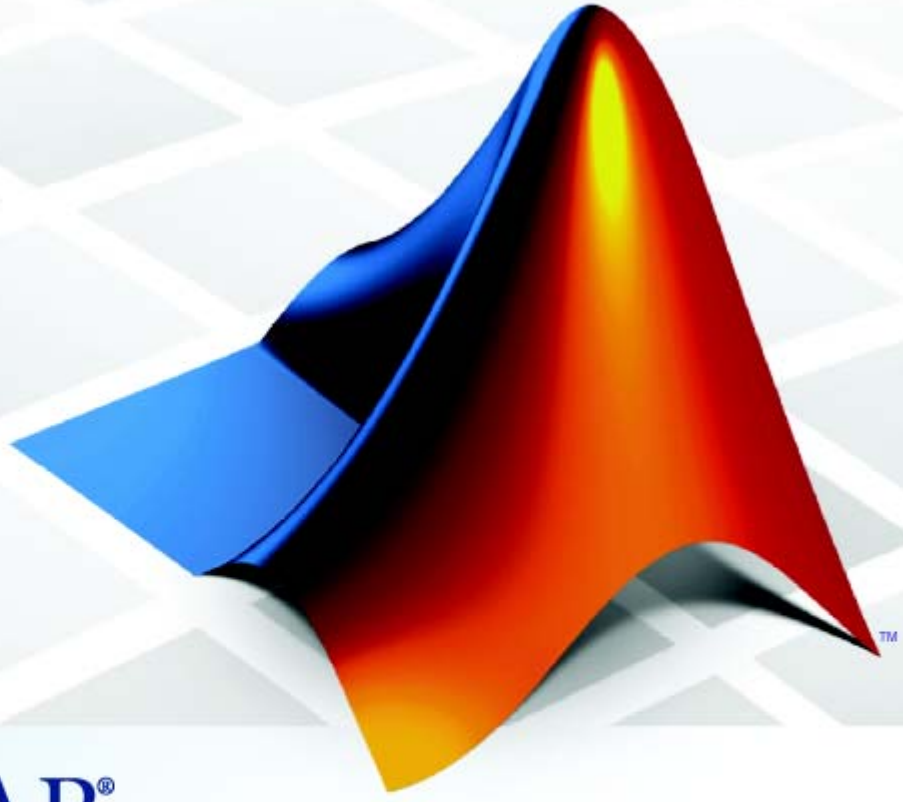
Neural Network Toolbox™ 6
User's Guide

Howard Demuth
Mark Beale
Martin Hagan



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Neural Network Toolbox™ User's Guide

© COPYRIGHT 1992–2009 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

The 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)

Acknowledgments

The authors would like to thank the following people:

Joe Hicklin of The MathWorks™ for getting Howard into neural network research years ago at the University of Idaho, for encouraging Howard and Mark to write the toolbox, for providing crucial help in getting the first toolbox Version 1.0 out the door, for continuing to help with the toolbox in many ways, and for being such a good friend.

Roy Lurie of The MathWorks for his continued enthusiasm for the possibilities for Neural Network Toolbox™ software.

Mary Ann Freeman for general support and for her leadership of a great team of people we enjoy working with.

Rakesh Kumar for cheerfully providing technical and practical help, encouragement, ideas and always going the extra mile for us.

Sarah Lemaire for facilitating our documentation work.

Tara Scott and Stephen Vanreusal for help with testing.

Orlando De Jesús of Oklahoma State University for his excellent work in developing and programming the dynamic training algorithms described in Chapter 6, “Dynamic Networks,” and in programming the neural network controllers described in Chapter 7, “Control Systems.”

Martin Hagan, **Howard Demuth**, and **Mark Beale** for permission to include various problems, demonstrations, and other material from *Neural Network Design*, January, 1996.

Neural Network Toolbox™ Design Book

The developers of the Neural Network 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 Neural Network Toolbox software. Demonstration programs from the book are used in various chapters of this user's guide. (You can find all the book demonstration programs in the Neural Network Toolbox software by typing `nnd`.)

This book can be obtained from John Stovall at (303) 492-3648, or by e-mail 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 e-mail 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.

Contents

1 Getting Started

Product Overview 1-2

Using the Documentation 1-3

Applications for Neural Network Toolbox™ Software 1-4
Applications in This Toolbox 1-4
Business Applications 1-4

Fitting a Function 1-7
Defining a Problem 1-7
Using Command-Line Functions 1-7
Using the Neural Network Toolbox™ Fitting Tool GUI 1-13

Recognizing Patterns 1-24
Defining a Problem 1-24
Using Command-Line Functions 1-25
Using the Neural Network Toolbox™ Pattern Recognition Tool GUI 1-31

Clustering Data 1-42
Defining a Problem 1-42
Using Command-Line Functions 1-43
Using the Neural Network Toolbox™ Clustering Tool GUI . . 1-47

2 Neuron Model and Network Architectures

Neuron Model 2-2
Simple Neuron 2-2
Transfer Functions 2-3
Neuron with Vector Input 2-5

Network Architectures **2-8**
A Layer of Neurons **2-8**
Multiple Layers of Neurons **2-10**
Input and Output Processing Functions **2-12**

Data Structures **2-14**
Simulation with Concurrent Inputs in a Static Network **2-14**
Simulation with Sequential Inputs in a Dynamic Network . . **2-15**
Simulation with Concurrent Inputs in a Dynamic Network . . **2-17**

Training Styles **2-20**
Incremental Training (of Adaptive and Other Networks) **2-20**
Batch Training **2-22**
Training Feedback **2-25**

3 Perceptrons

Introduction **3-2**
Important Perceptron Functions **3-2**

Neuron Model **3-3**

Perceptron Architecture **3-5**

Creating a Perceptron (newp) **3-6**
Simulation (sim) **3-8**
Initialization (init) **3-10**

Learning Rules **3-13**

Perceptron Learning Rule (learnp) **3-14**

Training (train) **3-17**

Limitations and Cautions **3-23**

Outliers and the Normalized Perceptron Rule 3-23

Graphical User Interface 3-25
Introduction to the GUI 3-25
Create a Perceptron Network (nntool) 3-25
Train the Perceptron 3-29
Export Perceptron Results to Workspace 3-31
Clear Network/Data Window 3-32
Importing from the Command Line 3-32
Save a Variable to a File and Load It Later 3-33

4 Linear Filters

Introduction 4-2

Neuron Model 4-3

Network Architecture 4-4
Creating a Linear Neuron (newlin) 4-4

Least Mean Square Error 4-8

Linear System Design (newlind) 4-9

Linear Networks with Delays 4-10
Tapped Delay Line 4-10
Linear Filter 4-10

LMS Algorithm (learnwh) 4-13

Linear Classification (train) 4-15

Limitations and Cautions 4-18
Overdetermined Systems 4-18
Underdetermined Systems 4-18

Linearly Dependent Vectors 4-18
Too Large a Learning Rate 4-19

5 Backpropagation

Introduction 5-2

Solving a Problem 5-4
Improving Results 5-6
Under the Hood 5-6

Architecture 5-8
Feedforward Network 5-10

Simulation (sim) 5-14

Training 5-15
Backpropagation Algorithm 5-15

Faster Training 5-19
Variable Learning Rate (traingda, traingdx) 5-19
Resilient Backpropagation (trainrp) 5-21
Conjugate Gradient Algorithms 5-22
Line Search Routines 5-26
Quasi-Newton Algorithms 5-29
Levenberg-Marquardt (trainlm) 5-30
Reduced Memory Levenberg-Marquardt (trainlm) 5-32

Speed and Memory Comparison 5-34
Summary 5-50

Improving Generalization 5-52
Early Stopping 5-53
Index Data Division (divideind) 5-54
Random Data Division (dividerand) 5-54
Block Data Division (divideblock) 5-54

Interleaved Data Division (dividerand) 5-55
Regularization 5-55
Summary and Discussion of Early Stopping and Regularization 5-58

Preprocessing and Postprocessing 5-61
Min and Max (mapminmax) 5-62
Mean and Stand. Dev. (mapstd) 5-63
Principal Component Analysis (processpca) 5-64
Processing Unknown Inputs (fixunknowns) 5-65
Representing Unknown or Don't Care Targets 5-66
Posttraining Analysis (postreg) 5-66

Sample Training Session 5-68

Limitations and Cautions 5-71

6 Dynamic Networks

Introduction 6-2
Examples of Dynamic Networks 6-2
Applications of Dynamic Networks 6-7
Dynamic Network Structures 6-8
Dynamic Network Training 6-9

Focused Time-Delay Neural Network (newfftd) 6-11

Distributed Time-Delay Neural Network (newdtdnn) 6-15

NARX Network (newnarx, newnarxsp, sp2narx) 6-18

Layer-Recurrent Network (newlrn) 6-24

7 Control Systems

Introduction **7-2**

NN Predictive Control **7-5**
System Identification **7-5**
Predictive Control **7-6**
Using the NN Predictive Controller Block **7-7**

NARMA-L2 (Feedback Linearization) Control **7-16**
Identification of the NARMA-L2 Model **7-16**
NARMA-L2 Controller **7-18**
Using the NARMA-L2 Controller Block **7-20**

Model Reference Control **7-25**
Using the Model Reference Controller Block **7-27**

Importing and Exporting **7-33**
Importing and Exporting Networks **7-33**
Importing and Exporting Training Data **7-37**

8 Radial Basis Networks

Introduction **8-2**
Important Radial Basis Functions **8-2**

Radial Basis Functions **8-3**
Neuron Model **8-3**
Network Architecture **8-4**
Exact Design (newrbe) **8-5**
More Efficient Design (newrb) **8-7**
Demonstrations **8-8**

Probabilistic Neural Networks **8-9**
Network Architecture **8-9**

Design (newpnn) 8-10

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

9 Self-Organizing and Learning Vector Quantization Nets

Introduction 9-2
Important Self-Organizing and LVQ Functions 9-2

Competitive Learning 9-3
Architecture 9-3
Creating a Competitive Neural Network (newc) 9-4
Kohonen Learning Rule (learnk) 9-5
Bias Learning Rule (learncon) 9-5
Training 9-6
Graphical Example 9-8

Self-Organizing Feature Maps 9-9
Topologies (gridtop, hextop, randtop) 9-10
Distance Functions (dist, linkdist, mandist, boxdist) 9-14
Architecture 9-17
Creating a Self-Organizing MAP Neural Network (newsom) . 9-18
Training (learnsomb) 9-19
Examples 9-22

Learning Vector Quantization Networks 9-35
Architecture 9-35
Creating an LVQ Network (newlvq) 9-36
LVQ1 Learning Rule (learnlv1) 9-39
Training 9-40
Supplemental LVQ2.1 Learning Rule (learnlv2) 9-42

10 Adaptive Filters and Adaptive Training

Introduction 10-2
Important Adaptive Functions 10-2

Linear Neuron Model 10-3

Adaptive Linear Network Architecture 10-4
Single ADALINE (newlin) 10-4

Least Mean Square Error 10-7

LMS Algorithm (learnwh) 10-8

Adaptive Filtering (adapt) 10-9
Tapped Delay Line 10-9
Adaptive Filter 10-9
Adaptive Filter Example 10-10
Prediction Example 10-13
Noise Cancellation Example 10-14
Multiple Neuron Adaptive Filters 10-16

11 Applications

Introduction 11-2
Application Scripts 11-2

Applin1: Linear Design 11-3
Problem Definition 11-3
Network Design 11-4
Network Testing 11-4
Thoughts and Conclusions 11-6

Applin2: Adaptive Prediction 11-7
Problem Definition 11-7

Network Initialization 11-8
Network Training 11-8
Network Testing 11-8
Thoughts and Conclusions 11-10

Appelm1: Amplitude Detection 11-11
Problem Definition 11-11
Network Initialization 11-11
Network Training 11-12
Network Testing 11-12
Network Generalization 11-13
Improving Performance 11-14

Appcr1: Character Recognition 11-15
Problem Statement 11-15
Neural Network 11-16
System Performance 11-19

12 Advanced Topics

Custom Networks 12-2
Custom Network 12-2
Network Definition 12-3
Network Behavior 12-13

Additional Toolbox Functions 12-16

Custom Functions 12-17

13 Historical Networks

Introduction 13-2

Important Recurrent Network Functions 13-2

Elman Networks 13-3
Architecture 13-3
Creating an Elman Network (newelm) 13-4
Training an Elman Network 13-5

Hopfield Network 13-8
Fundamentals 13-8
Architecture 13-8
Design (newhop) 13-10

14 Network Object Reference

Network Properties 14-2
Architecture 14-2
Subobject Structures 14-5
Functions 14-7
Parameters 14-10
Weight and Bias Values 14-11
Other 14-12

Subobject Properties 14-13
Inputs 14-13
Layers 14-15
Outputs 14-20
Biases 14-22
Input Weights 14-23
Layer Weights 14-25

15 Function Reference

Analysis Functions 15-3

Distance Functions 15-4

Graphical Interface Functions 15-5

Layer Initialization Functions 15-6

Learning Functions 15-7

Line Search Functions 15-8

Net Input Functions 15-9

Network Initialization Function 15-10

Network Use Functions 15-11

New Networks Functions 15-12

Performance Functions 15-13

Plotting Functions 15-14

Processing Functions 15-15

Simulink® Support Function 15-16

Topology Functions 15-17

Training Functions 15-18

Transfer Functions 15-19

Utility Functions 15-20

Vector Functions **15-21**

Weight and Bias Initialization Functions **15-22**

Weight Functions **15-23**

Transfer Function Graphs **15-24**

16 Functions — Alphabetical List

A Mathematical Notation

Mathematical Notation for Equations and Figures **A-2**
Basic Concepts **A-2**
Language **A-2**
Weight Matrices **A-2**
Bias Elements and Vectors **A-2**
Time and Iteration **A-2**
Layer Notation **A-3**
Figure and Equation Examples **A-3**

Mathematics and Code Equivalents **A-4**

B Demonstrations and Applications

Tables of Demonstrations and Applications **B-2**
Chapter 2, “Neuron Model and Network Architectures” **B-2**
Chapter 3, “Perceptrons” **B-2**
Chapter 4, “Linear Filters” **B-3**

Chapter 5, “Backpropagation” B-3
Chapter 8, “Radial Basis Networks” B-4
Chapter 9, “Self-Organizing and Learning Vector Quantization Nets” B-4
Chapter 10, “Adaptive Filters and Adaptive Training” B-4
Chapter 11, “Applications” B-5
Chapter 13, “Historical Networks” B-5

C Blocks for the Simulink® Environment

Blockset C-2
Transfer Function Blocks C-2
Net Input Blocks C-3
Weight Blocks C-3
Processing Blocks C-4

Block Generation C-5
Example C-5
Exercises C-7

D Code Notes

Dimensions D-2

Variables D-3
Utility Function Variables D-4

Functions D-6

Code Efficiency D-7

Argument Checking D-8

Bibliography

E

Glossary

Index

1

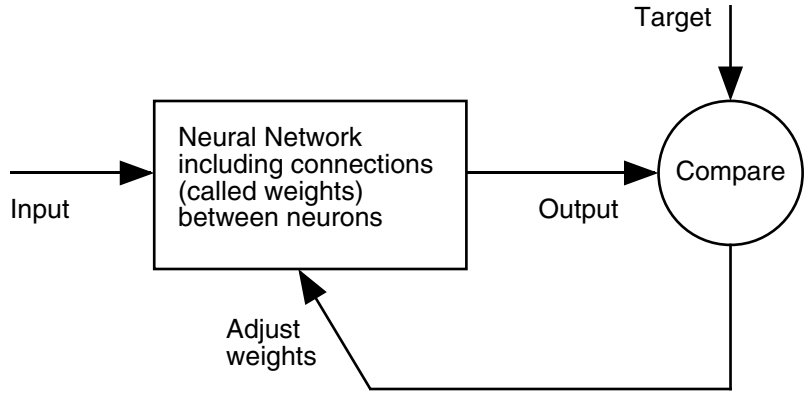
Getting Started

Product Overview (p. 1-2)
Using the Documentation (p. 1-3)
Applications for Neural Network Toolbox™ Software (p. 1-4)
Fitting a Function (p. 1-7)
Recognizing Patterns (p. 1-24)
Clustering Data (p. 1-42)

Product Overview

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

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



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

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

The next sections explain how to use three graphical tools for training neural networks to solve problems in function fitting, pattern recognition, and clustering.

Using the Documentation

The neuron model and the architecture of a neural network describe how a network transforms its input into an output. This transformation can be viewed as a computation.

This first chapter gives you an overview of the Neural Network Toolbox™ product and introduces you to the following tasks:

- Training a neural network to fit a function
- Training a neural network to recognize patterns
- Training a neural network to cluster data

These next two chapters explain the computations that are done and pave the way for an understanding of training methods for the networks. You should read them before advancing to later topics:

- Chapter 2, “Neuron Model and Network Architectures,” presents the fundamentals of the neuron model, the architectures of neural networks. It also discusses the notation used in this toolbox.
- Chapter 3, “Perceptrons,” explains how to create and train simple networks. It also introduces a graphical user interface (GUI) that you can use to solve problems without a lot of coding.

Applications for Neural Network Toolbox™ Software

Applications in This Toolbox

Chapter 7, “Control Systems” describes three practical neural network control system applications, including neural network model predictive control, model reference adaptive control, and a feedback linearization controller.

Chapter 11, “Applications” describes other neural network applications.

Business Applications

The *1988 DARPA Neural Network Study* [DARP88] lists various neural network applications, beginning in about 1984 with the adaptive channel equalizer. This device, which is an outstanding commercial success, is a single-neuron network used in long-distance telephone systems to stabilize voice signals. The DARPA report goes on to list other commercial applications, including a small word recognizer, a process monitor, a sonar classifier, and a risk analysis system.

Neural networks have been applied in many other fields since the DARPA report was written, as described in the next table.

Industry	Business Applications
Aerospace	High-performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, and aircraft component fault detection
Automotive	Automobile automatic guidance system, and warranty activity analysis
Banking	Check and other document reading and credit application evaluation

Industry	Business Applications
Defense	Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, and signal/image identification
Electronics	Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, and nonlinear modeling
Entertainment	Animation, special effects, and market forecasting
Financial	Real estate appraisal, loan advising, mortgage screening, corporate bond rating, credit-line use analysis, credit card activity tracking, portfolio trading program, corporate financial analysis, and currency price prediction
Industrial	Prediction of industrial processes, such as the output gases of furnaces, replacing complex and costly equipment used for this purpose in the past
Insurance	Policy application evaluation and product optimization
Manufacturing	Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, and dynamic modeling of chemical process system

Industry	Business Applications
Medical	Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, and emergency-room test advisement
Oil and gas	Exploration
Robotics	Trajectory control, forklift robot, manipulator controllers, and vision systems
Speech	Speech recognition, speech compression, vowel classification, and text-to-speech synthesis
Securities	Market analysis, automatic bond rating, and stock trading advisory systems
Telecommunications	Image and data compression, automated information services, real-time translation of spoken language, and customer payment processing systems
Transportation	Truck brake diagnosis systems, vehicle scheduling, and routing systems

Fitting a Function

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

Suppose, for instance, that you have data from a housing application [HaRu78]. You want to design a network that can predict the value of a house (in $1000s), given 13 pieces of geographical and real estate information. You have a total of 506 example homes for which you have those 13 items of data and their associated market values.

You can solve this problem in three ways:

- Use a command-line function, as described in "Using Command-Line Functions" on page 1-7.
- Use a graphical user interface, `nftool`, as described in "Using the Neural Network Fitting Tool GUI" on page 1-13.
- Use `nntool`, as described in "Graphical User Interface" on page 3-23.

Defining a Problem

To define a fitting problem for the toolbox, arrange a set of Q input vectors as columns in a matrix. Then, arrange another set of Q target vectors (the correct output vectors for each of the input vectors) into a second matrix. For example, you can define the fitting problem for a Boolean AND gate with four sets of two-element input vectors and one-element targets as follows:

```
inputs = [0 1 0 1; 0 0 1 1];
targets = [0 0 0 1];
```

The next section demonstrates how to train a network from the command line, after you have defined the problem. This example uses the housing data set provided with the toolbox.

Using Command-Line Functions

1 Load the data, consisting of input vectors and target vectors, as follows:

```
load house_dataset
```

2 Create a network. For this example, you use a feed-forward network with the default tan-sigmoid transfer function in the hidden layer and linear

transfer function in the output layer. This structure is useful for function approximation (or regression) problems. Use 20 neurons (somewhat arbitrary) in one hidden layer. The network has one output neuron, because there is only one target value associated with each input vector.

```
net = newfit(houseInputs,houseTargets,20);
```

Note More neurons require more computation, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently.

---

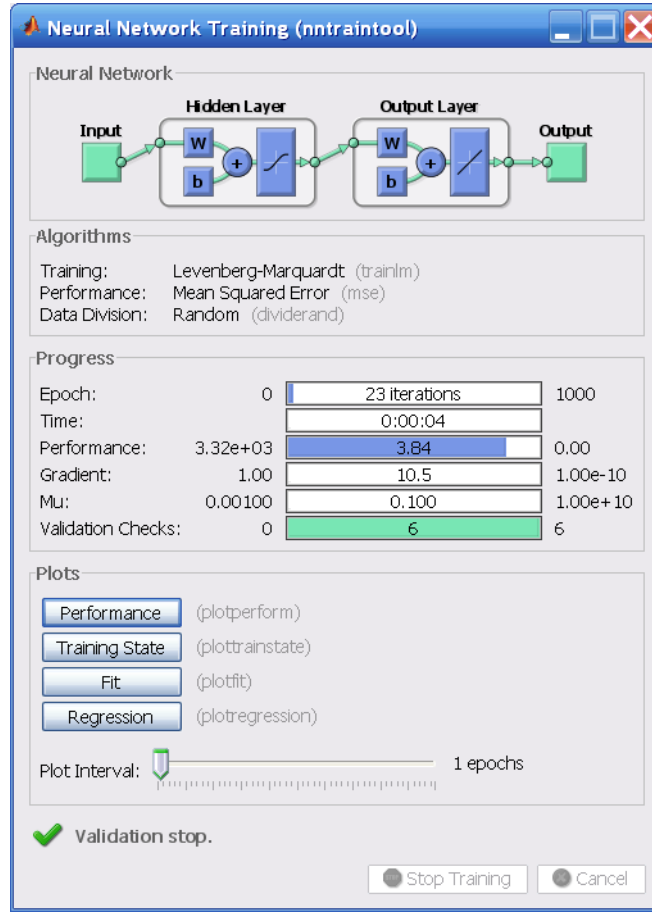
3 Train the network. The network uses the default Levenberg-Marquardt algorithm for training. The application randomly divides input vectors and target vectors into three sets as follows:

- 60% are used for training.
- 20% are used to validate that the network is generalizing and to stop training before overfitting.
- The last 20% are used as a completely independent test of network generalization.

To train the network, enter:

```
net=train(net,houseInputs,houseTargets);
```

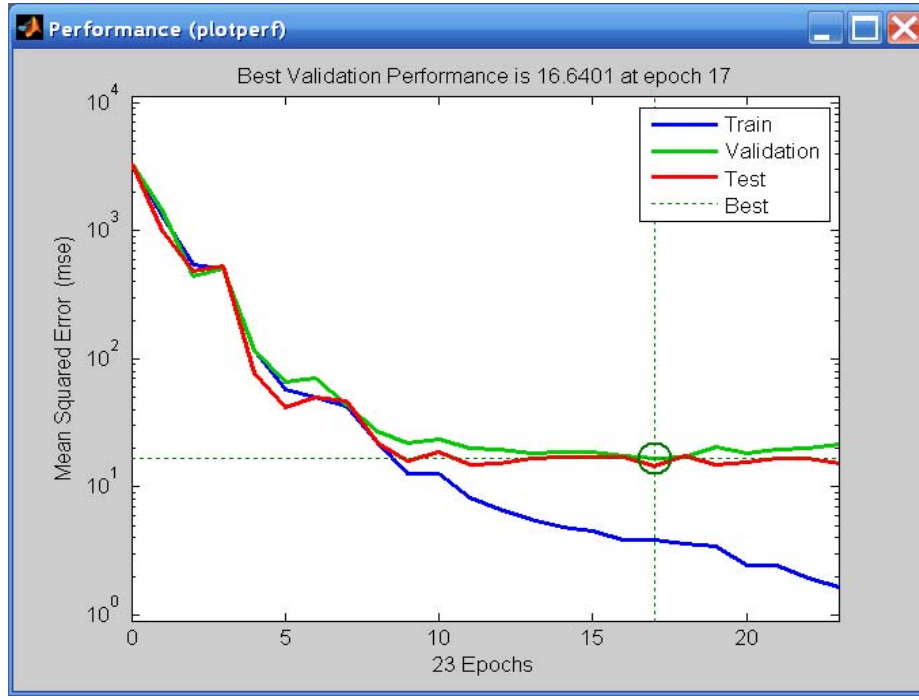
During training, the following training window opens. This window displays training progress and allows you to interrupt training at any point by clicking **Stop Training**.



This example used the `train` function. All the input vectors to the network appear at once in a batch. Alternatively, you can present the input vectors one at a time using the `adapt` function. “Training Styles” on page 2-20 describes the two training approaches.

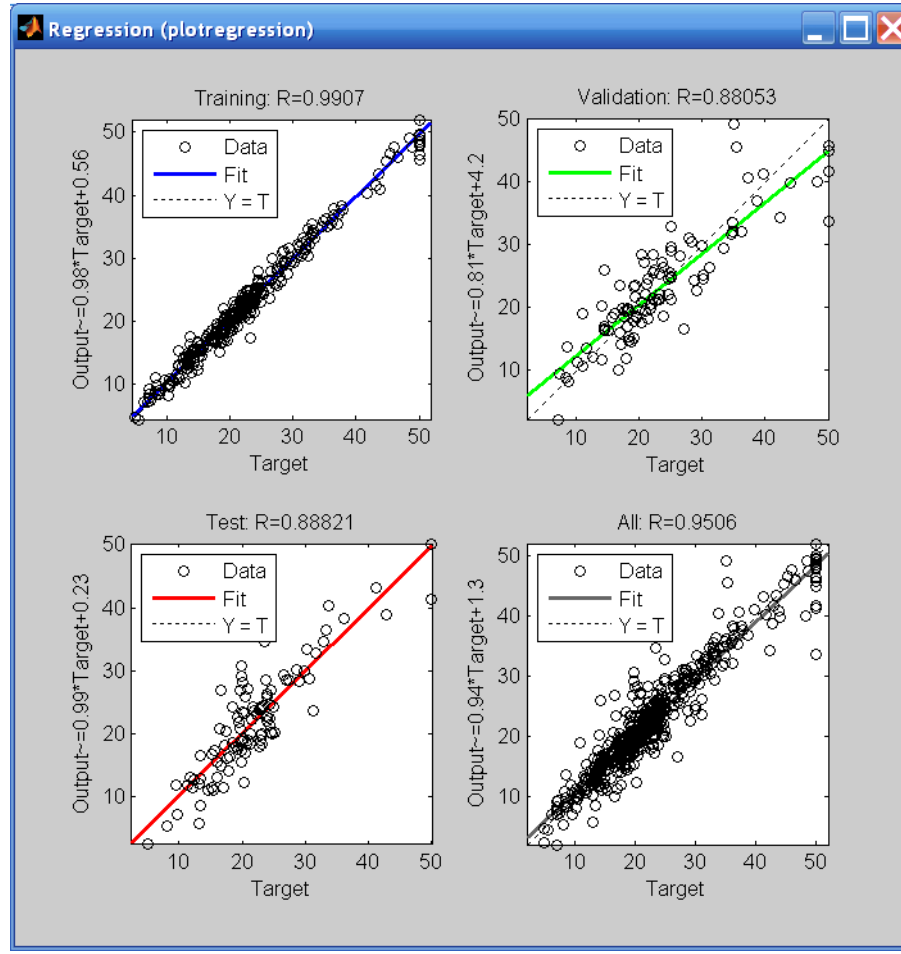
This training stopped when the validation error increased for six iterations, which occurred at iteration 23. If you click **Performance** in the training window, a plot of the training errors, validation errors, and test errors appears, as shown in the following figure. In this example, the result is reasonable because of the following considerations:

- The final mean-square error is small.
- The test set error and the validation set error have similar characteristics.
- No significant overfitting has occurred by iteration 17 (where the best validation performance occurs).



4 Perform some analysis of the network response. If you click **Regression** in the training window, you can perform a linear regression between the network outputs and the corresponding targets.

The following figure shows the results.



The output tracks the targets very well for training, testing, and validation, and the R-value is over 0.95 for the total response. If even more accurate results were required, you could try any of these approaches:

- Reset the initial network weights and biases to new values with `init` and train again.
- Increase the number of hidden neurons.
- Increase the number of training vectors.

- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Speed and Memory Comparison” on page 5-34).

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

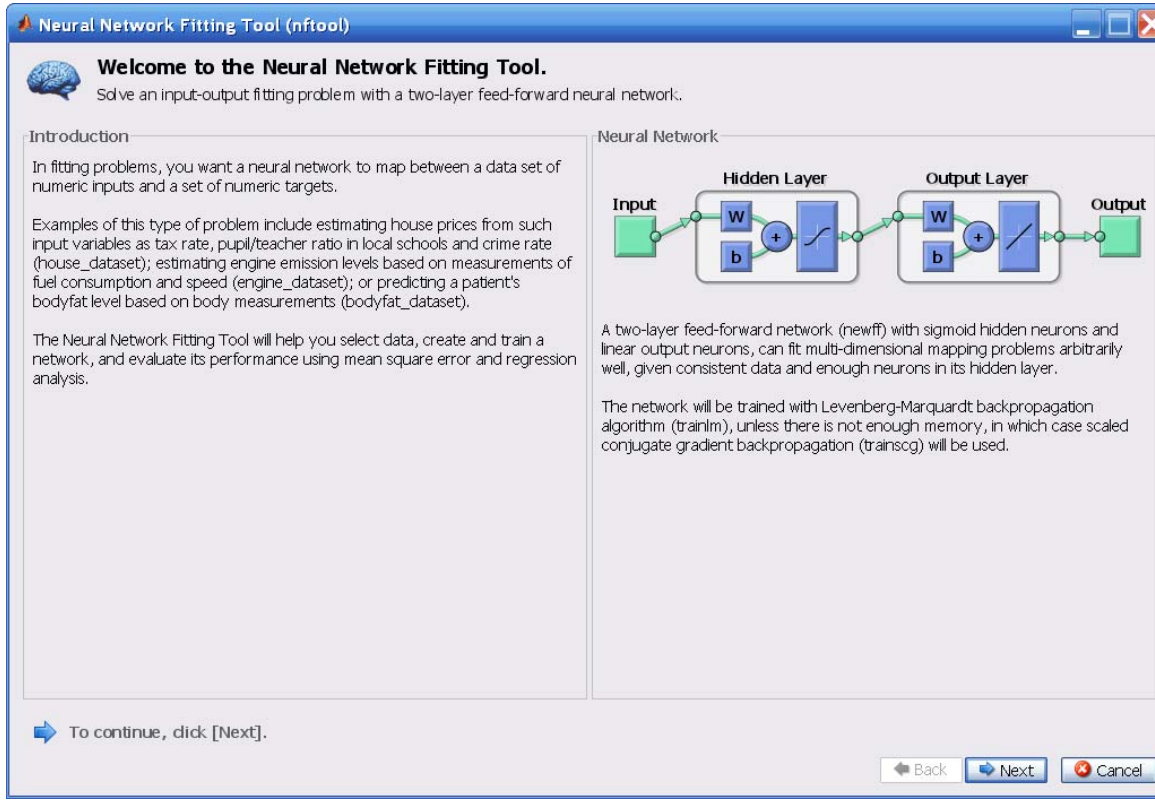
To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the regression plot), and watch it animate.
- Plot from the command line with functions such as `plotfit`, `plotregression`, `plottrainstate` and `plotperform`. (For more information on using these functions, see their reference pages.)

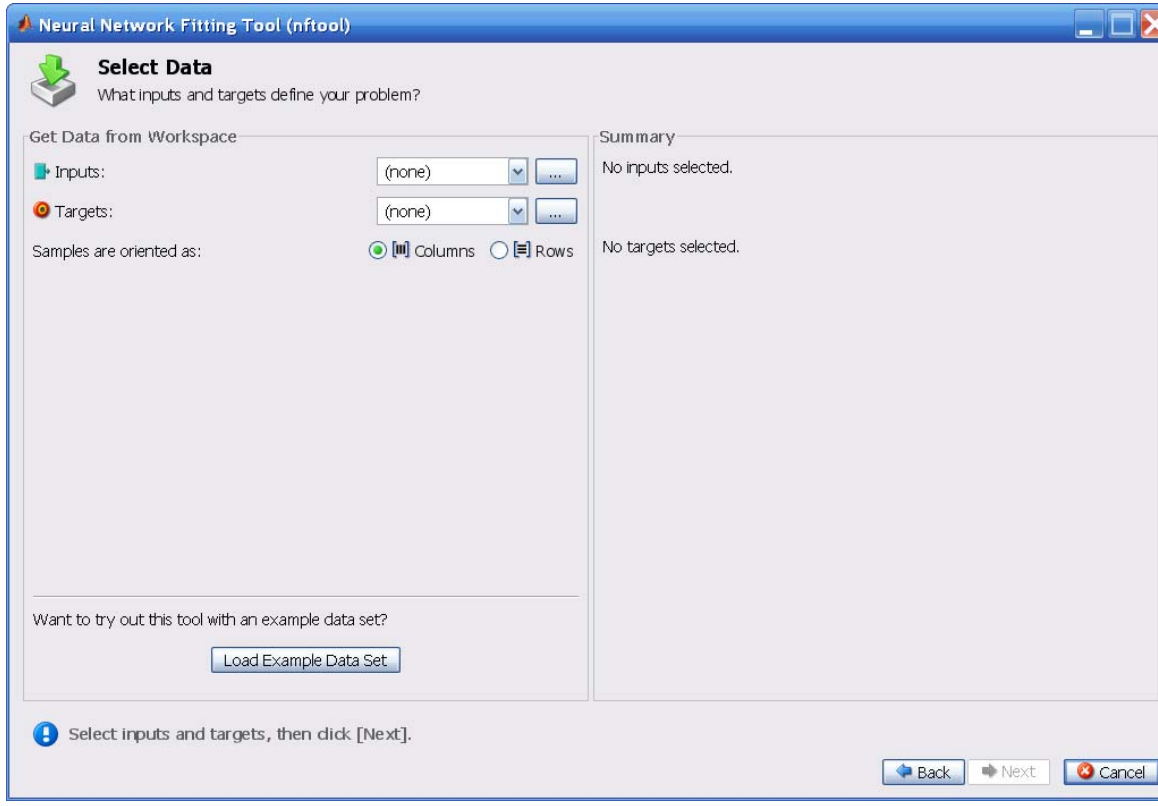
Using the Neural Network Fitting Tool GUI

1 Open the Neural Network Fitting Tool with this command:

```
nftool
```

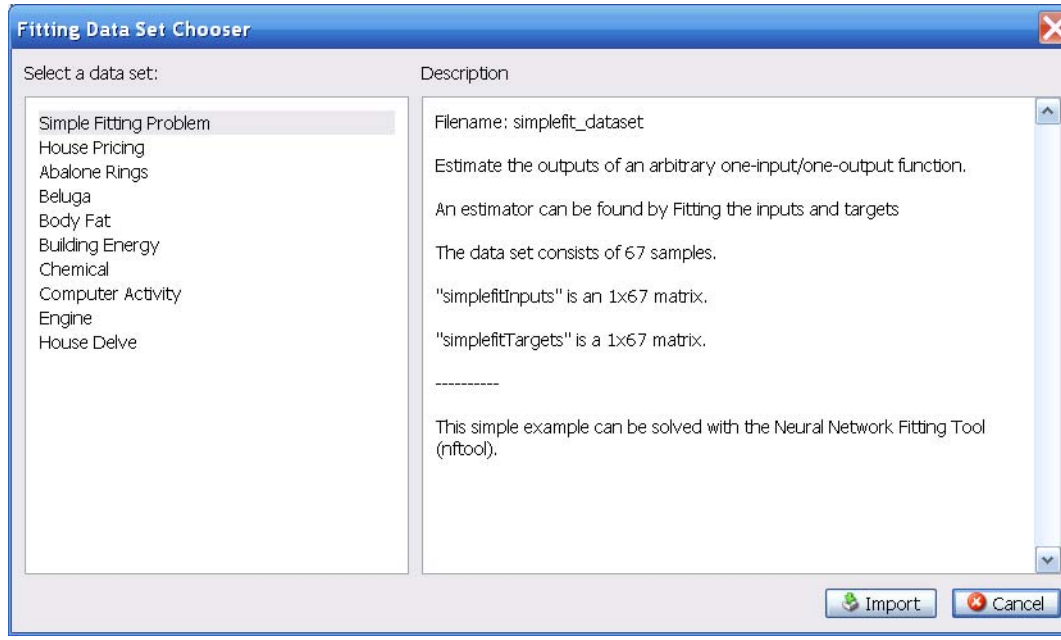


2 Click **Next** to proceed.



3 Click **Load Example Data Set** in the Select Data window. The Fitting Data Set Chooser window opens.

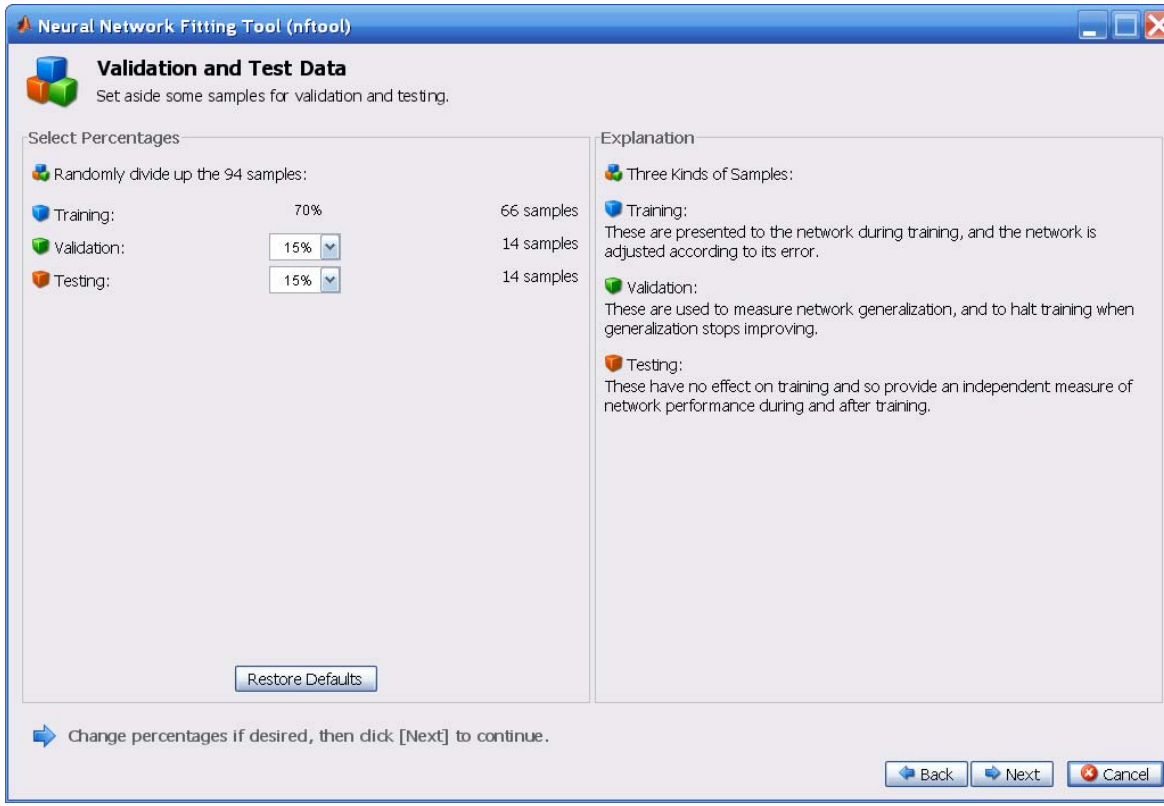
Note You use the **Inputs** and **Targets** options in the Select Data window when you need to load data from the MATLAB® workspace.



4 Select **Simple Fitting Problem**, and click **Import**. This brings you back to the Select Data window.

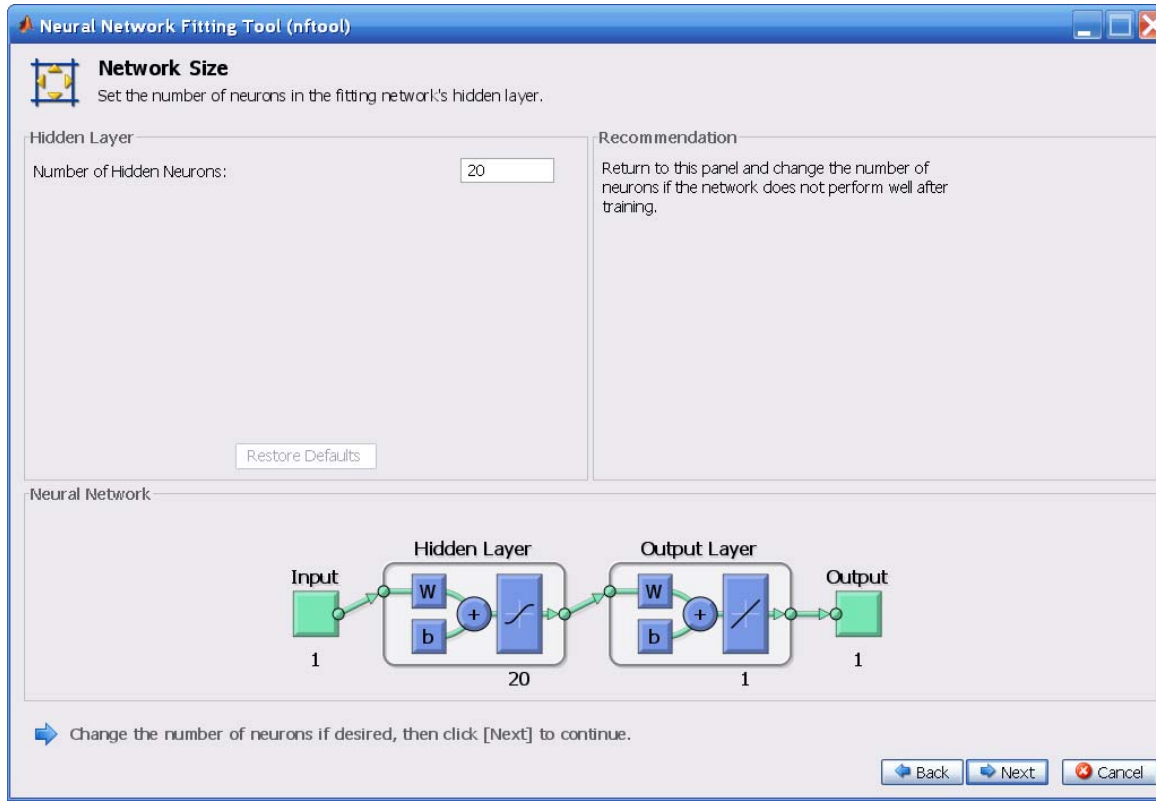
5 Click **Next** to display the Validate and Test Data window, shown in the following figure.

The validation and test data sets are each set to 15% of the original data.

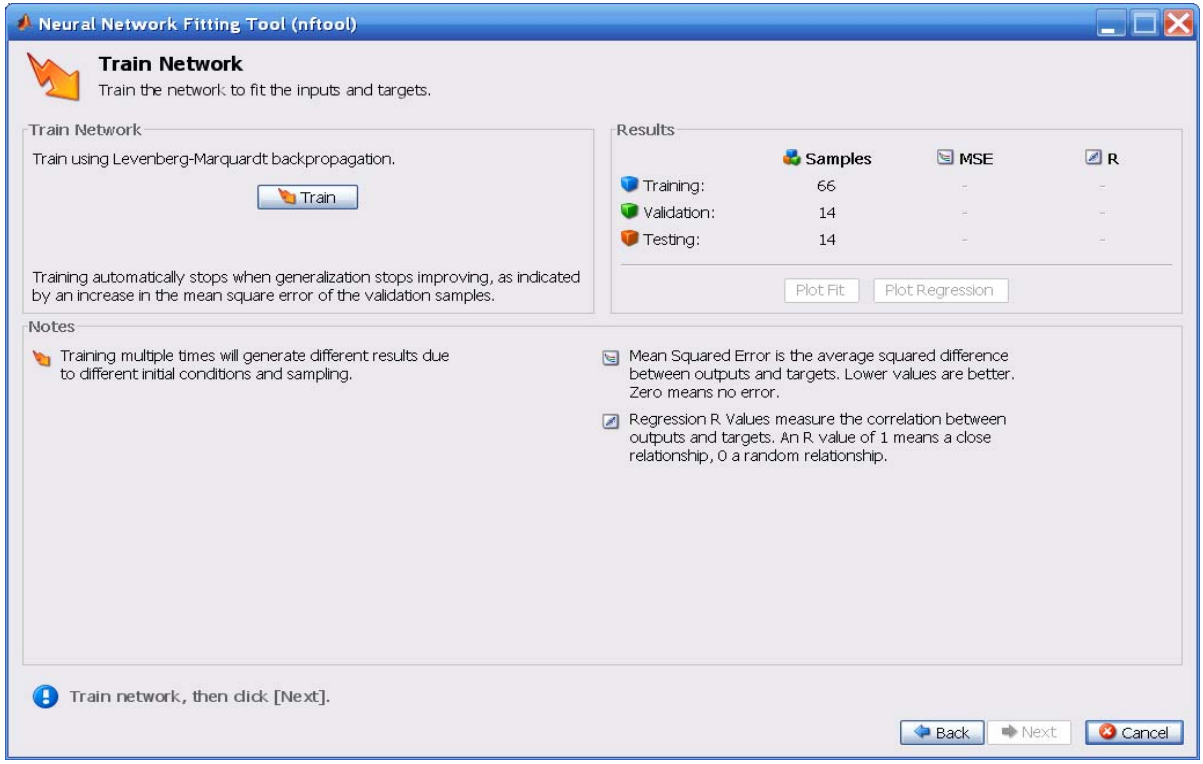


6 Click **Next**.

The number of hidden neurons is set to 20. You can change this value in another run if you want. You might want to change this number if the network does not perform as well as you expect.

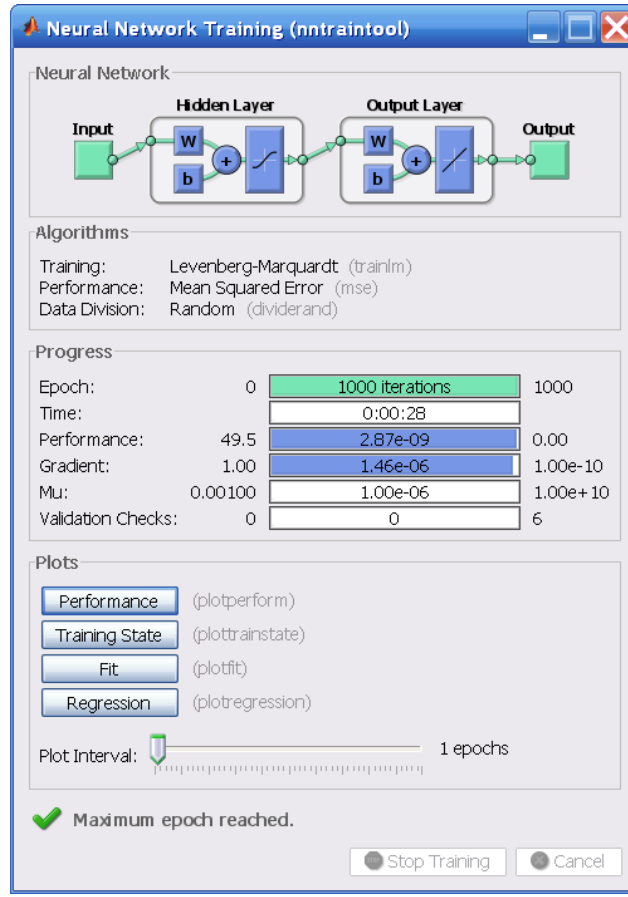


7 Click **Next**.


Neural Network Fitting Tool (nftool)
Train Network
Train the network to fit the inputs and targets.
Train Network
Train using Levenberg-Marquardt backpropagation.
Train
Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.
Results
Samples
MSE
R
Training: 66
Validation: 14
Testing: 14
Plot Fit
Plot Regression
Notes
Training multiple times will generate different results due to different initial conditions and sampling.
Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.
Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.
Train network, then click [Next].
Back
Next
Cancel

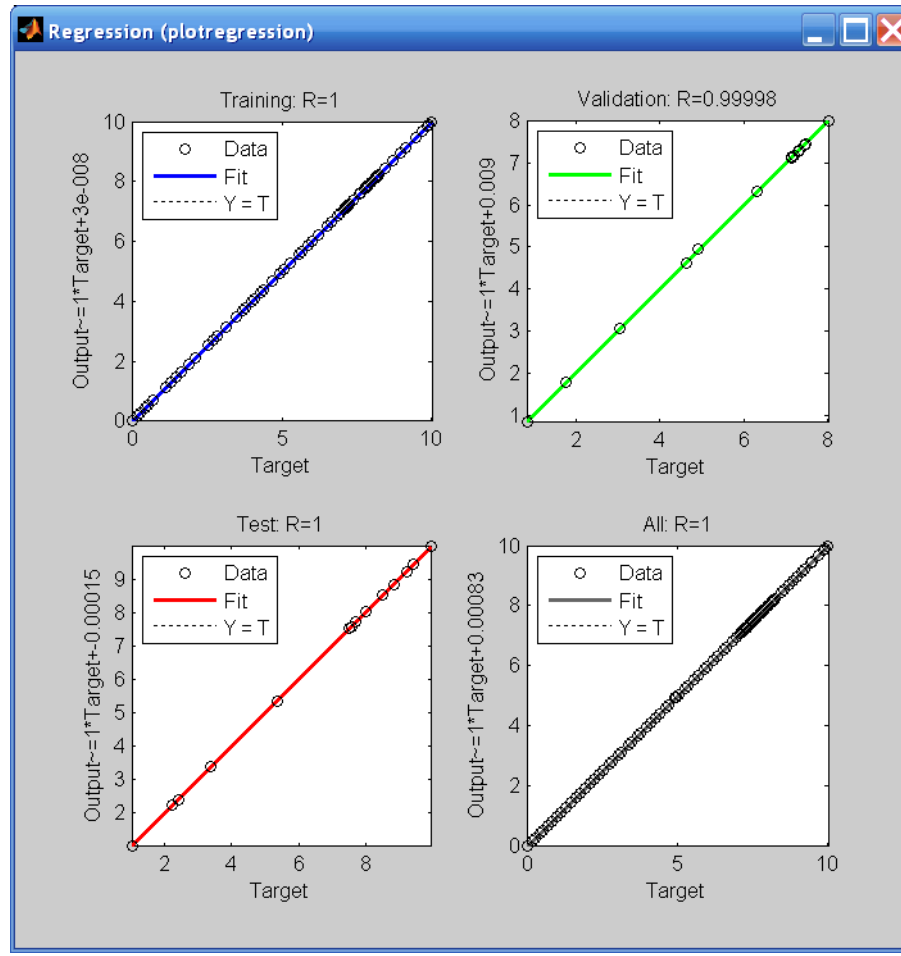
8 Click **Train.**

This time the training continued for the maximum of 1000 iterations.



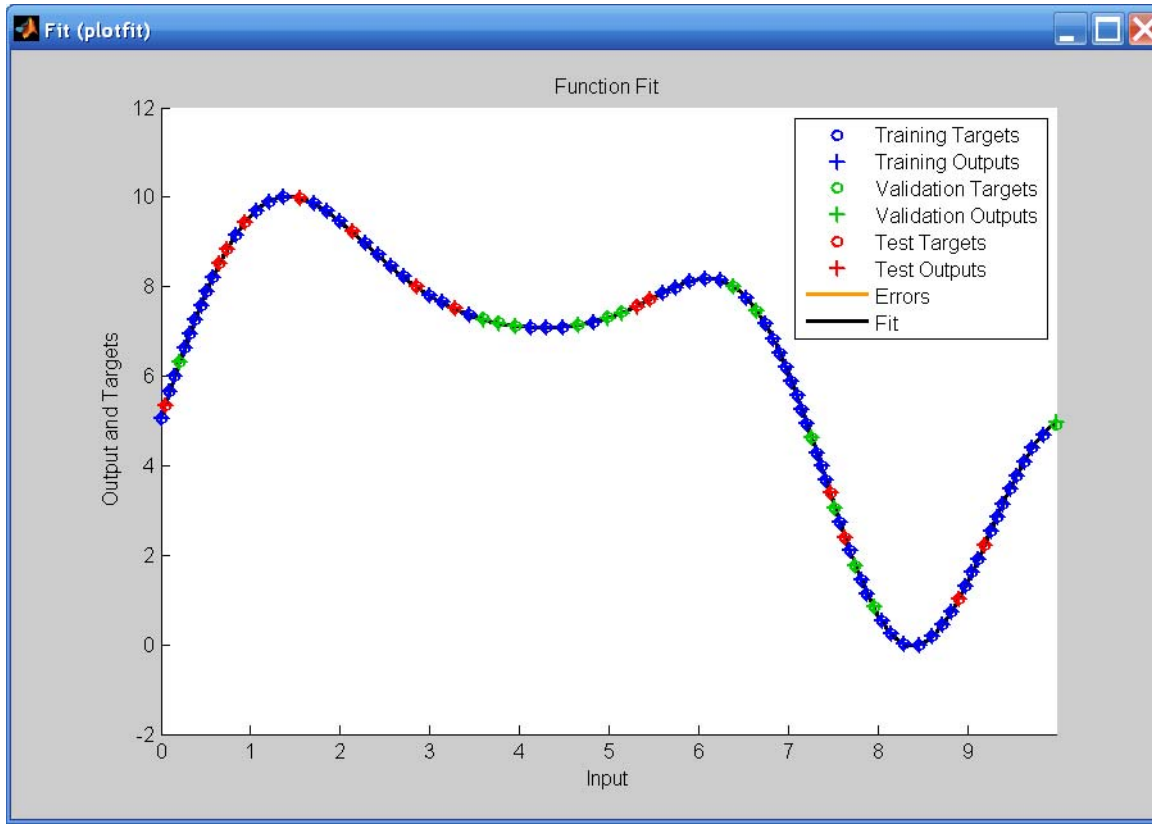
9 Under **Plots**, click **Regression**.

For this simple fitting problem, the fit is almost perfect for training, testing, and validation data.



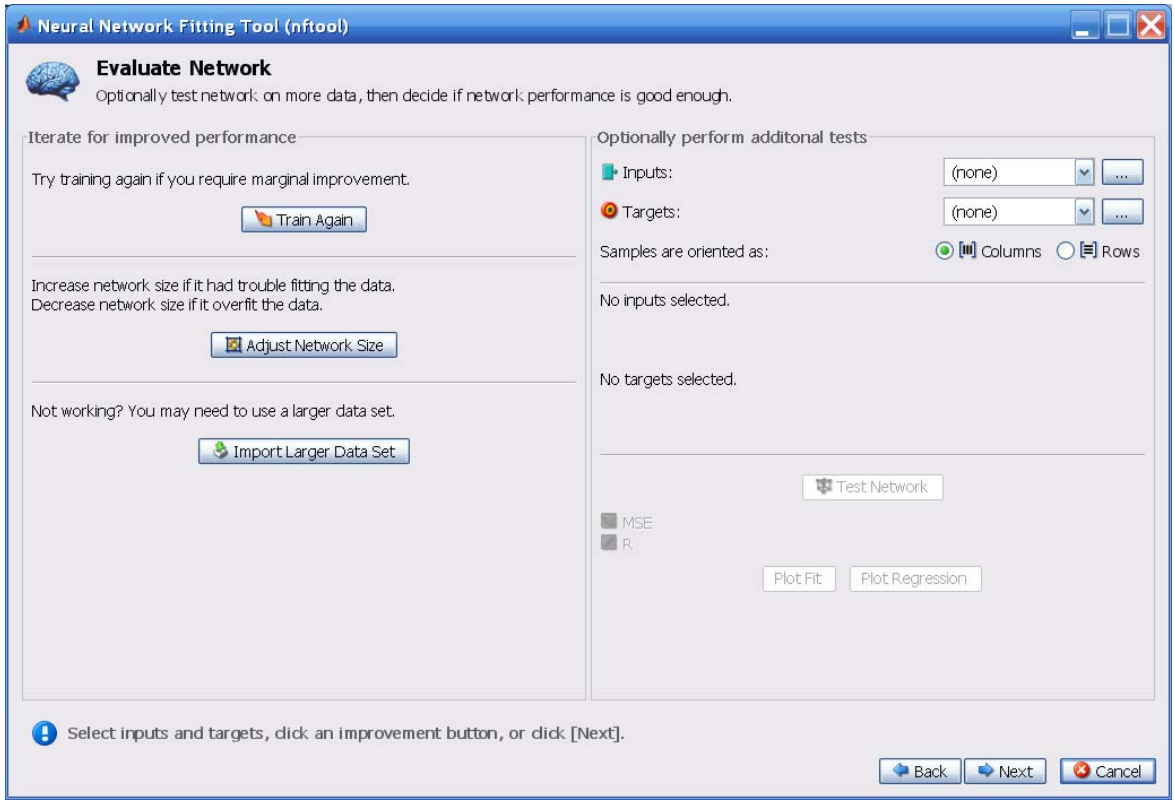
These plots are the regression plots for the output with respect to training, validation, and test data.

10 View the network response. For single-input/single-output problems, like this simple fitting problem, under the **Plots** pane, click **Fit**.



The blue symbols represent training data, the green symbols represent validation data, and the red symbols represent testing data. For this problem and this network, the network outputs match the targets for all three data sets.

11 Click **Next** in the Neural Network Fitting Tool to evaluate the network.



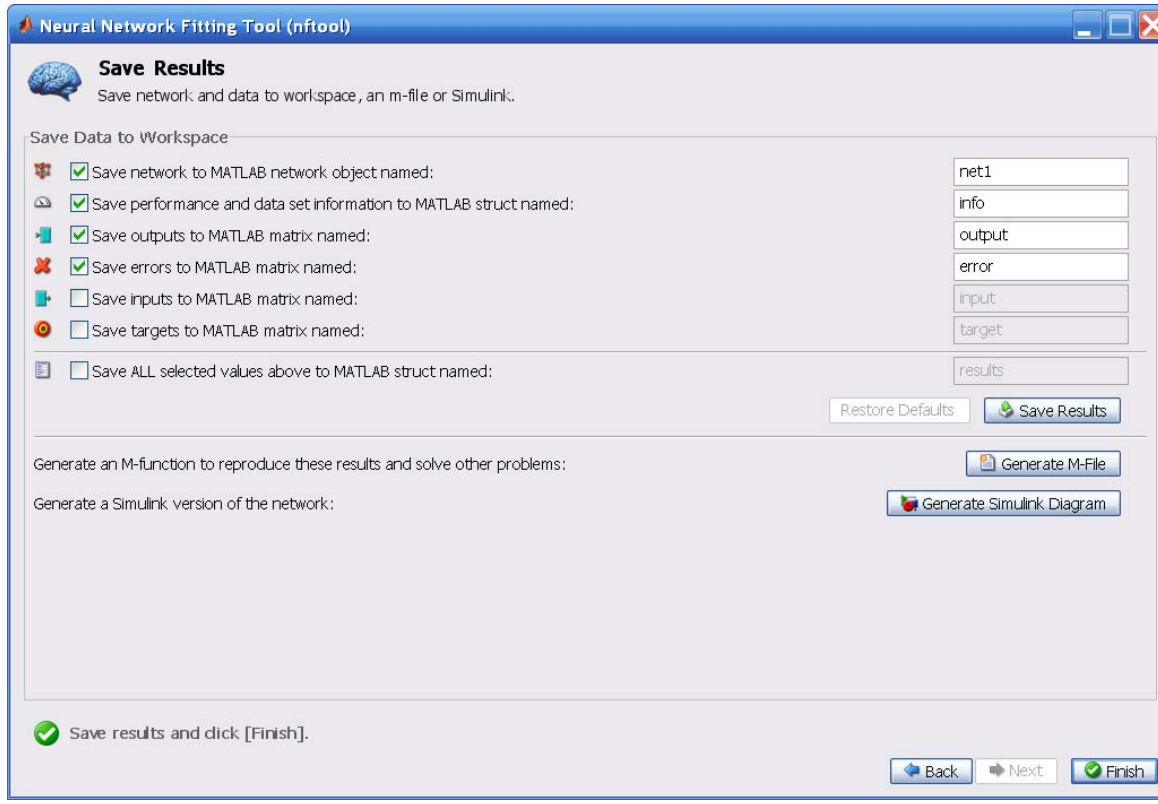
At this point, you can test the network against new data.

If you are dissatisfied with the network's performance on the original or new data, you can take any of the following steps:

- Train it again.
- Increase the number of neurons.
- Get a larger training data set.

12 If you are satisfied with the network performance, click **Next**.

13 Use the buttons on this screen to save your results.



- You have the network saved as `net1` in the workspace. You can perform additional tests on it or put it to work on new inputs, using the `sim` function.
- You can also click **Generate M-File** to create an M-file that can be used to reproduce all of the previous steps from the command line. Creating an M-file can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

14 When you have saved your results, click **Finish**.

Recognizing Patterns

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

For example, suppose you want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis, etc. [MuAh94]. You have 699 example cases for which you have 9 items of data and the correct classification as benign or malignant.

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

- Use a command-line solution, as described in "Using Command-Line Functions" on page 1-43.
- Use the `nprtool` GUI, as described in "Using the Neural Network Clustering Tool GUI" on page 1-47.
- Use `nntool`, as described in "Graphical User Interface" on page 3-23.

Defining a Problem

To define a pattern recognition problem, arrange a set of Q input vectors as columns in a matrix. Then arrange another set of Q target vectors so that they indicate the classes to which the input vectors are assigned. There are two approaches to creating the target vectors.

One approach can be used when there are only two classes; you set each scalar target value to either 1 or 0, indicating which class the corresponding input belongs to. For instance, you can define the exclusive-or classification problem as follows:

```
inputs = [0 1 0 1; 0 0 1 1];
targets = [0 1 0 1];
```

Alternately, target vectors can have *N* elements, where for each target vector, one element is 1 and the others are 0. This defines a problem where inputs are to be classified into *N* different classes. For example, the following lines show how to define a classification problem that divides the corners of a 5-by-5-by-5 cube into three classes:

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

```
inputs = [0 0 0 0 5 5 5 5; 0 0 5 5 0 0 5 5; 0 5 0 5 0 5 0 5];
targets = [1 0 0 0 0 0 0 0; 0 1 1 1 1 1 1 0; 0 0 0 0 0 0 0 1];
```

Classification problems involving only two classes can be represented using either format. The targets can consist of either scalar 1/0 elements or two-element vectors, with one element being 1 and the other element being 0.

The next section demonstrates how to train a network from the command line, after you have defined the problem.

Using Command-Line Functions

1 Use the cancer data set as an example. This data set consists of 699 nine-element input vectors and two-element target vectors.

Load the tumor classification data as follows:

```
load cancer_dataset
```

2 Create a network. For this example, you use a pattern recognition network, which is a feed-forward network with tan-sigmoid transfer functions in both the hidden layer and the output layer. As in the function-fitting example, use 20 neurons in one hidden layer:

- The network has two output neurons, because there are two categories associated with each input vector.
- Each output neuron represents a category.
- When an input vector of the appropriate category is applied to the network, the corresponding neuron should produce a 1, and the other neurons should output a 0.

To create a network, enter this command:

```
net = newpr(cancerInputs,cancerTargets,20);
```

3 Train the network. The pattern recognition network uses the default Scaled Conjugate Gradient algorithm for training. The application randomly divides the input vectors and target vectors into three sets:

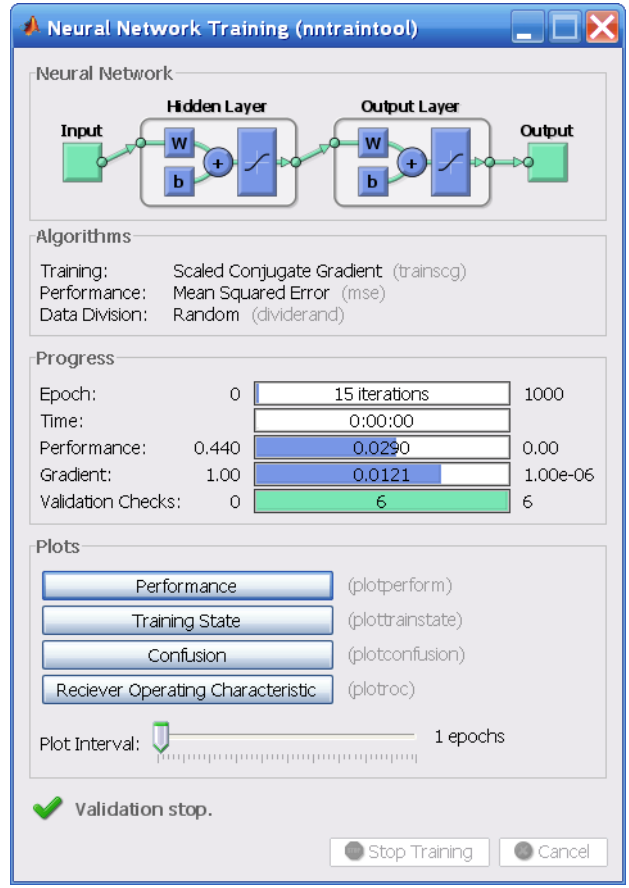
- 60% are used for training.
- 20% are used to validate that the network is generalizing and to stop training before overfitting.

- The last 20% are used as a completely independent test of network generalization.

To train the network, enter this command:

```
net=train(net,cancerInputs,cancerTargets);
```

During training, as in function fitting, the training window opens. This window displays training progress. To interrupt training at any point, click **Stop Training**.

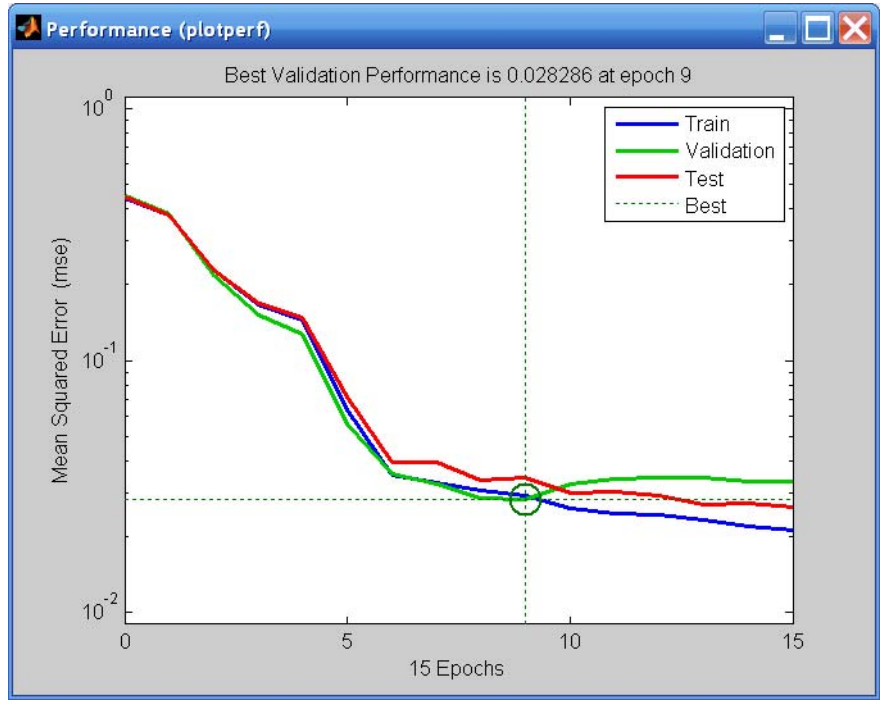


This example uses the `train` function. It presents all the input vectors to the network at once in a batch. Alternatively, you can present the input vectors one at a time using the `adapt` function. “Training Styles” on page 2-20 describes the two training approaches.

This training stopped when the validation error increased for six iterations, which occurred at iteration 15.

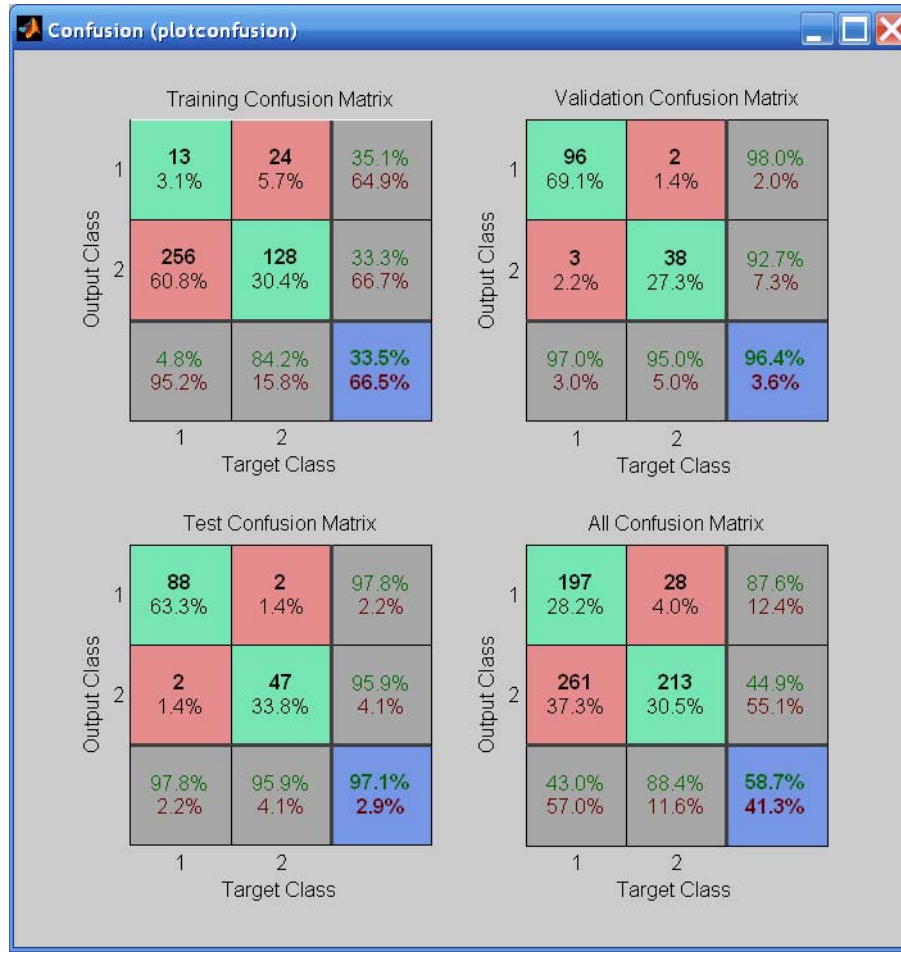
4 To find the validation error, click **Performance** in the training window. A plot of the training errors, validation errors, and test errors appears, as

shown in the following figure. The best validation performance occurred at iteration 9, and the network at this iteration is returned.



5 To analyze the network response, click **Confusion** in the training window. A display of the confusion matrix appears that shows various types of errors that occurred for the final trained network.

The next figure shows the results.



The diagonal cells in each table show the number of cases that were correctly classified, and the off-diagonal cells show the misclassified cases. The blue cell in the bottom right shows the total percent of correctly classified cases (in green) and the total percent of misclassified cases (in red). The results for all three data sets (training, validation, and testing) show very good recognition. If you needed even more accurate results, you could try any of the following approaches:

- Reset the initial network weights and biases to new values with `init` and train again.
- Increase the number of hidden neurons.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Speed and Memory Comparison” on page 5-34).

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

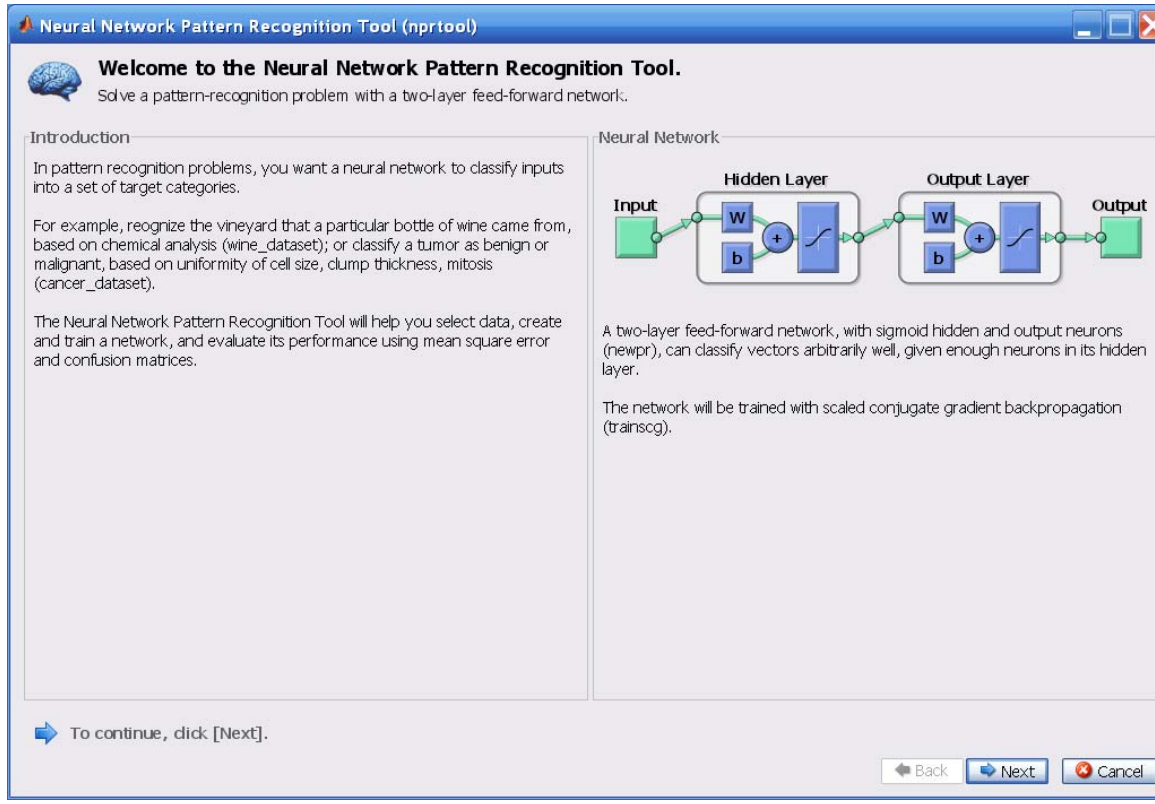
To get more experience in command-line operations, here are some tasks you can try:

- During training, open a plot window (such as the confusion plot), and watch it animate.
- Plot from the command line with functions such as `plotconfusion`, `plotroc`, `plottrainstate`, and `plotperform`. (For more information on using these functions, see their reference pages.)

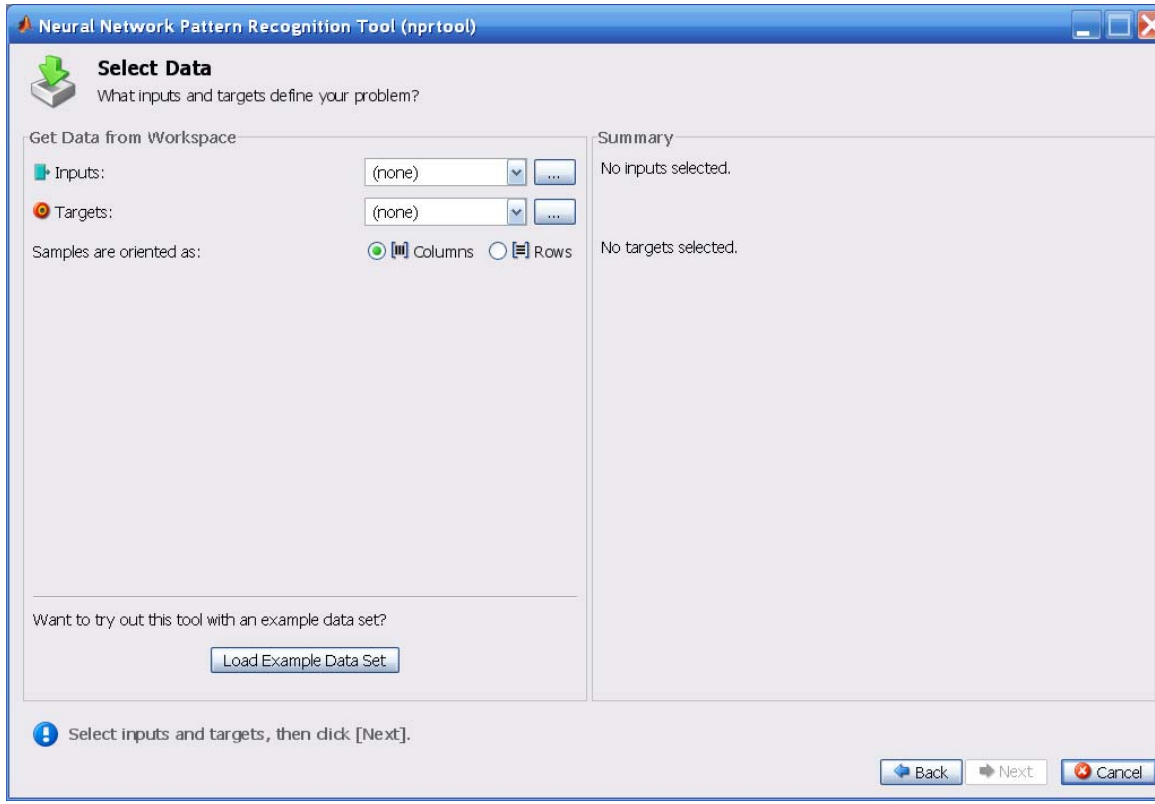
Using the Neural Network Pattern Recognition Tool GUI

1 Open the Neural Network Pattern Recognition Tool window with this command:

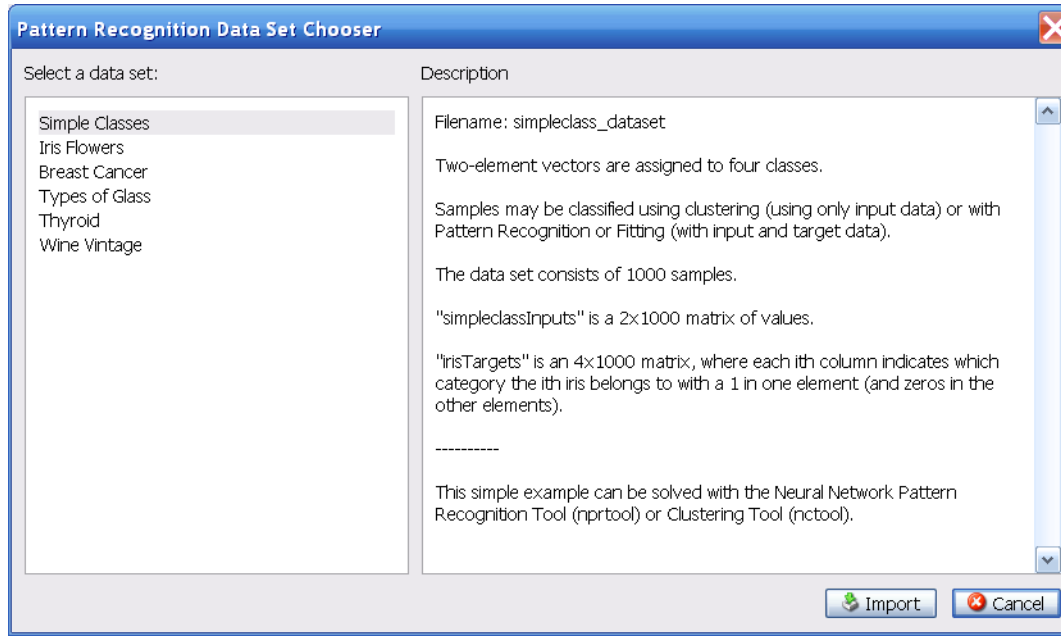
```
nprtool
```



2 Click **Next** to proceed. The Select Data window opens.



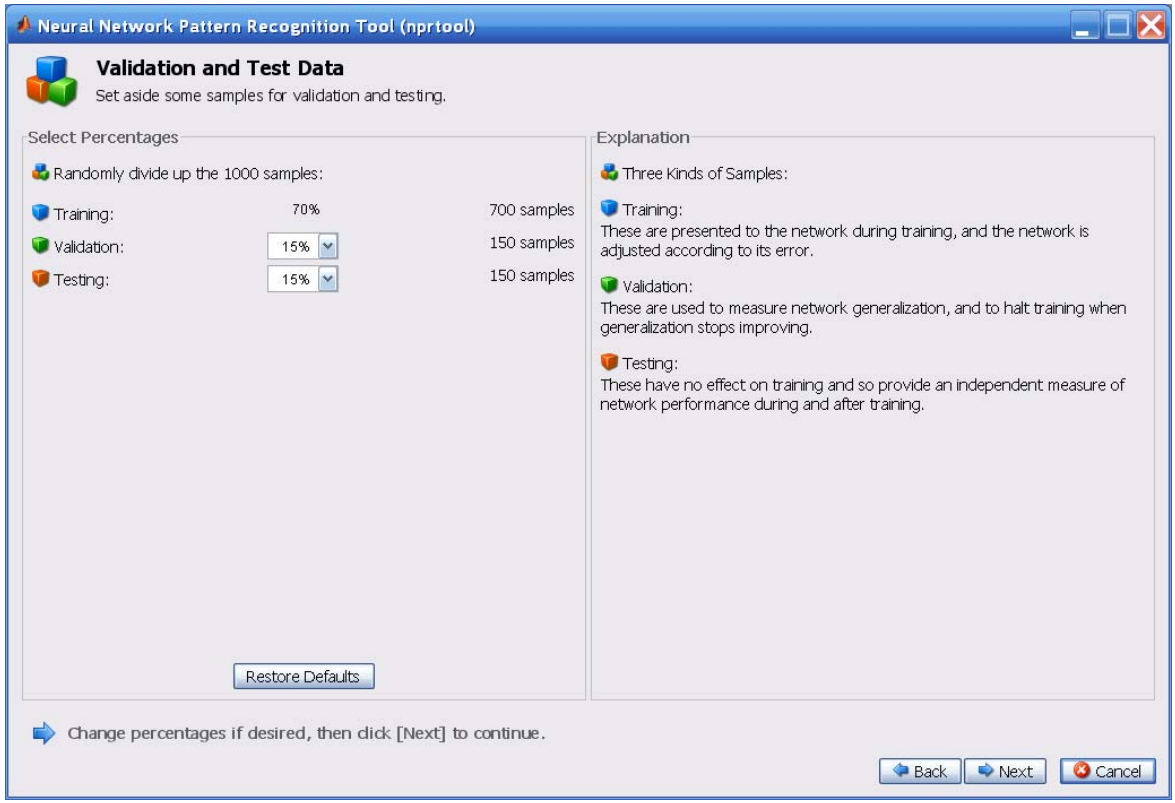
3 Click **Load Example Data Set**. The Pattern Recognition Data Set Chooser window opens.



4 In this window, select **Simple Classes**, and click **Import**. You return to the Select Data window.

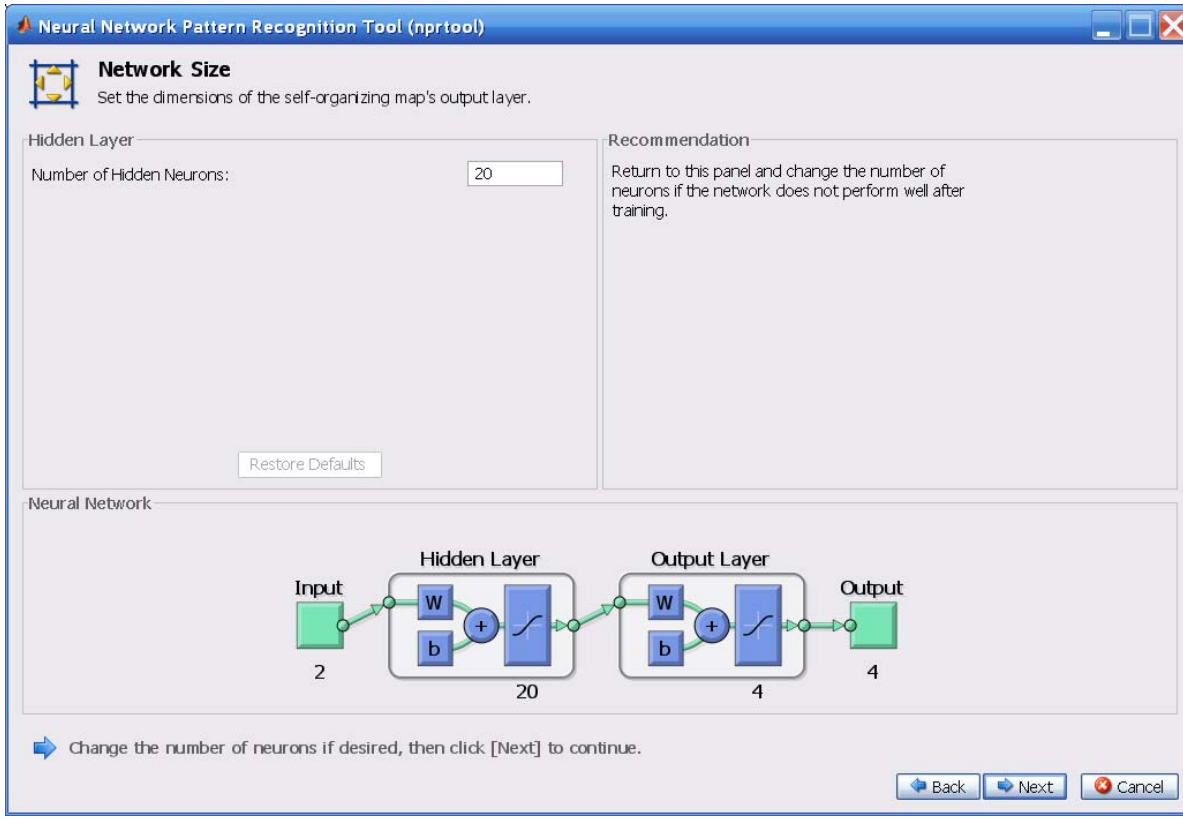
5 Click **Next** to continue to the Validate and Test Data window, shown in the following figure.

Validation and test data sets are each set to 15% of the original data.

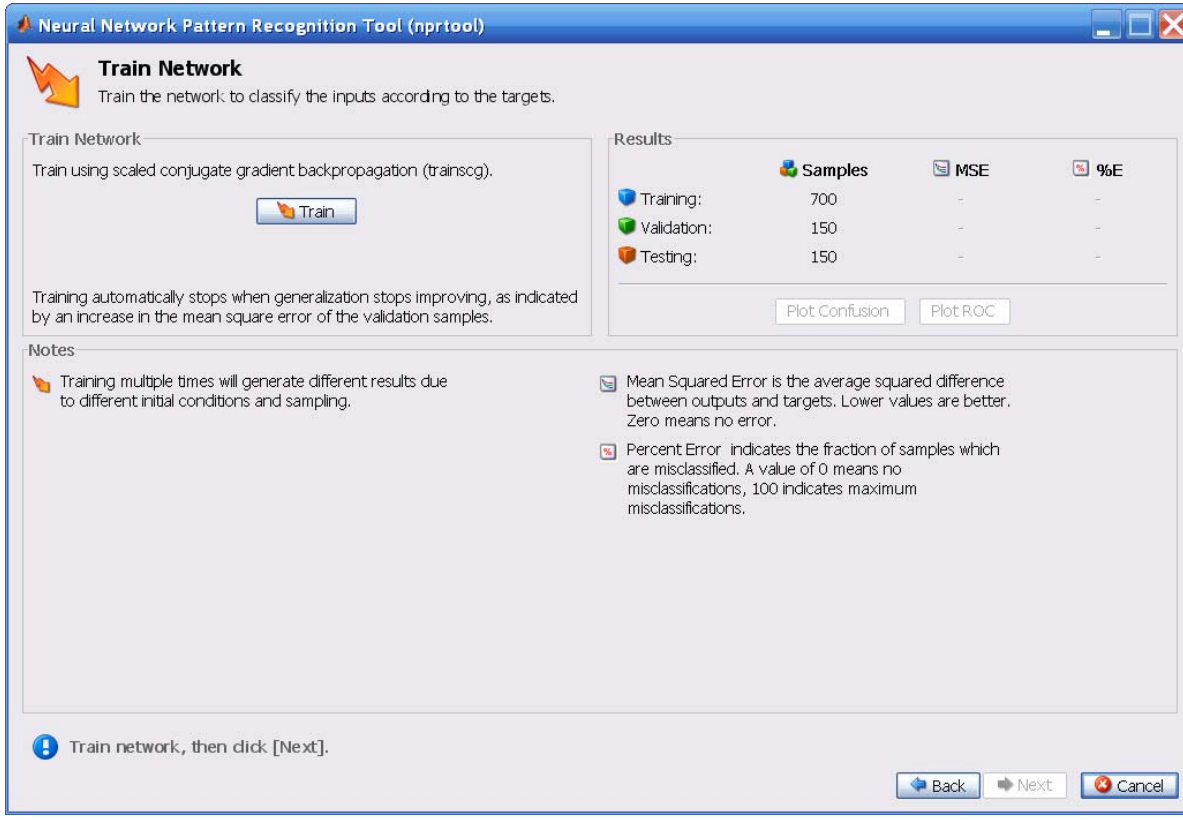


6 Click **Next**.

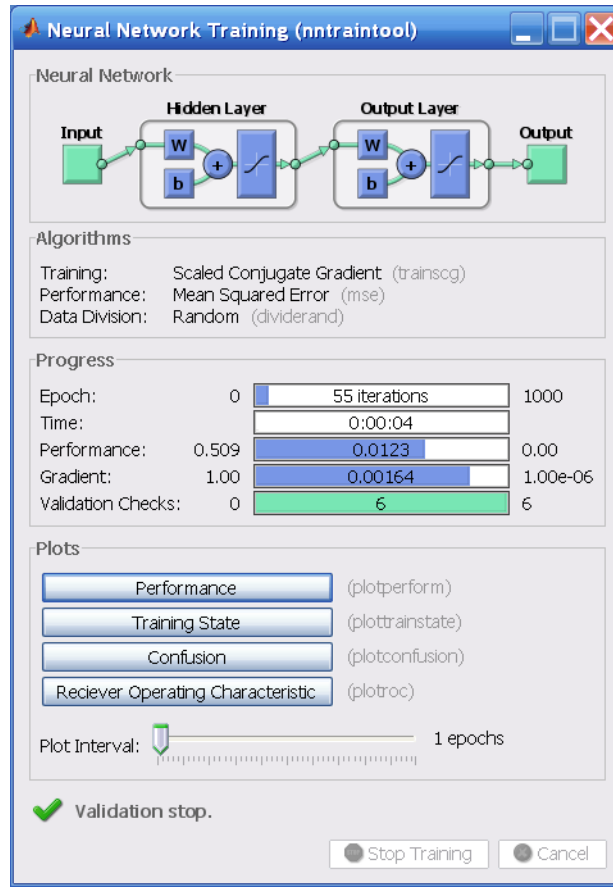
The number of hidden neurons is set to 20. You can change this in another run if you want. You might want to change this number if the network does not perform as well as you expect.



7 Click **Next**.


Neural Network Pattern Recognition Tool (nprtool)
Train Network
Train the network to classify the inputs according to the targets.
Train Network
Train using scaled conjugate gradient backpropagation (trainscg).
Train
Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.
Results
Samples
MSE
%E
Training:
700
Validation:
150
Testing:
150
Plot Confusion
Plot ROC
Notes
Training multiple times will generate different results due to different initial conditions and sampling.
Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.
Percent Error indicates the fraction of samples which are misclassified. A value of 0 means no misclassifications, 100 indicates maximum misclassifications.
Train network, then click [Next].
Back
Next
Cancel

8 Click **Train**.

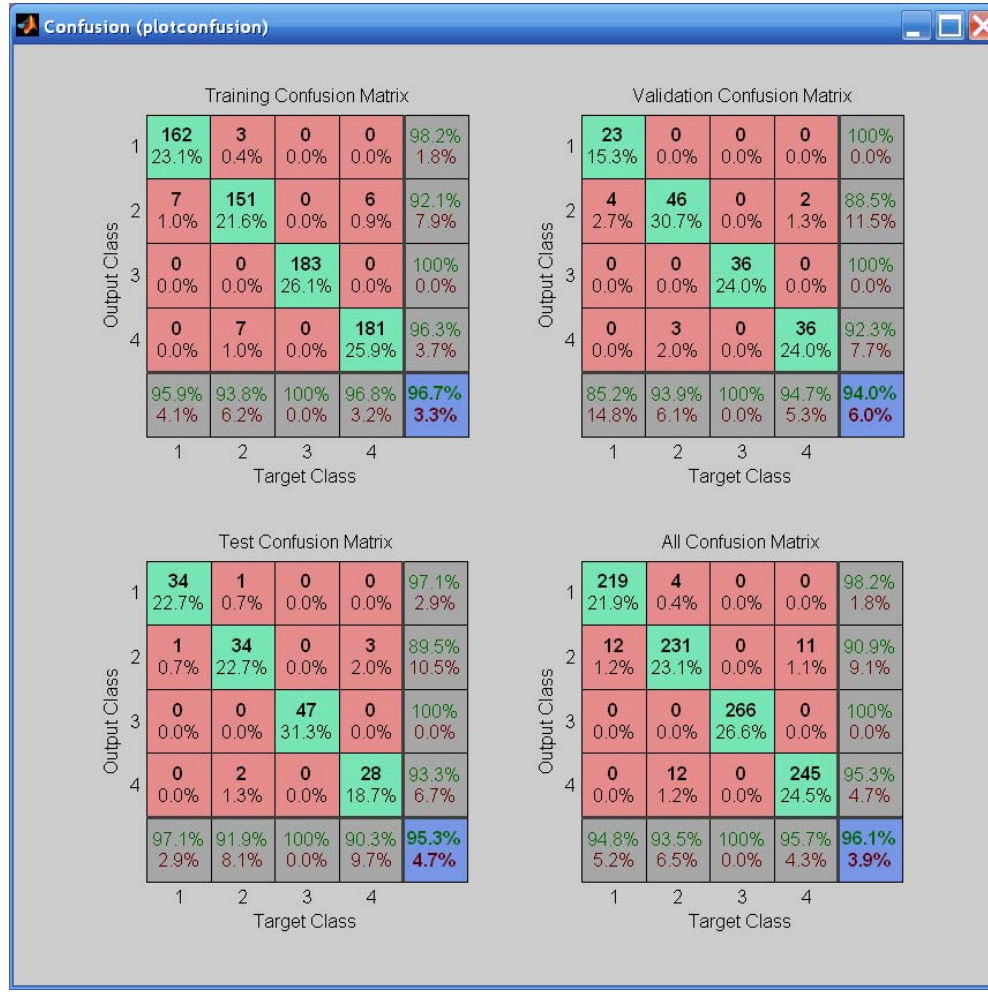


The training continues for 55 iterations.

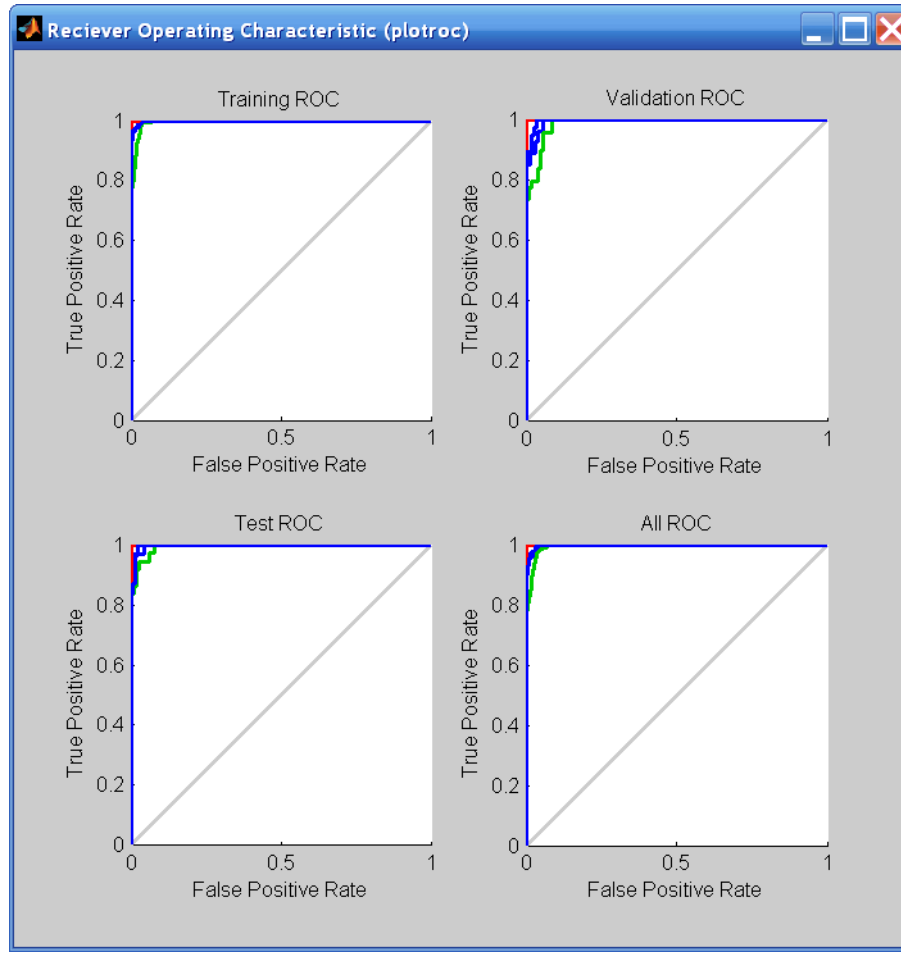
9 Under the **Plots** pane, click **Confusion** in the Neural Network Pattern Recognition Tool.

The next figure shows the confusion matrices for training, testing, and validation, and the three kinds of data combined. The network's outputs are almost perfect, as you can see by the high numbers of correct responses in

the green squares and the low numbers of incorrect responses in the red squares. The lower right blue squares illustrate the overall accuracies.

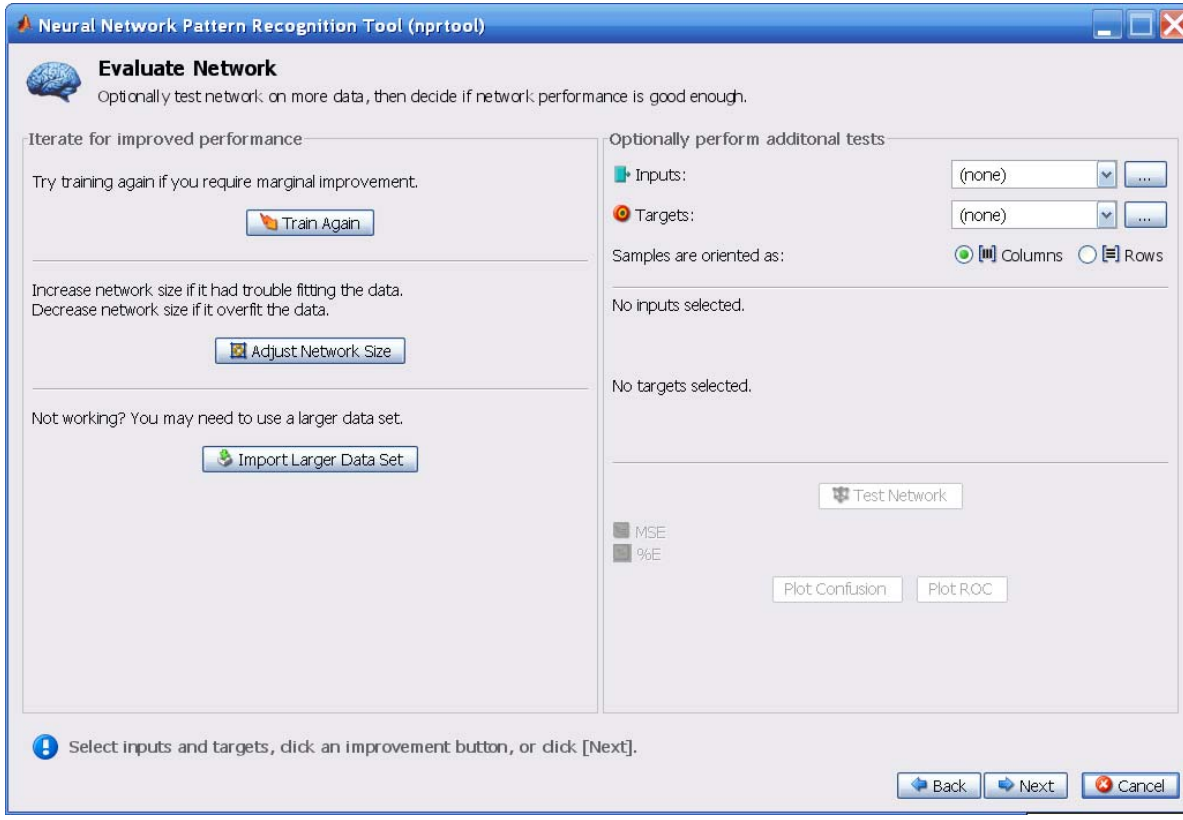


10 Plot the Receiver Operating Characteristic (ROC) curve. Under the **Plots** pane, click **Receiver Operating Characteristic** in the Neural Network Pattern Recognition Tool.



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

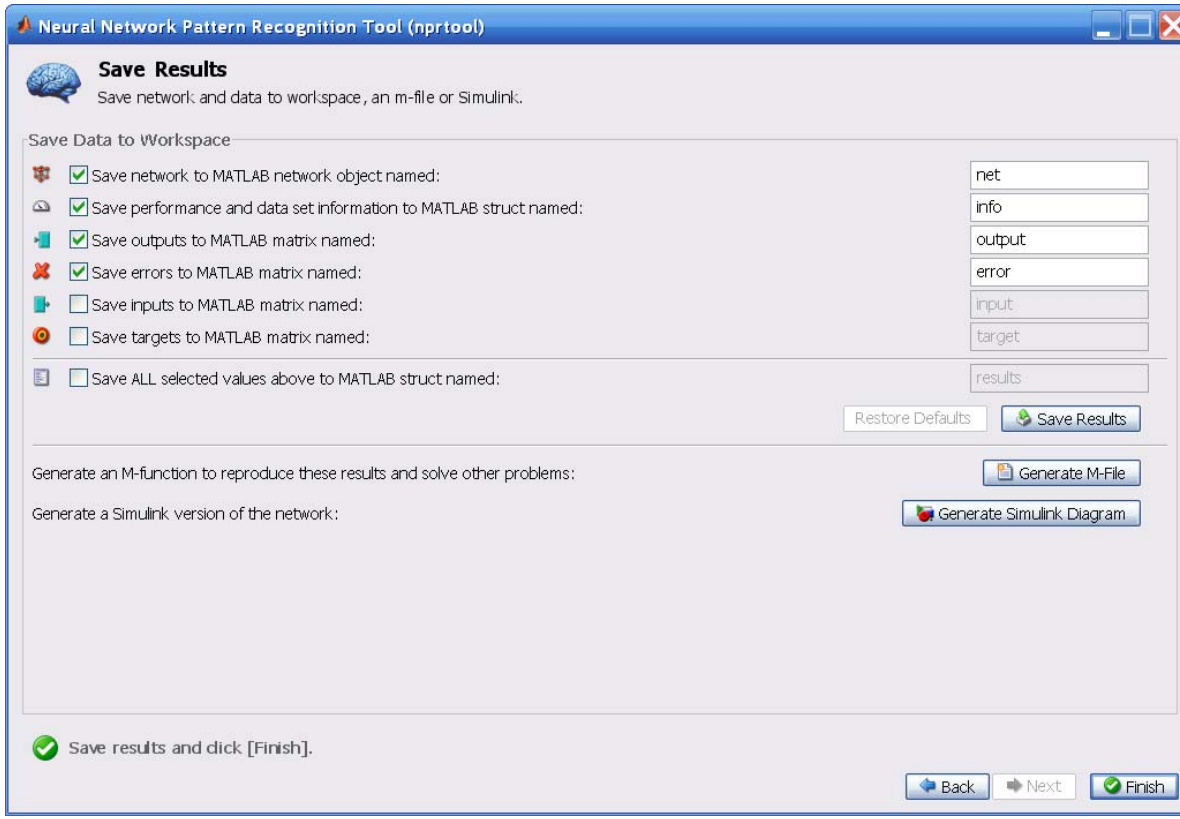
11 In the Neural Network Pattern Recognition Tool, click **Next** to evaluate the network.



At this point, you can test the network against new data.

If you are dissatisfied with the network's performance on the original or new data, you can train it again, increase the number of neurons, or perhaps get a larger training data set.

12 When you are satisfied with the network performance, click **Next**.



13 Use the buttons on this screen to save your results.

- You now have the network saved as `net1` in the workspace. You can perform additional tests on it or put it to work on new inputs using the `sim` function.
- If you click **Generate M-File**, the tool creates an M-file, with commands that recreate the steps that you have just performed from the command line. Generating an M-file is a good way to learn how to use the command-line operations of the Neural Network Toolbox™ software.

14 When you have saved your results, click **Finish**.

Clustering Data

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

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

Suppose that you want to cluster flower types according to petal length, petal width, sepal length, and sepal width [MuAh94]. You have 150 example cases for which you have these four measurements.

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

- Use a command-line solution, as described in "Using Command-Line Functions" on page 1-43.
- Use the nctool GUI, as described in "Using the Neural Network Clustering Tool GUI" on page 1-47.
- Use `nntool`, as described in "Graphical User Interface" on page 3-23.

Defining a Problem

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

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

The next section demonstrates how to train a network from the command line, after you have defined the problem.

Using Command-Line Functions

1. Use the flower data set as an example. The iris data set consists of 150 four-element input vectors.

   Load the data as follows:

   ```
   load iris_dataset
   ```

   This data set consists of input vectors and target vectors. However, you only need the input vectors for clustering.

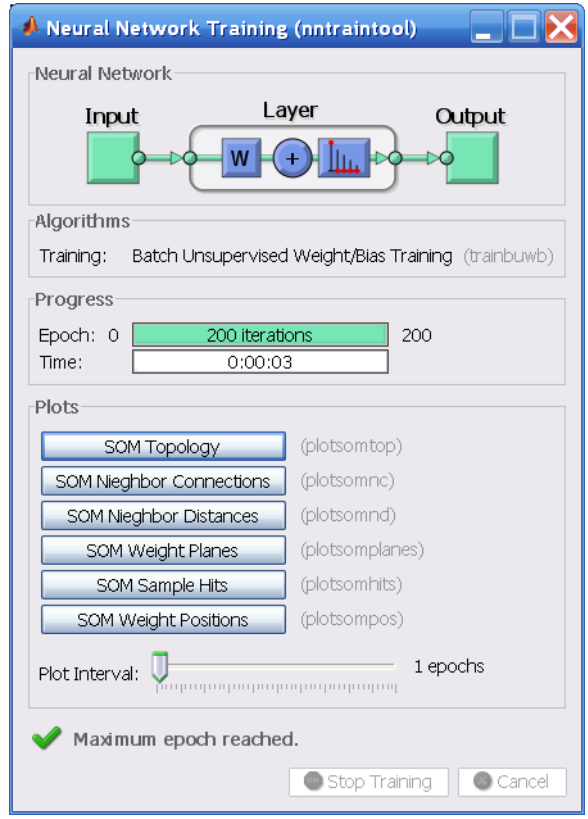
2. Create a network. For this example, you use a self-organizing map (SOM). This network has one layer, with the neurons organized in a grid. (For more information, see “Self-Organizing Feature Maps” on page 9-9.) When creating the network, you specify the number of rows and columns in the grid:

   ```
   net = newsom(irisInputs,[6,6]);
   ```

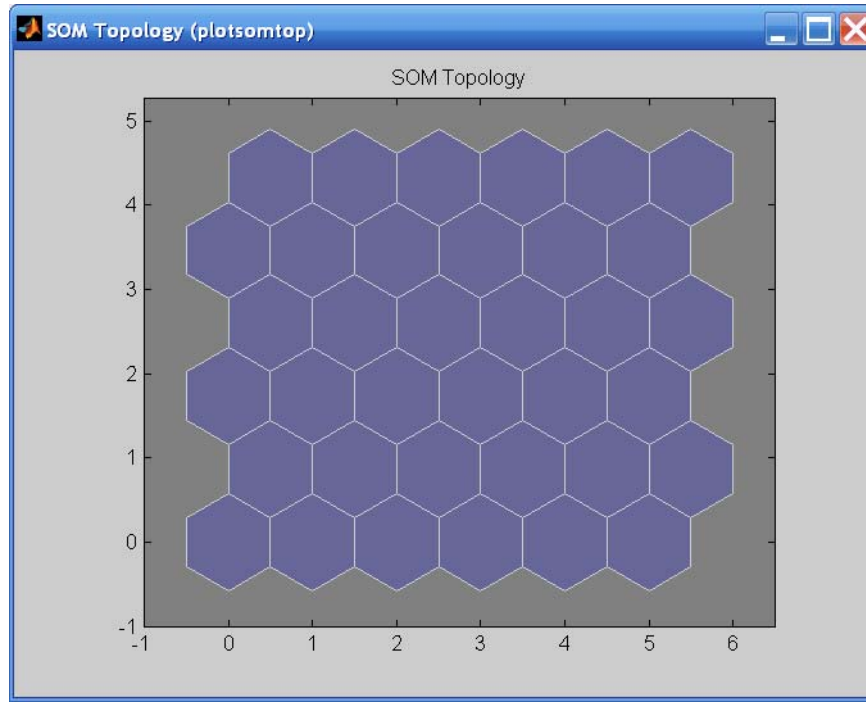
3. Train the network. The SOM network uses the default batch SOM algorithm for training.

   ```
   net=train(net,irisInputs);
   ```

4. During training, the training window opens and displays the training progress. To interrupt training at any point, click **Stop Training**.



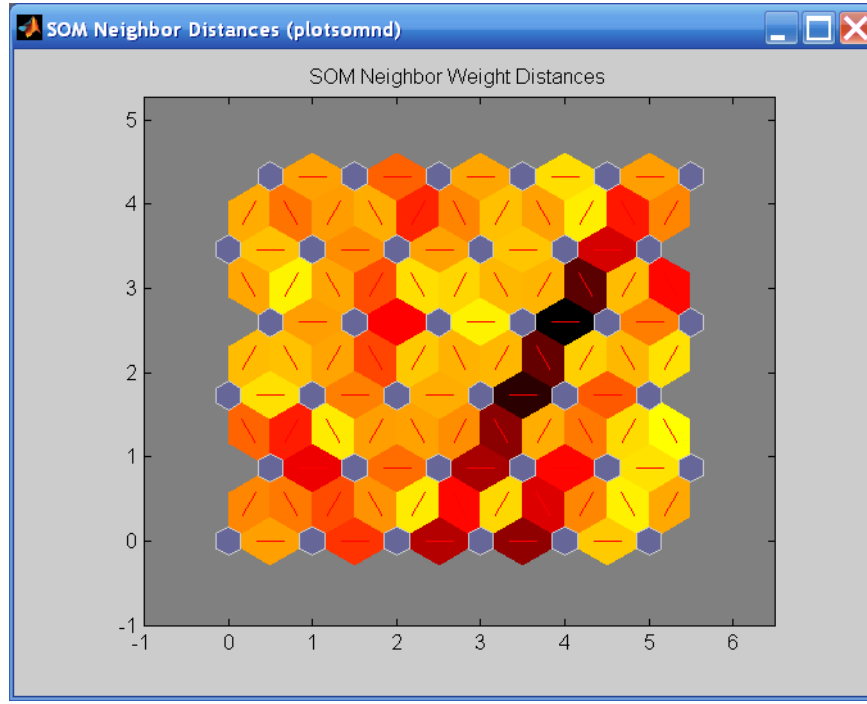
5 For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space. The default topology is hexagonal; to view it, click **SOM Topology** from the network training window.



In this figure, each of the hexagons represents a neuron. The grid is 6-by-6, so there are a total of 36 neurons in this network. There are four elements in each input vector, so the input space is four-dimensional. The weight vectors (cluster centers) fall within this space.

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

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



In this figure, the blue hexagons represent the neurons. The red lines connect neighboring neurons. The colors in the regions containing the red lines indicate the distances between neurons. The darker colors represent larger distances, and the lighter colors represent smaller distances.

A band of dark segments crosses from the lower-center region to the upper-right region. The SOM network appears to have clustered the flowers into two distinct groups.

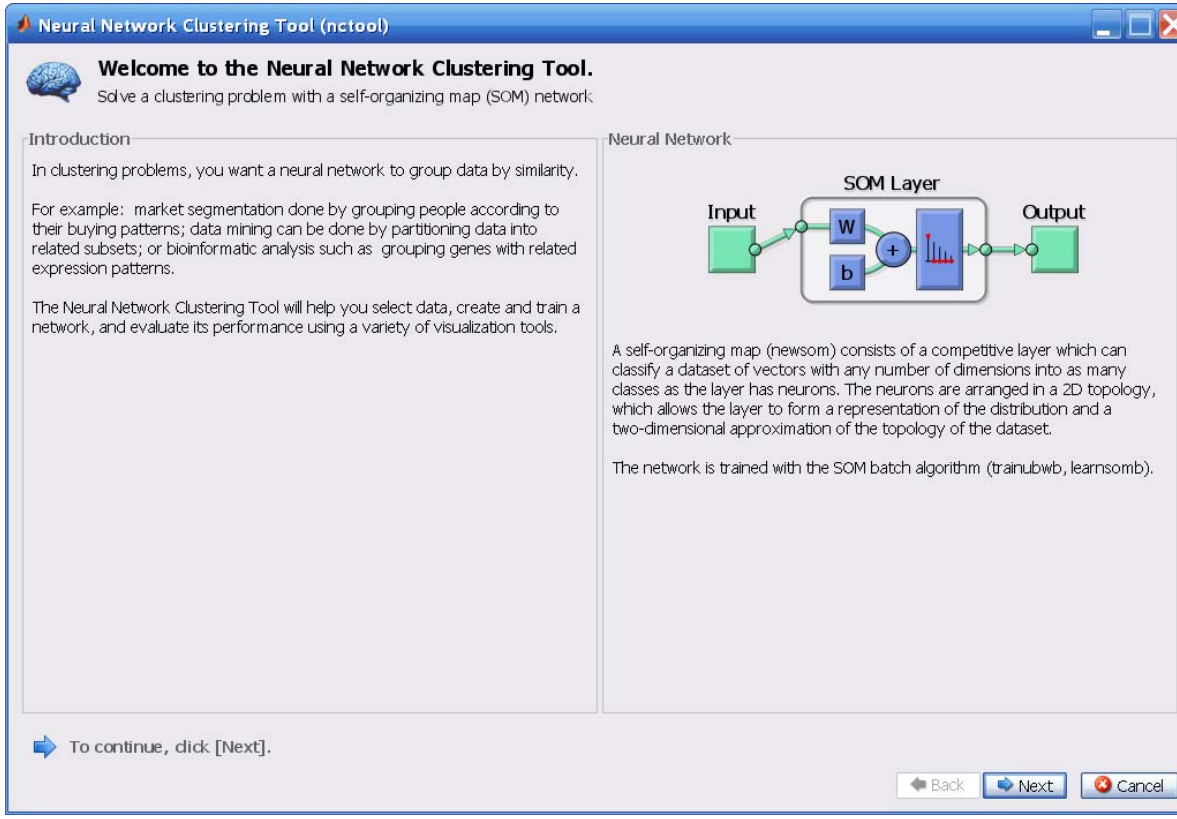
To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the SOM weight position plot) and watch it animate
- Plot from the command line with functions such as `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`. (For more information on using these functions, see their reference pages.)

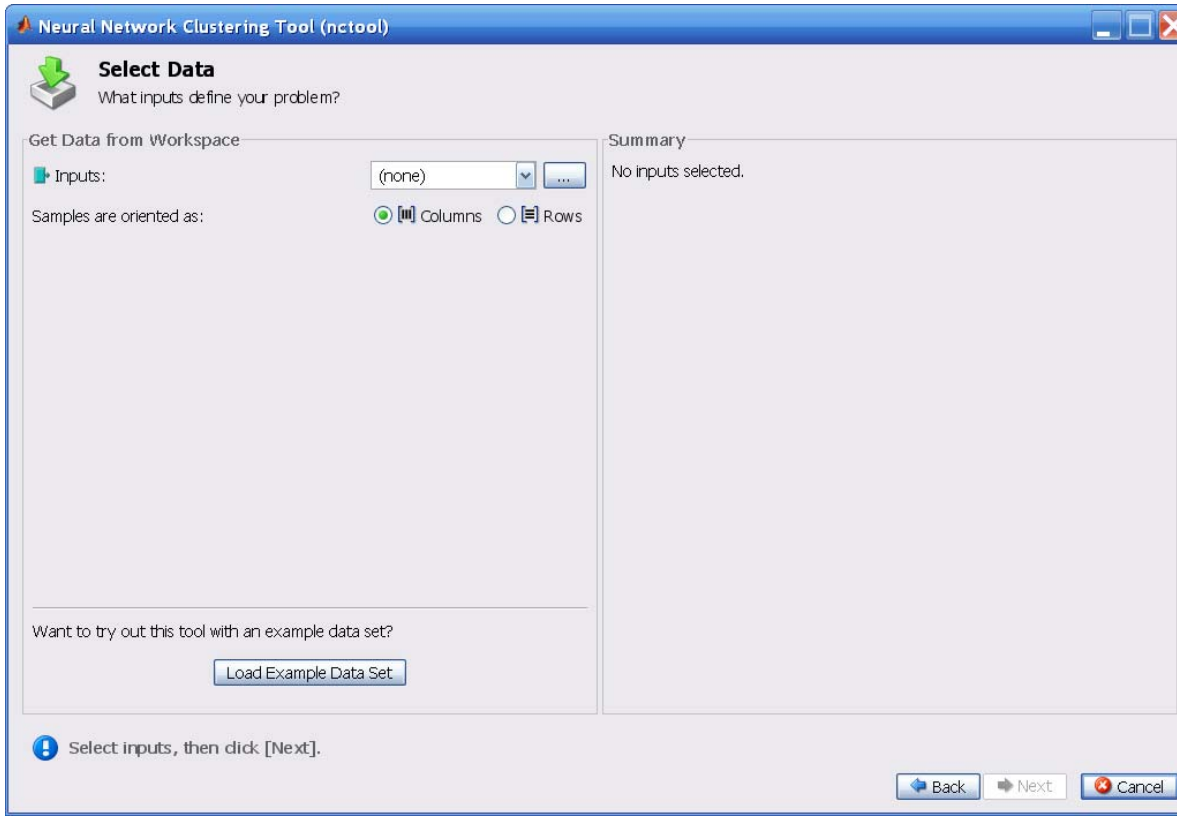
Using the Neural Network Clustering Tool GUI

1 Open the Neural Network Clustering Tool window with this command:

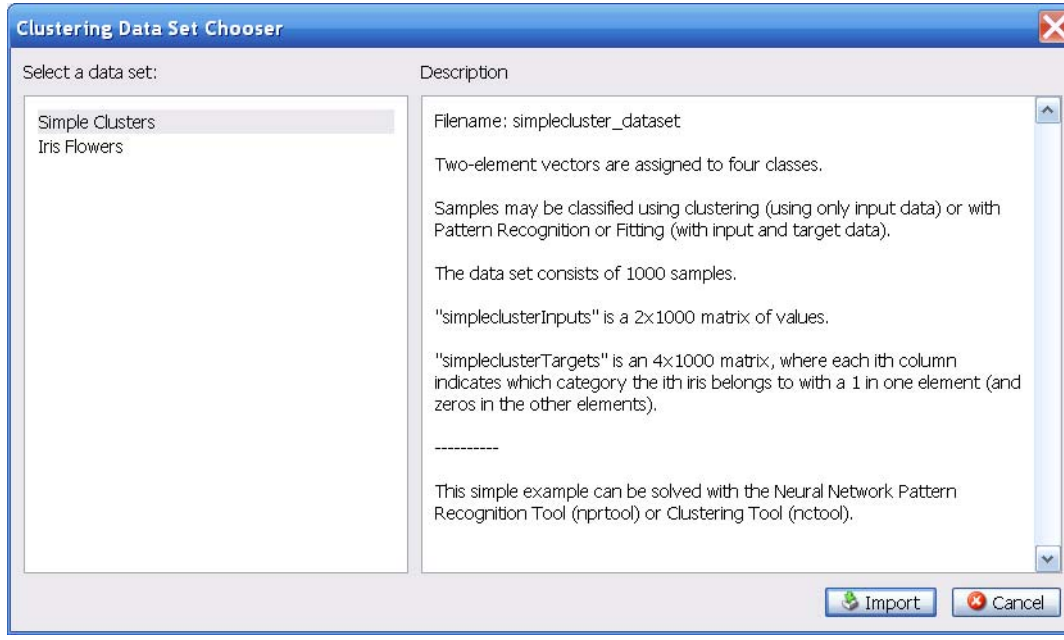
```
nctool
```



2 Click **Next**. The Select Data window appears.



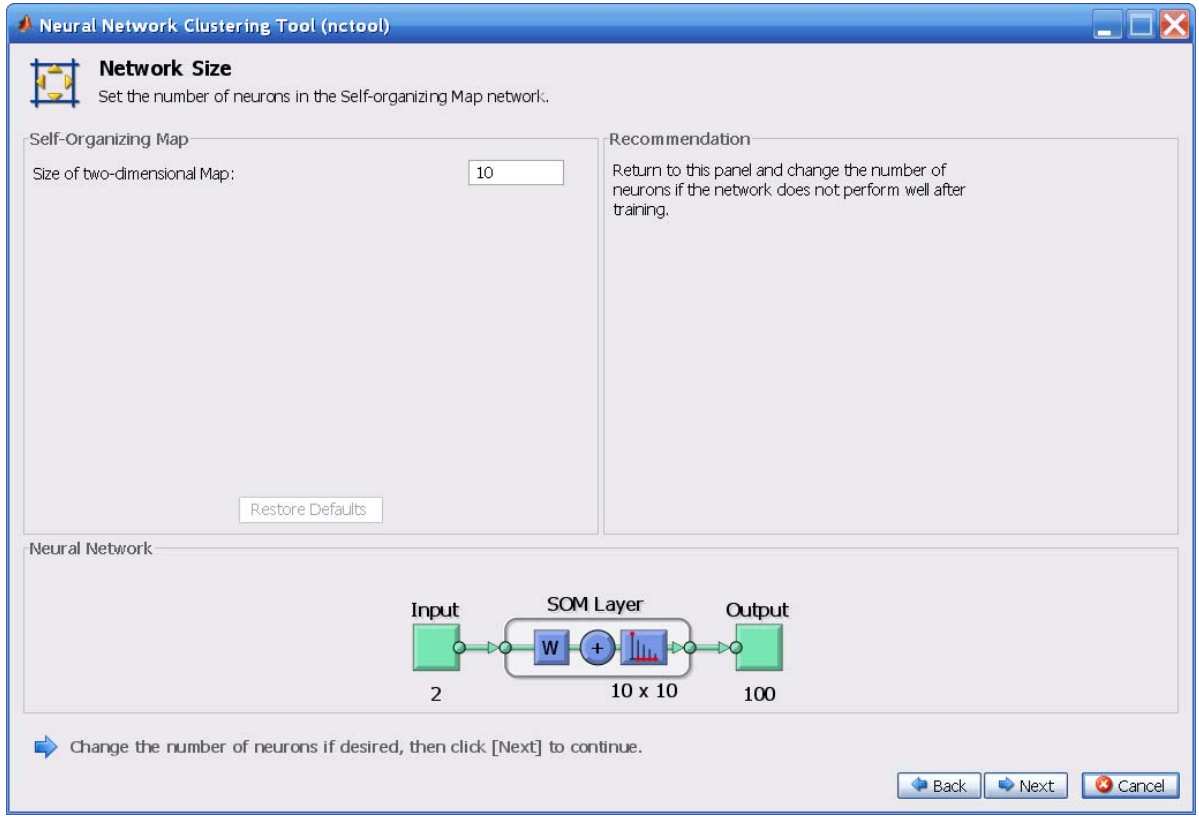
3 Click **Load Example Data Set**. The Clustering Data Set Chooser window appears.



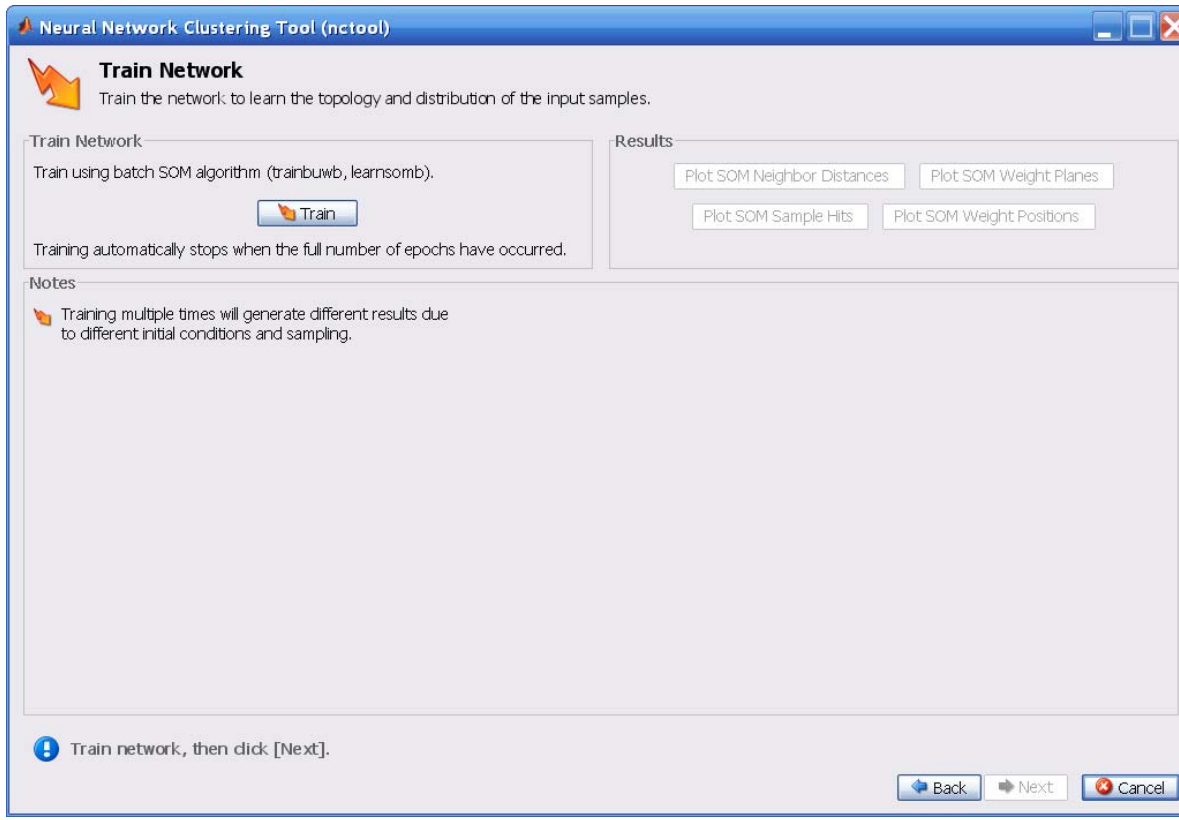
4 In this window, select **Simple Clusters**, and click **Import**. You return to the Select Data window.

5 Click **Next** to continue to the Network Size window, shown in the following figure.

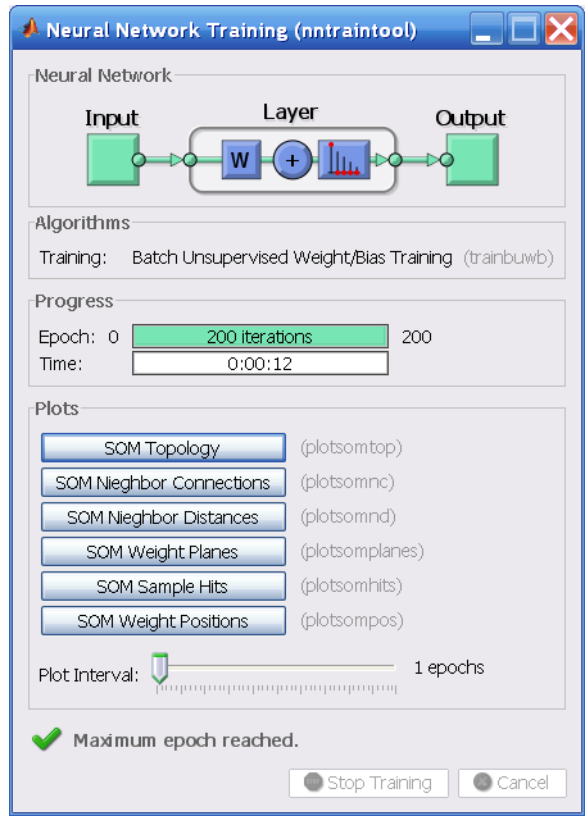
The size of the two-dimensional map is set to 10. This map represents one side of a two-dimensional grid. The total number of neurons is 100. You can change this number in another run if you want.



6 Click **Next**. The Train Network window appears.

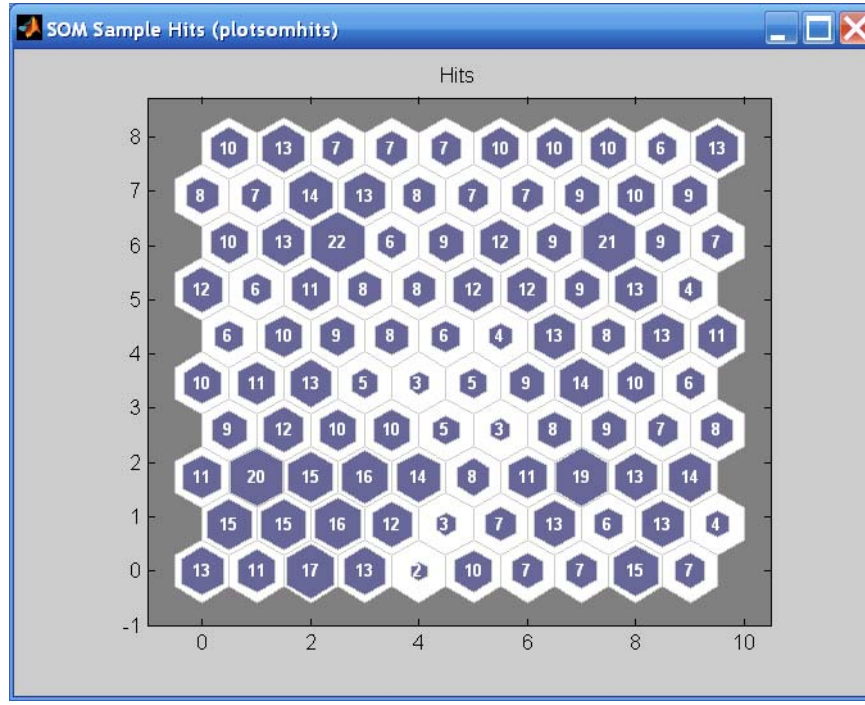


7 Click **Train**



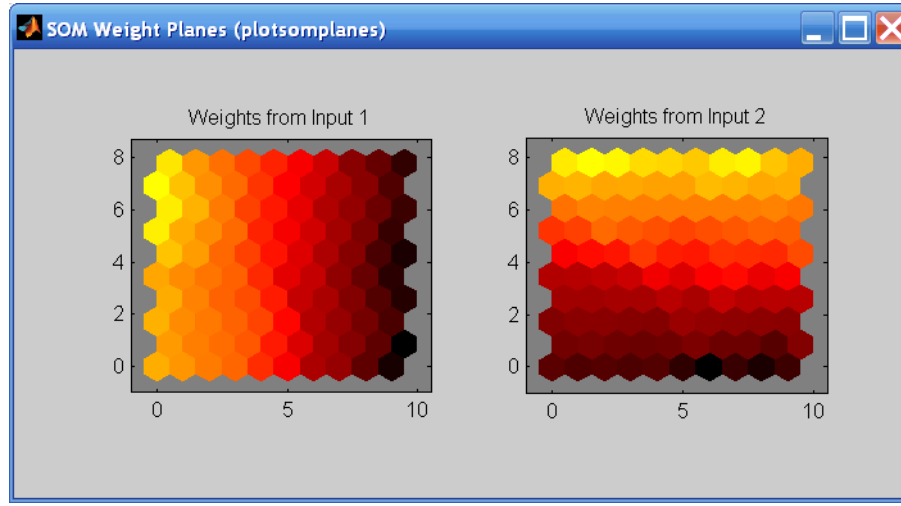
The training runs for the maximum number of epochs, which is 200.

8 Investigate some of the visualization tools for the SOM. Under the **Plots** pane, click **SOM Sample Hits**.



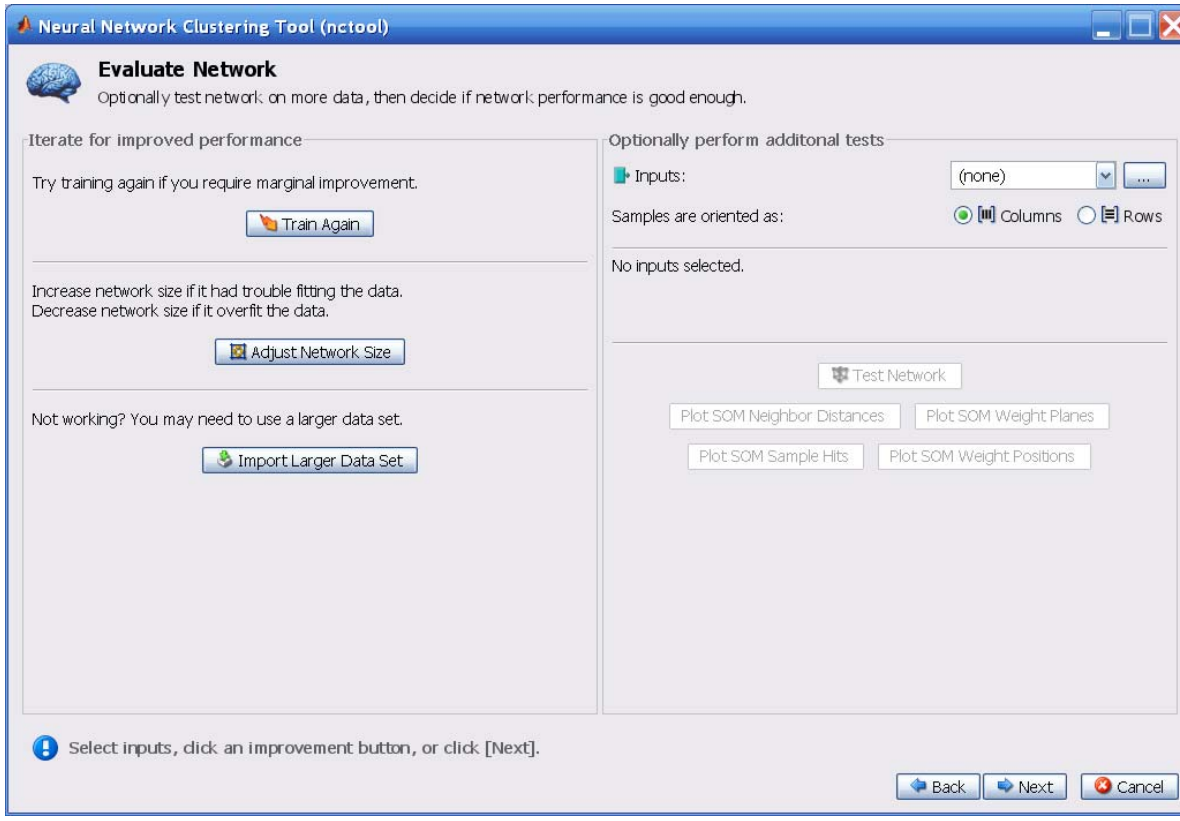
This figure shows how many of the training data are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated with any neuron is 22. Thus, there are 22 input vectors in that cluster.

9 You can also visualize the SOM by displaying weight places (also referred to as *component planes*). Click **SOM Weight Planes** in the Neural Network Clustering Tool.



This figure shows 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. (Darker colors represent larger weights.) If the connection patterns of two inputs were very similar, you can assume that the inputs are highly correlated. In this case, input 1 has connections that are very different than those of input 2.

10 In the Neural Network Clustering Tool, click **Next** to evaluate the network.

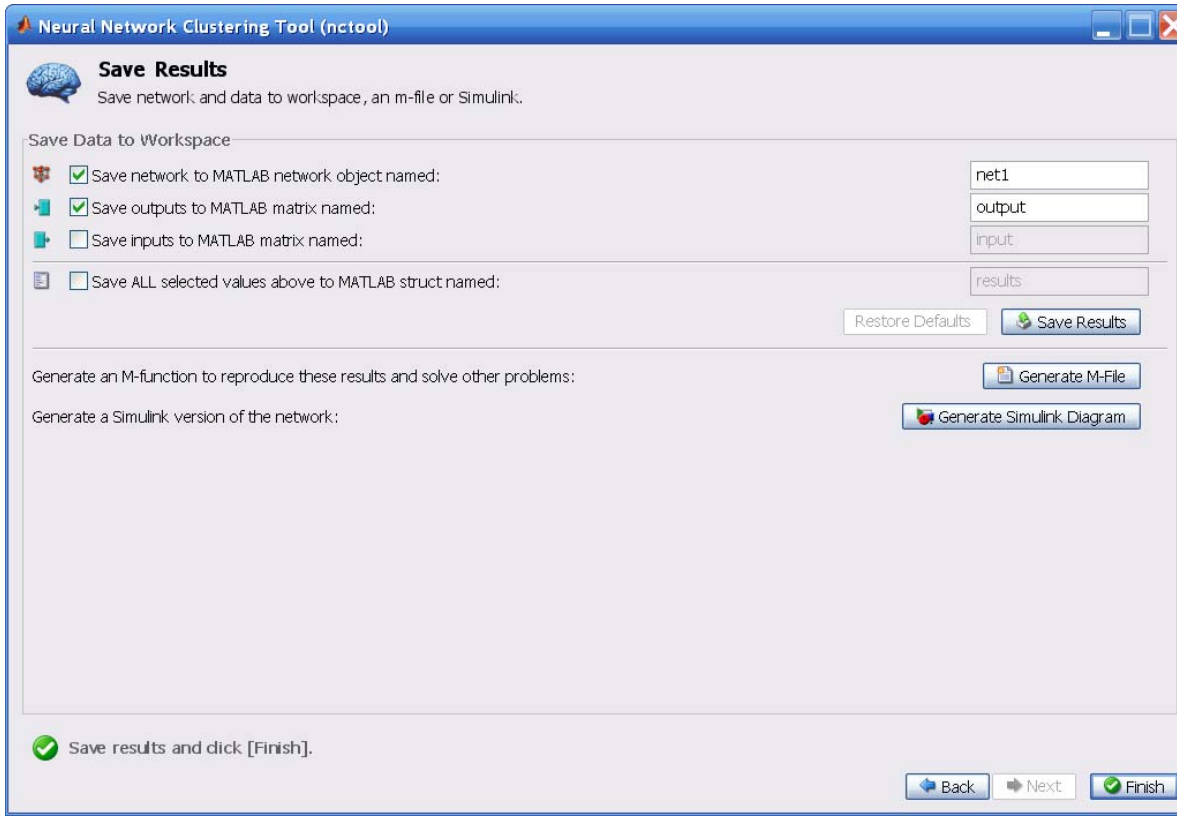


At this point you can test the network against new data.

If you are dissatisfied with the network's performance on the original or new data, you can increase the number of neurons, or perhaps get a larger training data set.

11 When you are satisfied with the network performance, click **Next**.

12 Use the buttons on this screen to save your results.



- You now have the network saved as `net1` in the workspace. You can perform additional tests on it, or put it to work on new inputs, using the function `sim`.
- If you click **Generate M-File**, the tool creates an M-file, with commands that recreate the steps that you have just performed from the command line. Generating an M-file is a good way to learn how to use the command-line operations of the Neural Network Toolbox™ software.

13 When you have saved your results, click **Finish**.

2

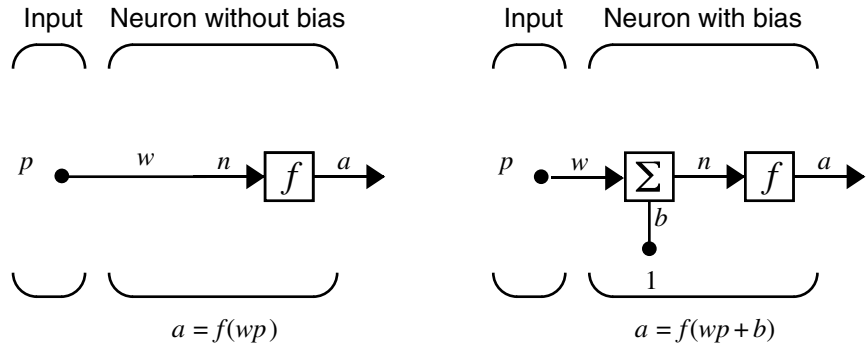
Neuron Model and Network Architectures

Neuron Model (p. 2-2)
Network Architectures (p. 2-8)
Data Structures (p. 2-14)
Training Styles (p. 2-20)

Neuron Model

Simple Neuron

A neuron with a single scalar input and no bias appears on the left below.



The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w to form the product wp, again a scalar. Here the weighted input wp is the only argument of the transfer function f, which produces the scalar output a. The neuron on the right has a scalar bias, b. You can view the bias as simply being added to the product wp as shown by the summing junction or 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.

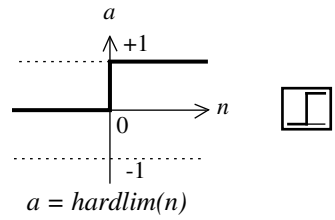
The transfer function net input n, again a scalar, is the sum of the weighted input wp and the bias b. This sum is the argument of the transfer function f. (Chapter 8, "Radial Basis Networks," discusses a different way to form the net input n.) Here f is a transfer function, typically a step function or a sigmoid function, that takes the argument n and produces the output a. Examples of various transfer functions are in "Transfer Functions" on page 2-3. 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, or perhaps the network itself will adjust these parameters to achieve some desired end.

All the neurons in the Neural Network 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.

As previously noted, the bias *b* is an adjustable (scalar) parameter of the neuron. It is *not* an input. However, the constant *1* that drives the bias is an input and must be treated as such when you consider the linear dependence of input vectors in Chapter 4, “Linear Filters.”

Transfer Functions

Many transfer functions are included the Neural Network Toolbox software. Three of the most commonly used functions are shown below.



Hard-Limit Transfer Function

The hard-limit transfer function shown above limits the output of the neuron to either 0, if the net input argument *n* is less than 0, or 1, if *n* is greater than or equal to 0. This function is used in Chapter 3, “Perceptrons,” to create neurons that make classification decisions.

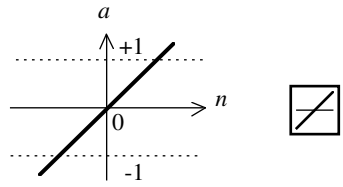
The toolbox has a function, `hardlim`, to realize the mathematical hard-limit transfer function shown above. Try the following code:

```
n = -5:0.1:5;
plot(n,hardlim(n),'c+:');
```

It produces a plot of the function `hardlim` over the range -5 to +5.

All the mathematical transfer functions in the toolbox can be realized with a function having the same name.

The following figure illustrates the linear transfer function.

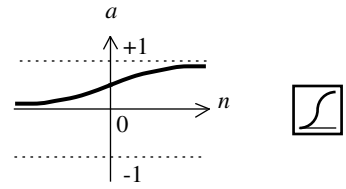


a = purelin(n)

Linear Transfer Function

Neurons of this type are used as linear approximators in Chapter 4, "Linear Filters."

The sigmoid transfer function shown below takes the input, which can have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



a = logsig(n)

Log-Sigmoid Transfer Function

This transfer function is commonly used in backpropagation 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 boxes of network diagrams to show the particular transfer function being used.

For a complete listing of transfer functions and their icons, You can also specify your own transfer functions.

You can experiment with a simple neuron and various transfer functions by running the demonstration program `nnd2n1`.

Neuron with Vector Input

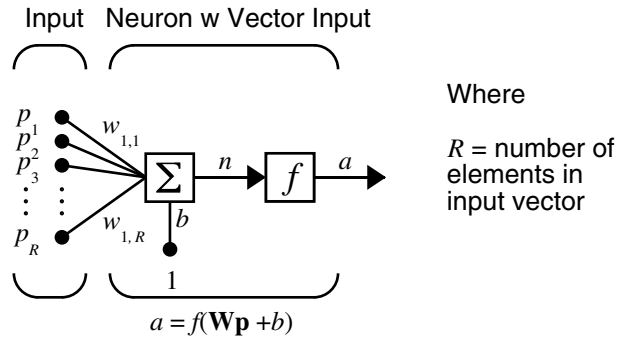
A neuron with a single R-element input vector is shown below. Here the individual element inputs

$$p_1, p_2, \ldots p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \ldots w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply **Wp**, the dot product of the (single row) matrix **W** and the vector **p**.



The neuron has a bias b, which is summed with the weighted inputs to form the net input n. This sum, n, is the argument of the transfer function f.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

This expression can, of course, be written in MATLAB® code as

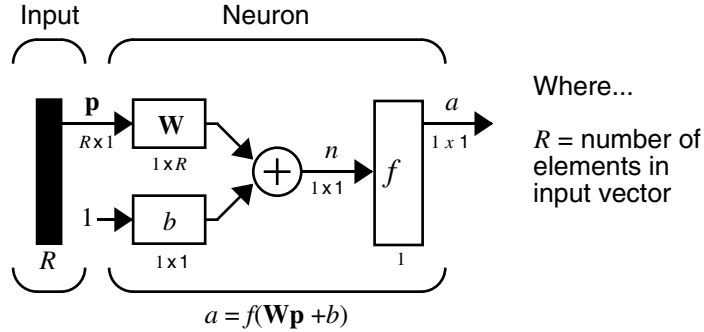
```
n = W*p + b
```

However, you will seldom be writing code at this level, for such code is already built into functions to define and simulate entire networks.

Abbreviated Notation

The figure of a single neuron shown above contains a lot of detail. When you consider networks with many neurons, and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the

authors have devised an abbreviated notation for an individual neuron. This notation, which is used later in circuits of multiple neurons, is shown.

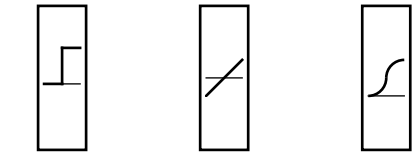


Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as *R*x1. (Note that a capital letter, such as *R* in the previous sentence, is used when referring to the *size* of a vector.) Thus, **p** is a vector of *R* input elements. These inputs postmultiply the single-row, *R*-column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias *b*. The net input to the transfer function *f* is *n*, the sum of the bias *b* and the product **Wp**. This sum is passed to the transfer function *f* to get the neuron's output *a*, which in this case is a scalar. Note that if there were more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the previous figure. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product **Wp**), the bias *b*, and the transfer function *f*. The array of inputs, vector **p**, is not included in or called a layer.

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 2-3, 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.



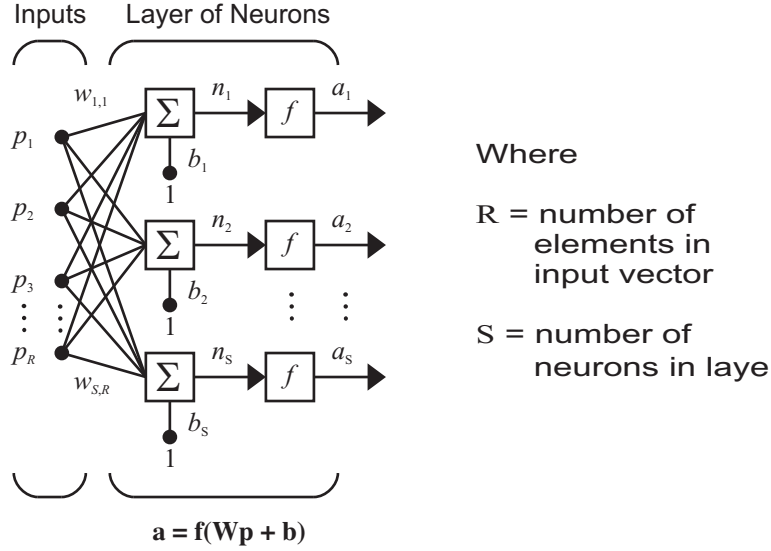
You can experiment with a two-element neuron by running the demonstration program `nnd2n2`.

Network Architectures

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.

A Layer of Neurons

A one-layer network with R input elements and S neurons follows.



In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$. The various $n(i)$ taken together form an S-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. The expression for **a** is shown at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., R is not necessarily equal to S). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

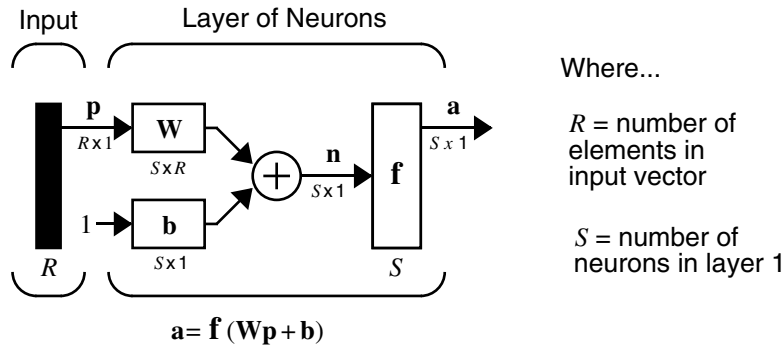
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R input one-layer network also can be drawn in abbreviated notation.



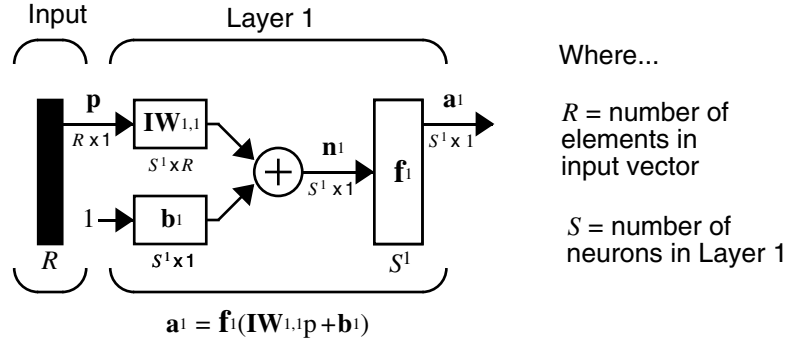
Here **p** is an R length input vector, **W** is an SxR matrix, and **a** and **b** are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function boxes.

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 coming from 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 below.

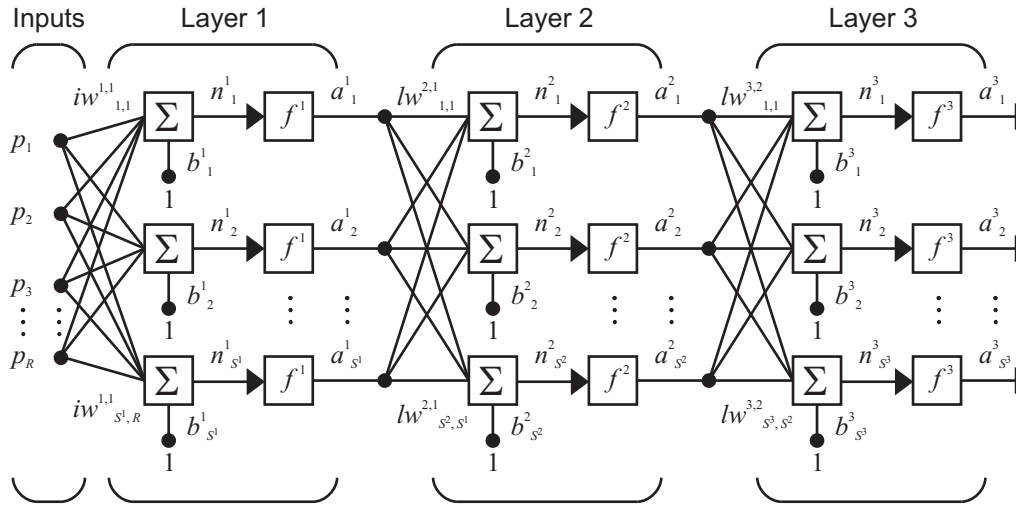


As you can see, the weight matrix connected to the input vector **p** is labeled as an input weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

"Multiple Layers of Neurons" 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 below, and in the equations at the bottom of the figure.



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2+\mathbf{b}^3)$$

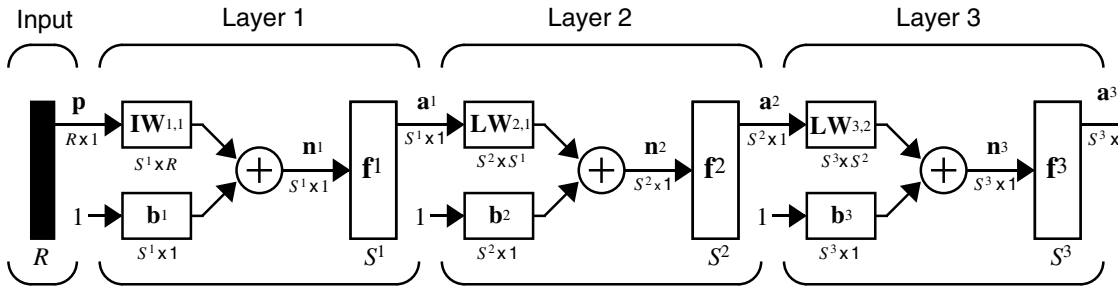
$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

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^2xS^1 weight matrix 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 same three-layer network can also be drawn using abbreviated notation.



$$\mathbf{a}^3 = \mathbf{f}^3\left(\mathbf{LW}^{3,2}\,\mathbf{f}^2\left(\mathbf{LW}^{2,1}\mathbf{f}^1\left(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1\right)+\mathbf{b}^2\right)+\mathbf{b}^3\right) = \mathbf{y}$$

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 Chapter 5, “Backpropagation.”

Here it is assumed that the output of the third layer, $\mathbf{a}^3$, is the network output of interest, and this output is labeled as **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 values for 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 altered for network use.

Processing functions are described in more detail in "Preprocessing and Postprocessing" in Chapter 5.

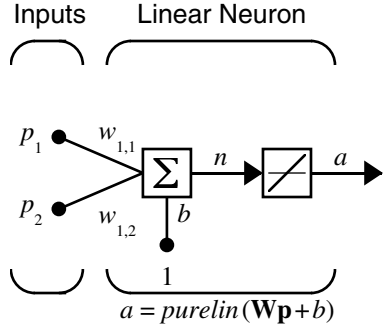
Data Structures

This section discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks.

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.



Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

Suppose that this network typically outputs -100, 50 and 100. This is arbitrary for this example. If you were solving a real problem, you would have actual values.

```
T = [-100 50 50 100];
```

To set up this feedforward network, use the following command:

```
net = newlin(P,T);
```

For simplicity assign the weight matrix and bias to be

$$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix} \text{ and } b = \begin{bmatrix} 0 \end{bmatrix}$$

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

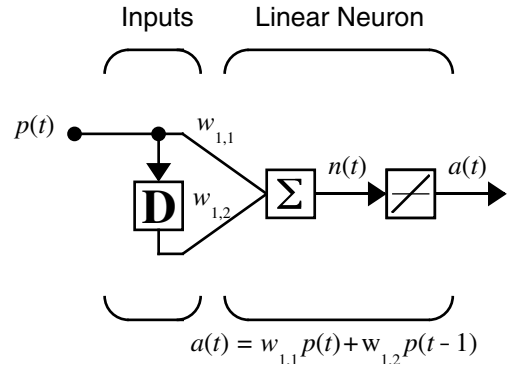
You can now simulate the network:

```
A = sim(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.



Suppose that the input sequence is

$$\mathbf{p}1 = \left[1\right],\ \mathbf{p}2 = \left[2\right],\ \mathbf{p}3 = \left[3\right],\ \mathbf{p}4 = \left[4\right]$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

Suppose you know that the typical output values include 10, 3 and -7. These values are arbitrary for this example; if you were solving a real problem, you would have real output values.

```
T = {10, 3, 7};
```

The following commands create this network:

```
net = newlin(P,T,[0 1]);
net.biasConnect = 0;
```

Assign the weight matrix to be

$$\mathbf{W} = \left[1\ 2\right]$$

The command is

```
net.IW{1,1} = [1 2];
```

You can now simulate the network:

```
A = sim(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 2-15, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When you simulate with concurrent inputs, you obtain

```
A = sim(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:

$$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix},\ \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix},\ \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$$

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:

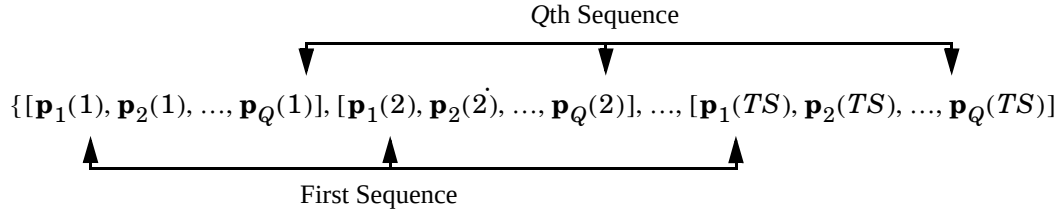
```
A = sim(net,P);
```

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 input `P` to the `sim` function 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 section, you apply sequential and concurrent inputs to dynamic networks. In “Simulation with Concurrent Inputs in a Static Network” on page 2-14, 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 “Training Styles” on page 2-20.

Training Styles

This section 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.

Incremental Training (of Adaptive and Other Networks)

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 demonstrates how incremental training is performed on both static and dynamic networks.

Incremental Training with 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

$$\mathbf{t}_1 = \begin{bmatrix} 4 \end{bmatrix}, \ \mathbf{t}_2 = \begin{bmatrix} 5 \end{bmatrix}, \ \mathbf{t}_3 = \begin{bmatrix} 7 \end{bmatrix}, \ \mathbf{t}_4 = \begin{bmatrix} 7 \end{bmatrix}$$

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 = newlin(P,T,0,0);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Recall from "Simulation with Concurrent Inputs in a Static Network" on page 2-14 that, for a static network, the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. However, this is not true when training the network. When you use the `adapt` function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As shown in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

You are now ready to train the network incrementally.

```
[net,a,e,pf] = adapt(net,P,T);
```

The network outputs remain zero, because the learning rate is zero, and the weights are not updated. The errors are equal to the targets:

```
a = [0]    [0]    [0]    [0]
e = [4]    [5]    [7]    [7]
```

If you now set the learning rate to 0.1 you can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr=0.1;
net.biases{1,1}.learnParam.lr=0.1;
[net,a,e,pf] = adapt(net,P,T);
a = [0]    [2]    [6]    [5.8]
e = [4]    [3]    [1]    [1.2]
```

The first output is the same as it was with zero learning rate, because no update is made until the first input is presented. The second output is different, because the weights have been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error is eventually driven to zero.

Incremental Training with Dynamic Networks

You can also train dynamic networks incrementally. In fact, this would be the most common situation.

Here are the initial input `Pi` and the inputs `P` and targets `T` as elements of cell arrays.

```
Pi = {1};
P = {2 3 4};
T = {3 5 7};
```

Create a 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 = newlin(P,T,[0 1],0.1);
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 (using `sim`) with the exception that you assign the first term in the sequence as the initial condition for the delay. You can now sequentially train the network using `adapt`.

```
[net,a,e,pf] = adapt(net,P,T,Pi);
a = [0] [2.4] [7.98]
e = [3] [2.6] [-0.98]
```

The first output is zero, because the weights have not yet been updated. The weights change at each subsequent time step.

Batch Training

Batch training, in which weights and biases are only updated after all the inputs and targets are presented, can be applied to both static and dynamic networks. Both types of networks are discussed in this section.

Batch Training with Static Networks

Batch training can be done using either `adapt` or `train`, although `train` is generally the best option, because it typically has access to more efficient training algorithms. Incremental training can only be done with `adapt`; `train` can only perform batch training.

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 0.1.

```
net = newlin(P,T,0,0.1);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

When you call `adapt`, it invokes `trains` (the default adaptation function for the linear network) and `learnwh` (the default learning function for the weights and biases). `trains` uses Widrow-Hoff learning.

```
[net,a,e,pf] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all the training set has been presented. If you display the weights, you find

```
»net.IW{1,1}
  ans = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

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 = newlin(P,T,0,0.1);
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 adaptation function was `trains`.

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1}.learnParam.lr = 0.1;
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 = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

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.)

```
Pi = {1};
P = {2 3 4};
T = {3 5 6};
net = newlin(P,T,[0 1],0.02);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
```

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')
```

3

Perceptrons

Introduction (p. 3-2)
Neuron Model (p. 3-3)
Perceptron Architecture (p. 3-5)
Creating a Perceptron (newp) (p. 3-6)
Learning Rules (p. 3-11)
Perceptron Learning Rule (learnp) (p. 3-12)
Training (train) (p. 3-15)
Limitations and Cautions (p. 3-21)
Graphical User Interface (p. 3-23)

Introduction

This chapter has a number of objectives. First, it introduces you to learning rules, methods of deriving the next changes that might be made in a network, and training, a procedure whereby a network is actually adjusted to do a particular job. Along the way, this chapter describes a toolbox function to create a simple perceptron network and functions to initialize and simulate such networks. The perceptron is used as a vehicle for tying these concepts together.

Rosenblatt [Rose61] 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.

This chapter defines what is meant by a learning rule, explains the perceptron network and its learning rule, and tells you how to initialize and simulate perceptron networks.

The discussion of perceptrons in this chapter is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

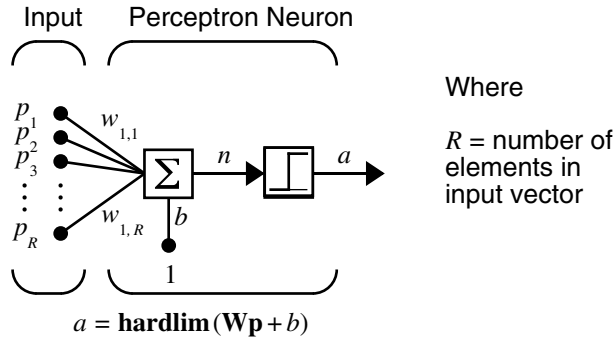
You might also want to refer to the original book on the perceptron, Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C., Spartan Press, 1961 [Rose61].

Important Perceptron Functions

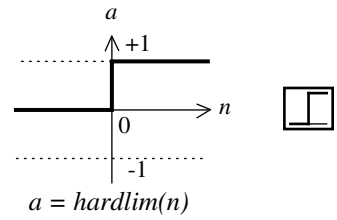
You can create perceptron networks with the function `newp`. These networks can be initialized, simulated, and trained with `init`, `sim`, and `train`. “Neuron Model” on page 3-3 describes how perceptrons work and introduces these functions.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



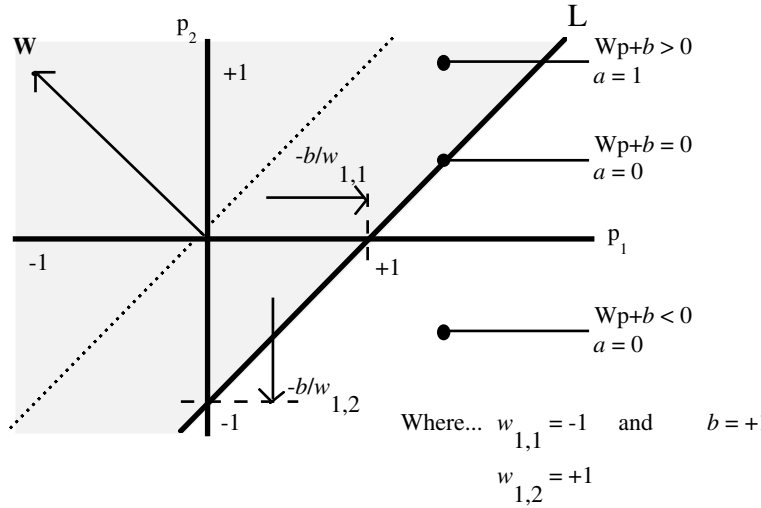
Each external input is weighted with an appropriate weight w_{1j}, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The following figure show the input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$.



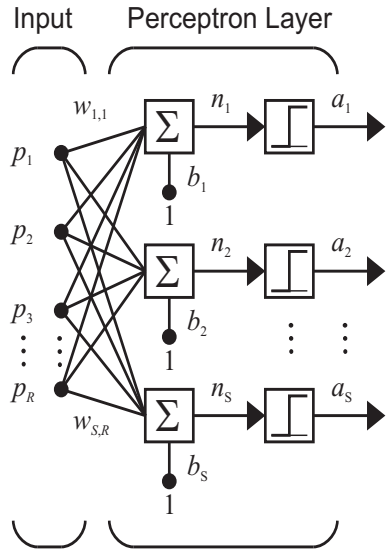
Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$. This line is perpendicular to the weight matrix **W** and shifted according to the bias *b*. Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin, as shown in the plot above.

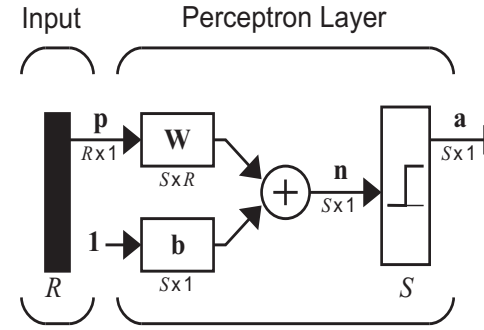
You might want to run the demonstration program `nnd4db`. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights $w_{i,j}$, as shown below in two forms. As before, the network indices i and j indicate that $w_{i,j}$ is the strength of the connection from the jth input to the ith neuron.



a = hardlim(Wp + b)



a = hardlim(Wp + b)

Where

R = number of elements in input

S = number of neurons in layer

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 3-21.

Creating a Perceptron (newp)

A perceptron can be created with the `newp` function:

```
net = newp(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 = newp(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: 1
      learnFcn: 'learnp'
    learnParam: []
          size: [1 1]
      userdata: [1x1 struct]
     weightFcn: 'dotprod'
   weightParam: [1x1 struct]
```

The default learning function is `learnp`, which is discussed in “Perceptron Learning Rule (learnp)” on page 3-12. 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.

Simulation (sim)

This section shows how `sim` works using a simple problem.

Suppose that you take a perceptron with a single two-element input vector, such as discussed in the decision boundary figure on page 3-4. This perceptron outputs the values 0 and 1.

Define the network using the following commands:

```
P = [-2 2;-2 2];
T = [0 1];
net = newp(P,[0 1]);
```

The network includes zero weights and biases. If you want weights and biases with values other than zero, you have to create them.

Set the two weights and the one bias to -1, 1, and 1, as they were in the decision boundary figure using the following commands:

```
net.IW{1,1}= [-1 1];
net.b{1} = [1];
```

To make sure that these parameters were set correctly, check them with

```
net.IW{1,1}
ans =
    -1     1
net.b{1}
ans =

     1
```

Now see if the network responds to two signals, one on each side of the perceptron boundary:

```
p1 = [1;1];
a1 = sim(net,p1)
a1 =

     1
```

and for

```
p2 = [1;-1];
a2 = sim(net,p2)
a2 =

     0
```

Sure enough, the perceptron classified the two inputs correctly.

You could present the two inputs in a sequence and get the outputs in a sequence as well:

```
p3 = {[1;1] [1;-1]};
a3 = sim(net,p3)
a3 =

    [1]    [0]
```

Initialization (init)

You can use the function `init` to reset the network weights and biases to their original values. Suppose, for instance, that you start with the network

```
net = newp([-2 2;-2 2],[0 1]);
```

Now check the weights using the following command:

```
wts = net.IW{1,1}
```

which gives, as expected,

```
wts =

     0     0
```

In the same way, you can verify that the bias is 0 with

```
bias = net.b{1}
```

which gives

```
bias =

     0
```

Now set the weights to the values 3 and 4 and the bias to the value 5 with

```
net.IW{1,1} = [3,4];
net.b{1} = 5;
```

Recheck the weights and bias as shown above to verify that the change has been made. Sure enough,

```
wts =
     3     4
bias =

     5
```

Now use `init` to reset the weights and bias to their original values:

```
net = init(net);
```

You can check as shown above to verify that

```
wts =
     0     0
bias =

     0
```

You can change the way that a perceptron is initialized with `init`. For instance, you can redefine the network input weights and bias `initFcns` as `rands`, and then apply `init`:

```
net.inputweights{1,1}.initFcn = 'rands';
net.biases{1}.initFcn = 'rands';
net = init(net);
```

Now check the weights and bias:

```
wts =
   -0.2371 -0.8976
biases =
    0.0775
```

You can see that the weights and bias are assigned random numbers.

Learning Rules

A *learning rule* is defined as a procedure for modifying the weights and biases of a network. (This procedure can also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules in this toolbox fall into two broad categories: supervised learning, and unsupervised learning.

In *supervised learning*, the learning rule is provided with a set of examples (the *training set*) of proper network behavior

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category.

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1\mathbf{t}_1,\mathbf{p}_2\mathbf{t}_1,\ldots,\mathrm{p}_Q\mathbf{t}_Q$$

where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference **t** – **a** between the neuron response **a** and the target vector **t**. The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector **p** and the associated error **e**. The target vector **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 **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 **w** and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct (**a** = **t** and **e** = **t** – **a** = 0), then the weight vector **w** is not altered.

CASE 2. If the neuron output is 0 and should have been 1 (**a** = 0 and **t** = 1, and **e** = **t** – **a** = 1), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 (**a** = 1 and **t** = 0, and **e** = **t** – **a** = –1), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$ 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:

$$\Delta\mathbf{w} = (t - a)\mathbf{p}^T = e\mathbf{p}^T$$

You can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - a)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^T = \mathbf{e}(\mathbf{p})^T$$

and

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = e$$

The perceptron learning rule can be summarized as follows:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

and

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$.

Now try a simple example. Start with a single neuron having an input vector with just two elements. Here are input vectors with the values -2 and 2, and outputs with values 0 and 1.

```
net = newp([-2 2;-2 2],[0 1]);
```

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 = sim(net,p)
a =
     0
e = t-a
e =
     1
```

and use the function `learnp` to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[])
dw =
     1     2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Recall that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are linearly separable. The objects to be classified in such cases can be separated by a single line.

You might want to try demo `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

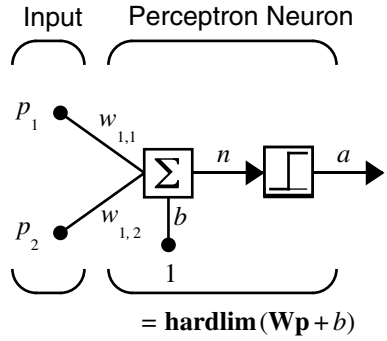
Training (train)

If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traversal through all the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. You must check the new values of **W** and **b** by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully you can train it further by calling `train` again with the new weights and biases for more training passes, or you can analyze the problem to see if it is a suitable problem for the perceptron. Problems that cannot be solved by the perceptron network are discussed in "Limitations and Cautions" on page 3-21.

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].

Suppose you have the following classification problem and would like to solve it with a single vector input, two-element perceptron network.

$$\mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \left\} \right. \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right.$$

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 $b(0)$.

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0$$

Start by calculating the perceptron's output a for the first input vector $\mathbf{p}_1$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0 \right) = hardlim(0) = 1$$

The output a does not equal the target value t_1, so use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)\begin{bmatrix} 2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix}$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the perceptron update rules.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = \begin{bmatrix} 0 & 0 \end{bmatrix} + \begin{bmatrix} -2 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix} = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, $\mathbf{p}_2$. The output is calculated below.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix}-2 & -2\end{bmatrix}$ and $p(2) = p(1) = -1$.

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 $\mathbf{W}(4) = \begin{bmatrix}-3 & -1\end{bmatrix}$ 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

$$\mathbf{W}(6) = \begin{bmatrix}-2 & -3\end{bmatrix} \text{ and } b(6) = 1$$

This concludes the hand calculation. Now, how can you do this using the `train` function?

The following code defines a perceptron like that shown in the previous figure, with initial weights and bias values of 0.

```
net = newp([-2 2;-2 2],[0 1]);
```

Consider the application of a single input.

```
p = [2; 2];
```

having the target

```
t = [0];
```

Set `epochs` to 1, so that `train` goes through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2    -2
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values [-2 -2] and -1, just as you hand calculated.

Now apply the second input vector $\mathbf{p}_2$. The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0, and the change will be zero. You could proceed in this way, starting from the previous result and applying a new input vector time after time. But you can do this job automatically with `train`.

Apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = newp([-2 2;-2 2],[0 1]);
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 = sim(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 =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 = sim(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 `newp` is `trainc`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You might want to try various demonstration 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`, which is discussed in detail in the next chapter. 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].

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:

$$\Delta \mathbf{w} = (t - a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

As shown above, the larger an input vector **p**, the larger its effect on the weight vector **w**. Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that the effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t - a)\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|} = e\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|}$$

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.

Graphical User Interface

Introduction to the GUI

The graphical user interface (GUI) is designed to be simple and user friendly. A simple example will get you started.

You bring up a GUI Network/Data Manager window. This window has its own work area, separate from the more familiar command-line workspace. Thus, when using the GUI, you might export the GUI results to the (command-line) workspace. Similarly, you might want to import results from the workspace to the GUI.

Once the Network/Data Manager window is up and running, you can create a network, view it, train it, simulate it, and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI.

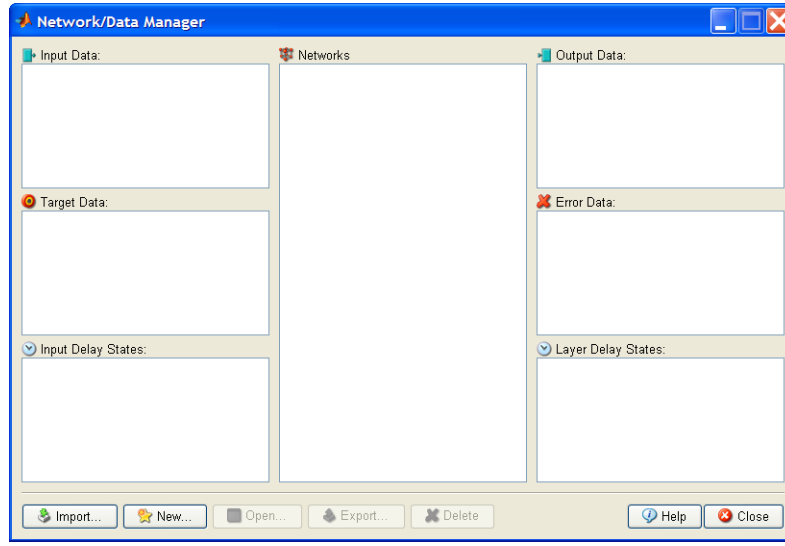
The following example deals with a perceptron network. It goes through all the steps of creating a network and shows what you might expect to see as you go along.

Create a Perceptron Network (nntool)

Create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. Call the network `ANDNet`. Once created, the network will be trained. You can then save the network, its output, etc., by exporting it to the workspace.

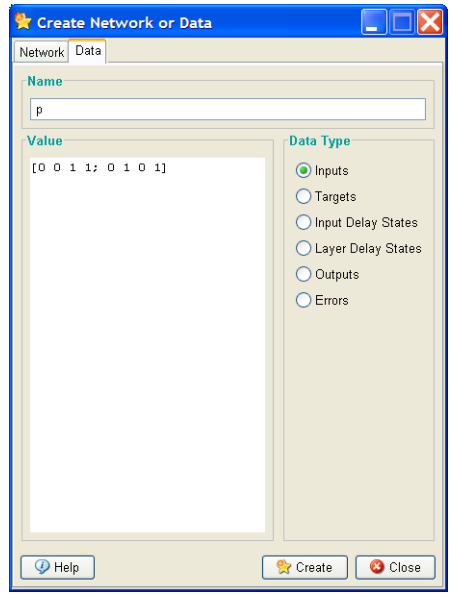
Input and Target

To start, type `nntool`. The following window appears.



Click **Help** to get started on a new problem and to see descriptions of the buttons and lists.

First, define the network input, called p, having the value [0 0 1 1;0 1 0 1]. Thus, the network has a two-element input, and four sets of such two-element vectors are presented to it in training. To define this data, click **New**, and a new window, Create Network or Data, appears. Select the **Data** tab. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.



Click **Create** and then click **OK** to create an input `p`. The Network/Data Manager window appears, and `p` shows as an input.

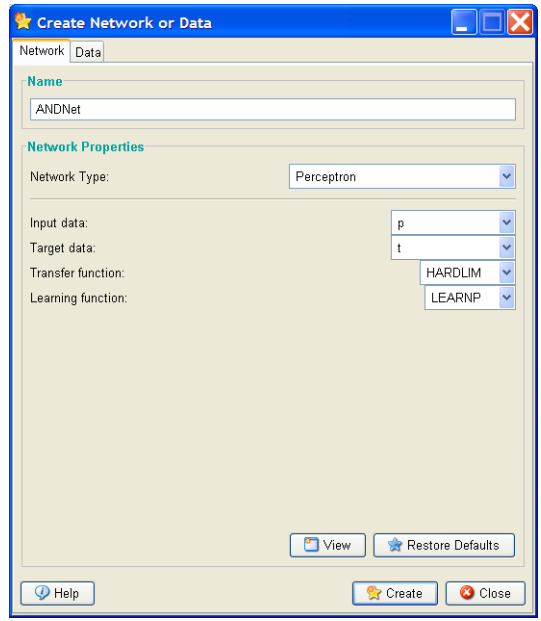
Next create a network target. This time enter the variable name `t`, specify the value `[0 0 0 1]`, and click **Target** under **Data Type**. Again click **Create** and **OK.** You will see in the resulting Network/Data Manager window that you now have `t` as a target as well as the previous `p` as an input.

Create Network

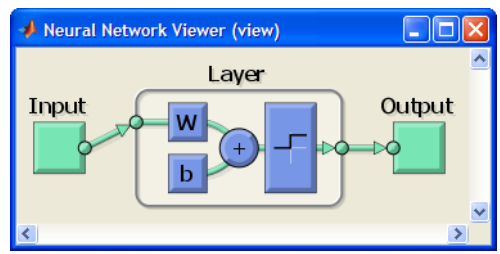
Now create a new network and call it `ANDNet`. Select the **Network** tab. Enter `ANDNet` under **Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network you want to create.

You can set the inputs to `p`, and the example targets to `t`.

You can use a `hardlim` transfer function with the output range [0, 1] that matches the target values and a `learnp` learning function. For the **Transfer function, select** `hardlim`. For the **Learning function**, select `learnp`. The Create Network or Data window now looks like the following figure.



To examine the network, click **View**.

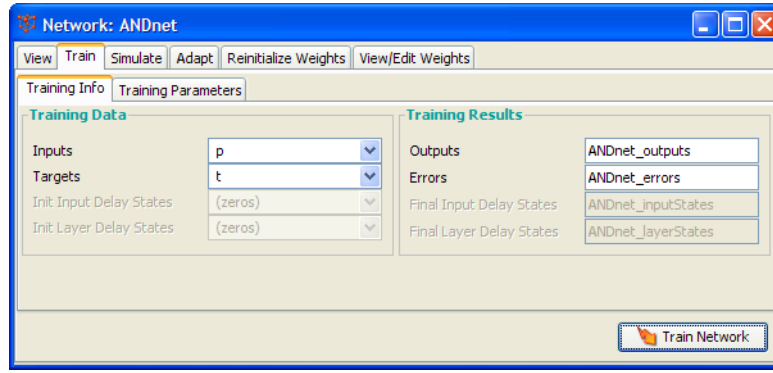


The View of New Network shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the desired perceptron network.

Now click **Create** and **OK** to generate the network. Now close the Create Network or Data window. You see the Network/Data Manager window with `ANDNet` listed as a network.

Train the Perceptron

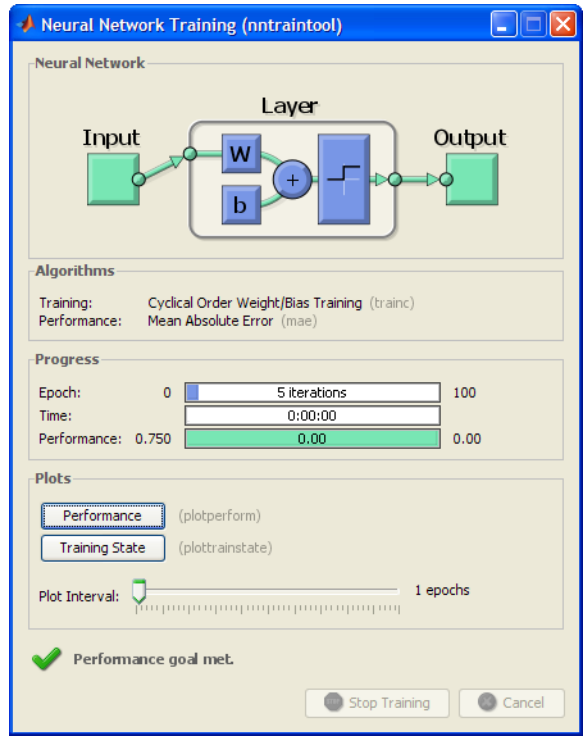
To train the network, click `ANDNet` to highlight it. Then click **Open**. This leads to a new window, labeled **Network: ANDNet**. At this point you can see the network again by clicking the **View** tab. You can also check on the initialization by clicking the **Initialize** tab. Now click the **Train** tab. Specify the inputs and output by clicking the **Training Info** tab and selecting `p` from the list of inputs and `t` from the list of targets. The Network: ANDNet window should look like



Note that the contents of the **Training Results Outputs and Errors** fields have the name `ANDNet_` prefixed to them. This makes them easy to identify later when they are exported to the workspace.

While you are here, click the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want.

Click **Train Network** to train the perceptron network. The following training results appear.



The network was trained to zero error in five epochs. (Other kinds of networks commonly do not train to zero error, and their errors can cover a much larger range. On that account, their errors are plotted on a log scale rather than on a linear scale such as that used above for perceptrons.)

Confirm that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, go to the **Network: ANDNet** window and click the **Simulate** tab**.** Use the **Inputs** menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Click **Simulate Network** in the lower right corner and click **OK**. In the Network/Data Manager you see a new variable in the output: `ANDNet_outputsSim`. Double-click it and a small window, **Data: ANDNet_outputsSim**, appears with the value

```
[0 0 0 1]
```

Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1. Close this window by clicking **OK**.

Export Perceptron Results to Workspace

To export the network outputs and errors to the MATLAB® workspace, go back to the Network/Data Manager window. The output and error for `ANDNet` are listed in the **Outputs** and **Errors** fields on the right side. Next click **Export**. This gives you an Export from Network/Data Manager window. Click `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button.

These two variables now should be in the MATLAB workspace. To confirm this, go to the command line and type `who` to see all the defined variables. The result should be

```
who
Your variables are:
ANDNet_errors       ANDNet_outputs
```

You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following:

```
ANDNet_outputs =
0     0     0     1
```

and

```
ANDNet_errors =
0     0     0     0
```

You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check using `who` to make sure that they got to the workspace.

Now that `ANDNet` has been exported, you can view the network description and examine the network weight matrix.

For instance, the command

```
ANDNet.iw{1,1}
```

gives

```
ans =
2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
ans =
-3
```

Your network might yield a different result.

Clear Network/Data Window

You can clear the Network/Data Manager window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, you start from a clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean Network/Data Manager window.

Recall however, that you exported `p`, `t`, etc., to the workspace from the perceptron example. They are still there for your use even after you clear the Network/Data Manager window.

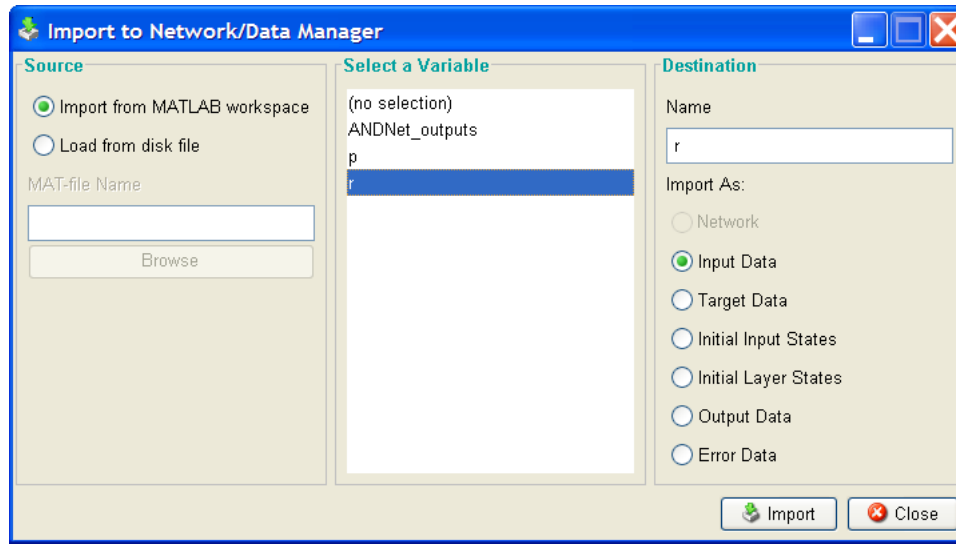
Importing from the Command Line

To make things simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Click **Import** and set the destination **Name** to `r` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this:



Click **Import** and verify by looking at the Network/Data Manager window that the variable `r` is there as an input.

Save a Variable to a File and Load It Later

Bring up the Network/Data Manager window and click **New Network**. Set the name to `mynet`. Click **Create**. The network name `mynet` should appear in the Network/Data Manager window. In this same window click **Export**. Select `mynet` in the variable list of the Export or Save window and click **Save**. This leads to the Save to a MAT File window. Save to the file `mynetfile`.

Now get rid of `mynet` in the GUI and retrieve it from the saved file. Go to the Network/ Data Manager window, highlight `mynet`, and click **Delete**. Click **Import**. This brings up the Import or Load to Network/Data Manager window. Click the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click **Browse**. This brings up the Select MAT File window, with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, click **Open**, and you return to the Import or Load to Network/Data Manager window. On the **Import As** list, select **Network**. Highlight `mynet` and click **Load** to bring `mynet` to the GUI. Now `mynet` is back in the GUI Network/Data Manager window.

4

Linear Filters

Introduction (p. 4-2)
Neuron Model (p. 4-3)
Network Architecture (p. 4-4)
Least Mean Square Error (p. 4-8)
Linear System Design (newlind) (p. 4-9)
Linear Networks with Delays (p. 4-10)
LMS Algorithm (learnwh) (p. 4-13)
Linear Classification (train) (p. 4-15)
Limitations and Cautions (p. 4-18)

Introduction

The linear networks discussed in this chapter 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.

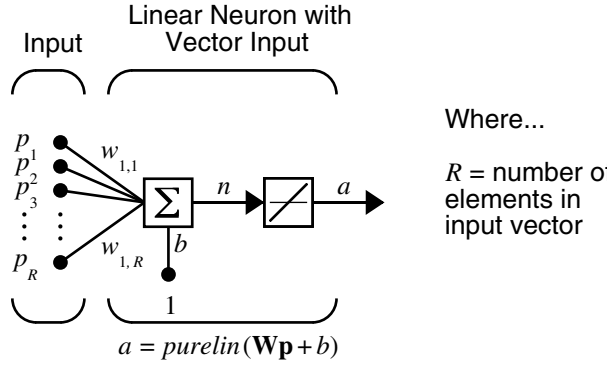
This chapter introduces `newlin`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

You can type `help linnet` to see a list of linear network functions, demonstrations, and applications.

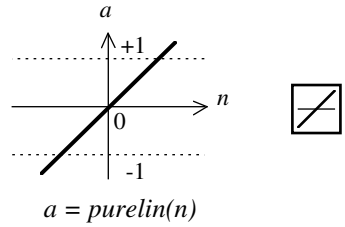
The use of linear filters in adaptive systems is discussed in Chapter 10, "Adaptive Filters and Adaptive Training."

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

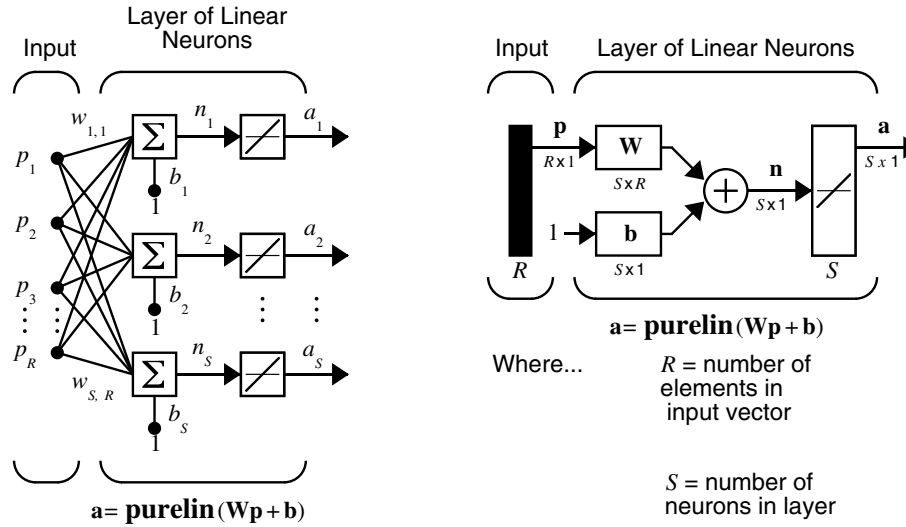
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

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 **W**.

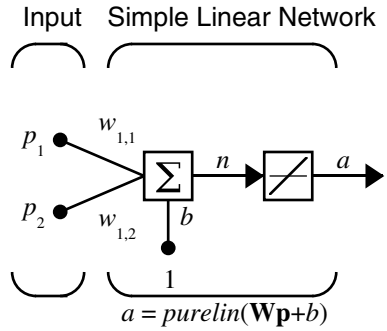


Note that the figure on the right defines an S-length output vector **a**.

A single-layer linear network is shown. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Creating a Linear Neuron (newlin)

Consider a single linear neuron with two inputs. The following figure shows the diagram for this network.



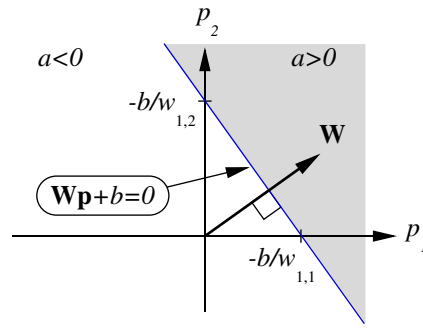
The weight matrix **W** in this case has only one row. The network output is

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

or

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary, as shown below (adapted with thanks from [HDB96]).



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 the following command, which specifies typical input vectors of [-1; -1] and [1; 1] and typical outputs of [-1 1]. (These values are arbitrary. For a real problem, use real values.)

```
net = newlin([-1 1; -1 1],[-1 1]);
```

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 = sim(net,p)
a =
    24
```

To summarize, you can create a linear network with `newlin`, adjust its elements as you want, and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96].

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 = sim(net,P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

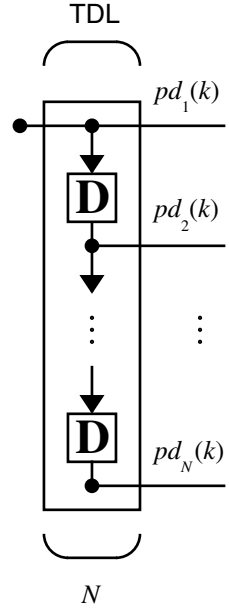
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

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 4-10. First, however, delays must be discussed.

Linear Networks with Delays

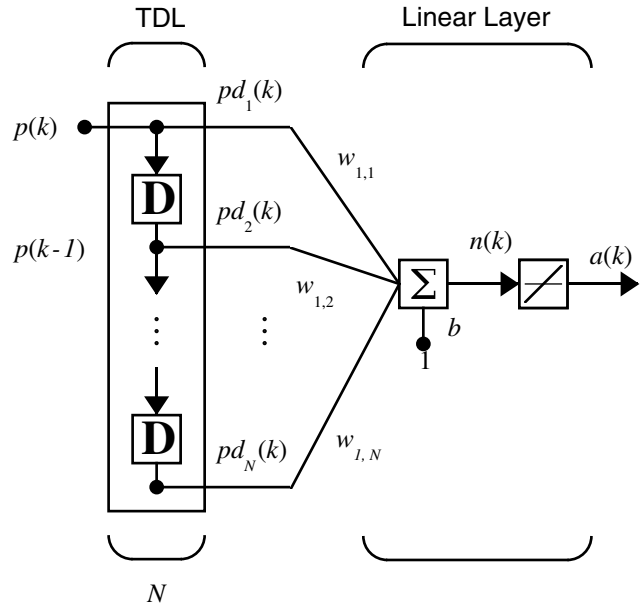
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left and passes through N-1 delays. The output of the tapped delay line (TDL) is an N-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Linear Filter

You can combine a tapped delay line with a linear network to create the linear *filter shown*.



The output of the filter is given by

$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i}a(k-i+1) + b$$

The network shown is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85]. 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 = sim(net,P,Pi)
```

to give

```
Y = [2.7297] [10.5405] [5.0090] [14.9550] [10.7838] [5.9820]
```

As you can see, the network outputs are not exactly equal to the targets, but they are close and the mean square error is minimized.

LMS Algorithm (learnwh)

The LMS algorithm, or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If you take the partial derivative of the squared error with respect to the weights and biases at the kth iteration, you have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1, 2, \ldots, R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - (\mathbf{W}\mathbf{p}(k) + b)]$$

or

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right]$$

Here $p_i(k)$ is the ith element of the input vector at the kth iteration.

This can be simplified to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

and

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, change the weight matrix, and the bias will be

$$2\alpha e(k)\mathbf{p}(k)$$

and

$$2\alpha e(k)$$

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors, and α is a *learning rate*. If α is large, learning occurs quickly, but if it is too large it can lead to instability and errors might even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix $\mathbf{p}^T\mathbf{p}$ of the input vectors.

You might want to read some of Chapter 10 of [HDB96] 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`.

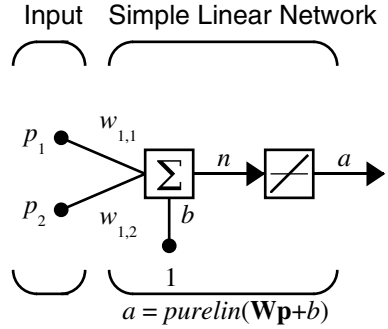
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

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`, discussed in Chapter 10, "Adaptive Filters and Adaptive Training," 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 classification problem presented in "Linear Filters" on page 4-1.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

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 = newlin(P,T);
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 = sim(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 4-18 for examples of various limitations.

This demonstration program, `demolin2`, shows the training of a linear neuron and plots the weight trajectory and error during training.

You might also try running the demonstration 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 demonstration `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. Demonstration `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.

5

Backpropagation

Introduction (p. 5-2)
Architecture (p. 5-8)
Faster Training (p. 5-19)
Speed and Memory Comparison (p. 5-34)
Improving Generalization (p. 5-52)
Preprocessing and Postprocessing (p. 5-61)
Sample Training Session (p. 5-68)
Limitations and Cautions (p. 5-71)

Introduction

Backpropagation is the generalization of the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term *backpropagation* refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods. The Neural Network Toolbox™ software implements a number of these variations. This chapter explains how to use each of these routines and discusses the advantages and disadvantages of each.

Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. There are two features of Neural Network Toolbox software that are designed to improve network generalization: regularization and early stopping. These features and their use are discussed in “Improving Generalization” on page 5-52.

This chapter also discusses preprocessing and postprocessing techniques, which can improve the efficiency of network training, in “Preprocessing and Postprocessing” on page 5-61.

Before beginning this chapter you may want to read a basic reference on backpropagation, such as D.E Rumelhart, G.E. Hinton, and R.J. Williams, “Learning internal representations by error propagation,” D.E. Rumelhart and J. McClelland, editors, *Parallel Data Processing*, Vol.1, Chapter 8, The M.I.T. Press, Cambridge, MA, 1986, pp. 318–362. This subject is also covered in detail in Chapters 11 and 12 of M.T. Hagan, H.B. Demuth, and M.H. Beale, *Neural*

Network Design, ISBN 0-9717321-0-8 (available from John Stovall, `john.stovall@colorado.edu`, 303.492.3648).

The primary objective of this chapter is to explain how to use the backpropagation training functions in the toolbox to train feedforward neural networks to solve specific problems. There are generally four steps in the training process:

1 Assemble the training data.

2 Create the network object.

3 Train the network.

4 Simulate the network response to new inputs.

This chapter discusses a number of different training functions, but using each function generally follows these four steps.

The next section, “Architecture,” describes the basic feedforward network structure and demonstrates how to create a feedforward network object. Then the simulation and training of the network objects are presented.

Solving a Problem

This section demonstrates the common steps of solving a problem with backpropagation.

The first step is to define your problem. For supervised networks, such as feed-forward networks trained with backpropagation, this means a set of input vectors and a set of associated desired output vectors called *target vectors*.

The file `housing.mat` contains a predefined set of input and target vectors. The input vectors define data regarding real-estate properties and the target values define relative values of the properties. Load the data using the following command:

```
load house_dataset
```

Loading this file creates two variables. The input matrix `houseInputs` consists of 506 column vectors of 13 real estate variables for 506 different properties. The target matrix `houseTargets` consists of the corresponding 506 relative valuations.

The next step is to create a network and train it until it has learned the relationship between the example inputs and targets.

The most common network used with backpropagation is the two-layer feed-forward network. The following call to `newff` creates a two-layer network with 20 neurons in the hidden layer. (The number of neurons in the output layer are automatically set to one, the number of elements in each vector of `t`.)

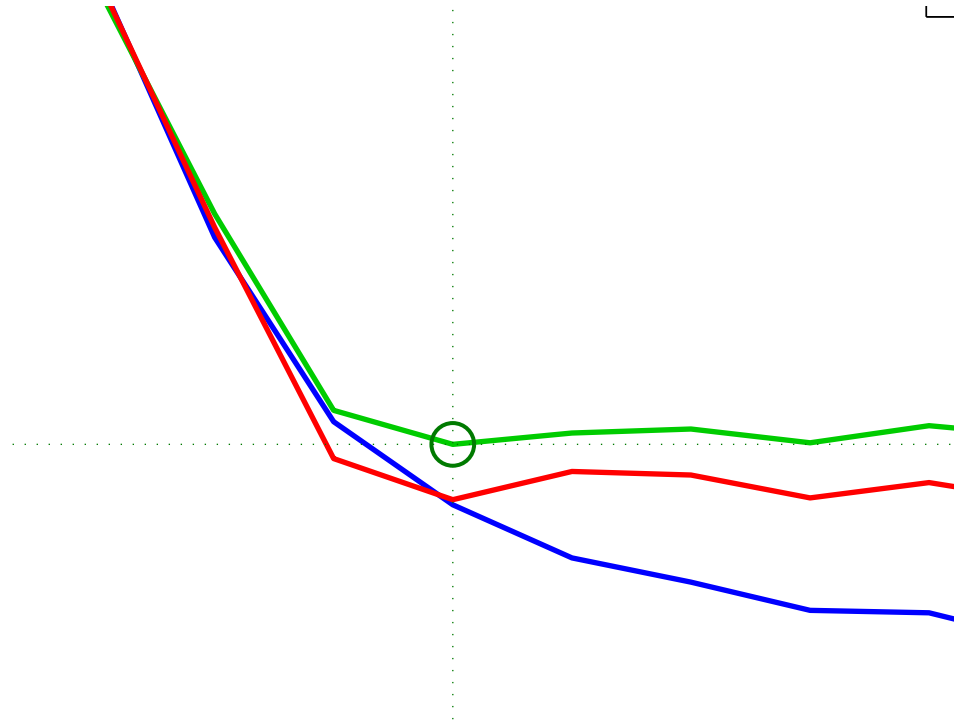
```
net = newff(houseInputs,houseTargets,20);
```

Two-layer feed-forward networks can potentially represent any input-output relationship with a finite number of discontinuities (which is a typical relationship you might want to model), assuming that there are enough neurons in the hidden layer.

Your next step is to train the network using the data.

```
net = train(net,houseInputs,houseTargets);
```

Click the **Performance** plot button in the training window to see a plot that resembles the following figure.



The plot shows the mean squared error of the network starting at a large value and decreasing to a smaller value. In other words, it shows that the network is learning.

The plot has three lines, because the 506 input and targets vectors are randomly divided into three sets. 60% of the vectors are used to train the network. 20% of the vectors are used to validate how well the network generalized. Training on the training vectors continues as long the training reduces the network's error on the validation vectors. After the network memorizes the training set (at the expense of generalizing more poorly), training is stopped. This technique automatically avoids the problem of overfitting, which plagues many optimization and learning algorithms.

Finally, the last 20% of the vectors provide an independent test of network generalization to data that the network has never seen.

After training the network, you can use it. Use `sim` to apply the network to the original vectors:

```
y = sim(net,p);
```

You can now apply the network to similar data. Assuming that the data used to train the network represents the general problem of pricing properties, you can expect the accuracy of the network for new data to be similar to the accuracy for test data during training.

Improving Results

The `house_dataset` example demonstrated some simple commands you can use to solve many types of problems. However, if your first attempt does not meet your needs or expectations, this section describes some ways to improve network accuracy.

If the network is not sufficiently accurate, you can try initializing the network and the training again. Each time your initialize a feed-forward network, the network parameters are different and might produce different solutions.

```
net = init(net);
net = train(net,houseInput,houseTargets);
```

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.)

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.

Under the Hood

The remainder of this chapter describes additional techniques for improving network performance. These advanced techniques require you to know how neural networks transform inputs into outputs and how backpropagation training works.

First, the feed-forward network architecture is introduced.

Networks that you train using backpropagation can have more than two hidden layers, which can make learning complex relationships easier for the network. Other architectures add more connections, which might help networks learn.

The default backpropagation training algorithm is Levenberg-Marquardt (`trainlm`). This is the fastest method in the toolbox, but it can use large amounts of memory. There are ways to reduce this memory requirement, and there are other algorithms that require less memory and might improve generalization.

Another way you might improve generalization is by modifying the default method that divides the data into training data, validation data, and test data. You can replace the default function `dividerand` by other methods, or you can change the relative percentages of vectors associated with `dividerand` to values other than 60%, 20% and 20%.

By default, networks trained with backpropagation have three input processing functions that automatically apply to network input data by `sim` or `train`. The first function `fixunknowns` re-encodes unknown input values represented by `NaN` values into numerical values so the network can operate on the values directly. The second function `removeconstantrows` removes inputs with values that are the same for all input vectors used to create the network. Such inputs contain no information and can cause numerical problems during network initialization or training. The third function `mapminmax` maps the range of input values to the range [-1 1], or normalizes the input values. Training is often faster when values are normalized.

`removeconstantrows` and `mapminmax` are also the default output processing functions.

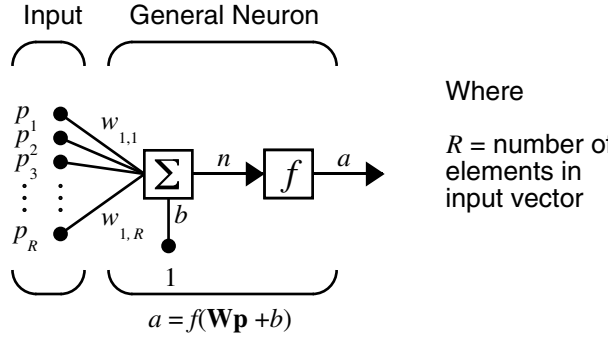
There are other input/output processing functions that can be used in addition to or instead of the default functions for particular kinds of problems.

Architecture

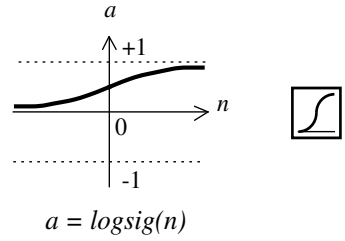
This section presents the architecture of the network that is most commonly used with the backpropagation algorithm—the multilayer feedforward network.

Neuron Model (logsig, tansig, purelin)

An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function f. Neurons can use any differentiable transfer function f to generate their output.



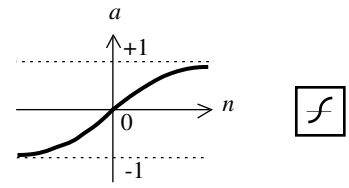
Multilayer networks often use the log-sigmoid transfer function `logsig`.



Log-Sigmoid Transfer Function

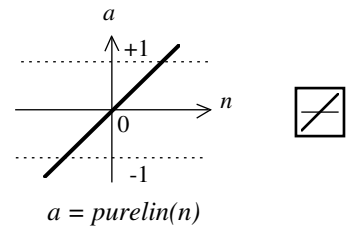
The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks can use the tan-sigmoid transfer function `tansig`.



Tan-Sigmoid Transfer Function

Occasionally, the linear transfer function `purelin` is used in backpropagation networks.



Linear Transfer Function

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

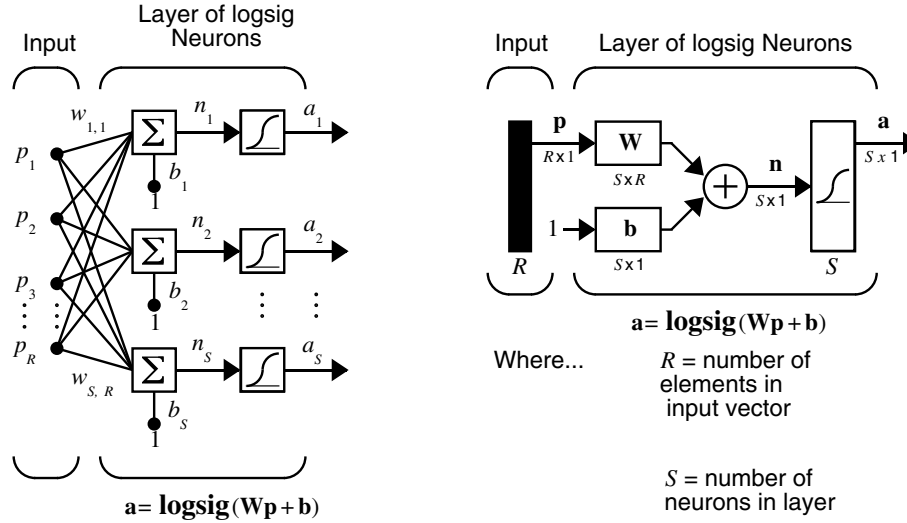
In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, `logsig`, `tansig`, and `purelin`, can be called to calculate its own derivative. To calculate a transfer function's derivative, call the transfer function with the string `'dn'`.

```
dn = tansig('dn',n,a)
```

The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired. See Chapter 12, "Advanced Topics."

Feedforward Network

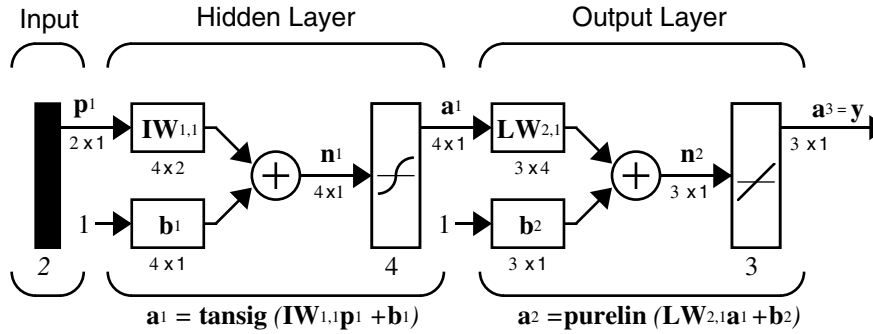
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



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 and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range –1 to +1.

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`).

As noted in Chapter 2, "Neuron Model and Network Architectures," for multiple-layer networks the number of layers determines the superscript on the weight matrices. 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.

Creating a Network (newff)

The first step in training a feedforward network is to create the network object. The function `newff` creates a feedforward network. It requires three arguments and returns the network object. The first argument is a matrix of sample R-element input vectors. The second argument is a matrix of sample S-element target vectors. The sample inputs and outputs are used to set up network input and output dimensions and parameters. The third argument is an array containing the sizes of each hidden layer. (The output layer size is determined from the targets.)

More optional arguments can be provided. For instance, the fourth argument is a cell array containing the names of the transfer functions to be used in each layer. The fifth argument contains the name of the training function to be used. If only three arguments are supplied, the default transfer function for hidden layers is `tansig` and the default for the output layer is `purelin`. The default training function is `trainlm`.

For example, the following commands create a two-layer network. To create a network, you provide typical input and output values that initialize weight and bias values and determine the size of the output layer. Assume three input vectors with two elements having values [-1;0], [2;5] and [1; 1]. Assume the output vector to have a single element shown below as t. (These are arbitrary numbers. For a real problem, use real values.)

```
p = [-1 2 1; 0 5 1];
t = [-10 0 10];
```

There are three neurons in one hidden layer. The default transfer functions for hidden layers is tan-sigmoid, and for the output layer is linear.

```
net = newff(p,t,3);
```

This command creates 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. The next section explains the details of the initialization process.

Other Architectures for Backpropagation

While two-layer feed-forward networks can potentially learn virtually any input-output relationship, feed-forward networks with more layers might learn complex relationships more quickly.

The function `newcf` creates cascade-forward networks. These are similar to feed-forward 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 layers 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.

Other networks can learn dynamic or time-series relationships. They are introduced in Chapter 6, “Dynamic Networks”.

Initializing Weights (init)

Before training a feedforward network, you must initialize the weights and biases. The `newff` 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);
```

For specifics on how the weights are initialized, see Chapter 12, “Advanced Topics.”

Simulation (sim)

The function `sim` simulates a network. `sim` takes the network input `p` and the network object `net` and returns the network outputs `a`. You can use `sim` to simulate the network created above for a single input vector:

```
p = [1;2];
a = sim(net,p)
a =
    -3.6686
```

(If you try these commands, your output might be different, depending on the state of your random number generator when the network was initialized.) Below, `sim` is called to calculate the outputs for a concurrent set of three input vectors. 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.

```
p = [1 3 2;2 4 1];
a = sim(net,p)
a =
  -3.6686 -0.9030 -4.3257
```

Training

Once the network weights and biases are initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior—network inputs `p` and target outputs `t`. During training the weights and biases of the network are iteratively adjusted to minimize 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`.

The remainder of this chapter describes several different training algorithms for feedforward networks. All these algorithms use the gradient of the performance function to determine how to adjust the weights to minimize performance. The gradient is determined using a technique called backpropagation, which involves performing computations backward through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapter 11 of [HDB96].

The basic backpropagation training algorithm, in which the weights are moved in the direction of the negative gradient, is described in the next section. Later sections describe more complex algorithms that increase the speed of convergence.

Backpropagation Algorithm

There are many variations of the backpropagation algorithm, several of which are described in this chapter. The simplest implementation of backpropagation learning 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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

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.

There are two different ways in which this gradient descent algorithm 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 are applied to the network before

the weights are updated. The next section describes the batch mode of training; incremental training is discussed in a later chapter.

Batch Training (train)

In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases. For a discussion of batch training with the backpropagation algorithm, see page 12-7 of [HDB96].

Batch Gradient Descent (traingd)

The batch steepest descent training function is `traingd`. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network `trainFcn` to `traingd`, and then call the function `train`. There is only one training function associated with a given network.

There are seven training parameters associated with `traingd`:

- `epochs`
- `show`
- `goal`
- `time`
- `min_grad`
- `max_fail`
- `lr`

The learning rate `lr` is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. See page 12–8 of [HDB96] for a discussion of the choice of learning rate.

The training status is displayed for every `show` iterations of the algorithm. (If `show` is set to `NaN`, then the training status is never displayed.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds `epochs`, if the performance function drops below `goal`, if the magnitude of the gradient is less than `mingrad`, or if the training

time is longer than `time` seconds. `max_fail`, which is associated with the early stopping technique, is discussed in "Improving Generalization" on page 5-52.

The following code creates a training set of inputs `p` and targets `t`. For batch training, all the input vectors are placed in one matrix.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
```

Create the feedforward network.

```
net = newff(p,t,3,{},'traingd');
```

In this simple example, turn off a feature that is introduced later.

```
net.divideFcn = '';
```

At this point, you might want to modify some of the default training parameters.

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

If you want to use the default training parameters, the preceding commands are not necessary.

Now you are ready to train the network.

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

The training record `tr` contains information about the progress of training. An example of its use is given in "Sample Training Session" on page 5-68.

Now you can simulate the trained network to obtain its response to the inputs in the training set.

```
a = sim(net,p)
a =
   -1.0026   -0.9962    1.0010    0.9960
```

Try the *Neural Network Design* demonstration `nnd12sd1`[HDB96] for an illustration of the performance of the batch gradient descent algorithm.

Gradient Descent with Momentum

In addition to `traingd`, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by traingdm, allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12–9 of [HDB96] for a discussion of momentum.

Gradient descent with momentum depends on two training parameters. The parameter `lr` indicates the learning rate, similar to the simple gradient descent. The parameter `mc` is the momentum constant that defines the amount of momentum. `mc` is set between 0 (no momentum) and values close to 1 (lots of momentum). A momentum constant of 1 results in a network that is completely insensitive to the local gradient and, therefore, does not learn properly.)

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traingdm');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net = train(net,p,t);
y = sim(net,p)
```

Try the *Neural Network Design* demonstration `nnd12mo` [HDB96] for an illustration of the performance of the batch momentum algorithm.

Faster Training

The previous section presented two backpropagation training algorithms: gradient descent, and gradient descent with momentum. These two methods are often too slow for practical problems. This section discusses several high-performance algorithms that can converge from ten to one hundred times faster than the algorithms discussed previously. All the algorithms in this section operate in batch mode and are invoked using `train`.

These faster algorithms fall into two categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. One heuristic modification is the momentum technique, which was presented in the previous section. This section discusses two more heuristic techniques: variable learning rate backpropagation, `traingda`, and resilient backpropagation, `trainrp`.

The second category of fast algorithms uses standard numerical optimization techniques. (See Chapter 9 of [HDB96] for a review of basic numerical optimization.) Later sections present three types of numerical optimization techniques for neural network training:

Conjugate gradient (`traincgf`, `traincgp`, `traincgb`, `trainscg`)	"Conjugate Gradient Algorithms" on page 5-22
Quasi-Newton (`trainbfg`, `trainoss`)	"Quasi-Newton Algorithms" on page 5-29
Levenberg-Marquardt (`trainlm`)	"Levenberg-Marquardt (trainlm)" on page 5-30

Variable Learning Rate (traingda, traingdx)

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

You can improve the performance of the steepest descent algorithm if you allow the learning rate to change during the training process. An adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by `traingd`. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, `max_perf_inc` (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by `lr_dec` = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by `lr_inc` = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it is decreased until stable learning resumes.

Try the *Neural Network Design* demonstration `nnd12vl` [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function `traingda`, which is called just like `traingd`, except for the additional training parameters `max_perf_inc`, `lr_dec`, and `lr_inc`. Here is how it is called to train the previous two-layer network:

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traingda');
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net = train(net,p,t);
y = sim(net,p)
```

The function traingdx combines adaptive learning rate with momentum training. It is invoked in the same way as traingda, except that it has the momentum coefficient mc as an additional training parameter.

Resilient Backpropagation (trainrp)

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called "squashing" functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative can determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value remains the same. Whenever the weights are oscillating, the weight change is reduced. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases. A complete description of the Rprop algorithm is given in [ReBr93].

The following code recreates the previous network and trains it using the Rprop algorithm. The training parameters for trainrp are epochs, show, goal, time, min_grad, max_fail, delt_inc, delt_dec, delta0, and deltamax. The first eight parameters have been previously discussed. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, the training parameters are left at the default values:

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainrp');
net = train(net,p,t);
```

```
y = sim(net,p)
```

`rprop` is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. You do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

Conjugate Gradient Algorithms

The basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient), the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. This section presents four variations of conjugate gradient algorithms.

See page 12-14 of [HDB96] for a discussion of conjugate gradient algorithms and their application to neural networks.

In most of the training algorithms discussed up to this point, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size that minimizes the performance function along that line. There are five different search functions included in the toolbox, and these are discussed in "Line Search Routines" on page 5-26. Any of these search functions can be used interchangeably with a variety of the training functions described in the remainder of this chapter. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application. An appropriate default search function is assigned to each training function, but you can modify this.

Fletcher-Reeves Update (traincgf)

All the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of the conjugate gradient algorithm are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

The following code reinitializes the previous network and retrains it using the Fletcher-Reeves version of the conjugate gradient algorithm. The training parameters for `traincgf` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `srchFcn`, `scal_tol`, `alpha`, `beta`, `delta`, `gama`, `low_lim`, `up_lim`, `maxstep`, `minstep`, and `bmax`. The first six parameters have been previously discussed. `srchFcn` is the name of the line search function. It can be any of the functions described in "Line Search Routines" on page 5-26 (or a user-supplied function). The remaining parameters are associated with specific line search routines and are described later in this section. The default line search routine `srchcha` is used in this example. `traincgf` generally converges in fewer iterations than `trainrp` (although there is more computation required in each iteration).

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traincgf');
net = train(net,p,t);
y = sim(net,p)
```

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than `trainrp`,

although the results vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms. Therefore, these algorithms are good for networks with a large number of weights.

Try the *Neural Network Design* demonstration `nnd12cg` [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

Polak-Ribiére Update (traincgp)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

The following code recreates the previous network and trains it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for `traincgp` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epochs` are set to the same values as they were for `traincgf`.

```
net=newff(p,t,3,{},'traincgp');
[net,tr]=train(net,p,t);
```

The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

Powell-Beale Restarts (traincgb)

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. This technique restarts if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

$$\left|\mathbf{g}_{k-1}^{T}\mathbf{g}_{k}\right| \geq 0.2\left\|\mathbf{g}_{k}\right\|^{2}$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The following code recreates the previous network and trains it using the Powell-Beale version of the conjugate gradient algorithm. The training parameters for `traincgb` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traincgb');
net = train(net,p,t);
y = sim(net,p)
```

The `traincgb` routine has somewhat better performance than `traincgp` for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

Scaled Conjugate Gradient (trainscg)

Each of the conjugate gradient algorithms discussed so far requires a line search at each iteration. This line search is computationally expensive, because it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller [Moll93], was designed to avoid the time-consuming line search. This algorithm combines the model-trust region approach (used in the Levenberg-Marquardt algorithm, described in “Levenberg-Marquardt (trainlm)” on page 5-30), with the conjugate gradient approach. See {Moll93] for a detailed explanation of the algorithm.

The following code reinitializes the previous network and retrains it using the scaled conjugate gradient algorithm. The training parameters for `trainscg` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `sigma`, and `lambda`. The first six parameters have been discussed previously. The parameter `sigma` determines the change in the weight for the second derivative approximation. The parameter `lambda` regulates the indefiniteness of the Hessian.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainscg');
net = train(net,p,t);
y = sim(net,p)
```

The `trainscg` routine can require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of Fletcher-Reeves.

Line Search Routines

Several of the conjugate gradient and quasi-Newton algorithms require that a line search be performed. This section describes five different line searches you can use. To use any of these search routines, you simply set the training parameter `srchFcn` equal to the name of the desired search function, as described in previous sections. It is often difficult to predict which of these routines provides the best results for any given problem, but the default search function is set to an appropriate initial choice for each training function, so you never need to modify this parameter.

Golden Section Search (srchgol)

The golden section search `srchgol` is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance function occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determine a section of the interval that can be discarded, and a new interior point is placed within the new interval. This

procedure is continued until the interval of uncertainty is reduced to a width of `tol`, which is equal to `delta/scale_tol`.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the *Neural Network Design* demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.

Brent's Search (srchbre)

Brent's search is a linear search that is a hybrid of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate that is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods can take many iterations to become apparent. Brent's search attempts to combine the best features of both approaches.

For Brent's search, you begin with the same interval of uncertainty used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.

See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm can require more performance evaluations than algorithms that use derivative information.

Hybrid Bisection-Cubic Search (srchhyb)

Like Brent's search, `srchhyb` is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty, and the performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is obtained by using the value of the performance and its derivative at the two

endpoints. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent's algorithm.

Charalambous' Search (srchcha)

The method of Charalambous, `srchcha`, was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like the previous two methods, it is a hybrid search. It uses a cubic interpolation together with a type of sectioning.

See [Char92] for a description of Charalambous' search. This routine is used as the default search for most of the conjugate gradient algorithms because it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with fewer steps. This is not true for all problems, and you might want to experiment with other line searches.

Backtracking (srchbac)

The backtracking search routine `srchbac` is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and a step multiplier of 1. It also uses the value of the derivative of performance at the current point to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in [DeSc83]. It is used as the default line search for the quasi-Newton algorithms, although it might not be the best technique for all problems.

Quasi-Newton Algorithms

BFGS Algorithm (trainbfg)

Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where $\mathbf{A}_k^{-1}$ is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm is implemented in the `trainbfg` routine.

The following code reinitializes the previous network and retrains it using the BFGS quasi-Newton algorithm. The training parameters for `trainbfg` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainbfg');
net = train(net,p,t);
y = sim(net,p)
```

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is $n \times n$, where n is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

One Step Secant Algorithm (trainoss)

Because the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

The following code reinitializes the previous network and retrains it using the one-step secant algorithm. The training parameters for `trainoss` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epochs` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainoss');
net = train(net,p,t);
y = sim(net,p)
```

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

Levenberg-Marquardt (trainlm)

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

where **J** is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and **e** is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The following code reinitializes the previous network and retrains it using the Levenberg-Marquardt algorithm. The training parameters for `trainlm` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `mu`, `mu_dec`, `mu_inc`, `mu_max`, and `mem_reduc`. The first six parameters were discussed earlier. The parameter `mu` is the initial value for μ. This value is multiplied by `mu_dec` whenever the performance function is reduced by a step. It is multiplied by `mu_inc` whenever a step would increase the performance function. If `mu` becomes larger than `mu_max`, the algorithm is stopped. The parameter `mem_reduc` is used to control the amount of memory used by the algorithm. It is discussed in the next section.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainlm');
net = train(net,p,t);
y = sim(net,p)
```

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an

efficient implementation in MATLAB® software, because the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB environment.

Try the *Neural Network Design* demonstration `nnd12m` [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

Reduced Memory Levenberg-Marquardt (trainlm)

The main drawback of the Levenberg-Marquardt algorithm is that it requires the storage of some matrices that can be quite large for certain problems. The size of the Jacobian matrix is $Q \times n$, where Q is the number of training sets and n is the number of weights and biases in the network. It turns out that this matrix does not have to be computed and stored as a whole. For example, if you were to divide the Jacobian into two equal submatrices you could compute the approximate Hessian matrix as follows:

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} = \begin{bmatrix}\mathbf{J}_1^T & \mathbf{J}_2^T\end{bmatrix}\begin{bmatrix}\mathbf{J}_1 \\ \mathbf{J}_2\end{bmatrix} = \mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{J}_2$$

Therefore, the full Jacobian does not have to exist at one time. You can compute the approximate Hessian by summing a series of subterms. Once one subterm has been computed, the corresponding submatrix of the Jacobian can be cleared.

When you use the training function `trainlm`, the parameter `mem_reduc` determines how many rows of the Jacobian are to be computed in each submatrix. If `mem_reduc` is set to 1, then the full Jacobian is computed, and no memory reduction is achieved. If `mem_reduc` is set to 2, then only half of the Jacobian is computed at one time. This saves half the memory used by the calculation of the full Jacobian.

There is a drawback to using memory reduction. A significant computational overhead is associated with computing the Jacobian in submatrices. If you have enough memory available, then it is better to set `mem_reduc` to 1 and to compute the full Jacobian. If you have a large training set, and you are running out of memory, then you should set `mem_reduc` to 2 and try again. If you still run out of memory, continue to increase `mem_reduc`.

Even if you use memory reduction, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions $n \times n$.

If your network is very large, then you might run out of memory. If this is the case, try `trainscg`, `trainrp`, or one of the conjugate gradient algorithms.

Speed and Memory Comparison

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	
LM	`trainlm`	Levenberg-Marquardt
BFG	`trainbfg`	BFGS Quasi-Newton
RP	`trainrp`	Resilient Backpropagation
SCG	`trainscg`	Scaled Conjugate Gradient
CGB	`traincgb`	Conjugate Gradient with Powell/Beale Restarts
CGF	`traincgf`	Fletcher-Powell Conjugate Gradient
CGP	`traincgp`	Polak-Ribiére Conjugate Gradient
OSS	`trainoss`	One Step Secant
GDX	`traingdx`	Variable Learning Rate Backpropagation

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

Problem Title	Problem Type	Network Structure	Error Goal	Computer
SIN	Function approximation	1-5-1	0.002	Sun Sparc 2
PARITY	Pattern recognition	3-10-10-1	0.001	Sun Sparc 2
ENGINE	Function approximation	2-30-2	0.005	Sun Enterprise 4000
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

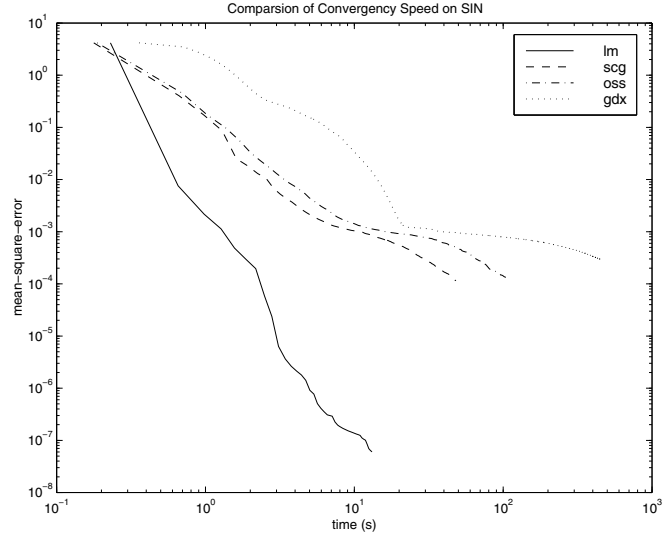
SIN Data Set

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

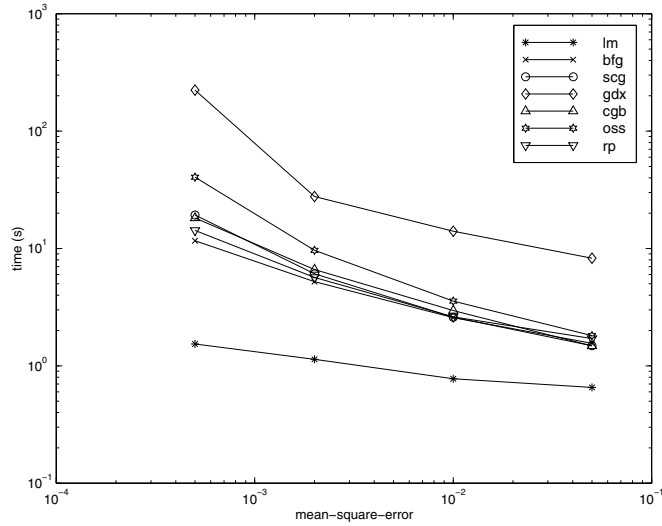
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
SCG	6.09	5.34	3.18	23.64	3.81
CGB	6.61	5.80	2.99	23.65	3.67
CGF	7.86	6.89	3.57	31.23	4.76
CGP	8.24	7.23	4.07	32.32	5.03
OSS	9.64	8.46	3.97	59.63	9.79
GDX	27.69	24.29	17.21	258.15	43.65

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is demonstrated 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).



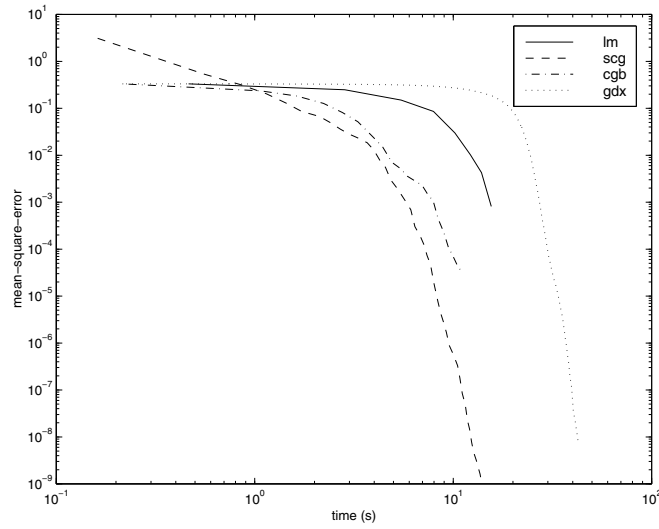
PARITY Data Set

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

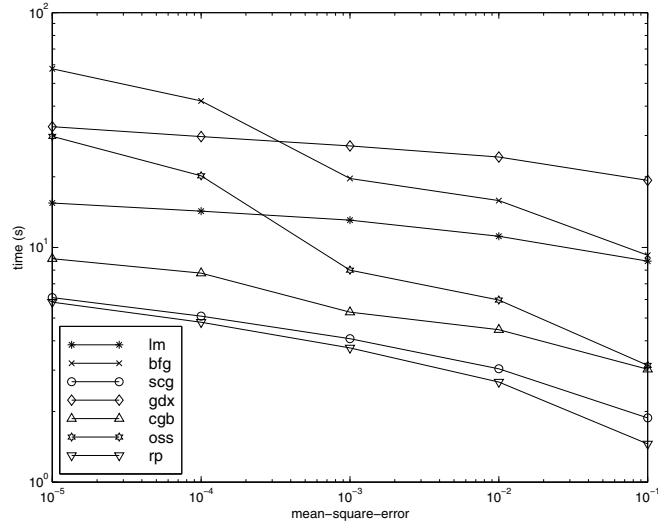
recognition problems are generally saturated, you will not be operating in the linear region.

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 demonstrated 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).

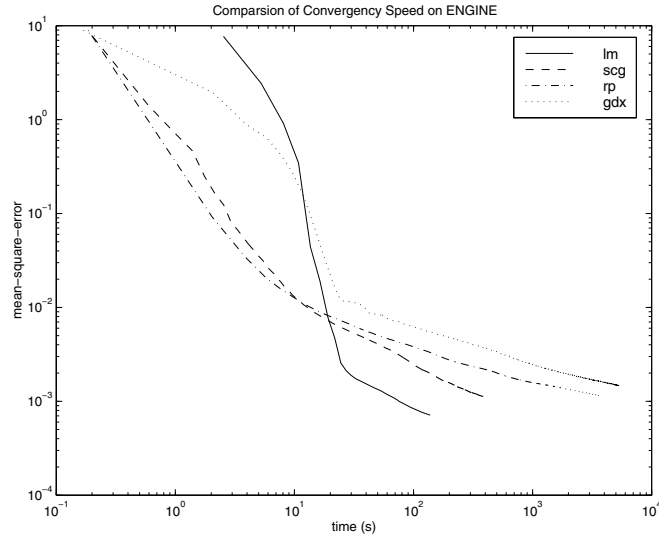


ENGINE Data Set

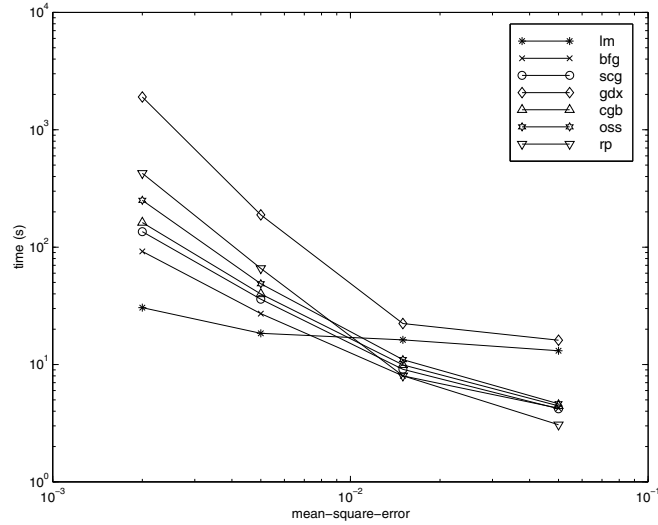
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem 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

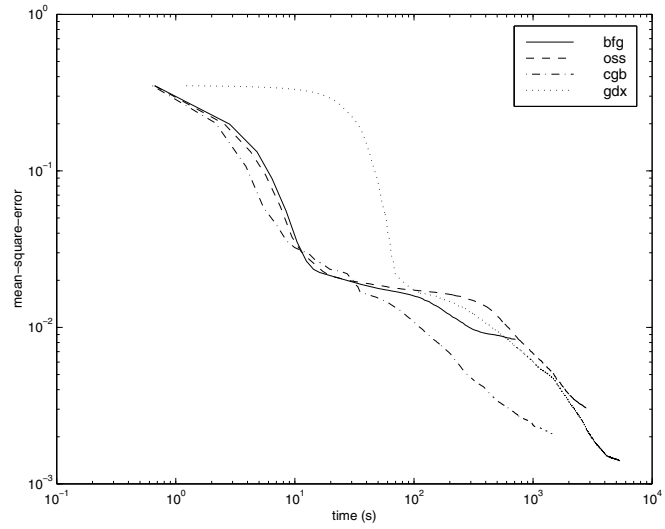
The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity 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

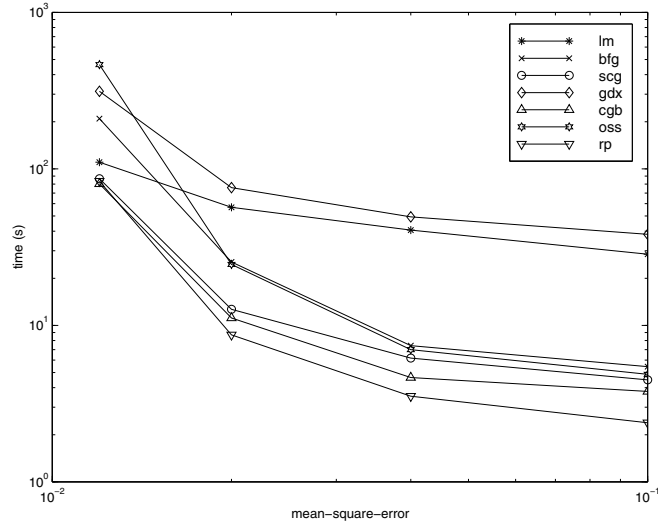
parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

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.



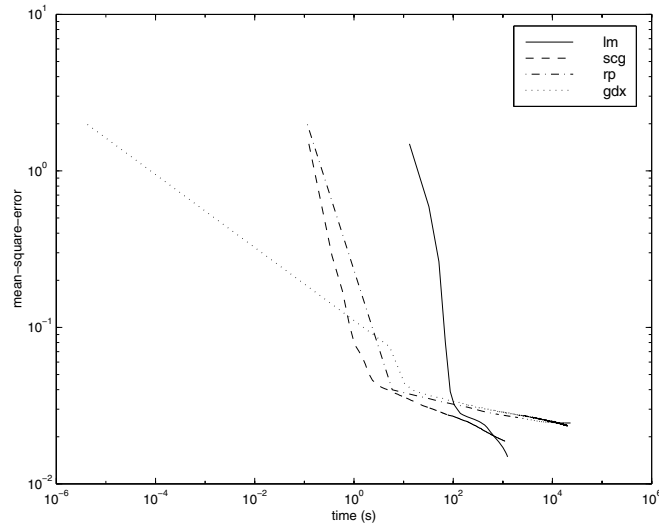
CHOLESTEROL Data Set

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]. 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.

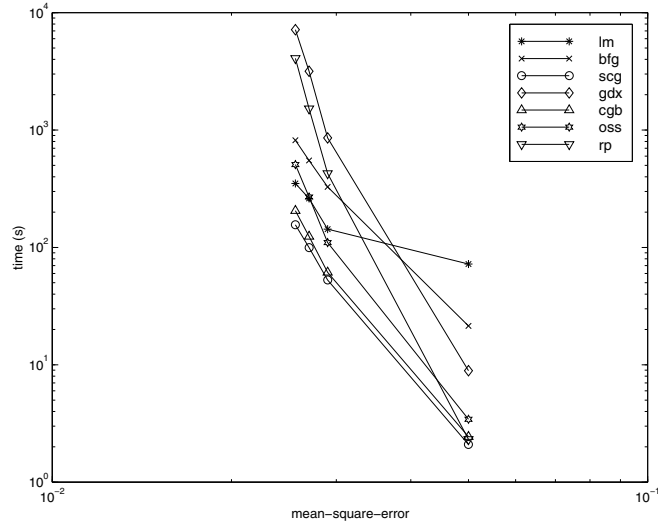
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases 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	1.24	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

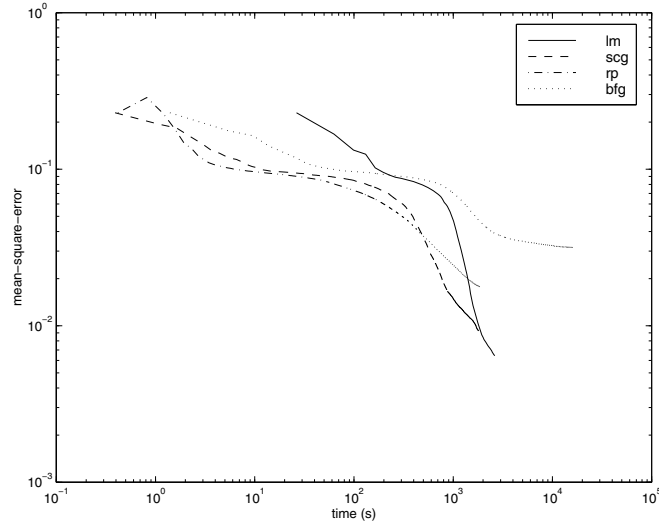
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance 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.

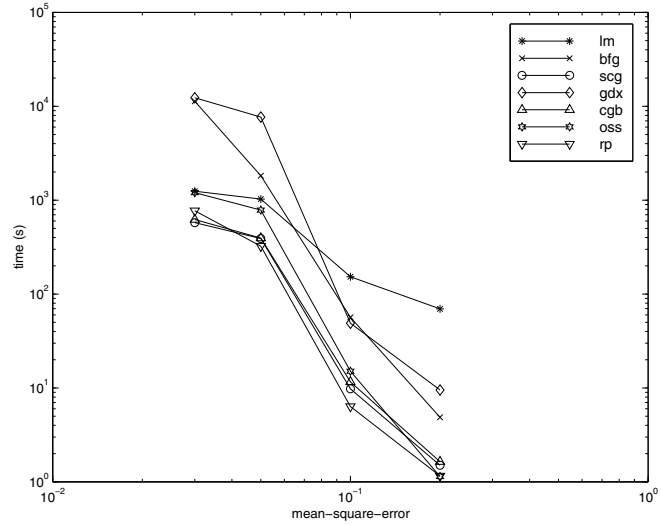
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	323.90	1.00	187.43	576.90	111.37
SCG	390.53	1.21	267.99	487.17	75.07
CGB	394.67	1.22	312.25	558.21	85.38
CGP	415.90	1.28	320.62	614.62	94.77
OSS	784.00	2.42	706.89	936.52	76.37
CGF	784.50	2.42	629.42	1082.20	144.63
LM	1028.10	3.17	802.01	1269.50	166.31

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
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. By adjusting the `mem_reduc` parameter, discussed earlier, the storage requirements can be reduced, but at the cost of increased execution time.

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.

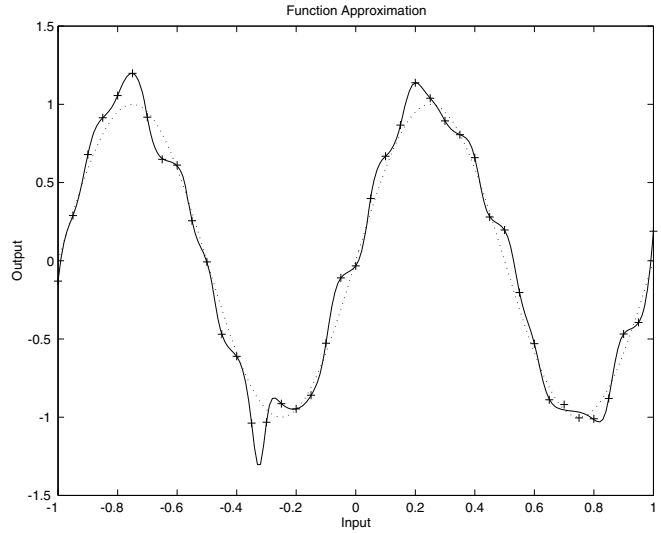
The performance of `trainbfg` is similar to that of `trainlm`. It does not require as much storage as `trainlm`, but the computation required does increase geometrically with the size of the network, because the equivalent of a matrix inverse must be computed at each iteration.

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 (as described in the next section) 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.

Improving Generalization

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* demonstration `nnd11gn` [HDB96] 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 Neural Network 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.

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 `newff`.

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

$$msereg = \gamma mse + (1-\gamma)msw$$

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. (Data division is cancelled by setting `net.divideFcn` so that the effects of `msereg` are isolated from early stopping.)

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(p,t,3,{},'trainbfg');
net.divideFcn = '';
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,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]. 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].

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 on page 5-52. (Data division is cancelled by setting `net.divideFcn` so that the effects of `trainbr` are isolated from early stopping.)

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net=newff(p,t,20,{},'trainbr');
net.divideFcn ='';
net.trainParam.show = 10;
net.trainParam.epochs = 50;
randn('seed',192736547);
net = init(net);
[net,tr]=train(net,p,t);
```

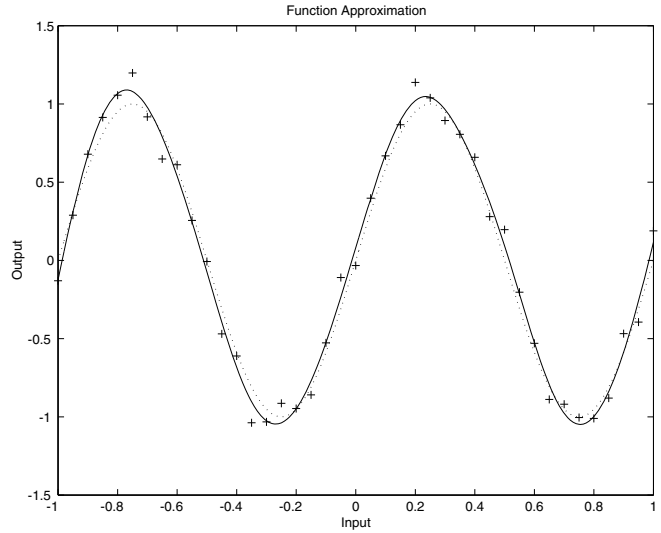
One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by `#Par` in the printout) out of the 61 total weights and biases 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 "Preprocessing and Postprocessing" on page 5-61.

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 `trainrp` 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.

Preprocessing and Postprocessing

Neural network training can be made 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.

Network-input processing functions transform inputs into a better form for the network use. Processing functions associated with a network output transform targets into a better form for network training, and reverse transformed outputs back to the characteristics of the original target data.

Most of the network creation functions in the toolbox, including the backpropagation network creation functions such as `newff`, automatically assign processing function to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for the network.

You can override the default input and output processing functions when you call a network creation function, or 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
```

To view the processing functions returned by the output of a two-layer network, access this network property:

```
net.outputs{2}.processFcns
```

You can use these properties to change the processing functions you want your network to apply to the inputs and outputs. However, we recommend that you use the defaults.

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 ith output processing function for the network output associated with the second layer, as follows:

```
net.outputs{2}.processParams{i}
```

For backpropagation network creation functions, such as newff, the default functions are fixunknowns, removeconstantrows and mapminmax. For outputs, the default processing functions are removeconstantrows and mapminmax.

Min and Max (mapminmax)

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function mapminmax scales inputs and targets so that they fall in the range [-1,1]. The following code illustrates how to use this function.

```
[pn,ps] = mapminmax(p);
[tn,ts] = mapminmax(t);
net = train(net,pn,tn);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets pn and tn that are returned will all fall in the interval [-1,1]. The structures ps and ts contain the settings, in this case the minimum and maximum values of the original inputs and targets. After the network has been trained, the ps settings should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If mapminmax is used to scale the targets, then the output of the network will be trained to produce outputs in the range [-1,1]. To convert these outputs back into the same units that were used for the original targets, use the settings ts. The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.

```
an = sim(net,pn);
a = mapminmax('reverse',an,ts);
```

The network output an corresponds to the normalized targets tn. The unnormalized network output a is in the same units as the original targets t.

If mapminmax is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums that were computed for the training set stored in the

settings ps. The following code applies a new set of inputs to the network already trained.

```
pnewn = mapminmax('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapminmax('reverse',anewn,ts);
```

Mean and Stand. Dev. (mapstd)

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. The function mapstd normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of mapstd.

```
[pn,ps] = mapstd(p);
[tn,ts] = mapstd(t);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets pn and tn that are returned will have zero means and unity standard deviation. The settings structures ps and ts contain the means and standard deviations of the original inputs and original targets. After the network has been trained, you should use these settings to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If mapstd is used to scale the targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. To convert these outputs back into the same units that were used for the original targets, use ts. The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.

```
an = sim(net,pn);
a = mapstd('reverse',an,ts);
```

The network output an corresponds to the normalized targets tn. The unnormalized network output a is in the same units as the original targets t.

If mapstd is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the means and standard deviations that were computed for the training set using ps. The following commands apply a new set of inputs to the network already trained:

```
pnewn = mapstd('apply',pnew,ps);
```

```
anewn = sim(net,pnewn);
anew = mapstd('reverse',anewn,ts);
```

Principal Component Analysis (processpca)

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the components of the input vectors (so that they are uncorrelated with each other), it orders the resulting orthogonal components (principal components) so that those with the largest variation come first, and it eliminates those components that contribute the least to the variation in the data set. The following code illustrates the use of `processpca`, which performs a principal-component analysis using the processing setting `maxfrac of 0.02`.

```
[pn,ps1] = mapstd(p);
[ptrans,ps2] = processpca(pn,0.02);
```

The input vectors are first normalized, using `mapstd`, so that they have zero mean and unity variance. This is a standard procedure when using principal components. In this example, the second argument passed to `processpca` is 0.02. This means that `processpca` eliminates those principal components that contribute less than 2% to the total variation in the data set. The matrix `ptrans` contains the transformed input vectors. The settings structure `ps2` contains the principal component transformation matrix. After the network has been trained, these settings should be used to transform any future inputs that are applied to the network. It effectively becomes a part of the network, just like the network weights and biases. If you multiply the normalized input vectors `pn` by the transformation matrix `transMat`, you obtain the transformed input vectors `ptrans`.

If `processpca` is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the transformation matrix that was computed for the training set, using `ps2`. The following code applies a new set of inputs to a network already trained.

```
pnewn = mapstd('apply',pnew,ps1);
pnewtrans = processpca('apply',pnewn,ps2);
a = sim(net,pnewtrans);
```

Principal component analysis is not reliably reversible. Therefore it is only recommended for input processing. Outputs requires reversible processing functions.

Processing Unknown Inputs (fixunknowns)

If you have input data with unknown values, you can represent them with `NaN` values. For example, here are five 2-element vectors with unknown values in the first element of two of the vectors:

```
p1 = [1 NaN 3 2 NaN; 3 1 -1 2 4];
```

The network will not be able to process the `NaN` values properly. Use the function `fixunknowns` to transform each row with `NaN` values (in this case only the first row) into two rows that encode that same information numerically.

```
[p2,ps] = fixunknowns(p1);
```

Here is how the first row of values was recoded as two rows.

```
p2 =
   1   2   3   2   2
   1   0   1   1   0
   3   1  -1   2   4
```

The first new row is the original first row, but with the mean value for that row (in this case `2`) replacing all `NaN` values. The elements of the second new row are now either `1`, indicating the original element was a known value, or `0` indicating that it was unknown. The original second row is now the new third row. In this way both known and unknown values are encoded numerically in a way that lets the network be trained and simulated.

Whenever supplying new data to the network, you should transform the inputs in the same way, using the settings `ps` returned by fixunknowns when it was used to transform the training input data.

```
p2new = fixunknowns('apply',p1new,ps);
```

The function `fixunkowns` is only recommended for input processing. Unknown targets represented by `NaN` values can be handled directly by the toolbox learning algorithms. For instance, performance functions used by backpropagation algorithms recognize `NaN` values as unknown or unimportant values.

Representing Unknown or Don't Care Targets

Unknown or "don't care" targets can also 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.

Posttraining Analysis (postreg)

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 `postreg` is designed to perform this analysis.

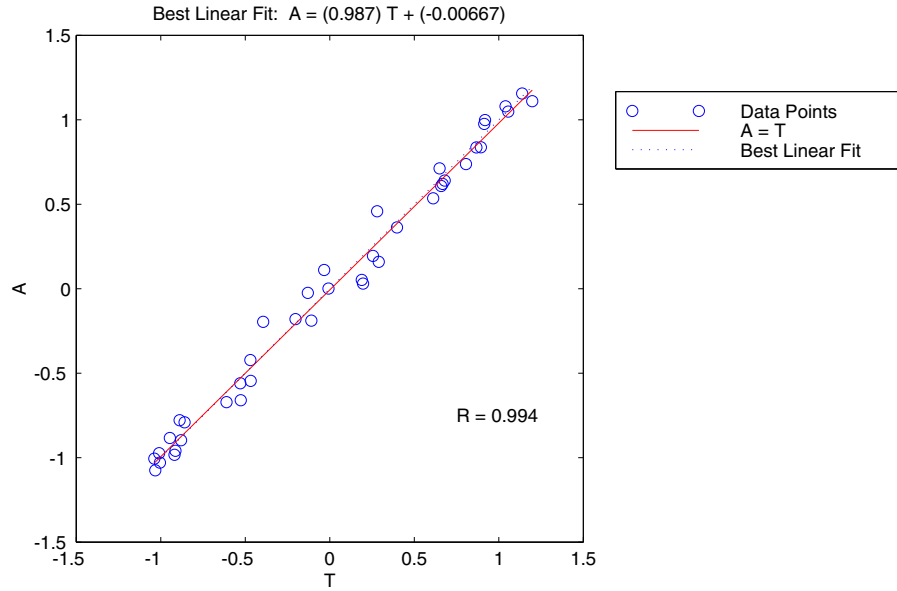
The following commands illustrate how to perform a regression analysis on the network trained in "Summary and Discussion of Early Stopping and Regularization" on page 5-58.

```
a = sim(net,p);
[m,b,r] = postreg(a,t)
m =
    0.9874
b =
   -0.0067
r =
    0.9935
```

The network output and the corresponding targets are passed to `postreg`. 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 `postreg` 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 `postreg`. 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.



Sample Training Session

A number of different concepts are covered in this chapter. At this point it might be useful to put some of these ideas together with an example of how a typical training session might go.

This example uses data from a medical application [PuLu92]. The goal is to design an instrument that can determine serum cholesterol levels from measurements of spectral content of a blood sample. There are a total of 264 patients for which there are measurements of 21 wavelengths of the spectrum. For the same patients there are also measurements of HDL, LDL, and VLDL cholesterol levels, based on serum separation. The first step is to load the data into the MATLAB® workspace.

```
load choles_all
```

Next, create the network. The network object should process inputs by normalizing their standard deviation and performing principal component-analysis using a maximum fraction of 0.001. The targets are normalized for standard deviation.

```
net = newff(p,t,5);
net.inputs{1}.processFcns = {'mapstd','processpca'};
net.inputs{1}.processParams{2}.maxfrac = 0.001;
net.outputs{2}.processFcns = {'mapstd'};
```

You are now ready to create a network and train it. For this example, try a two-layer network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This is a useful structure for function approximation (or regression) problems.

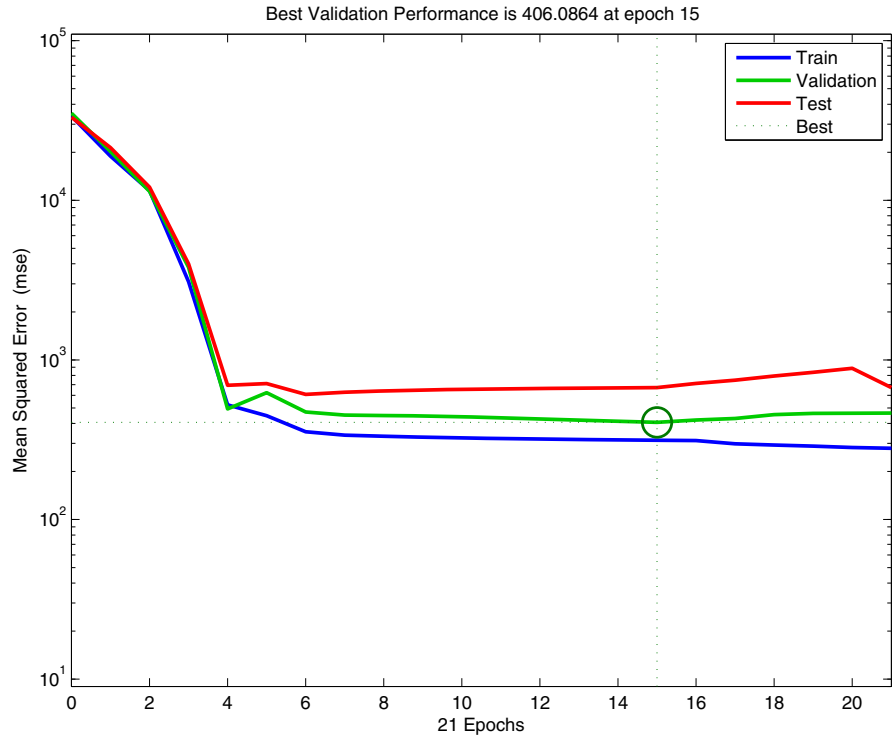
As an initial guess, use five neurons in the hidden layer. The resulting network will have three output neurons because the target vectors have three elements. The default Levenberg-Marquardt algorithm is used for training.

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

The training stopped after 21 iterations because the validation error increased. It is a useful diagnostic tool to plot the training, validation, and test errors to check the progress of training. You can do that with the following commands:

```
plotperform(tr)
```

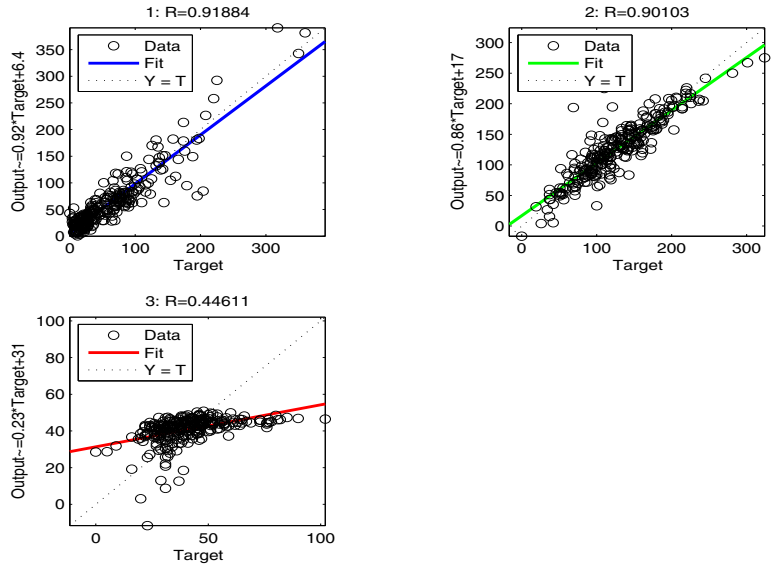
The following figure shows the result. The result here is reasonable, because the test set error and the validation set error have similar characteristics, and it doesn't appear that any significant overfitting has occurred.



The next step is to perform some analysis of the network response. Put the entire data set through the network (training, validation, and test) and perform a linear regression between the network outputs and the corresponding targets. First, calculate the network outputs:

```
y = sim(net,p);
plotregression(t(1,:),y(1,:),'1',t(2,:),y(2,:),'2',t(3,:),y(3,:)
,'3')
```

In this case, there are three outputs, so there are three regressions. The next figure shows the results.



The first two outputs seem to track the targets reasonably well (this is a difficult problem), and the R-values are around 0.9. Modeling the third output (VLDL levels) is not as successful and the problem needs more work. You might go on to try other network architectures (more hidden layer neurons), or to try Bayesian regularization instead of early stopping for the training technique. Of course there is also the possibility that VLDL levels cannot be accurately computed based on the given spectral components.

The demonstration `demobp1` contains the sample training session. The function `nnsample` contains all the commands used in this section. You can use it as a template for your own training sessions.

Limitations and Cautions

The gradient descent algorithm is generally very slow because it requires small learning rates for stable learning. The momentum variation is usually faster than simple gradient descent, because it allows higher learning rates while maintaining stability, but it is still too slow for many practical applications. These two methods are normally used only when incremental training is desired. You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrp`.

Multilayered networks are capable of performing just about any linear or nonlinear computation, and can approximate any reasonable function arbitrarily well. Such networks overcome the problems associated with the perceptron and linear networks. 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] for a discussion of convergence to local minimum points.

Picking the learning rate for a nonlinear network is a challenge. As with linear networks, a learning rate that is too large leads to unstable learning. Conversely, a learning rate that is too small results in incredibly long training times. Unlike linear networks, there is no easy way of picking a good learning rate for nonlinear multilayer networks. See page 12-8 of [HDB96] for examples of choosing the learning rate. With the faster training algorithms, the default parameter values normally perform adequately.

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], 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 it is possible for the network solution to become trapped in one of these local minima. This can happen, depending on the initial starting conditions. 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 “Improving Generalization” on page 5-52. This topic is also discussed starting on page 11-21 of [HDB96].

6

Dynamic Networks

Introduction (p. 6-2)
Focused Time-Delay Neural Network (newfftd) (p. 6-11)
Distributed Time-Delay Neural Network (newdtdnn) (p. 6-15)
NARX Network (newnarx, newnarxsp, sp2narx) (p. 6-18)
Layer-Recurrent Network (newlrn) (p. 6-24)

Introduction

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. The training of static networks was discussed in Chapter 5, “Backpropagation.” 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. You saw some linear dynamic networks in Chapter 4, “Linear Filters.”

Dynamic networks can also be divided into two categories: those that have only feedforward connections, and those that have feedback, or recurrent, connections.

Examples of Dynamic Networks

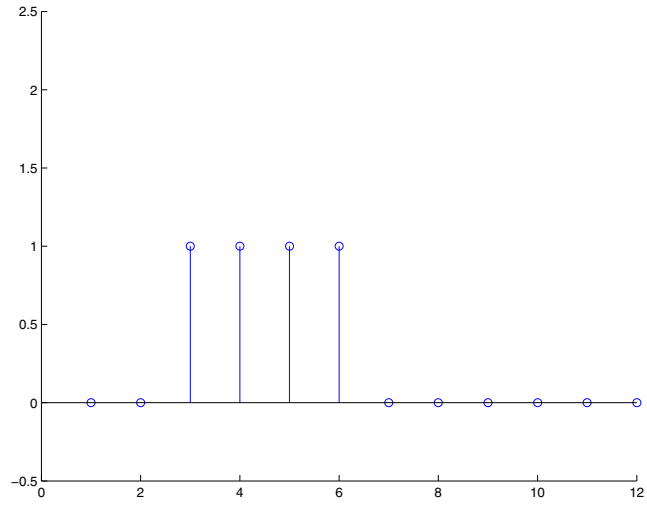
To understand the differences between static, feedforward-dynamic, and recurrent-dynamic networks, create some networks and see how they respond to an input sequence. (First, you might want to review the section on applying sequential inputs to a dynamic network on page 2-15.)

The following command creates a pulse input sequence and plots it:

```
p = {0 0 1 1 1 1 0 0 0 0 0 0};

stem(cell2mat(p))
```

The next figure show the resulting pulse.



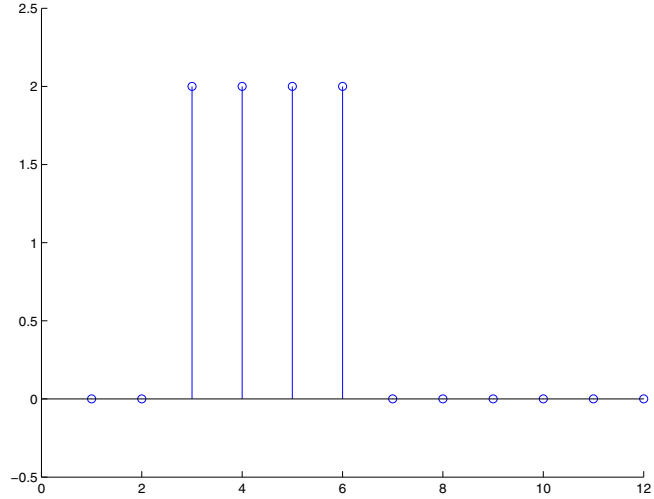
Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

```
net = newlin(p,1);
net.biasConnect = 0;
net.IW{1,1} = 2;
```

You can now simulate the network response to the pulse input and plot it:

```
a = sim(net,p);
stem(cell2mat(a))
```

The result is shown in the following figure. 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.



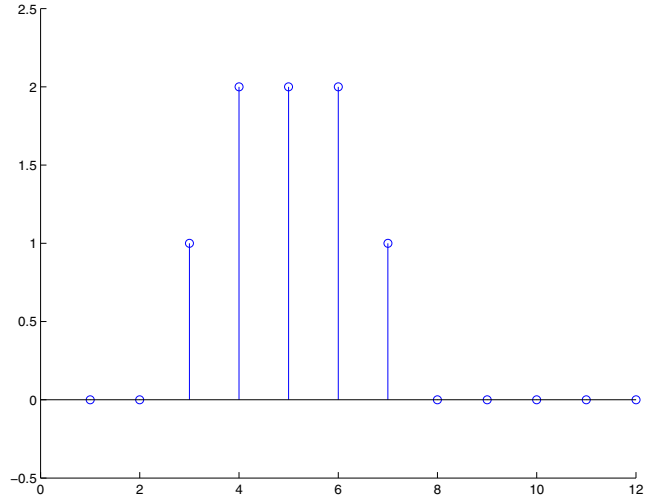
Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used on page 2-15, which was a linear network with a tapped delay line on the input:

```
net = newlin(p,1,[0 1]);
net.biasConnect = 0;
net.IW{1,1} = [1 1];
```

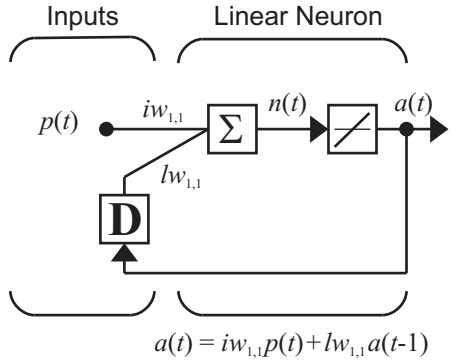
You can again simulate the network response to the pulse input and plot it:

```
a = sim(net,p);
stem(cell2mat(a))
```

The response of the dynamic network, shown in the following figure, 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.


2.5
2
1.5
1
0.5
0
−0.5
0
2
4
6
8
10
12

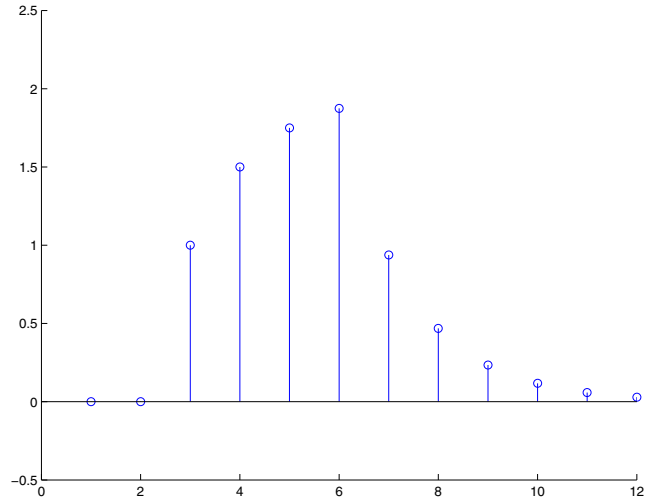
Now consider a simple recurrent-dynamic network, shown in the following figure.



You can create the network and simulate it with the following commands. The `newnarx` command is discussed in "NARX Network (newnarx, newnarxsp, sp2narx)" on page 6-18.

```
net = newnarx(p,[-1 1],0,1,[],{'purelin'});
net.biasConnect = 0;
net.LW{1} = .5;
frnet.IW{1} = 1;
a = sim(net,p);
stem(cell2mat(a))
```

The following figure is the plot of the network response.



Notice that the recurrent-dynamic networks typically have a longer response than the feedforward-dynamic networks. For linear networks, the 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. The 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], channel equalization in communication systems [FeTs03], phase detection in power systems [KaGr96], sorting [JaRa04], fault detection [ChDa99], speech recognition [Robin94], and even the prediction of protein structure in genetics [GiPr02]. You can find a discussion of many more dynamic network applications in [MeJa00].

One principal application of dynamic neural networks is in control systems. This application is discussed in detail in Chapter 7, “Control Systems.” Dynamic networks are also well suited for filtering. You have seen the use of some linear dynamic networks for filtering in Chapter 4, “Linear Filters,” and some of those ideas are extended in this chapter, using nonlinear dynamic networks.

Dynamic Network Structures

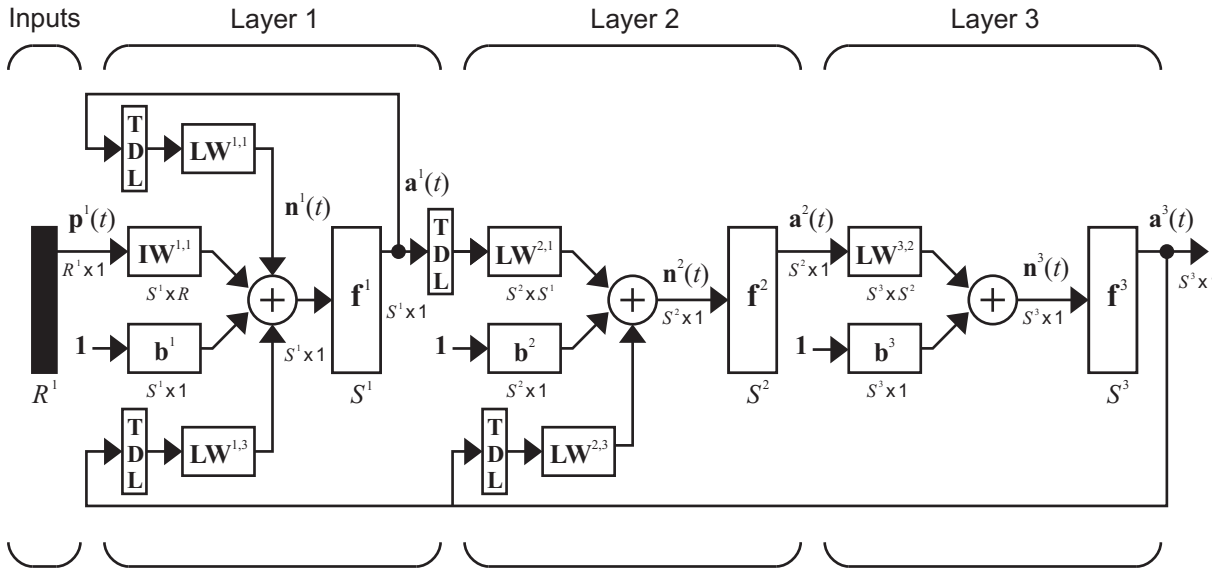
The Neural Network Toolbox™ software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

Each layer in the LDDN is made up of the following parts:

- Set of weight matrices that come into that layer (which can connect from other layers or from external inputs), associated weight function rule used to combine the weight matrix with its input (normally standard matrix multiplication, `dotprod`), and associated tapped delay line
- Bias vector
- Net input function rule that is used to combine the outputs of the various weight functions with the bias to produce the net input (normally a summing junction, `netprod`)
- Transfer function

The network has inputs that are connected to special weights, called input weights, and denoted by $\mathbf{IW}^{i,j}$ (`net.IW{i,j}` in the code), where j denotes the number of the input vector that enters the weight, and i denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called layer weights and are denoted by $\mathbf{LW}^{i,j}$ (`net.LW{i,j}` in the code), where j denotes the number of the layer coming into the weight and i denotes the number of the layer at the output of the weight.

The following figure is an example of a three-layer LDDN. The first layer has three weights associated with it: one input weight, a layer weight from layer 1, and a layer weight from layer 3. The two layer weights have tapped delay lines associated with them.



The 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 chapter 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 chapter can be created using the generic network command, as explained in Chapter 12, “Advanced Topics.”

Dynamic Network Training

Dynamic networks are trained in the Neural Network Toolbox software using the same gradient-based algorithms that were described in Chapter 5, “Backpropagation.” You can select from any of the training functions that were presented in that chapter. Examples are provided in the following sections.

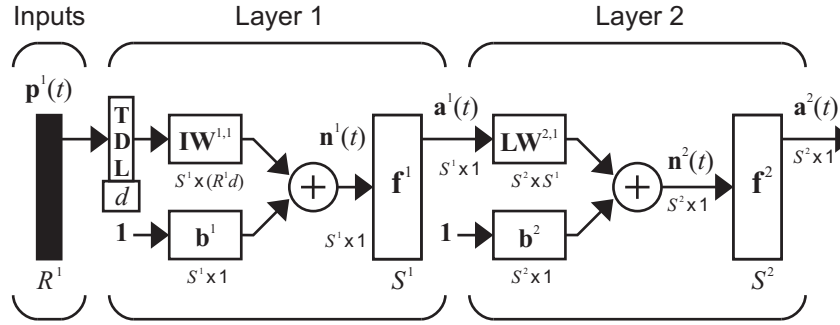
Although dynamic networks can be trained using the same gradient-based algorithms that are used for static networks, the performance of the algorithms on dynamic networks can be quite different, and the gradient must be computed in a more complex way. Consider the simple recurrent network shown on page 6-6. 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] and [DeHa01b].) 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 [DHM01] for some discussion on the training of dynamic networks.

The remaining sections of this chapter demonstrate 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.

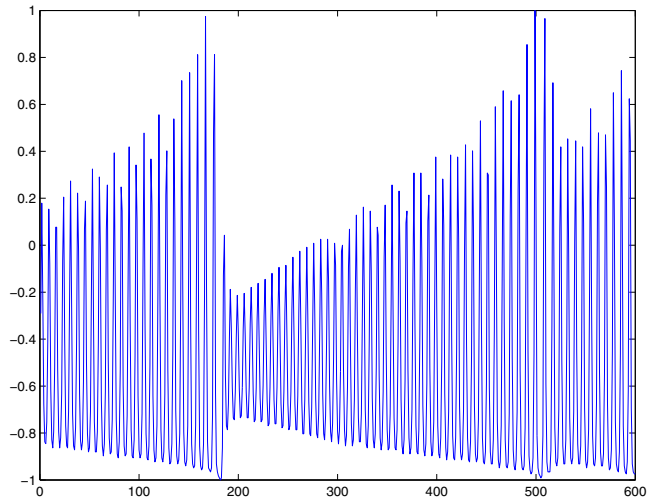
Focused Time-Delay Neural Network (newfftd)

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 demonstrates 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]. In the competition, the objective was to use the first 1000 points of the time series to predict the next 100 points. Because the objective is simply to illustrate how to use the FTDNN for prediction, the network is trained 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):

```
load laser
y = y(1:600)';
y = con2seq(y);
```

Now create the FTDNN network, using the `newfftd` command. This command is similar to the `newff` command, described on page 5-11, with the additional input of the tapped delay line vector (the second input). For this example, use a tapped delay line with delays from 1 to 8, and use five neurons in the hidden layer:

```
ftdnn_net = newfftd(y,y,[1:8],5);
ftdnn_net.trainParam.show = 10;
ftdnn_net.trainParam.epochs = 50;
```

Arrange the network inputs and targets for training. Because the network has a tapped delay line with a maximum delay of 8, begin by predicting the ninth value of the time series. You also need to load the tapped delay line with the eight initial values of the time series (contained in the variable `Pi`):

```
p = y(9:end);
```

```
t = y(9:end);
Pi=y(1:8);
ftdnn_net = train(ftdnn_net,p,t,Pi);
```

Notice that the input to the network is the same as the target. Because the network has a minimum delay of one time step, this means that you are performing a one-step-ahead prediction.

Now simulate the network and determine the prediction error.

```
yp = sim(ftdnn_net,p,Pi);
yp = cell2mat(yp);
e = yp-cell2mat(t);
rmse = sqrt(mse(e))

rmse =
    0.0260
```

This result is much better than you could have obtained using a linear predictor, such as those shown in Chapter 4, “Linear Filters.” You can verify this with the following commands, which design a linear filter with the same tapped delay line input as the previous FTDNN. (Because `newlind` creates a tapped delay line that contains a zero delay, you need to shift the input to the network by one time step.)

```
p = y(8:end-1);
clear Pi
Pi=y(1:7);
lin_net = newlind(p,t,Pi);
lin_yp = sim(lin_net,p,Pi);
lin_yp = cell2mat(lin_yp);
lin_e = lin_yp-cell2mat(t);
lin_rmse = sqrt(mse(lin_e))

lin_rmse =
   0.1807
```

The `rms` error is 0.1807 for the linear predictor, but 0.0260 for the nonlinear FTDNN predictor.

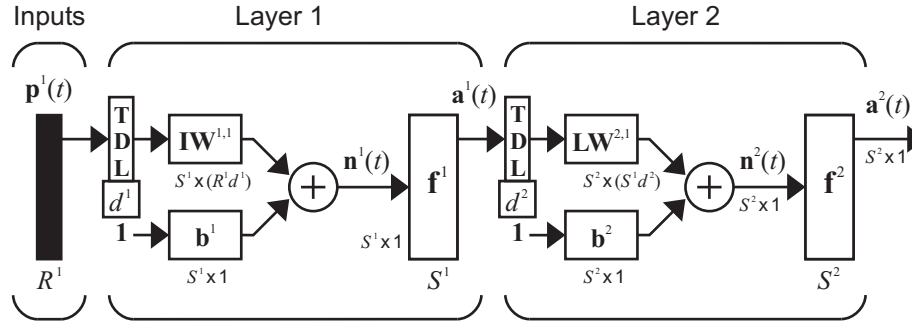
One nice feature of the FTDNN is that it does not require dynamic backpropagation to compute the network gradient. This is because the tapped delay line appears only at the input of the network, and contains no feedback

loops or adjustable parameters. For this reason, you will find that this network trains faster than other dynamic networks.

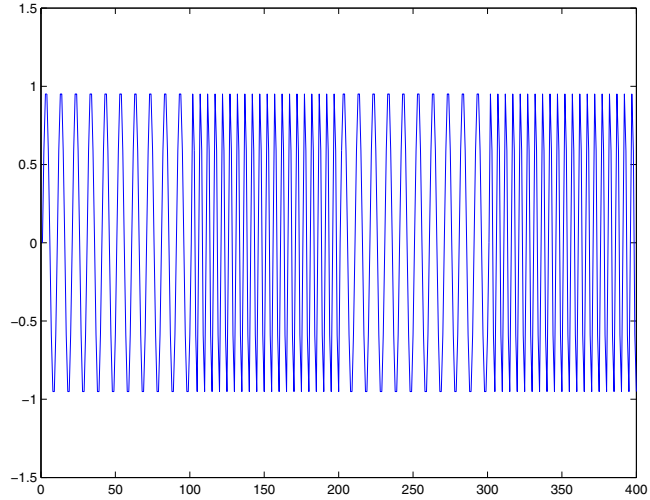
If you have an application for a dynamic network, try the linear network first (`newlind`) and then the FTDNN (`newfftd`). If neither network is satisfactory, try one of the more complex dynamic networks discussed in the remainder of this chapter.

Distributed Time-Delay Neural Network (newdtdnn)

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] for phoneme recognition. The original architecture was very specialized for that particular problem. The figure below 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.



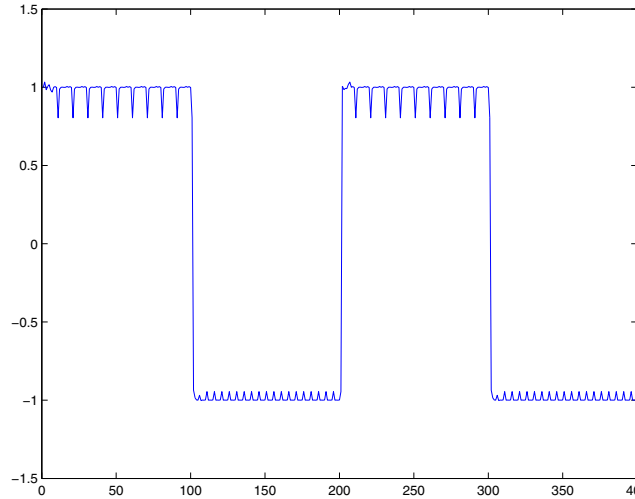
The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
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 `newdtdnn` function. The only difference between the `newdtdnn` function and the `newfftd` function is that the second 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 Chapter 5, "Backpropagation.")

```
d1 = 0:4;
d2 = 0:3;
p = con2seq(y);
t = con2seq(t);
dtdnn_net = newdtdnn(p,t,5,{d1,d2});
dtdnn_net.trainFcn = 'trainbr';
dtdnn_net.trainParam.show = 5;
dtdnn_net.trainParam.epochs = 30;
dtdnn_net = train(dtdnn_net,p,t);
yp = sim(dtdnn_net,p);
yp = cell2mat(yp);
plot(yp);
```

The following figure shows the trained network output. 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.

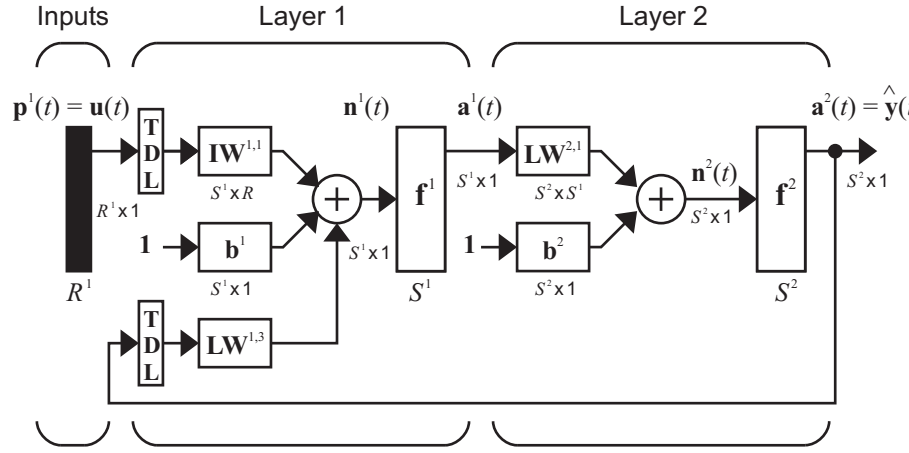
NARX Network (newnarx, newnarxsp, sp2narx)

All the specific dynamic networks discussed so far have either been focused networks, with the dynamics only at the input layer, or feedforward networks. The nonlinear autoregressive network with exogenous inputs (NARX) is a recurrent dynamic network, with feedback connections enclosing several layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time-series modeling.

The defining equation for the NARX model is

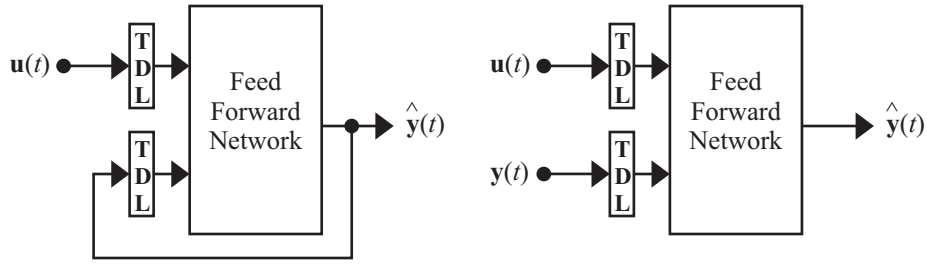
$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))$$

where the next value of the dependent output signal $y(t)$ is regressed on previous values of the output signal and previous values of an independent (exogenous) input signal. You can implement the NARX model by using a feedforward neural network to approximate the function f. A diagram of the resulting network is shown below, where a two-layer feedforward network is used for the approximation. This implementation also allows for a vector ARX model, where the input and output can be multidimensional.



There are many applications for the NARX network. It can be used as a predictor, to predict the next value of the input signal. It can also be used for nonlinear filtering, in which the target output is a noise-free version of the input signal. The use of the NARX network is demonstrated in another important application, the modeling of nonlinear dynamic systems.

Before demonstrating 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]), in which the true output is used instead of feeding back the estimated output, as shown in the right figure below. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training.



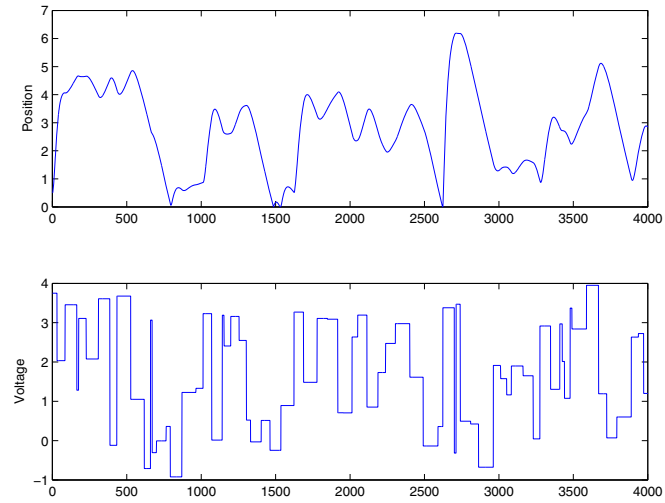
Parallel Architecture

Series-Parallel Architecture

The following demonstrates the use of the series-parallel architecture for training an NARX network to model a dynamic system.

The example of the NARX network is the magnetic levitation system described beginning on page 7-19. 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 an 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, so p is a cell array with two rows:

```
load magdata
[u,us] = mapminmax(u);
[y,ys] = mapminmax(y);
y = con2seq(y);
u = con2seq(u);
p = u(3:end);
t = y(3:end);
```

Create the series-parallel NARX network using the function `newnarxsp`. Use 10 neurons in the hidden layer and use `trainbr` for the training function:

```
d1 = [1:2];
d2 = [1:2];
narx_net = newnarxsp(p,t,d1,d2,10);
narx_net.trainFcn = 'trainbr';
narx_net.trainParam.show = 10;
```

```
narx_net.trainParam.epochs = 600;
```

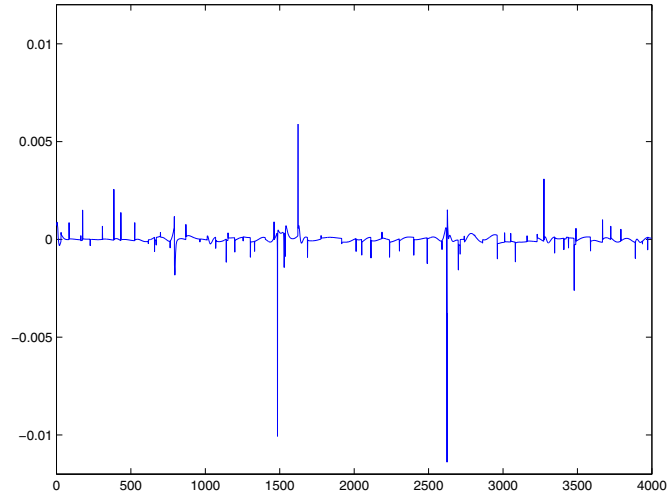
Now you are ready to train the network. First you need to load the tapped delay lines with the initial inputs and outputs. The following commands illustrate these steps.

```
Pi = [u(1:2); y(1:2)];
narx_net = train(narx_net,[p;t],t,Pi);
```

You can now simulate the network and plot the resulting errors for the series-parallel implementation.

```
yp = sim(narx_net,[p;t],Pi);
e = cell2mat(yp)-cell2mat(t);
plot(e)
```

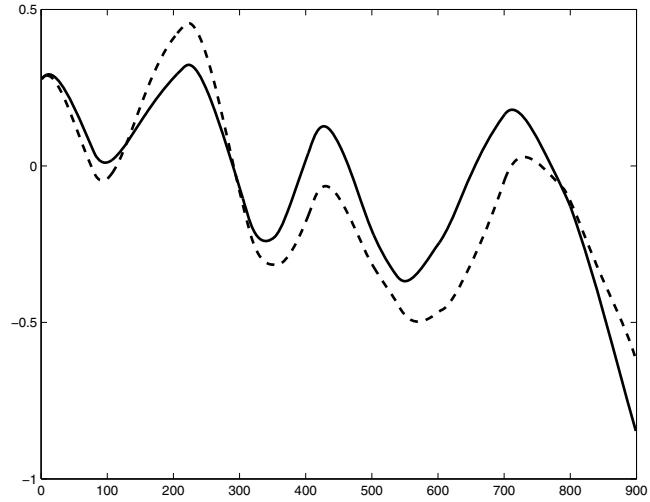
The result is displayed in the following plot. You can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form and then to perform an iterated prediction over many time steps. Now the parallel operation is demonstrated.



There is a toolbox function (sp2narx) for converting NARX networks from the series-parallel configuration, which is useful for training, to the parallel configuration. The following commands illustrate how to convert the network just trained to parallel form and then use that 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.

```
narx_net2 = sp2narx(narx_net);
y1=y(1700:2600);
u1=u(1700:2600);
p1 = u1(3:end);
t1 = y1(3:end);
for k=1:2,
  Ai1{1,k}=zeros(10,1);
  Ai1{2,k}=y1{k};
end
for k=1:2,
  Pi1{1,k}=u1{k};
end
yp1 = sim(narx_net2,p1,Pi1,Ai1);
plot([cell2mat(yp1)' cell2mat(t1)'])
```

The following figure illustrates the iterated prediction. The solid line is the actual position of the magnet, and the dashed line is the position predicted by the NARX neural network. Although the network prediction is noticeably different from the actual response after 50 time steps, the general behavior of the model is very similar to the behavior of the actual system. By collecting more data and training the network further, you can produce an even more accurate result.

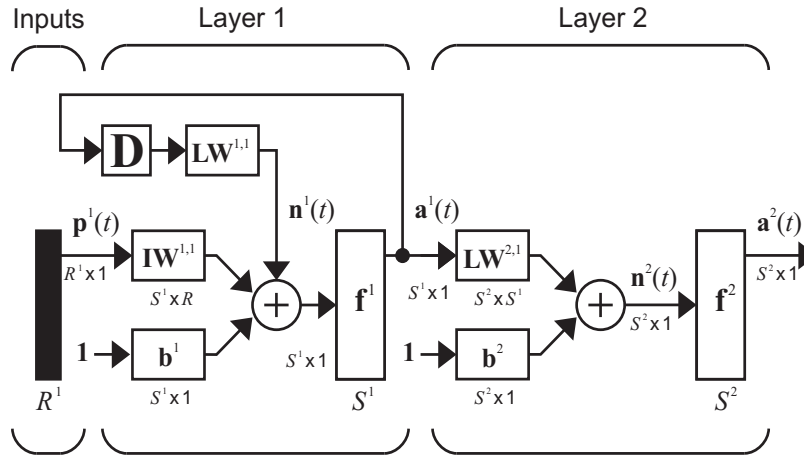


In order for the parallel response to be accurate, it is important that the network be trained so that the errors in the series-parallel configuration are very small.

You can also create a parallel NARX network, using the `newnarx` command, 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.

Layer-Recurrent Network (newlrn)

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]. 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 `newlrn` 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 Chapter 5, "Backpropagation." The following figure illustrates a two-layer LRN.



The LRN configurations are used in many filtering and modeling applications discussed already. To demonstrate its operation, the "phoneme" detection problem discussed on page 6-15 is used. Here is the code to load the data and to create and train the network:

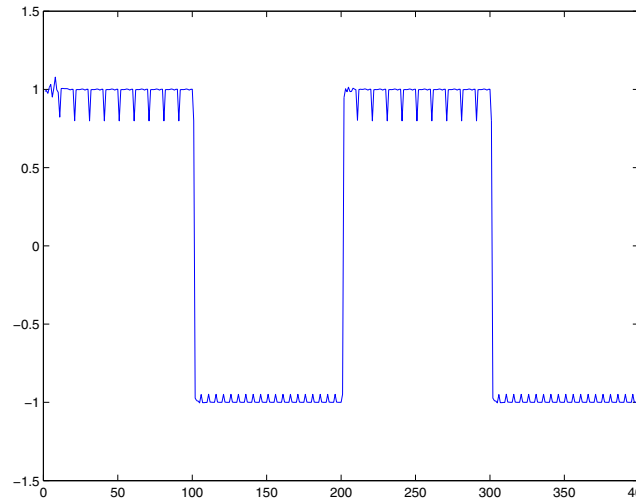
```
load phoneme
p = con2seq(y);
t = con2seq(t);
lrn_net = newlrn(p,t,8);
lrn_net.trainFcn = 'trainbr';
```

```
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 = sim(lrn_net,p);
plot(cell2mat(y));
```

The following plot demonstrates that the network was able to detect the "phonemes." The response is very similar to the one obtained using the TDNN.



7

Control Systems

Introduction (p. 7-2)
NN Predictive Control (p. 7-4)
NARMA-L2 (Feedback Linearization) Control (p. 7-15)
Model Reference Control (p. 7-24)
Importing and Exporting (p. 7-32)

Introduction

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]. This chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox™ software:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them.

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 chapter, 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 of this chapter discuss model predictive control, NARMA-L2 control, and model reference control. Each section consists of a brief description of the control concept, followed by a demonstration of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Neural Network 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, using any of the training algorithms discussed in Chapter 5, "Backpropagation." (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 7-15 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 the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

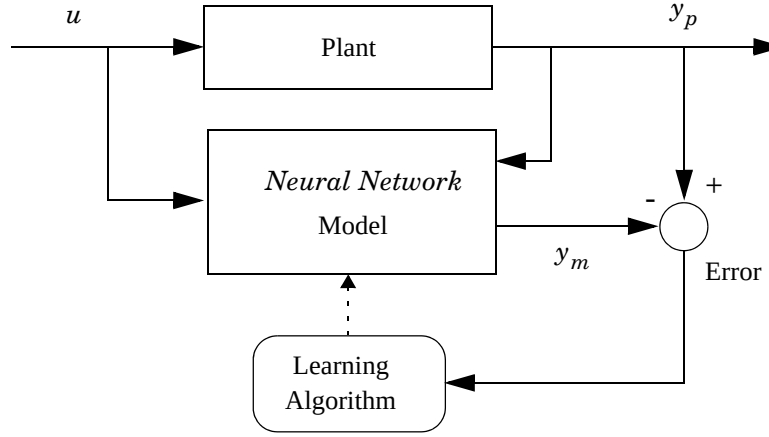
NN Predictive Control

The neural network predictive controller that is implemented in the Neural Network 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.)

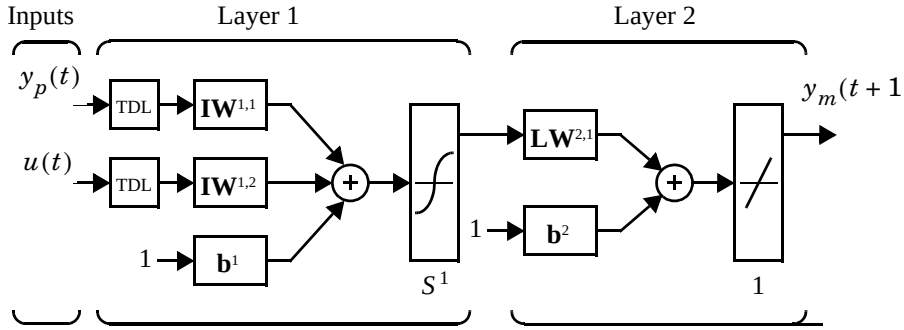
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink® environment.

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 Chapter 5, “Backpropagation,” for network training. This process is discussed in more detail later in this chapter.

Predictive Control

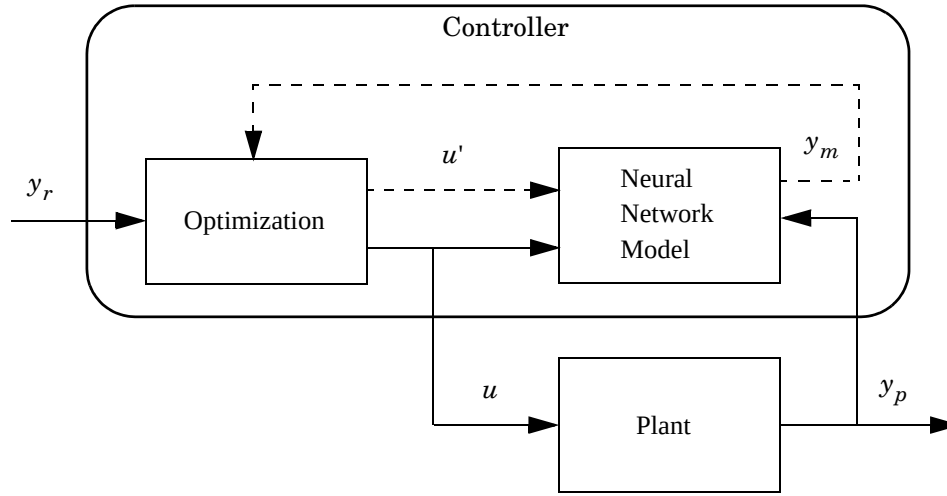
The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance 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 and 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

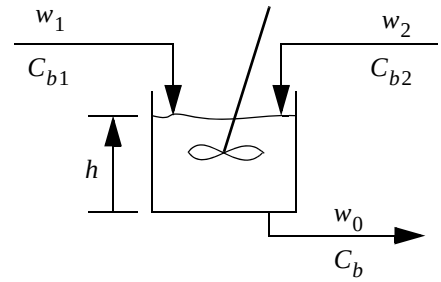
block. The optimization block determines the values of u' that minimize J, and then the optimal u is input to the plant. The controller block is implemented in Simulink, as described in the following section.



Using the NN Predictive Controller Block

This section demonstrates how the NN Predictive Controller block is used. The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the predictive controller. This demo uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.


w_1
C_{b1}
w_2
C_{b2}
h
w_0
C_b

The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

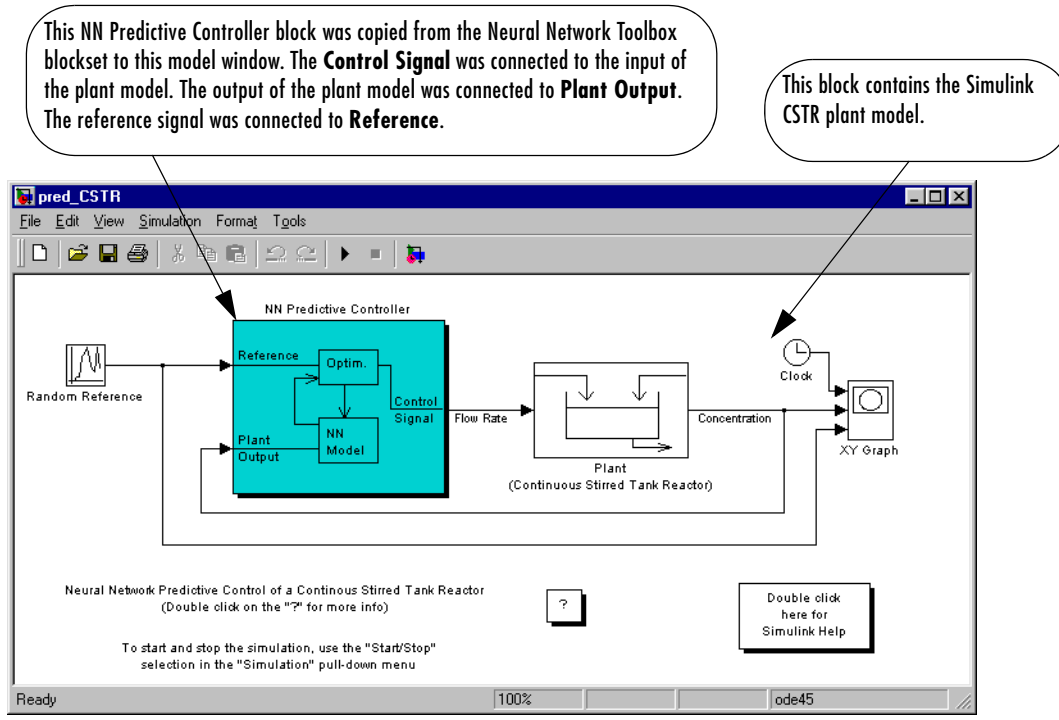
where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1}, and $w_2(t)$ is the flow rate of the diluted feed C_{b2}. The input concentrations are set to $C_{b1} = 24.9$ and $C_{b2} = 0.1$. The constants 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 demonstration, set $w_2(t) = 0.1$. The level of the tank $h(t)$ is not controlled for this experiment.

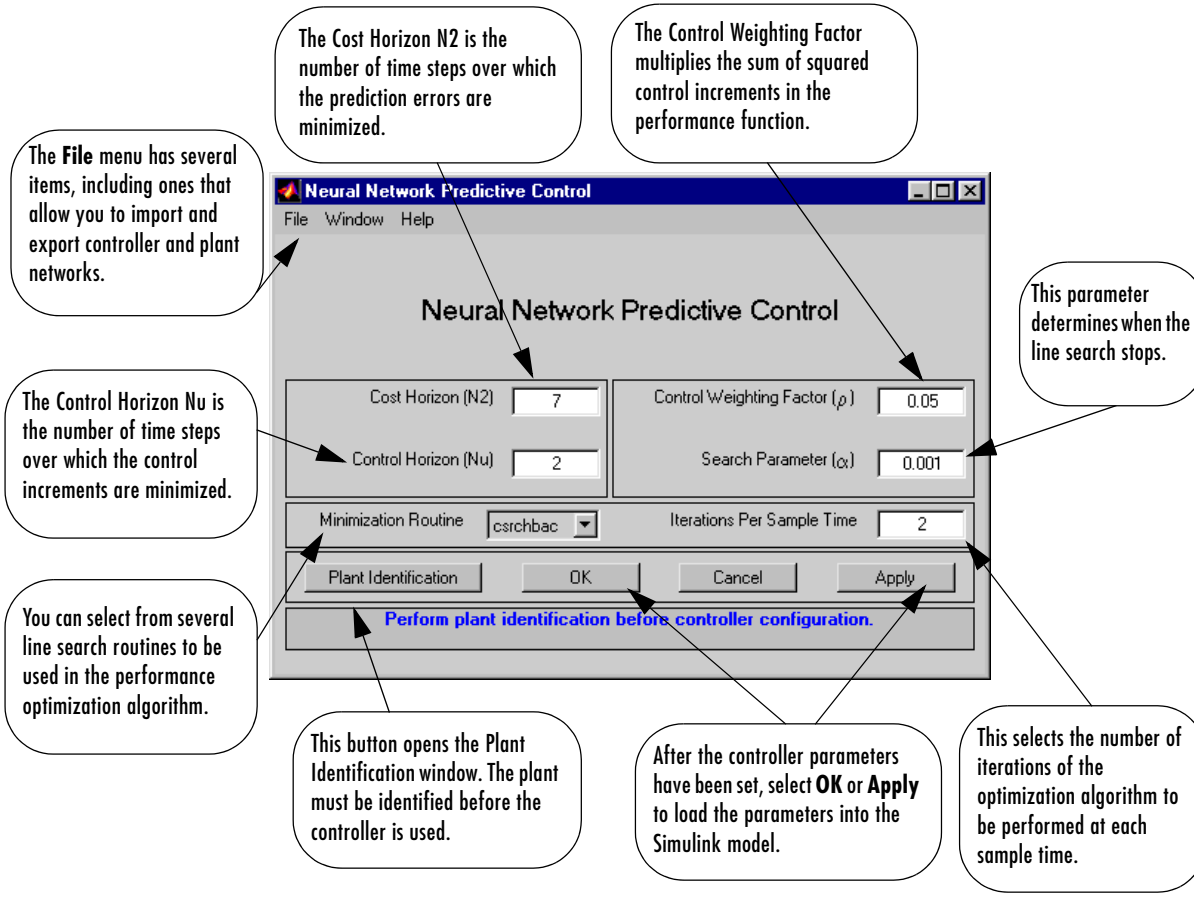
To run this demo, follow these steps:

1. Start MATLAB®.

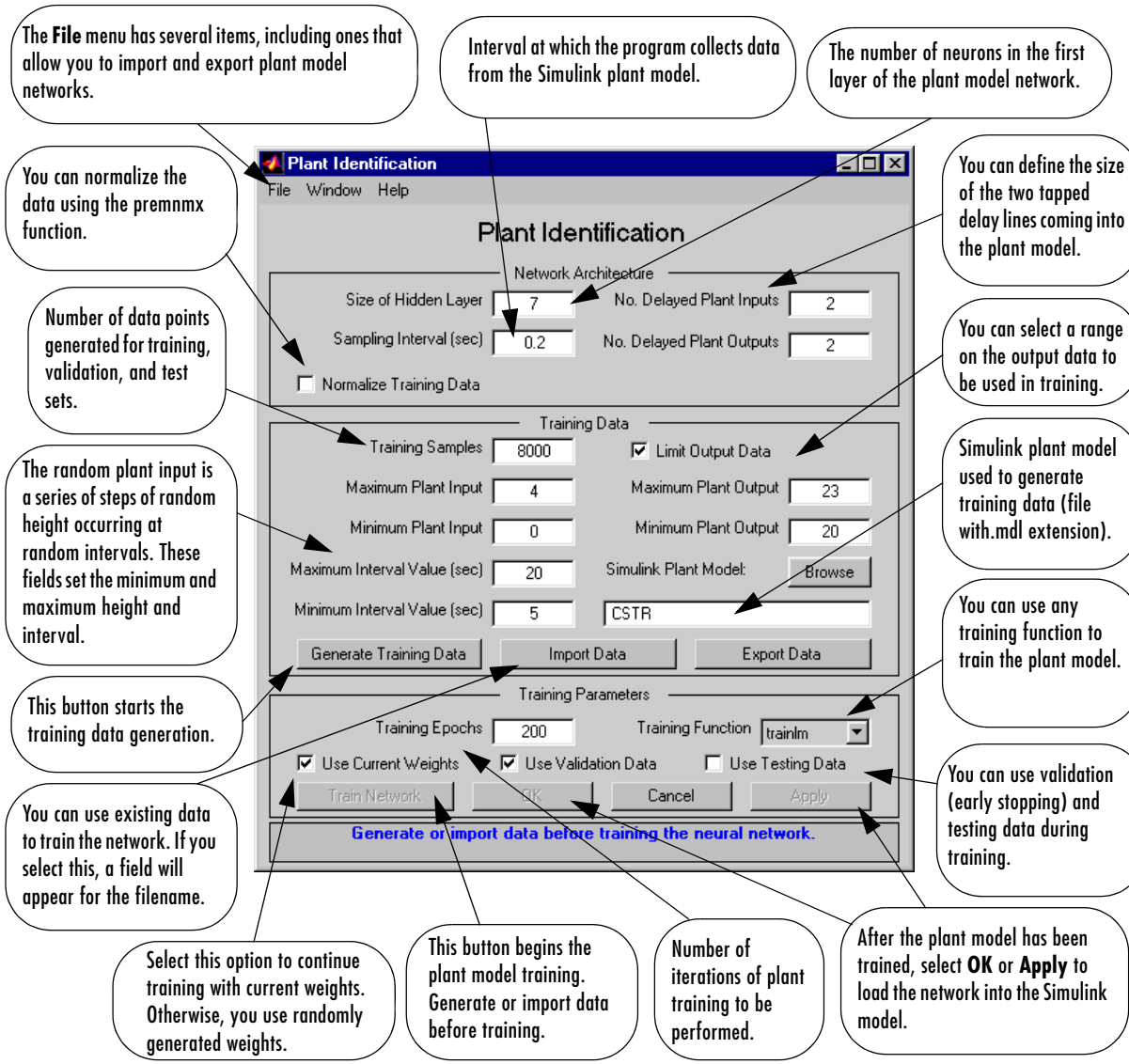
2. Run the demo model by typing `predcstr` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The NN Predictive Controller block is already in the model.



3 Double-click the NN Predictive Controller block. This brings up 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 p, 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 Chapter 5, “Backpropagation.”

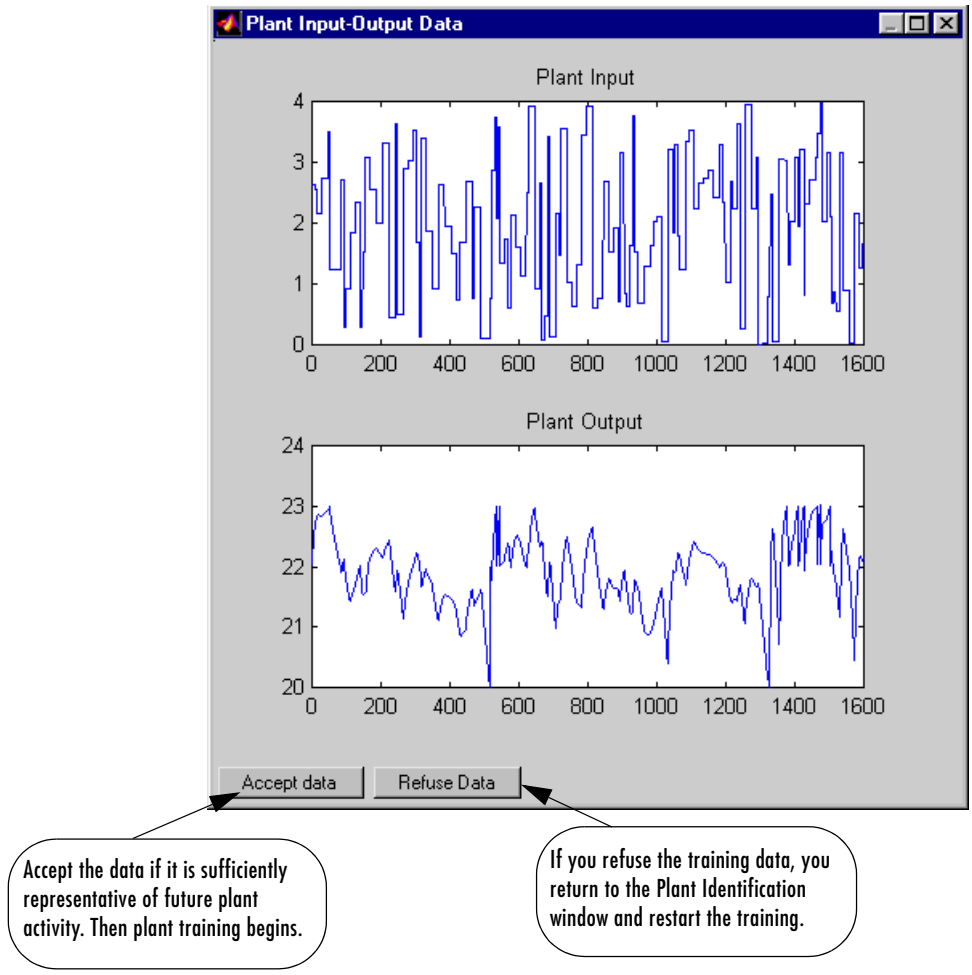


4 Select **Plant Identification**. This opens the following window. You must develop the neural network plant model before you can use the controller. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future 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 Chapter 5, "Backpropagation," to train the neural network plant model.



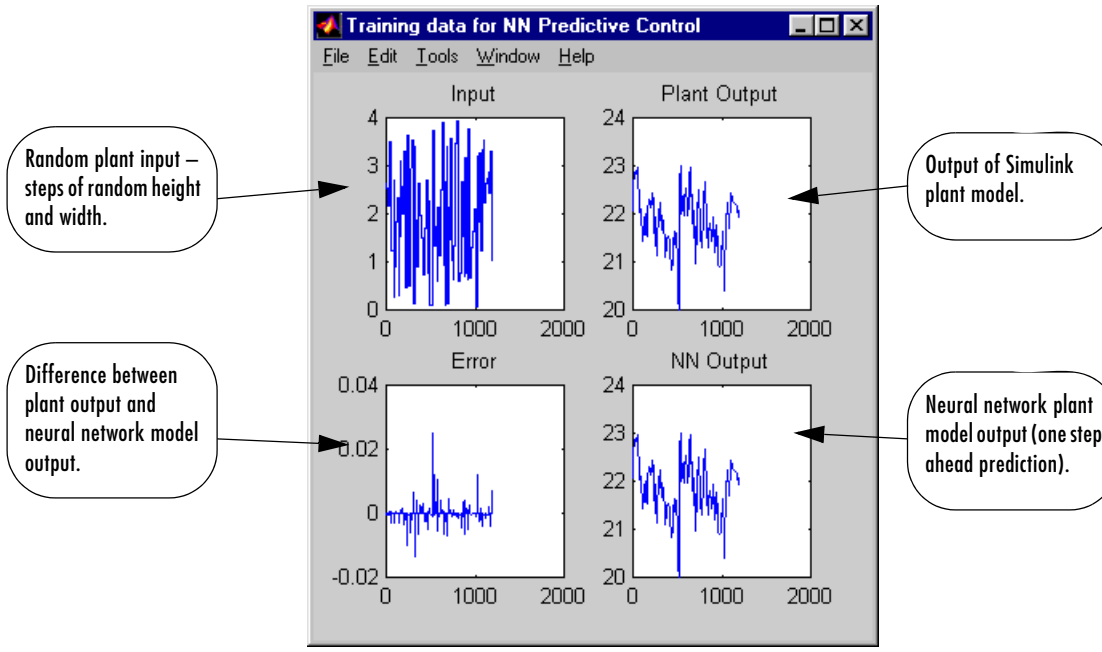
5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model.

The potential training data is then displayed in a figure similar to the following.



6 Select **Accept Data**, and then select **Train Network** from 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 Chapter 5, "Backpropagation." 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.)

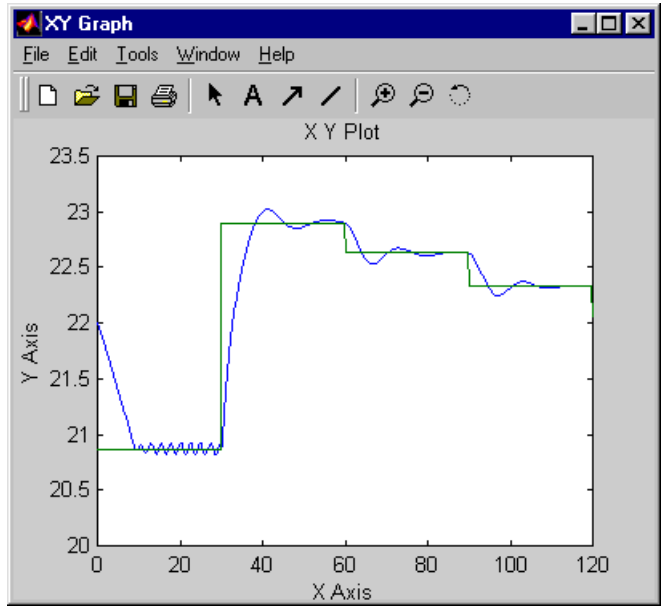


You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this demonstration, begin the simulation, as shown in the 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 model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
23.5
23
22.5
22
21.5
21
20.5
20
Y Axis
0
20
40
60
80
100
120
X Axis

NARMA-L2 (Feedback Linearization) Control

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 demonstrating 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 a demonstration of how to use the NARMA-L2 Control block, which is contained in the Neural Network Toolbox™ blockset.

Identification of the NARMA-L2 Model

As with model predictive control, the first step in using feedback linearization (or NARMA-L2) control is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that is used to represent general discrete-time nonlinear systems is the nonlinear autoregressive-moving average (NARMA) model:

$$y(k+d) = N[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)]$$

where $u(k)$ is the system input, and $y(k)$ is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function N. This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory $y(k+d) = y_r(k+d)$, the next step is to develop a nonlinear controller of the form

$$u(k) = G[y(k), y(k-1), \ldots, y(k-n+1), y_r(k+d), u(k-1), \ldots, u(k-m+1)]$$

The problem with using this controller is that if you want to train a neural network to create the function G to minimize mean square error, you need to use dynamic backpropagation ([NaPa91] or [HaJe99]). This can be quite slow. One solution, proposed by Narendra and Mukhopadhyay [NaMu97], is to use

approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\hat{y}(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)$$

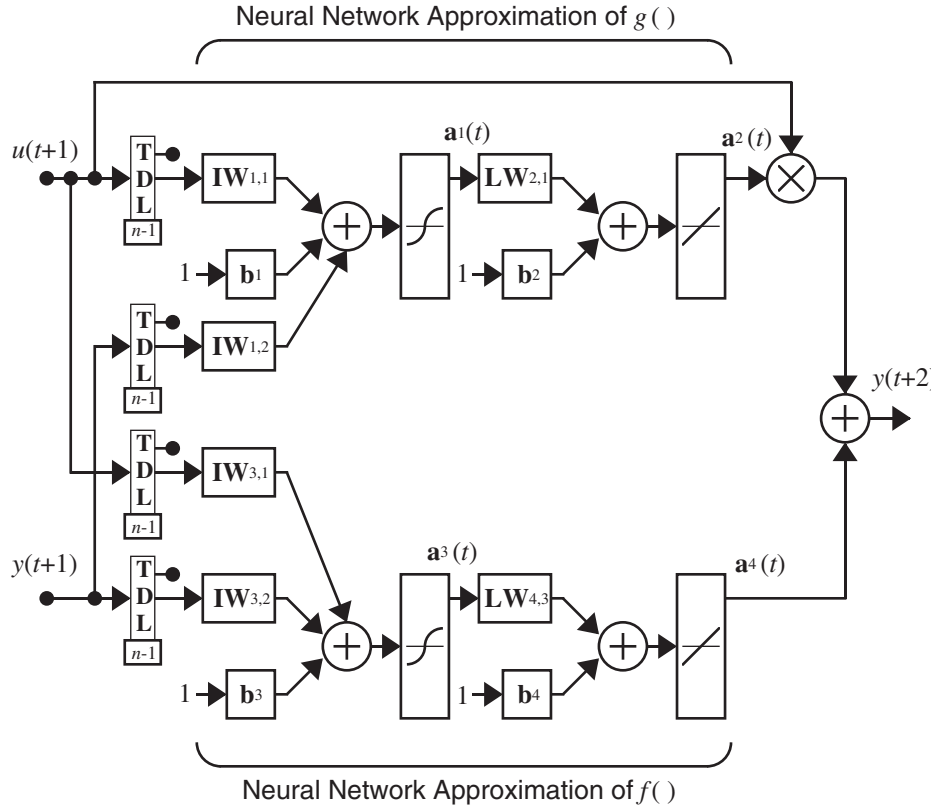
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input $u(k)$ based on the output at the same time, $y(k)$. So, instead, use the model

$$y(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)] + g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)] \cdot u(k+1)$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

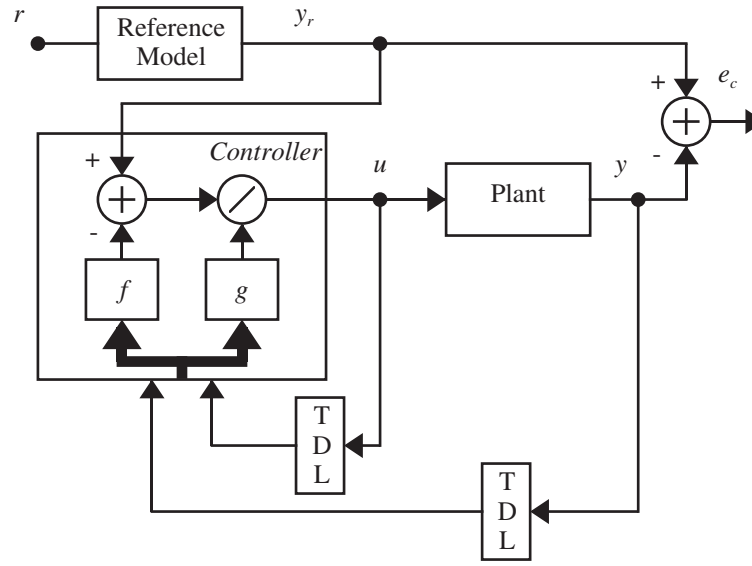


NARMA-L2 Controller

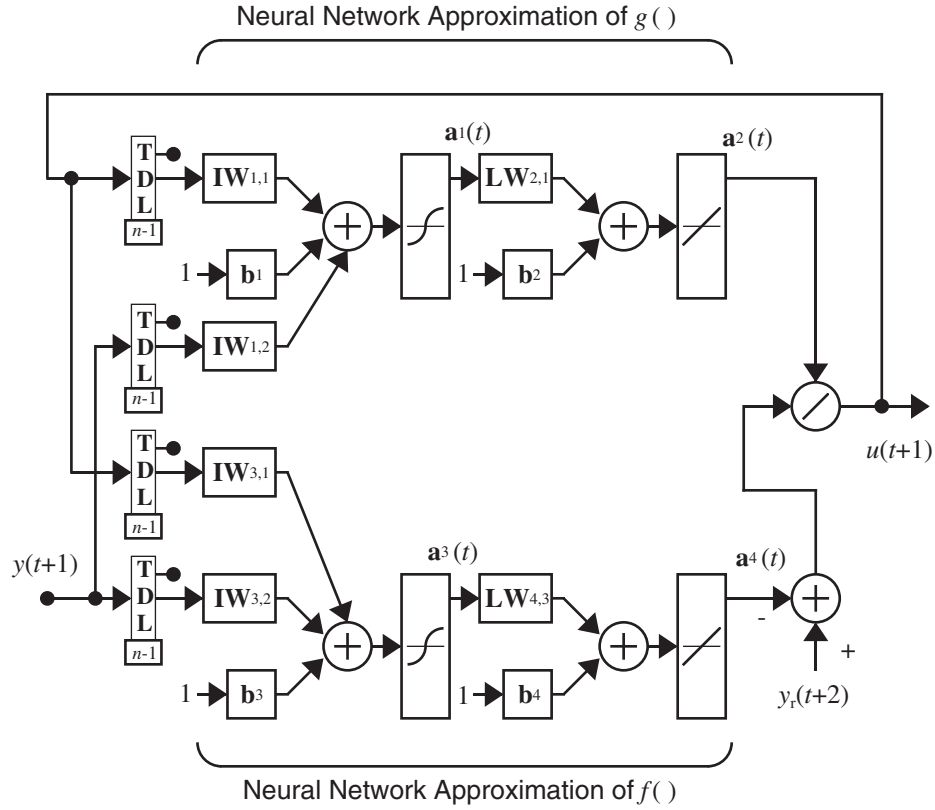
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



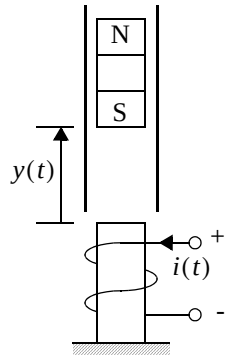
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



Using the NARMA-L2 Controller Block

This section demonstrates how the NARMA-L2 controller is trained. The first step is to copy the NARMA-L2 Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink® documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the NARMA-L2 controller. In this demo, 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

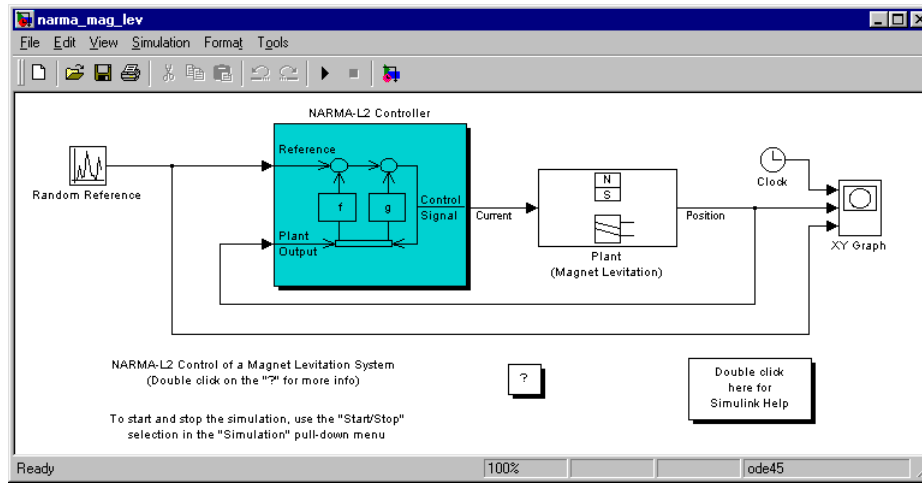
$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

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.

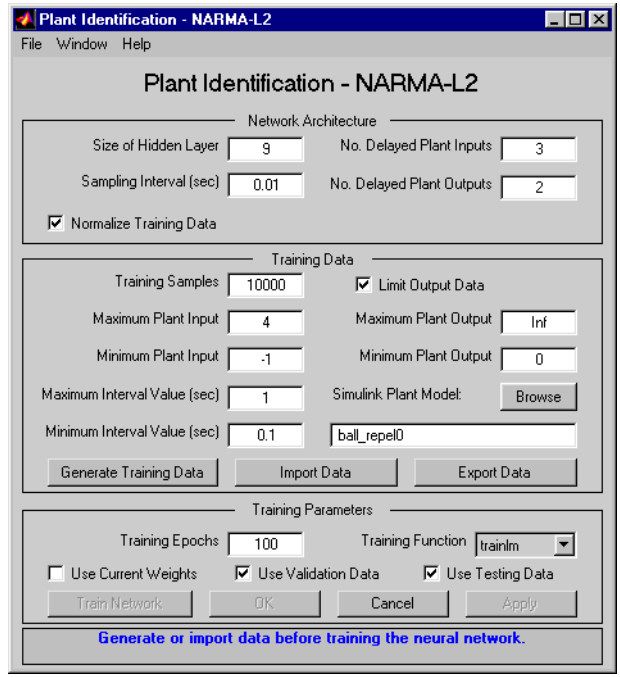
To run this demo, follow these steps:

1. Start MATLAB®.

2. Run the demo model by typing `narmamaglev` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The NARMA-L2 Control block is already in the model.

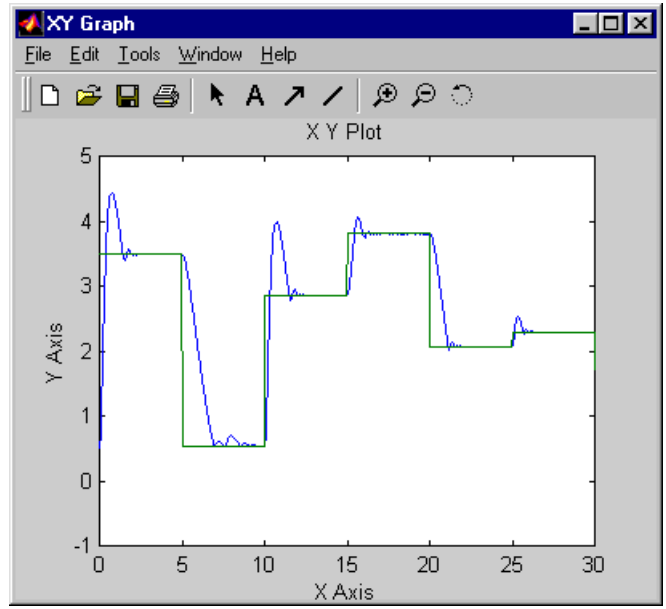


3 Double-click the NARMA-L2 Controller block. This brings up the following window. This window enables you to train the NARMA-L2 model. There is no separate window for the controller, because the controller is determined directly from the model, unlike the model predictive controller.



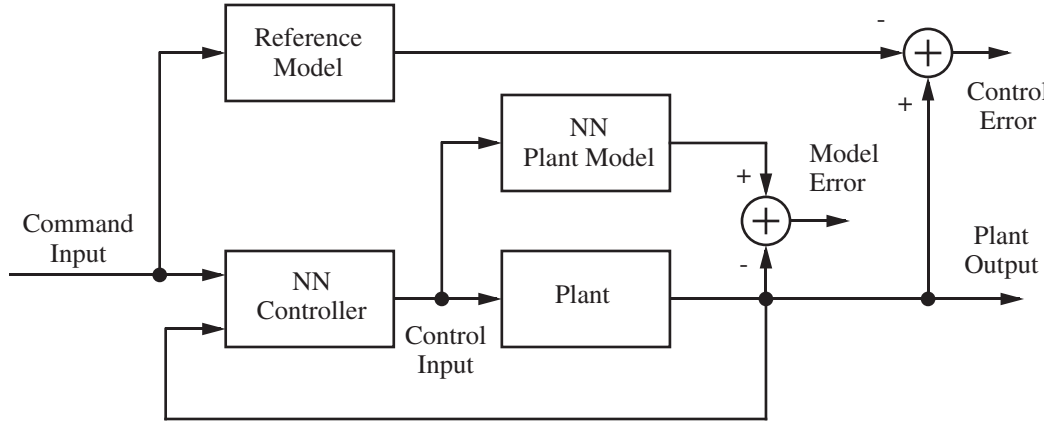
4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.

5 Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
0
5
10
15
20
25
30

Model Reference Control

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.



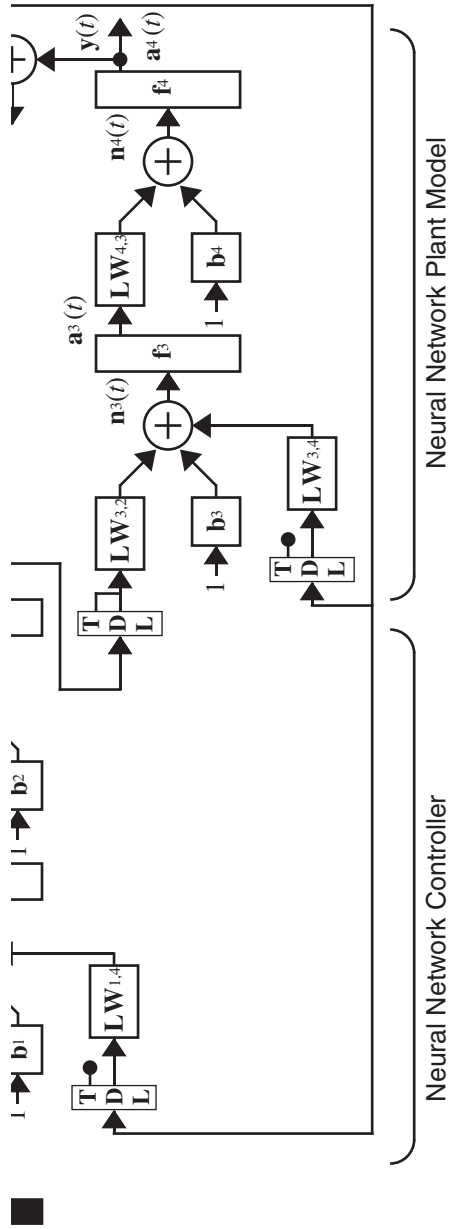
The figure on the following page shows the details of the neural network plant model and the neural network controller as they are implemented in the Neural Network Toolbox™ software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets 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

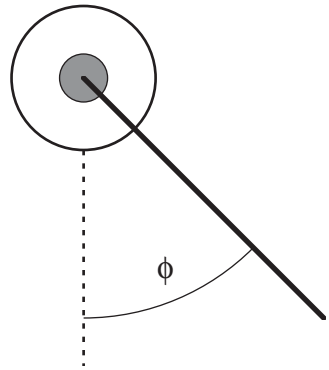
As with the controller, you can set the number of delays. The next section demonstrates how you can set the parameters.


Neural Network Controller
Neural Network Plant Model

Using the Model Reference Controller Block

This section demonstrates how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Neural Network Toolbox blockset to your model window. See your Simulink® documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the model reference controller. In this demo, 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

$$\frac{d^2y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

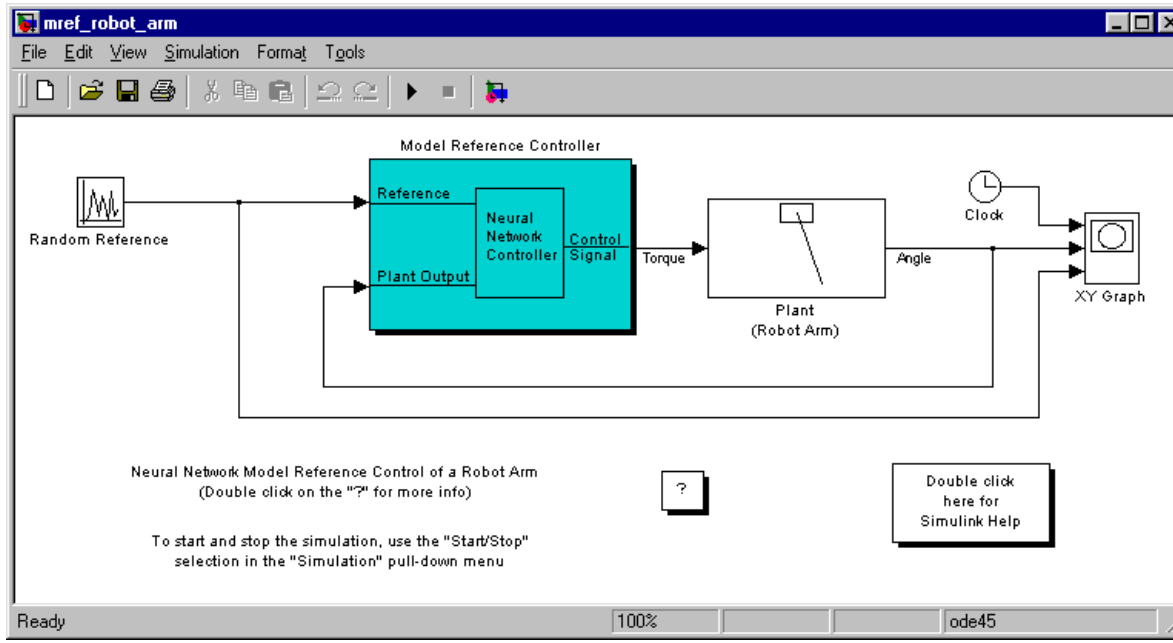
where y_r is the output of the reference model, and r is the input reference signal.

This demo 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.

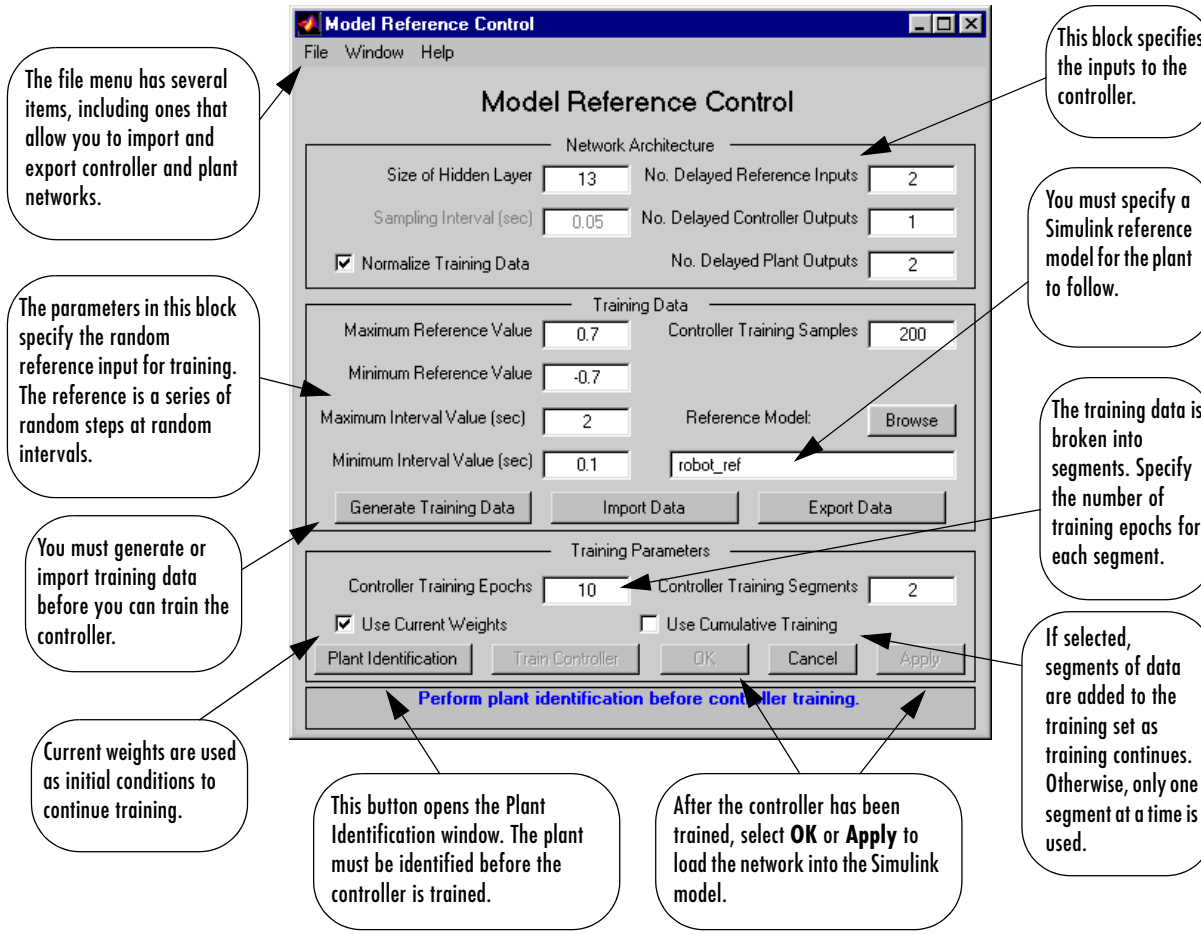
To run this demo, follow these steps.

1 Start MATLAB®.

2 Run the demo model by typing `mrefrobotarm` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The Model Reference Control block is already in the model.

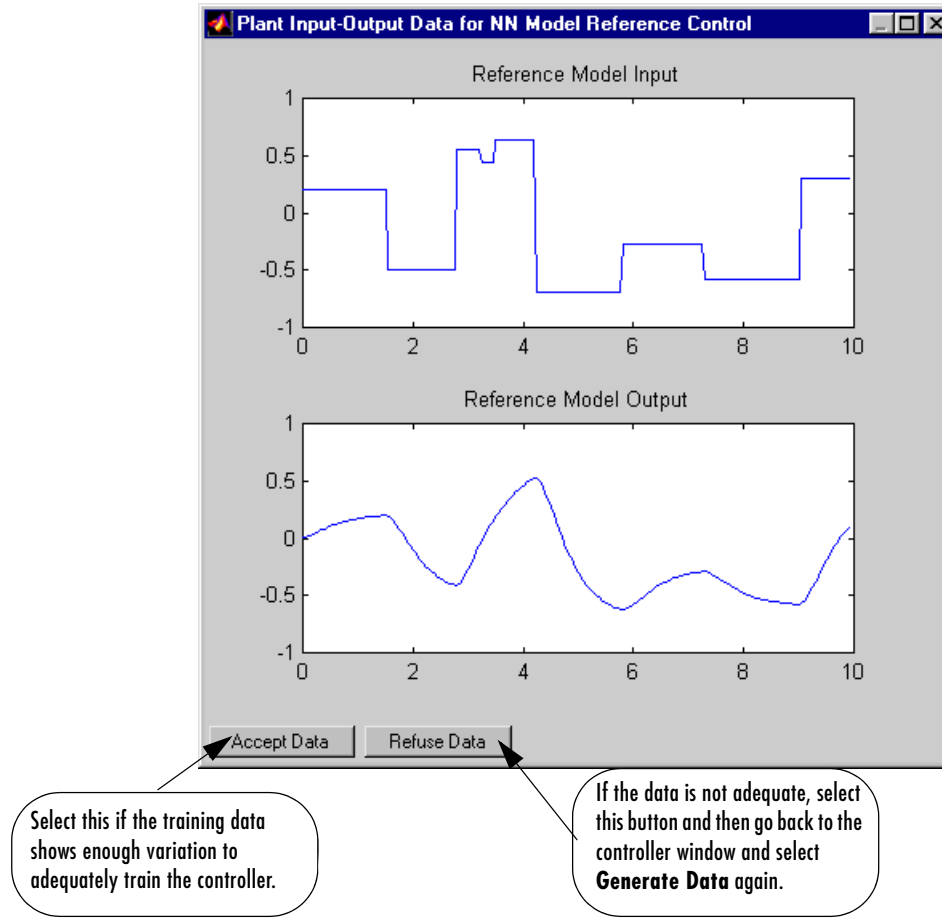


3 Double-click the Model Reference Control block. This brings up the following window for training the model reference controller.

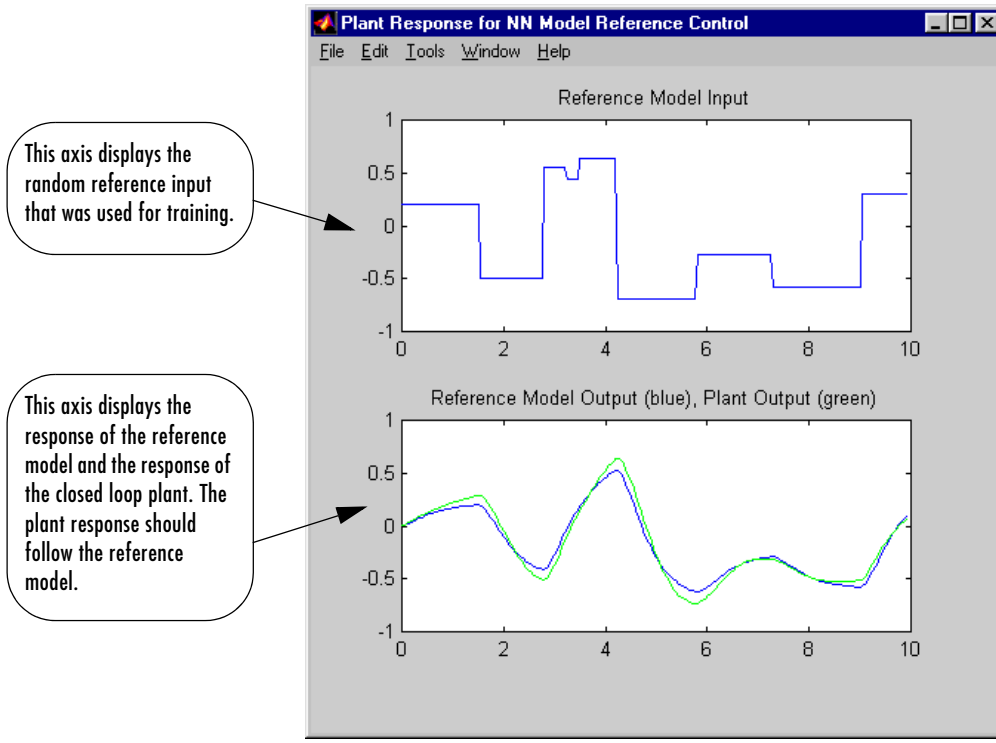


4 The next step would normally be to select **Plant Identification**, which opens the **Plant Identification** window. You would then train the plant model. Because the **Plant Identification** window is identical to the one used with the previous controllers, that process is omitted here.

5 Select **Generate Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.

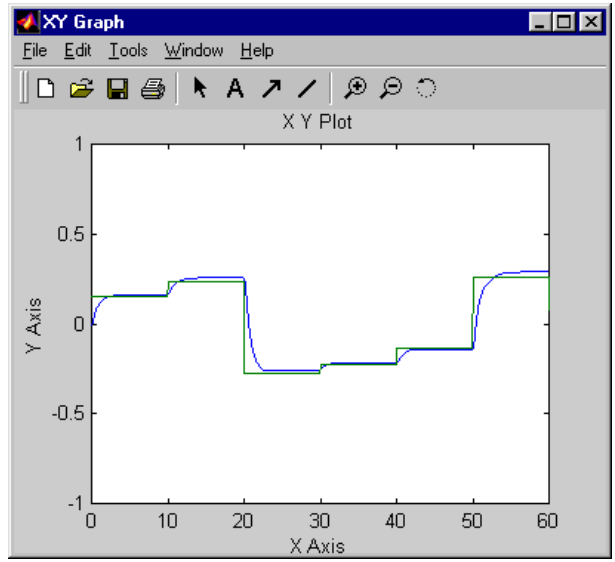


6 Select **Accept Data**. Return to the **Model Reference Control** window and select **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues, one segment at a time, until the 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]). 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 demonstration, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.

8 Return to the Simulink model and start the simulation by selecting the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
1
0.5
0
-0.5
-1
Y Axis
0 10 20 30 40 50 60
X Axis

Importing and Exporting

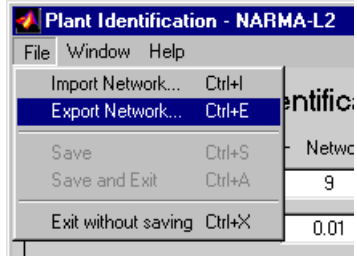
You can save networks and training data to the workspace or to a disk file. The following two sections demonstrate how you can do this.

Importing and Exporting Networks

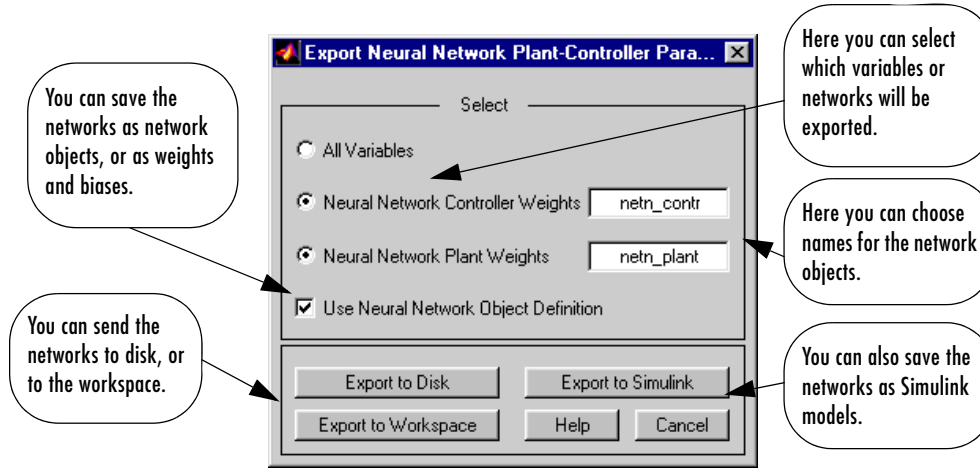
The controller and plant model networks that you develop are stored within Simulink® controller blocks. At some point you might want to transfer the networks into other applications, or you might want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following demonstration leads you through the export and import processes. (The NARMA-L2 window is used for this demonstration, but the same procedure applies to all the controllers.)

1 Repeat the first three steps of the NARMA-L2 demonstration "Using the NARMA-L2 Controller Block" on page 7-19. The **NARMA-L2 Plant Identification** window should then be open.

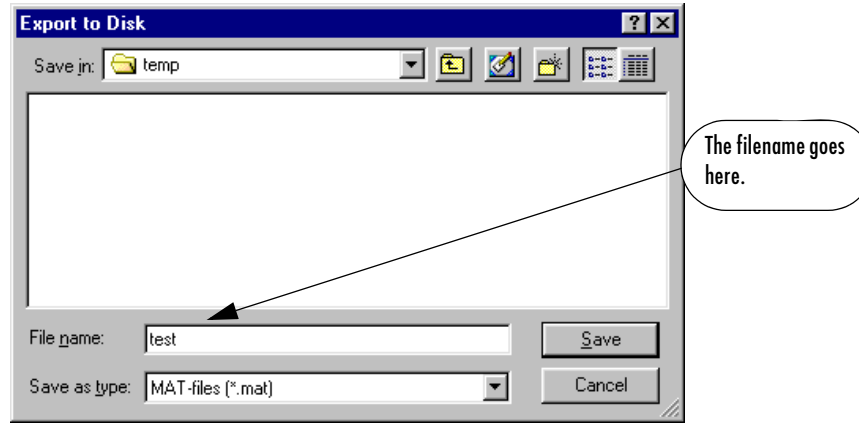
2 Select **Export** from the **File** menu, as shown below.



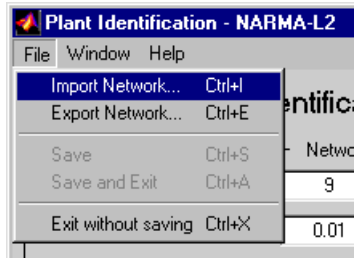
This causes the following window to open.



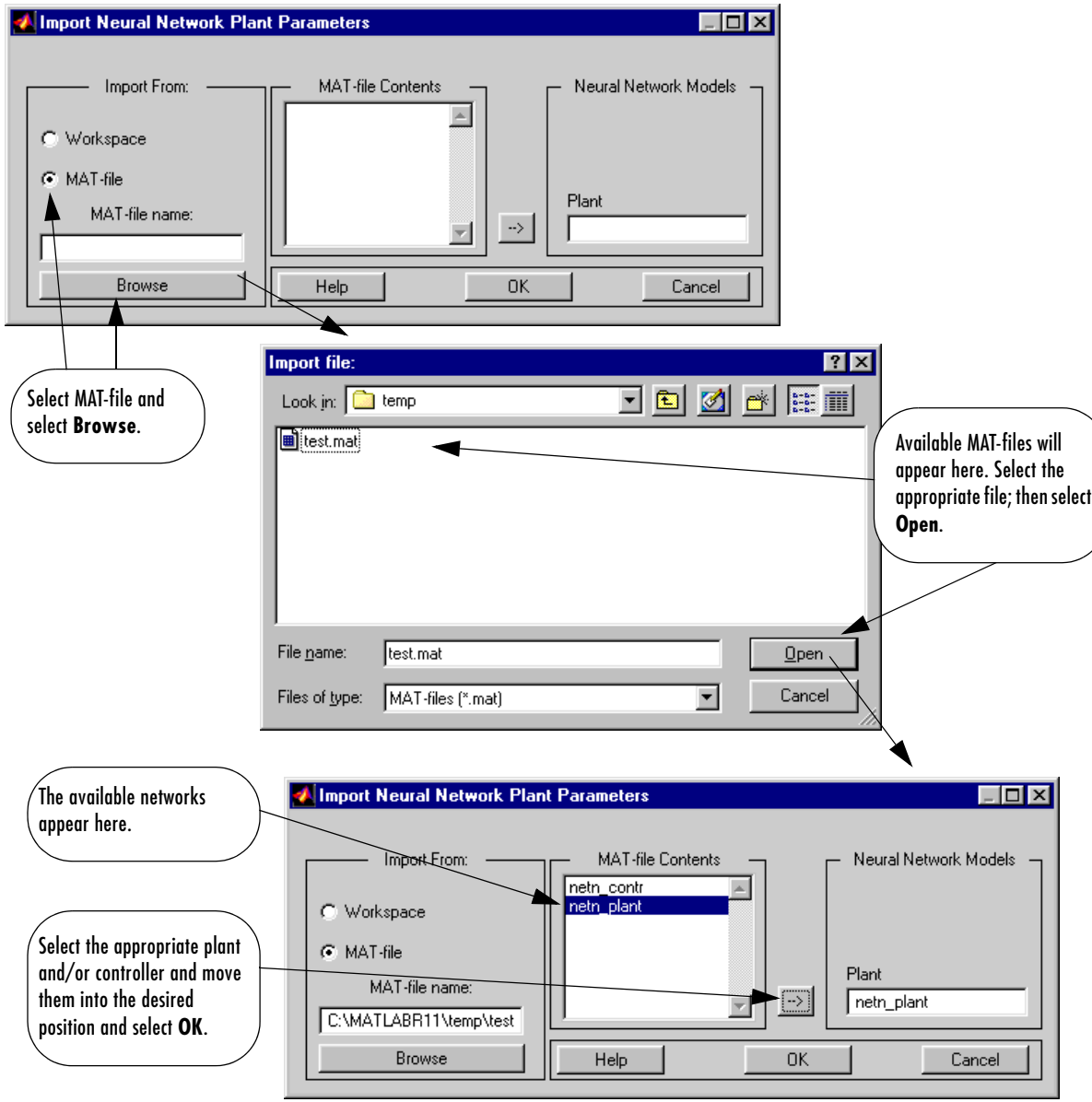
3 Select **Export to Disk**. The following window opens. Enter the filename `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



4 Retrieve that data with the **Import** menu option. Select **Import Network** from the **File** menu, 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 selecting **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 don't need to import both networks.

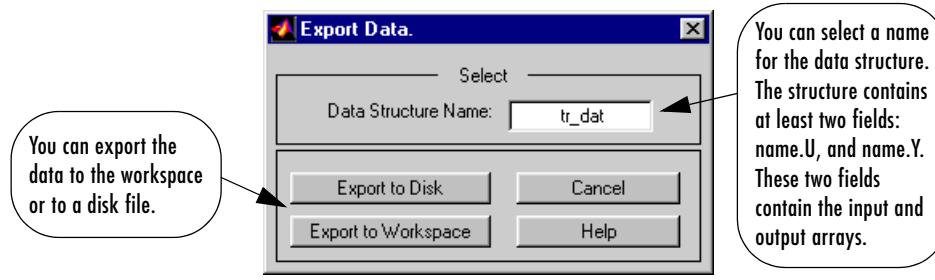

Import Neural Network Plant Parameters
Import From:
Workspace
MAT-file
MAT-file name:
Browse
MAT-file Contents
Neural Network Models
Plant
Help
OK
Cancel
Select MAT-file and select **Browse**.
Import file:
Look in:
temp
test.mat
File name:
test.mat
Files of type:
MAT-files (*.mat)
Open
Cancel
Available MAT-files will appear here. Select the appropriate file; then select **Open**.
The available networks appear here.
Select the appropriate plant and/or controller and move them into the desired position and select **OK**.
netn_contr
netn_plant
C:\MATLABR11\temp\test
netn_plant

Importing and Exporting Training Data

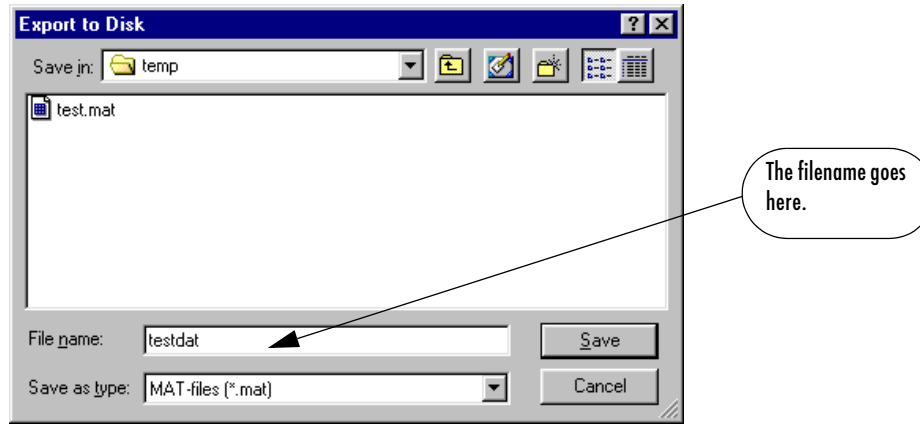
The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You might want to save the training data to the workspace or to a disk file so that you can load it again at a later time. You might also want to combine data sets manually and then load them back into the training window. You can do this by using the **Import** and **Export** buttons. The following demonstration leads you through the import and export processes. (The NN Predictive Control window is used for this demonstration, but the same procedure applies to all the controllers.)

1 Repeat the first five steps of the NN Predictive Control demonstration "Using the NN 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.

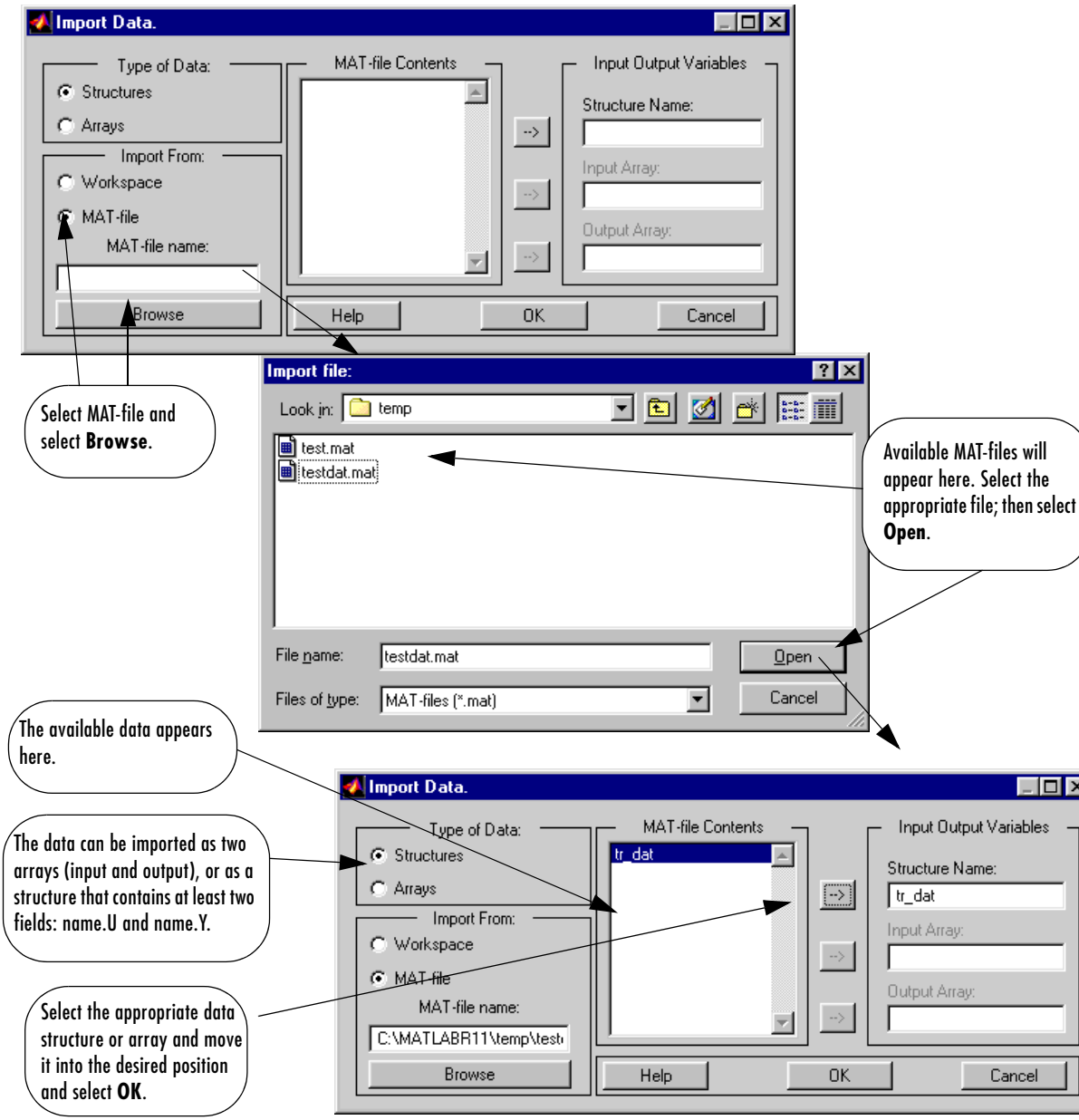
2 Select the **Export** button. This causes the following window to open.



3 Select **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. Select the **Import** button in the Plant Identification window. This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.


Import Data.
Type of Data:
Structures
Arrays
Import From:
Workspace
MAT-file
MAT-file name:
Browse
MAT-file Contents
Input Output Variables
Structure Name:
Input Array:
Output Array:
Help
OK
Cancel
Select MAT-file and select **Browse**.
Import file:
Look in:
temp
test.mat
testdat.mat
File name:
testdat.mat
Files of type:
MAT-files (*.mat)
Open
Cancel
Available MAT-files will appear here. Select the appropriate file; then select **Open**.
The available data appears here.
The data can be imported as two arrays (input and output), or as a structure that contains at least two fields: name.U and name.Y.
Select the appropriate data structure or array and move it into the desired position and select **OK**.
tr_dat
C:\MATLABR11\temp\test

8

Radial Basis Networks

Introduction (p. 8-2)
Radial Basis Functions (p. 8-3)
Probabilistic Neural Networks (p. 8-9)

Introduction

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 chapter 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

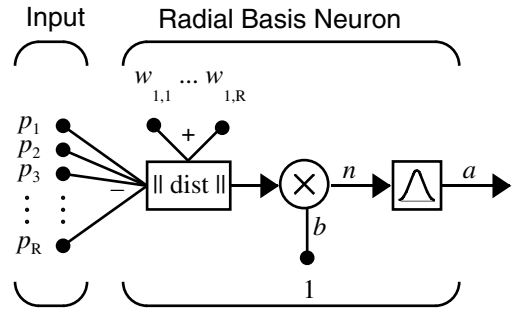
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

Type `help radbasis` to see a listing of all functions and demonstrations related to radial basis networks.

Radial Basis Functions

Neuron Model

Here is a radial basis network with R inputs.

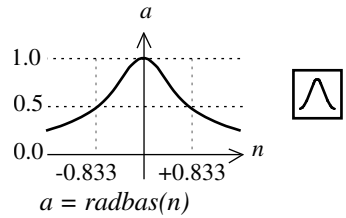


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The `|| dist ||` box in this figure accepts the input vector **p** and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is

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

Here is a plot of the `radbas` transfer function.



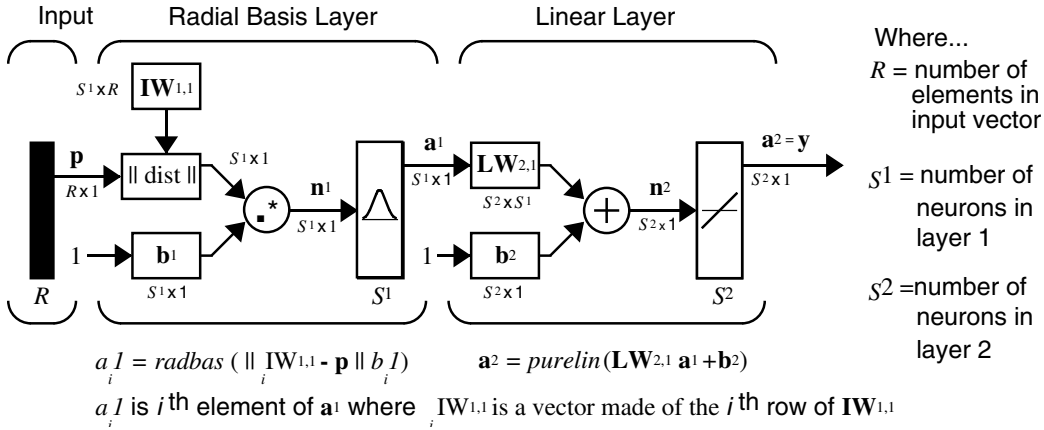
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **w**.

The bias *b* allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector **p** at vector distance of 8.326 (0.8326/*b*) from its weight vector **w**.

Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S^1 neurons, and an output linear layer of S^2 neurons.



The `|| dist ||` box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the distances between the input vector and vectors $_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector $\mathbf{b}^1$ and the output of `|| dist ||` are combined with the MATLAB® operation .* , which does element-by-element multiplication.

The output of the first layer for a feedforward network `net` can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All the details of designing this network are built into design functions `newrbe` and `newrb`, and you can obtain their outputs with `sim`.

You can understand how this network behaves by following an input vector **p** through the network to the output $\mathbf{a}^2$. If you present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1, its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0's (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] = 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/[P; 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 demonstration.

Demonstrations

The demonstration demorb1 shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Demonstrations 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 demonstrated 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. Demonstration 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.

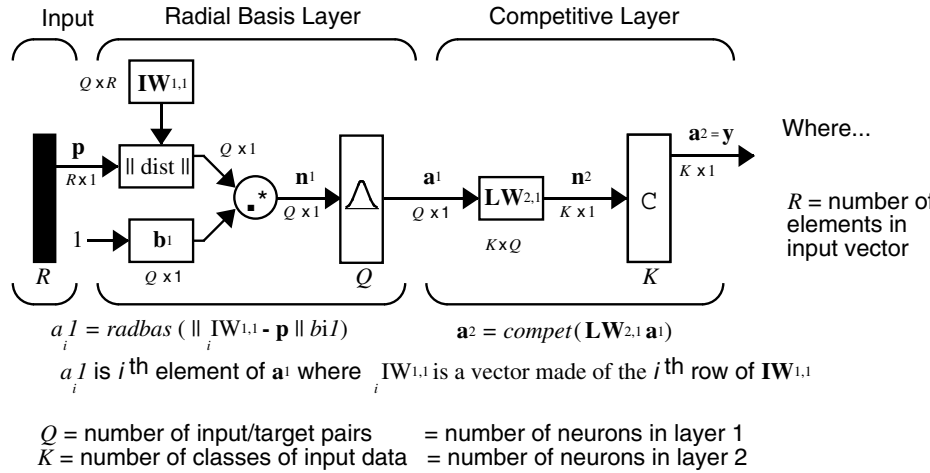
The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

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

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has `K` elements. One of these elements is 1 and the rest are 0. Thus, each input vector is associated with one of `K` classes.

The first-layer input weights, $IW^{1,1}$ (`net.IW{1,1}`), are set to the transpose of the matrix formed from the Q training pairs, $\mathbf{P}'$. When an input is presented, the `|| dist ||` box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent to the `radbas` transfer function. An input vector close to a training vector is represented by a number close to 1 in

the output vector $\mathbf{a}^1$. If an input is close to several training vectors of a single class, it is represented by several elements of $\mathbf{a}^1$ that are close to 1.

The second-layer weights, $LW^{1,2}$ (`net.LW{2,1}`), are set to the matrix **T** of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0's elsewhere. (Use function `ind2vec` to create the proper vectors.) The multiplication $\mathbf{Ta}^1$ sums the elements of $\mathbf{a}^1$ due to each of the `K` input classes. Finally, the second-layer transfer function, `compete`, produces a 1 corresponding to the largest element of $\mathbf{n}^2$, and 0's 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 1's in the right places. You can get it with the function `ind2vec`. It gives a matrix with 0's 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.

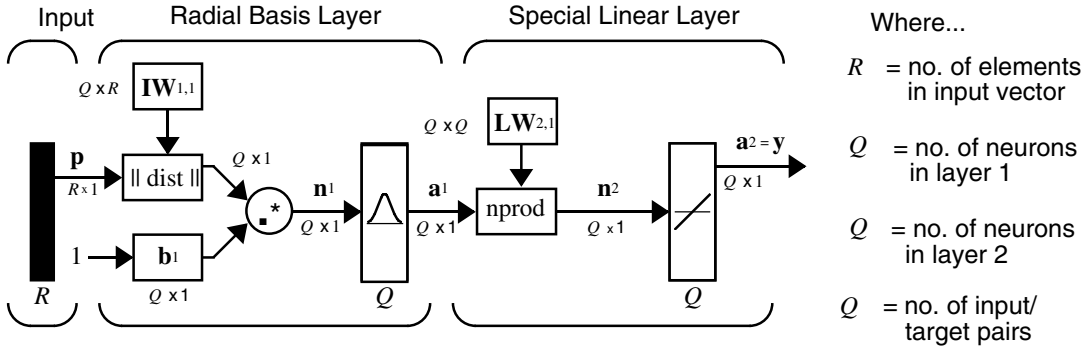
You might want to try `demopnn1`. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

Generalized Regression Networks

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

Network Architecture

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



Here the **nprod** box shown above (code function `normprod`) produces S^2 elements in vector $\mathbf{n}^2$. Each element is the dot product of a row of $LW^{2,1}$ and the input vector $\mathbf{a}^1$, all normalized by the sum of the elements of $\mathbf{a}^1$. For instance, suppose that

```
LW{2,1}= [1 -2;3 4;5 6];
a{1} = [0.7;0.3];
```

Then

```
aout = normprod(LW{2,1},a{1})
aout =
    0.1000
    3.3000
    5.3000
```

The first layer is just like that for `newrbe` networks. It has as many neurons as there are input/ target vectors in **P**. Specifically, the first-layer weights are set

to $\mathbf{P}'$. The bias $\mathbf{b}^1$ is set to a column vector of 0.8326/SPREAD. The user chooses SPREAD, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the newbe radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with dist. Each neuron's net input is the product of its weighted input with its bias, calculated with netprod. Each neuron's output is its net input passed through radbas. If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input will be spread, and its net input will be sqrt(-log(.5)) (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here LW{2,1} is set to T.

Suppose you have an input vector $\mathbf{p}$ 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 $\mathbf{a}^1$ 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 `demogrn1`. It shows how to approximate a function with a GRNN.

9

Self-Organizing and Learning Vector Quantization Nets

Introduction (p. 9-2)

Competitive Learning (p. 9-3)

Self-Organizing Feature Maps (p. 9-9)

Learning Vector Quantization Networks (p. 9-35)

Introduction

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.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner. 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 `newc` and `newsom`, respectively. You can type `help selforg` to find a listing of all self-organizing functions and demonstrations.

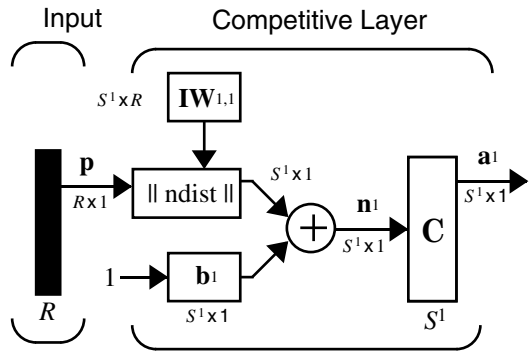
You can create an LVQ network with the function `newlvq`. For a list of all LVQ functions and demonstrations, type `help lvq`.

Competitive Learning

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

Architecture

The architecture for a competitive network is shown below.



The `|| dist ||` box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the negative of the distances between the input vector and vectors ${}_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

Compute the net input $\mathbf{n}^1$ of a competitive layer by finding the negative distance between input vector **p** and the weight vectors and adding the biases **b**. If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input $\mathbf{n}^1$. The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in "Bias Learning Rule (learncon)" on page 9-5.

Creating a Competitive Neural Network (newc)

You can create a competitive neural network with the function `newc`. 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 layer with two input elements ranging from 0 to 1. The first argument gives the ranges of the two input vectors, and the second argument says that there are to be two neurons.

```
net = newc([0 1; 0 1],2);
```

The weights are initialized to the centers of the input ranges with the function `midpoint`. You can check to see these initial values as follows:

```
wts = net.IW{1,1}
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, as you would expect when using `midpoint` for initialization.

The biases are computed by `initcon`, which gives

```
biases = net.b{1}
biases =
    5.4366
    5.4366
```

Now you have a network, but you need to train it to do the classification job.

Recall that each neuron competes to respond to an input vector **p**. If the biases are all 0, the neuron whose weight vector is closest to **p** gets the highest net input and, therefore, wins the competition and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

Kohonen Learning Rule (learnk)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the `i`th neuron wins, the elements of the `i`th row of the input weight matrix are adjusted as shown below.

$$_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1))$$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

Bias Learning Rule (learncon)

One of the limitations of competitive networks is that some neurons might not always be *allocated*. In other words, some neuron weight vectors might start out far from any input vectors and never win the competition, no matter how long the training is continued. The result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this, use biases to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently

active neurons become smaller, and biases of infrequently active neurons become larger.

As the biases of infrequently active neurons increase, the input space to which those neurons respond increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually, the neuron responds to the same number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias eventually becomes large enough so that it can win. When this happens, it moves toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk` to make sure that the running average is accurate.

Training

Now train the network for 500 epochs. You can use either `train` or `adapt`.

```
net.trainParam.epochs = 500;
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainr`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

This code produces

```
ans =
trainr
```

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)
```

This yields

```
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. They are

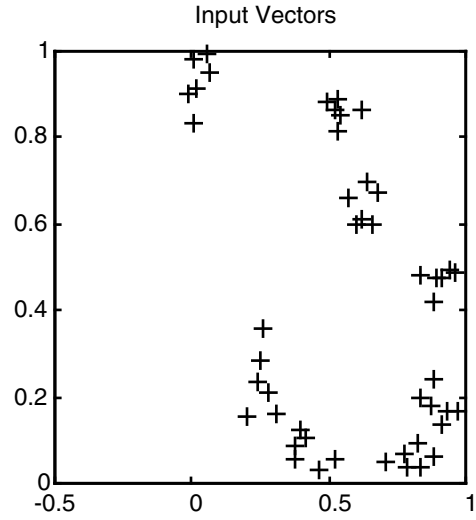
```
wts =
    0.1000    0.1467
    0.8474    0.8525
biases =
    5.4961
    5.3783
```

(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.

Self-Organizing Feature Maps

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:

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \text{ or}$$

$$_i\mathbf{w}(q) = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

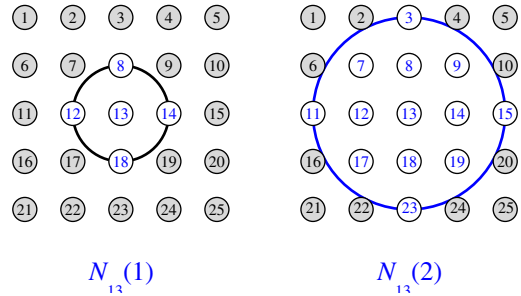
Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*.

$$N_i(d) = \{j, d_{ij} \le d\}$$

Thus, when a vector **p** is presented, the weights of the winning neuron *and* its close neighbors move toward **p**. Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.



These neighborhoods could be written as

$$N_{13}(1) = \{8, 12, 13, 14, 18\} \text{ and}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$

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.

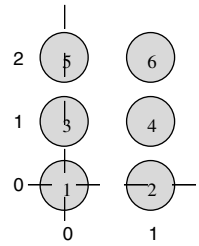
Topologies (gridtop, hextop, randtop)

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.



gridtop(2,3)

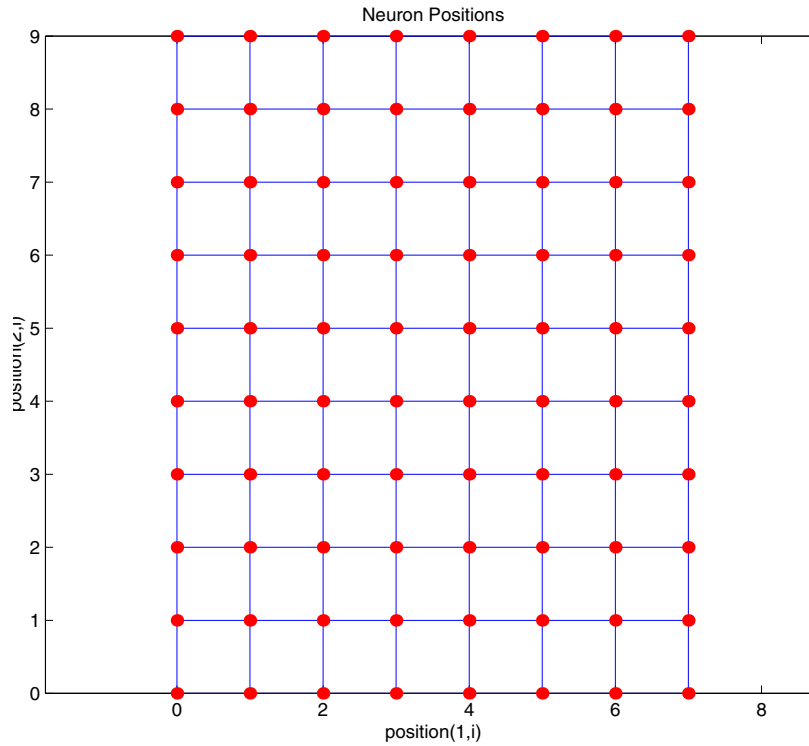
Note that had you asked for a `gridtop` with the arguments reversed, you would have gotten a slightly different arrangement:

```
pos = gridtop(3,2)
pos =
     0     1     2     0     1     2
     0     0     0     1     1     1
```

An 8-by-10 set of neurons in a `gridtop` topology can be created and plotted with the following code:

```
pos = gridtop(8,10);
plotsom(pos)
```

to give the following graph.



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
```

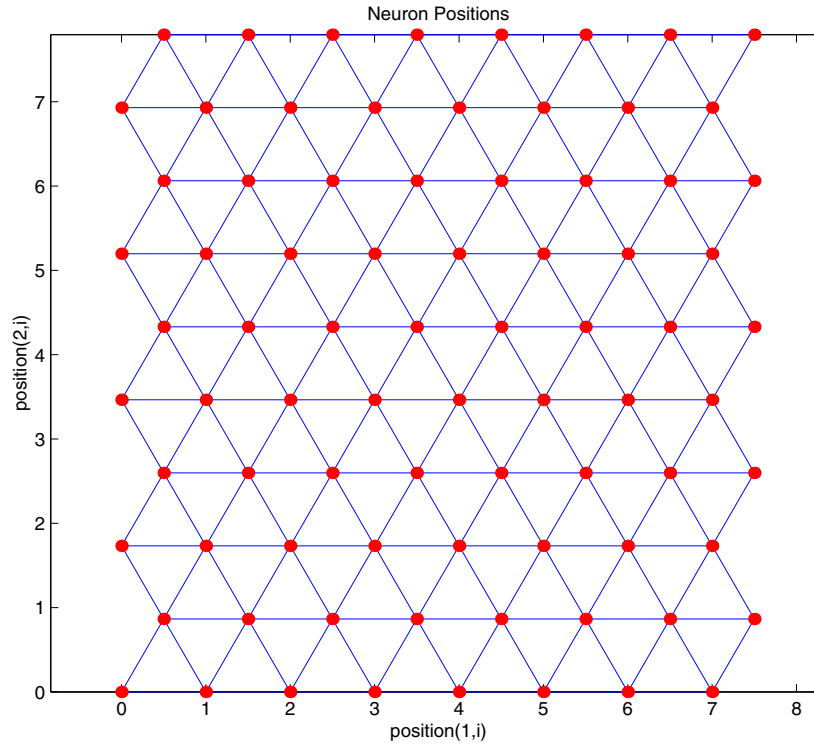
Note that `hextop` is the default pattern for SOFM networks generated with `newsom`.

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)
```

to give the following graph.



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.

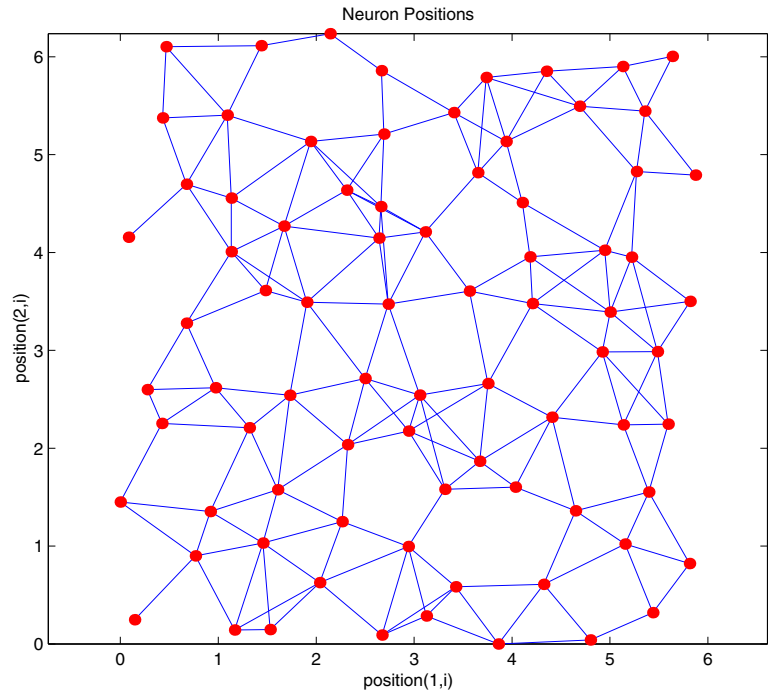
```
pos = randtop(2,3)
pos =
         0    0.7620    0.6268    1.4218    0.0663    0.7862
    0.0925         0    0.4984    0.6007    1.1222    1.4228
```

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)
```

to give the following graph.



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 has been discussed before. It calculates the Euclidean distance from a *home* neuron to any other neuron. Suppose you have three neurons:

```
pos2 = [0 1 2; 0 1 2]
pos2 =
```

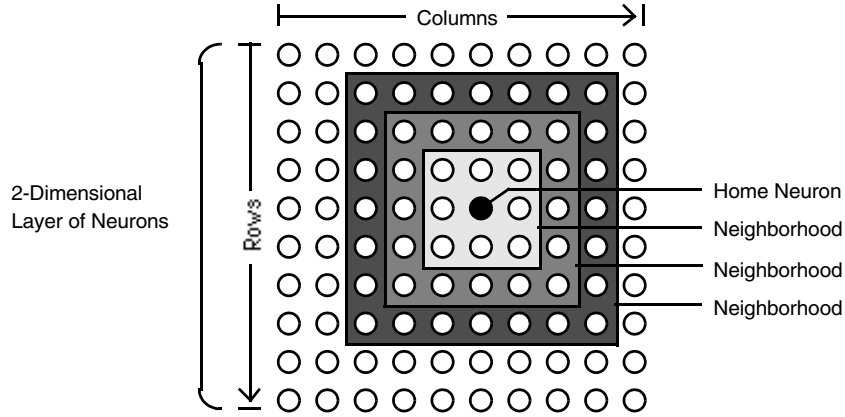
```
     0     1     2
     0     1     2
```

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 1.414, etc. These are indeed the Euclidean distances as you know them.

The graph below shows a home neuron in a two-dimensional (`gridtop`) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1 includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the `dist` function, all the neighborhoods for an S-neuron layer map are represented by an S-by-S matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function `boxdist`. Suppose that you have six neurons in a `gridtop` configuration.

```
pos = gridtop(2,3)
pos =
     0     1     0     1     0     1
     0     0     1     1     2     2
```

Then the box distances are

```
d = boxdist(pos)
d =
     0     1     1     1     2     2
     1     0     1     1     2     2
     1     1     0     1     1     1
     1     1     1     0     1     1
     2     2     1     1     0     1
     2     2     1     1     1     0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if you calculate the distances from the same set of neurons with `linkdist`, you get

```
dlink =
     0     1     1     2     2     3
     1     0     2     1     3     2
     1     2     0     1     1     2
     2     1     1     0     2     1
     2     3     1     2     0     1
     3     2     2     1     1     0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

Thus if you have

```
W1 = [1 2; 3 4; 5 6]
W1 =
     1     2
     3     4
     5     6
```

and

```
P1 = [1;1]
P1 =
     1
     1
```

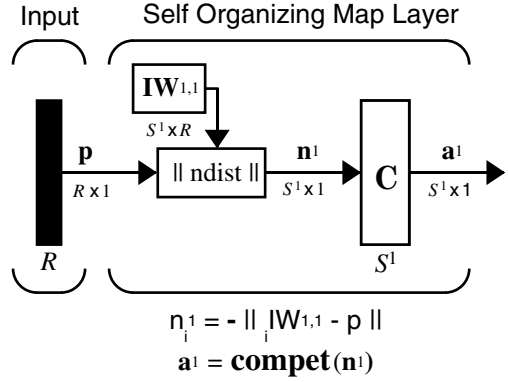
then you get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
     1
     5
     9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element $\mathbf{a}^1_i$ corresponding to i^*, the winning neuron. All other output elements in $\mathbf{a}^1$ are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. You can choose from various topologies of neurons. Similarly, you can choose from various distance expressions to calculate neurons that are close to the winning neuron.

Creating a Self-Organizing MAP Neural Network (newsom)

You can create a new SOM network with the function `newsom`. 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 that fall in the ranges 0 to 2 and 0 to 1, respectively. Further suppose that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is

```
net = newsom([0 2; 0 1],[2 3]);
```

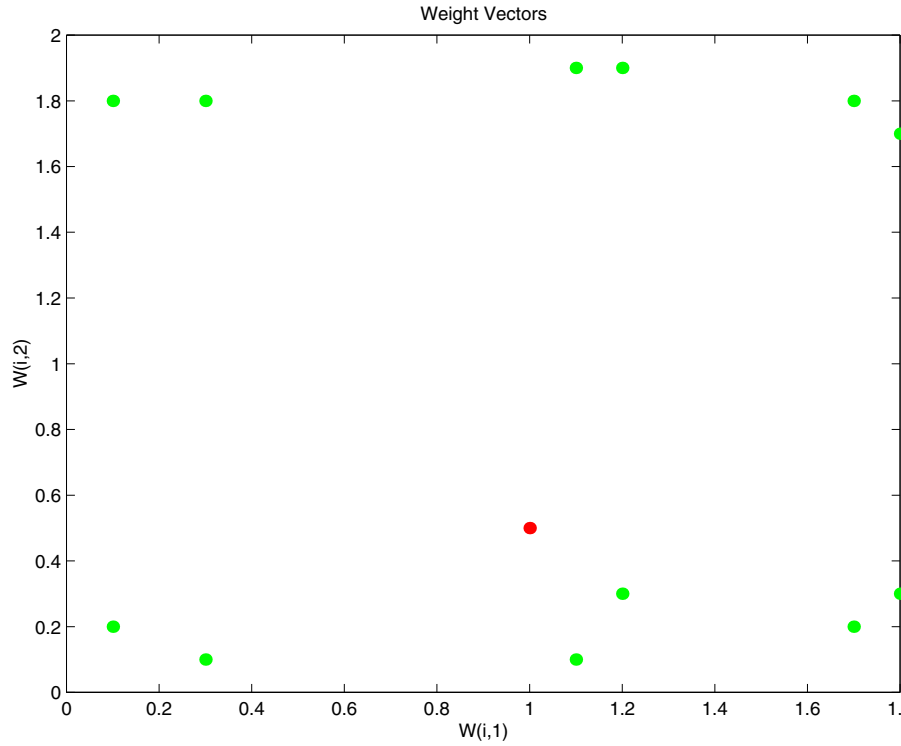
Suppose also that the vectors to train on are

```
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 plot all of this with

```
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

to give



The various training vectors are seen as fuzzy gray spots around the perimeter of this figure. The initialization for `newsom` is `midpoint`. Thus, the initial network neurons are all concentrated at the black spot at (1, 0.5).

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 (`compete`) 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 (`trainbuwb`). 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 distance stays at the tuning neighborhood distance, (which should include only close neighbors, i.e., typically 1.0). The small neighborhood fine-tunes the network, while keeping the ordering learned in the previous phase stable.

Now take a look at some of the specific values commonly used in these networks.

Learning occurs according to the `learnsomb` learning parameter, shown here with its default value.

Learning Parameter	Default Value	Purpose
`LP.init_neighborhood`	3	Initial neighborhood size
`LP.steps`	100	Ordering phase steps

The neighborhood size `NS` is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as `LP.steps`. During this phase, the algorithm adjusts `ND` from the initial neighborhood size `LP.init_neighborhood` down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, `ND` is always set to 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.

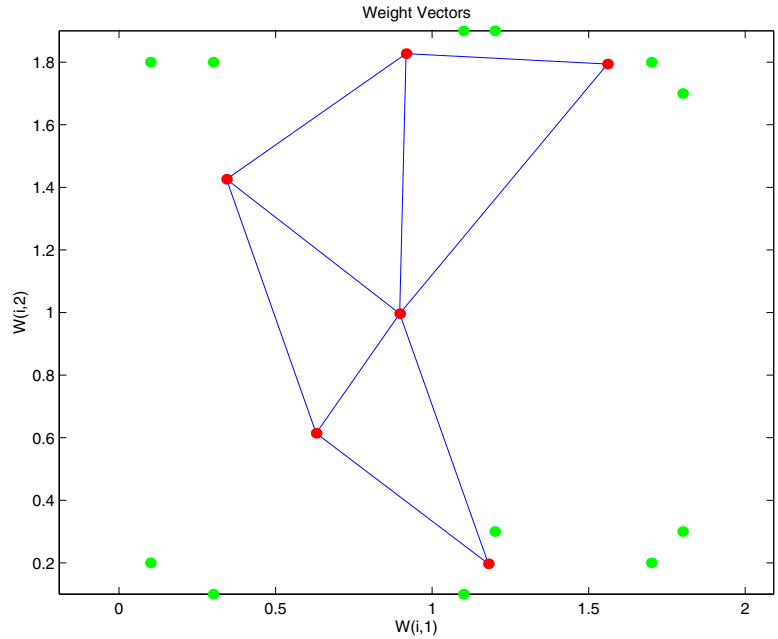
As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature 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);
```

Call `plotsom` to see the data produced by the training procedure, shown in the following plot.



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 might try the demonstrations `demosm1` and `demosm2` to see similar examples.

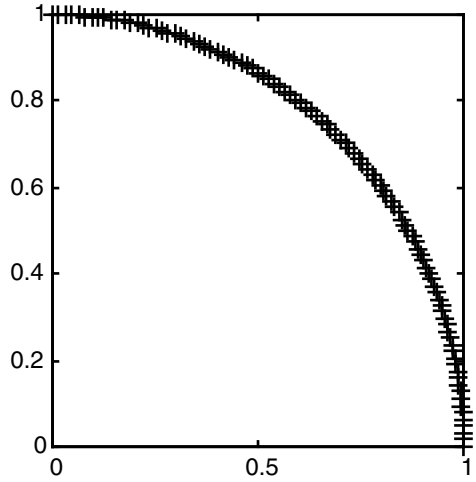
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

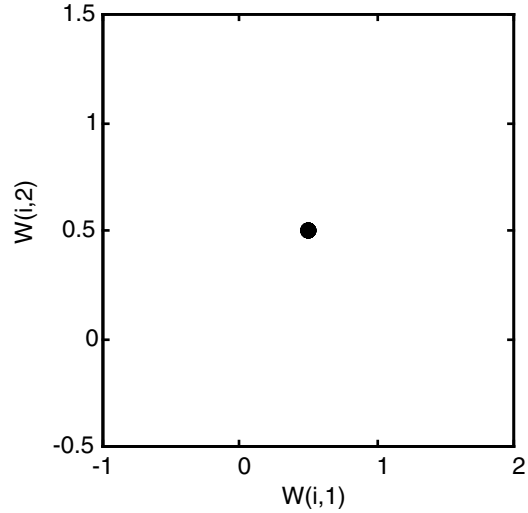
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

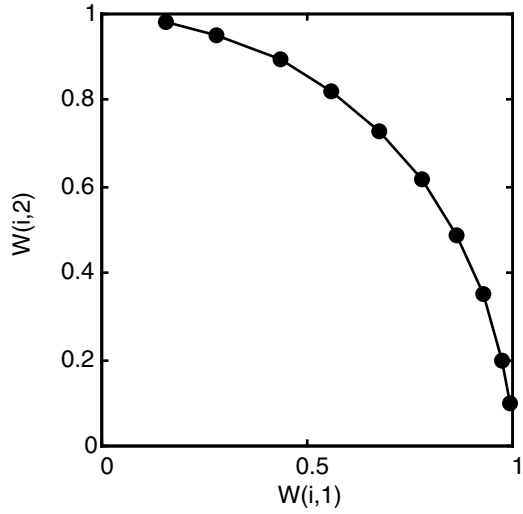


A a self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



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.

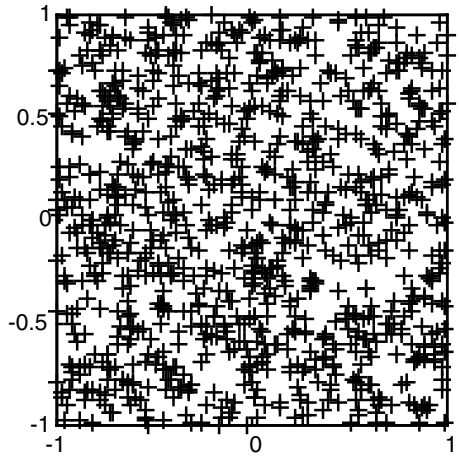
Two-Dimensional Self-Organizing Map

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);
```

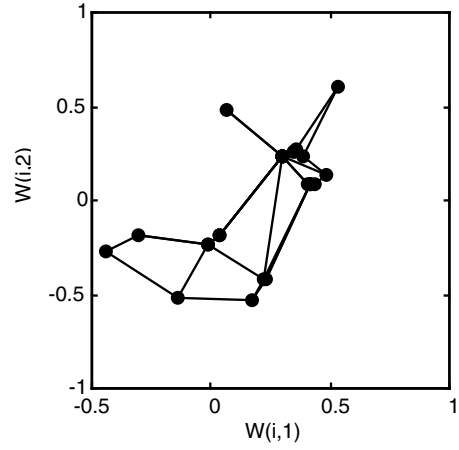
Here is a plot of these 1000 input vectors.



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`.

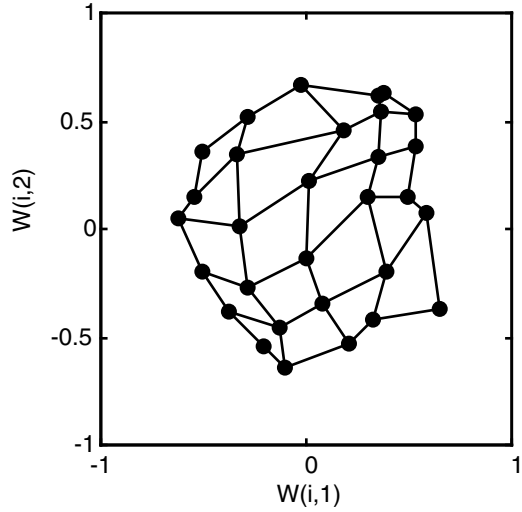
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



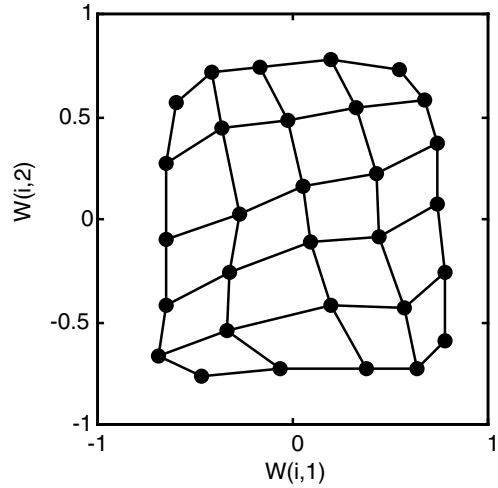
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

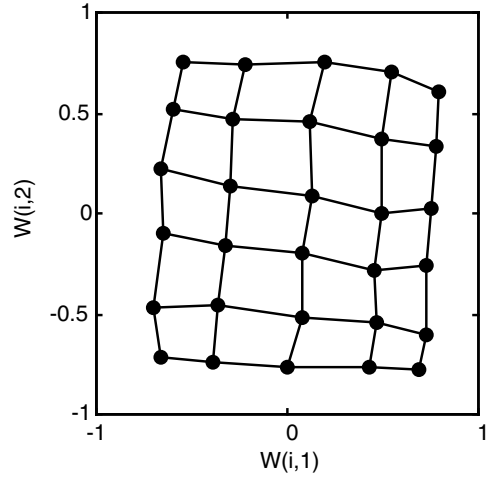


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

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.

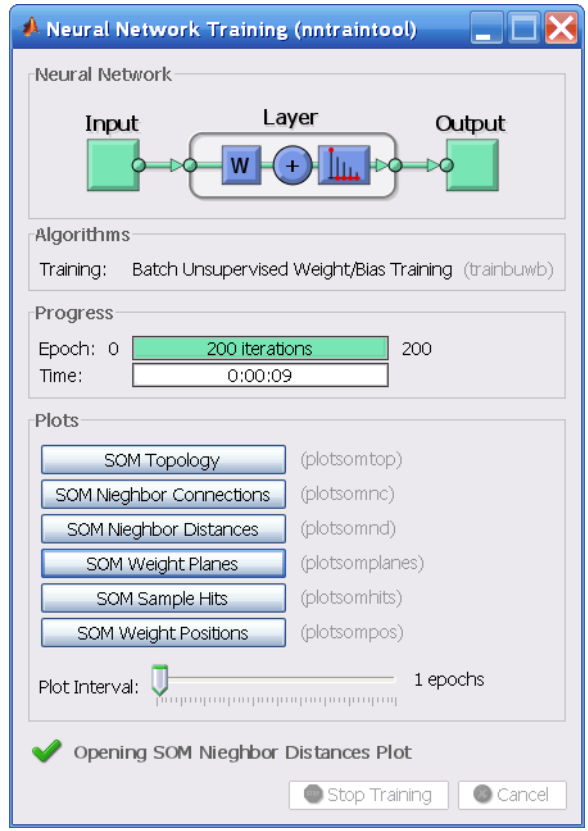
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

Training with the Batch Algorithm

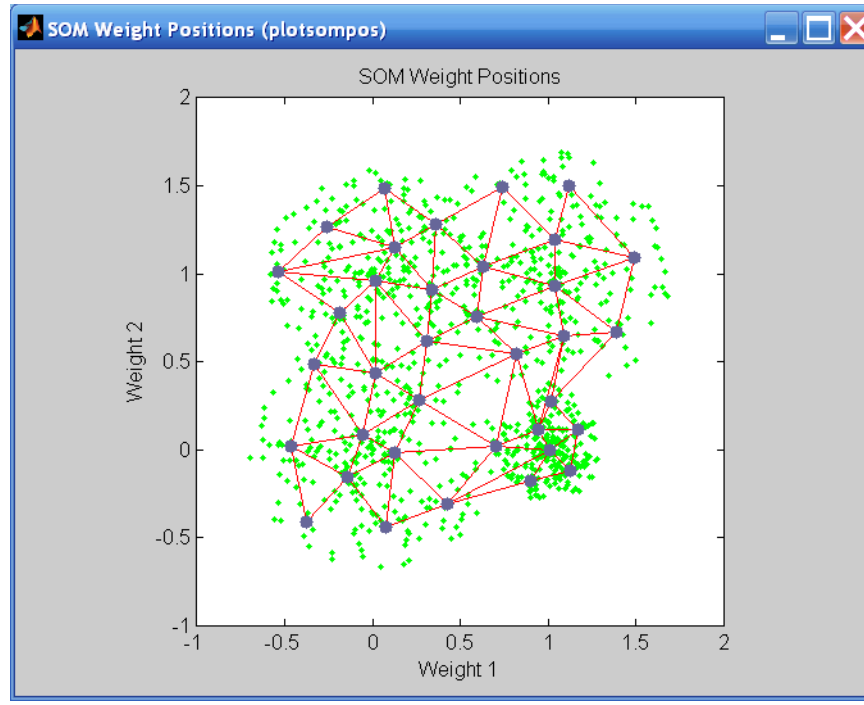
The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
load simplecluster_dataset
net = newsom(simpleclusterInputs,[6 6]);
[net2,tr] = train(net,simpleclusterInputs);
```

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.



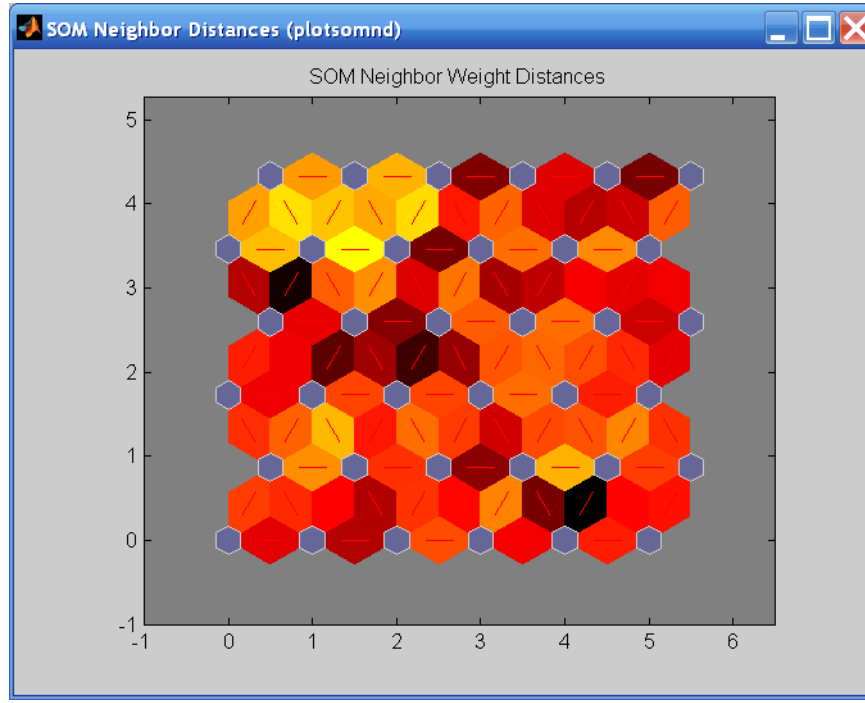
When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

This figure uses the following color coding:

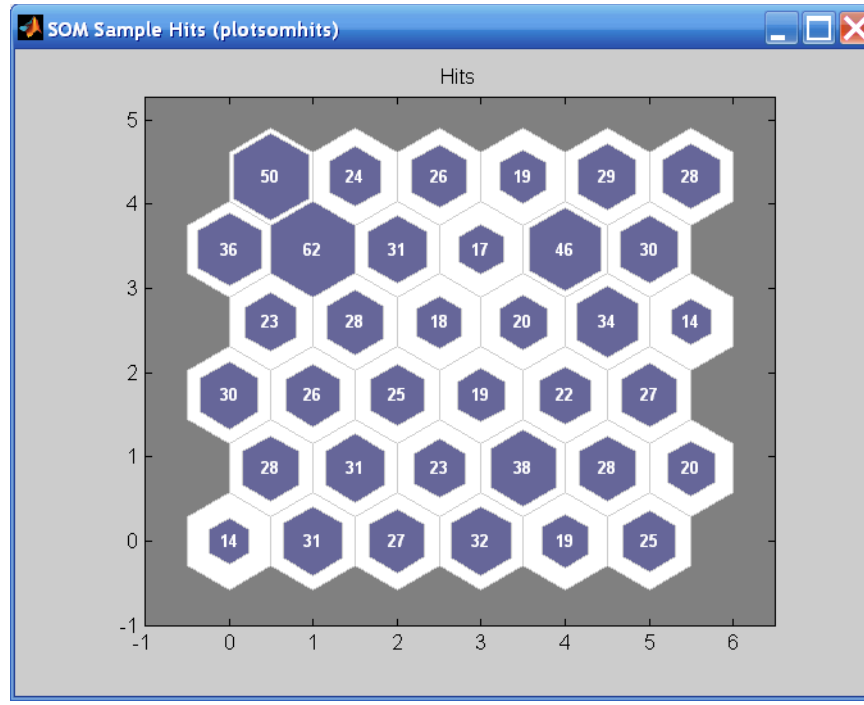
- The blue hexagons represent the neurons.
- The red lines connect neighboring neurons.
- The colors in the regions containing the red lines indicate the distances between neurons.
- The darker colors represent larger distances.
- The lighter colors represent smaller distances.

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

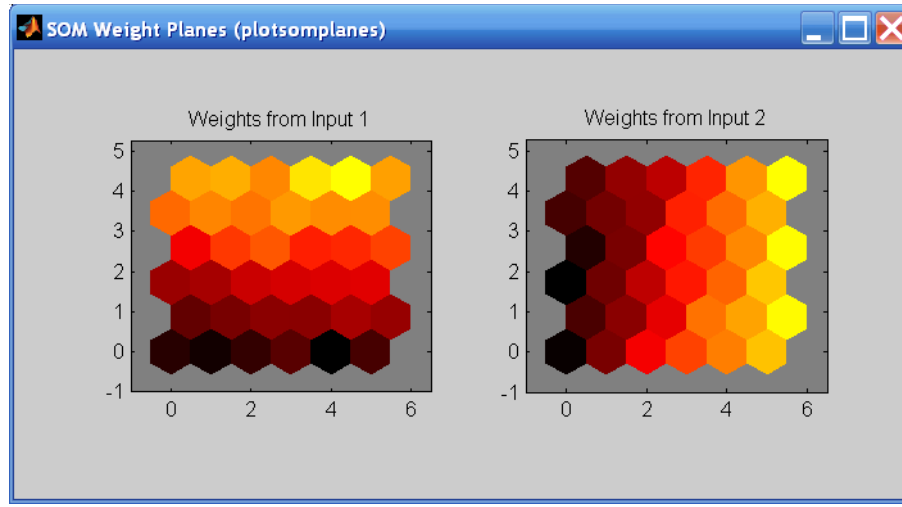
tightly clustered data points. The corresponding weights are closer together in this region, which is indicated by the lighter colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor 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. (Darker colors represent larger weights.) 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.

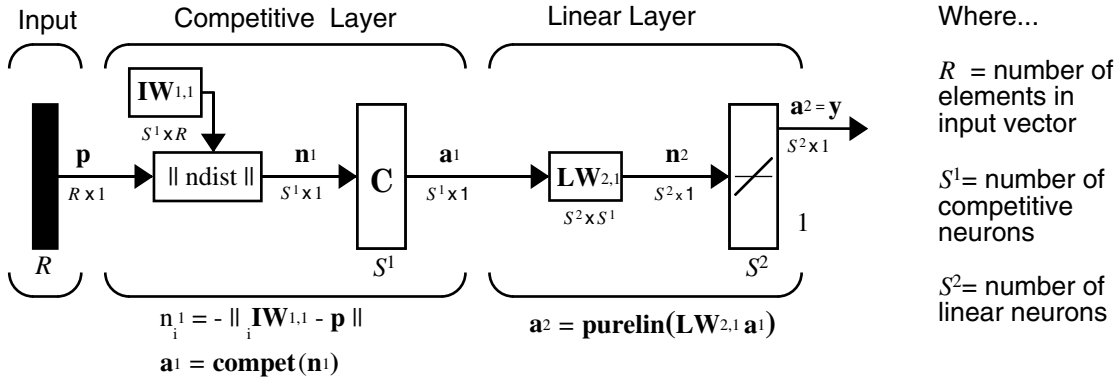


You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`. (See their reference pages for details.)

Learning Vector Quantization Networks

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 "Self-Organizing and Learning Vector Quantization Nets" described in this chapter. 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 1's 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 9-40. First, consider how to create the original network.

Creating an LVQ Network (newlvq)

You can create an LVQ network with the function `newlvq`,

```
net = newlvq(PR,S1,PC,LR,LF)
```

where

- `PR` is an R-by-2 matrix of minimum and maximum values for R input elements.
- S^1 is the number of first-layer hidden neurons.
- `PC` is an S^2-element vector of typical class percentages.
- `LR` is the learning rate (default 0.01).
- `LF` is the learning function (default is `learnlv1`).

Suppose you have 10 input vectors. Create a network that assigns each of these input vectors to one of four subclasses. Thus, there are four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2 0 0 0 0 2 2 3; 0 1 -1 2 1 -1 -2 1 -1 0];
```

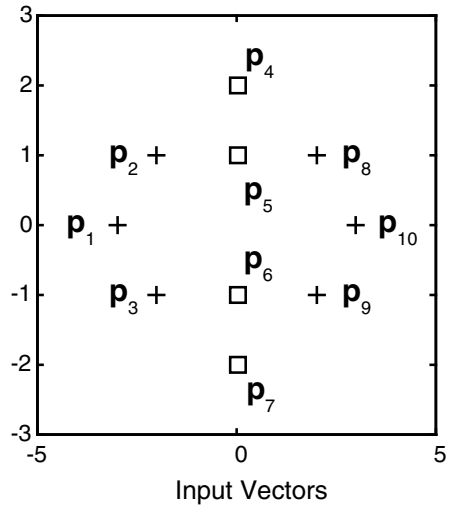
and

```
Tc = [1 1 1 2 2 2 2 1 1 1];
```

It might help to show the details of what you get from these two lines of code.

```
P =
    -3    -2    -2     0     0     0     0     2     2     3
     0     1    -1     2     1    -1    -2     1    -1     0
Tc =
     1     1     1     2     2     2     2     1     1     1
```

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. You want a network that classifies $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_8$, $\mathbf{p}_9$, and $\mathbf{p}_{10}$ to produce an output of 1, and that classifies vectors $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$, and $\mathbf{p}_7$ to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next convert the `Tc` matrix to target vectors.

```
T = ind2vec(Tc);
```

This gives a sparse matrix `T` that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
     1     1     1     0     0     0     0     1     1     1
     0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if you have the first column of `P` as input, you should get the first column of `targets` as an output; and that output says the input falls in class 1, which is correct. Now you are ready to call `newlvq`.

Call `newlvq` with the proper arguments so that it creates a network with four neurons in the first layer and two neurons in the second layer. The first-layer weights are initialized to the centers of the input ranges with the function `midpoint`. The second-layer weights have 60% (6 of the 10 in `Tc` above) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns will have a 1 in the second row (corresponding to class 2).

```
net = newlvq(P,4,[.6 .4]);
```

Confirm the initial values of the first-layer weight matrix.

```
net.IW{1,1}
ans =
     0     0
     0     0
     0     0
     0     0
```

These zero weights are indeed the values at the midpoint of the ranges (-3 to +3) of the inputs, as you would expect when using `midpoint` for initialization.

You can look at the second-layer weights with

```
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 = sim(net,P);
Yc = vec2ind(Y)
Yc =
     1     1     1     1     1     1     1     1     1     1
```

The network classifies all inputs into class 1. Because this is not what you want, you have to train the network (adjusting the weights of layer 1 only), before you can expect a good result. The next two sections discuss two LVQ learning rules and the training process.

LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right\}$$

Here there are input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector $\mathbf{p}$ 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 `ndist`. The hidden neurons of layer 1 compete. Suppose that the ith element of $\mathbf{n}^1$ is most positive, and neuron i^* wins the competition. Then the competitive

transfer function produces a 1 as the i^*th element of $\mathbf{a}^1$. All other elements of $\mathbf{a}^1$ are 0.

When $\mathbf{a}^1$ is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in $\mathbf{a}^1$ selects the class k^* associated with the input. Thus, the network has assigned the input vector $\mathbf{p}$ to class k^* and $a^2_{k^*}$ will be 1. Of course, this assignment can be a good one or a bad one, for t_{k^*} can be 1 or 0, depending on whether the input belonged to class k^* or not.

Adjust the i^*th row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector $\mathbf{p}$ 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,

$$(a^2_{k^*} = t_{k^*} = 1)$$

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,

$$(a^2_{k^*} = 1 \neq t_{k^*} = 0)$$

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`:

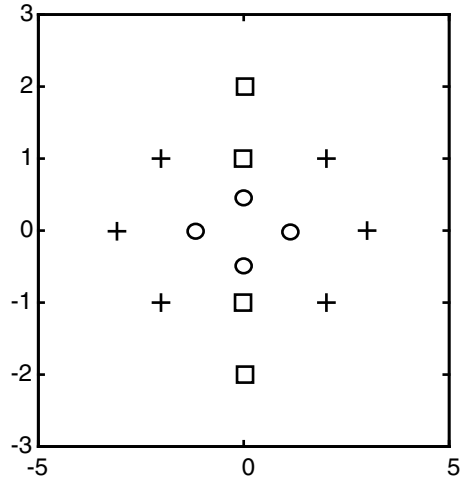
```
net.trainParam.epochs = 150;
```

```
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.



Weights (circles) after training

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 = sim(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 = sim(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 = sim(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 demonstration 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]) 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 `learnlv1` 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

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s \quad \text{where} \quad s \equiv \frac{1-w}{1+w}$$

(where d_i and d_j are the Euclidean distances of 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 **p** and ${}_{j*}\mathbf{IW}^{1,1}$ belong to the same class, and **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.

10

Adaptive Filters and Adaptive Training

Introduction (p. 10-2)
Linear Neuron Model (p. 10-3)
Adaptive Linear Network Architecture (p. 10-4)
Least Mean Square Error (p. 10-7)
LMS Algorithm (learnwh) (p. 10-8)
Adaptive Filtering (adapt) (p. 10-9)

Introduction

The ADALINE (adaptive linear neuron) networks discussed in this chapter 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 only solve 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 chapter, 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 cancellation, 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 chapter.

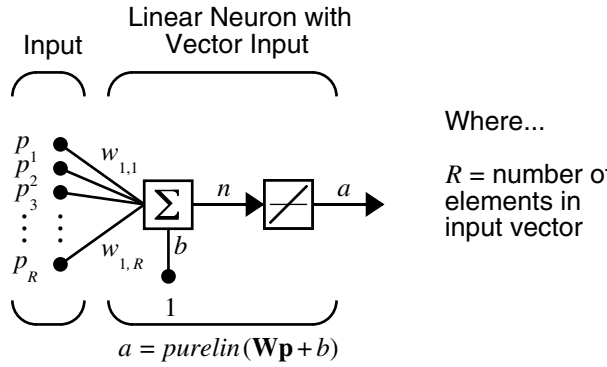
Important Adaptive Functions

This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

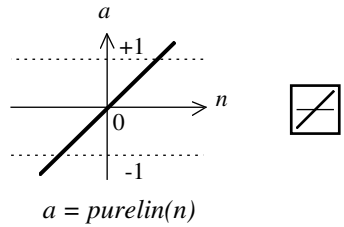
You can type `help linnet` to see a list of linear and adaptive network functions, demonstrations, and applications.

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

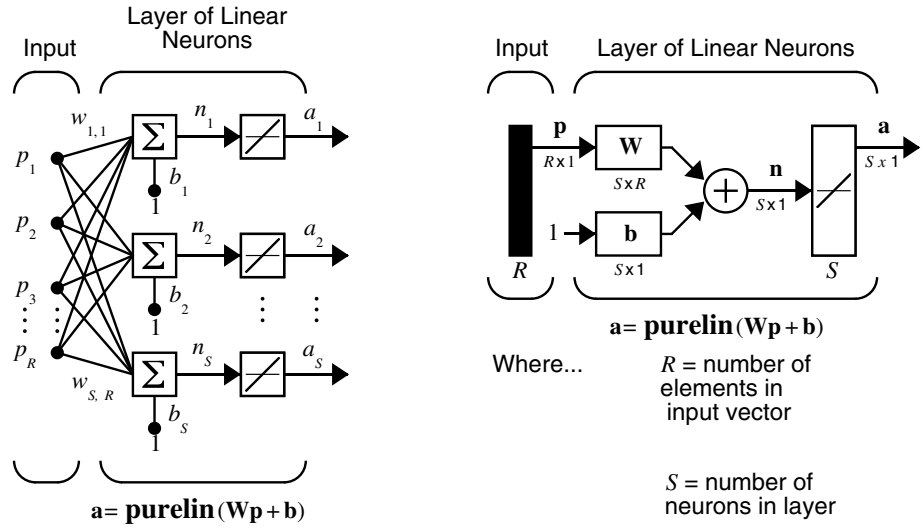
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p}+b) = \mathbf{W}\mathbf{p}+b$$

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 **W**.

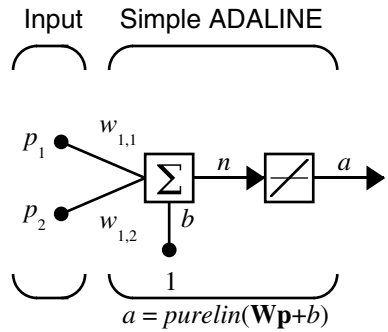


This network is sometimes called a MADALINE for Many ADALINEs. Note that the figure on the right defines an S-length output vector **a**.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (newlin)

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.



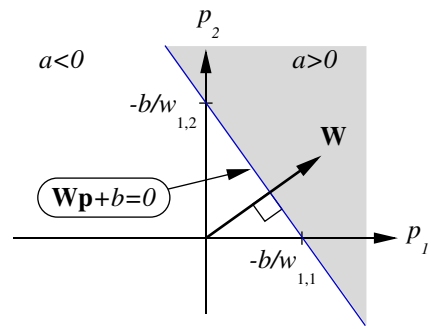
The weight matrix **W** in this case has only one row. The network output is

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

or

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

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]).



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.

We can create a network similar to the one shown using this command:

```
net = newlin([-1 1; -1 1],1);
```

The first matrix of arguments specifies typical two-element input vectors, and the last argument 1 indicates that the network has a single output.

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 ADAPLINE for a particular input vector.

```
p = [5; 6];
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE network with `newlin`, adjust its elements as you want, and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the 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].

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 below, is discussed in detail in Chapter 4, “Linear Filters.”

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^{T}(k)$$

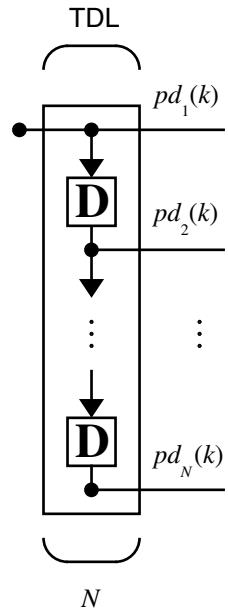
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

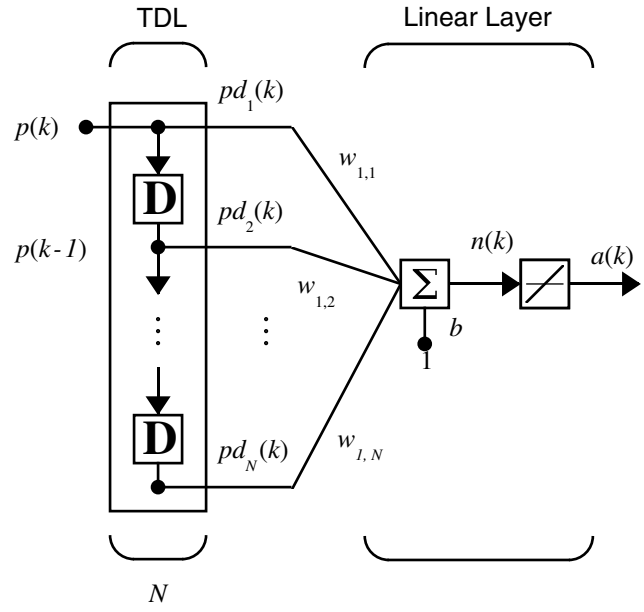
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown in the next figure. The input signal enters from the left and passes through *N*-1 delays. The output of the tapped delay line (TDL) is an *N*-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Adaptive Filter

You can combine a tapped delay line with an ADALINE network to create the *adaptive filter* shown in the next figure.



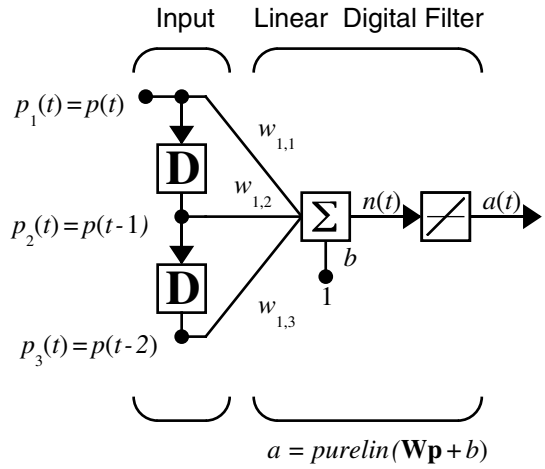
The output of the filter is given by

$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i}a(k-i+1) + b$$

In digital signal processing, this network is referred to as a f*inite impulse response (FIR)* filter [WiSt85]. 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 `newlin`.



Assume that the input values have a range from 0 to 10. You can now define the single output network.

```
net = newlin([0,10],1);
```

Specify the delays in the tapped delay line with

```
net.inputWeights{1,1}.delays = [0 1 2];
```

This definition indicates that the delay line connects to the network weight matrix through delays of 0, 1, and 2 time units. (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. Specify 10 passes through the input sequence with

```
net.adaptParam.passes = 10;
```

Then launch the training with

```
[net,y,E,pf,af] = adapt(net,p,t,pi);
```

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
```

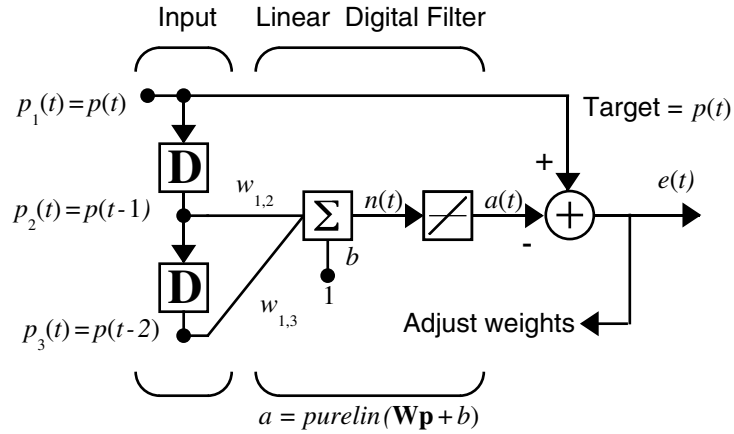
```
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

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 cancellation.

Prediction Example

Suppose that you want to use an adaptive filter to predict the next value of a stationary random process, *p(t)*. You can use the network shown in the following figure to do this prediction.



Predictive Filter: $a(t)$ is approximation to $p(t)$

The signal to be predicted, *p(t),* enters from the left into a tapped delay line. The previous two values of *p(t)* are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error *e(t)* on the far right. If this error is 0, the network output *a(t)* is exactly equal to *p(t)*, and the network has done its prediction properly.

Given the autocorrelation function of the stationary random process *p(t)*, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be

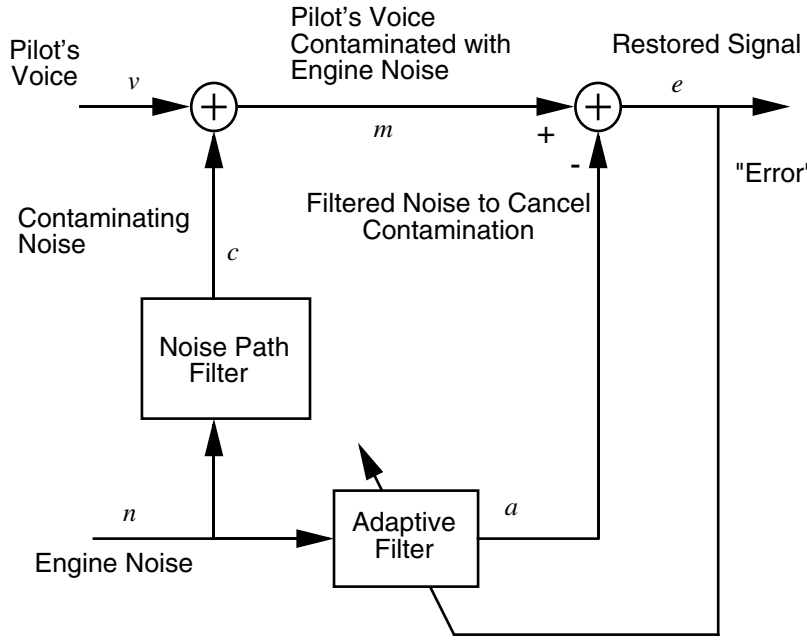
performed. This lack does not matter to the network. After it is initialized and operating, the network adapts at each time step to minimize the error and in a relatively short time is able to predict the input *p(t)*.

Chapter 10 of [HDB96] 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 demonstration `nnd10nc` to see an adaptive noise cancellation program example in action. This demonstration allows you to pick a learning rate and *momentum* (see Chapter 5, "Backpropagation"), and shows the learning trajectory, and the original and cancellation signals versus time.

Noise Cancellation 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.

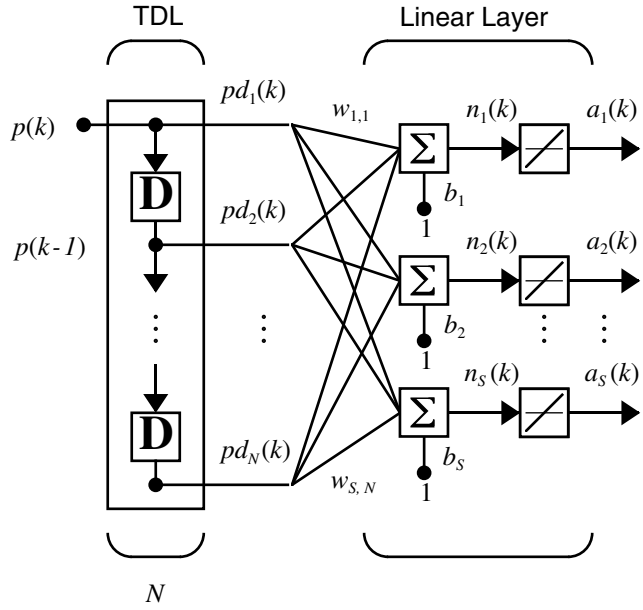
The network does its best to output m adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal m. The network error e is equal to m, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, e contains only the pilot's 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.

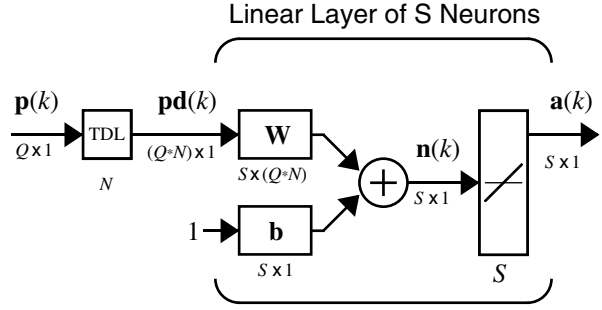
Try `demolin8` for an example of adaptive noise cancellation.

Multiple Neuron Adaptive Filters

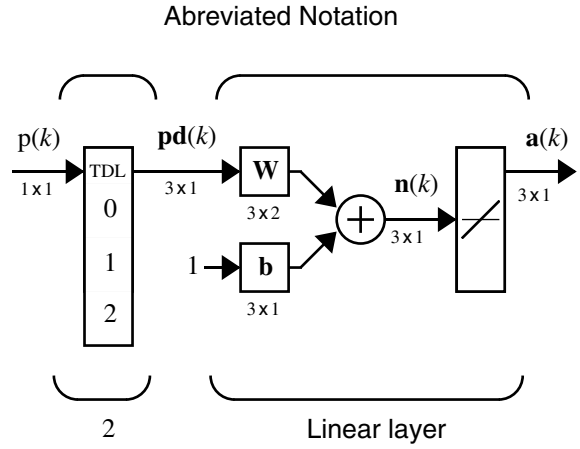
You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with S linear neurons, as shown in the next figure.



Alternatively, you can represent this same network in abbreviated form.



If you want to show more of the detail of the tapped delay line—and there are not too many delays—you can use the following notation:



Here, a tapped delay line sends to the weight matrix:

- The current signal
- The previous signal
- The signal delayed before that

You could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays must appears in increasing order as they go from top to bottom.

11

Applications

Introduction (p. 11-2)
Applin1: Linear Design (p. 11-3)
Applin2: Adaptive Prediction (p. 11-7)
Appelm1: Amplitude Detection (p. 11-11)
Appcr1: Character Recognition (p. 11-15)

Introduction

Today, neural networks can solve problems of economic importance that could not be approached previously in any practical way. Some of the recent neural network applications are discussed in this chapter. See Chapter 1, "Getting Started," for a list of many areas where neural networks already have been applied.

Note The rest of this chapter describes applications that are practical and make extensive use of the neural network functions described throughout this documentation.

Application Scripts

The following application scripts are available:

- `applin1` and `applin2` contain linear network applications.
- `appelm1` contains the Elman network amplitude detection application.
- `appcr1` contains the character recognition application.

Type `help nndemos` to see a listing of all neural network demonstrations or applications.

Applin1: Linear Design

Problem Definition

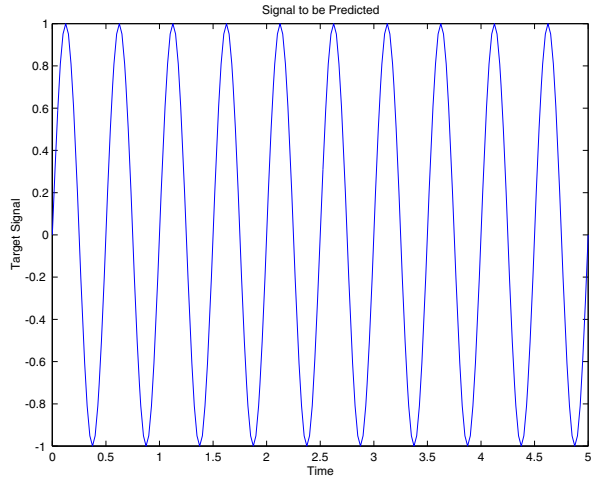
Here is the definition of a signal `T`, which lasts 5 seconds and is defined at a sampling rate of 40 samples per second.

```
time = 0:0.025:5;
T = sin(time*4*pi);
Q = length(T);
```

At any given time step, the network is given the last five values of the signal `t`, and expected to give the next value. The inputs `P` are found by delaying the signal `T` from one to five time steps.

```
P = zeros(5,Q);
P(1,2:Q) = T(1,1:(Q-1));
P(2,3:Q) = T(1,1:(Q-2));
P(3,4:Q) = T(1,1:(Q-3));
P(4,5:Q) = T(1,1:(Q-4));
P(5,6:Q) = T(1,1:(Q-5));
```

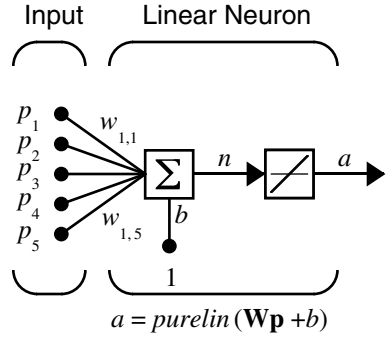
Here is a plot of the signal `T`.



Network Design

Because the relationship between past and future values of the signal is not changing, the network can be designed directly from examples, using `newlind`.

The problem as defined above has five inputs (the five delayed signal values), and one output (the next signal value). Thus, the network solution must consist of a single neuron with five inputs.



Here `newlind` finds the weights and biases for the neuron above that minimize the sum squared error for this problem.

```
net = newlind(P,T);
```

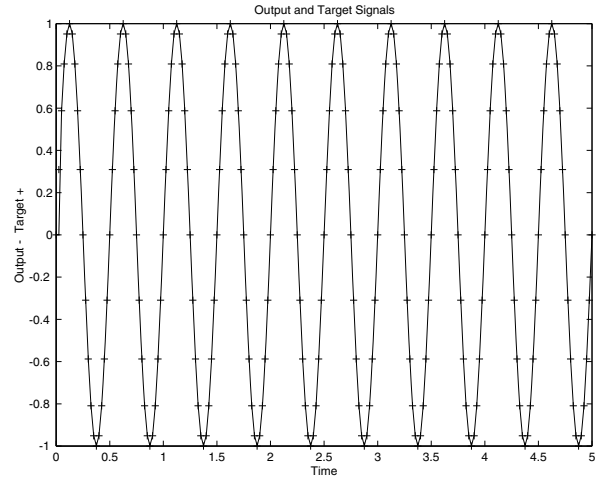
You can now test the resulting network.

Network Testing

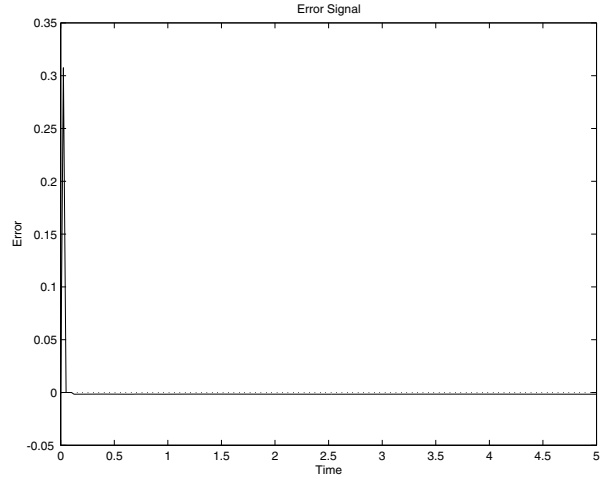
To test the network, its output `a` is computed for the five delayed signals `P` and compared with the actual signal `T`.

```
a = sim(net,P);
```

Here is a plot of `a` compared to `T`.



The network's output `a` and the actual signal `t` appear to match perfectly. Just to be sure, plot the error `e = T – a`.



The network did have some error for the first few time steps. This occurred because the network did not actually have five delayed signal values available until the fifth time step. However, after the fifth time step error was negligible. The linear network did a good job. Run the script `applin1` to see these plots.

Thoughts and Conclusions

While `newlind` is not able to return a zero error solution for nonlinear problems, it does minimize the sum squared error. In many cases, the solution, while not perfect, can model a nonlinear relationship well enough to meet the application specifications. Giving the linear network many delayed signal values gives it more information with which to find the lowest error linear fit for a nonlinear problem.

Of course, if the problem is very nonlinear and/or the desired error is very low, backpropagation or radial basis networks would be more appropriate.

Applin2: Adaptive Prediction

In application script `applin2`, a linear network is trained incrementally with `adapt` to predict a time series. Because the linear network is trained incrementally, it can respond to changes in the relationship between past and future values of the signal.

Problem Definition

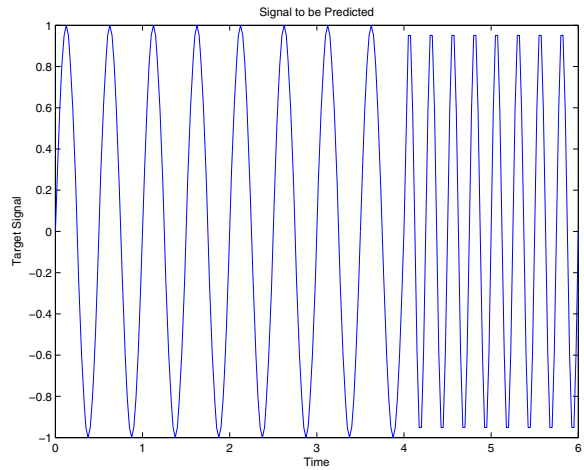
The signal `T` to be predicted lasts 6 seconds with a sampling rate of 20 samples per second. However, after 4 seconds the signal's frequency suddenly doubles.

```
time1 = 0:0.025:4;
time2 = 4.05:0.025:6;
time = [time1 time2];
T = [sin(time1*4*pi) sin(time2*8*pi)];
```

Because you are training the network incrementally, change `t` to a sequence.

```
T = con2seq(T);
```

Here is a plot of this signal.

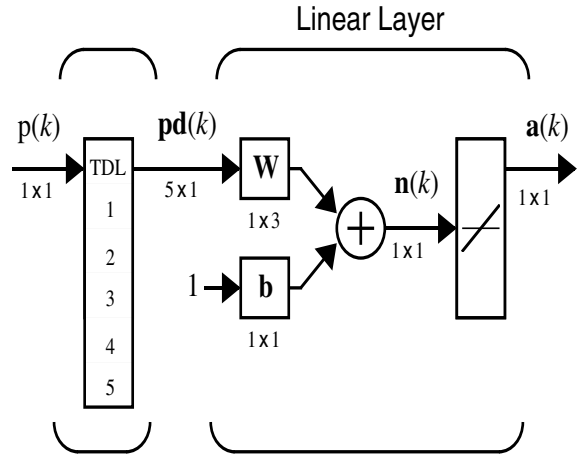


The input to the network is the same signal that makes up the target.

```
P = T;
```

Network Initialization

The network has only one neuron, as only one output value of the signal `T` is being generated at each time step. This neuron has five inputs, the five delayed values of the signal `T`.



The function `newlin` creates the network shown above. Use a learning rate of 0.1 for incremental training.

```
lr = 0.1;
delays = [1 2 3 4 5];
net = newlin(P,T,delays,lr);
```

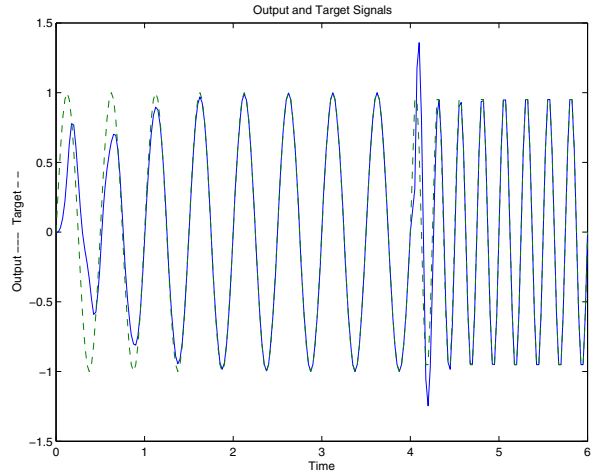
Network Training

The above neuron is trained incrementally with `adapt`. Here is the code to train the network on input/target signals `P` and `T`.

```
[net,a,e]=adapt(net,P,T);
```

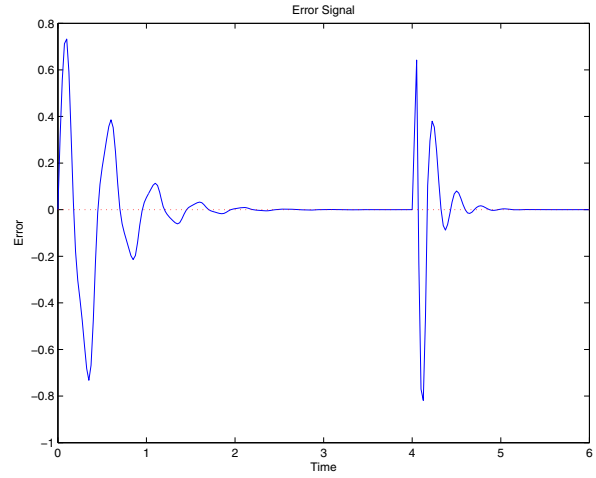
Network Testing

Once the network is adapted, you can plot its output signal and compare it to the target signal.



Initially, it takes the network 1.5 seconds (30 samples) to track the target signal. Then, the predictions are accurate until the fourth second when the target signal suddenly changes frequency. However, the adaptive network learns to track the new signal in an even shorter interval, because it has already learned a behavior (a sine wave) similar to the new signal.

A plot of the error signal makes these effects easier to see.



Thoughts and Conclusions

The linear network was able to adapt very quickly to the change in the target signal. The 30 samples required to learn the wave form are very impressive when one considers that in a typical signal processing application, a signal might be sampled at 20 kHz. At such a sampling frequency, 30 samples go by in 1.5 milliseconds.

The adaptive network can be monitored so as to give a warning that its constants are nearing values that would result in instability.

Another use for an adaptive linear model is suggested by its ability to find a minimum sum squared error linear estimate of a nonlinear system's behavior. An adaptive linear model is highly accurate as long as the nonlinear system stays near a given operating point. If the nonlinear system moves to a different operating point, the adaptive linear network changes to model it at the new point.

The sampling rate should be high to obtain the linear model of the nonlinear system at its current operating point in the shortest amount of time. However, there is a minimum amount of time that must occur for the network to see enough of the system's behavior to properly model it. To minimize this time, a small amount of noise can be added to the input signals of the nonlinear system. This allows the network to adapt faster as more of the operating point's dynamics are expressed in a shorter amount of time. Of course, this noise should be small enough so it does not affect the system's usefulness.

Appelm1: Amplitude Detection

Elman networks can be trained to recognize and produce both spatial and temporal patterns. An example of a problem where temporal patterns are recognized and classified with a spatial pattern is amplitude detection.

Amplitude detection requires that a wave form be presented to a network through time, and that the network output the amplitude of the wave form. This is not a difficult problem, but it demonstrates the Elman network design process.

The following material describes code that is contained in the demonstration `appelm1`.

Problem Definition

The following code defines two sine wave forms, one with an amplitude of 1.0, the other with an amplitude of 2.0.

```
p1 = sin(1:20);
p2 = sin(1:20)*2;
```

The target outputs for these wave forms are their amplitudes.

```
t1 = ones(1,20);
t2 = ones(1,20)*2;
```

These wave forms can be combined into a sequence where each wave form occurs twice. These longer wave forms are used to train the Elman network.

```
p = [p1 p2 p1 p2];
t = [t1 t2 t1 t2];
```

You want the inputs and targets to be considered a sequence, so you need to make the conversion from the matrix format.

```
Pseq = con2seq(p);
Tseq = con2seq(t);
```

Network Initialization

This problem requires the Elman network to detect a single value (the signal), and to output a single value (the amplitude) at each time step. This network has one input element and one output neuron.

The recurrent layer can have any number of neurons. However, as the complexity of the problem grows, more neurons are needed in the recurrent layer for the network to do a good job.

You can use the function `newelm` to create the initial network. Because this problem is fairly simple, only 10 recurrent neurons exist in the single hidden layer. A variable learning rate (`traingdx`) is used for this example.

```
net = newelm(Pseq,Tseq,10);
```

Network Training

Now call `train`.

```
net = train(net,Pseq,Tseq);
```

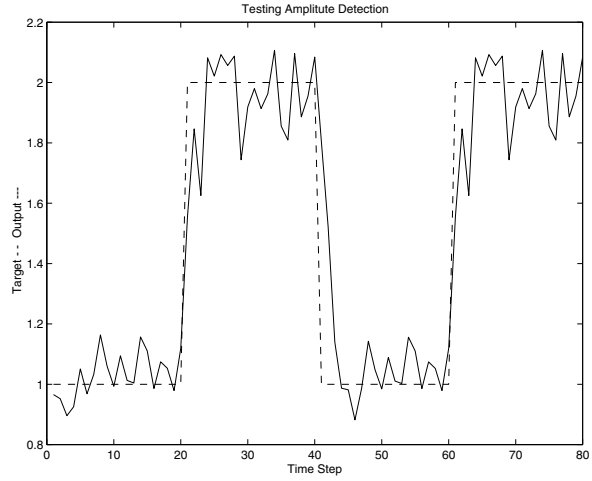
A typical final mean squared error is about 1.8e–2. Test the network to see what this means.

Network Testing

To test the network, the original inputs are presented, and its outputs are calculated with `simuelm`.

```
a = sim(net,Pseq);
```

Here is the plot.



The network does a good job. New wave amplitudes are detected with a few samples. More neurons in the recurrent layer and longer training times would result in even better performance.

The network has successfully learned to detect the amplitudes of incoming sine waves.

Network Generalization

Of course, even if the network detects the amplitudes of the training wave forms, it might not detect the amplitude of a sine wave with an amplitude it has not seen before.

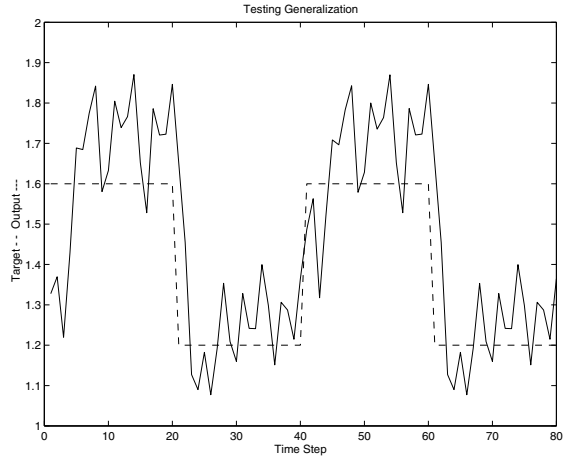
The following code defines a new wave form made up of two repetitions of a sine wave with amplitude 1.6 and another with amplitude 1.2.

```
p3 = sin(1:20)*1.6;
t3 = ones(1,20)*1.6;
p4 = sin(1:20)*1.2;
t4 = ones(1,20)*1.2;
pg = [p3 p4 p3 p4];
tg = [t3 t4 t3 t4];
pgseq = con2seq(pg);
```

The input sequence `pg` and target sequence `tg` are used to test the ability of the network to generalize to new amplitudes.

Once again the function `sim` is used to simulate the Elman network, and the results are plotted.

```
a = sim(net,pgseq);
```



This time the network did not do as well. It seems to have a vague idea as to what it should do, but is not very accurate.

You could improve generalization by training the network on more amplitudes than just 1.0 and 2.0. The use of three or four different wave forms with different amplitudes can result in a much better amplitude detector.

Improving Performance

Run `appelm1` to see plots similar to those above. Then make a copy of this file and try improving the network by adding more neurons to the recurrent layer, using longer training times, and giving the network more examples in its training data.

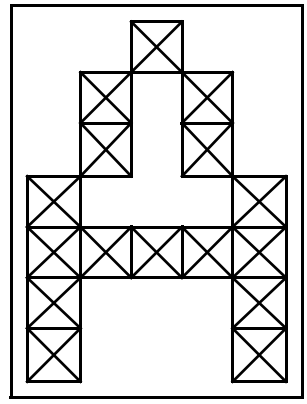
Appcr1: Character Recognition

It is often useful to have a machine perform pattern recognition. In particular, machines that can read symbols are very cost effective. A machine that reads banking checks can process many more checks than a human being in the same time. This kind of application saves time and money, and eliminates the requirement that a human perform such a repetitive task. The demonstration `appcr1` shows how character recognition can be done with a backpropagation network.

Problem Statement

A network is to be designed and trained to recognize the 26 letters of the alphabet. An imaging system that digitizes each letter centered in the system's field of vision is available. The result is that each letter is represented as a 5 by 7 grid of Boolean values.

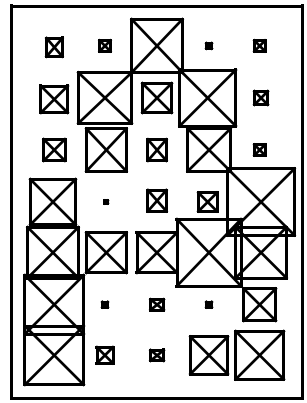
For example, here is the letter A.



Load the alphabet letter definitions and their target representations.

```
[alphabet,targets] = prprob;
```

However, the imaging system is not perfect, and the letters can suffer from noise.



Perfect classification of ideal input vectors is required, and reasonably accurate classification of noisy vectors.

The twenty-six 35-element input vectors are defined in the function `prprob` as a matrix of input vectors called `alphabet`. The target vectors are also defined in this file with a variable called `targets`. Each target vector is a 26-element vector with a 1 in the position of the letter it represents, and 0's everywhere else. For example, the letter A is to be represented by a 1 in the first element (as A is the first letter of the alphabet), and 0's in elements two through twenty-six.

Neural Network

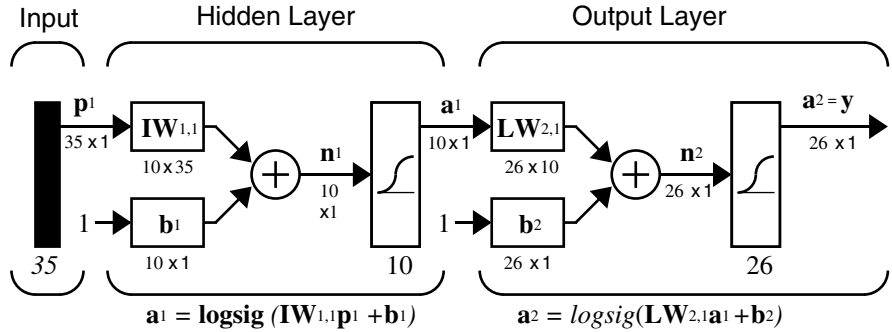
The network receives the 35 Boolean values as a 35-element input vector. It is then required to identify the letter by responding with a 26-element output vector. The 26 elements of the output vector each represent a letter. To operate correctly, the network should respond with a 1 in the position of the letter being presented to the network. All other values in the output vector should be 0.

In addition, the network should be able to handle noise. In practice, the network does not receive a perfect Boolean vector as input. Specifically, the network should make as few mistakes as possible when classifying vectors with noise of mean 0 and standard deviation of 0.2 or less.

Architecture

The neural network needs 35 inputs and 26 neurons in its output layer to identify the letters. The network is a two-layer `log-sigmoid/log-sigmoid`

network. The log-sigmoid transfer function was picked because its output range (0 to 1) is perfect for learning to output Boolean values.



The hidden (first) layer has 25 neurons. This number was picked by guesswork and experience. If the network has trouble learning, then neurons can be added to this layer. If the network solves the problem well, but a smaller more efficient network is desired, fewer neurons could be tried.

The network is trained to output a 1 in the correct position of the output vector and to fill the rest of the output vector with 0's. However, noisy input vectors can result in the network's not creating perfect 1's and 0's. After the network is trained the output is passed through the competitive transfer function `compet`. This makes sure that the output corresponding to the letter most like the noisy input vector takes on a value of 1, and all others have a value of 0. The result of this postprocessing is the output that is actually used.

Initialization

Create the two-layer network with `newff`.

```
net = newff(alphabet,targets,25);
```

Training

To create a network that can handle noisy input vectors, it is best to train the network on both ideal and noisy vectors. To do this, the network is first trained on ideal vectors until it has a low sum squared error.

Then the network is trained on 10 sets of ideal and noisy vectors. The network is trained on two copies of the noise-free alphabet at the same time as it is trained on noisy vectors. The two copies of the noise-free alphabet are used to maintain the network's ability to classify ideal input vectors.

Unfortunately, after the training described above the network might have learned to classify some difficult noisy vectors at the expense of properly classifying a noise-free vector. Therefore, the network is again trained on just ideal vectors. This ensures that the network responds perfectly when presented with an ideal letter.

All training is done using backpropagation with both adaptive learning rate and momentum, with the function `trainbpx`.

Training Without Noise

The network is initially trained without noise. Normally the data would be divided up into training, validation and test sets because there is only one sample of each letter and we need to train on all of them. This is acheived by setting the data division function to the empty string. (Validation data is used to stop training at the point where it has generalized as well as it can and further training will only optimize the network's performance on the training set at the expense of generalization as measured by its performance on the validation set.)

```
net1 = net;
net1.divideFcn = '';
[net,tr] = train(net,alphabet,targets);
```

Training with Noise

To obtain a network not sensitive to noise, train the network with one ideal copies and ten noisy copies of the vectors in `alphabet`. The target vectors consist of eleven copies of the vectors in `target`. The noisy vectors have noise of mean 0.2 added to them. This forces the neuron to learn how to properly identify noisy letters, while requiring that it can still respond well to ideal vectors.

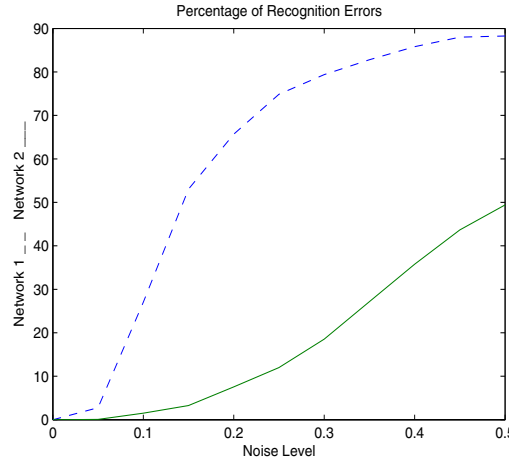
```
numNoisy = 10;
alphabet2 = [alphabet
repmat(alphabet,1,numNoisy)+randn(35,25*numNoisy)*0.2];
targets2 = [targets repmat(targets1,numNoise)];
net2 = train(net,alphabet2,targets2);
```

System Performance

The reliability of the neural network pattern recognition system is measured by testing the network with hundreds of input vectors with varying quantities

of noise. The script file `appcr1` tests the network at various noise levels, and then graphs the percentage of network errors versus noise. Noise with a mean of 0 and a standard deviation from 0 to 0.5 is added to input vectors. At each noise level, 100 presentations of different noisy versions of each letter are made and the network's output is calculated. The output is then passed through the competitive transfer function so that only one of the 26 outputs (representing the letters of the alphabet), has a value of 1.

The number of erroneous classifications is then added and percentages are obtained.



The dashed line on the graph shows the reliability for the network trained without noise. The reliability of the same network when it was trained with noise is shown with a solid line. Thus, training the network on noisy inputs greatly reduces its errors when it has to classify noisy vectors.

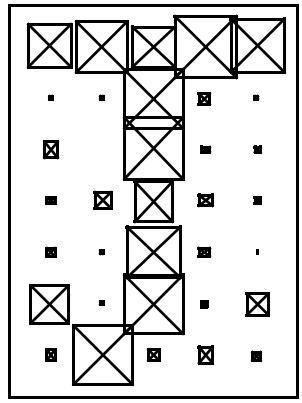
Neither network made any errors for vectors with noise of mean 0.00. As the noise mean increased both networks began making errors.

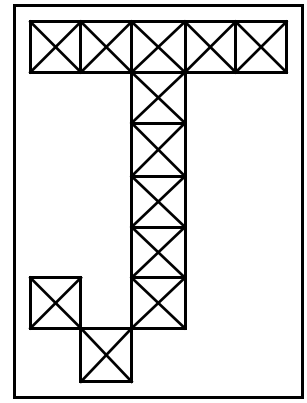
If a higher accuracy is needed, the network can be trained with more neurons in its hidden layer or more samples. Also, the resolution of the input vectors can be increased to a 10-by-14 grid. Finally, the network could be trained on input vectors with greater amounts of noise if greater reliability were needed for higher levels of noise.

To test the system, create a letter with noise and present it to the network.

```
noisyJ = alphabet(:,10)+randn(35,1) * 0.2;
plotchar(noisyJ);
y = sim(net,noisyJ);
y = compet(y);
answer = find(compet(y) == 1);
plotchar(alphabet(:,answer));
```

Here is the noisy letter and the letter the network picked (correctly).





12

Advanced Topics

Custom Networks (p. 12-2)
Additional Toolbox Functions (p. 12-15)
Custom Functions (p. 12-16)

Custom Networks

Neural Network Toolbox™ software provides a flexible network object type that allows many kinds of networks to be created and then used with functions such as `init`, `sim`, and `train`.

Type the following to see all the network creation functions in the toolbox.

```
help nnnetwork
```

This flexibility is possible because networks have an object-oriented representation. The representation allows you to define various architectures and assign various algorithms to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

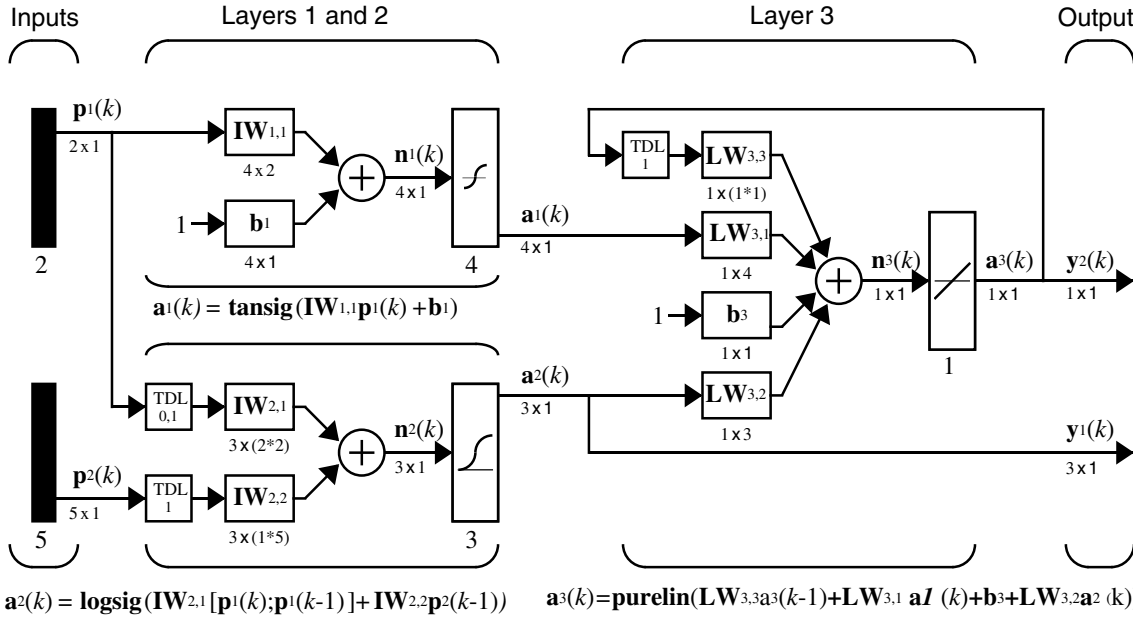
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network. See Chapter 14, “Network Object Reference,” for descriptions of all network properties.

The following sections demonstrate 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 are `numInputs` and `numLayers`. These properties allow you to select how many inputs and layers you want the network to have.

```
net =

    Neural Network object:

    architecture:

         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 object:

    architecture:

         numInputs: 2
         numLayers: 3
```

Examine the next four properties:

```
          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 1's and 0's 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 is the number of outputs:

```
numOutputs: 2 (read-only)
```

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

Subobject Properties

The next group of properties is

```
subobject structures:

            inputs: {2x1 cell} of inputs
            layers: {3x1 cell} of layers
           outputs: {1x3 cell} containing 1 output
            biases: {3x1 cell} containing 2 biases
      inputWeights: {3x2 cell} containing 3 input weights
      layerWeights: {3x3 cell} containing 3 layer 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 struct]
    [1x1 struct]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows:

```
ans =

    exampleInput: [0 1]
     processFcns: {}
   processParams: {}
 processSettings: {}
  processedRange: [0 1]
   processedSize: 1
           range: [0 1]
            size: 1
        userdata: [1x1 struct]
```

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 pages for each processing function to learn more about the function 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 =

     dimensions: 1
    distanceFcn: 'dist'
      distances: 0
         initFcn: 'initwb'
    netInputFcn: 'netsum'
  netInputParam: [1x1 struct]
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'purelin'
  transferParam: [1x1 struct]
       userdata: [1x1 struct]
```

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 only need to set its initialization function, as follows:

```
net.layers{3}.initFcn = 'initnw';
```

Outputs. Look at how the `outputs` property is arranged with this line of code.

```
net.outputs
ans =

     []    [1x1 struct]    [1x1 struct]
```

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 =

  exampleOutput: []
    processFcns: {}
  processParams: {}
processSettings: {}
 processedRange: []
  processedSize: 1
          range: []
           size: 3
       userdata: [1x1 struct]
```

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 like.

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 struct]
              []
    [1x1 struct]
```

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:

```
ans =
       initFcn: ''
         learn: 1
       learnFcn: ''
     learnParam: ''
           size: 4
       userdata: [1x1 struct]
```

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)
   divideFcn: (none)
 gradientFcn: (none)
     initFcn: (none)
  performFcn: (none)
   plotFcns: {}
    trainFcn: (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, look at the final group of network properties (aside from the `userdata` property).

```
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 1's 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 =
```

```
    epochs: 100
      goal: 0
  max_fail: 5
 mem_reduc: 1
  min_grad: 1.0000e-10
        mu: 1.0000e-03
    mu_dec: 0.1000
    mu_inc: 10
    mu_max: 1.0000e+10
      show: 25
      time:
```

You will not often need to modify these values. See the documentation for the training function for information about what each of these mean. They have been initialized with default values that work well for a large range of problems, so we will not 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`.

Additional Toolbox Functions

Most toolbox functions are explained in chapters dealing with networks that use them. However, some functions are not used by toolbox networks, but are included because they might be useful to you in creating custom networks.

For instance, `satlin` and `softmax` are two transfer functions not used by any standard network in the toolbox, but which you can use in your custom networks. See the

Custom 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.

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 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 `template_transfer.m`. Rename the new file `mytransfer`. Start editing the file by changing the function name at the top from `template_transfer` 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 directory, and assign the name of your transfer function to the `transferFcn` property of any layer of any network object to put it to use.

13

Historical Networks

Introduction (p. 13-2)
Elman Networks (p. 13-3)
Hopfield Network (p. 13-8)

Introduction

Recurrent networks are a topic of considerable interest. This chapter covers two recurrent networks: Elman and Hopfield networks.

Elman networks are two-layer backpropagation networks, with the addition of a feedback connection from the output of the hidden layer to its input. This feedback path allows Elman networks to learn to recognize and generate temporal patterns, as well as spatial patterns. The best paper on the Elman network is Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

The Hopfield network is used to store one or more stable target vectors. These stable vectors can be viewed as memories that the network recalls when provided with similar vectors that act as a cue to the network memory. You might want to peruse a basic paper in this field:

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, November 1989, pp. 1405–1422.

Important Recurrent Network Functions

You can create Elman networks with the function `newelm`.

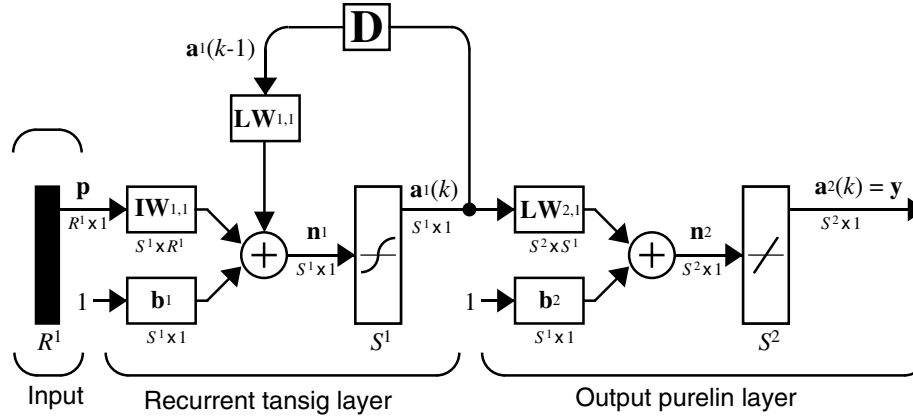
You can create Hopfield networks with the function `newhop`.

Type `help elman` or `help hopfield` to see a list of functions and demonstrations related to either of these networks.

Elman Networks

Architecture

The Elman network commonly is a two-layer network with feedback from the first-layer output to the first-layer input. This recurrent connection allows the Elman network to both detect and generate time-varying patterns. A two-layer Elman network is shown below.



$$\mathbf{a}^1(k) = \mathbf{tansig}(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{LW}^{1,1}\mathbf{a}^1(k\text{-}1) + \mathbf{b}^1) \qquad \mathbf{a}^2(k) = \mathbf{purelin}(\mathbf{LW}^{2,1}\mathbf{a}^1(k) + \mathbf{b}^2)$$

The Elman network has `tansig` neurons in its hidden (recurrent) layer, and `purelin` neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fitted increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step.

Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different because of different feedback states.

Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

Creating an Elman Network (newelm)

You can use the function `newelm` to create an Elman network with two or more layers. The hidden layers commonly have `tansig` transfer functions and this is the default for `newelm`. The architecture diagram shows that `purelin` is commonly the output-layer transfer function.

The default backpropagation training function is `trainbfg`. You might use `trainlm`, but it tends to proceed so rapidly that it does not necessarily do well in the Elman network. The backpropagation weight/bias learning function default is `learngdm`, and the default performance function is `mse`.

When the network is created, the weights and biases of each layer are initialized with the Nguyen-Widrow layer-initialization method, which is implemented in the function `initnw`.

For example, consider a sequence of single-element input vectors in the range from 0 to 1 and with outputs in the same range. Suppose that you want to have five hidden-layer `tansig` neurons and a single `logsig` output layer. The following command creates this network:

```
net = newelm([0 1],[0 1],5,{'tansig','logsig'});
```

Simulation

Suppose that you want to find the response of this network to an input sequence of eight digits that are either 0 or 1:

```
P = round(rand(1,8))
P =
     0     1     0     1     1     0     0     0
```

Recall that a sequence to be presented to a network is to be in cell array form. Convert `P` to this form:

```
Pseq = con2seq(P)
Pseq =
    [0]    [1]    [0]    [1]    [1]    [0]    [0]    [0]
```

Now you can find the output of the network with the function `sim`:

```
Y = sim(net,Pseq)
Y =
Columns 1 through 5
   [1.9875e-04]   [0.1146]   [5.0677e-05]   [0.0017]   [0.9544]
Columns 6 through 8
    [0.0014]    [5.7241e-05]    [3.6413e-05]
```

Convert this back to concurrent form with

```
z = seq2con(Y);
```

and display the output in concurrent form with

```
z{1,1}
ans =
  Columns 1 through 7
   0.0002   0.1146   0.0001   0.0017   0.9544   0.0014   0.0001
Column 8
    0.0000
```

Thus, once the network is created and the input specified, you need only call `sim`.

Training an Elman Network

Elman networks can be trained with either of two functions, `train` or `adapt`.

When you use the function `train` to train an Elman network the following occurs:

At each epoch,

1. The entire input sequence is presented to the network, and its outputs are calculated and compared with the target sequence to generate an error sequence.

2. For each time step, the error is backpropagated to find *gradients* of errors for each weight and bias. This *gradient* is actually an approximation, because the contributions of weights and biases to errors via the delayed recurrent connection are ignored.

3. This gradient is then used to update the weights with the chosen backprop training function. The function `traingdx` is recommended.

When you use the function `adapt` to train an Elman network, the following occurs:

At each time step,

1. Input vectors are presented to the network, and it generates an error.
2. The error is backpropagated to find gradients of errors for each weight and bias. This gradient is actually an approximation, because the contributions of weights and biases to the error, via the delayed recurrent connection, are ignored.
3. This approximate gradient is then used to update the weights with the chosen learning function. The function `learngdm` is recommended.

Elman networks are not as reliable as some other kinds of networks, because both training and adaption happen using an approximation of the error gradient.

For an Elman to have the best chance at learning a problem, it needs more hidden neurons in its hidden layer than are actually required for a solution by another method. While a solution might be available with fewer neurons, the Elman network is less able to find the most appropriate weights for hidden neurons because the error gradient is approximated. Therefore, having a fair number of neurons to begin with makes it more likely that the hidden neurons will start out dividing up the input space in useful ways.

The function `train` trains an Elman network to generate a sequence of target vectors when it is presented with a given sequence of input vectors. The input vectors and target vectors are passed to `train` as matrices `P` and `T`. `Train` takes these vectors and the initial weights and biases of the network, trains the network using backpropagation with momentum and an adaptive learning rate, and returns new weights and biases.

Continue with the example, and suppose that you want to train a network with an input `P` and targets `T` as defined below,

```
P = round(rand(1,8))
P =
     1     0     1     1     1     0     1     1
```

and

```
T = [0 (P(1:end-1)+P(2:end) == 2)]
T =
     0     0     0     1     1     0     0     1
```

Here `T` is defined to be 0, except when two 1's occur in `P`, in which case `T` is 1.

As noted previously, the network has five hidden neurons in the first layer.

```
net = newelm(P,T,5,{'tansig','logsig'});
```

Use `trainbfg` as the training function and train for 100 epochs. After training, simulate the network with the input `P` and calculate the difference between the target output and the simulated network output.

```
Pseq = con2seq(P);
Tseq = con2seq(T);
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq);
z = seq2con(Y);
z{1,1};
diff1 = T - z{1,1};
```

Note that the difference between the target and the simulated output of the trained network is very small. Thus, the network is trained to produce the desired output sequence on presentation of the input vector **P**.

See "Appelm1: Amplitude Detection" on page 11-11 for an application of the Elman network to the detection of wave amplitudes.

Hopfield Network

Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points

The design method presented is not perfect in that the designed network can have spurious undesired equilibrium points in addition to the desired ones. However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

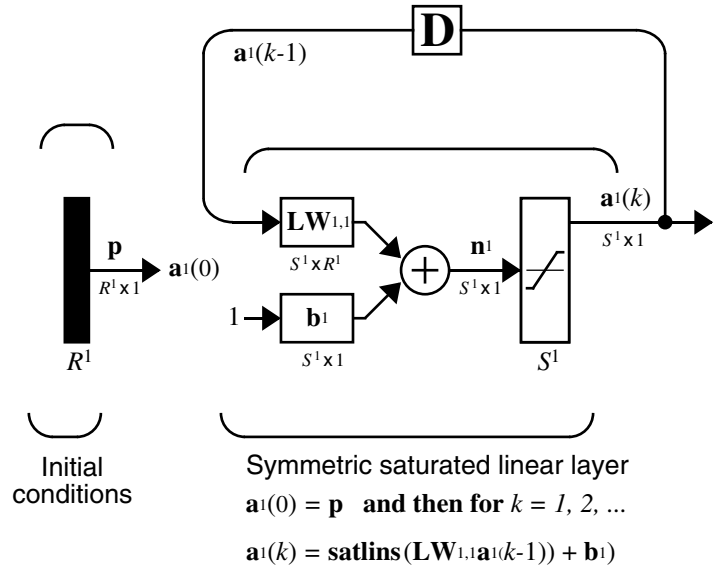
The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

The material in this section is based on the following paper: Jian-Hua Li, Anthony N. Michel, and Wolfgang Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Trans. on Circuits and Systems*, Vol. 36, No. 11, November 1989, pp. 1405–22.

For further information on Hopfield networks, read Chapter 18 of the *Hopfield Network* [HDB96].

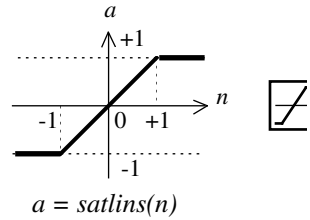
Architecture

The architecture of the Hopfield network follows.



As noted, the *input* **p** to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function `satlins`.



Satlins Transfer Function

For inputs less than -1 `satlins` produces -1. For inputs in the range -1 to +1 it simply returns the input value. For inputs greater than +1 it produces +1.

This network can be tested with one or more input vectors that are presented as initial conditions to the network. After the initial conditions are given, the network produces an output that is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each

output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

Design (newhop)

Li et al. [LiMi89] have studied a system that has the basic structure of the Hopfield network but is, in Li's own words, "easier to analyze, synthesize, and implement than the Hopfield model." The authors are enthusiastic about the reference article, as it has many excellent points and is one of the most readable in the field. However, the design is mathematically complex, and even a short justification of it would burden this guide. Thus the Li design method is presented, with thanks to Li et al., as a recipe that is found in the function `newhop`.

Given a set of target equilibrium points represented as a matrix **T** of vectors, `newhop` returns weights and biases for a recursive network. The network is guaranteed to have stable equilibrium points at the target vectors, but it could contain other spurious equilibrium points as well. The number of these undesired points is made as small as possible by the design method.

Once the network has been designed, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium points will find their targets. As suggested by the network figure, an array of input vectors is presented one at a time or in a batch. The network proceeds to give output vectors that are fed back as inputs. These output vectors can be can be compared to the target vectors to see how the solution is proceeding.

The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time. First you might check to see that the target equilibrium point vectors are indeed contained in the network. Then you could try other input vectors to determine the domains of attraction of the target equilibrium points and the locations of spurious equilibrium points if they are present.

Consider the following design example. Suppose that you want to design a network with two stable points in a three-dimensional space.

```
T = [-1 -1 1; 1 -1 1]'
T =
     -1      1
     -1     -1
      1      1
```

You can execute the design with

```
net = newhop(T);
```

Next, check to make sure that the designed network is at these two points, as follows. (Because Hopfield networks have no inputs, the second argument to `sim` below is Q = 2 when you are using matrix notation).

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

This gives you

```
Y =
     -1      1
     -1     -1
      1      1
```

Thus, the network has indeed been designed to be stable at its design points. Next you can try another input condition that is not a design point, such as

```
Ai = {[-0.9; -0.8; 0.7]};
```

This point is reasonably close to the first design point, so you might anticipate that the network would converge to that first point. To see if this happens, run the following code. Note, incidentally, that the original point was specified in cell array form. This allows you to run the network for more than one step.

```
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

This produces

```
ans =
     -1
     -1
      1
```

Thus, an original condition close to a design point did converge to that point.

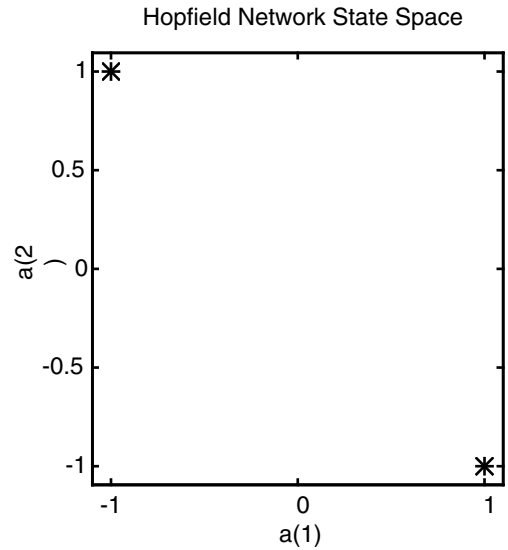
This is, of course, the hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include spurious undesired stable points that attract the solution.

Example

Consider a Hopfield network with just two neurons. Each neuron has a bias and weights to accommodate two-element input vectors weighted. The target equilibrium points are defined to be stored in the network as the two columns of the matrix **T**.

```
T = [1 -1; -1 1]'
T =
      1     -1
     -1      1
```

Here is a plot of the Hopfield state space with the two stable points labeled with '*' markers.



These target stable points are given to `newhop` to obtain weights and biases of a Hopfield network.

```
net = newhop(T);
```

The design returns a set of weights and a bias for each neuron. The results are obtained from

```
W= net.LW{1,1}
```

which gives

```
W =
    0.6925   -0.4694
   -0.4694    0.6925
```

and from

```
b = net.b{1,1}
```

which gives

```
b =
    0
    0
```

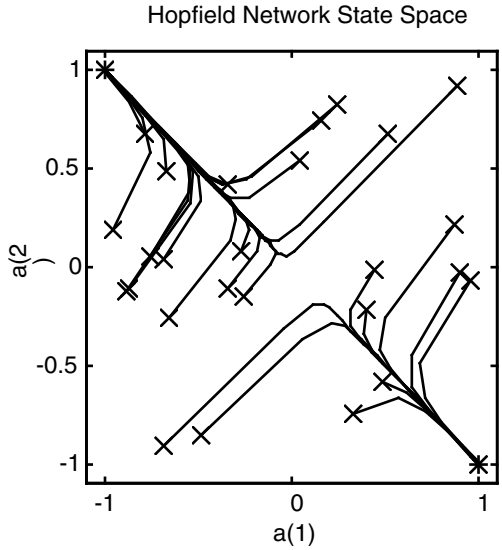
Next test the design with the target vectors **T** to see if they are stored in the network. The targets are used as inputs for the simulation function `sim`.

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

```
Y =
     1    -1
    -1     1
```

As hoped, the new network outputs are the target vectors. The solution stays at its initial conditions after a single update and, therefore, will stay there for any number of updates.

Now you might wonder how the network performs with various random input vectors. Here is a plot showing the paths that the network took through its state space to arrive at a target point.



This plot show the trajectories of the solution for various starting points. You can try demonstration `demohop1` to see more of this kind of network behavior.

Hopfield networks can be designed for an arbitrary number of dimensions. You can try `demohop3` to see a three-dimensional design.

Unfortunately, Hopfield networks can have both unstable equilibrium points and spurious stable points. You can try demonstrations `demohop2` and `demohop4` to investigate these issues.

14

Network Object Reference

Network Properties (p. 14-2)
Subobject Properties (p. 14-13)

Network Properties

These properties define the basic features of a network. “Subobject Properties” on page 14-13 describes properties that define network details.

Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

net.numInputs

This property defines the number of inputs a network receives. It can be set to 0 or a positive integer.

Clarification. The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e., the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

Side Effects. Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

net.numLayers

This property defines the number of layers a network has. It can be set to 0 or a positive integer.

Side Effects. Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers:

```
net.biasConnect
net.inputConnect
net.layerConnect
net.outputConnect
```

and changes the size of each cell array of subobject structures whose size depends on the number of layers:

```
net.biases
```

```
net.inputWeights
net.layerWeights
net.outputs
```

and also changes the size of each of the network's adjustable parameter's properties:

```
net.IW
net.LW
net.b
```

net.biasConnect

This property defines which layers have biases. It can be set to any N-by-1 matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the ith layer is indicated by a 1 (or 0) at

```
net.biasConnect(i)
```

Side Effects. Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

net.inputConnect

This property defines which layers have weights coming from inputs.

It can be set to any N_l x N_i matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the ith layer from the jth input is indicated by a 1 (or 0) at `net.inputConnect(i,j)`.

Side Effects. Any change to this property alters the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

net.layerConnect

This property defines which layers have weights coming from other layers. It can be set to any N_l x 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 ith layer from the jth 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 x 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 ith 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
```

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 "Subobject Properties" on page 14-13.

net.inputs

This property holds structures of properties for each of the network's inputs. It is always an N_i x 1 cell array of input structures, where N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the ith network input is located at

```
net.inputs{i}
```

Input Properties. See "Inputs" on page 14-13 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 x 1 cell array of layer structures, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the ith layer is located at `net.layers{i}`.

Layer Properties. See "Layers" on page 14-15 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 x 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 ith layer (or a null matrix `[]`) is located at `net.outputs{i}` if net.outputConnect(i) is 1 (or 0).

Output Properties. See "Outputs" on page 14-20 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 x 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 ith 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 14-22 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 x 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 ith layer from the jth input (or a null matrix `[]`) is located at `net.inputWeights{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

Input Weight Properties. See "Input Weights" on page 14-23 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 x 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 14-25 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.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 nnformat`.

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.gradientFcn

This property defines the function used to calculate the relationship between the network's weights and biases and performance either as a gradient or Jacobian. The gradient function is used by many training functions.

Side Effects. Whenever this property is altered, the network's gradient parameters (`net.gradientParam`) are set to contain the parameters and default values of the new function.

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)
```

For a list of functions, type

```
help nninit
```

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.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.plotFcn

This property defines the plot functions associated with a network. The neural network training window, which is launched by the `train` function, shows a button for each plotting function. Click the button during or after training to open the desired plot.

net.trainFcn

This property defines the function used to train the network. The training function 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.

Parameters

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.divideFcn

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.gradientParam

This property defines the parameters and values of the current gradient function. Call `help` on the current initialization function to get a description of what each field means:

```
help(net.gradientFcn)
```

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.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.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 x 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 x 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 x 1 cell array, where N_l is the number of network layers (`net.numLayers`).

The bias vector for the ith 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
```

Other

The only `other` property is a user data property.

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 Neural Network Toolbox™ software users:

```
net.userdata.note
```

Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

Inputs

These properties define the details of each *i*th network input.

net.inputs{i}.exampleInput

This property defines an example of the kinds of values for the *i*th network input.

Set the value to any R_i x Q matrix, where R_i is the number of elements in the input (`net.inputs{i}.size`).

Side Effects. Whenever this property is altered, the input `range`, `size`, `processedRange`, and `processedSize` are updated to match `net.inputs{i}.exampleInput` dimensions and the range of values in each of the rows.

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`.

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.

For a list of functions, type

```
help nnprocess
```

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 the `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 the `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 the `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 x 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}.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 is used to calculate `distances` between neurons in the *i*th layer from the neuron `positions`. Neuron distances are used by self-organizing maps. It can be set to the name of any distance function.

For a list of functions, type

```
help nndistance
```

Side Effects. Whenever this property is altered, the distances between the layer's neurons (`net.layers{i}.distances`) are updated.

net.layers{i}.distances (read-only)

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps:

```
net.layers{i}.distances
```

It is always set to the result of applying the layer's distance function (`net.layers{i}.distanceFcn`) to the positions of the layer's neurons (`net.layers{i}.positions`).

net.layers{i}.initFcn

This property defines which of the 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.

For a list of functions, type

```
help nninit
```

net.layers{i}.netInputFcn

This property defines which of the 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)

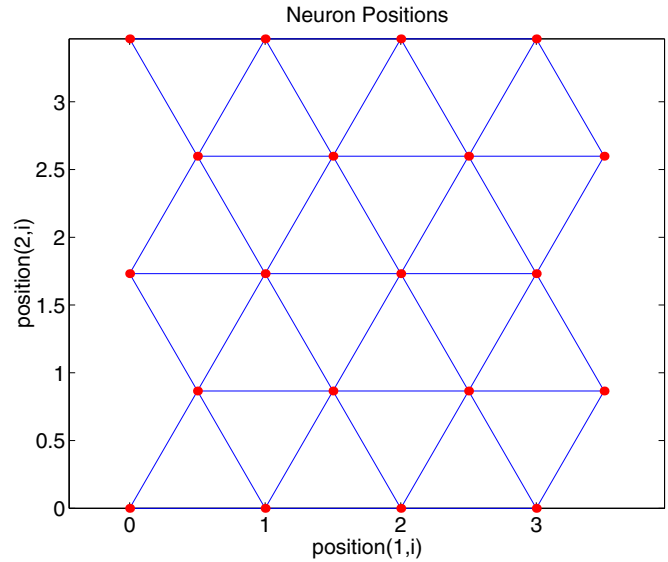
This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

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}.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.inputWeights{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 are used to calculate the *i*th layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

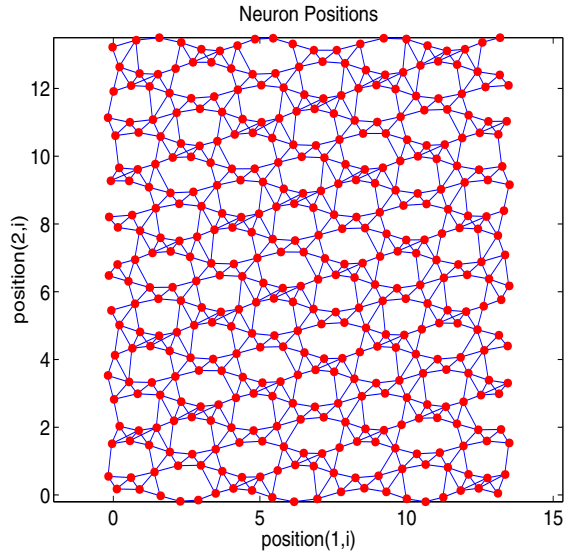
For a list of functions, type

```
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)
```


Neuron Positions
position(2,i)
position(1,i)
0
2
4
6
8
10
12
0
5
10
15

net.layers{i}.transferFcn

This function defines which of the is used to calculate the ith 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 ith network layer.

Outputs

net.outputs{i}.exampleOutput

This property defines an example of the kinds of target values to be expected by ith network output.

It can be set to any S_i x Q matrix, where S_i is the desired size of the output.

Side Effects. Whenever this property is altered, the output `range`, `size`, `processedRange`, and `processedSize` are updated to match its dimensions and the range of values in each of its rows. The size of the ith layer size is also updated to match the `processedSize`.

net.outputs{i}.processFcns

This property defines a row cell array of processing function names to be used by the ith 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 the `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 the `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 the `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 the `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 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`.

For a list of functions, type

```
help nninit
```

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 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 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.

For a list of functions, type

```
help nninit
```

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 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 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 is used to initialize the weight matrix (`net.LW{i,j}`) going to the *i*th layer from the *j*th layer, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

For a list of functions, type

```
help nninit
```

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 is used to update the weight matrix (`net.LW{i,j}`) going to the *i*th layer from the *j*th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type

```
help nnlearn
```

net.layerWeights{i,j}.learnParam

This property defines the learning parameters fields and values for the current learning function of the *i*th layer's weight coming from the *j*th layer. The fields

of this property depend on the current learning function. Call `help` on the current net input function to get a description of each field.

net.layerWeights{i,j}.size (read-only)

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th layer. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.LW{i,j}`). The first element is equal to the size of the *i*th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the *j*th layer.

net.layerWeights{i,j}.userdata

This property provides a place for users to add custom information to the (*i*,*j*)th layer weight.

net.layerWeights{i,j}.weightFcn

This property defines which of the 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.

15

Function Reference

"Analysis Functions" on page 15-29	Analyze network properties
"Distance Functions" on page 15-30	Compute distance between two vectors
"Graphical Interface Functions" on page 15-31	Open GUIs for building neural networks
"Layer Initialization Functions" on page 15-32	Initialize layer weights
"Learning Functions" on page 15-33	Learning algorithms used to adapt networks
"Line Search Functions" on page 15-34	Line-search algorithms
"Net Input Functions" on page 15-35	Sum excitations of layer
"Network Initialization Function" on page 15-36	Initialize network weights
"New Networks Functions" on page 15-38	Create network architectures
"Network Use Functions" on page 15-37	High-level functions to manipulate networks
"Performance Functions" on page 15-39	Measure network performance
"Plotting Functions" on page 15-40	Plot and analyze networks and network performance
"Processing Functions" on page 15-41	Preprocess and postprocess data
"Simulink® Support Function" on page 15-42	Generate Simulink® block for network simulation
"Topology Functions" on page 15-43	Arrange neurons of layer according to specific topology
"Training Functions" on page 15-44	Train networks

"Transfer Functions" on page 15-45	Transform output of network layer
"Utility Functions" on page 15-46	Internal utility functions
"Vector Functions" on page 15-47	Internal functions for network computations
"Weight and Bias Initialization Functions" on page 15-48	Initialize weights and biases
"Weight Functions" on page 15-49	Convolution, dot product, scalar product, and distances weight functions

Analysis Functions

`errsurf`	Error surface of single-input neuron
`confusion`	Classification confusion matrix
`maxlinlr`	Maximum learning rate for linear neuron
`roc`	Receiver operating characteristic

Distance Functions

`boxdist`	Distance between two position vectors
`dist`	Euclidean distance weight function
`linkdist`	Link distance function
`mandist`	Manhattan distance weight function

Graphical Interface Functions

`nctool`	Neural network classification tool
`nftool`	Open Neural Network Fitting Tool
`nntool`	Open Network/Data Manager
`nntraintool`	Neural network training tool
`nprtool`	Neural network pattern recognition tool
`view`	View a neural network

Layer Initialization Functions

`initnw`	Nguyen-Widrow layer initialization function
`initwb`	By-weight-and-bias layer initialization function

Learning Functions

`learncon`	Conscience bias learning function
`learngd`	Gradient descent weight/bias learning function
`learngdm`	Gradient descent with momentum weight/bias learning function
`learnh`	Hebb weight learning function
`learnhd`	Hebb with decay weight learning rule
`learnis`	Instar weight learning function
`learnk`	Kohonen weight learning function
`learnlv1`	LVQ1 weight learning function
`learnlv2`	LVQ2 weight learning function
`learnos`	Outstar weight learning function
`learnp`	Perceptron weight and bias learning function
`learnpn`	Normalized perceptron weight and bias learning function
`learnsom`	Self-organizing map weight learning function
`learnsomb`	Batch self-organizing map weight learning function
`learnwh`	Widrow-Hoff weight and bias learning rule

Line Search Functions

`srchbac`	1-D minimization using backtracking search
`srchbre`	1-D interval location using Brent's method
`srchcha`	1-D minimization using Charalambous' method
`srchgol`	1-D minimization using golden section search
`srchhyb`	1-D minimization using hybrid bisection/cubic search

Net Input Functions

`netprod`	Product net input function
`netsum`	Sum net input function

Network Initialization Function

`initlay`	Layer-by-layer network initialization function

Network Use Functions

`adapt`	Allow neural network to change weights and biases on inputs
`disp`	Neural network's properties
`display`	Name and properties of neural network's variables
`init`	Initialize neural network
`sim`	Simulate neural network
`train`	Train neural network

New Networks Functions

`network`	Create custom neural network
`newc`	Create competitive layer
`newcf`	Create cascade-forward backpropagation network
`newdtdnn`	Create distributed time delay neural network
`newelm`	Create Elman backpropagation network
`newff`	Create feedforward backpropagation network
`newfftd`	Create feedforward input-delay backpropagation network
`newfit`	Create a fitting network
`newgrnn`	Design generalized regression neural network
`newhop`	Create Hopfield recurrent network
`newlin`	Create linear layer
`newlind`	Design linear layer
`newlrn`	Create layered-recurrent network
`newlvq`	Create learning vector quantization network
`newnarx`	Create feedforward backpropagation network with feedback from output to input
`newnarxsp`	Create NARX network in series-parallel arrangement
`newp`	Create perceptron
`newpnn`	Design probabilistic neural network
`newpr`	Create a pattern recognition network
`newrb`	Design radial basis network
`newrbe`	Design exact radial basis network
`newsom`	Create self-organizing map
`sp2narx`	Convert series-parallel NARX network to parallel (feedback) form

Performance Functions

`mae`	Mean absolute error performance function
`mse`	Mean squared error performance function
`msne`	Mean squared normalized error performance function
`msnereg`	Mean squared normalized error with regularization performance functions
`msereg`	Mean squared error with regularization performance function
`mseregec`	Mean squared error with regularization and economization performance function
`sse`	Sum squared error performance function

Plotting Functions

`hintonw`	Hinton graph of weight matrix
`hintonwb`	Hinton graph of weight matrix and bias vector
`plotbr`	Plot network performance for Bayesian regularization training
`plotconfusion`	Plot classification confusion matrix
`plotep`	Plot weight and bias position on error surface
`plotes`	Plot error surface of single-input neuron
`plotfit`	Plot function fit
`plotpc`	Plot classification line on perceptron vector plot
`plotperf`	Plot network performance
`plotperform`	Plot network performance
`plotpv`	Plot perceptron input target vectors
`plotregression`	Plot linear regression
`plotroc`	Plot receiver operating characteristic
`plotsom`	Plot self-organizing map
`plotsomhits`	Plot self-organizing map sample hits
`plotsomnc`	Plot self-organizing map neighbor connections
`plotsomnd`	Plot self-organizing map neighbor distances
`plotsomplanes`	Plot self-organizing map weight planes
`plotsompos`	Plot self-organizing map weight positions
`plotsomtop`	Plot self-organizing map topology
`plottrainstate`	Plot training state values
`plotv`	Plot vectors as lines from origin
`plotvec`	Plot vectors with different colors
`postreg`	Postprocess trained network response with linear regression

Processing Functions

`fixunknowns`	Process data by marking rows with unknown values
`mapminmax`	Process matrices by mapping row minimum and maximum values to `[-1 1]`
`mapstd`	Process matrices by mapping each row's means to 0 and deviations to 1
`processpca`	Process columns of matrix with principal component analysis
`removeconstantrows`	Process matrices by removing rows with constant values
`removerows`	Process matrices by removing rows with specified indices

Simulink® Support Function

`gensim`	Generate Simulink® block for neural network simulation

Topology Functions

`gridtop`	Gridtop layer topology function
`hextop`	Hexagonal layer topology function
`randtop`	Random layer topology function

Training Functions

`trainb`	Batch training with weight and bias learning rules
`trainbfg`	BFGS quasi-Newton backpropagation
`trainbfgc`	BFGS quasi-Newton backpropagation for use with NN model reference adaptive controller
`trainbr`	Bayesian regularization
`trainbuwb`	Batch unsupervised weight/bias training
`trainc`	Cyclical order incremental update
`traincgb`	Powell-Beale conjugate gradient backpropagation
`traincgf`	Fletcher-Powell conjugate gradient backpropagation
`traincgp`	Polak-Ribiére conjugate gradient backpropagation
`traingd`	Gradient descent backpropagation
`traingda`	Gradient descent with adaptive learning rule backpropagation
`traingdm`	Gradient descent with momentum backpropagation
`traingdx`	Gradient descent with momentum and adaptive learning rule backpropagation
`trainlm`	Levenberg-Marquardt backpropagation
`trainoss`	One step secant backpropagation
`trainr`	Random order incremental training with learning functions
`trainrp`	Resilient backpropagation (Rprop)
`trains`	Sequential order incremental training with learning functions
`trainscg`	Scaled conjugate gradient backpropagation

Transfer Functions

`compet`	Competitive transfer function
`hardlim`	Hard limit transfer function
`hardlims`	Symmetric hard limit transfer function
`logsig`	Log-sigmoid transfer function
`netinv`	Inverse transfer function
`poslin`	Positive linear transfer function
`purelin`	Linear transfer function
`radbas`	Radial basis transfer function
`satlin`	Saturating linear transfer function
`satlins`	Symmetric saturating linear transfer function
`softmax`	Softmax transfer function
`tansig`	Hyperbolic tangent sigmoid transfer function
`tribas`	Triangular basis transfer function

Utility Functions

`calcgx`	Calculate weight and bias performance gradient as single vector
`calcjejj`	Calculate Jacobian performance vector
`calcjx`	Calculate weight and bias performance Jacobian as single matrix
`calcpd`	Calculate delayed network inputs
`calcperf`	Calculate network outputs, signals, and performance
`getx`	All network weight and bias values as single vector
`setx`	Set all network weight and bias values with single vector

Vector Functions

`combvec`	Create all combinations of vectors
`con2seq`	Convert concurrent vectors to sequential vectors
`concur`	Create concurrent bias vectors
`ind2vec`	Convert indices to vectors
`minmax`	Ranges of matrix rows
`normc`	Normalize columns of matrix
`normr`	Normalize rows of matrix
`pnormc`	Pseudonormalize columns of matrix
`quant`	Discretize values as multiples of quantity
`seq2con`	Convert sequential vectors to concurrent vectors
`vec2ind`	Convert vectors to indices

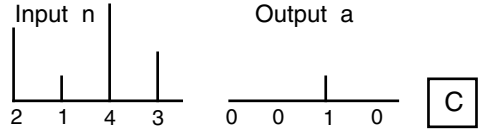
Weight and Bias Initialization Functions

`initcon`	Conscience bias initialization function
`initsompc`	Initialize SOM weights with principal components
`initzero`	Zero weight and bias initialization function
`midpoint`	Midpoint weight initialization function
`randnc`	Normalized column weight initialization function
`randnr`	Normalized row weight initialization function
`rands`	Symmetric random weight/bias initialization function
`revert`	Change network weights and biases to previous initialization values

Weight Functions

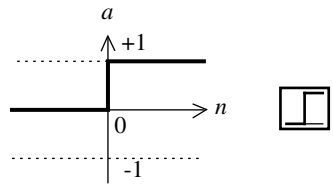
`convwf`	Convolution weight function
`dist`	Euclidean distance weight function
`dotprod`	Dot product weight function
`mandist`	Manhattan distance weight function
`negdist`	Negative distance weight function
`normprod`	Normalized dot product weight function
`scalprod`	Scalar product weight function

Transfer Function Graphs



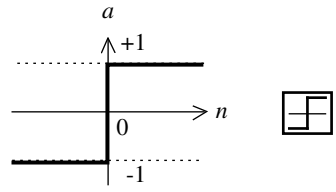
a = compet(n)

Compet Transfer Function



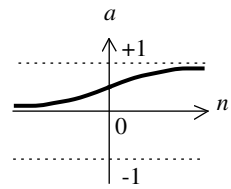
a = hardlim(n)

Hard-Limit Transfer Function



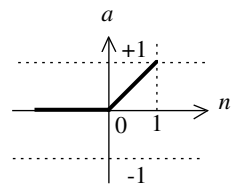
a = hardlims(n)

Symmetric Hard-Limit Transfer Function



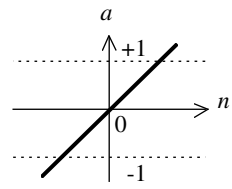
a = logsig(n)

Log-Sigmoid Transfer Function



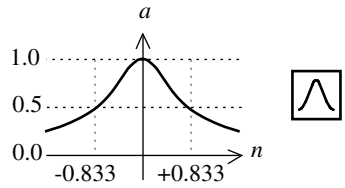
a = poslin(n)

Positive Linear Transfer Function



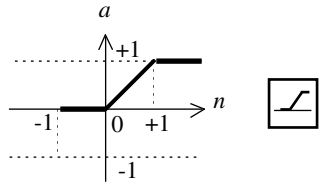
a = purelin(n)

Linear Transfer Function



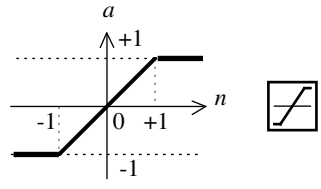
a = radbas(n)

Radial Basis Function



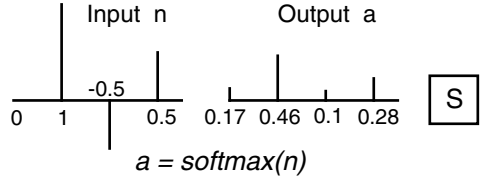
a = satlin(n)

Satlin Transfer Function



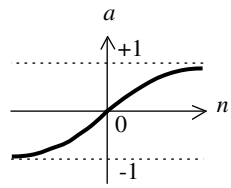
a = satlins(n)

Satlins Transfer Function



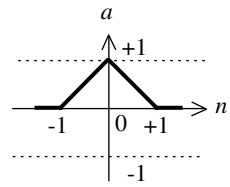
a = softmax(n)

Softmax Transfer Function



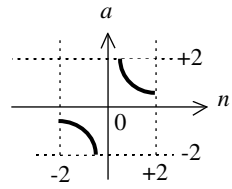
a = tansig(n)

Tan-Sigmoid Transfer Function



a = tribas(n)

Triangular Basis Function



a = netinv(n)

Netinv Transfer Function

16

Functions — Alphabetical List

Purpose

Allow neural network to change weights and biases on inputs

Syntax

```
[net,Y,E,Pf,Af] = adapt(net,P,T,Pi,Ai)
```

To Get Help

Type `help network/adapt`.

Description

This function calculates network outputs and errors after each presentation of an input.

`[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)` takes

`net`	Network
`P`	Network inputs
`T`	Network targets (default = zeros)
`Pi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

`net`	Updated network
`Y`	Network outputs
`E`	Network errors
`Pf`	Final input delay conditions
`Af`	Final layer delay conditions
`tr`	Training record (`epoch` and `perf`)

Note that `T` is optional and is only needed for networks that require targets. `Pi` and `Pf` are also optional and only need to be used for networks that have input or layer delays.

`adapt`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented,

`P`	`Ni` x `TS` cell array	Each element `P{i,ts}` is an `Ri x Q` matrix.
`T`	`Nt` x `TS` cell array	Each element `T{i,ts}` is a `Vi x Q` matrix.
`Pi`	`Ni` x `ID` cell array	Each element `Pi{i,k}` is an `Ri x Q` matrix.
`Ai`	`Nl` x `LD` cell array	Each element `Ai{i,k}` is an `Si x Q` matrix.
`Y`	`No` x `TS` cell array	Each element `Y{i,ts}` is a `Ui x Q` matrix.
`E`	`No` x `TS` cell array	Each element `E{i,ts}` is a `Ui x Q` matrix.
`Pf`	`Ni` x `ID` cell array	Each element `Pf{i,k}` is an `Ri x Q` matrix.
`Af`	`Nl` x `LD` cell array	Each element `Af{i,k}` is an `Si x Q` matrix.

where

`Ni = net.numInputs`

`Nl = net.numLayers`

`No = net.numOutputs`

`ID = net.numInputDelays`

`LD = net.numLayerDelays`

`TS` = Number of time steps

`Q` = Batch size

`Ri = net.inputs{i}.size`

`Si = net.layers{i}.size`

`Ui = net.outputs{i}.size`

The columns of Pi, Pf, Ai, and Af are ordered from oldest delay condition to most recent:

Pi{i,k} = Input i at time ts = k ID

Pf{i,k} = Input i at time ts = TS + k ID

Ai{i,k} = Layer output i at time ts = k LD

Af{i,k} = Layer output i at time ts = TS + k LD

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

P (sum of Ri) x Q matrix

T (sum of Vi) x Q matrix

Pi (sum of Ri) x (ID*Q) matrix

Ai (sum of Si) x (LD*Q) matrix

Y (sum of Ui) x Q matrix

E (sum of Ui) x Q matrix

Pf (sum of Ri) x (ID*Q) matrix

Af (sum of Si) x (LD*Q) matrix

Examples

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.

```
p1 = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t1 = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
```

Here newlin is used to create a layer with an input range of [-1 1], one neuron, input delays of 0 and 1, and a learning rate of 0.5. The linear layer is then simulated.

```
net = newlin([-1 1],1,[0 1],0.5);
```

Here the network adapts for one pass through the sequence.

The network's mean squared error is displayed. (Because this is the first call to `adapt`, the default `Pi` is used.)

```
[net,y,e,pf] = adapt(net,p1,t1);
mse(e)
```

Note that the errors are quite large. Here the network adapts to another 12 time steps (using the previous `Pf` as the new initial delay conditions).

```
p2 = {1 -1 -1 1 1 -1  0 0 0 1 -1 -1};
t2 = {2  0 -2 0 2  0 -1 0 0 1  0 -1};
[net,y,e,pf] = adapt(net,p2,t2,pf);
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```
p3 = [p1 p2];
t3 = [t1 t2];
net.adaptParam.passes = 100;
[net,y,e] = adapt(net,p3,t3);
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

Algorithm

`adapt` calls the function indicated by `net.adaptFcn`, using the adaption parameter values indicated by `net.adaptParam`.

Given an input sequence with `TS` steps, the network is updated as follows: Each step in the sequence of inputs is presented to the network one at a time. The network's weight and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated `TS` times.

See Also

`sim`, `init`, `train`, `revert`

boxdist

Purpose

Distance between two position vectors

Syntax

```
d = boxdist(pos);
```

Description

`boxdist` is a layer distance function that is used to find the distances between the layer's neurons, given their positions.

`boxdist(pos)` takes one argument,

`pos` `N` x `S` matrix of neuron positions

and returns the `S` x `S` matrix of distances.

`boxdist` is most commonly used with layers whose topology function is `gridtop`.

Examples

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
d = boxdist(pos)
```

Network Use

You can create a standard network that uses `boxdist` as a distance function by calling `newsom`.

To change a network so that a layer's topology uses `boxdist`, set `net.layers{i}.distanceFcn` to `'boxdist'`.

In either case, call `sim` to simulate the network with `boxdist`. See `newsom` for training and adaption examples.

Algorithm

The box distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

```
Dij = max(abs(Pi-Pj))
```

See Also

`sim`, `dist`, `mandist`, `linkdist`

Purpose

Calculate weight and bias performance gradient as single vector

Syntax

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS);
```

Description

This function calculates the gradient of a network's performance with respect to its vector of weight and bias values `X`.

If the network has no layer delays with taps greater than 0, the result is the true gradient.

If the network has layer delays greater than 0, the result is the Elman gradient, an approximation of the true gradient.

`[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS)` takes

`net`	Neural network
`X`	Vector of weight and bias values
`Pd`	Delayed inputs
`BZ`	Concurrent biases
`IWZ`	Weighted inputs
`LWZ`	Weighted layer outputs
`N`	Net inputs
`Ac`	Combined layer outputs
`El`	Layer errors
`perf`	Network performance
`Q`	Concurrent size
`TS`	Time steps

and returns

`gX`	Gradient `dPerf/dX`
`normgX`	Norm of gradient

Examples

Here is a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
X = getx(net);
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally you can use `calcgz` to calculate the gradient of performance with respect to the weight and bias values `X`.

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,1,5);
```

See Also

`calcjx`, `calcjejj`

Purpose

Calculate Jacobian performance vector

Syntax

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)
```

Description

This function calculates two values (related to the Jacobian of a network) required to calculate the network's Hessian, in a memory-efficient way.

Two values needed to calculate the Hessian of a network are `J*E` (Jacobian times errors) and `J'J` (Jacobian squared). However the Jacobian `J` can take up a lot of memory. This function calculates `J*E` and `J'J` by dividing training vectors into groups, calculating partial Jacobians `Ji` and its associated values `Ji*Ei` and `Ji'Ji`, then summing the partial values into the full `J*E` and `J'J` values.

This allows the `J*E` and `J'J` values to be calculated with a series of smaller `Ji` matrices instead of a larger `J` matrix.

`[je,jj,normgX] = calcjejj(net,PD,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)` takes

`net`	Neural network
`PD`	Delayed inputs
`BZ`	Concurrent biases
`IWZ`	Weighted inputs
`LWZ`	Weighted layer outputs
`N`	Net inputs
`Ac`	Combined layer outputs
`El`	Layer errors
`Q`	Concurrent size
`TS`	Time steps
`MR`	Memory reduction factor

and returns

`je`	Jacobian times errors

`jj`	Jacobian transposed times the Jacobian.normgX
`normgX`	Norm of gradient

Examples

Here is a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally you can use calcjejj to calculate the Jacobian times error, Jacobian squared, and the norm of the Jacobian times error, using a memory reduction of 2.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,2);
```

The results should be the same whatever the memory reduction used. Here a memory reduction of 3 is used.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,3);
```

See Also `calcjx`, `calcjejj`

calcjx

Purpose

Calculate weight and bias performance Jacobian as single matrix

Syntax

```
jx = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS)
```

Description

This function calculates the Jacobian of a network's errors with respect to its vector of weight and bias values `X`.

`[jX] = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS)` takes

`net`	Neural network
`PD`	Delayed inputs
`BZ`	Concurrent biases
`IWZ`	Weighted inputs
`LWZ`	Weighted layer outputs
`N`	Net inputs
`Ac`	Combined layer outputs
`Q`	Concurrent size
`TS`	Time steps

and returns

`jX`	Jacobian of network errors with respect to `X`

Examples

Here is a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally you can use `calcjx` to calculate the Jacobian.

```
jX = calcjx(net,Pd,BZ,IWZ,LWZ,N,Ac,1,5);calcpd
```

See Also

`calcgx`, `calcjejj`

calcpd

Purpose

Calculate delayed network inputs

Syntax

```
Pd = calcpd(net,TS,Q,Pc)
```

Description

This function calculates the results of passing the network inputs through each input weight's tap delay line.

`Pd = calcpd(net,TS,Q,Pc)` takes

`net`	Neural network
`TS`	Time steps
`Q`	Concurrent size
`Pc`	Combined inputs = [initial delay conditions, network inputs]

and returns

`Pd`	Delayed inputs

Examples

Here is a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps.

```
net = newlin([0 1],3,[0 2 4]);
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`).

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
```

Here the four initial input delay conditions `Pi` are defined.

```
Pi = {0.2 0.3 0.4 0.1};
```

The delayed inputs (the inputs after passing through the tap delays) can be calculated with `calcpd`.

```
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

You can view the delayed inputs for the input weight going to layer 1 from input 1 at time steps 1 and 2.

```
Pd{1,1,1}
Pd{1,1,2}
```

calcperf

Purpose

Calculate network outputs, signals, and performance

Syntax

```
[perf,El,Ac,N,BZ,IWZ,LWZ]=calcperf(net,X,Pd,Tl,Ai,Q,TS)
```

Description

This function calculates the outputs of each layer in response to a network's delayed inputs and initial layer delay conditions.

`[perf,El,Ac,N,LWZ,IWZ,BZ] = calcperf(net,X,Pd,Tl,Ai,Q,TS)` takes

`net`	Neural network
`X`	Network weight and bias values in a single vector
`Pd`	Delayed inputs
`Tl`	Layer targets
`Ai`	Initial layer delay conditions
`Q`	Concurrent size
`TS`	Time steps

and returns

`perf`	Network performance
`El`	Layer errors
`Ac`	Combined layer outputs = [`Ai`, calculated layer outputs]
`N`	Net inputs
`LWZ`	Weighted layer outputs
`IWZ`	Weighted inputs
`BZ`	Concurrent biases

Examples

Here is a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
```

```
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
```

Here the layer targets for the two neurons for each of the five time steps are defined.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted.

```
X = getx(net);
```

Here the network's combined outputs `Ac` and other signals described above are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5)
```

See Also

`calcpd`

combvec

Purpose

Create all combinations of vectors

Syntax

```
combvec(a1,a2...)
```

Description

`combvec(A1,A2...)` takes any number of inputs,

`A1`	Matrix of `N1` (column) vectors
`A2`	Matrix of `N2` (column) vectors

and returns a matrix of `(N1*N2*...)` column vectors, where the columns consist of all possibilities of `A2` vectors, appended to `A1` vectors, etc.

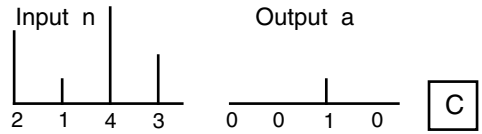
Examples

```
a1 = [1 2 3; 4 5 6];
a2 = [7 8; 9 10];
a3 = combvec(a1,a2)
```

Purpose

Competitive transfer function

Graph and Symbol



a = compet(n)

Compet Transfer Function

Syntax

```
A = compet(N,FP)
dA_dN = compet('dn',N,A,FP)
info = compet(code)
```

Description

`compet` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`compet(N,FP)` takes `N` and optional function parameters,

`N`	`S` x `Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns the `S` x `Q` matrix `A` with a 1 in each column where the same column of `N` has its maximum value, and 0 elsewhere.

`compet('dn',N,A,FP)` returns the derivative of `A` with respect to `N`. If `A` or `FP` is not supplied or is set to `[]`, `FP` reverts to the default parameters, and `A` is calculated from `N`.

`compet('name')` returns the name of this function.

`compet('output',FP)` returns the [min max] output range.

`compet('active',FP)` returns the [min max] active input range.

`compet('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S` x `S` x `Q` or `S` x `Q`.

`compet('fpnames')` returns the names of the function parameters.

`compet('fpdefaults')` returns the default function parameters.

Examples

Here you define a net input vector `N`, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = compet(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'compet';
```

See Also

`sim`, `softmax`

Purpose

Convert concurrent vectors to sequential vectors

Syntax

```
s = con2seq(b)
```

Description

The Neural Network Toolbox™ software arranges concurrent vectors with a matrix, and sequential vectors with a cell array (where the second index is the time step).

`con2seq` and `seq2con` allow concurrent vectors to be converted to sequential vectors, and back again.

`con2seq(b)` takes one input,

`b`	`R` x `TS` matrix

and returns one output,

`S`	`1` x `TS` cell array of `R` x `1` vectors

`con2seq(b,TS)` can also convert multiple batches,

`b`	`N` x `1` cell array of matrices with `M`*`TS` columns
`TS`	Time steps

and returns

`S`	`N` x `TS` cell array of matrices with `M` columns

Examples

Here a batch of three values is converted to a sequence.

```
p1 = [1 4 2]
p2 = con2seq(p1)
```

Here two batches of vectors are converted to two sequences with two time steps.

```
p1 = {[1 3 4 5; 1 1 7 4]; [7 3 4 4; 6 9 4 1]}
p2 = con2seq(p1,2)
```

See Also

`seq2con`, `concur`

concur

Purpose

Create concurrent bias vectors

Syntax

```
concur(B,Q)
```

Description

`concur(B,Q)`

`B`	`S` x `1` bias vector (or `Nl` x `1` cell array of vectors)
`Q`	Concurrent size

Returns an `S` x `B` matrix of copies of `B` (or `Nl` x `1` cell array of matrices).

Examples

Here `concur` creates three copies of a bias vector.

```
b = [1; 3; 2; -1];
concur(b,3)
```

Network Use

To calculate a layer's net input, the layer's weighted inputs must be combined with its biases. The following expression calculates the net input for a layer with the `netsum` net input function, two input weights, and a bias:

```
n = netsum(z1,z2,b)
```

The above expression works if `Z1`, `Z2`, and `B` are all `S` x `1` vectors. However, if the network is being simulated by `sim` (or `adapt` or `train`) in response to `Q` concurrent vectors, then `Z1` and `Z2` will be `S` x `Q` matrices. Before `B` can be combined with `Z1` and `Z2`, you must make `Q` copies of it.

```
n = netsum(z1,z2,concur(b,q))
```

See Also

`netsum`, `netprod`, `sim`, `seq2con`, `con2seq`

Purpose

Classification confusion matrix

Syntax

```
[c,cm,ind,per] = confusion(targets,outputs)
```

Description

`[c,cm,ind,per] = confusion(targets,outputs)` takes these values:

`targets` — `S` x `Q` matrix, where each column vector contains a single `1` value, with all other elements `0`. The index of the `1` indicates which of `S` categories that vector represents.

`outputs` — `S` x `Q` matrix, where each column contains values in the range `[0.1]`. The index of the largest element in the column indicates which of `S` categories that vector represents.

and returns these values:

`c` — Confusion value = fraction of samples misclassified

`cm` — `S` x `S` confusion matrix, where `cm(i,j)` is the number of samples whose target is the `i`th class that was classified as `j`

`ind` — `S` x `S` cell array, where `ind{i,j}` contains the indices of samples with the `i`th target class, but `j`th output class

`per` — `S` x `3` matrix, where each `i`th row represents the percentage of false negatives, false positives, and true positives for the `i`th category

`[c,cm,ind,per] = confusion(TARGETS,OUTPUTS)` takes these values:

`targets` — `1` x `Q` vector of 1/0 values representing membership

`outputs` — `S` x `Q` matrix, of value in `[0.1]` interval, where values greater than or equal to `0.5` indicate class membership

and returns these values:

`c` — Confusion value = fraction of samples misclassified

`cm` — `2` x `2` confusion matrix

`ind` 2 x 2 cell array, where `ind{i,j}` contains the indices of samples whose target is `1` versus `0`, and whose output was greater than or equal to `0.5` versus less than `0.5`

`per` 2 x 3 matrix where each `i`th row represents the percentage of false negatives, false positives, and true positives for the class and out-of-class

Examples

```
load simpleclass_dataset
net = newpr(simpleclassInputs,simpleclassTargets,20);
net = train(net,simpleclassInputs,simpleclassTargets);
simpleclassOutputs = sim(net,simpleclassInputs);
[c,cm,ind,per] = ...
confusion(simpleclassTargets,simpleclassOutputs)
```

See Also

`plotconfusion`, `roc`

Purpose

Convolution weight function

Syntax

```
Z = convwf(W,P)
dim = convwf('size',S,R,FP)
dp = convwf('dp',W,P,Z,FP)
dw = convwf('dw',W,P,Z,FP)
info = convwf(code)
```

Description

`convwf` is the convolution weight function. Weight functions apply weights to an input to get weighted inputs.

`convwf(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Reduced derivative = 2, full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

`convwf('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size.

`convwf('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`convwf('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,1);
P = rand(8,1);
```

```
Z = convwf(W,P)
```

Network Use

To change a network so an input weight uses `convwf`, set `net.inputWeight{i,j}.weightFcn` to `'convwf'`. For a layer weight, set `net.layerWeight{i,j}.weightFcn` to `'convwf'`.

In either case, call `sim` to simulate the network with `convwf`.

Purpose

Neural network's properties

Syntax

```
disp(net)
```

To Get Help

Type `help network/disp`.

Description

`disp(net)` displays a network's properties.

Examples

Here a perceptron is created and displayed.

```
net = newp([-1 1; 0 2],3);
disp(net)
```

See Also

`display`, `sim`, `init`, `train`, `adapt`

display

Purpose

Name and properties of neural network's variables

Syntax

```
display(net)
```

To Get Help

Type `help network/display`.

Description

`display(net)` displays a network variable's name and properties.

Examples

Here a perceptron variable is defined and displayed.

```
net = newp([-1 1; 0 2],3);
display(net)
```

`display` is automatically called as follows:

```
net
```

See Also

`disp`, `sim`, `init`, `train`, `adapt`

Purpose

Euclidean distance weight function

Syntax

```
Z = dist(W,P,FP)
info = dist(code)
dim = dist('size',S,R,FP)
dp = dist('dp',W,P,Z,FP)
dw = dist('dw',W,P,Z,FP)
D = dist(pos)
```

Description

`dist` is the Euclidean distance weight function. Weight functions apply weights to an input to get weighted inputs.

`dist(W,P,FP)` takes these inputs,

`W`	`S` x `R` weight matrix
`P`	`R` x `Q` matrix of `Q` input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

and returns the `S` x `Q` matrix of vector distances.

`dist(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

`dist('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S` x `R`].

`dist('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`dist('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

dist is also a layer distance function which can be used to find the distances between neurons in a layer.

dist(pos) takes one argument,

pos	N x S matrix of neuron positions

and returns the S x S matrix of distances.

Examples

Here you define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = dist(W,P)
```

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = dist(pos)
```

Network Use

You can create a standard network that uses dist by calling newpnn or newgrnn.

To change a network so an input weight uses dist, set net.inputWeight{i,j}.weightFcn to 'dist'. For a layer weight, set net.layerWeight{i,j}.weightFcn to 'dist'.

To change a network so that a layer's topology uses dist, set net.layers{i}.distanceFcn to 'dist'.

In either case, call sim to simulate the network with dist.

See newpnn or newgrnn for simulation examples.

Algorithm

The Euclidean distance d between two vectors X and Y is

```
d = sum((x-y).^2).^0.5
```

See Also

sim, dotprod, negdist, normprod, mandist, linkdist

Purpose

Divide vectors into three sets using blocks of indices

Syntax

```
[trainV,valV,testV,trainInd,valInd,testInd] =
  divideblock(allV,trainRatio,valRatio,testRatio)
```

Description

`divideblock` is used to separate input and target vectors into three sets: training, validation, and testing. It takes the following inputs:

`allV`	`R` x `Q` matrix of `Q` `R`-element vectors.
`trainRatio`	Ratio of vectors for training. Default = `0.6`.
`valRatio`	Ratio of vectors for validation. Default = `0.2`.
`testRatio`	Ratio of vectors for testing. Default = `0.2`.

and returns

`trainV`	Training vectors
`valV`	Validation vectors
`testV`	Test vectors
`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
p = rands(3,1000);
t = [p(1,:).*p(2,:); p(2,:).*p(3,:)];
[trainP,valP,testV,trainInd,valInd,testInd] =
  divideblock(p,0.6,0.2,0.2);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Network Use

Here are the network properties that define which data division function to use, and what its parameters are, when `train` is called.

```
net.divideFcn
net.divideParam
```

See Also

`divideind`, `divideint`, `dividerand`

divideind

Purpose

Divide vectors into three sets using specified indices

Syntax

```
[trainV,valV,testV,trainInd,valInd,testInd] =
  divideind(allV,trainInd,valInd,testInd)
```

Description

`divideind` is used to separate input and target vectors into three sets: training, validation, and testing. It takes the following inputs,

`allV`	`R` x `Q` matrix of `Q` `R`-element vectors
`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

and returns

`trainV`	Training vectors
`valV`	Validation vectors
`testV`	Test vectors
`trainInd`	Training indices (unchanged)
`valInd`	Validation indices (unchanged)
`testInd`	Test indices (unchanged)

Examples

```
p = rands(3,1000);
trainInd = [(1:100) (301:800)];
valInd = [(101:200) (801:900)];
testInd = [(201:300) (901:1000)];
[trainP,valP,testV] = divideind(p,trainInd,valInd,testInd);
```

Network Use

Here are the network properties that define which data division function to use, and what its parameters are, when `train` is called.

```
net.divideFcn
net.divideParam
```

See Also

`divideblock`, `divideint`, `dividerand`

Purpose

Divide vectors into three sets using interleaved indices

Syntax

```
[trainV,valV,testV,trainInd,valInd,testInd] =
  divideint(allV,trainRatio,valRatio,testRatio)
```

Description

`divideint` is used to separate input and target vectors into three sets: training, validation, and testing. It takes the following inputs,

`allV`	`R` x `Q` matrix of `Q` `R`-element vectors.
`trainRatio`	Ratio of vectors for training. Default = `0.6`.
`valRatio`	Ratio of vectors for validation. Default = `0.2`.
`testRatio`	Ratio of vectors for testing. Default = `0.2`.

and returns

`trainV`	Training vectors
`valV`	Validation vectors
`testV`	Test vectors
`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
p = rands(3,1000);
t = [p(1,:).*p(2,:); p(2,:).*p(3,:)];
[trainP,valP,testV,trainInd,valInd,testInd] =
   divideint(p,0.6,0.2,0.2);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Network Use

Here are the network properties that define which data division function to use, and what its parameters are, when `train` is called.

```
net.divideFcn
net.divideParam
```

See Also

`divideblock`, `divideind`, `dividerand`

dividerand

Purpose

Divide vectors into three sets using random indices

Syntax

```
[trainV,valV,testV,trainInd,valInd,testInd] =
  dividerand(allV,trainRatio,valRatio,testRatio)
```

Description

`dividerand` is used to separate input and target vectors into three sets: training, validation, and testing. It takes the following inputs,

`allV`	`R` x `Q` matrix of `Q` `R`-element vectors.
`trainRatio`	Ratio of vectors for training. Default = `0.6`.
`valRatio`	Ratio of vectors for validation. Default = `0.2`.
`testRatio`	Ratio of vectors for testing. Default = `0.2`.

and returns

`trainV`	Training vectors
`valV`	Validation vectors
`testV`	Test vectors
`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
p = rands(3,1000);
t = [p(1,:).*p(2,:); p(2,:).*p(3,:)];
[trainP,valP,testV,trainInd,valInd,testInd] =
   dividerand(p,0.6,0.2,0.2);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Network Use

Here are the network properties that define which data division function to use, and what its parameters are, when `train` is called.

```
net.divideFcn
net.divideParam
```

See Also

`divideblock`, `divideind`, `divideint`

Purpose

Dot product weight function

Syntax

```
Z = dotprod(W,P,FP)
info = dotprod(code)
dim = dotprod('size',S,R,FP)
dp = dotprod('dp',W,P,Z,FP)
dw = dotprod('dw',W,P,Z,FP)
```

Description

`dotprod` is the dot product weight function. Weight functions apply weights to an input to get weighted inputs.

`dotprod(W,P,FP)` takes these inputs,

`W`	`S` x `R` weight matrix
`P`	`R` x `Q` matrix of `Q` input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

and returns the `S` x `Q` dot product of `W` and `P`.

`dotprod(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

`dotprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S` x `R`].

`dotprod('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`dotprod('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

Examples

Here you define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = dotprod(W,P)
```

Network Use

You can create a standard network that uses `dotprod` by calling `newp` or `newlin`.

To change a network so an input weight uses `dotprod`, set `net.inputWeight{i,j}.weightFcn` to `'dotprod'`. For a layer weight, set `net.layerWeight{i,j}.weightFcn` to `'dotprod'`.

In either case, call `sim` to simulate the network with `dotprod`.

See `newp` and `newlin` for simulation examples.

See Also

`sim`, `dist`, `negdist`, `normprod`

Purpose

Error surface of single-input neuron

Syntax

```
errsurf(P,T,WV,BV,F)
```

Description

`errsurf(P,T,WV,BV,F)` takes these arguments,

`P`	1 x `Q` matrix of input vectors
`T`	1 x `Q` matrix of target vectors
`WV`	Row vector of values of `W`
`BV`	Row vector of values of `B`
`F`	Transfer function (string)

and returns a matrix of error values over `WV` and `BV`.

Examples

```
p = [-6.0 -6.1 -4.1 -4.0 +4.0 +4.1 +6.0 +6.1];
t = [+0.0 +0.0 +.97 +.99 +.01 +.03 +1.0 +1.0];
wv = -1:.1:1; bv = -2.5:.25:2.5;
es = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```

See Also

`plotes`

fixunknowns

Purpose

Process data by marking rows with unknown values

Syntax

```
[y,ps] = fixunknowns(x)
[y,ps] = fixunknowns(x,fp)
y = fixunknowns('apply',x,ps)
x = fixunknowns('reverse',y,ps)
dx_dy = fixunknowns('dx',x,y,ps)
dx_dy = fixunknowns('dx',x,[],ps)
name = fixunknowns('name');
fp = fixunknowns('pdefaults');
names = fixunknowns('pnames');
fixunknowns('pcheck',fp);
```

Description

`fixunknowns` processes matrixes by replacing each row containing unknown values (represented by NaN) with two rows of information.

The first row contains the original row, with NaN values replaced by the row's mean. The second row contains 1 and 0 values, indicating which values in the first row were known or unknown, respectively.

`fixunknowns(X)` takes these inputs,

`X`	Single `N` x `Q` matrix or a `1` x `TS` row cell array of `N` x `Q` matrices

and returns

`Y`	Each `M` x `Q` matrix with `M` `N` rows added (optional)
`PS`	Process settings that allow consistent processing of values

`fixunknowns(X,FP)` takes an empty struct `FP` of parameters.

`fixunknowns('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`fixunknowns('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`fixunknowns('dx',X,Y,PS)` returns the `M` x `N` x `Q` derivative of `Y` with respect to `X`.

`fixunknowns('dx',X,[],PS)` returns the derivative, less efficiently.

`fixunknowns('name')` returns the name of this process method.

`fixunknowns('pdefaults')` returns the default process parameter structure.

`fixunknowns('pdesc')` returns the process parameter descriptions.

`fixunknowns('pcheck',fp)` throws an error if any parameter is illegal.

Examples

Here is how to format a matrix with a mixture of known and unknown values in its second row:

```
x1 = [1 2 3 4; 4 NaN 6 5; NaN 2 3 NaN]
[y1,ps] = fixunknowns(x1)
```

Next, apply the same processing settings to new values:

```
x2 = [4 5 3 2; NaN 9 NaN 2; 4 9 5 2]
y2 = fixunknowns('apply',x2,ps)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = fixunknowns('reverse',y1,ps)
```

See Also

`mapminmax`, `mapstd`, `processpca`

gensim

Purpose

Generate Simulink® block for neural network simulation

Syntax

```
gensim(net,st)
```

To Get Help

Type `help network/gensim`.

Description

`gensim(net,st)` creates a Simulink system containing a block that simulates neural network `net`.

`gensim(net,st)` takes these inputs:

`net`	Neural network
`st`	Sample time (default = 1)

and creates a Simulink system containing a block that simulates neural network `net` with a sampling time of `st`.

If `net` has no input or layer delays (`net.numInputDelays` and `net.numLayerDelays` are both 0) then you can use -1 for `st` to get a network that samples continuously.

Examples

```
p = [0 1 2; 1 2 0];
t = [1 0 1];
net = newff(p,t,5);
gensim(net)
```

Purpose

All network weight and bias values as single vector

Syntax

```
X = getx(net)
```

Description

`getx` gets a network's weight and biases as a vector of values.

```
X = getx(net)
```

`net`	Neural network
`X`	Vector of weight and bias values

Examples

Here is a network with a two-element input and one layer of three neurons.

```
net = newff([0 1 2; -1 1 0],[-1 1 0]);
```

You can get its weight and bias values as follows:

```
net.iw{1,1}
net.b{1}
```

Get these values as a single vector as follows:

```
x = getx(net);
```

See Also

`setx`

gridtop

Purpose

Grid layer topology function

Syntax

```
pos = gridtop(dim1,dim2,...,dimN)
```

Description

`gridtop` calculates neuron positions for layers whose neurons are arranged in an `N`-dimensional grid.

`gridtop(dim1,dim2,...,dimN)` takes `N` arguments,

`dimi` Length of layer in dimension `i`

and returns an `N` x `S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in an 8-by-5 grid.

```
pos = gridtop(8,5); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly, so the layer is very disorganized, as is evident in the plot generated by the following code:

```
W = rands(40,2); plotsom(W,dist(pos))
```

See Also

`hextop`, `randtop`

Purpose

Hard limit transfer function

Graph and Symbol

Syntax

```
A = hardlim(N,FP)
dA_dN = hardlim('dn',N,A,FP)
info = hardlim(code)
```

Description

hardlim is a neural transfer function. Transfer functions calculate a layer's output from its net input.

hardlim(N,FP) takes N and optional function parameters,

and returns A, the S x Q Boolean matrix with 1's where N 0.

hardlim('dn',N,A,FP) returns the S x Q derivative of A with respect to N. If A or FP is not supplied or is set to [] , FP reverts to the default parameters, and A is calculated from N.

hardlim('name') returns the name of this function.

hardlim('output',FP) returns the [min max] output range.

hardlim('active',FP) returns the [min max] active input range.

hardlim('fullderiv') returns 1 or 0, depending on whether dA_dN is S x S x Q or S x Q.

hardlim ('fpnames') returns the names of th e function parameters.

hardlim('fpdefaults') returns the default function parameters.

Training occurs according to `trainrp`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.delt_inc`	`1.2`	Increment to weight change
`net.trainParam.delt_dec`	`0.5`	Decrement to weight change
`net.trainParam.delta0`	`0.07`	Initial weight change
`net.trainParam.deltamax`	`50.0`	Maximum weight change

`trainrp('info')` returns useful information about this function.

Network Use

You can create a standard network that uses `trainrp` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainrp`,

1. Set `net.trainFcn` to `'trainrp'`. This sets `net.trainParam` to `trainrp`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainrp`.

Examples

Here is a problem consisting of inputs `p` and targets `t` to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

A two-layer feed-forward network is created. The network's input ranges from `[0` to `10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `trainrp` network training function is to be used.

Create and test a network.

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainrp');
a = sim(net,p)
```

Here the network is trained and retested.

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

Algorithm

`trainrp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
dX = deltaX.*sign(gX);
```

where the elements of `deltaX` are all initialized to `delta0`, and `gX` is the gradient. At each iteration the elements of `deltaX` are modified. If an element of `gX` changes sign from one iteration to the next, then the corresponding element of `deltaX` is decreased by `delta_dec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltaX` is increased by `delta_inc`. See Riedmiller, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, 1993, pp. 586 to 591.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

Reference

Riedmiller, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, 1993, pp. 586–591

See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `traincgp`, `traincgf`, `traincgb`, `trainscg`, `trainoss`, `trainbfg`

trains

Purpose

Sequential order incremental training with learning functions

Syntax

```
[net,TR,Ac,El] = trains(net,Pd,Tl,Ai,Q,TS)
info = trains(code)
```

Description

`trains` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trains'`.

`trains` trains a network with weight and bias learning rules with sequential updates. The sequence of inputs is presented to the network with updates occurring after each time step.

This incremental training algorithm is commonly used for adaptive applications.

`trains` takes these inputs:

`net`	Neural network
`Pd`	Delayed inputs
`Tl`	Layer targets
`Ai`	Initial input conditions
`Q`	Batch size
`TS`	Time steps

and after training the network with its weight and bias learning functions returns

`net`	Updated network	
`TR`	Training record:	
	`TR.timesteps`	Number of time steps
	`TR.perf`	Performance for each time step
`Ac`	Collective layer outputs	
`El`	Layer errors	

Training occurs according to trains's training parameter, shown here with its default value:

net.trainParam.passes	1	Number of times to present sequence

Dimensions for these variables are

Pd	Nl x Ni x TS cell array	Each Pd{i,j,ts} is a Dij x Q matrix.
Tl	Nl x TS cell array	Each Tl{i,ts} is a Ui x Q matrix or [].
Ai	Nl x LD cell array	Each Ai{i,k} is an Si x Q matrix.
Ac	Nl x (LD+TS) cell array	Each Ac{i,k} is an Si x Q matrix.
El	Nl x TS cell array	Each El{i,k} is an Si x Q matrix or [].

where

```
Ni   =  net.numInputs
Nl   =  net.numLayers
LD   =  net.numLayerDelays
Ri   =  net.inputs{i}.size
Si   =  net.layers{i}.size
Ui   =  net.outputs{i}.size
Dij  =  Ri * length(net.inputWeights{i,j}.delays)
```

trains(code) returns useful information for each code string:

'pnames'	Names of training parameters
'pdefaults'	Default training parameters

Network Use

You can create a standard network that uses trains for adapting by calling newp or newlin.

To prepare a custom network to adapt with trains,

1. Set net.adaptFcn to 'trains'. This sets net.adaptParam to trains's default parameters.

2 Set each `net.inputWeights{i,j}.learnFcn` to a learning function. Set each `net.layerWeights{i,j}.learnFcn` to a learning function. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To allow the network to adapt,

1 Set weight and bias learning parameters to desired values.

2 Call `adapt`.

See `newp` and `newlin` for adaption examples.

Algorithm

Each weight and bias is updated according to its learning function after each time step in the input sequence.

See Also

`newp`, `newlin`, `train`, `trainb`, `trainc`, `trainr`

Purpose

Scaled conjugate gradient backpropagation

Syntax

```
[net,TR,Ac,El] = trainscg(net,TR,trainV,valV,testV)
info = trainscg('info')
```

Description

`trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

`trainscg(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`NoxTS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.
`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `trainscg`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (NaN for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.sigma`	`5.0e-5`	Determine change in weight for second derivative approximation
`net.trainParam.lambda`	`5.0e-7`	Parameter for regulating the indefiniteness of the Hessian

`trainscg('info')` returns useful information about this function.

Network Use

You can create a standard network that uses `trainscg` with `newff`, `newcf`, or `newelm`. To prepare a custom network to be trained with `trainscg`,

1. Set `net.trainFcn` to `'trainscg'`. This sets `net.trainParam` to `trainscg`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainscg`.

Examples

Here is a problem consisting of inputs `p` and targets `t` to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `trainscg` network training function is to be used.

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainscg');
a = sim(net,p)
```

Here the network is trained and retested.

```
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

Algorithm

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`.

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincgf`, and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*, Vol. 6, 1993, pp. 525 to 533) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

Reference

Moller, *Neural Networks*, Vol. 6, 1993, pp. 525–533

See Also

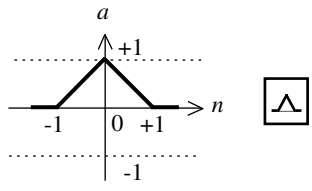
`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `trainrp`, `traincgf`, `traincgb`, `trainbfg`, `traincgp`, `trainoss`

tribas

Purpose

Triangular basis transfer function

Graph and Symbol



a = tribas(n)

Triangular Basis Function

Syntax

```
A = tribas(N,FP)
dA_dN = tribas('dn',N,A,FP)
info = tribas(code)
```

Description

`tribas` is a neural transfer function. Transfer functions calculate a layer's output from its net input.

`tribas(N,FP)` takes `N` and optional function parameters,

`N`	`S` x `Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, an `S` x `Q` matrix of the triangular basis function applied to each element of `N`.

`tribas('dn',N,A,FP)` returns the `S` x `Q` derivative of `A` with respect to `N`. If `A` or `FP` is not supplied or is set to `[]`, `FP` reverts to the default parameters, and `A` is calculated from `N`.

`tribas('name')` returns the name of this function.

`tribas('output',FP)` returns the [min max] output range.

`tribas('active',FP)` returns the [min max] active input range.

`tribas('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S` x `S` x `Q` or `S` x `Q`.

`tribas('fpnames')` returns the names of the function parameters.

`tribas('fpdefaults')` returns the default function parameters.

Examples

Here you create a plot of the `tribas` transfer function.

```
n = -5:0.1:5;
a = tribas(n);
plot(n,a)
```

Assign this transfer function to layer `i` of a network.

```
net.layers{i}.transferFcn = 'tribas';
```

Algorithm

```
a = tribas(n) = 1 - abs(n), if -1 <= n <= 1
              = 0, otherwise
```

See Also

`sim`, `radbas`

vec2ind

Purpose

Convert vectors to indices

Syntax

```
ind = vec2ind(vec)
```

Description

`ind2vec` and `vec2ind` allow indices to be represented either by themselves or as vectors containing a 1 in the row of the index they represent.

`vec2ind(vec)` takes one argument,

`vec`	Matrix of vectors, each containing a single 1

and returns the indices of the 1s.

Examples

Here four vectors (each containing only one “1” element) are defined, and the indices of the 1s are found.

```
vec = [1 0 0 0; 0 0 1 0; 0 1 0 1]
ind = vec2ind(vec)
```

See Also

`ind2vec`

Purpose

View neural network

Syntax

```
view(net)
```

Description

Use this function to launch a window that shows your neural network (specified in net) as a graphical diagram.

A

Mathematical Notation

Mathematical Notation for Equations and Figures (p. A-2)
Mathematics and Code Equivalents (p. A-4)

Mathematical Notation for Equations and Figures

Basic Concepts

	Description	Example
Scalars	Small *italic* letters	a, b, c
Vectors	Small **bold** nonitalic letters	$\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$
Matrices	Capital **BOLD** nonitalic letters	$\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$

Language

Vector means a column of numbers.

Weight Matrices

Scalar element	$w_{i,j}$	
Matrix	$\mathbf{W}$	
Column vector	$\mathbf{w}_j$	
Row vector	${}_i\mathbf{w}$	Vector made of `i`th row of weight matrix $\mathbf{W}$

Bias Elements and Vectors

Scalar element	b_i
Bias vector	$\mathbf{b}$

Time and Iteration

Weight matrix at time t	$\mathbf{W}(t)$
Weight matrix on iteration k	$\mathbf{W}(k)$

Layer Notation

A single superscript is used to identify elements of a layer. For instance, the net input of layer 3 would be shown as $\mathbf{n}^3$.

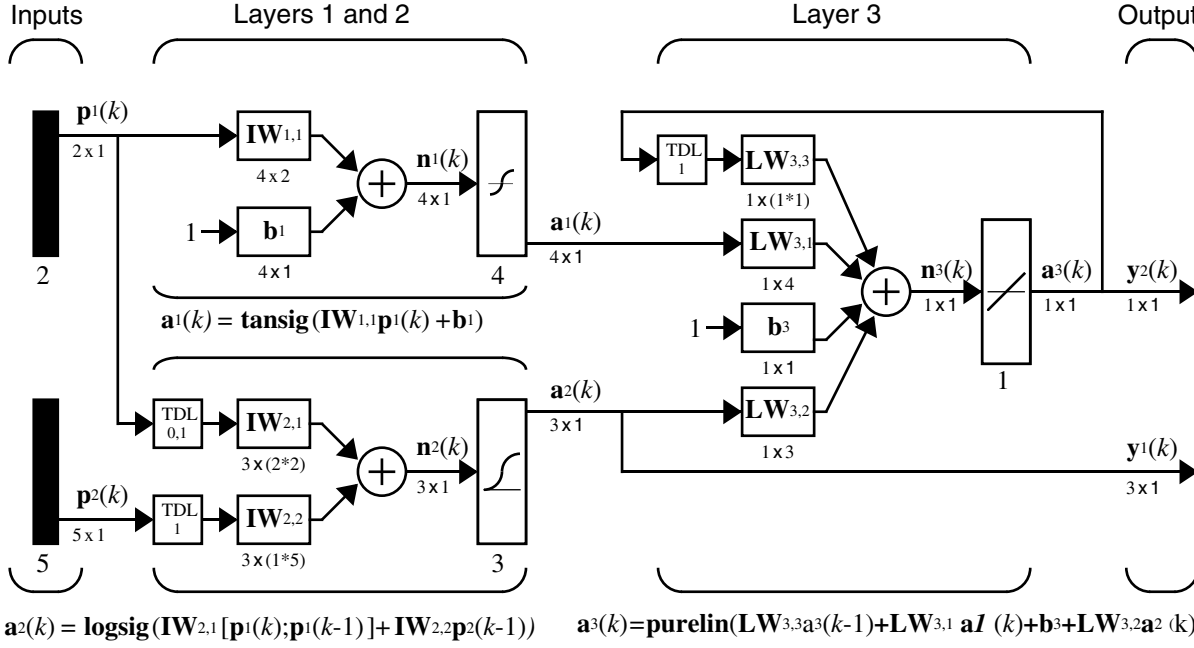
Superscripts k, l are used to identify the source (l) connection and the destination (k) connection of layer weight matrices and input weight matrices. For instance, the layer weight matrix from layer 2 to layer 4 would be shown as $\mathbf{LW}^{4,2}$.

Input weight matrix $\mathbf{IW}^{k,l}$

Layer weight matrix $\mathbf{LW}^{k,l}$

Figure and Equation Examples

The following figure, taken from Chapter 12, "Advanced Topics," illustrates notation used in such advanced figures.



Mathematics and Code Equivalents

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.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + ... + w_{1,R}p_R + b$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

B

Demonstrations and Applications

Tables of Demonstrations and Applications

Chapter 2, "Neuron Model and Network Architectures"

	File name	Page
Simple neuron and transfer functions	nnd2n1	2-4
Neuron with vector input	nnd2n2	2-7

Chapter 3, "Perceptrons"

	File name	Page
Decision boundaries	nnd4db	3-4
Perceptron learning rule, picking boundaries	nnd4pr	3-14
Classification with a two-input perceptron	demop1	3-20
Outlier input vectors	demop4	3-21
Normalized perceptron rule	demop5	3-22
Linearly nonseparable vectors	demop6	3-21

Chapter 4, "Linear Filters"

	File name	Page
Pattern association showing error surface	demolin1	4-9
Training a linear neuron	demolin2	4-16
Linear classification system	nnd10lc	4-16
Linear fit of nonlinear problem	demolin4	4-18
Underdetermined problem	demolin5	4-18
Linearly dependent problem	demolin6	4-19
Too large a learning rate	demolin7	4-19

Chapter 5, "Backpropagation"

	File name	Page
Generalization	nnd11gn	5-52
Steepest descent backpropagation	nnd12sd1	5-16
Momentum backpropagation	nnd12mo	5-18
Variable learning rate backpropagation	nnd12vl	5-20
Conjugate gradient backpropagation	nnd12cg	5-24
Marquardt backpropagation	nnd12m	5-32
Sample training session	demobp1	5-70

Chapter 8, "Radial Basis Networks"

	File name	Page
Radial basis approximation	demorb1	8-8
Radial basis underlapping neurons	demorb3	8-8
Radial basis overlapping neurons	demorb4	8-8
GRNN function approximation	demogrn1	8-14
PNN classification	demopnn1	8-11

Chapter 9, "Self-Organizing and Learning Vector Quantization Nets"

	File name	Page
Competitive learning	democ1	9-8
One-dimensional self-organizing map	demosm1	9-22
Two-dimensional self-organizing map	demosm2	9-22
Learning vector quantization	demolvq1	9-42

Chapter 10, "Adaptive Filters and Adaptive Training"

	File name	Page
Adaptive noise cancellation, toolbox example	demolin8	10-16
Adaptive noise cancellation in airplane cockpit	nnd10nc	10-14

Chapter 11, "Applications"

	File name	Page
Linear design	`applin1`	11-3
Adaptive linear prediction	`applin2`	11-7
Elman amplitude detection	`appelm1`	11-11
Character recognition	`appcr1`	11-15

Chapter 13, "Historical Networks"

	File name	Page
Hopfield two neuron design	`demohop1`	13-14
Hopfield unstable equilibria	`demohop2`	13-14
Hopfield three neuron design	`demohop3`	13-14
Hopfield spurious stable points	`demohop4`	13-14

B

Blocks for the Simulink® Environment

Blockset (p. B-2)

Block Generation (p. B-5)

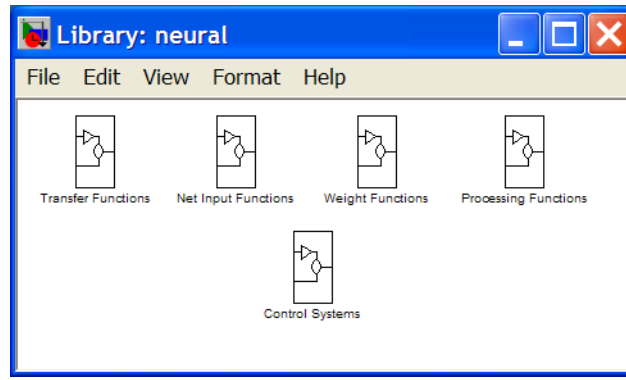
Blockset

The Neural Network 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.

Bring up the Neural Network Toolbox blockset with this command:

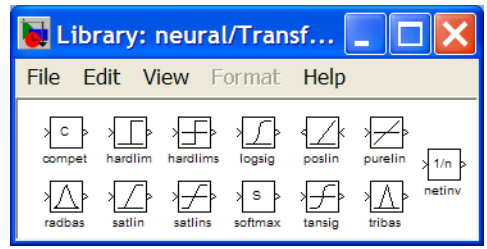
```
neural
```

The result is a window that contains five blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

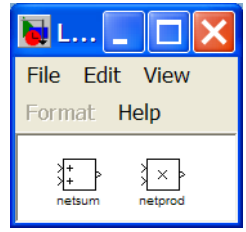
Double-click the Transfer Functions block in the Neural window to bring up a window containing several transfer function blocks.



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

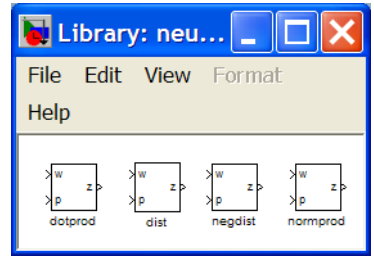
Double-click the Net Input Functions block in the Neural window to bring up a window containing two net-input function blocks.



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 window to bring up 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.

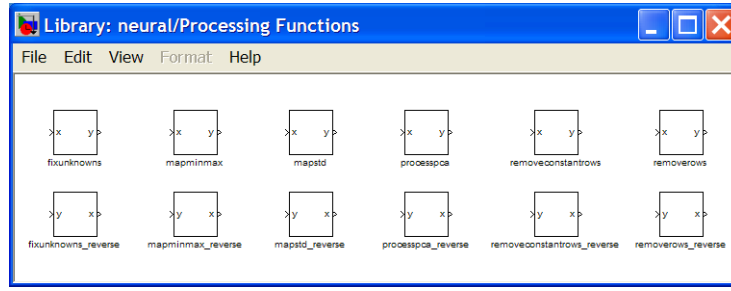
It is important to note that the blocks above expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

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 window to bring up a window containing five processing blocks and their corresponding reverse-processing blocks.



Each of these blocks can be used to pre-process inputs and post-process outputs.

Block Generation

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 tells 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 our original inputs with sim.

```
y = sim(net,p)
```

The results show the network has solved the problem.

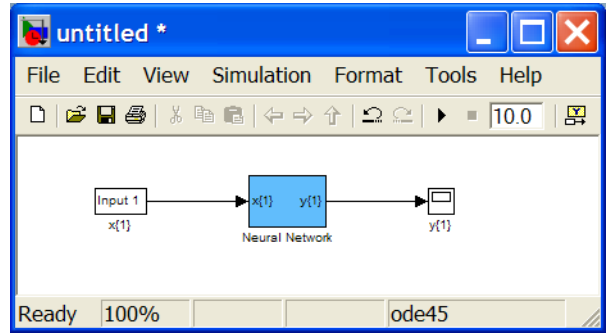
```
y =
      1.0000    3.0000    5.0000    7.0000    9.0000
```

Call gensim as follows to generate a Simulink version of the network.

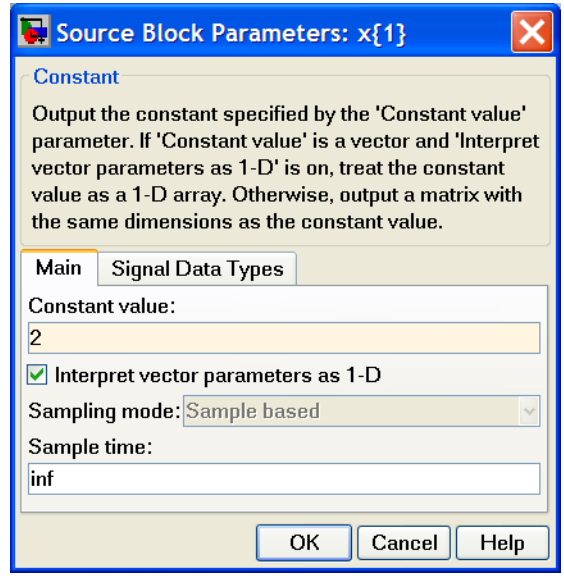
```
gensim(net,-1)
```

The second argument is -1, so the resulting network block samples continuously.

The call to gensim results in the following screen. It contains a Simulink system consisting of the linear network connected to a sample input and a scope.



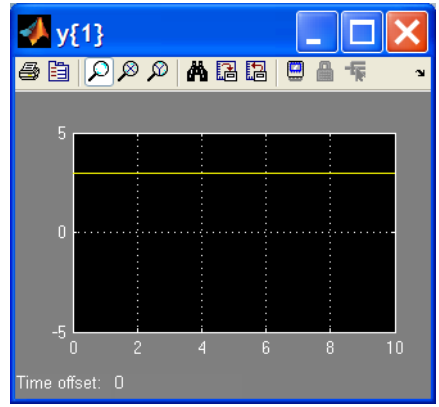
To test the network, double-click the Input 1 block at left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **Close**.

Select **Start** from the **Simulation** menu. Simulink momentarily pauses as it simulates the system.

When the simulation is over, double-click the scope at 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.

Exercises

Here are a couple of exercises you can try.

Changing Input Signal

Replace the constant input block with a signal generator from the standard Simulink blockset Sources. Simulate the system and view the network's response.

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.

C

Code Notes

Dimensions (p. C-2)
Variables (p. C-3)
Functions (p. C-6)
Code Efficiency (p. C-7)
Argument Checking (p. C-8)

Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

`Ni =`	Number of network inputs	`= net.numInputs`
`Ri =`	Number of elements in input `i`	`= net.inputs{i}.size`
`Nl =`	Number of layers	`= net.numLayers`
`Si =`	Number of neurons in layer `i`	`= net.layers{i}.size`
`Nt =`	Number of targets	
`Vi =`	Number of elements in target `i`, equal to `Sj`, where `j` is the `i`th layer with a target. (A layer `n` has a target if `net.targets(n) == 1`.)	
`No =`	Number of network outputs	
`Ui =`	Number of elements in output `i`, equal to `Sj`, where `j` is the `i`th layer with an output (A layer `n` has an output if `net.outputs(n) == 1`.)	
`ID =`	Number of input delays	`net.numInputDelays`
`LD =`	Number of layer delays	`net.numLayerDelays`
`TS =`	Number of time steps	
`Q =`	Number of concurrent vectors or sequences	

Variables

The variables a user commonly uses when defining a simulation or training session are

`P`	Network inputs	`Ni`-by-`TS` cell array, where each element `P{i,ts}` is an `Ri`-by-`Q` matrix
`Pi`	Initial input delay conditions	`Ni`-by-`ID` cell array, where each element `Pi{i,k}` is an `Ri`-by-`Q` matrix
`Ai`	Initial layer delay conditions	`Nl`-by-`LD` cell array, where each element `Ai{i,k}` is an `Si`-by-`Q` matrix
`T`	Network targets	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-by-`Q` matrix

These variables are returned by simulation and training calls:

`Y`	Network outputs	`No`-by-`TS` cell array, where each element `Y{i,ts}` is a `Ui`-by-`Q` matrix
`E`	Network errors	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-by-`Q` matrix
`perf`	Network performance	

Utility Function Variables

These variables are used only by the utility functions.

`Pc`	Combined inputs	`Ni`-by-`(ID+TS)` cell array, where each element `P{i,ts}` is an `Ri`-by-`Q` matrix `Pc = [Pi P]` = Initial input delay conditions and network inputs

`Pd`	Delayed inputs	`Ni`-by-`Nj`-by-`TS` cell array, where each element `Pd{i,j,ts}` is an `(Ri*IWD(i,j))`-by-`Q` matrix, and where `IWD(i,j)` is the number of delay taps associated with the input weight to layer `i` from input `j`
		
		Equivalently,
		
		`IWD(i,j) = length(net.inputWeights{i,j}.delays)`
		
		`Pd` is the result of passing the elements of `P` through each input weight's tap delay lines. Because inputs are always transformed by input delays in the same way, it saves time to do that operation only once instead of for every training step.

`BZ`	Concurrent bias vectors	`Nl`-by-1 cell array, where each element `BZ{i}` is an `Si`-by-`Q` matrix Each matrix is simply `Q` copies of the `net.b{i}` bias vector.
`IWZ`	Weighted inputs	`Ni`-by-`Nl`-by-`TS` cell array, where each element `IWZ{i,j,ts}` is an `Si`-by-`???`-by-`Q` matrix
`LWZ`	Weighted layer outputs	`Ni`-by-`Nl`-by-`TS` cell array, where each element `LWZ{i,j,ts}` is an `Si`-by-`Q` matrix
`N`	Net inputs	`Ni`-by-`TS` cell array, where each element `N{i,ts}` is an `Si`-by-`Q` matrix
A	Layer outputs	`Nl`-by-`TS` cell array, where each element `A{i,ts}` is an `Si`-by-`Q` matrix
Ac	Combined layer outputs	`Nl`-by-`(LD+TS)` cell array, where each element `A{i,ts}` is an `Si`-by-`Q` matrix `Ac = [Ai A]` = Initial layer delay conditions and layer outputs.

`Tl`	Layer targets	`Nl`-by-`TS` cell array, where each element `Tl{i,ts}` is an `Si`-by-`Q` matrix `Tl` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`El`	Layer errors	`Nl`-by-`TS` cell array, where each element `El{i,ts}` is a `Si`-by-`Q` matrix `El` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`X`	Column vector of all weight and bias values	

Functions

The following functions are the utility functions that you can call to perform a lot of the work of simulating or training a network. You can read about them in their respective help comments.

These functions calculate signals.

```
calcpd, calca, calca1, calce, calce1, calcperf
```

These functions calculate derivatives, Jacobians, and values associated with Jacobians.

```
calcgx, calcjx, calcjejj
```

`calcgx` is used for gradient algorithms; `calcjx` and `calcjejj` can be used for calculating approximations of the Hessian for algorithms like Levenberg-Marquardt.

These functions allow network weight and bias values to be accessed and altered in terms of a single vector X.

```
setx, getx, formx
```

Code Efficiency

The functions `sim`, `train`, and `adapt` all convert a network object to a structure,

```
net = struct(net);
```

before simulation and training, and then recast the structure back to a network.

```
net = class(net,'network')
```

This is done for speed efficiency since structure fields are accessed directly, while object fields are accessed using the MATLAB® object method handling system. If users write any code that uses utility functions outside of `sim`, `train`, or `adapt`, they should use the same technique.

Argument Checking

These functions are only recommended for advanced users.

None of the utility functions do any argument checking, which means that the only feedback you get from calling them with incorrectly sized arguments is an error.

The lack of argument checking allows these functions to run as fast as possible.

For “safer” simulation and training, use `sim`, `train`, and `adapt`.

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.

[DeSc83] Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[DHM01] De Jesús, O., J.M. Horn, and M.T. Hagan, "Analysis of Recurrent Network Training and Suggestions for Improvements," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2632–2637.

[Elma90] Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

This paper is a superb introduction to the Elman networks described in Chapter 10, “Recurrent Networks.”

[FeTs03] Feng, J., C.K. Tse, and F.C.M. Lau, “A neural-network-based channel-equalization strategy for chaos-based communication systems,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 50, No. 7, 2003, pp. 954–957.

[FlRe64] Fletcher, R., and C.M. Reeves, “Function minimization by conjugate gradients,” *Computer Journal*, Vol. 7, 1964, pp. 149–154.

[FoHa97] Foresee, F.D., and M.T. Hagan, “Gauss-Newton approximation to Bayesian regularization,” *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997, pp. 1930–1935.

[GiMu81] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

[GiPr02] Gianluca, P., D. Przybylski, B. Rost, P. Baldi, “Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles,” *Proteins: Structure, Function, and Genetics*, Vol. 47, No. 2, 2002, pp. 228–235.

[Gros82] Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg’s theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, “Neural Networks for Control,” *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642–1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, “Training Recurrent Networks for Filtering and Control,” Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

[HaMe94] Hagan, M.T., and M. Menhaj, “Training feed-forward networks with the Marquardt algorithm,” *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the

algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

[HaRu78] Harrison, D., and Rubinfeld, D.L., "Hedonic prices and the demand for clean air," *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

[HDB96] Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has demonstration programs, an instructor's guide, and transparency overheads for teaching.

[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.

[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.

[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.

[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 Chapter 9 of this 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.

[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.

[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 demonstrate 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. Hilton, 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.

Glossary

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

bias	Neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.
bias vector	Column vector of bias values for a layer of neurons.
Brent's search	Linear search that is a hybrid of the golden section search and a quadratic interpolation.
cascade-forward network	Layered network in which each layer only receives inputs from previous layers.
Charalambous' search	Hybrid line search that uses a cubic interpolation together with a type of sectioning.
classification	Association of an input vector with a particular target vector.
competitive layer	Layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector.
competitive learning	Unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.
competitive transfer function	Accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the winner, the neuron associated with the most positive element of the net input **n**.
concurrent input vectors	Name given to a matrix of input vectors that are to be presented to a network simultaneously. All the vectors in the matrix are used in making just one set of changes in the weights and biases.
conjugate gradient algorithm	In the conjugate gradient algorithms, a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.
connection	One-way link between neurons in a network.
connection strength	Strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.
cycle	Single presentation of an input vector, calculation of output, and new weights and biases.

dead neuron Competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.

decision boundary Line, determined by the weight and bias vectors, for which the net input n is zero.

delta rule See **Widrow-Hoff learning rule**.

delta vector The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.

distance Distance between neurons, calculated from their positions with a distance function.

distance function Particular way of calculating distance, such as the Euclidean distance between two vectors.

early stopping Technique based on dividing the data into three subsets. The first subset is the training set, used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design.

epoch Presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

error jumping Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

error ratio Training parameter used with adaptive learning rate and momentum training of backpropagation networks.

error vector Difference between a network's output vector in response to an input vector and an associated target output vector.

feedback network Network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.

feedforward network Layered network in which each layer only receives inputs from previous layers.

Fletcher-Reeves update	Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.
function approximation	Task performed by a network trained to respond to inputs with an approximation of a desired function.
generalization	Attribute of a network whose output for a new input vector tends to be close to outputs for similar input vectors in its training set.
generalized regression network	Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.
global minimum	Lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.
golden section search	Linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.
gradient descent	Process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.
hard-limit transfer function	Transfer function that maps inputs greater than or equal to 0 to 1, and all other values to 0.
Hebb learning rule	Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and postweight neurons.
hidden layer	Layer of a network that is not connected to the network output (for instance, the first layer of a two-layer feedforward network).
home neuron	Neuron at the center of a neighborhood.
hybrid bisection-cubic search	Line search that combines bisection and cubic interpolation.
initialization	Process of setting the network weights and biases to their original values.
input layer	Layer of neurons receiving inputs directly from outside the network.
input space	Range of all possible input vectors.
input vector	Vector presented to the network.

input weight vector	Row vector of weights going to a neuron.
input weights	Weights connecting network inputs to layers.
Jacobian matrix	Contains the first derivatives of the network errors with respect to the weights and biases.
Kohonen learning rule	Learning rule that trains a selected neuron's weight vectors to take on the values of the current input vector.
layer	Group of neurons having connections to the same inputs and sending outputs to the same destinations.
layer diagram	Network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias, and weight matrices are shown. Individual neurons and connections are not shown. (See Chapter 2, "Neuron Model and Network Architectures.")
layer weights	Weights connecting layers to other layers. Such weights need to have nonzero delays if they form a recurrent connection (i.e., a loop).
learning	Process by which weights and biases are adjusted to achieve some desired network behavior.
learning rate	Training parameter that controls the size of weight and bias changes during learning.
learning rule	Method of deriving the next changes that might be made in a network *or* a procedure for modifying the weights and biases of a network.
Levenberg-Marquardt	Algorithm that trains a neural network 10 to 100 times faster than the usual gradient descent backpropagation method. It always computes the approximate Hessian matrix, which has dimensions n-by-n.
line search function	Procedure for searching along a given search direction (line) to locate the minimum of the network performance.
linear transfer function	Transfer function that produces its input as its output.
link distance	Number of links, or steps, that must be taken to get to the neuron under consideration.

local minimum Minimum of a function over a limited range of input values. A local minimum might not be the global minimum.

log-sigmoid transfer function Squashing function of the form shown below that maps the input to the interval (0,1). (The toolbox function is `logsig`.)

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

Manhattan distance The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

maximum performance increase Maximum amount by which the performance is allowed to increase in one iteration of the variable learning rate training algorithm.

maximum step size Maximum step size allowed during a linear search. The magnitude of the weight vector is not allowed to increase by more than this maximum step size in one iteration of a training algorithm.

mean square error function Performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.

momentum Technique often used to make it less likely for a backpropagation network to get caught in a shallow minimum.

momentum constant Training parameter that controls how much momentum is used.

mu parameter Initial value for the scalar μ.

neighborhood Group of neurons within a specified distance of a particular neuron. The neighborhood is specified by the indices for all the neurons that lie within a radius d of the winning neuron i^*:

$$N_i(d) = \{j, d_{ij} \le d\}$$

net input vector Combination, in a layer, of all the layer's weighted input vectors with its bias.

neuron Basic processing element of a neural network. Includes weights and bias, a summing junction, and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

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

positive linear transfer function Transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

postprocessing Converts normalized outputs back into the same units that were used for the original targets.

Powell-Beale restarts Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient.

preprocessing Transformation of the input or target data before it is presented to the neural network.

principal component analysis Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.

quasi-Newton algorithm Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.

radial basis networks Neural network that can be designed directly by fitting special response elements where they will do the most good.

radial basis transfer function The transfer function for a radial basis neuron is

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

regularization Modification of the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights.

resilient backpropagation Training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid squashing transfer functions.

saturating linear transfer function Function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is `satlin`.)

scaled conjugate gradient algorithm Avoids the time-consuming line search of the standard conjugate gradient algorithm.

sequential input vectors Set of vectors that are to be presented to a network one after the other. The network weights and biases are adjusted on the presentation of each input vector.

sigma parameter Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.

sigmoid Monotonic S-shaped function that maps numbers in the interval (-∞,∞) to a finite interval such as (-1,+1) or (0,1).

simulation Takes the network input **p**, and the network object `net`, and returns the network outputs **a**.

spread constant Distance an input vector must be from a neuron's weight vector to produce an output of 0.5.

squashing function Monotonically increasing function that takes input values between -∞ and +∞ and returns values in a finite interval.

star learning rule Learning rule that trains a neuron's weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron's output.

sum-squared error Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

supervised learning Learning process in which changes in a network's weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.

symmetric hard-limit transfer function Transfer that maps inputs greater than or equal to 0 to +1, and all other values to -1.

symmetric saturating linear transfer function Produces the input as its output as long as the input is in the range -1 to 1. Outside that range the output is -1 and +1, respectively.

tan-sigmoid transfer function Squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is `tansig`.)

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

tapped delay line Sequential set of delays with outputs available at each delay output.

target vector	Desired output vector for a given input vector.
test vectors	Set of input vectors (not used directly in training) that is used to test the trained network.
topology functions	Ways to arrange the neurons in a grid, box, hexagonal, or random topology.
training	Procedure whereby a network is adjusted to do a particular job. Commonly viewed as an offline job, as opposed to an adjustment made during each time interval, as is done in adaptive training.
training vector	Input and/or target vector used to train a network.
transfer function	Function that maps a neuron's (or layer's) net output **n** to its actual output.
tuning phase	Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.
underdetermined system	System that has more variables than constraints.
unsupervised learning	Learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases.
update	Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.
validation vectors	Set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.
weight function	Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.
weight matrix	Matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix `W` refers to the connection strength from input `j` to neuron `i`.
weighted input vector	Result of applying a weight to a layer's input, whether it is a network input or the output of another layer.

Widrow-Hoff learning rule Learning rule used to train single-layer linear networks. This rule is the predecessor of the backpropagation rule and is sometimes referred to as the delta rule.

Index

A

ADALINE networks
  decision boundary 4-5
`adapt` function 15-11, 16-4
`adaptFcn`
  function property 14-7
adaptive filters
  example 10-10
  noise cancellation example 10-14
  prediction application 11-7
  prediction example 10-13
  training 2-20
adaptive linear networks 10-2
`adaptParam`
  parameter property 14-10
`addnoise` function 16-8
amplitude detection 11-11
applications
  adaptive filtering 10-9
  aerospace 1-4
  automotive 1-4
  banking 1-4
  defense 1-5
  electronics 1-5
  entertainment 1-5
  financial 1-5
  industrial 1-5
  insurance 1-5
  manufacturing 1-5
  medical 1-6
  oil and gas exploration 1-6
  robotics 1-6
  securities 1-6
  speech 1-6
  telecommunications 1-6
  transportation 1-6
architecture
  bias connection 12-4
  input connection 12-5
  layer connection 12-5
  number of inputs 12-4
  number of layers 12-4
  number of outputs 12-6
  number of targets 12-6
  output connection 12-6
  target connection 12-6
architecture properties 14-2

B

`b`
  bias vector property 14-12
backpropagation
  algorithm 5-15
  definition 5-2
  example 5-68
  resilient 5-21
backtracking search 5-28
batch algorithm 9-9
batch training
  compared 2-20
  definition 2-22
  dynamic networks 2-24
  static networks 2-22
batch training algorithm 9-29
Bayesian framework 5-56
benchmark data sets 5-59
BFGS quasi-Newton algorithm 5-29

biasConnect
  architecture property 14-3
biases
  connection 12-4
  definition 2-2
  subobject 12-10
  subobject and network object 14-22
  value 12-12
biases
  subobject property 14-6
box distance 9-16
boxdist function 16-9
Brent's search 5-27

C

calcgx function 16-10
calcjejj function 16-12
calcjx function 16-15
calcpd function 16-17
calcperf function 16-19
cell arrays
  bias vectors 12-12
  input P 2-18
  input vectors 12-13
  inputs 2-22
  inputs property 12-6
  layers property 12-8
  matrix of concurrent vectors 2-18
  matrix of sequential vectors 2-21
  sequence of outputs 2-17
  sequential inputs 2-16
  targets 2-22
  weight matrices 12-12
Charalambous' search 5-28
classification
  input vectors 3-3
  linear 4-15
  regions 3-4
  using probabilistic neural networks 8-9
clustering 1-42
clustering problems
  defining 1-42
  solving with command-line functions 1-43
  solving with nctool 1-47
combvec function 16-21
command-line functions
  solving clustering problems 1-43
  solving fitting problems 1-7
  solving pattern recognition problems 1-25
compet function 16-22
competitive layers 9-3
competitive neural networks
  creating 9-4
  example 9-8
competitive transfer functions 9-3
con2seq function 16-26
concur function 16-27
concurrent inputs
  compared 2-14
confusion function 16-24
confusion matrix 1-28
conjugate gradient algorithms
  definition 5-22
  Fletcher-Reeves update 5-23
  Polak-Ribiére update 5-24
  Powell-Beale restarts 5-25
  scaled 5-25
continuous stirred tank reactor example 7-7
control
  control design 7-2
  electromagnet 7-20
  feedback linearization 7-16
  feedback linearization (NARMA-L2) 7-3

model predictive 7-3
model predictive control 7-6
model reference 7-3
NARMA-L2 7-16
plant 7-25
plant for predictive control 7-2
robot arm 7-27
time horizon 7-6
training data 7-12
controller
NARMA-L2 controller 7-18
convwf function 16-28
CSTR 7-7
custom neural networks 12-2

D

dead neurons 9-5
decision boundary 4-5
definition 3-4
delays
input weight property 14-23
layer weight property 14-25
demonstrations
appelm1 11-11
applin3 11-10
demohop1 13-14
demohop2 13-14
demorb4 8-8
nnd10lc 4-16
nnd11gn 5-52
nnd12cg 5-24
nnd12m 5-32
nnd12mo 5-18
nnd12sd1 5-27
nnd12sd1 batch gradient 5-17
nnd12vl 5-20
dimensions
layer property 14-15
disp function 16-30
display function 15-11, 16-31
dist function 16-32
distance 9-9
box 9-16
Euclidean 9-14
link 9-16
Manhattan 9-16
tuning phase 9-18
distance functions 9-14
distanceFcn
layer property 14-15
distances
layer property 14-16
distributed time delay neural network 6-15
divideblock function 16-34
divideind function 16-35
divideint function 16-36
dividerand function 16-37
dotprod function 16-38
dynamic networks
applications 6-7
compared with static networks 6-2
concurrent inputs 2-17
nonrecurrent 6-4
recurrent 6-6
sequential inputs 2-15
training
batch 2-24
incremental 2-22

E

early stopping
improving generalization 5-53

electromagnet example 7-20
Elman networks 13-1
  recurrent connection 13-3
errsurf function 16-40
Euclidean distance 9-14
examples
  continuous stirred tank reactor 7-7
  electromagnet 7-20
  robot arm 7-27
exporting networks 7-33
exporting training data 7-37

F

feedback linearization
  companion form model 7-16
  See also NARMA-L2
feedforward networks 5-10
finite impulse response filters
  example 4-11
FIR 6-7
fit plot 1-20
fitting functions 1-7
fitting problems
  defining 1-7
  solving with command-line functions 1-7
  solving with nftool 1-13
fixunknowns function 16-41
Fletcher-Reeves update 5-23
focused time-delay network (FTDNN) 6-11
FTDNN
  See focused time-delay network 6-11
functions
  fitting 1-7

G

generalization
  improving 5-52
  regularization 5-55
generalized regression networks 8-12
gensim function 16-43
golden section search 5-26
gradient descent algorithm
  batch 5-16
  modes 5-15
gradientFcn
  function property 14-7
gradientParam
  parameter property 14-10
graphical user interface
  introduction 3-25
gridtop function 16-45
gridtop topology 9-10

H

hard limit transfer function 2-3
  hardlim 3-3
hardlim function 16-46
hardlims function 16-48
heuristic techniques 5-19
hextop function 16-50
hextop topology 9-12
hidden layers
  definition 2-11
hintonw function 16-51
hintonwb function 16-53
home neuron 9-15
Hopfield networks
  architecture 13-8
  design equilibrium point 13-10
  solution trajectories 13-14

spurious equilibrium points 13-10
stable equilibrium point 13-10
target equilibrium points 13-10
horizon 7-6
hybrid bisection cubic search 5-28

I

IIR 6-7
importing networks 7-33
importing training data 7-37
incremental training 2-20
static networks 2-20
ind2vec function 16-55
init function 16-56
initcon function 16-58
initFcn
bias property 14-22
function property 14-8
input weight property 14-23
layer property 14-16
layer weight property 14-25
initial step size function 5-21
initialization
definition 3-10
initlay function 16-59
initnw function 16-60
initParam
parameter property 14-10
initsompc function 16-62
initwb function 16-63
initzero function 16-64
input vectors
classification 3-3
dimension reduction 5-64
distance 9-9
outlier 3-23
topology 9-9
input weights
definition 2-10
subobject 14-23
inputConnect
architecture property 14-3
inputs
concurrent 2-14
connection 12-5
number 12-4
sequential 2-14
subobject 12-6
inputs
input property 14-13
subobject property 14-5
inputWeights
subobject property 14-6
IW
weight property 14-11

J

Jacobian matrix 5-31

K

Kohonen learning rule 9-5

L

lambda parameter 5-26
layer weights
definition 2-10
subobject 14-25
layerConnect
architecture property 14-3
layered digital dynamic network (LDDN) 6-8

layer-recurrent network (LRN) 6-24
layers
  connection 12-5
  number 12-4
  subobject 12-8
layers
  subobject property 14-5
layers property 14-15
layerWeights
  subobject property 14-6
LDDN
  See layered digital dynamic network 6-8
learn
  bias property 14-22
  input weight property 14-23
  layer weight property 14-25
learncon function 16-65
learnFcn
  bias property 14-22
  input weight property 14-23
  layer weight property 14-25
learngd function 16-68
learngdm function 16-70
learnh function 16-73
learnhd function 16-76
learning rates
  adaptive 5-20
  maximum stable 4-14
  optimal 5-19
  ordering phase 9-18
  too large 4-19
  tuning phase 9-18
learning rules
  introduction 3-2
  Kohonen 9-5
  LMS
    See also Widrow-Hoff learning rule 10-2
  LVQ1 9-39
  LVQ2.1 9-42
  perceptron 3-2
  supervised learning 3-13
  unsupervised learning 3-13
  Widrow-Hoff 4-13
learning vector quantization
  creation 9-36
  learning rule 9-42
    LVQ1 9-39
  LVQ network 9-35
  subclasses 9-35
  supervised training 9-2
  target classes 9-35
  union of two subclasses 9-39
learnis function 16-79
learnk function 16-82
learnlv1 function 16-85
learnlv2 function 16-88
learnos function 16-91
learnp function 16-94
learnParam
  bias property 14-22
  input weight property 14-24
  layer weight property 14-25
learnpn function 16-97
learnsom function 16-100
learnwh function 16-106
least mean square error learning rule 10-7
Levenberg-Marquardt algorithm 5-30
  reduced memory 5-32
line search functions
  backtracking search 5-28
  Brent's search 5-27
  Charalambous' search 5-28
  golden section search 5-26
  hybrid bisection cubic search 5-28

linear feedforward-dynamic networks
FIR 6-7
linear networks
design 4-9
linear recurrent-dynamic networks
IIR 6-7
linear transfer functions 4-3
linearly dependent vectors 4-19
link distance 9-16
linkdist function 16-109
logsig function 16-110
log-sigmoid transfer function
logsig 5-8
log-sigmoid transfer functions 2-4
LRN
See layer-recurrent network 6-24
LVQ networks 9-35
LW
weight property 14-11

M

MADALINE networks 10-4
mae function 16-112
magnet 7-20
mandist function 16-114
Manhattan distance 9-16
mapminmax function 15-15, 16-116
mapstd function 16-118
maximum step size function 5-21
maxlinlr function 16-120
mean square error function 5-15
least 10-7
memory reduction 5-32
midpoint function 16-121
minmax function 16-122
model predictive control 7-6
model reference control 7-2
Model Reference Control block 7-27
mse function 16-123
msereg function 16-125
mseregec function 16-127
mu parameter 5-31

N

NARMA 7-2
NARMA-L2 control 7-16
NARMA-L2 controller 7-18
NARMA-L2 Controller block 7-20
NARX
See nonlinear autoregressive network with exogenous inputs 6-18
nctool
solving clustering problems 1-47
nctool function 16-129
negdist function 16-130
neighbor distances plot 1-45, 9-31
neighborhood 9-9
netInputFcn
layer property 14-16
netInputParam
layer property 14-16
netinv function 16-132
netprod function 16-133
netsum function 16-135
network function 16-137
network functions 12-11
network layers
competitive 9-3
definition 2-6
Network/Data Manager window 3-25
networks
definition 12-3

dynamic
concurrent inputs 2-17
sequential inputs 2-15
static 2-14
Neural Network Toolbox Clustering Tool
See nctool.
Neural Network Toolbox Fitting Tool. See nftool.
Neural Network Toolbox Pattern Recognition Tool. See nprtool.
neural networks
adaptive linear 10-2
competitive 9-4
custom 12-2
definition 1-2
feedforward 5-10
generalized regression 8-12
one-layer 2-8
figure 4-4
probabilistic 8-9
radial basis 8-2
self-organizing 9-2
self-organizing feature map 9-9
neurons
dead (not allocated) 9-5
definition 2-2
home 9-15
See also distance, topologies
newc function 16-142
newcf function 16-144
newdtdnn function 16-147
compared to newfftd 6-16
newelm function 16-150
newff function 16-153
newfftd function 16-156
newfit function 16-159
newgrnn function 16-161
newhop function 16-163
newlin function 16-165
newlind function 16-167
newlrn function 16-169
newlvq function 16-171
newnarx function 16-173
newnarxsp function 16-176
newp function 16-179
newpnn function 16-181
newpr function 16-183
newrb function 16-185
newrbe function 16-187
newsom function 16-189
Newton's method 5-31
nftool
solving fitting problems 1-13
nftool function 16-191
NN Predictive Control block 7-7
nnt2c function 16-192
nnt2elm function 16-193
nnt2ff function 16-194
nnt2hop function 16-195
nnt2lin function 16-196
nnt2lvq function 16-197
nnt2p function 16-198
nnt2rb function 16-199
nnt2som function 16-200
nntool function 15-5, 16-201
nntraintool function 16-202
nonlinear autoregressive network with exogenous inputs (NARX) 6-18
normalization
inputs and targets 5-62
mean and standard deviation 5-63
normc function 16-203
normprod function 16-204
normr function 16-206

notation
  abbreviated 2-6
  layer 2-10
  transfer function symbols 2-4
nprtool
  solving pattern recognition problems 1-31
nprtool function 16-207
numerical optimization 5-19
numInputDelays
  architecture property 14-4
numInputs
  architecture property 14-2
numLayerDelays
  architecture property 14-5
numLayers
  architecture property 14-2
numOutputs
  architecture property 14-4

O

one step secant algorithm 5-30
one-step-ahead prediction 6-13
ordering phase learning rate 9-18
outlier input vectors 3-23
output layers
  definition 2-11
  linear 5-10
outputConnect
  architecture property 14-4
outputs
  connection 12-6
  number 12-6
  subobject 12-9
  subobject properties 14-20
outputs
  subobject property 14-6
overdetermined systems 4-18
overfitting 5-52

P

parallel NARX network 6-23
parameter properties 14-10
pass
  definition 3-17
pattern recognition 1-24, 11-15
pattern recognition problems
  defining 1-24
  solving with command-line functions 1-25
perceptron learning rule 3-2
  learnp 3-14
  normalized 3-24
perceptron network
  limitations 3-23
perceptron networks
  creation 3-2
  introduction 3-2
performance functions
  modifying 5-55
performFcn
  function property 14-8
performParam
  parameter property 14-10
plant 7-25
plant identification 7-25
  NARMA-L2 model 7-16
Plant Identification window 7-11
plant model 7-2
  in model predictive control 7-3
plotbr function 16-208
plotconfusion function 16-209
plotep function 16-210
plotes function 16-211

plotpc function 16-213
plotperf function 16-214
plotperform function 16-215
plotpv function 16-216
plotroc function 16-218
plotsom function 16-219
plotsomhits function 16-220
plotsomnc function 16-221
plotsomnd function 16-221, 16-222
plotsomplanes function 16-223
plotsompos function 16-224
plotsomtop function 16-225
plottrainstate function 16-226
plotv function 16-227
plotvec function 16-228
pnormc function 16-229
Polak-Ribiére update 5-24
positions
  layer property 14-16
poslin function 16-230
postreg function 16-232
posttraining analysis 5-66
Powell-Beale restarts 5-25
predictive control 7-6
preprocessing 5-61
principal component analysis 5-64
probabilistic neural networks 8-9
  design 8-10
processpca function 16-234
properties that determine algorithms 14-7
purelin function 16-236

Q

quant function 16-238
quasi-Newton algorithm 5-28
  BFGS 5-29

R

radbas function 16-239
radial basis
  design 8-14
  efficient network 8-7
  function 8-2
  networks 8-2
radial basis transfer function 8-4
randnc function 16-241
randnr function 16-242
rands function 16-243
randtop function 16-244
randtop topology 9-13
recurrent connections 13-3
recurrent networks 13-2
regression plots 1-20
regularization 5-55
  automated 5-56
removeconstantrows function 16-245
removerows function 16-247
resilient backpropagation 5-21
revert function 16-249
robot arm example 7-27
roc curve 1-39

S

sample hits plot 1-53, 9-32
satlin function 16-252
satlins function 16-254
scalprod function 16-256
self-organizing feature map (SOFM) networks 9-9
  batch algorithm 9-9
  neighbor distances plot 1-45, 9-31
  neighborhood 9-9
  one-dimensional example 9-22
  sample hits plot 1-53, 9-32

SOM topology 1-44
two-dimensional example 9-25
weight planes plot 1-53, 9-33
weight positions plot 9-30
self-organizing networks 9-2
seq2con function 16-258
sequential inputs 2-14
series-parallel network 6-20
setx function 16-259
S-function 15-3
sigma parameter 5-26
sim function 16-260
simulation 5-14
definition 3-8
Simulink
generating networks C-5
NNT blockset code D-2
NNT blockset simulation C-2
size
bias property 14-22
bias vector property 14-22
input property 14-15
input weight property 14-24
layer property 14-17
layer weight property 14-25
output property 14-21
soft max transfer function 16-265
softmax function 16-265
SOM topology 1-44
sp2narx function 16-267
spread constant 8-5
squashing functions 5-21
srchbac function 16-268
srchbre function 16-272
srchcha function 16-276
srchgol function 16-280
srchhyb function 16-284
sse function 16-288
static networks
batch training 2-22
concurrent inputs 2-14
defined 2-14
incremental training 2-20
subobject properties 14-13
network definition 12-6
subobject structure properties 14-5
subobjects
bias code 12-10
bias definition 14-22
input 12-6
input weight properties 14-23
layer 12-8
layer weight properties 14-25
output code 12-9
output definition 14-20
target code 12-9
weight code 12-10
weight definition 14-22
supervised learning 3-13
target output 3-13
training set 3-13
symbols
transfer function representation 2-4
system identification 7-5

T

tansig function 16-290
tan-sigmoid transfer function 5-9
tapped delay lines 4-10
target outputs 3-13
targets
connection 12-6
number 12-6

subobject 12-9
time horizon 7-6
topologies
self-organizing feature map 9-9
topologies for SOFM neuron locations
gridtop 9-10
hextop 9-12
randtop 9-13
topologyFcn
layer property 14-18
train function 16-292
trainb function 16-296
trainbfg function 16-299
trainbfgc function 16-303
trainbr function 16-306
trainc function 16-313
traincgb function 16-316
traincgf function 16-321
traincgp function 16-326
trainFcn
function property 14-9
traingd function 16-330
traingda function 16-333
traingdm function 16-336
traingdx function 16-339
training
batch 2-20
competitive networks 9-6
definition 2-2
efficient 5-61
faster 5-19
heuristic techniques 5-19
incremental 2-20
numerical optimization 5-19
ordering phase 9-20
perceptron 3-2
posttraining analysis 5-66
self-organizing feature map 9-19
styles 2-20
tuning phase 9-20
training data 7-12
training set 3-13
training styles 2-20
training with noise 11-18
trainlm function 16-342
trainoss function 16-346
trainParam
parameter property 14-11
trainr function 15-18, 16-350
trainrp function 16-353
trains function 16-357
trainscg function 16-360
transfer functions
competitive 9-3
definition 2-2
derivatives 5-9
hard limit 2-3
hard limit in perceptron 3-3
linear 4-3
log-sigmoid 2-4
log-sigmoid in backpropagation 5-8
radial basis 8-4
tan-sigmoid 5-9
transferFcn
layer property 14-20
transferParam
layer property 14-20
transformation matrix 5-64
tribas function 16-363
tuning phase learning rate 9-18
tuning phase neighborhood distance 9-18

U

underdetermined systems 4-18
unsupervised learning 3-13
`userdata` property 14-12

V

variable learning rate algorithm 5-20
`vec2ind` function 16-365
vectors
  linearly dependent 4-19
`view` function 16-366

W

weight and bias value properties 14-11
weight matrix
  definition 2-8
weight planes plot 1-53, 9-33
weight positions plot 9-30
`weightFcn`
  input weight property 14-24
  layer weight property 14-26
`weightParam`
  input weight property 14-24
  layer weight property 14-26
weights
  definition 2-2
  subobject code 12-10
  subobject definition 14-22
  value 12-12
Widrow-Hoff learning rule 4-13
  adaptive networks 10-8
  and mean square error 10-2
  generalization 5-2
workspace (command line) 3-25