (1.0). As the neighborhood distance decreases over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood size has decreased below 1 so only the winning neuron learns for each sample.

Now take a look at some of the specific values commonly used in these networks.

Learning occurs according to the `learnsomb` learning parameter, shown here with its default value.

Learning Parameter	Default Value	Purpose
<code>LP.init_neighborhood</code>	3	Initial neighborhood size
<code>LP.steps</code>	100	Ordering phase steps

The neighborhood size NS is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as `LP.steps`. During this phase, the algorithm adjusts ND from the initial neighborhood size `LP.init_neighborhood` down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, ND is less than 1. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases

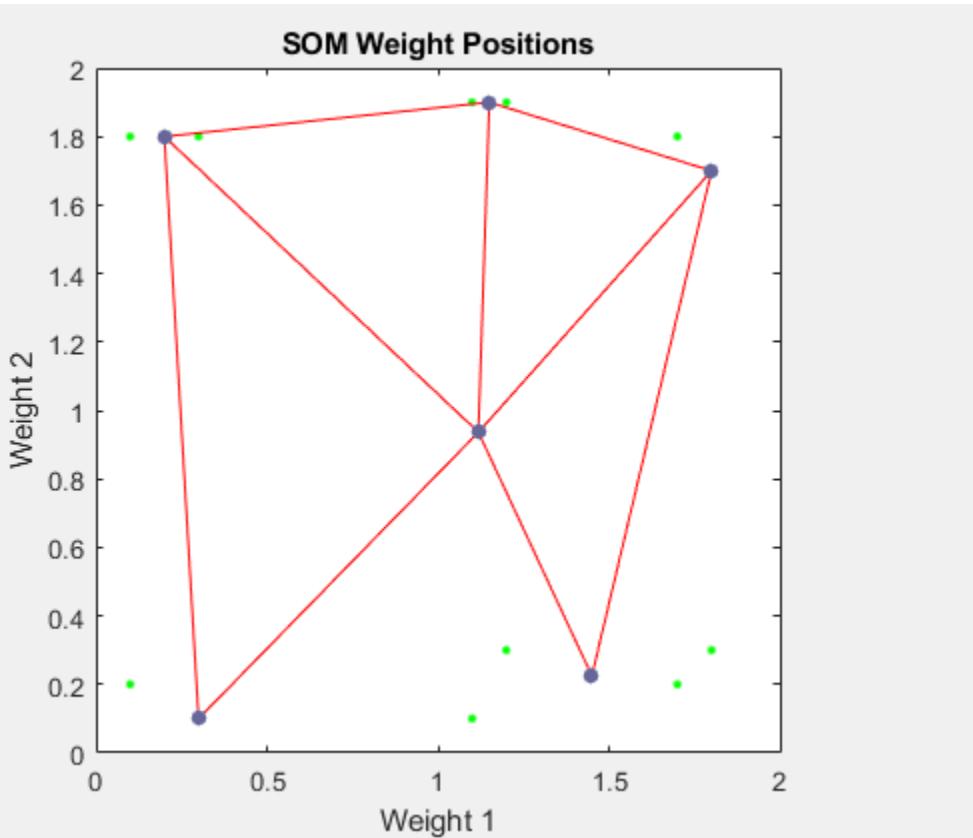
to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered. The training continues in order to give the neurons time to spread out evenly across the input vectors.

As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

You can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;
net = train(net,P);
plotsompos(net,P)
```



You can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

Examples

Two examples are described briefly below. You also might try the similar examples `demosm1` and `demosm2`.

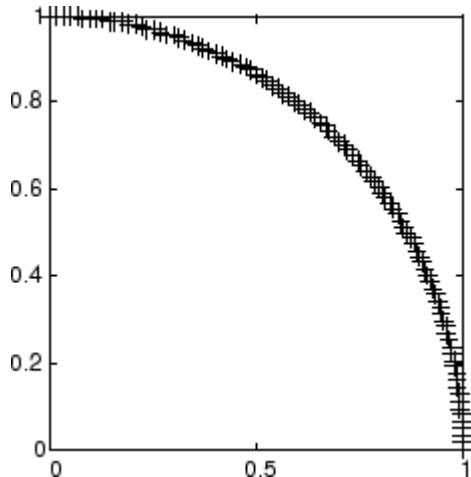
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90° .

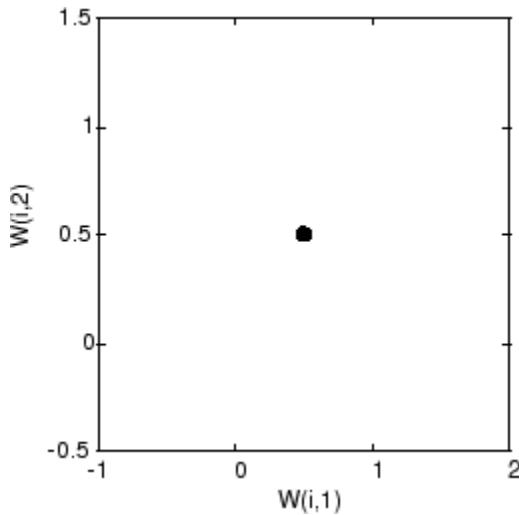
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

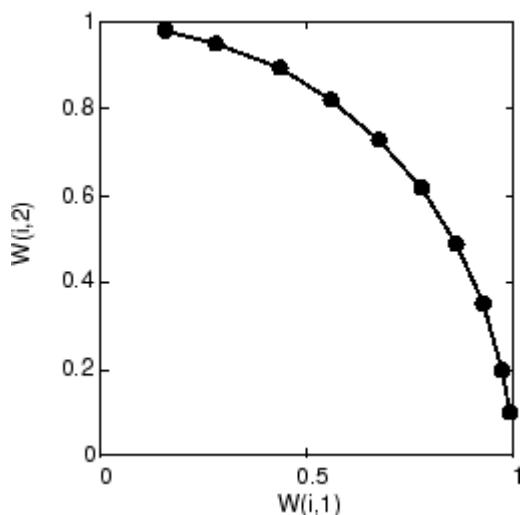


A self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



Of course, because all the weight vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial vectors does not guarantee identical training results.

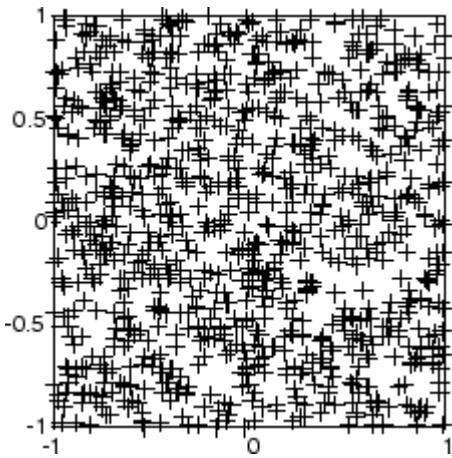
Two-Dimensional Self-Organizing Map

This example shows how a two-dimensional self-organizing map can be trained.

First some random input data is created with the following code:

```
P = rands(2,1000);
```

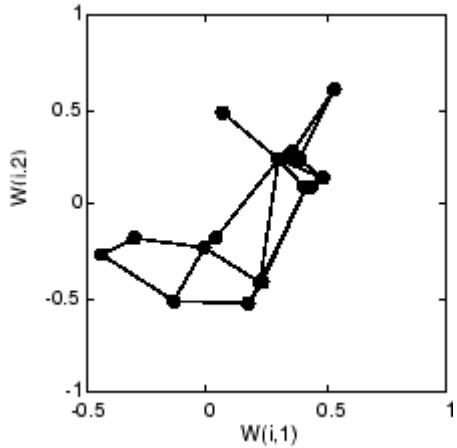
Here is a plot of these 1000 input vectors.



A 5-by-6 two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with distances calculated according to the Manhattan distance neighborhood function `mandist`.

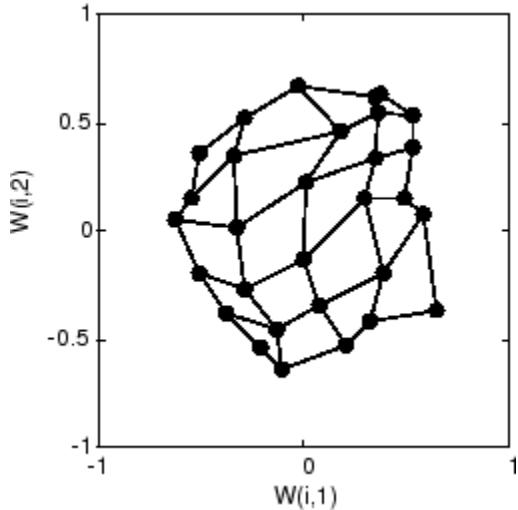
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



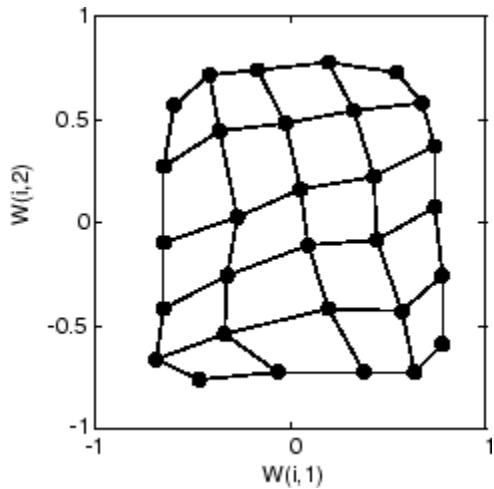
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

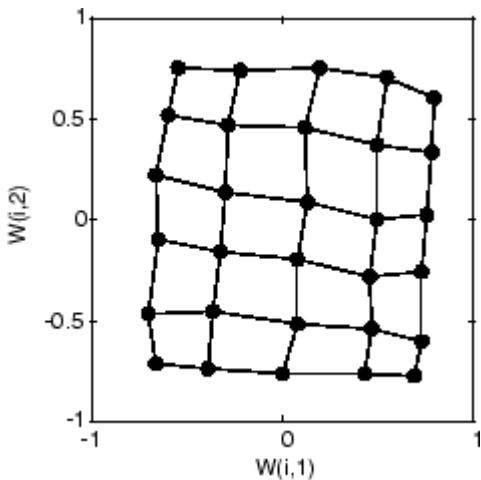


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

The following plot, after 500 cycles, shows the map more evenly distributed across the input space.



Finally, after 5000 cycles, the map is rather evenly spread across the input space. In addition, the neurons are very evenly spaced, reflecting the even distribution of input vectors in this problem.



Thus a two-dimensional self-organizing map has learned the topology of its inputs' space.

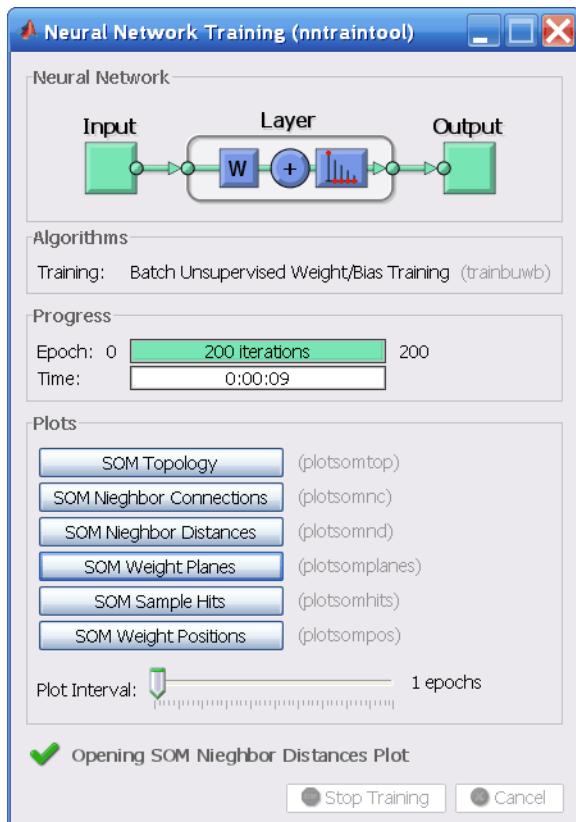
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

Training with the Batch Algorithm

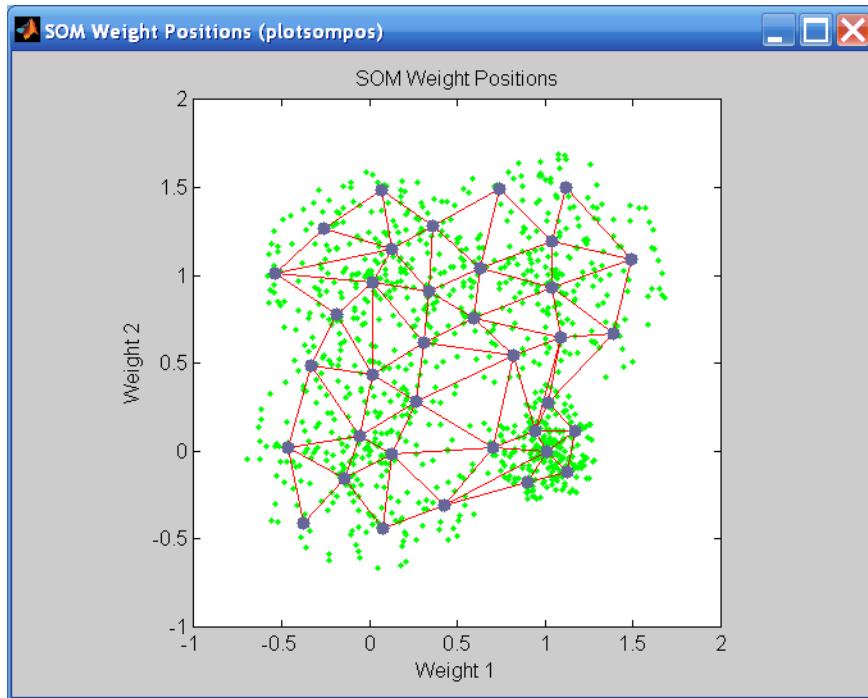
The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
x = simplecluster_dataset  
net = selforgmap([6 6]);  
net = train(net,x);
```

This command sequence creates and trains a 6-by-6 two-dimensional map of 36 neurons. During training, the following figure appears.



There are several useful visualizations that you can access from this window. If you click **SOM Weight Positions**, the following figure appears, which shows the locations of the data points and the weight vectors. As the figure indicates, after only 200 iterations of the batch algorithm, the map is well distributed through the input space.



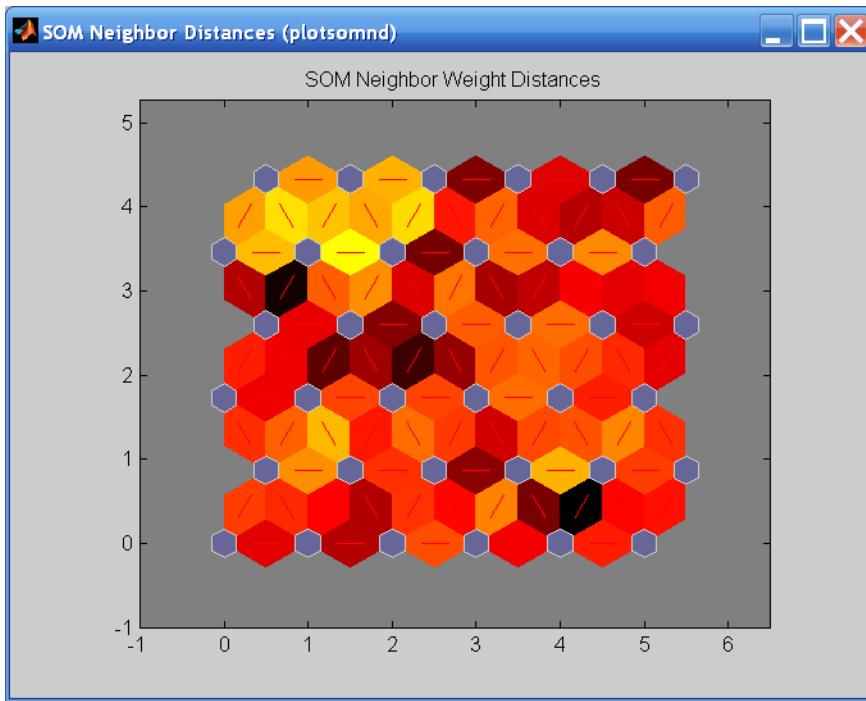
When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

This figure uses the following color coding:

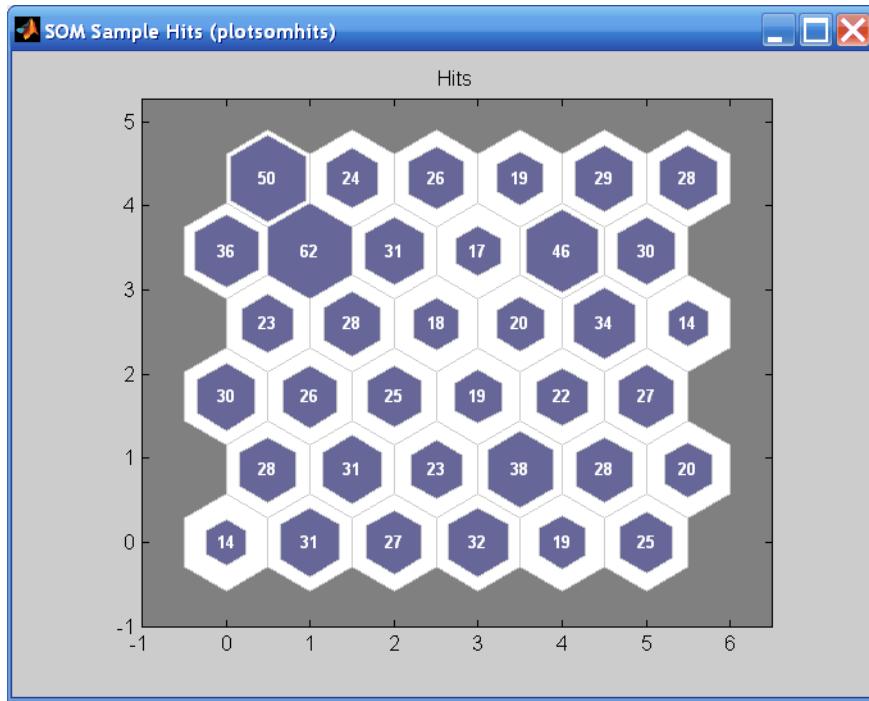
- The blue hexagons represent the neurons.
- The red lines connect neighboring neurons.
- The colors in the regions containing the red lines indicate the distances between neurons.
- The darker colors represent larger distances.
- The lighter colors represent smaller distances.

A group of light segments appear in the upper-left region, bounded by some darker segments. This grouping indicates that the network has clustered the data into two groups. These two groups can be seen in the previous weight position figure. The lower-

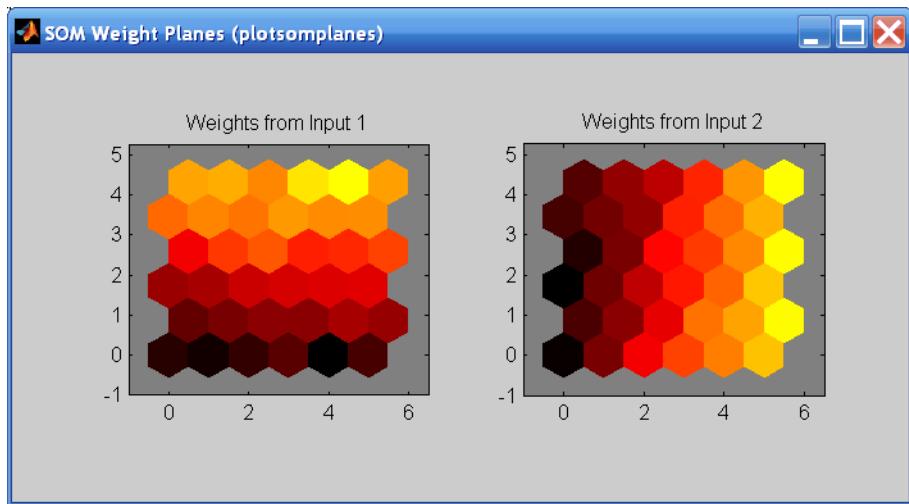
right region of that figure contains a small group of tightly clustered data points. The corresponding weights are closer together in this region, which is indicated by the lighter colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor distance figure. The segments in the lower-right region of the neighbor distance figure are darker than those in the upper left. This color difference indicates that data points in this region are farther apart. This distance is confirmed in the weight positions figure.



Another useful figure can tell you how many data points are associated with each neuron. Click **SOM Sample Hits** to see the following figure. It is best if the data are fairly evenly distributed across the neurons. In this example, the data are concentrated a little more in the upper-left neurons, but overall the distribution is fairly even.



You can also visualize the weights themselves using the weight plane figure. Click **SOM Weight Planes** in the training window to obtain the next figure. There is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2.



You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`.

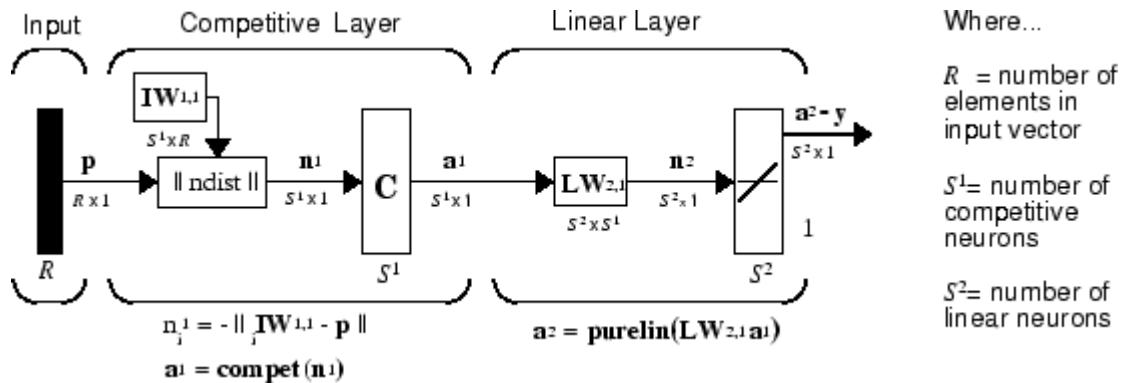
Learning Vector Quantization (LVQ) Neural Networks

In this section...

- “Architecture” on page 9-34
- “Creating an LVQ Network” on page 9-35
- “LVQ1 Learning Rule (learnlv1)” on page 9-38
- “Training” on page 9-39
- “Supplemental LVQ2.1 Learning Rule (learnlv2)” on page 9-41

Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of “Cluster with Self-Organizing Map Neural Network” on page 9-9 described in this topic. The linear layer transforms the competitive layer’s classes into target classifications defined by the user. The classes learned by the competitive layer are referred to as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to S^1 subclasses. These, in turn, are combined by the linear layer to form S^2 target classes. (S^1 is always larger than S^2 .)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that belongs to the linear layer target class 2. Then competitive neurons

1, 2, and 3 will have $\mathbf{LW}^{2,1}$ weights of 1.0 to neuron \mathbf{n}^2 in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1, 2, or 3) wins the competition and outputs a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a 1 in the i th row of \mathbf{a}^1 (the rest to the elements of \mathbf{a}^1 will be zero) effectively picks the i th column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1s from layer 1 are put into various classes by the $\mathbf{LW}^{2,1}\mathbf{a}^1$ multiplication in layer 2.

You know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so you can specify the elements of $\mathbf{LW}^{2,1}$ at the start. However, you have to go through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. This training is discussed in “Training” on page 9-6. First, consider how to create the original network.

Creating an LVQ Network

You can create an LVQ network with the function `lvqnet`,

```
net = lvqnet(S1,LR,LF)
```

where

- $S1$ is the number of first-layer hidden neurons.
- LR is the learning rate (default 0.01).
- LF is the learning function (default is `learnlvl1`).

Suppose you have 10 input vectors. Create a network that assigns each of these input vectors to one of four subclasses. Thus, there are four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2 0 0 0 0 2 2 3; 0 1 -1 2 1 -1 -2 1 -1 0];
```

and

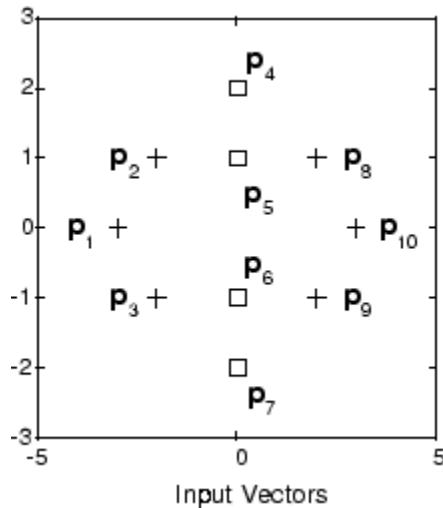
```
Tc = [1 1 1 2 2 2 1 1 1];
```

It might help to show the details of what you get from these two lines of code.

```
P,Tc  
P =
```

$$Tc = \begin{matrix} -3 & -2 & -2 & 0 & 0 & 0 & 0 & 2 & 2 & 3 \\ 0 & 1 & -1 & 2 & 1 & -1 & -2 & 1 & -1 & 0 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 & 1 & 1 \end{matrix}$$

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. You want a network that classifies \mathbf{p}_1 , \mathbf{p}_2 , \mathbf{p}_3 , \mathbf{p}_8 , \mathbf{p}_9 , and \mathbf{p}_{10} to produce an output of 1, and that classifies vectors \mathbf{p}_4 , \mathbf{p}_5 , \mathbf{p}_6 , and \mathbf{p}_7 to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next convert the Tc matrix to target vectors.

```
T = ind2vec(Tc);
```

This gives a sparse matrix T that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
    1     1     1     0     0     0     0     1     1     1
    0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if you have the first column of P as input, you should get the first column of **targets** as an output; and that output says the input falls in class 1, which is correct. Now you are ready to call **lvqnet**.

Call **lvqnet** to create a network with four neurons.

```
net = lvqnet(4);
```

Configure and confirm the initial values of the first-layer weight matrix are initialized by the function midpoint to values in the center of the input data range.

```
net = configure(net,P,T);
net.IW{1}
ans =
    0     0
    0     0
    0     0
    0     0
```

Confirm that the second-layer weights have 60% (6 of the 10 in T_c) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns have a 1 in the second row (corresponding to class 2). With only four columns, the 60% and 40% actually round to 50% and there are two 1's in each row.

```
net.LW{2,1}
ans =
    1     1     0     0
    0     0     1     1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element, the input vector is classified as class 1; otherwise it is a class 2.

You might notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1), while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

You can simulate the network with **sim**. Use the original P matrix as input just to see what you get.

```
Y = net(P);
Yc = vec2ind(Y)
```

$$Y_C = \begin{matrix} & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

The network classifies all inputs into class 1. Because this is not what you want, you have to train the network (adjusting the weights of layer 1 only), before you can expect a good result. The next two sections discuss two LVQ learning rules and the training process.

LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$$

Here there are input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector \mathbf{p} is presented, and the distance from \mathbf{p} to each row of the input weight matrix $\mathbf{IW}^{1,1}$ is computed with the function `negdist`. The hidden neurons of layer 1 compete. Suppose that the i th element of \mathbf{n}^1 is most positive, and neuron i^* wins the competition. Then the competitive transfer function produces a 1 as the i^* th element of \mathbf{a}^1 . All other elements of \mathbf{a}^1 are 0.

When \mathbf{a}^1 is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in \mathbf{a}^1 selects the class k^* associated with the input. Thus, the network has assigned the input vector \mathbf{p} to class k^* and $a_{k^*}^2$ will be 1. Of course, this assignment can be a good one or a bad one, for t_{k^*} can be 1 or 0, depending on whether the input belonged to class k^* or not.

Adjust the i^* th row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector \mathbf{p} if the assignment is correct, and to move the row away from \mathbf{p} if the assignment is incorrect. If \mathbf{p} is classified correctly,

$$(a_{k^*}^2 = t_{k^*} = 1)$$

compute the new value of the i^{th} row of $\mathbf{IW}^{1,1}$ as

$$i * \mathbf{IW}^{1,1}(q) = i * \mathbf{IW}^{1,1}(q - 1) + \alpha(\mathbf{p}(q) - i * \mathbf{IW}^{1,1}(q - 1))$$

On the other hand, if \mathbf{p} is classified incorrectly,

$$(\alpha_{k*}^2 = 1 \neq t_{k*} = 0)$$

compute the new value of the i^{th} row of $\mathbf{IW}^{1,1}$ as

$$i * \mathbf{IW}^{1,1}(q) = i * \mathbf{IW}^{1,1}(q - 1) - \alpha(\mathbf{p}(q) - i * \mathbf{IW}^{1,1}(q - 1))$$

You can make these corrections to the i^{th} row of $\mathbf{IW}^{1,1}$ automatically, without affecting other rows of $\mathbf{IW}^{1,1}$, by back-propagating the output errors to layer 1.

Such corrections move the hidden neuron toward vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is `learnlv1`. It can be applied during training.

Training

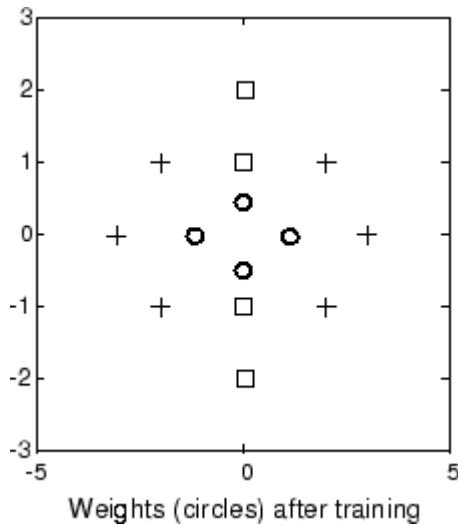
Next you need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. You do this with `train` as with the following commands. First, set the training epochs to 150. Then, use `train`:

```
net.trainParam.epochs = 150;
net = train(net,P,T);
```

Now confirm the first-layer weights.

```
net.IW{1,1}
ans =
    0.3283    0.0051
   -0.1366    0.0001
   -0.0263    0.2234
     0      -0.0685
```

The following plot shows that these weights have moved toward their respective classification groups.



To confirm that these weights do indeed lead to the correct classification, take the matrix P as input and simulate the network. Then see what classifications are produced by the network.

```
Y = net(P);
Yc = vec2ind(Y)
```

This gives

```
Yc =
    1      1      1      2      2      2      2      1      1      1
```

which is expected. As a last check, try an input close to a vector that was used in training.

```
pchk1 = [0; 0.5];
Y = net(pchk1);
Yc1 = vec2ind(Y)
```

This gives

```
Yc1 =
    2
```

This looks right, because $pchk1$ is close to other vectors classified as 2. Similarly,

```
pchk2 = [1; 0];
Y = net(pchk2);
Yc2 = vec2ind(Y)
```

gives

```
Yc2 =
    1
```

This looks right too, because pchk2 is close to other vectors classified as 1.

You might want to try the example program `demolvq1`. It follows the discussion of training given above.

Supplemental LVQ2.1 Learning Rule (`learnlv2`)

The following learning rule is one that might be applied *after* first applying LVQ1. It can improve the result of the first learning. This particular version of LVQ2 (referred to as LVQ2.1 in the literature [Koho97 on page 14-2]) is embodied in the function `learnlv2`. Note again that LVQ2.1 is to be used only after LVQ1 has been applied.

Learning here is similar to that in `learnlv2` except now two vectors of layer 1 that are closest to the input vector can be updated, provided that one belongs to the correct class and one belongs to a wrong class, and further provided that the input falls into a “window” near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s$$

where

$$s \equiv \frac{1 - w}{1 + w}$$

(where d_i and d_j are the Euclidean distances of \mathbf{p} from $_{i^*}\mathbf{IW}^{1,1}$ and $_{j^*}\mathbf{IW}^{1,1}$, respectively). Take a value for w in the range 0.2 to 0.3. If you pick, for instance, 0.25, then $s = 0.6$. This means that if the minimum of the two distance ratios is greater than 0.6, the two vectors are adjusted. That is, if the input is near the midplane, adjust the two vectors, provided also that the input vector \mathbf{p} and $_{j^*}\mathbf{IW}^{1,1}$ belong to the same class, and \mathbf{p} and $_{i^*}\mathbf{IW}^{1,1}$ do not belong in the same class.

The adjustments made are

$${}_i * \mathbf{IW}^{1,1}(q) = {}_i * \mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_i * \mathbf{IW}^{1,1}(q-1))$$

and

$${}_j * \mathbf{IW}^{1,1}(q) = {}_j * \mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_j * \mathbf{IW}^{1,1}(q-1))$$

Thus, given two vectors closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors are adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

Function	Description
<code>competlayer</code>	Create a competitive layer.
<code>learnk</code>	Kohonen learning rule.
<code>selforgmap</code>	Create a self-organizing map.
<code>learncon</code>	Conscience bias learning function.
<code>boxdist</code>	Distance between two position vectors.
<code>dist</code>	Euclidean distance weight function.
<code>linkdist</code>	Link distance function.
<code>mandist</code>	Manhattan distance weight function.
<code>gridtop</code>	Gridtop layer topology function.
<code>hextop</code>	Hexagonal layer topology function.
<code>randtop</code>	Random layer topology function.
<code>lvqnet</code>	Create a learning vector quantization network.
<code>learnlv1</code>	LVQ1 weight learning function.
<code>learnlv2</code>	LVQ2 weight learning function.

Adaptive Filters and Adaptive Training

Adaptive Neural Network Filters

In this section...

- “Adaptive Functions” on page 10-2
- “Linear Neuron Model” on page 10-3
- “Adaptive Linear Network Architecture” on page 10-4
- “Least Mean Square Error” on page 10-6
- “LMS Algorithm (learnwh)” on page 10-7
- “Adaptive Filtering (adapt)” on page 10-7

The ADALINE (adaptive linear neuron) networks discussed in this topic are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can solve only linearly separable problems. However, here the LMS (least mean squares) learning rule, which is much more powerful than the perceptron learning rule, is used. The LMS, or Widrow-Hoff, learning rule minimizes the mean square error and thus moves the decision boundaries as far as it can from the training patterns.

In this section, you design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancelation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this subject is Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985.

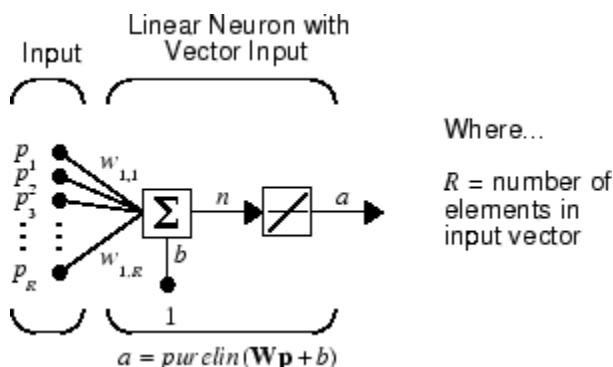
The adaptive training of self-organizing and competitive networks is also considered in this section.

Adaptive Functions

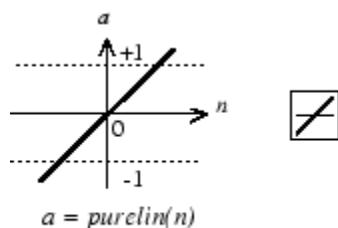
This section introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

Linear Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, named `purelin`.



Linear Transfer Function

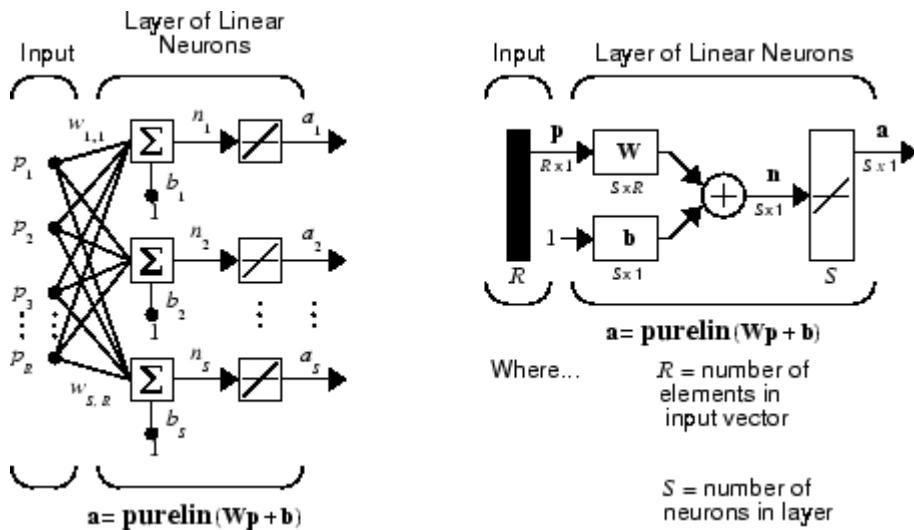
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\alpha = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p}) + b = \mathbf{W}\mathbf{p} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights \mathbf{W} .

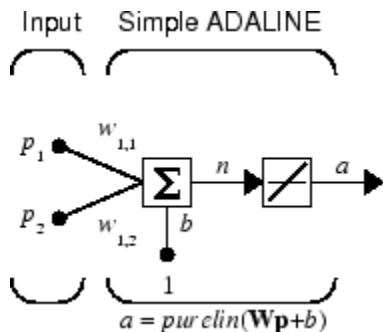


This network is sometimes called a MADALINE for Many ADALINEs. Note that the figure on the right defines an S -length output vector \mathbf{a} .

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (linearlayer)

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.



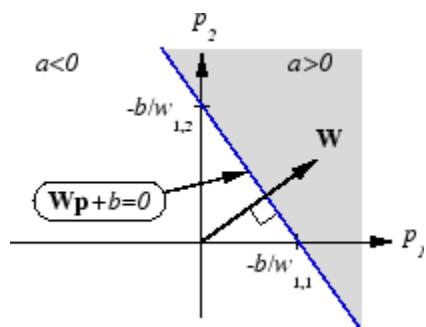
The weight matrix \mathbf{W} in this case has only one row. The network output is

$$\alpha = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p}) + b = \mathbf{W}\mathbf{p} + b$$

or

$$\alpha = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{W}\mathbf{p} + b = 0$ specifies such a decision boundary, as shown below (adapted with thanks from [HDB96 on page 14-2]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

However, ADALINE can classify objects in this way only when the objects are linearly separable. Thus, ADALINE has the same limitation as the perceptron.

You can create a network similar to the one shown using this command:

```
net = linearlayer;
net = configure(net,[0;0],[0]);
```

The sizes of the two arguments to `configure` indicate that the layer is to have two inputs and one output. Normally `train` does this configuration for you, but this allows us to inspect the weights before training.

The network weights and biases are set to zero, by default. You can see the current values using the commands:

```
W = net.IW{1,1}
W =
    0      0
```

and

```
b = net.b{1}
b =
    0
```

You can also assign arbitrary values to the weights and bias, such as 2 and 3 for the weights and -4 for the bias:

```
net.IW{1,1} = [2 3];
net.b{1} = -4;
```

You can simulate the ADALINE for a particular input vector.

```
p = [5; 6];
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE network with `linearlayer`, adjust its elements as you want, and simulate it with `sim`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here \mathbf{p}_q is an input to the network, and \mathbf{t}_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q} \sum_{k=1}^Q e(k)^2 = \frac{1}{Q} \sum_{k=1}^Q t(k) - \alpha(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can learn more about this topic in Chapter 10 of [HDB96 on page 14-2].

LMS Algorithm (learnwh)

Adaptive networks will use the LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown here, is discussed in detail in “Linear Neural Networks” on page 12-18.

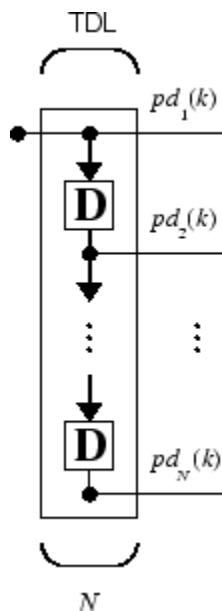
$$\begin{array}{rclcrcl} \mathbf{W}(k) & + & 1) & = & \mathbf{W}(k) & + & 2\alpha \mathbf{e}(k) \mathbf{p}^T(k) \\ \mathbf{b}(k) & + & 1) & = & \mathbf{b}(k) & + & 2\alpha \mathbf{e}(k) \end{array}$$

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

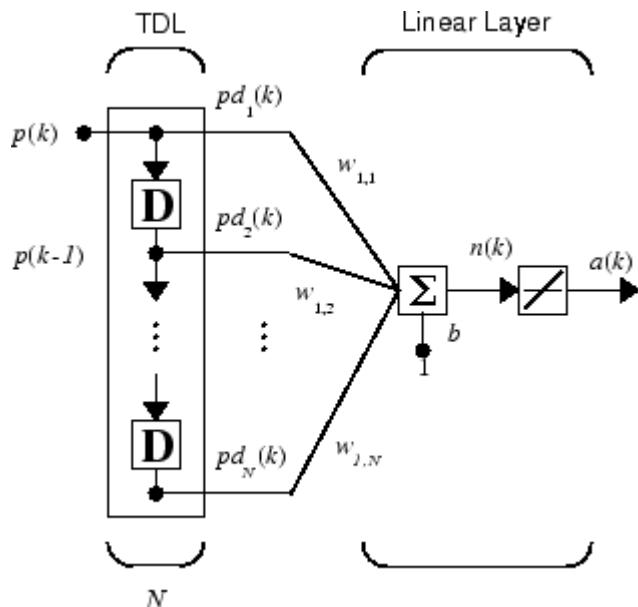
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown in the next figure. The input signal enters from the left and passes through $N-1$ delays. The output of the tapped delay line (TDL) is an N -dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Adaptive Filter

You can combine a tapped delay line with an ADALINE network to create the *adaptive filter* shown in the next figure.



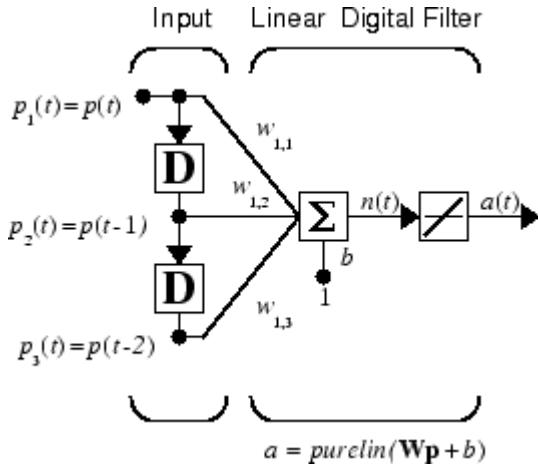
The output of the filter is given by

$$\alpha(k) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^R w_{1,i} \alpha(k - i + 1) + b$$

In digital signal processing, this network is referred to as a *finite impulse response (FIR)* filter [WiSt85 on page 14-2]. Take a look at the code used to generate and simulate such an adaptive network.

Adaptive Filter Example

First, define a new linear network using `linearlayer`.



Assume that the linear layer has a single neuron with a single input and a tap delay of 0, 1, and 2 delays.

```
net = linearlayer([0 1 2]);
net = configure(net,0,0);
```

You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.

You can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
net.b{1} = [0];
```

Finally, define the initial values of the outputs of the delays as

```
pi = {1 2};
```

These are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network.

To set up the input, assume that the input scalars arrive in a sequence: first the value 3, then the value 4, next the value 5, and finally the value 6. You can indicate this sequence by defining the values as elements of a cell array in curly braces.

```
p = {3 4 5 6};
```

Now, you have a network and a sequence of inputs. Simulate the network to see what its output is as a function of time.

```
[a,pf] = sim(net,p,pi)
```

This simulation yields an output sequence

```
a  
[46] [70] [94] [118]
```

and final values for the delay outputs of

```
pf  
[5] [6]
```

The example is sufficiently simple that you can check it without a calculator to make sure that you understand the inputs, initial values of the delays, etc.

The network just defined can be trained with the function `adapt` to produce a particular output sequence. Suppose, for instance, you want the network to produce the sequence of values 10, 20, 30, 40.

```
t = {10 20 30 40};
```

You can train the defined network to do this, starting from the initial delay conditions used above.

Let the network adapt for 10 passes over the data.

```
for i = 1:10  
    [net,y,E,pf,af] = adapt(net,p,t,pi);  
end
```

This code returns the final weights, bias, and output sequence shown here.

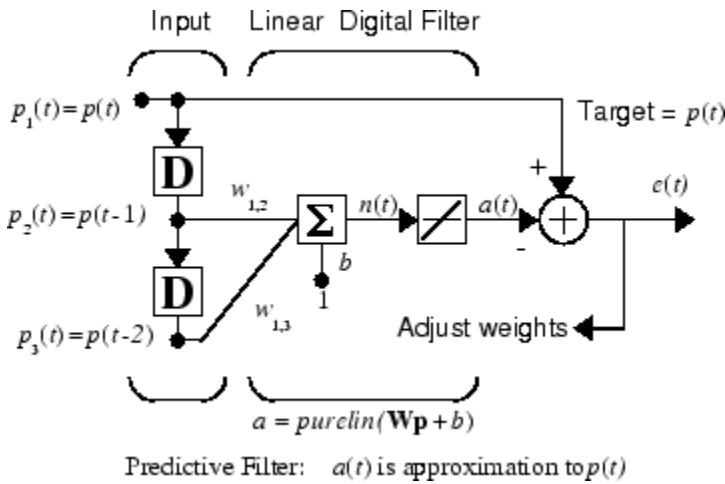
```
wts = net.IW{1,1}  
wts =  
    0.5059    3.1053    5.7046  
bias = net.b{1}  
bias =  
    -1.5993  
y  
y =  
[11.8558]    [20.7735]    [29.6679]    [39.0036]
```

Presumably, if you ran additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with `adapt`. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as prediction or noise cancelation.

Prediction Example

Suppose that you want to use an adaptive filter to predict the next value of a stationary random process, $p(t)$. You can use the network shown in the following figure to do this prediction.



The signal to be predicted, $p(t)$, enters from the left into a tapped delay line. The previous two values of $p(t)$ are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error $e(t)$ on the far right. If this error is 0, the network output $a(t)$ is exactly equal to $p(t)$, and the network has done its prediction properly.

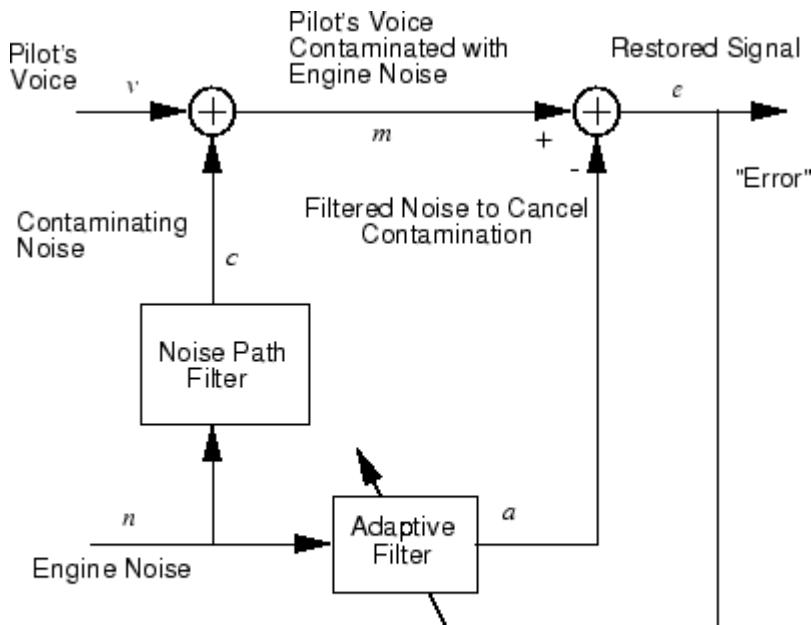
Given the autocorrelation function of the stationary random process $p(t)$, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. This lack does not matter to the network. After it is initialized and operating, the network adapts at each time step to minimize the error and in a relatively short time is able to predict the input $p(t)$.

Chapter 10 of [HDB96 on page 14-2] presents this problem, goes through the analysis, and shows the weight trajectory during training. The network finds the optimum weights on its own without any difficulty whatsoever.

You also can try the example `nnd10nc` to see an adaptive noise cancelation program example in action. This example allows you to pick a learning rate and *momentum* (see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 5-2), and shows the learning trajectory, and the original and cancelation signals versus time.

Noise Cancelation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit combines with the voice signal. This additional noise makes the resultant signal heard by passengers of low quality. The goal is to obtain a signal that contains the pilot's voice, but not the engine noise. You can cancel the noise with an adaptive filter if you obtain a sample of the engine noise and apply it as the input to the adaptive filter.



Adaptive Filter Adjusts to Minimize Error.
 This removes the engine noise from contaminated signal, leaving the pilot's voice as the “error.”

As the preceding figure shows, you adaptively train the neural linear network to predict the combined pilot/engine signal m from an engine signal n . The engine signal n does not tell the adaptive network anything about the pilot's voice signal contained in m . However, the engine signal n does give the network information it can use to predict the engine's contribution to the pilot/engine signal m .

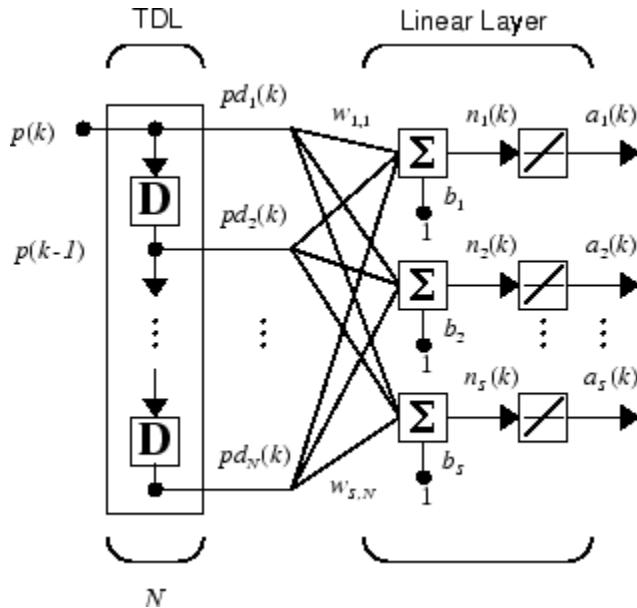
The network does its best to output m adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal m . The network error e is equal to m , the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, e contains only the pilot's voice. The linear adaptive network adaptively learns to cancel the engine noise.

Such adaptive noise canceling generally does a better job than a classical filter, because it subtracts from the signal rather than filtering it out the noise of the signal m .

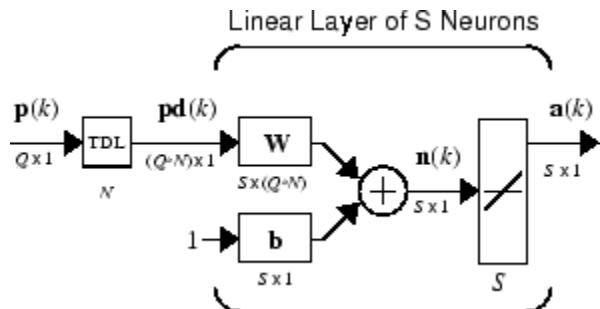
Try `demolin8` for an example of adaptive noise cancelation.

Multiple Neuron Adaptive Filters

You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with S linear neurons, as shown in the next figure.

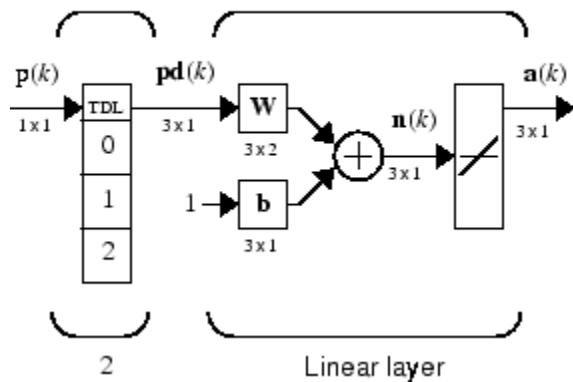


Alternatively, you can represent this same network in abbreviated form.



If you want to show more of the detail of the tapped delay line—and there are not too many delays—you can use the following notation:

Abbreviated Notation



Here, a tapped delay line sends to the weight matrix:

- The current signal
- The previous signal
- The signal delayed before that

You could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays must appear in increasing order as they go from top to bottom.

Advanced Topics

- “Neural Networks with Parallel and GPU Computing” on page 11-2
- “Optimize Neural Network Training Speed and Memory” on page 11-12
- “Choose a Multilayer Neural Network Training Function” on page 11-16
- “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32
- “Edit Shallow Neural Network Properties” on page 11-46
- “Custom Neural Network Helper Functions” on page 11-60
- “Automatically Save Checkpoints During Neural Network Training” on page 11-61
- “Deploy Shallow Neural Network Functions” on page 11-63
- “Deploy Training of Shallow Neural Networks” on page 11-68

Neural Networks with Parallel and GPU Computing

In this section...

- “Deep Learning” on page 11-2
- “Modes of Parallelism” on page 11-2
- “Distributed Computing” on page 11-3
- “Single GPU Computing” on page 11-5
- “Distributed GPU Computing” on page 11-8
- “Parallel Time Series” on page 11-10
- “Parallel Availability, Fallbacks, and Feedback” on page 11-10

Deep Learning

You can train a convolutional neural network (CNN, ConvNet) or long short-term memory networks (LSTM or BiLSTM networks) using the `trainNetwork` function. You can choose the execution environment (CPU, GPU, multi-GPU, and parallel) using `trainingOptions`.

Training in parallel, or on a GPU, requires Parallel Computing Toolbox. For more information on deep learning with GPUs and in parallel, see “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud” on page 1-8.

Modes of Parallelism

Neural networks are inherently parallel algorithms. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can take advantage of this parallelism.

Parallel Computing Toolbox, when used in conjunction with Deep Learning Toolbox, enables neural network training and simulation to take advantage of each mode of parallelism.

For example, the following shows a standard single-threaded training and simulation session:

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

The two steps you can parallelize in this session are the call to `train` and the implicit call to `sim` (where the network `net2` is called as a function).

In Deep Learning Toolbox you can divide any data, such as `x` and `t` in the previous example code, across samples. If `x` and `t` contain only one sample each, there is no parallelism. But if `x` and `t` contain hundreds or thousands of samples, parallelism can provide both speed and problem size benefits.

Distributed Computing

Parallel Computing Toolbox allows neural network training and simulation to run across multiple CPU cores on a single PC, or across multiple CPUs on multiple computers on a network using MATLAB Parallel Server.

Using multiple cores can speed calculations. Using multiple computers can allow you to solve problems using data sets too big to fit in the RAM of a single computer. The only limit to problem size is the total quantity of RAM available across all computers.

To manage cluster configurations, use the Cluster Profile Manager from the MATLAB **Home** tab **Environment** menu **Parallel > Manage Cluster Profiles**.

To open a pool of MATLAB workers using the default cluster profile, which is usually the local CPU cores, use this command:

```
pool = parpool
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

When `parpool` runs, it displays the number of workers available in the pool. Another way to determine the number of workers is to query the pool:

```
pool.NumWorkers
```

```
4
```

Now you can train and simulate the neural network with data split by sample across all the workers. To do this, set the `train` and `sim` parameter '`useParallel`' to '`yes`'.

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

Use the '`showResources`' argument to verify that the calculations ran across multiple workers.

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

MATLAB indicates which resources were used. For example:

```
Computing Resources:
Parallel Workers
Worker 1 on MyComputer, MEX on PCWIN64
Worker 2 on MyComputer, MEX on PCWIN64
Worker 3 on MyComputer, MEX on PCWIN64
Worker 4 on MyComputer, MEX on PCWIN64
```

When `train` and `sim` are called, they divide the input matrix or cell array data into distributed Composite values before training and simulation. When `sim` has calculated a Composite, this output is converted back to the same matrix or cell array form before it is returned.

However, you might want to perform this data division manually if:

- The problem size is too large for the host computer. Manually defining the elements of Composite values sequentially allows much bigger problems to be defined.
- It is known that some workers are on computers that are faster or have more memory than others. You can distribute the data with differing numbers of samples per worker. This is called load balancing.

The following code sequentially creates a series of random datasets and saves them to separate files:

```
pool = gcp;
for i=1:pool.NumWorkers
    x = rand(2,1000);
    save(['inputs' num2str(i)],'x');
    t = x(1,:) .* x(2,:) + 2 * (x(1,:) + x(2,:));
    save(['targets' num2str(i)],'t');
    clear x t
end
```

Because the data was defined sequentially, you can define a total dataset larger than can fit in the host PC memory. PC memory must accommodate only a sub-dataset at a time.

Now you can load the datasets sequentially across parallel workers, and train and simulate a network on the Composite data. When `train` or `sim` is called with Composite data, the '`useParallel`' argument is automatically set to '`yes`'. When using

Composite data, configure the network's input and outputs to match one of the datasets manually using the `configure` function before training.

```
xc = Composite;
tc = Composite;
for i=1:pool.NumWorkers
    data = load(['inputs' num2str(i)], 'x');
    xc{i} = data.x;
    data = load(['targets' num2str(i)], 't');
    tc{i} = data.t;
    clear data
end
net2 = configure(net1, xc{1}, tc{1});
net2 = train(net2, xc, tc);
yc = net2(xc);
```

To convert the Composite output returned by `sim`, you can access each of its elements, separately if concerned about memory limitations.

```
for i=1:pool.NumWorkers
    yi = yc{i}
end
```

Combined the Composite value into one local value if you are not concerned about memory limitations.

```
y = {yc{:}};
```

When load balancing, the same process happens, but, instead of each dataset having the same number of samples (1000 in the previous example), the numbers of samples can be adjusted to best take advantage of the memory and speed differences of the worker host computers.

It is not required that each worker have data. If element `i` of a Composite value is undefined, worker `i` will not be used in the computation.

Single GPU Computing

The number of cores, size of memory, and speed efficiencies of GPU cards are growing rapidly with each new generation. Where video games have long benefited from improved GPU performance, these cards are now flexible enough to perform general numerical computing tasks like training neural networks.

For the latest GPU requirements, see the web page for Parallel Computing Toolbox; or query MATLAB to determine whether your PC has a supported GPU. This function returns the number of GPUs in your system:

```
count = gpuDeviceCount  
count =  
1
```

If the result is one or more, you can query each GPU by index for its characteristics. This includes its name, number of multiprocessors, SIMDWidth of each multiprocessor, and total memory.

```
gpu1 = gpuDevice(1)  
gpu1 =  
CUDADevice with properties:  
  
    Name: 'GeForce GTX 470'  
    Index: 1  
ComputeCapability: '2.0'  
    SupportsDouble: 1  
    DriverVersion: 4.1000  
MaxThreadsPerBlock: 1024  
    MaxShmemPerBlock: 49152  
MaxThreadBlockSize: [1024 1024 64]  
    MaxGridSize: [65535 65535 1]  
    SIMDWidth: 32  
    TotalMemory: 1.3422e+09  
AvailableMemory: 1.1056e+09  
MultiprocessorCount: 14  
    ClockRateKHz: 1215000  
    ComputeMode: 'Default'  
GPUOverlapsTransfers: 1  
KernelExecutionTimeout: 1  
    CanMapHostMemory: 1  
DeviceSupported: 1  
DeviceSelected: 1
```

The simplest way to take advantage of the GPU is to specify call **train** and **sim** with the parameter argument '**useGPU**' set to '**yes**' ('**no**' is the default).

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

If `net1` has the default training function `trainlm`, you see a warning that GPU calculations do not support Jacobian training, only gradient training. So the training function is automatically changed to the gradient training function `trainscg`. To avoid the notice, you can specify the function before training:

```
net1.trainFcn = 'trainscg';
```

To verify that the training and simulation occur on the GPU device, request that the computer resources be shown:

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

Each of the above lines of code outputs the following resources summary:

```
Computing Resources:
GPU device #1, GeForce GTX 470
```

Many MATLAB functions automatically execute on a GPU when any of the input arguments is a `gpuArray`. Normally you move arrays to and from the GPU with the functions `gpuArray` and `gather`. However, for neural network calculations on a GPU to be efficient, matrices need to be transposed and the columns padded so that the first element in each column aligns properly in the GPU memory. Deep Learning Toolbox provides a special function called `nndata2gpu` to move an array to a GPU and properly organize it:

```
xg = nndata2gpu(x);
tg = nndata2gpu(t);
```

Now you can train and simulate the network using the converted data already on the GPU, without having to specify the `'useGPU'` argument. Then convert and return the resulting GPU array back to MATLAB with the complementary function `gpu2nndata`.

Before training with `gpuArray` data, the network's input and outputs must be manually configured with regular MATLAB matrices using the `configure` function:

```
net2 = configure(net1,x,t); % Configure with MATLAB arrays
net2 = train(net2,xg,tg); % Execute on GPU with NNET formatted gpuArrays
yg = net2(xg); % Execute on GPU
y = gpu2nndata(yg); % Transfer array to local workspace
```

On GPUs and other hardware where you might want to deploy your neural networks, it is often the case that the exponential function `exp` is not implemented with hardware, but

with a software library. This can slow down neural networks that use the `tansig` sigmoid transfer function. An alternative function is the Elliot sigmoid function whose expression does not include a call to any higher order functions:

(equation) $a = n / (1 + \text{abs}(n))$

Before training, the network's `tansig` layers can be converted to `elliotsig` layers as follows:

```
for i=1:net.numLayers
    if strcmp(net.layers{i}.transferFcn,'tansig')
        net.layers{i}.transferFcn = 'elliotsig';
    end
end
```

Now training and simulation might be faster on the GPU and simpler deployment hardware.

Distributed GPU Computing

Distributed and GPU computing can be combined to run calculations across multiple CPUs and/or GPUs on a single computer, or on a cluster with MATLAB Parallel Server.

The simplest way to do this is to specify `train` and `sim` to do so, using the parallel pool determined by the cluster profile you use. The '`showResources`' option is especially recommended in this case, to verify that the expected hardware is being employed:

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

These lines of code use all available workers in the parallel pool. One worker for each unique GPU employs that GPU, while other workers operate as CPUs. In some cases, it might be faster to use only GPUs. For instance, if a single computer has three GPUs and four workers each, the three workers that are accelerated by the three GPUs might be speed limited by the fourth CPU worker. In these cases, you can specify that `train` and `sim` use only workers with unique GPUs.

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

As with simple distributed computing, distributed GPU computing can benefit from manually created Composite values. Defining the Composite values yourself lets you indicate which workers to use, how many samples to assign to each worker, and which workers use GPUs.

For instance, if you have four workers and only three GPUs, you can define larger datasets for the GPU workers. Here, a random dataset is created with different sample loads per Composite element:

```
numSamples = [1000 1000 1000 300];
xc = Composite;
tc = Composite;
for i=1:4
    xi = rand(2,numSamples(i));
    ti = xi(1,:).^2 + 3*xi(2,:);
    xc{i} = xi;
    tc{i} = ti;
end
```

You can now specify that `train` and `sim` use the three GPUs available:

```
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc,'useGPU','yes','showResources','yes');
yc = net2(xc,'showResources','yes');
```

To ensure that the GPUs get used by the first three workers, manually converting each worker's Composite elements to gpuArrays. Each worker performs this transformation within a parallel executing `spmd` block.

```
spmd
if labindex <= 3
    xc = nnData2gpu(xc);
    tc = nnData2gpu(tc);
end
end
```

Now the data specifies when to use GPUs, so you do not need to tell `train` and `sim` to do so.

```
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc,'showResources','yes');
yc = net2(xc,'showResources','yes');
```

Ensure that each GPU is used by only one worker, so that the computations are most efficient. If multiple workers assign gpuArray data on the same GPU, the computation will still work but will be slower, because the GPU will operate on the multiple workers' data sequentially.

Parallel Time Series

For time series networks, simply use cell array values for x and t , and optionally include initial input delay states xi and initial layer delay states ai , as required.

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

net2 = train(net1,x,t,xi,ai,'useParallel','yes')
y = net2(x,xi,ai,'useParallel','yes','useGPU','only')

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

Note that parallelism happens across samples, or in the case of time series across different series. However, if the network has only input delays, with no layer delays, the delayed inputs can be precalculated so that for the purposes of computation, the time steps become different samples and can be parallelized. This is the case for networks such as `timedelaynet` and open-loop versions of `narxnet` and `narnet`. If a network has layer delays, then time cannot be “flattened” for purposes of computation, and so single series data cannot be parallelized. This is the case for networks such as `layrecnet` and closed-loop versions of `narxnet` and `narnet`. However, if the data consists of multiple sequences, it can be parallelized across the separate sequences.

Parallel Availability, Fallbacks, and Feedback

As mentioned previously, you can query MATLAB to discover the current parallel resources that are available.

To see what GPUs are available on the host computer:

```
gpuCount = gpuDeviceCount
for i=1:gpuCount
    gpuDevice(i)
end
```

To see how many workers are running in the current parallel pool:

```
poolSize = pool.NumWorkers
```

To see the GPUs available across a parallel pool running on a PC cluster using MATLAB Parallel Server:

```
spmd
    worker.index = labindex;
    worker.name = system('hostname');
    worker.gpuCount = gpuDeviceCount;
    try
        worker.gpuInfo = gpuDevice;
    catch
        worker.gpuInfo = [];
    end
    worker
end
```

When 'useParallel' or 'useGPU' are set to 'yes', but parallel or GPU workers are unavailable, the convention is that when resources are requested, they are used if available. The computation is performed without error even if they are not. This process of falling back from requested resources to actual resources happens as follows:

- If 'useParallel' is 'yes' but Parallel Computing Toolbox is unavailable, or a parallel pool is not open, then computation reverts to single-threaded MATLAB.
- If 'useGPU' is 'yes' but the gpuDevice for the current MATLAB session is unassigned or not supported, then computation reverts to the CPU.
- If 'useParallel' and 'useGPU' are 'yes', then each worker with a unique GPU uses that GPU, and other workers revert to CPU.
- If 'useParallel' is 'yes' and 'useGPU' is 'only', then workers with unique GPUs are used. Other workers are not used, unless no workers have GPUs. In the case with no GPUs, all workers use CPUs.

When unsure about what hardware is actually being employed, check `gpuDeviceCount`, `gpuDevice`, and `pool.NumWorkers` to ensure the desired hardware is available, and call `train` and `sim` with `'showResources'` set to 'yes' to verify what resources were actually used.

Optimize Neural Network Training Speed and Memory

In this section...

- “Memory Reduction” on page 11-12
- “Fast Elliot Sigmoid” on page 11-12

Memory Reduction

Depending on the particular neural network, simulation and gradient calculations can occur in MATLAB or MEX. MEX is more memory efficient, but MATLAB can be made more memory efficient in exchange for time.

To determine whether MATLAB or MEX is being used, use the 'showResources' option, as shown in this general form of the syntax:

```
net2 = train(net1,x,t,'showResources','yes')
```

If MATLAB is being used and memory limitations are a problem, the amount of temporary storage needed can be reduced by a factor of N, in exchange for performing the computations N times sequentially on each of N subsets of the data.

```
net2 = train(net1,x,t,'reduction',N);
```

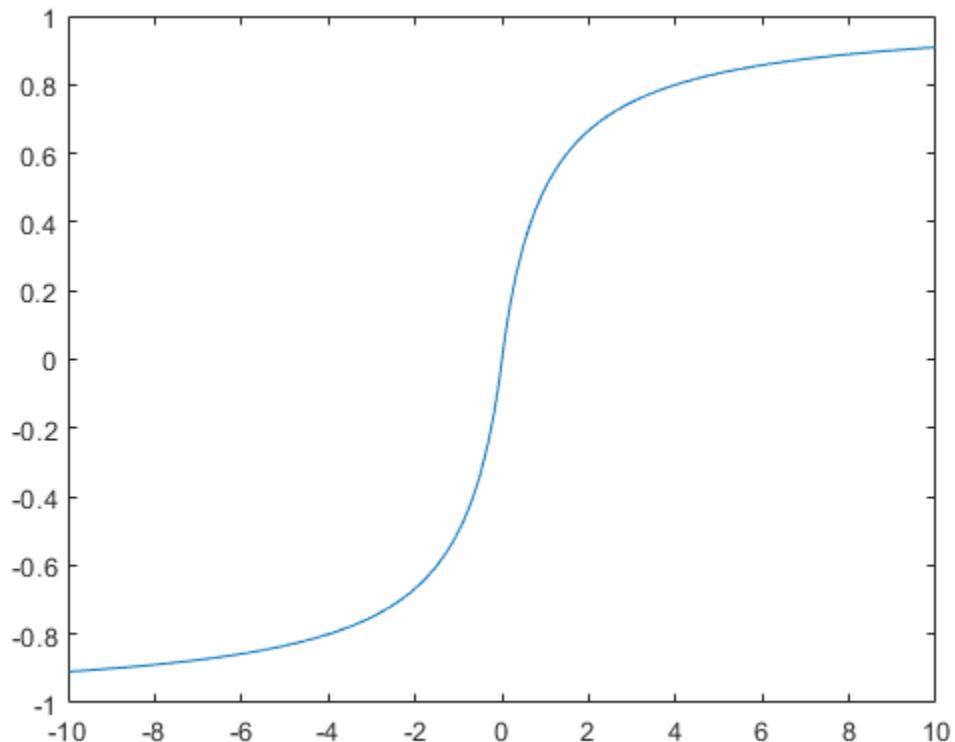
This is called memory reduction.

Fast Elliot Sigmoid

Some simple computing hardware might not support the exponential function directly, and software implementations can be slow. The Elliot sigmoid `elliotsig` function performs the same role as the symmetric sigmoid `tansig` function, but avoids the exponential function.

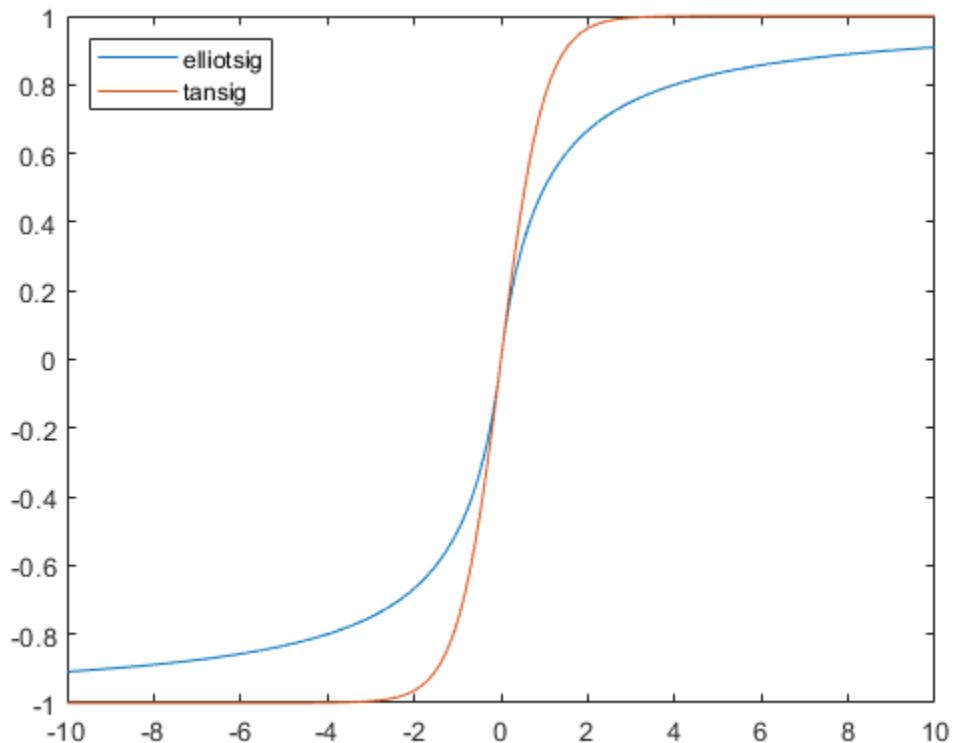
Here is a plot of the Elliot sigmoid:

```
n = -10:0.01:10;
a = elliotsig(n);
plot(n,a)
```



Next, `elliotsig` is compared with `tansig`.

```
a2 = tansig(n);
h = plot(n,a,n,a2);
legend(h,'elliotsig','tansig','Location','NorthWest')
```



To train a neural network using `elliotsig` instead of `tansig`, transform the network's transfer functions:

```
[x,t] = bodyfat_dataset;
net = feedforwardnet;
view(net)
net.layers{1}.transferFcn = 'elliotsig';
view(net)
net = train(net,x,t);
y = net(x)
```

Here, the times to execute `elliotsig` and `tansig` are compared. `elliotsig` is approximately four times faster on the test system.

```
n = rand(5000,5000);
tic,for i=1:100,a=tansig(n); end, tansigTime = toc;
tic,for i=1:100,a=elliotsig(n); end, elliotTime = toc;
speedup = tansigTime / elliotTime

speedup =
4.1406
```

However, while simulation is faster with `elliotsig`, training is not guaranteed to be faster, due to the different shapes of the two transfer functions. Here, 10 networks are each trained for `tansig` and `elliotsig`, but training times vary significantly even on the same problem with the same network.

```
[x,t] = bodyfat_dataset;
tansigNet = feedforwardnet;
tansigNet.trainParam.showWindow = false;
elliotNet = tansigNet;
elliotNet.layers{1}.transferFcn = 'elliotsig';
for i=1:10, tic, net = train(tansigNet,x,t); tansigTime = toc, end
for i=1:10, tic, net = train(elliotNet,x,t), elliotTime = toc, end
```

Choose a Multilayer Neural Network Training Function

In this section...

- “SIN Data Set” on page 11-17
- “PARITY Data Set” on page 11-19
- “ENGINE Data Set” on page 11-22
- “CANCER Data Set” on page 11-24
- “CHOLESTEROL Data Set” on page 11-26
- “DIABETES Data Set” on page 11-28
- “Summary” on page 11-30

It is very difficult to know which training algorithm will be the fastest for a given problem. It depends on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). This section compares the various training algorithms. Feedforward networks are trained on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple “toy” problems, while the other four are “real world” problems. Networks with a variety of different architectures and complexities are used, and the networks are trained to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms used to identify them.

Acronym	Algorithm	Description
LM	<code>trainlm</code>	Levenberg-Marquardt
BFG	<code>trainbfg</code>	BFGS Quasi-Newton
RP	<code>trainrp</code>	Resilient Backpropagation
SCG	<code>trainscg</code>	Scaled Conjugate Gradient
CGB	<code>traincgb</code>	Conjugate Gradient with Powell/Beale Restarts
CGF	<code>traincfg</code>	Fletcher-Powell Conjugate Gradient
CGP	<code>traincgp</code>	Polak-Ribi�re Conjugate Gradient

Acronym	Algorithm	Description
OSS	<code>trainoss</code>	One Step Secant
GDX	<code>traingdx</code>	Variable Learning Rate Backpropagation

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

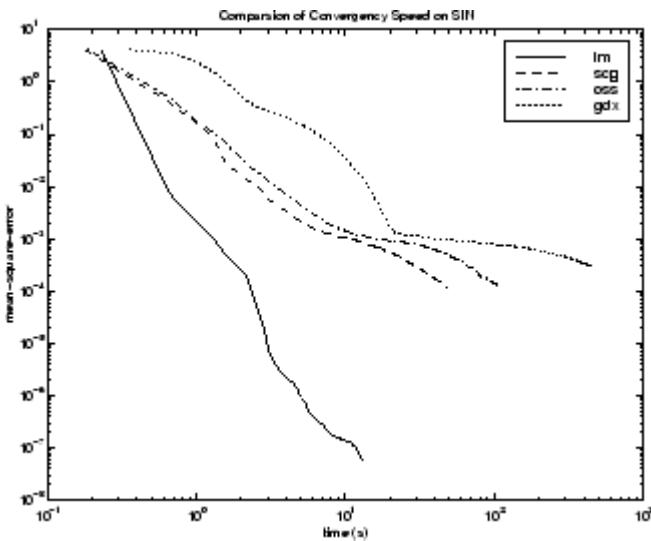
Problem Title	Problem Type	Network Structure	Error Goal	Computer
SIN	Function approximation	1-5-1	0.002	Sun Sparc 2
PARITY	Pattern recognition	3-10-10-1	0.001	Sun Sparc 2
ENGINE	Function approximation	2-30-2	0.005	Sun Enterprise 4000
CANCER	Pattern recognition	9-5-5-2	0.012	Sun Sparc 2
CHOLESTEROL	Function approximation	21-15-3	0.027	Sun Sparc 20
DIABETES	Pattern recognition	8-15-15-2	0.05	Sun Sparc 20

SIN Data Set

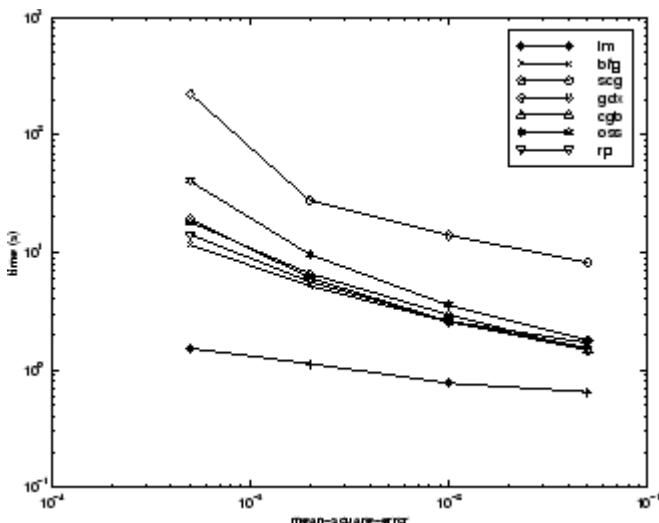
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with `tansig` transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training the network using nine different training algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.002. The fastest algorithm for this problem is the Levenberg-Marquardt algorithm. On the average, it is over four times faster than the next fastest algorithm. This is the type of problem for which the LM algorithm is best suited—a function approximation problem where the network has fewer than one hundred weights and the approximation must be very accurate.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	1.14	1.00	0.65	1.83	0.38
BFG	5.22	4.58	3.17	14.38	2.08
RP	5.67	4.97	2.66	17.24	3.72
SCG	6.09	5.34	3.18	23.64	3.81
CGB	6.61	5.80	2.99	23.65	3.67
CGF	7.86	6.89	3.57	31.23	4.76
CGP	8.24	7.23	4.07	32.32	5.03
OSS	9.64	8.46	3.97	59.63	9.79
GDX	27.69	24.29	17.21	258.15	43.65

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is shown in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here you can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Here you can see that as the error goal is reduced, the improvement provided by the LM algorithm becomes more pronounced. Some algorithms perform better as the error goal is reduced (LM and BFG), and other algorithms degrade as the error goal is reduced (OSS and GDX).



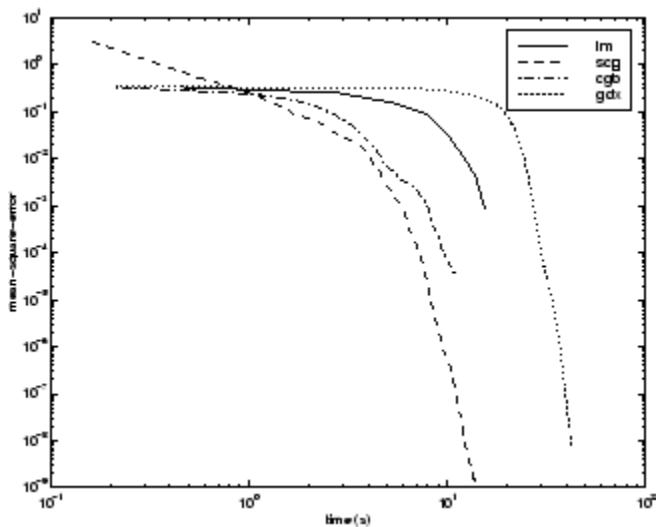
PARITY Data Set

The second benchmark problem is a simple pattern recognition problem—detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a 1; otherwise, it should output a -1. The network used for this problem is a 3-10-10-1 network with tansig neurons in each layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Because the output neurons in

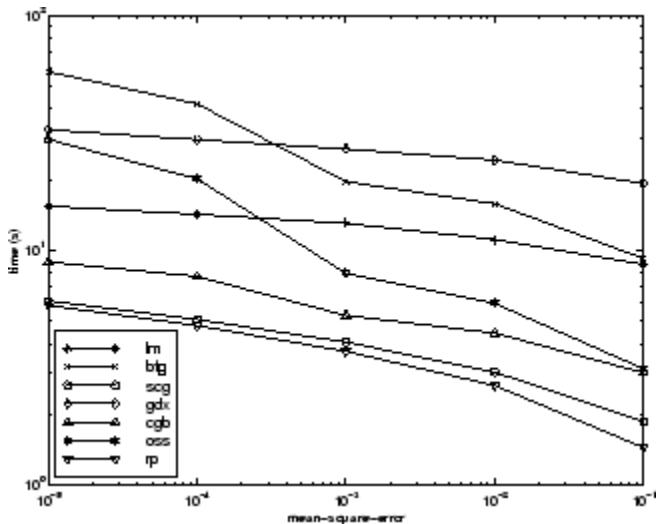
pattern recognition problems are generally saturated, you will not be operating in the linear region.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	3.73	1.00	2.35	6.89	1.26
SCG	4.09	1.10	2.36	7.48	1.56
CGP	5.13	1.38	3.50	8.73	1.05
CGB	5.30	1.42	3.91	11.59	1.35
CGF	6.62	1.77	3.96	28.05	4.32
OSS	8.00	2.14	5.06	14.41	1.92
LM	13.07	3.50	6.48	23.78	4.96
BFG	19.68	5.28	14.19	26.64	2.85
GDX	27.07	7.26	25.21	28.52	0.86

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is shown in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG).

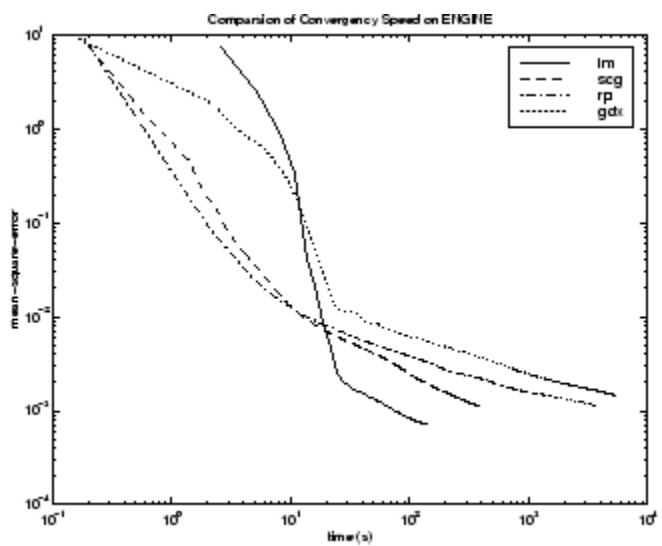


ENGINE Data Set

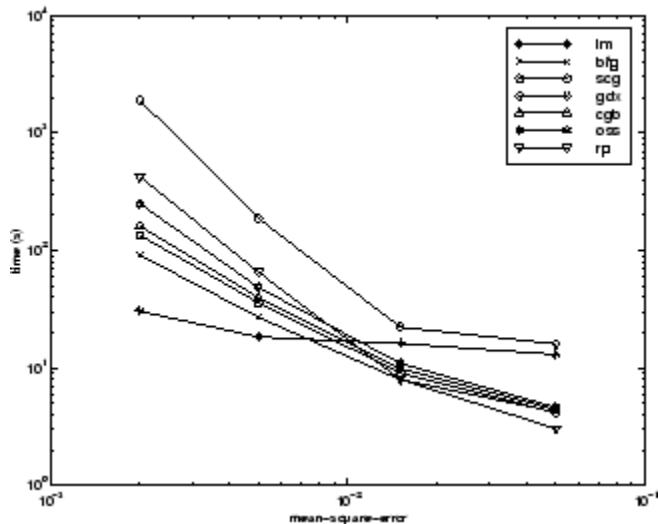
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tansig neurons in the hidden layer and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus 16), and the advantages of the LM algorithm decrease as the number of network parameters increases.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	18.45	1.00	12.01	30.03	4.27
BFG	27.12	1.47	16.42	47.36	5.95
SCG	36.02	1.95	19.39	52.45	7.78
CGF	37.93	2.06	18.89	50.34	6.12
CGB	39.93	2.16	23.33	55.42	7.50
CGP	44.30	2.40	24.99	71.55	9.89
OSS	48.71	2.64	23.51	80.90	12.33
RP	65.91	3.57	31.83	134.31	34.24
GDX	188.50	10.22	81.59	279.90	66.67

The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (GDX and RP), while the LM algorithm improves.



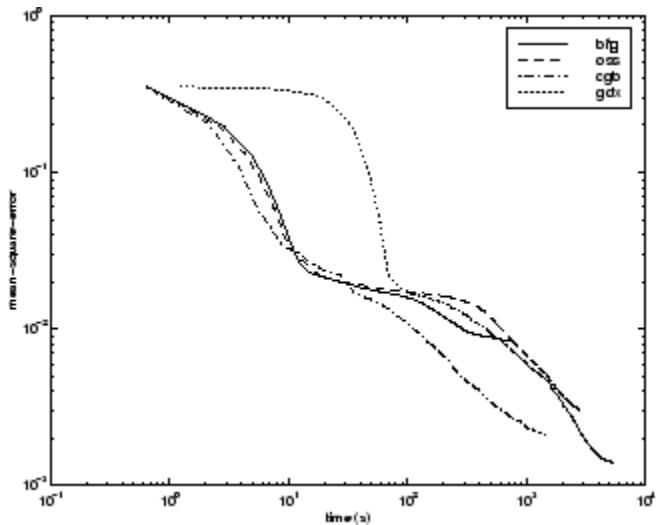
CANCER Data Set

The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. The data was obtained from the University of Wisconsin Hospitals, Madison, from Dr. William H. Wolberg. The network used for this problem is a 9-5-5-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

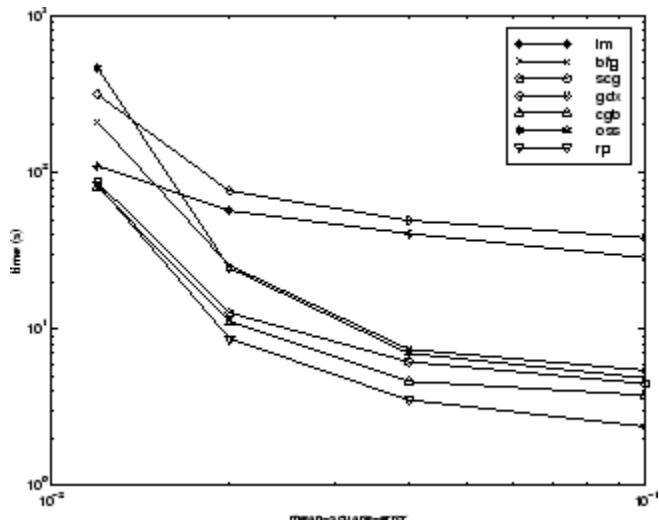
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
CGB	80.27	1.00	55.07	102.31	13.17
RP	83.41	1.04	59.51	109.39	13.44
SCG	86.58	1.08	41.21	112.19	18.25
CGP	87.70	1.09	56.35	116.37	18.03
CGF	110.05	1.37	63.33	171.53	30.13
LM	110.33	1.37	58.94	201.07	38.20
BFG	209.60	2.61	118.92	318.18	58.44
GDX	313.22	3.90	166.48	446.43	75.44
OSS	463.87	5.78	250.62	599.99	97.35

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem there is not as much variation in performance as in previous problems.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG) while the LM algorithm improves. It is typical of the LM algorithm on any problem that its performance improves relative to other algorithms as the error goal is reduced.



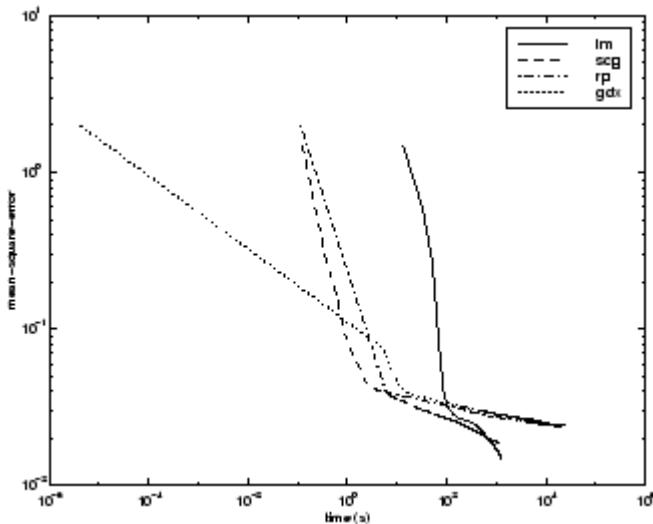
CHOLESTEROL Data Set

The fifth benchmark problem is a realistic function approximation (or nonlinear regression) problem. The objective of the network is to predict cholesterol levels (ldl, hdl, and vldl) based on measurements of 21 spectral components. The data was obtained from Dr. Neil Purdie, Department of Chemistry, Oklahoma State University [PuLu92 on page 14-2]. The network used for this problem is a 21-15-3 network with tansig neurons in the hidden layers and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 20 different trials (10 trials for RP and GDX), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.027.

The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases in the network has increased again (378 versus 152 versus 16). As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

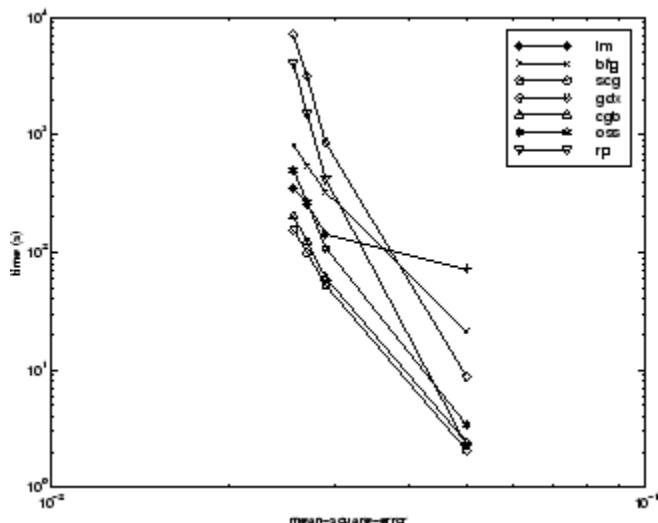
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
SCG	99.73	1.00	83.10	113.40	9.93
CGP	121.54	1.22	101.76	162.49	16.34
CGB	124.06	1.2	107.64	146.90	14.62
CGF	136.04	1.36	106.46	167.28	17.67
LM	261.50	2.62	103.52	398.45	102.06
OSS	268.55	2.69	197.84	372.99	56.79
BFG	550.92	5.52	471.61	676.39	46.59
RP	1519.00	15.23	581.17	2256.10	557.34
GDX	3169.50	31.78	2514.90	4168.20	610.52

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, you can see that the LM algorithm is able to drive the mean square error to a lower level than the other algorithms. The SCG and RP algorithms provide the fastest initial convergence.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence

goal. You can see that the LM and BFG algorithms improve relative to the other algorithms as the error goal is reduced.



DIABETES Data Set

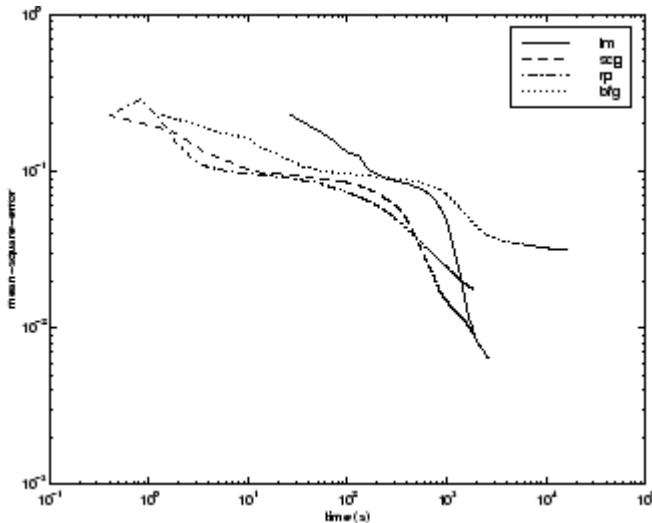
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance test, etc.). The data was obtained from the University of California, Irvine, machine learning data base. The network used for this problem is an 8-15-15-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 10 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.05.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems considered. The RP algorithm works well on all the pattern recognition problems. This is reasonable, because that algorithm was designed to overcome the difficulties caused by training with sigmoid functions, which have very small slopes when operating far from the center point. For pattern recognition problems,

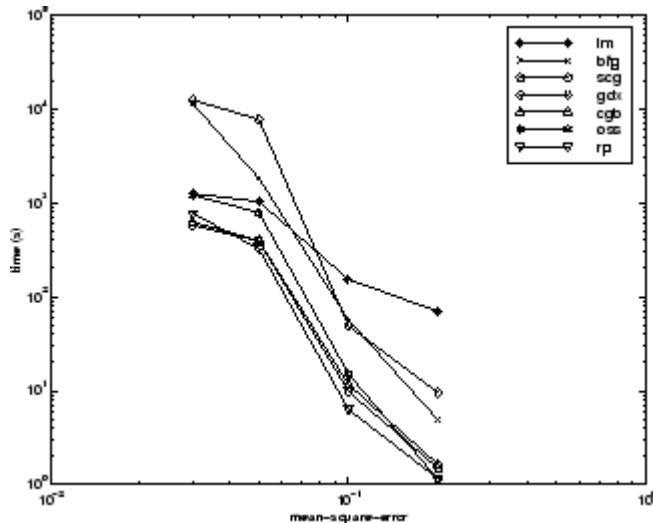
you use sigmoid transfer functions in the output layer, and you want the network to operate at the tails of the sigmoid function.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	323.90	1.00	187.43	576.90	111.37
SCG	390.53	1.21	267.99	487.17	75.07
CGB	394.67	1.22	312.25	558.21	85.38
CGP	415.90	1.28	320.62	614.62	94.77
OSS	784.00	2.42	706.89	936.52	76.37
CGF	784.50	2.42	629.42	1082.20	144.63
LM	1028.10	3.17	802.01	1269.50	166.31
BFG	1821.00	5.62	1415.80	3254.50	546.36
GDX	7687.00	23.73	5169.20	10350.00	2015.00

The following figure plots the mean square error versus execution time for some typical algorithms. As with other problems, you see that the SCG and RP have fast initial convergence, while the LM algorithm is able to provide smaller final error.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. In this case, you can see that the BFG algorithm degrades as the error goal is reduced, while the LM algorithm improves. The RP algorithm is best, except at the smallest error goal, where SCG is better.



Summary

There are several algorithm characteristics that can be deduced from the experiments described. In general, on function approximation problems, for networks that contain up to a few hundred weights, the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. In many cases, `trainlm` is able to obtain lower mean square errors than any of the other algorithms tested. However, as the number of weights in the network increases, the advantage of `trainlm` decreases. In addition, `trainlm` performance is relatively poor on pattern recognition problems. The storage requirements of `trainlm` are larger than the other algorithms tested.

The `trainrp` function is the fastest algorithm on pattern recognition problems. However, it does not perform well on function approximation problems. Its performance also degrades as the error goal is reduced. The memory requirements for this algorithm are relatively small in comparison to the other algorithms considered.

The conjugate gradient algorithms, in particular `trainscg`, seem to perform well over a wide variety of problems, particularly for networks with a large number of weights. The SCG algorithm is almost as fast as the LM algorithm on function approximation problems (faster for large networks) and is almost as fast as `trainrp` on pattern recognition problems. Its performance does not degrade as quickly as `trainrp` performance does when the error is reduced. The conjugate gradient algorithms have relatively modest memory requirements.

The performance of `trainbfg` is similar to that of `trainlm`. It does not require as much storage as `trainlm`, but the computation required does increase geometrically with the size of the network, because the equivalent of a matrix inverse must be computed at each iteration.

The variable learning rate algorithm `traingdx` is usually much slower than the other methods, and has about the same storage requirements as `trainrp`, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping you can have inconsistent results if you use an algorithm that converges too quickly. You might overshoot the point at which the error on the validation set is minimized.

Improve Shallow Neural Network Generalization and Avoid Overfitting

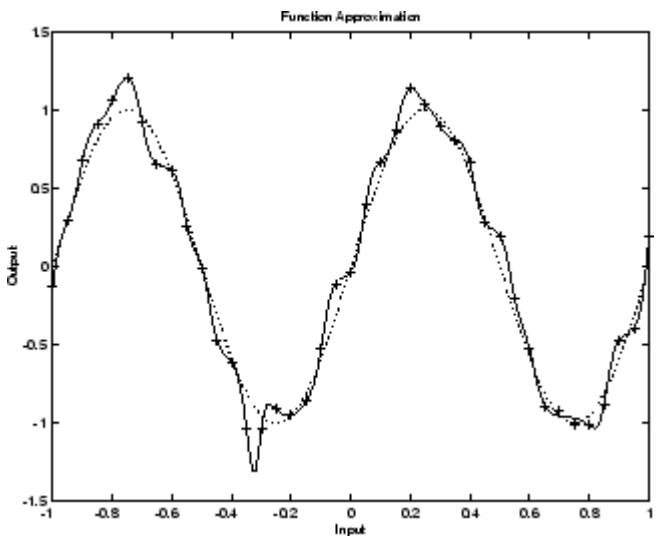
In this section...

- “Retraining Neural Networks” on page 11-34
- “Multiple Neural Networks” on page 11-35
- “Early Stopping” on page 11-36
- “Index Data Division (divideind)” on page 11-37
- “Random Data Division (dividerand)” on page 11-37
- “Block Data Division (divideblock)” on page 11-37
- “Interleaved Data Division (divideint)” on page 11-38
- “Regularization” on page 11-38
- “Summary and Discussion of Early Stopping and Regularization” on page 11-41
- “Posttraining Analysis (regression)” on page 11-43

Tip To learn how to set up parameters for a deep learning network, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-52.

One of the problems that occur during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network that has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the + symbols, and the neural network response is given by the solid line. Clearly this network has overfitted the data and will not generalize well.



One method for improving network generalization is to use a network that is just large enough to provide an adequate fit. The larger network you use, the more complex the functions the network can create. If you use a small enough network, it will not have enough power to overfit the data. Run the *Neural Network Design* example `nnd1gn` [HDB96 on page 14-2] to investigate how reducing the size of a network can prevent overfitting.

Unfortunately, it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving generalization that are implemented in Deep Learning Toolbox software: regularization and early stopping. The next sections describe these two techniques and the routines to implement them.

Note that if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. If you can easily collect more data and increase the size of the training set, then there is no need to worry about the following techniques to prevent overfitting. The rest of this section only applies to those situations in which you want to make the most of a limited supply of data.

Retraining Neural Networks

Typically each backpropagation training session starts with different initial weights and biases, and different divisions of data into training, validation, and test sets. These different conditions can lead to very different solutions for the same problem.

It is a good idea to train several networks to ensure that a network with good generalization is found.

Here a dataset is loaded and divided into two parts: 90% for designing networks and 10% for testing them all.

```
[x, t] = bodyfat_dataset;
Q = size(x, 2);
Q1 = floor(Q * 0.90);
Q2 = Q - Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1 + (1:Q2));
x1 = x(:, ind1);
t1 = t(:, ind1);
x2 = x(:, ind2);
t2 = t(:, ind2);
```

Next a network architecture is chosen and trained ten times on the first part of the dataset, with each network's mean square error on the second part of the dataset.

```
net = feedforwardnet(10);
numNN = 10;
NN = cell(1, numNN);
perfs = zeros(1, numNN);
for i = 1:numNN
    fprintf('Training %d/%d\n', i, numNN);
    NN{i} = train(net, x1, t1);
    y2 = NN{i}(x2);
    perfs(i) = mse(net, t2, y2);
end
```

Each network will be trained starting from different initial weights and biases, and with a different division of the first dataset into training, validation, and test sets. Note that the test sets are a good measure of generalization for each respective network, but not for all the networks, because data that is a test set for one network will likely be used for training or validation by other neural networks. This is why the original dataset was divided into two parts, to ensure that a completely independent test set is preserved.

The neural network with the lowest performance is the one that generalized best to the second part of the dataset.

Multiple Neural Networks

Another simple way to improve generalization, especially when caused by noisy data or a small dataset, is to train multiple neural networks and average their outputs.

For instance, here 10 neural networks are trained on a small problem and their mean squared errors compared to the means squared error of their average.

First, the dataset is loaded and divided into a design and test set.

```
[x, t] = bodyfat_dataset;
Q = size(x, 2);
Q1 = floor(Q * 0.90);
Q2 = Q - Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1 + (1:Q2));
x1 = x(:, ind1);
t1 = t(:, ind1);
x2 = x(:, ind2);
t2 = t(:, ind2);
```

Then, ten neural networks are trained.

```
net = feedforwardnet(10);
numNN = 10;
nets = cell(1, numNN);
for i = 1:numNN
    fprintf('Training %d/%d\n', i, numNN)
    nets{i} = train(net, x1, t1);
end
```

Next, each network is tested on the second dataset with both individual performances and the performance for the average output calculated.

```
perfs = zeros(1, numNN);
y2Total = 0;
for i = 1:numNN
    neti = nets{i};
    y2 = neti(x2);
    perfs(i) = mse(neti, t2, y2);
```

```
y2Total = y2Total + y2;
end
perfs
y2AverageOutput = y2Total / numNN;
perfAveragedOutputs = mse(nets{1}, t2, y2AverageOutput)
```

The mean squared error for the average output is likely to be lower than most of the individual performances, perhaps not all. It is likely to generalize better to additional new data.

For some very difficult problems, a hundred networks can be trained and the average of their outputs taken for any input. This is especially helpful for a small, noisy dataset in conjunction with the Bayesian Regularization training function `trainbr`, described below.

Early Stopping

The default method for improving generalization is called *early stopping*. This technique is automatically provided for all of the supervised network creation functions, including the backpropagation network creation functions such as `feedforwardnet`.

In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. When the validation error increases for a specified number of iterations (`net.trainParam.max_fail`), the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of these functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

Index Data Division (divideind)

Create a simple test problem. For the full data set, generate a noisy sine wave with 201 input points ranging from -1 to 1 at steps of 0.01:

```
p = [-1:0.01:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Divide the data by index so that successive samples are assigned to the training set, validation set, and test set successively:

```
trainInd = 1:3:201
valInd = 2:3:201;
testInd = 3:3:201;
[trainP,valP,testP] = divideind(p,trainInd,valInd,testInd);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Random Data Division (dividerand)

You can divide the input data randomly so that 60% of the samples are assigned to the training set, 20% to the validation set, and 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = dividerand(p);
```

This function not only divides the input data, but also returns indices so that you can divide the target data accordingly using `divideind`:

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Block Data Division (divideblock)

You can also divide the input data randomly such that the first 60% of the samples are assigned to the training set, the next 20% to the validation set, and the last 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = divideblock(p);
```

Divide the target data accordingly using `divideind`:

```
[trainT, valT, testT] = divideind(t, trainInd, valInd, testInd);
```

Interleaved Data Division (`divideint`)

Another way to divide the input data is to cycle samples between the training set, validation set, and test set according to percentages. You can interleave 60% of the samples to the training set, 20% to the validation set and 20% to the test set as follows:

```
[trainP, valP, testP, trainInd, valInd, testInd] = divideint(p);
```

Divide the target data accordingly using `divideind`.

```
[trainT, valT, testT] = divideind(t, trainInd, valInd, testInd);
```

Regularization

Another method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next section explains how the performance function can be modified, and the following section describes a routine that automatically sets the optimal performance function to achieve the best generalization.

Modified Performance Function

The typical performance function used for training feedforward neural networks is the mean sum of squares of the network errors.

$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2$$

It is possible to improve generalization if you modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases $msereg = \gamma * msw + (1 - \gamma) * mse$, where γ is the performance ratio, and

$$msw = \frac{1}{n} \sum_{j=1}^n w_j^2$$

Using this performance function causes the network to have smaller weights and biases, and this forces the network response to be smoother and less likely to overfit.

The following code reinitializes the previous network and retrains it using the BFGS algorithm with the regularized performance function. Here the performance ratio is set to 0.5, which gives equal weight to the mean square errors and the mean square weights.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10,'trainbfg');
net.divideFcn = '';
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
net.performParam.regularization = 0.5;
net = train(net,x,t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If you make this parameter too large, you might get overfitting. If the ratio is too small, the network does not adequately fit the training data. The next section describes a routine that automatically sets the regularization parameters.

Automated Regularization (trainbr)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92 on page 14-2]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. You can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this user guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97 on page 14-2].

Bayesian regularization has been implemented in the function `trainbr`. The following code shows how you can train a 1-20-1 network using this function to approximate the noisy sine wave shown in the figure in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 11-32. (Data division is cancelled by setting `net.divideFcn` so that the effects of `trainbr` are isolated from early stopping.)

```
x = -1:0.05:1;
t = sin(2*pi*x) + 0.1*randn(size(x));
```

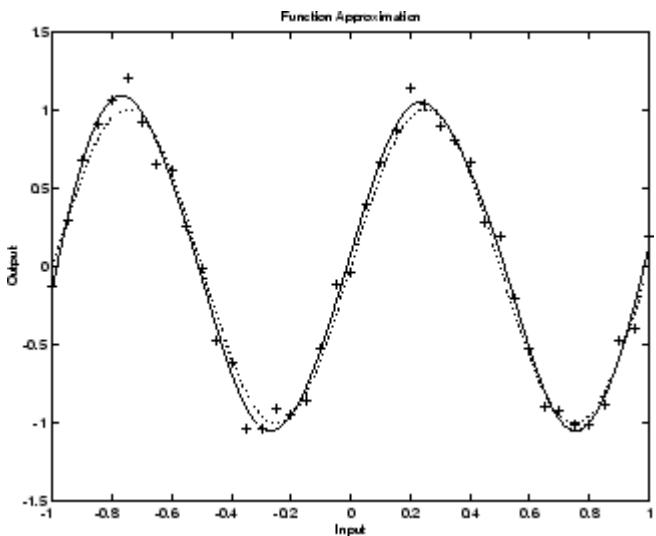
```
net = feedforwardnet(20,'trainbr');
net = train(net,x,t);
```

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by #Par in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The `trainbr` algorithm generally works best when the network inputs and targets are scaled so that they fall approximately in the range $[-1,1]$. That is the case for the test problem here. If your inputs and targets do not fall in this range, you can use the function `mapminmax` or `mapstd` to perform the scaling, as described in “Choose Neural Network Input-Output Processing Functions” on page 5-9. Networks created with `feedforwardnet` include `mapminmax` as an input and output processing function by default.

The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfits the data, here you see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. You could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using `trainbr`, it is important to let the algorithm run until the effective number of parameters has converged. The training might stop with the message “Maximum MU reached.” This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively constant over several iterations. When this occurs you might want to click the **Stop Training** button in the training window.



Summary and Discussion of Early Stopping and Regularization

Early stopping and regularization can ensure network generalization when you apply them properly.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), set the training parameters so that the convergence is relatively slow. For example, set `mu` to a relatively large value, such as 1, and set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively. The training functions `trainscg` and `trainbr` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

When you use Bayesian regularization, it is important to train the network until it reaches convergence. The sum-squared error, the sum-squared weights, and the effective number of parameters should reach constant values when the network has converged.

With both early stopping and regularization, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

When the data set is small and you are training function approximation networks, Bayesian regularization provides better generalization performance than early stopping. This is because Bayesian regularization does not require that a validation data set be separate from the training data set; it uses all the data.

To provide some insight into the performance of the algorithms, both early stopping and Bayesian regularization were tested on several benchmark data sets, which are listed in the following table.

Data Set Title	Number of Points	Network	Description
BALL	67	2-10-1	Dual-sensor calibration for a ball position measurement
SINE (5% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 5% level
SINE (2% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 2% level
ENGINE (ALL)	1199	2-30-2	Engine sensor—full data set
ENGINE (1/4)	300	2-30-2	Engine sensor—1/4 of data set
CHOLEST (ALL)	264	5-15-3	Cholesterol measurement—full data set
CHOLEST (1/2)	132	5-15-3	Cholesterol measurement—1/2 data set

These data sets are of various sizes, with different numbers of inputs and targets. With two of the data sets the networks were trained once using all the data and then retrained using only a fraction of the data. This illustrates how the advantage of Bayesian regularization becomes more noticeable when the data sets are smaller. All the data sets are obtained from physical systems except for the SINE data sets. These two were artificially created by adding various levels of noise to a single cycle of a sine wave. The performance of the algorithms on these two data sets illustrates the effect of noise.

The following table summarizes the performance of early stopping (ES) and Bayesian regularization (BR) on the seven test sets. (The `trainscg` algorithm was used for the early stopping tests. Other algorithms provide similar performance.)

Mean Squared Test Set Error

Method	Ball	Engine (All)	Engine (1/4)	Choles (All)	Choles (1/2)	Sine (5% N)	Sine (2% N)
ES	1.2e-1	1.3e-2	1.9e-2	1.2e-1	1.4e-1	1.7e-1	1.3e-1
BR	1.3e-3	2.6e-3	4.7e-3	1.2e-1	9.3e-2	3.0e-2	6.3e-3
ES/BR	92	5	4	1	1.5	5.7	21

You can see that Bayesian regularization performs better than early stopping in most cases. The performance improvement is most noticeable when the data set is small, or if there is little noise in the data set. The BALL data set, for example, was obtained from sensors that had very little noise.

Although the generalization performance of Bayesian regularization is often better than early stopping, this is not always the case. In addition, the form of Bayesian regularization implemented in the toolbox does not perform as well on pattern recognition problems as it does on function approximation problems. This is because the approximation to the Hessian that is used in the Levenberg-Marquardt algorithm is not as accurate when the network output is saturated, as would be the case in pattern recognition problems. Another disadvantage of the Bayesian regularization method is that it generally takes longer to converge than early stopping.

Posttraining Analysis (regression)

The performance of a trained network can be measured to some extent by the errors on the training, validation, and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine `regression` is designed to perform this analysis.

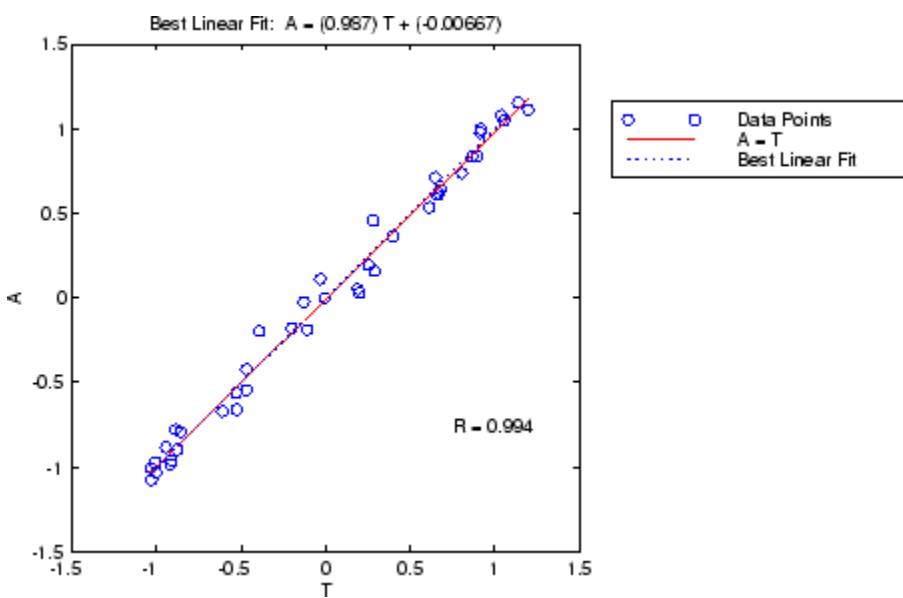
The following commands illustrate how to perform a regression analysis on a network trained.

```
x = [-1:.05:1];
t = sin(2*pi*x)+0.1*randn(size(x));
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
[r,m,b] = regression(t,y)
```

```
r =  
    0.9935  
m =  
    0.9874  
b =  
    -0.0067
```

The network output and the corresponding targets are passed to **regression**. It returns three parameters. The first two, **m** and **b**, correspond to the slope and the y-intercept of the best linear regression relating targets to network outputs. If there were a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. In this example, you can see that the numbers are very close. The third variable returned by **regression** is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In the example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by **regression**. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from the perfect fit line because the fit is so good.



Edit Shallow Neural Network Properties

In this section...

- “Custom Network” on page 11-46
- “Network Definition” on page 11-47
- “Network Behavior” on page 11-57

Tip To learn how to define your own layers for deep learning networks, see “Define Custom Deep Learning Layers” on page 1-77.

Deep Learning Toolbox software provides a flexible network object type that allows many kinds of networks to be created and then used with functions such as `init`, `sim`, and `train`.

Type the following to see all the network creation functions in the toolbox.

```
help nnetwork
```

This flexibility is possible because networks have an object-oriented representation. The representation allows you to define various architectures and assign various algorithms to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

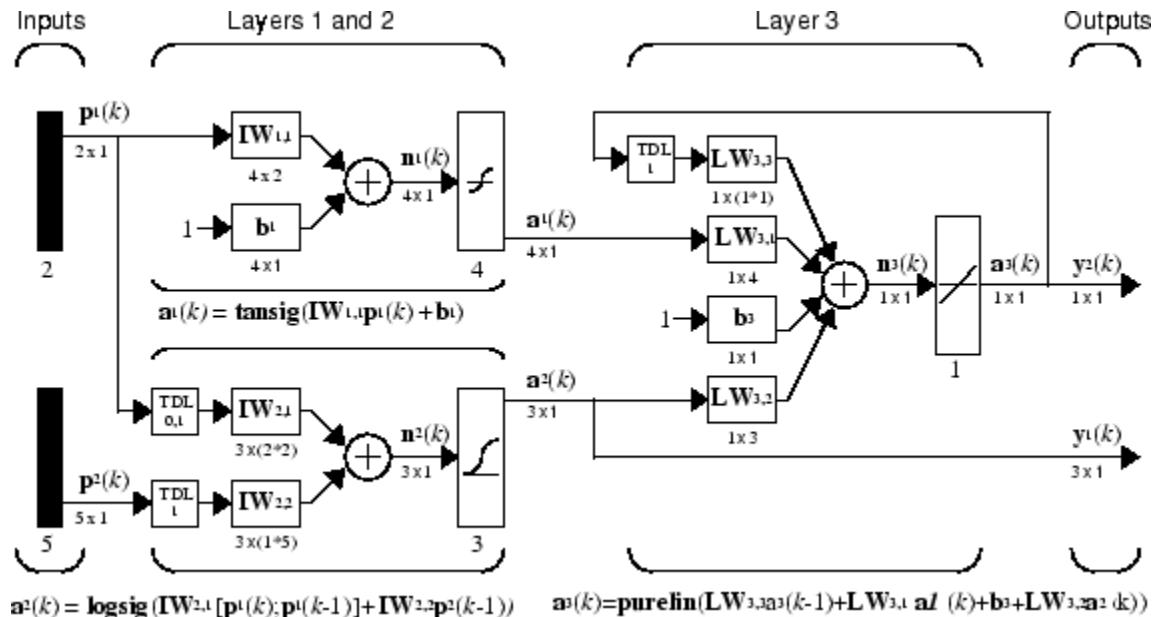
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network.

The following sections show how to create a custom network by using these properties.

Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input ranges from -2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

Each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method (`initnw`). The network is trained with Levenberg-Marquardt backpropagation (`trainlm`), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error (`mse`).

Network Definition

The first step is to create a new network. Type the following code to create a network and view its many properties:

```
net = network
```

Architecture Properties

The first group of properties displayed is labeled architecture properties. These properties allow you to select the number of inputs and layers and their connections.

Number of Inputs and Layers

The first two properties displayed in the dimensions group are `numInputs` and `numLayers`. These properties allow you to select how many inputs and layers you want the network to have.

```
net =  
  
    dimensions:  
        numInputs: 0  
        numLayers: 0  
        ...
```

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in the custom network diagram.

```
net.numInputs = 2;  
net.numLayers = 3;
```

`net.numInputs` is the number of input sources, not the number of elements in an input vector (`net.inputs{i}.size`).

Bias Connections

Type `net` and press **Enter** to view its properties again. The network now has two inputs and three layers.

```
net =  
    Neural Network:  
    dimensions:  
        numInputs: 2  
        numLayers: 3
```

Examine the next four properties in the connections group:

```
biasConnect: [0; 0; 0]  
inputConnect: [0 0; 0 0; 0 0]
```

```
layerConnect: [0 0 0; 0 0 0; 0 0 0]
outputConnect: [0 0 0]
```

These matrices of 1s and 0s represent the presence and absence of bias, input weight, layer weight, and output connections. They are currently all zeros, indicating that the network does not have any such connections.

The bias connection matrix is a 3-by-1 vector. To create a bias connection to the *i*th layer you can set `net.biasConnect(i)` to 1. Specify that the first and third layers are to have bias connections, as the diagram indicates, by typing the following code:

```
net.biasConnect(1) = 1;
net.biasConnect(3) = 1;
```

You could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

Input and Layer Weight Connections

The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the *i*th layer from the *j*th input.

To connect the first input to the first and second layers, and the second input to the second layer (as indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;
net.inputConnect(2,1) = 1;
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i,j)` represents the presence of a layer-weight connection going to the *i*th layer from the *j*th layer. Connect layers 1, 2, and 3 to layer 3 as follows:

```
net.layerConnect = [0 0 0; 0 0 0; 1 1 1];
```

Output Connections

The output connections are a 1-by-3 matrix, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to the network output, type

```
net.outputConnect = [0 1 1];
```

Number of Outputs

Type `net` and press **Enter** to view the updated properties. The final three architecture properties are read-only values, which means their values are determined by the choices made for other properties. The first read-only property in the dimension group is the number of outputs:

```
numOutputs: 2
```

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

Subobject Properties

The next group of properties in the output display is subobjects:

```
subobjects:
    inputs: {2x1 cell array of 2 inputs}
    layers: {3x1 cell array of 3 layers}
    outputs: {1x3 cell array of 2 outputs}
    biases: {3x1 cell array of 2 biases}
    inputWeights: {3x2 cell array of 3 weights}
    layerWeights: {3x3 cell array of 3 weights}
```

Inputs

When you set the number of inputs (`net.numInputs`) to 2, the `inputs` property becomes a cell array of two input structures. Each *i*th input structure (`net.inputs{i}`) contains additional properties associated with the *i*th input.

To see how the input structures are arranged, type

```
net.inputs
ans =
[1x1 nnetInput]
[1x1 nnetInput]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows:

```
ans =
    name: 'Input'
    feedbackOutput: []
    processFcns: {}
    processParams: {1x0 cell array of 0 params}
    processSettings: {0x0 cell array of 0 settings}
    processedRange: []
    processedSize: 0
    range: []
    size: 0
    userdata: (your custom info)
```

If you set the `exampleInput` property, the `range`, `size`, `processedSize`, and `processedRange` properties will automatically be updated to match the properties of the value of `exampleInput`.

Set the `exampleInput` property as follows:

```
net.inputs{1}.exampleInput = [0 10 5; 0 3 10];
```

If you examine the structure of the first input again, you see that it now has new values.

The property `processFcns` can be set to one or more processing functions. Type `help nnprocess` to see a list of these functions.

Set the second input vector ranges to be from -2 to 2 for five elements as follows:

```
net.inputs{1}.processFcns = {'removeconstantrows', 'mapminmax'};
```

View the new input properties. You will see that `processParams`, `processSettings`, `processedRange` and `processedSize` have all been updated to reflect that inputs will be processed using `removeconstantrows` and `mapminmax` before being given to the network when the network is simulated or trained. The property `processParams` contains the default parameters for each processing function. You can alter these values, if you like. See the reference page for each processing function to learn more about their parameters.

You can set the size of an input directly when no processing functions are used:

```
net.inputs{2}.size = 5;
```

Layers

When you set the number of layers (`net.numLayers`) to 3, the `layers` property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}
ans =
    Neural Network Layer

        name: 'Layer'
        dimensions: 0
        distanceFcn: (none)
        distanceParam: (none)
        distances: []
        initFcn: 'initwb'
        netInputFcn: 'netsum'
        netInputParam: (none)
        positions: []
        range: []
        size: 0
        topologyFcn: (none)
        transferFcn: 'purelin'
        transferParam: (none)
        userdata: (your custom info)
```

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to `tansig`, and its initialization function to the Nguyen-Widrow function, as required for the custom network diagram.

```
net.layers{1}.size = 4;
net.layers{1}.transferFcn = 'tansig';
net.layers{1}.initFcn = 'initnw';
```

The second layer is to have three neurons, the `logsig` transfer function, and be initialized with `initnw`. Set the second layer's properties to the desired values as follows:

```
net.layers{2}.size = 3;
net.layers{2}.transferFcn = 'logsig';
net.layers{2}.initFcn = 'initnw';
```

The third layer's size and transfer function properties don't need to be changed, because the defaults match those shown in the network diagram. You need to set only its initialization function, as follows:

```
net.layers{3}.initFcn = 'initnw';
```

Outputs

Use this line of code to see how the `outputs` property is arranged:

```
net.outputs  
ans =  
[] [1x1 nnetOutput] [1x1 nnetOutput]
```

Note that `outputs` contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when `net.outputConnect` is set to `[0 1 1]`.

View the second layer's output structure with the following expression:

```
net.outputs{2}  
ans =  
Neural Network Output  
  
    name: 'Output'  
    feedbackInput: []  
    feedbackDelay: 0  
    feedbackMode: 'none'  
    processFcns: {}  
    processParams: {1x0 cell array of 0 params}  
    processSettings: {0x0 cell array of 0 settings}  
    processedRange: [3x2 double]  
    processedSize: 3  
        range: [3x2 double]  
        size: 3  
    userdata: (your custom info)
```

The `size` is automatically set to 3 when the second layer's size (`net.layers{2}.size`) is set to that value. Look at the third layer's output structure if you want to verify that it also has the correct `size`.

Outputs have processing properties that are automatically applied to target values before they are used by the network during training. The same processing settings are applied in reverse on layer output values before they are returned as network output values during network simulation or training.

Similar to input-processing properties, setting the `exampleOutput` property automatically causes `size`, `range`, `processedSize`, and `processedRange` to be updated. Setting `processFcns` to a cell array list of processing function names causes `processParams`, `processSettings`, `processedRange` to be updated. You can then alter the `processParam` values, if you want to.

Biases, Input Weights, and Layer Weights

Enter the following commands to see how bias and weight structures are arranged:

```
net.biases  
net.inputWeights  
net.layerWeights
```

Here are the results of typing `net.biases`:

```
ans =  
[1x1 nnetBias]  
[]  
[1x1 nnetBias]
```

Each contains a structure where the corresponding connections (`net.biasConnect`, `net.inputConnect`, and `net.layerConnect`) contain a 1.

Look at their structures with these lines of code:

```
net.biases{1}  
net.biases{3}  
net.inputWeights{1,1}  
net.inputWeights{2,1}  
net.inputWeights{2,2}  
net.layerWeights{3,1}  
net.layerWeights{3,2}  
net.layerWeights{3,3}
```

For example, typing `net.biases{1}` results in the following output:

```
initFcn: (none)  
learn: true  
learnFcn: (none)  
learnParam: (none)  
size: 4  
userdata: (your custom info)
```

Specify the weights' tap delay lines in accordance with the network diagram by setting each weight's **delays** property:

```
net.inputWeights{2,1}.delays = [0 1];
net.inputWeights{2,2}.delays = 1;
net.layerWeights{3,3}.delays = 1;
```

Network Functions

Type **net** and press **Return** again to see the next set of properties.

```
functions:
    adaptFcn: (none)
    adaptParam: (none)
    derivFcn: 'defaultderiv'
    divideFcn: (none)
    divideParam: (none)
    divideMode: 'sample'
    initFcn: 'initlay'
    performFcn: 'mse'
    performParam: .regularization, .normalization
    plotFcns: {}
    plotParams: {1x0 cell array of 0 params}
    trainFcn: (none)
    trainParam: (none)
```

Each of these properties defines a function for a basic network operation.

Set the initialization function to **initlay** so the network initializes itself according to the layer initialization functions already set to **initnw**, the Nguyen-Widrow initialization function.

```
net.initFcn = 'initlay';
```

This meets the initialization requirement of the network.

Set the performance function to **mse** (mean squared error) and the training function to **trainlm** (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = 'mse';
net.trainFcn = 'trainlm';
```

Set the divide function to **dividerand** (divide training data randomly).

```
net.divideFcn = 'dividerand';
```

During supervised training, the input and target data are randomly divided into training, test, and validation data sets. The network is trained on the training data until its performance begins to decrease on the validation data, which signals that generalization has peaked. The test data provides a completely independent test of network generalization.

Set the plot functions to `plotperform` (plot training, validation and test performance) and `plottrainstate` (plot the state of the training algorithm with respect to epochs).

```
net.plotFcns = {'plotperform','plottrainstate'};
```

Weight and Bias Values

Before initializing and training the network, type `net` and press **Return**, then look at the weight and bias group of network properties.

```
weight and bias values:  
IW: {3x2 cell} containing 3 input weight matrices  
LW: {3x3 cell} containing 3 layer weight matrices  
b: {3x1 cell} containing 2 bias vectors
```

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (`net.inputConnect`, `net.layerConnect`, `net.biasConnect`) contain 1s and the subobject properties (`net.inputWeights`, `net.layerWeights`, `net.biases`) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}  
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}  
net.b{1}, net.b{3}
```

Each input weight `net.IW{i,j}`, layer weight `net.LW{i,j}`, and bias vector `net.b{i}` has as many rows as the size of the *i*th layer (`net.layers{i}.size`).

Each input weight `net.IW{i,j}` has as many columns as the size of the *j*th input (`net.inputs{j}.size`) multiplied by the number of its delay values (`length(net.inputWeights{i,j}.delays)`).

Likewise, each layer weight has as many columns as the size of the j th layer (`net.layers{j}.size`) multiplied by the number of its delay values (`length(net.layerWeights{i,j}.delays)`).

Network Behavior

Initialization

Initialize your network with the following line of code:

```
net = init(net);
```

Check the network's biases and weights again to see how they have changed:

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

For example,

```
net.IW{1,1}
ans =
-0.3040    0.4703
-0.5423   -0.1395
 0.5567    0.0604
 0.2667    0.4924
```

Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

```
X = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1] [-1; -1; 1; 0; 1]};
```

You want the network to respond with the following target sequences for the second layer, which has three neurons, and the third layer with one neuron:

```
T = {[1; 1; 1] [0; 0; 0]; 1 -1};
```

Before training, you can simulate the network to see whether the initial network's response Y is close to the target T.

```
Y = sim(net,X)
Y =
```

```
[3x1 double]    [3x1 double]
[      1.7148]    [      2.2726]
```

The cell array Y is the output sequence of the network, which is also the output sequence of the second and third layers. The values you got for the second row can differ from those shown because of different initial weights and biases. However, they will almost certainly not be equal to targets T, which is also true of the values shown.

The next task is optional. On some occasions you may wish to alter the training parameters before training. The following line of code displays the default Levenberg-Marquardt training parameters (defined when you set `net.trainFcn` to `trainlm`).

```
net.trainParam
```

The following properties should be displayed.

```
ans =
Show Training Window Feedback    showWindow: true
Show Command Line Feedback showCommandLine: false
Command Line Frequency           show: 25
Maximum Epochs                  epochs: 1000
Maximum Training Time          time: Inf
Performance Goal                goal: 0
Minimum Gradient                 min_grad: 1e-07
Maximum Validation Checks       max_fail: 6
Mu                               mu: 0.001
Mu Decrease Ratio              mu_dec: 0.1
Mu Increase Ratio              mu_inc: 10
Maximum mu                      mu_max: 100000000000
```

You will not often need to modify these values. See the documentation for the training function for information about what each of these means. They have been initialized with default values that work well for a large range of problems, so there is no need to change them here.

Next, train the network with the following call:

```
net = train(net,X,T);
```

Training launches the neural network training window. To open the performance and training state plots, click the plot buttons.

After training, you can simulate the network to see if it has learned to respond correctly:

```
Y = sim(net,X)

[3x1 double]    [3x1 double]
[      1.0000]    [      -1.0000]
```

The second network output (i.e., the second row of the cell array Y), which is also the third layer's output, matches the target sequence T.

Custom Neural Network Helper Functions

The toolbox allows you to create and use your own custom functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train your networks.

Be aware, however, that custom functions may need updating to remain compatible with future versions of the software. Backward compatibility of custom functions cannot be guaranteed.

Template functions are available for you to copy, rename and customize, to create your own versions of these kinds of functions. You can see the list of all template functions by typing the following:

```
help nncustom
```

Each template is a simple version of a different type of function that you can use with your own custom networks.

For instance, make a copy of the file `tansig.m` with the new name `mytransfer.m`. Start editing the new file by changing the function name at the top from `tansig` to `mytransfer`.

You can now edit each of the sections of code that make up a transfer function, using the help comments in each of those sections to guide you.

Once you are done, store the new function in your working folder, and assign the name of your transfer function to the `transferFcn` property of any layer of any network object to put it to use.

Automatically Save Checkpoints During Neural Network Training

During neural network training, intermediate results can be periodically saved to a MAT file for recovery if the computer fails or you kill the training process. This helps protect the value of long training runs, which if interrupted would need to be completely restarted otherwise. This feature is especially useful for long parallel training sessions, which are more likely to be interrupted by computing resource failures.

Checkpoint saves are enabled with the optional '`'CheckpointFile'`' training argument followed by the checkpoint file name or path. If you specify only a file name, the file is placed in the working directory by default. The file must have the `.mat` file extension, but if this is not specified it is automatically appended. In this example, checkpoint saves are made to the file called `MyCheckpoint.mat` in the current working directory.

```
[x,t] = bodyfat_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t,'CheckpointFile','MyCheckpoint.mat');
```

```
22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the previous short training example, this results in only two checkpoint saves: one at the beginning and one at the end of training.

The optional training argument '`'CheckpointDelay'`' can change the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,[],t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);

22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, you can reload the checkpoint structure containing the best neural network obtained before the interruption, and the training record. In this case, the `stage` field value is '`'Final'`', indicating the last save was at the

final epoch because training completed successfully. The first epoch checkpoint is indicated by 'First', and intermediate checkpoints by 'Write'.

```
load('MyCheckpoint.mat')

checkpoint =
    file: '/WorkdingDir/MyCheckpoint.mat'
    time: [2013 3 22 5 0 9.0712]
    number: 6
    stage: 'Final'
    net: [1x1 network]
    tr: [1x1 struct]
```

You can resume training from the last checkpoint by reloading the dataset (if necessary), then calling train with the recovered network.

```
net = checkpoint.net;
[x,t] = maglev_dataset;
load('MyCheckpoint.mat');
[X,Xi,Ai,T] = prepares(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);
```

Deploy Shallow Neural Network Functions

In this section...

[“Deployment Functions and Tools for Trained Networks” on page 11-63](#)

[“Generate Neural Network Functions for Application Deployment” on page 11-64](#)

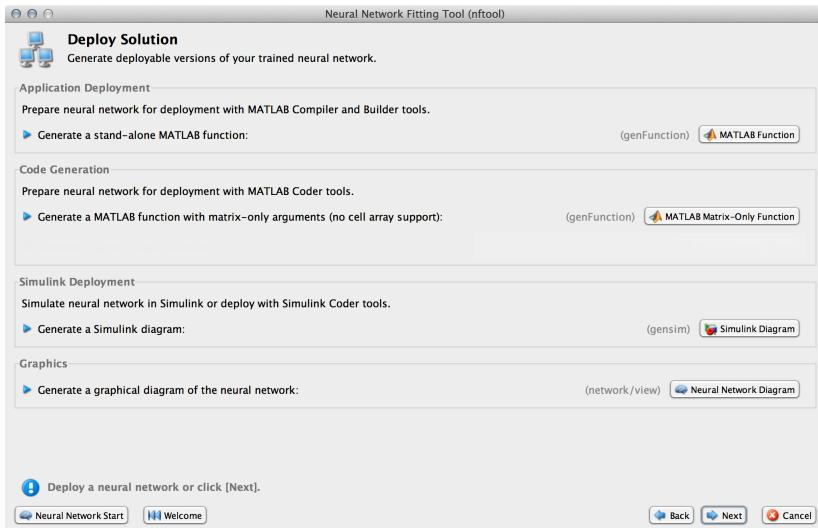
[“Generate Simulink Diagrams” on page 11-67](#)

Deployment Functions and Tools for Trained Networks

The function `genFunction` allows stand-alone MATLAB functions for a trained shallow neural network. The generated code contains all the information needed to simulate a neural network, including settings, weight and bias values, module functions, and calculations.

The generated MATLAB function can be used to inspect the exact simulation calculations that a particular shallow neural network performs, and makes it easier to deploy neural networks for many purposes with a wide variety of MATLAB deployment products and tools.

The function `genFunction` is introduced in the deployment panels in the tools `nftool`, `nctool`, `nprtool` and `ntstool`. For information on these tool features, see “Fit Data with a Shallow Neural Network”, “Classify Patterns with a Shallow Neural Network”, “Cluster Data with a Self-Organizing Map”, and “Shallow Neural Network Time-Series Prediction and Modeling”.



The advanced scripts generated on the Save Results panel of each of these tools includes an example of deploying networks with `genFunction`.

Generate Neural Network Functions for Application Deployment

The function `genFunction` generates a stand-alone MATLAB function for simulating any trained shallow neural network and preparing it for deployment. This might be useful for several tasks:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Use the MATLAB Function block to create a Simulink block
- Use MATLAB Compiler™ to:
 - Generate stand-alone executables
 - Generate Excel® add-ins
- Use MATLAB Compiler SDK™ to:
 - Generate C/C++ libraries

- Generate .COM components
- Generate Java® components
- Generate .NET components
- Use MATLAB Coder™ to:
 - Generate C/C++ code
 - Generate efficient MEX-functions

`genFunction(net, 'pathname')` takes a neural network and file path, and produces a standalone MATLAB function file `filename.m`.

`genFunction(..., 'MatrixOnly', 'yes')` overrides the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks, the matrix columns are interpreted as independent samples. For dynamic networks, the matrix columns are interpreted as a series of time steps. The default value is 'no'.

`genFunction(___, 'ShowLinks', 'no')` disables the default behavior of displaying links to generated help and source code. The default is 'yes'.

Here a static network is trained and its outputs calculated.

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

The following code generates, tests, and displays a MATLAB function with the same interface as the neural network object.

```
genFunction(bodyfatNet, 'bodyfatFcn');
y2 = bodyfatFcn(x);
accuracy2 = max(abs(y - y2))
edit bodyfatFcn
```

You can compile the new function with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libBodyfat -T link:lib bodyfatFcn
```

The next code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a

MEX-function with the MATLAB Coder tool `codegen` (license required), which is also tested.

```
genFunction(bodyfatNet, 'bodyfatFcn', 'MatrixOnly', 'yes');
y3 = bodyfatFcn(x);
accuracy3 = max(abs(y - y3))

x1Type = coder.typeof(double(0), [13, Inf]); % Coder type of input 1
codegen bodyfatFcn.m -config:mex -o bodyfatCodeGen -args {x1Type}
y4 = bodyfatCodeGen(x);
accuracy4 = max(abs(y - y4))
```

Here a dynamic network is trained and its outputs calculated.

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,{},t);
maglevNet = train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next a MATLAB function is generated and tested. The function is then used to create a shared/dynamically linked library with `mcc`.

```
genFunction(maglevNet,'maglevFcn');
[y2,xf,af] = maglevFcn(X,Xi,Ai);
accuracy2 = max(abs(cell2mat(y)-cell2mat(y2)))
mcc -W lib:libMaglev -T link:lib maglevFcn
```

The following code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen`, which is also tested.

```
genFunction(maglevNet,'maglevFcn','MatrixOnly','yes');
x1 = cell2mat(X(1,:)); % Convert each input to matrix
x2 = cell2mat(X(2,:));
xi1 = cell2mat(Xi(1,:)); % Convert each input state to matrix
xi2 = cell2mat(Xi(2,:));
[y3,xf1,xf2] = maglevFcn(x1,x2,xi1,xi2);
accuracy3 = max(abs(cell2mat(y)-y3))

x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen ...
    -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

Generate Simulink Diagrams

For information on simulating shallow neural networks and deploying trained neural networks with Simulink tools, see “Deploy Shallow Neural Network Simulink Diagrams” on page B-5.

See Also

More About

- “Deploy Training of Shallow Neural Networks” on page 11-68

Deploy Training of Shallow Neural Networks

Tip To learn about code generation for deep learning, see “Deep Learning Code Generation”.

Use MATLAB Runtime to deploy functions that can train a model. You can deploy MATLAB code that trains neural networks as described in “Create Standalone Application from Command Line” (MATLAB Compiler).

The following methods and functions are NOT supported in deployed mode:

- Training progress dialog, `nntool`.
- `genFunction` and `gensim` to generate MATLAB code or Simulink blocks
- `view` method
- `nctool`, `nftool`, `nnstart`, `nprtool`, `ntstool`
- Plot functions (such as `plotperform`, `plottrainstate`, `ploterrhist`, `plotregression`, `plotfit`, and so on)
- `perceptron`, `newlind`, and `elmannet` functions.

Here is an example of how you can deploy training of a network. Create a script to train a neural network, for example, `mynntraining.m`:

```
% Create the predictor and response (target)
x = [0.054 0.78 0.13 0.47 0.34 0.79 0.53 0.6 0.65 0.75 0.084 0.91 0.83
      0.53 0.93 0.57 0.012 0.16 0.31 0.17 0.26 0.69 0.45 0.23 0.15 0.54];
t = [0.46 0.079 0.42 0.48 0.95 0.63 0.48 0.51 0.16 0.51 1 0.28 0.3];
% Create and display the network
net = fitnet();
disp('Training fitnet')
% Train the network using the data in x and t
net = train(net,x,t);
% Predict the responses using the trained network
y = net(x);
% Measure the performance
perf = perform(net,y,t)
```

Compile the script `mynntraining.m` by using the command line:

```
mcc -m 'mynntraining.m'
```

`mcc` invokes the MATLAB Compiler to compile code at the prompt. The flag `-m` compiles a MATLAB function and generates a standalone executable. The EXE file is now in your local computer in the working directory.

To run the compiled EXE application on computers that do not have MATLAB installed, you need to download and install MATLAB Runtime. The `readme.txt` created in your working folder has more information about the deployment requirements.

See Also

More About

- “Deploy Shallow Neural Network Functions” on page 11-63

Historical Neural Networks

- “Historical Neural Networks Overview” on page 12-2
- “Perceptron Neural Networks” on page 12-3
- “Linear Neural Networks” on page 12-18

Historical Neural Networks Overview

This section covers networks that are of historical interest, but that are not as actively used today as networks presented in other sections. Two of the networks are single-layer networks that were the first neural networks for which practical training algorithms were developed: perceptron networks and ADALINE networks.

The perceptron network is a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. This perceptron rule was the first training algorithm developed for neural networks. The original book on the perceptron is Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C., Spartan Press, 1961 [Rose61 on page 14-2].

At about the same time that Rosenblatt developed the perceptron network, Widrow and Hoff developed a single-layer linear network and associated learning rule, which they called the ADALINE (Adaptive Linear Neuron). This network was used to implement adaptive filters, which are still actively used today. The original paper describing this network is Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96-104.

Perceptron Neural Networks

In this section...

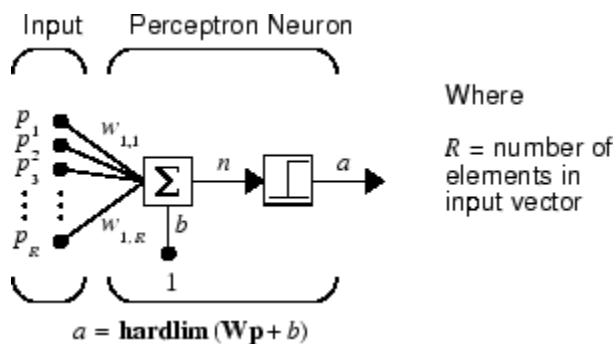
- “Neuron Model” on page 12-3
- “Perceptron Architecture” on page 12-5
- “Create a Perceptron” on page 12-6
- “Perceptron Learning Rule (learnp)” on page 12-8
- “Training (train)” on page 12-10
- “Limitations and Cautions” on page 12-15

Rosenblatt [Rose61 on page 14-2] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

The discussion of perceptrons in this section is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996 on page 14-2], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

Neuron Model

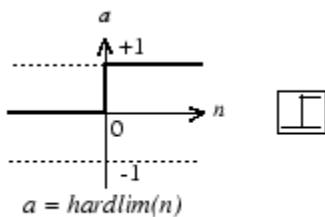
A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



Where

R = number of elements in input vector

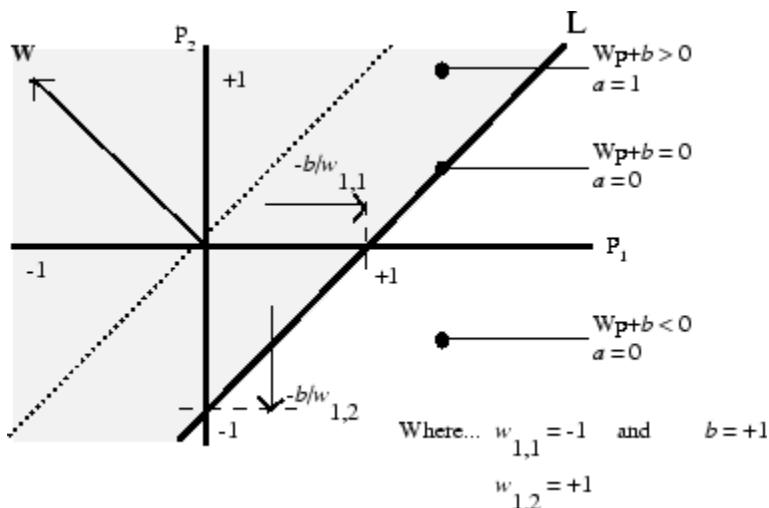
Each external input is weighted with an appropriate weight w_{1j} , and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The following figure show the input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$.



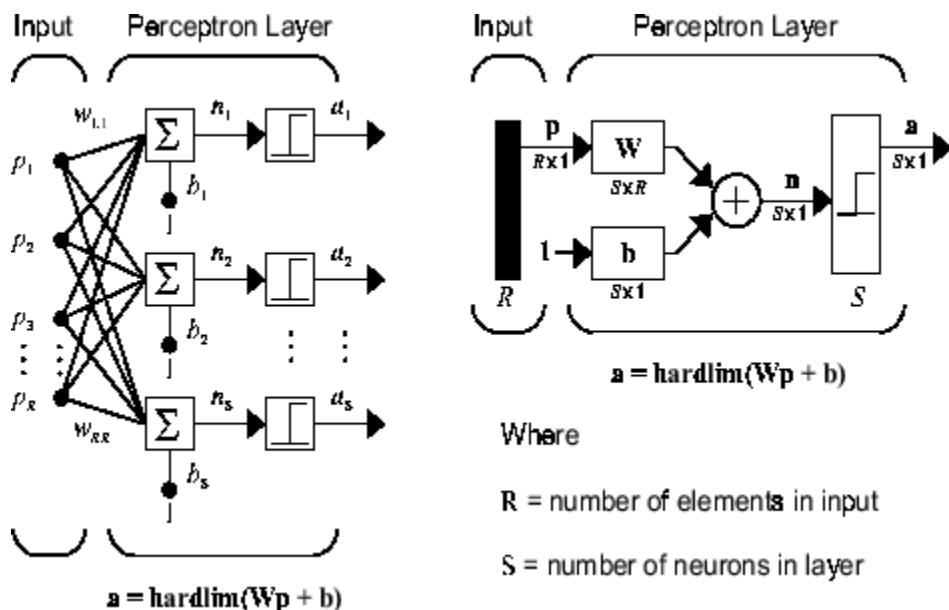
Two classification regions are formed by the *decision boundary* line L at $\mathbf{W}\mathbf{p} + b = 0$. This line is perpendicular to the weight matrix \mathbf{W} and shifted according to the bias b . Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin, as shown in the plot above.

You might want to run the example program nnd4db. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights w_{ij} , as shown below in two forms. As before, the network indices i and j indicate that w_{ij} is the strength of the connection from the j th input to the i th neuron.



The perceptron learning rule described shortly is capable of training only a single layer. Thus only one-layer networks are considered here. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed in "Limitations and Cautions" on page 12-15.

Create a Perceptron

You can create a perceptron with the following:

```
net = perceptron;
net = configure(net, P, T);
```

where input arguments are as follows:

- P is an R -by- Q matrix of Q input vectors of R elements each.
- T is an S -by- Q matrix of Q target vectors of S elements each.

Commonly, the `hardlim` function is used in perceptrons, so it is the default.

The following commands create a perceptron network with a single one-element input vector with the values 0 and 2, and one neuron with outputs that can be either 0 or 1:

```
P = [0 2];
T = [0 1];
net = perceptron;
net = configure(net,P,T);
```

You can see what network has been created by executing the following command:

```
inputweights = net.inputweights{1,1}
```

which yields

```
inputweights =
    delays: 0
    initFcn: 'initzero'
    learn: true
    learnFcn: 'learnp'
    learnParam: (none)
    size: [1 1]
    weightFcn: 'dotprod'
    weightParam: (none)
    userdata: (your custom info)
```

The default learning function is `learnp`, which is discussed in “Perceptron Learning Rule (`learnp`)” on page 12-8. The net input to the `hardlim` transfer function is `dotprod`, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

The default initialization function `initzero` is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
    initFcn: 'initzero'
    learn: 1
    learnFcn: 'learnp'
    learnParam: []
    size: 1
    userdata: [1x1 struct]
```

You can see that the default initialization for the bias is also 0.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1 \mathbf{t}_1, \mathbf{p}_2 \mathbf{t}_2, \dots, \mathbf{p}_Q \mathbf{t}_Q$$

where \mathbf{p} is an input to the network and \mathbf{t} is the corresponding correct (target) output. The objective is to reduce the error \mathbf{e} , which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response \mathbf{a} and the target vector \mathbf{t} . The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector \mathbf{p} and the associated error \mathbf{e} . The target vector \mathbf{t} must contain values of either 0 or 1, because perceptrons (with `hardlim` transfer functions) can only output these values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector \mathbf{w} to point toward input vectors to be classified as 1 and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to \mathbf{w} and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector \mathbf{p} is presented and the network's response \mathbf{a} is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct ($\mathbf{a} = \mathbf{t}$ and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$), then the weight vector \mathbf{w} is not altered.

CASE 2. If the neuron output is 0 and should have been 1 ($\mathbf{a} = 0$ and $\mathbf{t} = 1$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$), the input vector \mathbf{p} is added to the weight vector \mathbf{w} . This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 ($\mathbf{a} = 1$ and $\mathbf{t} = 0$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = -1$), the input vector \mathbf{p} is subtracted from the weight vector \mathbf{w} . This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$ and the change to be made to the weight vector $\Delta\mathbf{w}$:

CASE 1. If $\mathbf{e} = 0$, then make a change $\Delta\mathbf{w}$ equal to 0.

CASE 2. If $\mathbf{e} = 1$, then make a change $\Delta\mathbf{w}$ equal to \mathbf{p}^T .

CASE 3. If $\mathbf{e} = -1$, then make a change $\Delta\mathbf{w}$ equal to $-\mathbf{p}^T$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t - \alpha)\mathbf{p}^T = \mathbf{e}\mathbf{p}^T$$

You can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - \alpha)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^T = \mathbf{e}(\mathbf{p})^T$$

and

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = \mathbf{e}$$

The perceptron learning rule can be summarized as follows:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

and

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$.

Now try a simple example. Start with a single neuron having an input vector with just two elements.

```
net = perceptron;
net = configure(net,[0;0],0);
```

To simplify matters, set the bias equal to 0 and the weights to 1 and -0.8:

```
net.b{1} = [0];
w = [1 -0.8];
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];
t = [1];
```

You can compute the output and error with

```
a = net(p)
a =
    0
e = t-a
e =
    1
```

and use the function `learnp` to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],[],[],[],[],[])
dw =
    1      2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Recall that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are linearly separable. The objects to be classified in such cases can be separated by a single line.

You might want to try the example `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

Training (train)

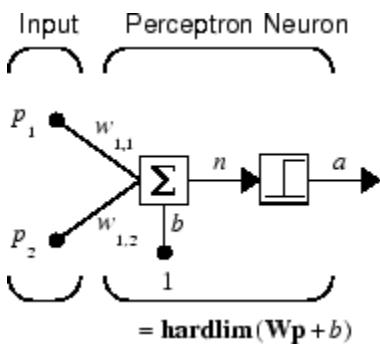
If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually

find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traversal through all the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. You must check the new values of \mathbf{W} and \mathbf{b} by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully you can train it further by calling `train` again with the new weights and biases for more training passes, or you can analyze the problem to see if it is a suitable problem for the perceptron. Problems that cannot be solved by the perceptron network are discussed in “Limitations and Cautions” on page 12-15.

To illustrate the training procedure, work through a simple problem. Consider a one-neuron perceptron with a single vector input having two elements:



This network, and the problem you are about to consider, are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996 on page 14-2].

Suppose you have the following classification problem and would like to solve it with a single vector input, two-element perceptron network.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Use the initial weights and bias. Denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, the initial values are $\mathbf{W}(0)$ and $b(0)$.

$$\mathbf{W}(0) = [0 \ 0] \quad b(0) = 0$$

Start by calculating the perceptron's output a for the first input vector \mathbf{p}_1 , using the initial weights and bias.

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(0)\mathbf{p}_1 + b(0)) \\ &= \text{hardlim}\left([0 \ 0]\begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0\right) = \text{hardlim}(0) = 1\end{aligned}$$

The output a does not equal the target value t_1 , so use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$\begin{aligned}e &= t_1 - \alpha = 0 - 1 = -1 \\ \Delta\mathbf{W} &= e\mathbf{p}_1^T = (-1)[2 \ 2] = [-2 \ -2] \\ \Delta b &= e = -1\end{aligned}$$

You can calculate the new weights and bias using the perceptron update rules.

$$\begin{aligned}\mathbf{W}^{new} &= \mathbf{W}^{old} + e\mathbf{p}_1^T = [0 \ 0] + [-2 \ -2] = [-2 \ -2] = \mathbf{W}(1) \\ b^{new} &= b^{old} + e = 0 + (-1) = -1 = b(1)\end{aligned}$$

Now present the next input vector, \mathbf{p}_2 . The output is calculated below.

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(1)\mathbf{p}_2 + b(1)) \\ &= \text{hardlim}\left([-2 \ -2]\begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1\right) = \text{hardlim}(1) = 1\end{aligned}$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = [-2 \ -2]$ and $b(2) = b(1) = -1$.

You can continue in this fashion, presenting \mathbf{p}_3 next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values $\mathbf{W}(4) = [-3 \ -1]$ and $b(4) = 0$. To determine whether a satisfactory solution is obtained, make one pass through all input vectors to see if they all produce the desired target values. This is not true for the fourth input, but the algorithm does converge on the sixth presentation of an input. The final values are

$$\mathbf{W}(6) = [-2 \quad -3] \quad \text{and} \quad b(6) = 1.$$

This concludes the hand calculation. Now, how can you do this using the `train` function?

The following code defines a perceptron.

```
net = perceptron;
```

Consider the application of a single input

```
p = [2; 2];
```

having the target

```
t = [0];
```

Set `epochs` to 1, so that `train` goes through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2      -2
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values $[-2 \quad -2]$ and -1 , just as you hand calculated.

Now apply the second input vector \mathbf{p}_2 . The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0, and the change will be zero. You could proceed in this way, starting from the previous result and applying a new input vector time after time. But you can do this job automatically with `train`.

Apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = perceptron;
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]  
t = [0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}  
w =  
    -3      -1  
b =  
    0
```

This is the same result as you got previously by hand.

Finally, simulate the trained network for each of the inputs.

```
a = net(p)  
a =  
    0      0      1      1
```

The outputs do not yet equal the targets, so you need to train the network for more than one pass. Try more epochs. This run gives a mean absolute error performance of 0 after two epochs:

```
net.trainParam.epochs = 1000;  
net = train(net,p,t);
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As you know from hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w = net.iw{1,1}, b = net.b{1}  
w =  
    -2      -3  
b =  
    1
```

The simulated output and errors for the various inputs are

```
a = net(p)  
a =
```

```

          0           1           0           1
error = a-t
error =
          0           0           0           0

```

You confirm that the training procedure is successful. The network converges and produces the correct target outputs for the four input vectors.

The default training function for networks created with `perceptron` is `traininc`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You might want to try various example programs. For instance, `demop1` illustrates classification and training of a simple perceptron.

Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of `adapt` in this way guarantees that any linearly separable problem is solved in a finite number of training presentations.

As noted in the previous pages, perceptrons can also be trained with the function `train`. Commonly when `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) because of the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. However, it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a

solution in finite time. You might want to try `demop6`. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two-neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996 on page 14-2].

Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try `demop4` to see how an outlier affects the training.

By changing the perceptron learning rule slightly, you can make training times insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t - \alpha) \mathbf{p}^T = e \mathbf{p}^T$$

As shown above, the larger an input vector \mathbf{p} , the larger its effect on the weight vector \mathbf{w} . Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that the effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t - \alpha) \frac{\mathbf{p}^T}{\|\mathbf{p}\|} = e \frac{\mathbf{p}^T}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnp`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces the number of epochs considerably if there are outlier input vectors. You might try `demop5` to see how this normalized training rule works.

Linear Neural Networks

In this section...

- “Neuron Model” on page 12-18
- “Network Architecture” on page 12-19
- “Least Mean Square Error” on page 12-22
- “Linear System Design (newlind)” on page 12-23
- “Linear Networks with Delays” on page 12-24
- “LMS Algorithm (learnwh)” on page 12-26
- “Linear Classification (train)” on page 12-28
- “Limitations and Cautions” on page 12-30

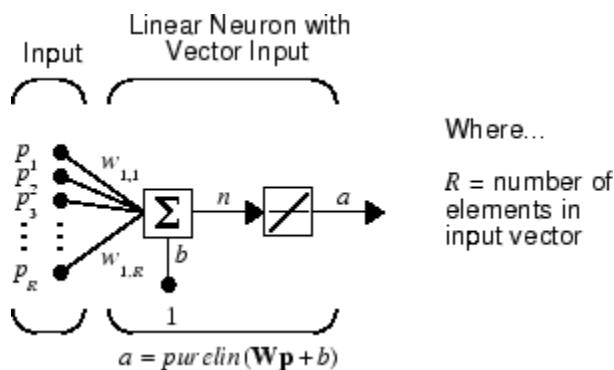
The linear networks discussed in this section are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here you design a linear network that, when presented with a set of given input vectors, produces outputs of corresponding target vectors. For each input vector, you can calculate the network's output vector. The difference between an output vector and its target vector is the error. You would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, you can calculate a linear network directly, such that its error is a minimum for the given input vectors and target vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, you can always train the network to have a minimum error by using the least mean squares (Widrow-Hoff) algorithm.

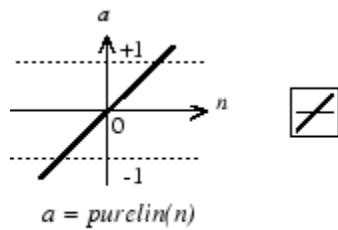
This section introduces `linearlayer`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function `purelin`.



Linear Transfer Function

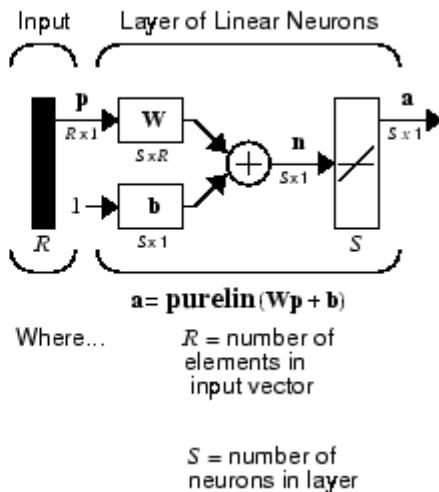
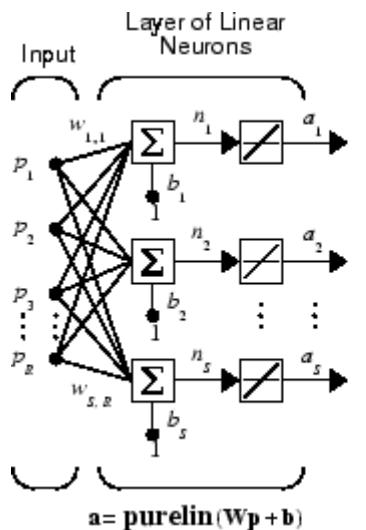
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Network Architecture

The linear network shown below has one layer of S neurons connected to R inputs through a matrix of weights \mathbf{W} .

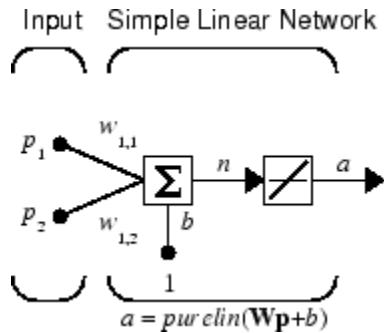


Note that the figure on the right defines an S -length output vector \mathbf{a} .

A single-layer linear network is shown. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Create a Linear Neuron (linearlayer)

Consider a single linear neuron with two inputs. The following figure shows the diagram for this network.



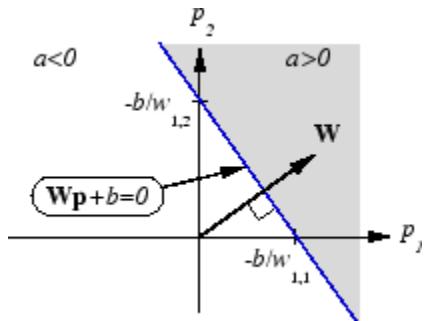
The weight matrix \mathbf{W} in this case has only one row. The network output is

$$\alpha = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

or

$$\alpha = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{W}\mathbf{p} + b = 0$ specifies such a decision boundary, as shown below (adapted with thanks from [HDB96 on page 14-2]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear network has the same limitation as the perceptron.

You can create this network using `linearlayer`, and configure its dimensions with two values so the input has two elements and the output has one.

```
net = linearlayer;
net = configure(net,[0;0],0);
```

The network weights and biases are set to zero by default. You can see the current values with the commands

```
W = net.IW{1,1}
W =
    0      0
```

and

```
b= net.b{1}
b =
    0
```

However, you can give the weights any values that you want, such as 2 and 3, respectively, with

```
net.IW{1,1} = [2 3];
W = net.IW{1,1}
W =
    2      3
```

You can set and check the bias in the same way.

```
net.b{1} = [-4];
b = net.b{1}
b =
    -4
```

You can simulate the linear network for a particular input vector. Try

```
p = [5;6];
```

You can find the network output with the function `sim`.

```
a = net(p)
a =
    24
```

To summarize, you can create a linear network with `linearlayer`, adjust its elements as you want, and simulate it with `sim`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here \mathbf{p}_q is an input to the network, and \mathbf{t}_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q} \sum_{k=1}^Q e(k)^2 = \frac{1}{Q} \sum_{k=1}^Q (t(k) - \alpha(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96 on page 14-2].

Linear System Design (newlind)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. You can obtain specific network values for weights and biases to minimize the mean square error by using the function **newlind**.

Suppose that the inputs and targets are

```
P = [1 2 3];
T= [2.0 4.1 5.9];
```

Now you can design a network.

```
net = newlind(P,T);
```

You can simulate the network behavior to check that the design was done properly.

```
Y = net(P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

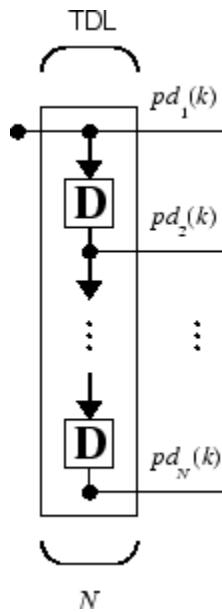
You might try **demolin1**. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

You can also use the function **newlind** to design linear networks having delays in the input. Such networks are discussed in "Linear Networks with Delays" on page 12-24. First, however, delays must be discussed.

Linear Networks with Delays

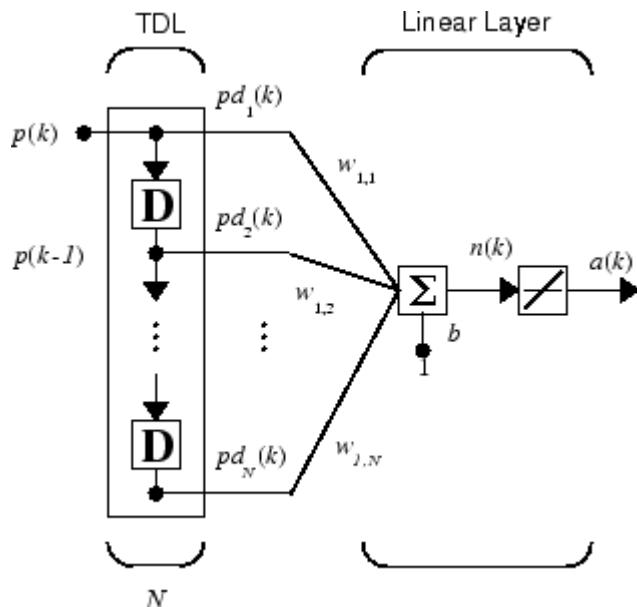
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left and passes through $N-1$ delays. The output of the tapped delay line (TDL) is an N -dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Linear Filter

You can combine a tapped delay line with a linear network to create the linear *filter* shown.



The output of the filter is given by

$$a(k) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^R w_{1,i}p(k-i+1) + b$$

The network shown is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85 on page 14-2]. Look at the code used to generate and simulate such a network.

Suppose that you want a linear layer that outputs the sequence T, given the sequence P and two initial input delay states Pi.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5 6 4 20 7 8};
```

You can use `newlind` to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument, as shown below.

```
net = newlind(P,T,Pi);
```

You can obtain the output of the designed network with

`Y = net(P,Pi)`

to give

`Y = [2.7297] [10.5405] [5.0090] [14.9550] [10.7838] [5.9820]`

As you can see, the network outputs are not exactly equal to the targets, but they are close and the mean square error is minimized.

LMS Algorithm (`learnwh`)

The LMS algorithm, or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If you take the partial derivative of the squared error with respect to the weights and biases at the k th iteration, you have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k) \frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1, 2, \dots, R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k) \frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - \alpha(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} [t(k) - (\mathbf{W}\mathbf{p}(k) + b)]$$

or

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} \left[t(k) - \left(\sum_{i=1}^R w_{1,i} p_i(k) + b \right) \right]$$

Here $p_i(k)$ is the i th element of the input vector at the k th iteration.

This can be simplified to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

and

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, change the weight matrix, and the bias will be

$$2\alpha e(k)\mathbf{p}(k)$$

and

$$2\alpha e(k)$$

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors, and α is a *learning rate*. If α is large, learning occurs quickly, but if it is too large it can lead to instability and errors might even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix $\mathbf{p}^T \mathbf{p}$ of the input vectors.

You might want to read some of Chapter 10 of [HDB96 on page 14-2] for more information about the LMS algorithm and its convergence.

Fortunately, there is a toolbox function, `learnwh`, that does all the calculation for you. It calculates the change in weights as

$$dw = lr * e * p'$$

and the bias change as

$$db = lr * e$$

The constant 2, shown a few lines above, has been absorbed into the code learning rate `lr`. The function `maxlinlr` calculates this maximum stable learning rate `lr` as `0.999 * P'*P`.

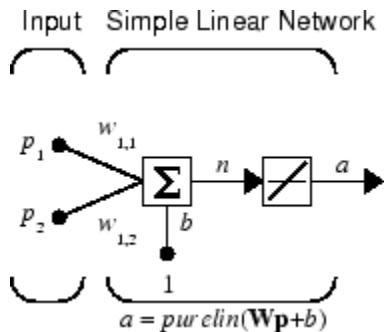
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

Linear Classification (train)

Linear networks can be trained to perform linear classification with the function `train`. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to `learnp`. Then the network is adjusted with the sum of all these corrections. Each pass through the input vectors is called an *epoch*. This contrasts with `adapt` which adjusts weights for each input vector as it is presented.

Finally, `train` applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped, and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

A simple problem illustrates this procedure. Consider the linear network introduced earlier.



Suppose you have the following classification problem.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Here there are four input vectors, and you want a network that produces the output corresponding to each input vector when that vector is presented.

Use `train` to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network are 0 by default. Set the error goal to 0.1 rather than accept its default of 0.

```
P = [2 1 -2 -1;2 -2 2 1];
T = [0 1 0 1];
net = linearlayer;
net.trainParam.goal= 0.1;
net = train(net,P,T);
```

The problem runs for 64 epochs, achieving a mean square error of 0.0999. The new weights and bias are

```
weights = net.iw{1,1}
weights =
-0.0615    -0.2194
bias = net.b(1)
bias =
[0.5899]
```

You can simulate the new network as shown below.

```
A = net(P)
A =
0.0282    0.9672    0.2741    0.4320
```

You can also calculate the error.

```
err = T - sim(net,P)
err =
-0.0282    0.0328   -0.2741    0.5680
```

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had a smaller error goal been chosen, but in this problem it is not possible to obtain a goal of 0. The network is limited in its capability. See “Limitations and Cautions” on page 12-30 for examples of various limitations.

This example program, `demolin2`, shows the training of a linear neuron and plots the weight trajectory and error during training.

You might also try running the example program `nnd10lc`. It addresses a classic and historically interesting problem, shows how a network can be trained to classify various

patterns, and shows how the trained network responds when noisy patterns are presented.

Limitations and Cautions

Linear networks can only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate lr is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Because parabolas have only one minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have various other limitations. Some of them are discussed below.

Overdetermined Systems

Consider an overdetermined system. Suppose that you have a network to be trained with four one-element input vectors and four targets. A perfect solution to $wp + b = t$ for each of the inputs might not exist, for there are four constraining equations, and only one weight and one bias to adjust. However, the LMS rule still minimizes the error. You might try `demolin4` to see how this is done.

Underdetermined Systems

Consider a single linear neuron with one input. This time, in `demolin5`, train it on only one one-element input vector and its one-element target vector:

```
P = [1.0];  
T = [0.5];
```

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can try `demolin5` to explore this topic.

Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ($S * R + S$ = number of weights and biases) as constraints (Q = pairs of input/target

vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with the example `demolin6`, the network cannot solve the problem with zero error. You might want to try `demolin6`.

Too Large a Learning Rate

You can always train a linear network with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Example `demolin7` shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by `maxlinlr`. The network is trained with two different learning rates to show the results of using too large a learning rate.

Neural Network Object Reference

- “Neural Network Object Properties” on page 13-2
- “Neural Network Subobject Properties” on page 13-14

Neural Network Object Properties

In this section...

- “General” on page 13-2
- “Architecture” on page 13-2
- “Subobject Structures” on page 13-6
- “Functions” on page 13-8
- “Weight and Bias Values” on page 13-12

These properties define the basic features of a network. “Neural Network Subobject Properties” on page 13-14 describes properties that define network details.

General

Here are the general properties of neural networks.

net.name

This property consists of a string defining the network name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net userdata

This property provides a place for users to add custom information to a network object. Only one field is predefined. It contains a *secret* message to all Deep Learning Toolbox users:

```
net userdata.note
```

Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

net.numInputs

This property defines the number of inputs a network receives. It can be set to 0 or a positive integer.

Clarification

The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e., the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

Side Effects

Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

`net.numLayers`

This property defines the number of layers a network has. It can be set to 0 or a positive integer.

Side Effects

Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers:

```
net.biasConnect  
net.inputConnect  
net.layerConnect  
net.outputConnect
```

and changes the size of each cell array of subobject structures whose size depends on the number of layers:

```
net.biases  
net.inputWeights  
net.layerWeights  
net.outputs
```

and also changes the size of each of the network's adjustable parameter's properties:

```
net.IW  
net.LW  
net.b
```

net.biasConnect

This property defines which layers have biases. It can be set to any N_l -by-1 matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the i th layer is indicated by a 1 (or 0) at

```
net.biasConnect(i)
```

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

net.inputConnect

This property defines which layers have weights coming from inputs.

It can be set to any $N_l \times N_i$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the i th layer from the j th input is indicated by a 1 (or 0) at `net.inputConnect(i,j)`.

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

net.layerConnect

This property defines which layers have weights coming from other layers. It can be set to any $N_l \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a weight going to the i th layer from the j th layer is indicated by a 1 (or 0) at

```
net.layerConnect(i,j)
```

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of layer weight subobjects (`net.layerWeights`) and the presence or absence of matrices in the cell array of layer weight matrices (`net.LW`).

net.outputConnect

This property defines which layers generate network outputs. It can be set to any $1 \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a network output from the i th layer is indicated by a 1 (or 0) at `net.outputConnect(i)`.

Side Effects

Any change to this property alters the number of network outputs (`net.numOutputs`) and the presence or absence of structures in the cell array of output subobjects (`net.outputs`).

net.numOutputs (read only)

This property indicates how many outputs the network has. It is always equal to the number of 1s in `net.outputConnect`.

net.numInputDelays (read only)

This property indicates the number of time steps of past inputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's input weights:

```
numInputDelays = 0;
for i=1:net.numLayers
    for j=1:net.numInputs
        if net.inputConnect(i,j)
            numInputDelays = max( ...
                [numInputDelays net.inputWeights{i,j}.delays]);
        end
    end
end
```

net.numLayerDelays (read only)

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's layer weights:

```
numLayerDelays = 0;
for i=1:net.numLayers
    for j=1:net.numLayers
        if net.layerConnect(i,j)
```

```
    numLayerDelays = max( ...
        [numLayerDelays net.layerWeights{i,j}.delays]);
    end
end
end
```

net.numWeightElements (read only)

This property indicates the number of weight and bias values in the network. It is the sum of the number of elements in the matrices stored in the two cell arrays:

```
net.IW  
new.b
```

Subobject Structures

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

The properties for each kind of subobject are described in "Neural Network Subobject Properties" on page 13-14.

net.inputs

This property holds structures of properties for each of the network's inputs. It is always an $N_i \times 1$ cell array of input structures, where N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the i th network input is located at

```
net.inputs{i}
```

If a neural network has only one input, then you can access `net.inputs{1}` without the cell array notation as follows:

```
net.input
```

Input Properties

See "Inputs" on page 13-14 for descriptions of input properties.

net.layers

This property holds structures of properties for each of the network's layers. It is always an $N_l \times 1$ cell array of layer structures, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the i th layer is located at `net.layers{i}`.

Layer Properties

See “Layers” on page 13-16 for descriptions of layer properties.

net.outputs

This property holds structures of properties for each of the network's outputs. It is always a $1 \times N_l$ cell array, where N_l is the number of network outputs (`net.numOutputs`).

The structure defining the properties of the output from the i th layer (or a null matrix []) is located at `net.outputs{i}` if `net.outputConnect(i)` is 1 (or 0).

If a neural network has only one output at layer i , then you can access `net.outputs{i}` without the cell array notation as follows:

```
net.output
```

Output Properties

See “Outputs” on page 13-22 for descriptions of output properties.

net.biases

This property holds structures of properties for each of the network's biases. It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the i th layer (or a null matrix []) is located at `net.biases{i}` if `net.biasConnect(i)` is 1 (or 0).

Bias Properties

See “Biases” on page 13-24 for descriptions of bias properties.

net.inputWeights

This property holds structures of properties for each of the network's input weights. It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the i th layer from the j th input (or a null matrix []) is located at `net.inputWeights{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

Input Weight Properties

See "Input Weights" on page 13-25 for descriptions of input weight properties.

net.layerWeights

This property holds structures of properties for each of the network's layer weights. It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the weight going to the i th layer from the j th layer (or a null matrix []) is located at `net.layerWeights{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

Layer Weight Properties

See "Layer Weights" on page 13-27 for descriptions of layer weight properties.

Functions

These properties define the algorithms to use when a network is to adapt, is to be initialized, is to have its performance measured, or is to be trained.

net.adaptFcn

This property defines the function to be used when the network adapts. It can be set to the name of any network adapt function. The network adapt function is used to perform adaption whenever `adapt` is called.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

Side Effects

Whenever this property is altered, the network's adaption parameters (`net.adaptParam`) are set to contain the parameters and default values of the new function.

net.adaptParam

This property defines the parameters and values of the current adapt function. Call `help` on the current adapt function to get a description of what each field means:

```
help(net.adaptFcn)
```

net.derivFcn

This property defines the derivative function to be used to calculate error gradients and Jacobians when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of any derivative function.

For a list of functions, type `help nnDerivative`.

net.divideFcn

This property defines the data division function to be used when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of a division function.

For a list of functions, type `help nnDivision`.

Side Effects

Whenever this property is altered, the network's adaption parameters (`net.divideParam`) are set to contain the parameters and default values of the new function.

net.divideParam

This property defines the parameters and values of the current data-division function. To get a description of what each field means, type the following command:

```
help(net.divideFcn)
```

net.divideMode

This property defines the target data dimensions which to divide up when the data division function is called. Its default value is 'sample' for static networks and 'time' for dynamic networks. It may also be set to 'sampletime' to divide targets by both sample and timestep, 'all' to divide up targets by every scalar value, or 'none' to not divide up data at all (in which case all data is used for training, none for validation or testing).

net.initFcn

This property defines the function used to initialize the network's weight matrices and bias vectors. . The initialization function is used to initialize the network whenever `init` is called:

```
net = init(net)
```

Side Effects

Whenever this property is altered, the network's initialization parameters (`net.initParam`) are set to contain the parameters and default values of the new function.

net.initParam

This property defines the parameters and values of the current initialization function. Call `help` on the current initialization function to get a description of what each field means:

```
help(net.initFcn)
```

net.performFcn

This property defines the function used to measure the network's performance. The performance function is used to calculate network performance during training whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nnperformance`.

Side Effects

Whenever this property is altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

net.performParam

This property defines the parameters and values of the current performance function. Call `help` on the current performance function to get a description of what each field means:

```
help(net.performFcn)
```

net.plotFcns

This property consists of a row cell array of strings, defining the plot functions associated with a network. The neural network training window, which is opened by the `train` function, shows a button for each plotting function. Click the button during or after training to open the desired plot.

net.plotParams

This property consists of a row cell array of structures, defining the parameters and values of each plot function in `net.plotFcns`. Call `help` on the each plot function to get a description of what each field means:

```
help(net.plotFcns{i})
```

net.trainFcn

This property defines the function used to train the network. It can be set to the name of any of the training functions, which is used to train the network whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

Side Effects

Whenever this property is altered, the network's training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

net.trainParam

This property defines the parameters and values of the current training function. Call `help` on the current training function to get a description of what each field means:

```
help(net.trainFcn)
```

Weight and Bias Values

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

net.IW

This property defines the weight matrices of weights going to layers from network inputs. It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The weight matrix for the weight going to the i th layer from the j th input (or a null matrix `[]`) is located at `net.IW{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the input size with the number of delays associated with the weight:

```
net.inputs{j}.size * length(net.inputWeights{i,j}.delays)
```

These dimensions can also be obtained from the input weight properties:

```
net.inputWeights{i,j}.size
```

net.LW

This property defines the weight matrices of weights going to layers from other layers. It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The weight matrix for the weight going to the i th layer from the j th layer (or a null matrix `[]`) is located at `net.LW{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight:

```
net.layers{j}.size * length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties:

```
net.layerWeights{i,j}.size
```

net.b

This property defines the bias vectors for each layer with a bias. It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The bias vector for the i th layer (or a null matrix []) is located at `net.b{i}` if `net.biasConnect(i)` is 1 (or 0).

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties:

```
net.biases{i}.size
```

Neural Network Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

In this section...

- "Inputs" on page 13-14
- "Layers" on page 13-16
- "Outputs" on page 13-22
- "Biases" on page 13-24
- "Input Weights" on page 13-25
- "Layer Weights" on page 13-27

Inputs

These properties define the details of each *i*th network input.

net.inputs{i}.name

This property consists of a string defining the input name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.inputs{i}.feedbackInput (read only)

If this network is associated with an open-loop feedback output, then this property will indicate the index of that output. Otherwise it will be an empty matrix.

net.inputs{i}.processFcns

This property defines a row cell array of processing function names to be used by *i*th network input. The processing functions are applied to input values before the network uses them.

Side Effects

Whenever this property is altered, the input `processParams` are set to default values for the given processing functions, `processSettings`, `processedSize`, and

`processedRange` are defined by applying the process functions and parameters to `exampleInput`.

For a list of processing functions, type `help nnprocess`.

net.inputs{i}.processParams

This property holds a row cell array of processing function parameters to be used by *i*th network input. The processing parameters are applied by the processing functions to input values before the network uses them.

Side Effects

Whenever this property is altered, the input `processSettings`, `processedSize`, and `processedRange` are defined by applying the process functions and parameters to `exampleInput`.

net.inputs{i}.processSettings (read only)

This property holds a row cell array of processing function settings to be used by *i*th network input. The processing settings are found by applying the processing functions and parameters to `exampleInput` and then used to provide consistent results to new input values before the network uses them.

net.inputs{i}.processedRange (read only)

This property defines the range of `exampleInput` values after they have been processed with `processingFcns` and `processingParams`.

net.inputs{i}.processedSize (read only)

This property defines the number of rows in the `exampleInput` values after they have been processed with `processingFcns` and `processingParams`.

net.inputs{i}.range

This property defines the range of each element of the *i*th network input.

It can be set to any $R_i \times 2$ matrix, where R_i is the number of elements in the input (`net.inputs{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum values of the *j*th input element, in that order:

```
net.inputs{i}(j,:)
```

Uses

Some initialization functions use input ranges to find appropriate initial values for input weight matrices.

Side Effects

Whenever the number of rows in this property is altered, the input `size`, `processedSize`, and `processedRange` change to remain consistent. The sizes of any weights coming from this input and the dimensions of the weight matrices also change.

net.inputs{i}.size

This property defines the number of elements in the *i*th network input. It can be set to 0 or a positive integer.

Side Effects

Whenever this property is altered, the input `range`, `processedRange`, and `processedSize` are updated. Any associated input weights change size accordingly.

net.inputs{i}.userdata

This property provides a place for users to add custom information to the *i*th network input.

Layers

These properties define the details of each *i*th network layer.

net.layers{i}.name

This property consists of a string defining the layer name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.layers{i}.dimensions

This property defines the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements becomes the number of neurons in the layer (`net.layers{i}.size`).

Uses

Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

Side Effects

Whenever this property is altered, the layer's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

net.layers{i}.distanceFcn

This property defines which of the distance functions is used to calculate **distances** between neurons in the *i*th layer from the neuron **positions**. Neuron distances are used by self-organizing maps. It can be set to the name of any distance function.

For a list of functions, type `help nnndistance`.

Side Effects

Whenever this property is altered, the distances between the layer's neurons (`net.layers{i}.distances`) are updated.

net.layers{i}.distances (read only)

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps:

`net.layers{i}.distances`

It is always set to the result of applying the layer's distance function (`net.layers{i}.distanceFcn`) to the positions of the layer's neurons (`net.layers{i}.positions`).

net.layers{i}.initFcn

This property defines which of the layer initialization functions are used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is `initlay`. If the network

initialization is set to `initlay`, then the function indicated by this property is used to initialize the layer's weights and biases.

net.layers{i}.netInputFcn

This property defines which of the net input functions is used to calculate the *i*th layer's net input, given the layer's weighted inputs and bias during simulating and training.

For a list of functions, type `help nnnetinput`.

net.layers{i}.netInputParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field:

```
help(net.layers{i}.netInputFcn)
```

net.layers{i}.positions (read only)

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

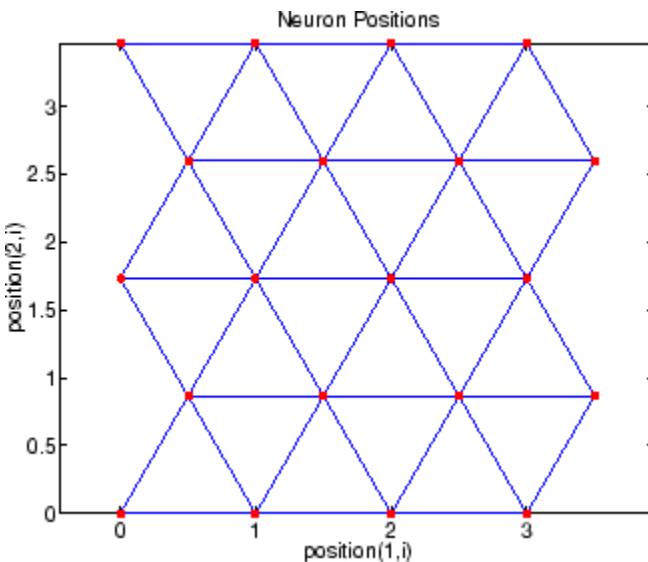
It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

Plotting

Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5], and the topology function (`net.layers{1}.topologyFcn`) is `hextop`, the neurons' positions can be plotted as follows:

```
plotsom(net.layers{1}.positions)
```



net.layers{i}.range (read only)

This property defines the output range of each neuron of the i th layer.

It is set to an $S_i \times 2$ matrix, where S_i is the number of neurons in the layer (`net.layers{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each j th row defines the minimum and maximum output values of the layer's transfer function `net.layers{i}.transferFcn`.

net.layers{i}.size

This property defines the number of neurons in the i th layer. It can be set to 0 or a positive integer.

Side Effects

Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.layerWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`), change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`), and biases (`net.b{i}`) also change.

Changing this property also changes the size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the `dimensions` property directly instead of using `size`.)

net.layers{i}.topologyFcn

This property defines which of the topology functions are used to calculate the *i*th layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

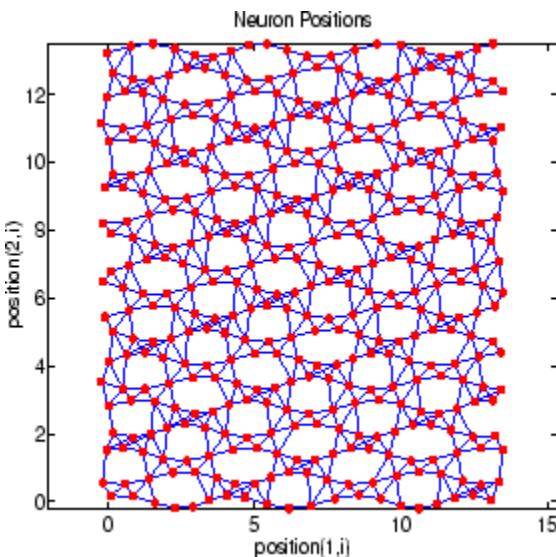
For a list of functions, type `help nntopology`.

Side Effects

Whenever this property is altered, the positions of the layer's neurons (`net.layers{i}.positions`) are updated.

Use `plotsom` to plot the positions of the layer neurons. For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is `randtop`, the neuron positions are arranged to resemble the following plot:

```
plotsom(net.layers{1}.positions)
```

**net.layers{i}.transferFcn**

This function defines which of the transfer functions is used to calculate the *i*th layer's output, given the layer's net input, during simulation and training.

For a list of functions, type `help nntransfer`.

net.layers{i}.transferParam

This property defines the parameters of the layer's transfer function. Call `help` on the current transfer function to get a description of what each field means:

```
help(net.layers{i}.transferFcn)
```

net.layers{i}.userdata

This property provides a place for users to add custom information to the *i*th network layer.

Outputs

net.outputs{i}.name

This property consists of a string defining the output name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.outputs{i}.feedbackInput

If the output implements open-loop feedback (`net.outputs{i}.feedbackMode = 'open'`), then this property indicates the index of the associated feedback input, otherwise it will be an empty matrix.

net.outputs{i}.feedbackDelay

This property defines the timestep difference between this output and network inputs. Input-to-output network delays can be removed and added with `removedelay` and `adddelay` functions resulting in this property being incremented or decremented respectively. The difference in timing between inputs and outputs is used by `prepares` to properly format simulation and training data, and used by `closeloop` to add the correct number of delays when closing an open-loop output, and `openloop` to remove delays when opening a closed loop.

net.outputs{i}.feedbackMode

This property is set to the string '`none`' for non-feedback outputs. For feedback outputs it can either be set to '`open`' or '`closed`'. If it is set to '`open`', then the output will be associated with a feedback input, with the property `feedbackInput` indicating the input's index.

net.outputs{i}.processFcns

This property defines a row cell array of processing function names to be used by the *i*th network output. The processing functions are applied to target values before the network uses them, and applied in reverse to layer output values before being returned as network output values.

Side Effects

When you change this property, you also affect the following settings: the output parameters `processParams` are modified to the default values of the specified

processing functions; `processSettings`, `processedSize`, and `processedRange` are defined using the results of applying the process functions and parameters to `exampleOutput`; the *i*th layer size is updated to match the `processedSize`.

For a list of functions, type `help nnprocess`.

net.outputs{i}.processParams

This property holds a row cell array of processing function parameters to be used by *i*th network output on target values. The processing parameters are applied by the processing functions to input values before the network uses them.

Side Effects

Whenever this property is altered, the output `processSettings`, `processedSize` and `processedRange` are defined by applying the process functions and parameters to `exampleOutput`. The *i*th layer's size is also updated to match `processedSize`.

net.outputs{i}.processSettings (read only)

This property holds a row cell array of processing function settings to be used by *i*th network output. The processing settings are found by applying the processing functions and parameters to `exampleOutput` and then used to provide consistent results to new target values before the network uses them. The processing settings are also applied in reverse to layer output values before being returned by the network.

net.outputs{i}.processedRange (read only)

This property defines the range of `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

net.outputs{i}.processedSize (read only)

This property defines the number of rows in the `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

net.outputs{i}.size (read only)

This property defines the number of elements in the *i*th layer's output. It is always set to the size of the *i*th layer (`net.layers{i}.size`).

net.outputs{i}.userdata

This property provides a place for users to add custom information to the *i*th layer's output.

Biases

net.biases{i}.initFcn

This property defines the weight and bias initialization functions used to set the *i*th layer's bias vector (`net.b{i}`) if the network initialization function is `initlay` and the *i*th layer's initialization function is `initwb`.

net.biases{i}.learn

This property defines whether the *i*th bias vector is to be altered during training and adaption. It can be set to 0 or 1.

It enables or disables the bias's learning during calls to `adapt` and `train`.

net.biases{i}.learnFcn

This property defines which of the learning functions is used to update the *i*th layer's bias vector (`net.b{i}`) during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

For a list of functions, type `help nnlearn`.

Side Effects

Whenever this property is altered, the biases learning parameters (`net.biases{i}.learnParam`) are set to contain the fields and default values of the new function.

net.biases{i}.learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias. The fields of this property depend on the current learning function. Call `help` on the current learning function to get a description of what each field means.

net.biases{i}.size (read only)

This property defines the size of the i th layer's bias vector. It is always set to the size of the i th layer (`net.layers{i}.size`).

net.biases{i}.userdata

This property provides a place for users to add custom information to the i th layer's bias.

Input Weights

net.inputWeights{i,j}.delays

This property defines a tapped delay line between the j th input and its weight to the i th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

Side Effects

Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

net.inputWeights{i,j}.initFcn

This property defines which of the Weight and Bias Initialization Functions is used to initialize the weight matrix (`net.IW{i,j}`) going to the i th layer from the j th input, if the network initialization function is `initlay`, and the i th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

net.inputWeights{i,j}.initSettings (read only)

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

net.inputWeights{i,j}.learn

This property defines whether the weight matrix to the i th layer from the j th input is to be altered during training and adaption. It can be set to 0 or 1.

net.inputWeights{i,j}.learnFcn

This property defines which of the learning functions is used to update the weight matrix (`net.IW{i,j}`) going to the i th layer from the j th input during training, if the network

training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

net.inputWeights{i,j}.learnParam

This property defines the learning parameters and values for the current learning function of the i th layer's weight coming from the j th input.

The fields of this property depend on the current learning function (`net.inputWeights{i,j}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

net.inputWeights{i,j}.size (read only)

This property defines the dimensions of the i th layer's weight matrix from the j th network input. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.IW{i,j}`). The first element is equal to the size of the i th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the j th input:

```
length(net.inputWeights{i,j}.delays) * net.inputs{j}.size
```

net.inputWeights{i,j}.userdata

This property provides a place for users to add custom information to the (i,j) th input weight.

net.inputWeights{i,j}.weightFcn

This property defines which of the weight functions is used to apply the i th layer's weight from the j th input to that input. It can be set to the name of any weight function. The weight function is used to transform layer inputs during simulation and training.

For a list of functions, type `help nnweight`.

net.inputWeights{i,j}.weightParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

Layer Weights

net.layerWeights{i,j}.delays

This property defines a tapped delay line between the j th layer and its weight to the i th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

net.layerWeights{i,j}.initFcn

This property defines which of the weight and bias initialization functions is used to initialize the weight matrix ($\text{net}.\text{LW}\{i,j\}$) going to the i th layer from the j th layer, if the network initialization function is `initlay`, and the i th layer's initialization function is `initwrb`. This function can be set to the name of any weight initialization function.

net.layerWeights{i,j}.initSettings (read only)

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

net.layerWeights{i,j}.learn

This property defines whether the weight matrix to the i th layer from the j th layer is to be altered during training and adaption. It can be set to 0 or 1.

net.layerWeights{i,j}.learnFcn

This property defines which of the learning functions is used to update the weight matrix ($\text{net}.\text{LW}\{i,j\}$) going to the i th layer from the j th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

net.layerWeights{i,j}.learnParam

This property defines the learning parameters fields and values for the current learning function of the i th layer's weight coming from the j th layer. The fields of this property depend on the current learning function. Call `help` on the current net input function to get a description of each field.

net.layerWeights{i,j}.size (read only)

This property defines the dimensions of the i th layer's weight matrix from the j th layer. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.LW{i,j}`). The first element is equal to the size of the i th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the j th layer.

net.layerWeights{i,j}.userdata

This property provides a place for users to add custom information to the (i,j) th layer weight.

net.layerWeights{i,j}.weightFcn

This property defines which of the weight functions is used to apply the i th layer's weight from the j th layer to that layer's output. It can be set to the name of any weight function. The weight function is used to transform layer inputs when the network is simulated.

For a list of functions, type `help nnweight`.

net.layerWeights{i,j}.weightParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

Shallow Neural Networks

Bibliography

Shallow Neural Networks Bibliography

[Batt92] Battiti, R., "First and second order methods for learning: Between steepest descent and Newton's method," *Neural Computation*, Vol. 4, No. 2, 1992, pp. 141-166.

[Beal72] Beale, E.M.L., "A derivation of conjugate gradients," in F.A. Lootsma, Ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.

[Bren73] Brent, R.P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

[Caud89] Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[CaBu92] Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations, Vols. 1 and 2*, Cambridge, MA: The MIT Press, 1992.

This is a two-volume workbook designed to give students "hands on" experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear, and helpful in understanding a field that traditionally has been buried in mathematics.

[Char92] Charalambous, C., "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEEE Proceedings*, Vol. 139, No. 3, 1992, pp. 301-310.

[ChCo91] Chen, S., C.F.N. Cowan, and P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, 1991, pp. 302-309.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[ChDa99] Chengyu, G., and K. Danai, "Fault diagnosis of the IFAC Benchmark Problem with a model-based recurrent neural network," *Proceedings of the 1999 IEEE International Conference on Control Applications*, Vol. 2, 1999, pp. 1755-1760.

[DARP88] *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

[DeHa01a] De Jesús, O., and M.T. Hagan, "Backpropagation Through Time for a General Class of Recurrent Network," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15-19, 2001, pp. 2638-2642.

[DeHa01b] De Jesús, O., and M.T. Hagan, "Forward Perturbation Algorithm for a General Class of Recurrent Network," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15-19, 2001, pp. 2626-2631.

[DeHa07] De Jesús, O., and M.T. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," *IEEE Transactions on Neural Networks*, Vol. 18, No. 1, January 2007, pp. 14 -27.

This paper provides detailed algorithms for the calculation of gradients and Jacobians for arbitrarily-connected neural networks. Both the backpropagation-through-time and real-time recurrent learning algorithms are covered.

[DeSc83] Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[DHH01] De Jesús, O., J.M. Horn, and M.T. Hagan, "Analysis of Recurrent Network Training and Suggestions for Improvements," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15-19, 2001, pp. 2632-2637.

[Elma90] Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179-211.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

[FeTs03] Feng, J., C.K. Tse, and F.C.M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 50, No. 7, 2003, pp. 954-957.

[FlRe64] Fletcher, R., and C.M. Reeves, "Function minimization by conjugate gradients," *Computer Journal*, Vol. 7, 1964, pp. 149-154.

[FoHa97] Foressee, F.D., and M.T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997, pp. 1930-1935.

[GiMu81] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

[GiPr02] Gianluca, P., D. Przybylski, B. Rost, P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," *Proteins: Structure, Function, and Genetics*, Vol. 47, No. 2, 2002, pp. 228-235.

[Gros82] Grossberg, S., *Studies of the Mind and Brain*, Dordrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642-1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311-340.

[HaMe94] Hagan, M.T., and M. Menhaj, "Training feed-forward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989-993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

[HaRu78] Harrison, D., and Rubinfeld, D.L., "Hedonic prices and the demand for clean air," *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

[HDB96] Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor's guide, and transparency overheads for teaching.

[HDH09] Horn, J.M., O. De Jesús and M.T. Hagan, "Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance," *IEEE Transactions on Neural Networks*, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

[Hebb49] Hebb, D.O., *The Organization of Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

[Himm72] Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

[HuSb92] Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, "Neural Networks for Control System – A Survey," *Automatica*, Vol. 28, 1992, pp. 1083-1112.

[JaRa04] Jayadeva and S.A.Rahman, "A neural network with $O(N)$ neurons for ranking N numbers in $O(1/N)$ time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044-2051.

[Joll86] Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

[KaGr96] Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen, and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657-664.

[Koho87] Kohonen, T., *Self-Organization and Associative Memory*, 2nd Edition, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

[Koho97] Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A.N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405-1422.

This paper discusses a class of neural networks described by first-order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User's Guide*.

[Lipp87] Lippman, R.P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, 1987, pp. 4-22.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

[MacK92] MacKay, D.J.C., "Bayesian interpolation," *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415-447.

[Marq63] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431-441.

[McPi43] McCulloch, W.S., and W.H. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, 1943, pp. 115-133.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

[MeJa00] Medsker, L.R., and L.C. Jain, *Recurrent neural networks: design and applications*, Boca Raton, FL: CRC Press, 2000.

[Moll93] Moller, M.F., "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, Vol. 6, 1993, pp. 525-533.

[MuNe92] Murray, R., D. Neumerkel, and D. Sbarbaro, "Neural Networks for Modeling and Control of a Non-linear Dynamic System," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404-409.

[NaMu97] Narendra, K.S., and S. Mukhopadhyay, "Adaptive Control Using Neural Networks and Approximate Models," *IEEE Transactions on Neural Networks*, Vol. 8, 1997, pp. 475-485.

[NaPa91] Narendra, Kumpati S. and Kannan Parthasarathy, "Learning Automata Approach to Hierarchical Multiobjective Analysis," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, No. 1, January/February 1991, pp. 263-272.

[NgWi89] Nguyen, D., and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, 1989, pp. 357-363.

This paper describes a two-layer network that first learned the truck dynamics and then learned how to back the truck to a specified position at a loading dock. To do this, the neural network had to solve a highly nonlinear control systems problem.

[NgWi90] Nguyen, D., and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1990, pp. 21-26.

Nguyen and Widrow show that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

[Powe77] Powell, M.J.D., "Restart procedures for the conjugate gradient method," *Mathematical Programming*, Vol. 12, 1977, pp. 241-254.

[Pulu92] Purdie, N., E.A. Lucas, and M.B. Talley, "Direct measure of total cholesterol and its distribution among major serum lipoproteins," *Clinical Chemistry*, Vol. 38, No. 9, 1992, pp. 1645-1647.

[RiBr93] Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[Robin94] Robinson, A.J., "An application of recurrent nets to phone probability estimation," *IEEE Transactions on Neural Networks*, Vol. 5 , No. 2, 1994.

[RoJa96] Roman, J., and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, Vol. 2, 1996, pp. 454-460.

[Rose61] Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt's results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

[RuHi86a] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," in D.E. Rumelhart and J.L. McClelland, Eds., *Parallel Data Processing*, Vol. 1, Cambridge, MA: The M.I.T. Press, 1986, pp. 318-362.

This is a basic reference on backpropagation.

[RuHi86b] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, 1986, pp. 533-536.

[RuMc86] Rumelhart, D.E., J.L. McClelland, and the PDP Research Group, Eds., *Parallel Distributed Processing, Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of publication.

[Scal85] Scales, L.E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[SoHa96] Soloway, D., and P.J. Haley, "Neural Generalized Predictive Control," *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277-281.

[VoMa88] Vogl, T.P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, Vol. 59, 1988, pp. 256-264.

Backpropagation learning can be speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

[WaHa89] Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, 1989, pp. 328-339.

[Wass93] Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

[WeGe94] Weigend, A. S., and N. A. Gershenfeld, eds., *Time Series Prediction: Forecasting the Future and Understanding the Past*, Reading, MA: Addison-Wesley, 1994.

[WiHo60] Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96-104.

[WiSt85] Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

Mathematical Notation

Mathematics and Code Equivalents

In this section...

[“Mathematics Notation to MATLAB Notation” on page A-2](#)

[“Figure Notation” on page A-2](#)

The transition from mathematics to code or vice versa can be made with the aid of a few rules. They are listed here for reference.

Mathematics Notation to MATLAB Notation

To change from mathematics notation to MATLAB notation:

- Change superscripts to cell array indices. For example,

$$p^1 \rightarrow p\{1\}$$

- Change subscripts to indices within parentheses. For example,

$$p_2 \rightarrow p(2)$$

and

$$p_2^1 \rightarrow p\{1\}(2)$$

- Change indices within parentheses to a second cell array index. For example,

$$p^1(k - 1) \rightarrow p\{1, k - 1\}$$

- Change mathematics operators to MATLAB operators and toolbox functions. For example,

$$ab \rightarrow a * b$$

Figure Notation

The following equations illustrate the notation used in figures.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

Neural Network Blocks for the Simulink Environment

Neural Network Simulink Block Library

In this section...

[“Transfer Function Blocks” on page B-3](#)

[“Net Input Blocks” on page B-3](#)

[“Weight Blocks” on page B-3](#)

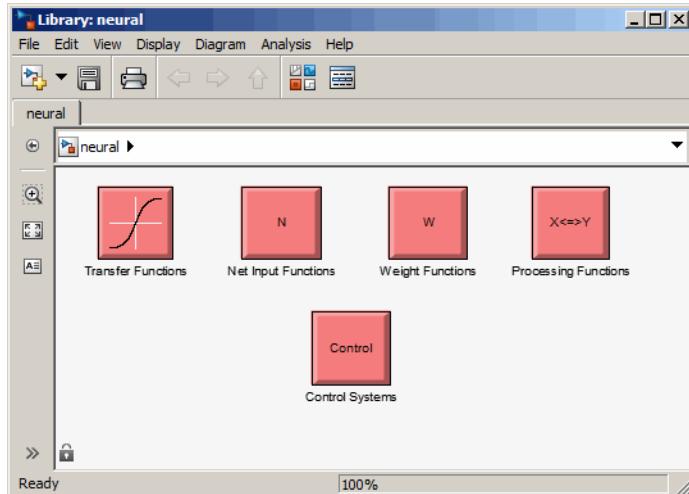
[“Processing Blocks” on page B-4](#)

The Deep Learning Toolbox product provides a set of blocks you can use to build neural networks using Simulink software, or that the function `gensim` can use to generate the Simulink version of any network you have created using MATLAB software.

Open the Deep Learning Toolbox block library with the command:

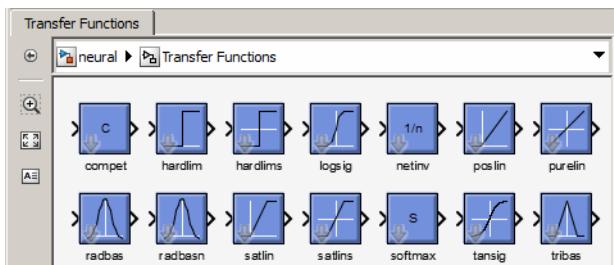
```
neural
```

This opens a library window that contains five blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

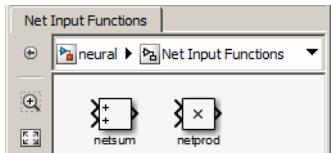
Double-click the Transfer Functions block in the Neural library window to open a window containing several transfer function blocks.



Each of these blocks takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector.

Net Input Blocks

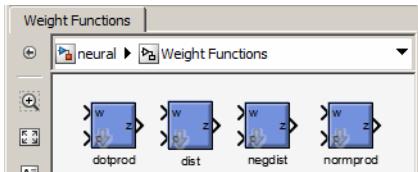
Double-click the Net Input Functions block in the Neural library window to open a window containing two net-input function blocks.



Each of these blocks takes any number of weighted input vectors, weight layer output vectors, and bias vectors, and returns a net-input vector.

Weight Blocks

Double-click the Weight Functions block in the Neural library window to open a window containing three weight function blocks.



Each of these blocks takes a neuron's weight vector and applies it to an input vector (or a layer output vector) to get a weighted input value for a neuron.

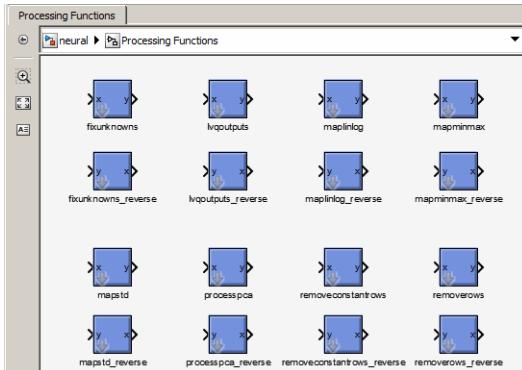
It is important to note that these blocks expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

It is also important to note that because of this limitation you have to create S weight function blocks (one for each row), to implement a weight matrix going to a layer with S neurons.

This contrasts with the other two kinds of blocks. Only one net input function and one transfer function block are required for each layer.

Processing Blocks

Double-click the Processing Functions block in the Neural library window to open a window containing processing blocks and their corresponding reverse-processing blocks.



Each of these blocks can be used to preprocess inputs and postprocess outputs.

Deploy Shallow Neural Network Simulink Diagrams

In this section...

"Example" on page B-5

"Suggested Exercises" on page B-7

"Generate Functions and Objects" on page B-8

The function `gensim` generates block descriptions of networks so you can simulate them using Simulink software.

```
gensim(net,st)
```

The second argument to `gensim` determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated with its input weights or layer weights, this value can be set to -1. A value of -1 causes `gensim` to generate a network with continuous sampling.

Example

Here is a simple problem defining a set of inputs `p` and corresponding targets `t`.

```
p = [1 2 3 4 5];  
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p,t)
```

You can test the network on your original inputs with `sim`.

```
y = sim(net,p)
```

The results show the network has solved the problem.

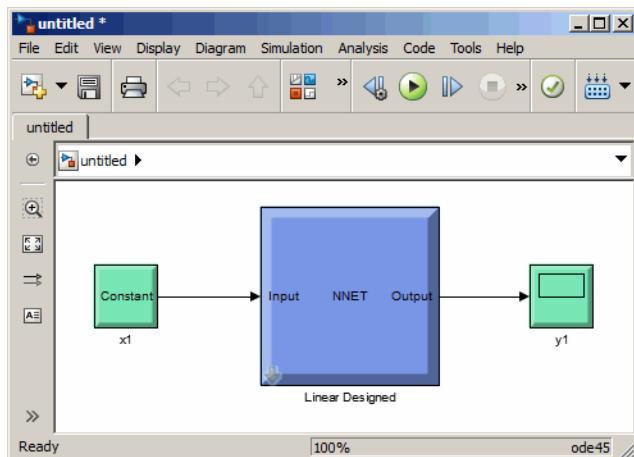
```
y =  
1.0000    3.0000    5.0000    7.0000    9.0000
```

Call `gensim` as follows to generate a Simulink version of the network.

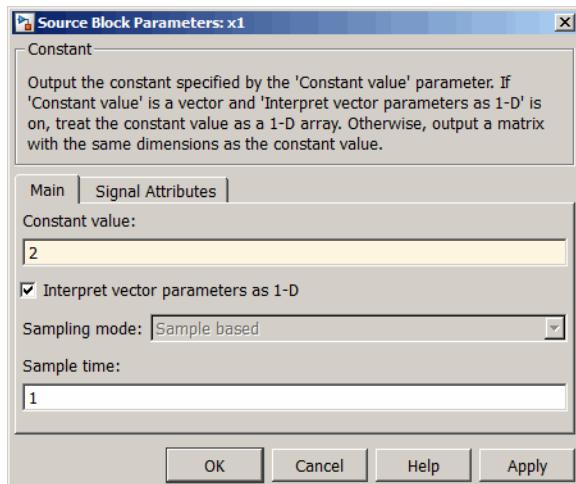
```
gensim(net,-1)
```

The second argument is -1, so the resulting network block samples continuously.

The call to `gensim` opens the following Simulink Editor, showing a system consisting of the linear network connected to a sample input and a scope.



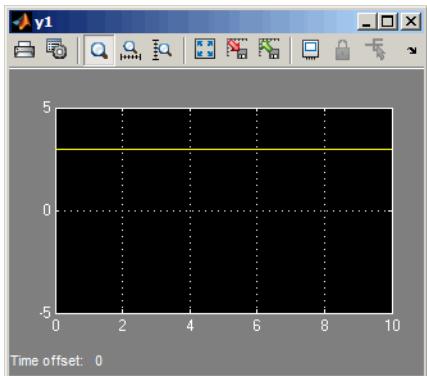
To test the network, double-click the input Constant x_1 block on the left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **OK**.

Select the menu option **Simulation > Run**. Simulink takes a moment to simulate the system.

When the simulation is complete, double-click the output **y1** block on the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

Suggested Exercises

Here are a couple exercises you can try.

Change the Input Signal

Replace the constant input block with a signal generator from the standard Simulink Sources blockset. Simulate the system and view the network's response.

Use a Discrete Sample Time

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net, 0.5)
```

Again, replace the constant input with a signal generator. Simulate the system and view the network's response.

Generate Functions and Objects

For information on simulating and deploying shallow neural networks with MATLAB functions, see “Deploy Shallow Neural Network Functions” on page 11-63.

Code Notes

Deep Learning Toolbox Data Conventions

In this section...

[“Dimensions” on page C-2](#)

[“Variables” on page C-2](#)

Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

<code>Ni</code> = Number of network inputs	= <code>net.numInputs</code>
<code>Ri</code> = Number of elements in input <i>i</i>	= <code>net.inputs{i}.size</code>
<code>Nl</code> = Number of layers	= <code>net.numLayers</code>
<code>Si</code> = Number of neurons in layer <i>i</i>	= <code>net.layers{i}.size</code>
<code>Nt</code> = Number of targets	
<code>Vi</code> = Number of elements in target <i>i</i> , equal to <code>Sj</code> , where <i>j</i> is the <i>i</i> th layer with a target. (A layer <i>n</i> has a target if <code>net.targets(n) == 1.</code>)	
<code>No</code> = Number of network outputs	
<code>Ui</code> = Number of elements in output <i>i</i> , equal to <code>Sj</code> , where <i>j</i> is the <i>i</i> th layer with an output (A layer <i>n</i> has an output if <code>net.outputs(n) == 1.</code>)	
<code>ID</code> = Number of input delays	= <code>net.numInputDelays</code>
<code>LD</code> = Number of layer delays	= <code>net.numLayerDelays</code>
<code>TS</code> = Number of time steps	
<code>Q</code> = Number of concurrent vectors or sequences	

Variables

The variables a user commonly uses when defining a simulation or training session are

P	Network inputs	Ni-by-TS cell array, where each element $P\{i, ts\}$ is an Ri-by-Q matrix
Pi	Initial input delay conditions	Ni-by-ID cell array, where each element $Pi\{i, k\}$ is an Ri-by-Q matrix
Ai	Initial layer delay conditions	Nl-by-LD cell array, where each element $Ai\{i, k\}$ is an Si-by-Q matrix
T	Network targets	Nt-by-TS cell array, where each element $P\{i, ts\}$ is a Vi-by-Q matrix

These variables are returned by simulation and training calls:

Y	Network outputs	No-by-TS cell array, where each element $Y\{i, ts\}$ is a Ui-by-Q matrix
E	Network errors	Nt-by-TS cell array, where each element $P\{i, ts\}$ is a Vi-by-Q matrix
perf	Network performance	

Utility Function Variables

These variables are used only by the utility functions.

Pc	Combined inputs	Ni-by-(ID+TS) cell array, where each element $P\{i, ts\}$ is an Ri-by-Q matrix Pc = [Pi P] = Initial input delay conditions and network inputs
----	-----------------	---

Pd	Delayed inputs	<p>Ni-by-Nj-by-TS cell array, where each element $Pd\{i,j,ts\}$ is an $(Ri*IWD(i,j))$-by-Q matrix, and where $IWD(i,j)$ is the number of delay taps associated with the input weight to layer i from input j</p> <p>Equivalently,</p> <pre>IWD(i,j) = length(net.inputWeights{i,j}.delays)</pre> <p>Pd is the result of passing the elements of P through each input weight's tap delay lines. Because inputs are always transformed by input delays in the same way, it saves time to do that operation only once instead of for every training step.</p>
BZ	Concurrent bias vectors	<p>Nl-by-1 cell array, where each element $BZ\{i\}$ is an Si-by-Q matrix</p> <p>Each matrix is simply Q copies of the $net.b\{i\}$ bias vector.</p>
IWZ	Weighted inputs	Ni-by-Nl-by-TS cell array, where each element $IWZ\{i,j,ts\}$ is an Si-by-???-by-Q matrix
LWZ	Weighted layer outputs	Ni-by-Nl-by-TS cell array, where each element $LWZ\{i,j,ts\}$ is an Si-by-Q matrix
N	Net inputs	Ni-by-TS cell array, where each element $N\{i,ts\}$ is an Si-by-Q matrix
A	Layer outputs	Nl-by-TS cell array, where each element $A\{i,ts\}$ is an Si-by-Q matrix
Ac	Combined layer outputs	<p>Nl-by-(LD+TS) cell array, where each element $A\{i,ts\}$ is an Si-by-Q matrix</p> <p>$Ac = [Ai A]$ = Initial layer delay conditions and layer outputs.</p>
Tl	Layer targets	<p>Nl-by-TS cell array, where each element $Tl\{i,ts\}$ is an Si-by-Q matrix</p> <p>Tl contains empty matrices [] in rows of layers i not associated with targets, indicated by $net.targets(i) == 0$.</p>

E _l	Layer errors	N _l -by-TS cell array, where each element E _l {i, ts} is an S _i -by-Q matrix E _l contains empty matrices [] in rows of layers i not associated with targets, indicated by net.targets(i) == 0.
X	Column vector of all weight and bias values	

