

NEURAL NETWORKS. Applications and examples using MATLAB

J. Smith

NEURAL NETWORKS. APPLICATIONS AND EXAMPLES USING MATLAB

J. SMITH

CONTENTS

**FITTING NEURAL NETWORKS
WITH MATLAB. EXAMPLES**

1.1 A COMPLETE EXAMPLE: HOUSE PRICE ESTIMATION

1.1.1 The Problem: Estimate House Values

1.1.2 Why Neural Networks?

1.1.3 Preparing the Data

1.1.4 Fitting a Function with a Neural Network

1.1.5 Testing the Neural Network

1.2 FUNCTION FITTING NEURAL NETWORK. EXAMPLES

1.2.1 Construct and Train a Function Fitting Network

1.2.2 Create and train Feedforward Neural Network

1.2.3 Create and Train a Cascade

Network

1.3 NETWORK PERFORMANCE

1.3.1 Description

1.3.2 Examples

1.4 FIT REGRESSION MODEL AND PLOT FITTED VALUES VERSUS TARGETS. EXAMPLES

1.4.1 Description

1.4.2 Examples

1.5 PLOT OUTPUT AND TARGET VALUES. EXAMPLES

1.5.1 Description

1.5.2 Examples

1.6 PLOT TRAINING STATE VALUES. EXAMPLES

1.7 PLOT PERFORMANCES.

EXAMPLES

1.8 PLOT HISTOGRAM OF ERROR VALUES. EXAMPLES

1.8.1 Syntax

1.8.2 Description

1.8.3 Examples

1.9 GENERATE MATLAB FUNCTION FOR SIMULATING NEURAL NETWORK. EXAMPLES

1.9.1 Create Functions from Static Neural Network

1.9.2 Create Functions from Dynamic Neural Network

CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK. EXAMPLES

2.5.1 One-Dimensional Self-Organizing Map

2.5.2 Two-Dimensional Self-Organizing Map

2.5.3 Training with the Batch Algorithm

2.6 SELFORGMAP

2.7 FUNCTIONS FOR SELF-ORGANIZNG MAPS AND EXAMPLES

2.7.1 plotsomhits

2.7.2 plotsomnc

2.7.3 plotsomnd

2.7.4 plotsomplanes

2.7.5 plotsompos

2.7.6 plotsomtop

2.8 A COMPLETE EXAMPLE. IRIS CLUSTERING

2.8.1 Why Self-Organizing Map Neural Networks?

2.8.2 Preparing the Data

2.8.3 Clustering with a Neural Network

2.9 GENE EXPRESSION ANALYSIS. CLUSTER ANALYSIS AND PRINCIPAL COMPONENTS

2.9.1 The Problem: Analyzing Gene Expressions in Baker's Yeast (*Saccharomyces Cerevisiae*)

2.9.2 The Data

2.9.3 Filtering the Genes

2.9.4 Principal Component Analysis

2.9.5 Cluster Analysis using principal components: Self-Organizing Maps

2.10 COMPETITIVE LEARNING

2.11 ONE-DIMENSIONAL SELF-ORGANIZING MAP

2.12 TWO-DIMENSIONAL SELF-ORGANIZING MAP

2.13 CREATE A COMPETITIVE NEURAL NETWORK. BIAS AND KOHONEN LEARNING RULE

2.13.1 Kohonen Learning Rule (learnk)

2.13.2 Bias Learning Rule (learncon)

2.13.3 Training

2.13.4 Graphical Example

2.14 COMPETITIVE LAYERS FUNCTIONS

2.14.1 competlayer

2.14.2 view

2.14.3 trainru

2.14.4 learnk

2.14.5 learncon

PATTERN RECOGNITION AND CLASSIFICATION WITH NEURAL NETWORKS. DEEP LEARNING. EXAMPLES

3.1 INTRODUCTION

3.2 FUNCTIONS FOR PATTENWRN
RECOGNITION AND
CLASSIFICATION. EXAMPLES

3.3 VIEW NEURAL NETWORK

3.4 PATTERN RECOGNITION AND LEARNING VECTOR QUANTIZATION

3.4.1 Pattern recognition network: patternnet

3.4.2 Learning vector quantization neural network: lvqnet

3.5 TRAINING OPTIONS AND NETWORK PERFORMANCE

3.5.1 Receiver operating characteristic: roc

3.5.2 Plot receiver operating characteristic: plotroc

3.5.3 Plot classification confusion matrix: plotconfusion

3.5.4 Neural network performance:

crossentropy

3.6 AUTOENCODER CLASS. DEEP LEARNING

3.6.1 trainAutoencoder

3.6.2 Construct Deep Network Using Autoencoders

3.6.3 decode

3.6.4 encode

3.6.5 predict

3.6.6 stack

3.7 TRAIN STACKED AUTOENCODERS FOR IMAGE CLASSIFICATION. DEEP NEURAL NETWORK

3.7.1 Data set

3.7.2 Training the first autoencoder

3.7.3 Visualizing the weights of the first autoencoder

3.7.4 Training the second autoencoder

3.7.5 Training the final softmax layer

3.7.6 Forming a stacked neural network

3.7.7 Fine tuning the deep neural network

3.7.8 Summary

3.9 TRANSFER LEARNING USING CONVOLUTIONAL NEURAL NETWORKS

3.10 CRAB CLASSIFICATION

3.10.1 Why Neural Networks?

3.10.2 Preparing the Data

3.10.3 Building the Neural Network Classifier

3.10.4 Testing the Classifier

3.11 WINE CLASSIFICATION. PATTERN RECOGNITION

3.11.1 The Problem: Classify Wines

3.11.2 Why Neural Networks?

3.11.3 Preparing the Data

3.11.4 Pattern Recognition with a Neural Network

3.11.5 Testing the Neural Network

3.12 CANCER DETECTION

3.12.1 Formatting the Data

3.12.2 Ranking Key Features

3.12.3 Classification Using a Feed Forward Neural Network

3.13 CHARACTER RECOGNITION

3.13.1 Creating the First Neural Network

3.13.2 Training the first Neural Network

3.13.3 Training the Second Neural Network

3.13.4 Testing Both Neural Networks

3.14 LEARNING VECTOR QUANTIZATION (LVQ). EXAMPLE

ADAPTATIVE LINEAR FILTERS

4.1 ADAPTATIVE FILTERS

4.2 PATTERN ASSOCIATION SHOWING ERROR SURFACE

4.3 TRAINING A LINEAR NEURON

4.4 ADAPTIVE NOISE CANCELLATION

4.5 LINEAR FIT OF NONLINEAR PROBLEM

4.6 UNDERDETERMINED PROBLEM

4.7 LINEARLY DEPENDENT PROBLEM

4.8 TOO LARGE A LEARNING RATE

RADIAL BASIS NETWORKS

5.1 RADIAL BASIS FUNCTION NETWORK

5.2 RADIAL BASIS APPROXIMATION

5.3 RADIAL BASIS

UNDERLAPPING NEURONS

5.4 GRNN FUNCTION

APPROXIMATION

5.5 PNN CLASSIFICATION

SIMPLE APPLICATIONS AND HOLFIELD NETWORKS

6.1 LINEAR PREDICTION DESIGN

6.1.1 Defining a Wave Form

6.1.2 Setting up the Problem for a Neural Network

6.1.3 Designing the Linear Layer

6.1.4 Testing the Linear Layer

6.2 ADAPTIVE LINEAR PREDICTION

6.2.1 Defining a Wave Form

6.2.2 Setting up the Problem for a Neural Network

6.2.3 Creating the Linear Layer

6.2.4 Adapting the Linear Layer

6.3 HOPFIELD NETWORK

6.4 HOPFIELD TWO NEURON DESIGN

6.5 HOPFIELD UNSTABLE EQUILIBRIA

6.6 HOPFIELD THREE NEURON DESIGN

6.7 HOPFIELD SPURIOUS STABLE POINTS

PERCEPTRONS

7.1 PERCEPTRON

7.2 CLASSIFICATION WITH A 2-INPUT PERCEPTRON

7.3 OUTLIER INPUT VECTORS

7.4 NORMALIZED PERCEPTRON RULE

7.5 LINEARLY NON-SEPARABLE VECTORS

TIME SERIES NEURAL NETWORKS. EXAMPLES

8.1 FUNCTIONS FOR MODELING AND PREDICTION

8.2 TIMEDELAYNET

8.3 NARXNET

8.4 NARNET

8.5 LAYRECNET

8.6 DISTDELAYNET

8.7 TRAIN

8.8 USING COMMAND-LINE FUNCTIONS

8.9

8.10 A COMPLTE EXAMPLE. MAGLEV MODELING

8.10.1 The Problem: Model a Magnetic Levitation System

8.10.2 Why Neural Networks?

8.10.3 Preparing the Data

8.10.4 Time Series Modelling with a Neural Network

8.10.5 Testing the Neural Network

FIT DATA WITH A NEURAL

NETWORK GRAPHICAL INTERFACE

9.1 INTRODUCTION

9.2 USING THE NEURAL NETWORK FITTING TOOL

9.3 USING COMMAND-LINE FUNCTIONS

CLASSIFY PATTERNS WITH A NEURAL NETWORK GRAPHICAL INTERFACE

10.1 INTRODUCTION

10.2 USING THE NEURAL NETWORK PATTERN RECOGNITION TOOL

10.3 USING COMMAND-LINE FUNCTIONS

CLUSTER DATA WITH A SELF-ORGANIZING MAP. GRAPHICAL INTERFACE

11.1 INTRODUCTION

11.2 USING THE NEURAL NETWORK CLUSTERING TOOL

11.3 USING COMMAND-LINE FUNCTIONS

NEURAL NETWORK TIME-SERIES PREDICTION AND MODELING. GRAPHICAL INTERFACE

12.1 INTRODUCTION

12.2 USING THE NEURAL NETWORK TIME SERIES TOOL

12.3 USING COMMAND-LINE FUNCTIONS

BIBLIOGRAPHY

13.1 NEURAL NETWORK

BIBLIOGRAPHY

Chapter 1

FITTING NEURAL NETWORKS WITH MATLAB. EXAMPLES

1.1 A COMPLETE EXAMPLE: HOUSE PRICE ESTIMATION

This example illustrates how a function fitting neural network can estimate median house prices for a neighborhood based on neighborhood demographics.

1.1.1 The Problem: Estimate House Values

In this example we attempt to build a neural network that can estimate the median price of a home in a neighborhood described by thirteen demographic attributes:

- Per capita crime rate per town
- Proportion of residential land zoned for lots over 25,000 sq. ft.
- Proportion of non-retail business acres per town
- 1 if tract bounds Charles river, 0 otherwise
- Nitric oxides concentration

(parts per 10 million)

- Average number of rooms per dwelling
- Proportion of owner-occupied units built prior to 1940
- Weighted distances to five Boston employment centres
- Index of accessibility to radial highways
- Full-value property-tax rate per \$10,000
- Pupil-teacher ratio by town
- $1000(Bk - 0.63)^2$
- Percent lower status of the population

This is an example of a fitting problem,

where inputs are matched up to associated target outputs, and we would like to create a neural network which not only estimates the known targets given known inputs, but can generalize to accurately estimate outputs for inputs that were not used to design the solution.

1.1.2 Why Neural Networks?

Neural networks are very good at function fit problems. A neural network with enough elements (called neurons) can fit any data with arbitrary accuracy. They are particularly well suited for addressing non-linear problems. Given the non-linear nature of real world phenomena, like house valuation, neural networks are a good candidate for solving the problem.

The thirteen neighborhood attributes will act as inputs to a neural network, and the median home price will be the target.

The network will be designed by using the attributes of neighborhoods whose median house value is already known to train it to produce the target valuations.

1.1.3 Preparing the Data

Data for function fitting problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have thirteen elements representing a neighborhood whose median house value is already known.

Each corresponding column of the target matrix will have one element, representing the median house price in 1000's of dollars.

Here such a dataset is loaded.

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

We can view the sizes of inputs X and targets T.

Note that both X and T have 506 columns. These represent 506 neighborhood attributes (inputs) and associated median house values (targets).

Input matrix X has thirteen rows, for the thirteen attributes. Target matrix T has only one row, as for each example we only have one desired output, the median house value.

size(x)

size(t)

ans =

13 506

ans =

1 506

1.1.4 Fitting a Function with a Neural Network

The next step is to create a neural network that will learn to estimate median house values.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

setdemorandstream(49)

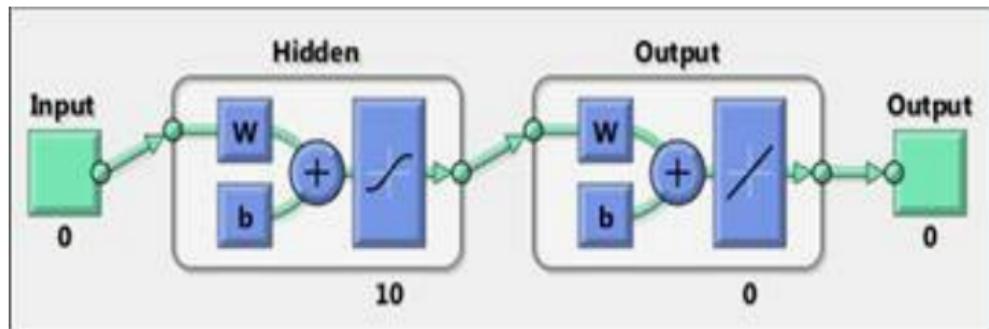
Two-layer (i.e. one-hidden-layer) feed forward neural networks can fit any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target

data. This will happen when the network is trained.

```
net = fitnet(10);  
  
view(net)
```



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets.

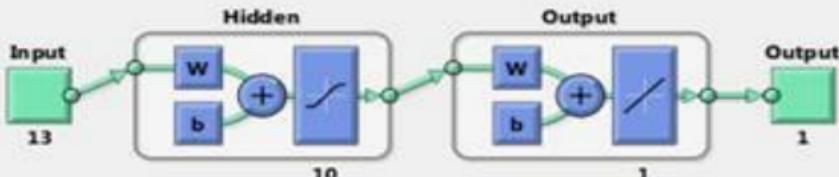
The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

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

Neural Network



Algorithms

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

Progress

Epoch:	0	16 iterations	1000
Time:		0:00:00	
Performance:	2.14e+03	8.69	0.00
Gradient:	5.99e+03	9.30	1.00e-07
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

Plots



Validation stop.

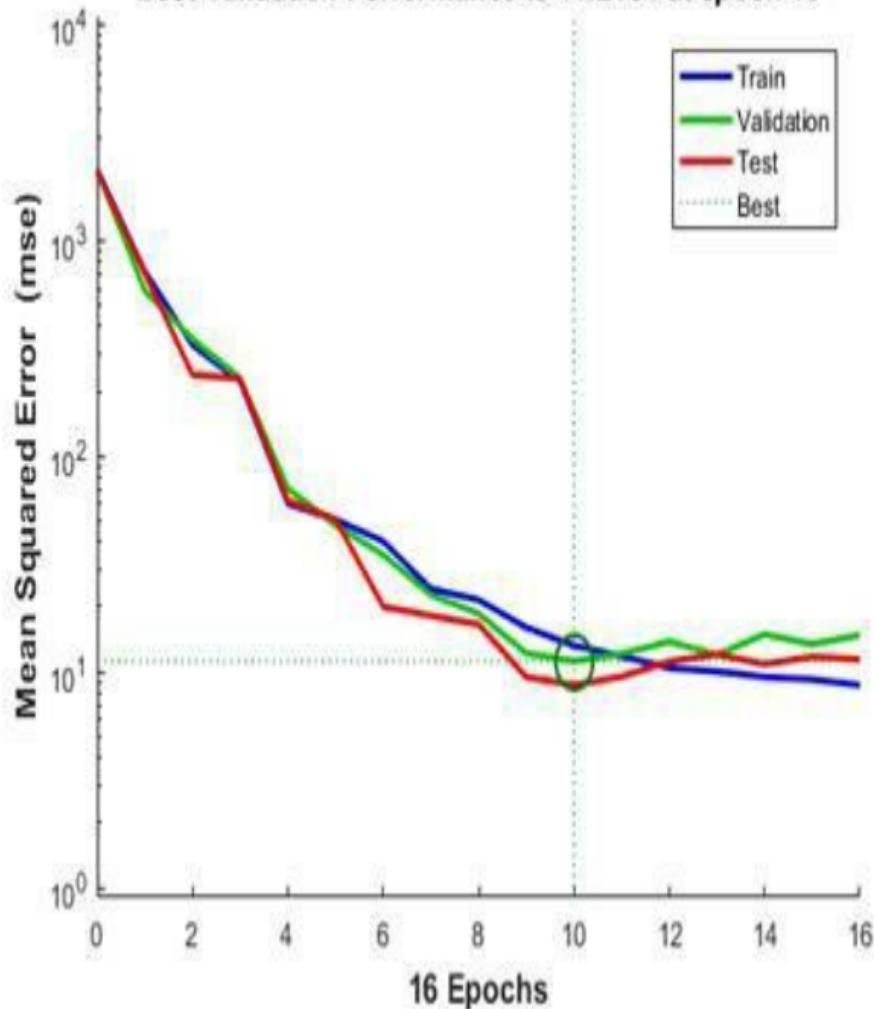
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

Best Validation Performance is 11.2164 at epoch 10



1 .1 .5 Testing the Neural Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

```
testX =  
x(:, tr.testInd);
```

```
testT =
```

```
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
perf =
```

```
mse(net,testT,testY)
```

```
perf =
```

8 . 6959

Another measure of how well the neural network has fit the data is the regression plot. Here the regression is plotted across all samples.

The regression plot shows the actual network outputs plotted in terms of the associated target values. If the network has learned to fit the data well, the linear fit to this output-target relationship should closely intersect the bottom-left

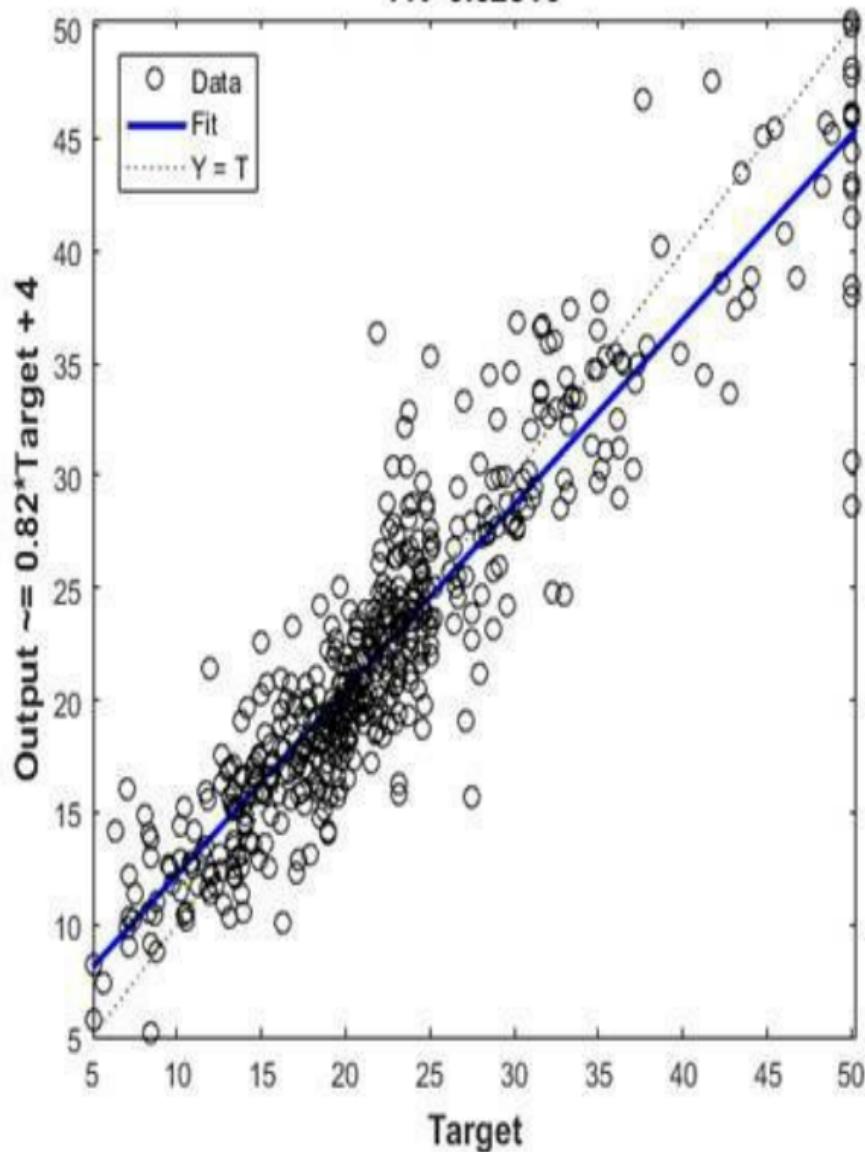
and top-right corners of the plot.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

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

```
plotregression(t,y)
```

: R=0.92516

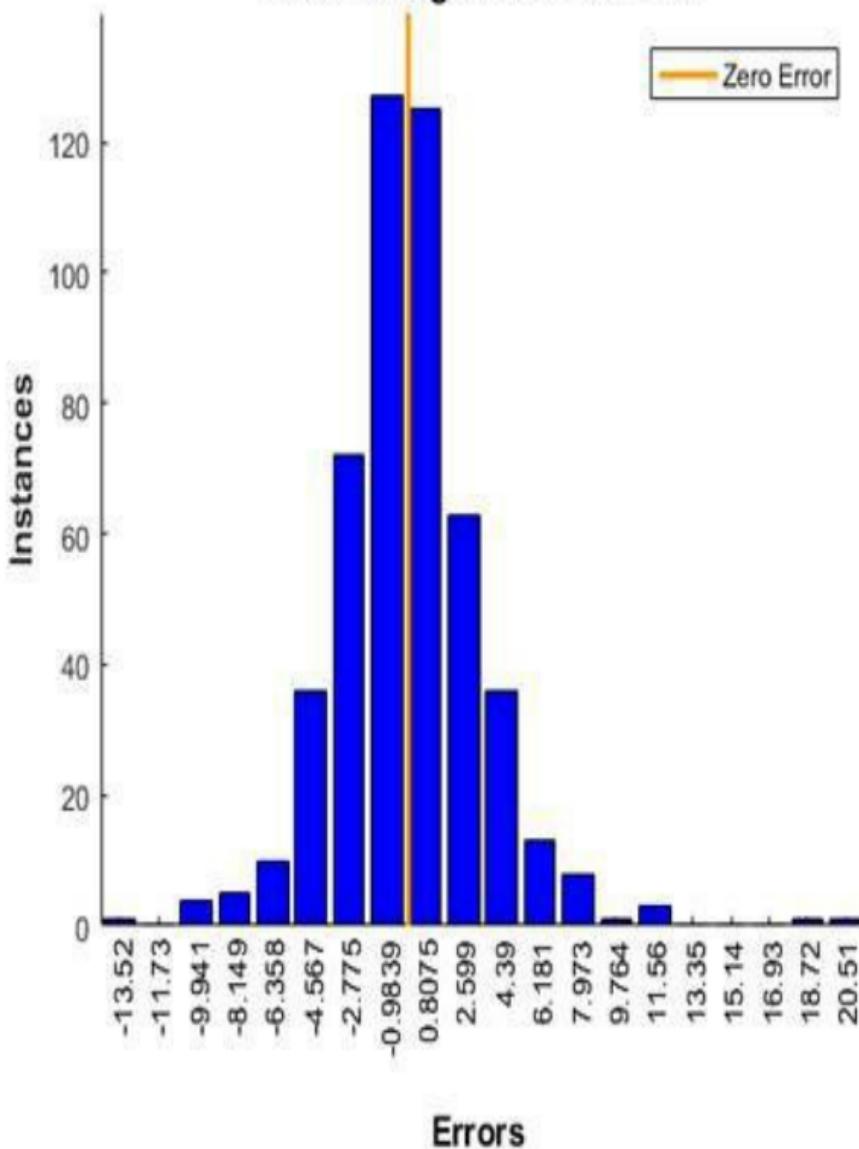


Another third measure of how well the neural network has fit data is the error histogram. This shows how the error sizes are distributed. Typically most errors are near zero, with very few errors far from that.

$$e = t - \hat{y};$$

```
ploterrhist(e)
```

Error Histogram with 20 Bins



This example illustrated how to design a neural network that estimates the median house value from neighborhood characteristics.

1.2 FUNCTION FITTING NEURAL NETWORK. EXAMPLES

```
net = fitnet(hiddenSizes)
```

```
net =
```

```
fitnet(hiddenSizes,trainFcn)
```

net = fitnet(hiddenSizes) returns a function fitting neural network with a hidden layer size of hiddenSizes (default=10).

The argument hiddenSizes represents the size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network. For example, you can specify a network

with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: [10, 8, 5]

net =
fitnet(hiddenSizes, trainFcn) returns a function fitting neural network with a hidden layer size of `hiddensizes` and training function, specified by `trainFcn` (default='trainlm'). The training functions are the following:

Training Function	Algorithm
'trainlm'	Levenberg-Marquardt
'trainbr'	Bayesian Regularization
'trainbfg'	BFGS Quasi-Newton
'trainrp'	Resilient Backpropagation
'trainscg'	Scaled Conjugate Gradient
'traincgb'	Conjugate Gradient with Powell/Beale I
'traincfg'	Fletcher-Powell Conjugate Gradient
'traincgp'	Polak-Ribière Conjugate Gradient

'trainoss'	One Step Secant
'traingdx'	Variable Learning Rate Gradient Descent
'traingdm'	Gradient Descent with Momentum
'traingd'	Gradient Descent

1.2.1 Construct and Train a Function Fitting Network

Load the training data.

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

The 1-by-94 matrix x contains the input values and the 1-by-94 matrix t contains the associated target output values.

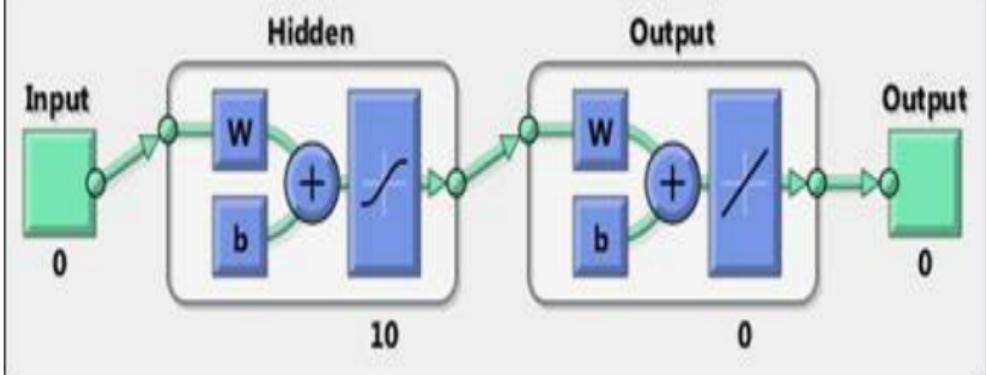
Construct a function fitting neural network with one hidden layer of size

10.

```
net = fitnet(10);
```

View the network.

```
view(net)
```



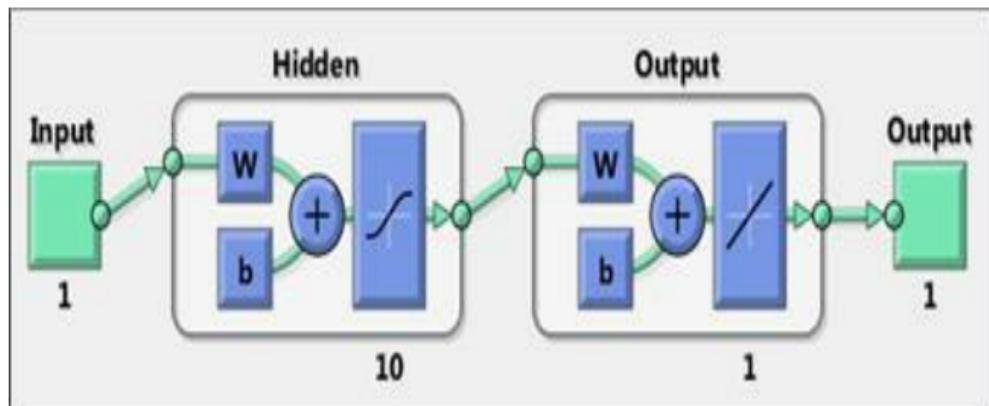
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

Train the network `net` using the training data.

```
net =  
train(net,x,t);
```

View the trained network.

```
view(net)
```



You can see that the sizes of the input and output are 1.

Estimate the targets using the trained network.

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

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf =  
perform(net, y, t)
```

```
perf =
```

1.4639e-04

The default training algorithm for a function fitting network is Levenberg-Marquardt ('trainlm'). Use the Bayesian regularization training algorithm and compare the performance results.

```
net =  
fitnet(10, 'trainbr')
```

```
net =  
train(net,x,t);
```

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

```
perf =  
perform(net,y,t)
```

```
perf =
```

3.3416e-10

The Bayesian regularization training algorithm improves the performance of the network in terms of estimating the target values.

1.2.2 Create and train Feedforward Neural Network

```
feedforwardnet(hiddenSizes, train)
```

This command construct the feedforward neural network. Feedforward networks consist of a series of layers. The first layer has a connection from the network input. Each subsequent layer has a connection from the previous layer. The final layer produces the network's output.

Feedforward networks can be used for any kind of input to output mapping. A feedforward network with one hidden

layer and enough neurons in the hidden layers, can fit any finite input-output mapping problem.

Specialized versions of the feedforward network include fitting ([fitnet](#)) and pattern recognition ([patternnet](#)) networks. A variation on the feedforward network is the cascade forward network ([cascadeforwardnet](#)) which has additional connections from the input to every layer, and from each layer to all following layers.

feedforwardnet(hiddenSizes, train)
these arguments,

hiddenSizes	Row vector of one or more hidden layer sizes (de-
trainFcn	Training function (default = 'trainlm')

and returns a feedforward neural network.

This example shows how to use feedforward neural network to solve a simple problem.

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

```
net =  
feedforwardnet(10);
```

```
net =  
train(net, x, t);
```

```
view(net)
```

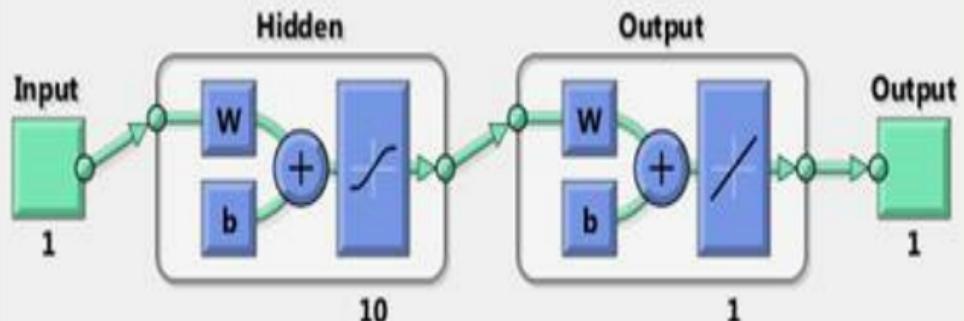
```
y = net(x);
```

```
perf =
```

```
perform(net, y, t)
```

```
perf =
```

```
1.4639e-04
```



1.2.3 Create and Train a Cascade Network

```
cascadeforwardnet(hiddenSizes, trainFcn)
```

Cascade-forward networks are similar to feed-forward networks, but include a connection from the input and every previous layer to following layers. As with feed-forward networks, a two-or more layer cascade-network can learn any finite input-output relationship arbitrarily well given enough hidden neurons.

```
cascadeforwardnet(hiddenSizes, trainFcn)
```

these arguments,

hiddenSizes	Row vector of one or more hidden layer sizes (default = [1 1 1])
trainFcn	Training function (default = 'trainlm')

and returns a new cascade-forward neural network.

Here a cascade network is created and trained on a simple fitting problem.

```
[x, t] =  
simplefit_dataset;  
  
net =  
cascadeforwardnet(10  
  
net =  
train(net, x, t);
```

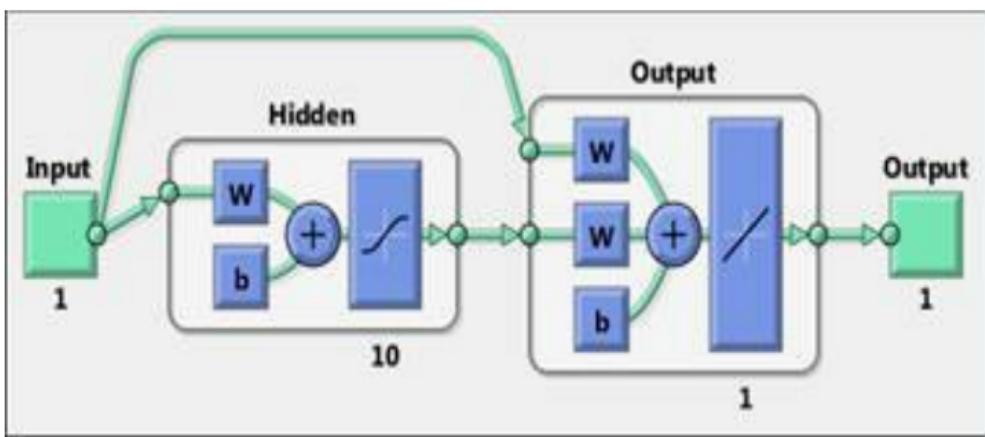
```
view(net)
```

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

```
perf =  
perform(net,y,t)
```

```
perf =
```

```
1.9372e-05
```



1.3 NETWORK PERFORMANCE

In MATLAB `mse` is a network performance function. It measures the network's performance according to the mean of squared errors.

1.3.1 Description

`perf = mse(net, t, y, ew)` takes these arguments:

<code>net</code>	Neural network
<code>t</code>	Matrix or cell array of targets
<code>y</code>	Matrix or cell array of outputs
<code>ew</code>	Error weights (optional)

and returns the mean squared error.

This function has two optional parameters, which are associated with networks whose `net.trainFcn` is set to this function:

- '`regularization`' can be set to any value between 0 and 1. The greater the regularization value, the more squared weights and

biases are included in the performance calculation relative to errors. The default is 0, corresponding to no regularization.

'normalization' can be set to 'none' (the default); 'standard', which normalizes errors between -2 and 2, corresponding to normalizing outputs and targets between -1 and 1; and 'percent', which normalizes errors between -1 and 1. This feature is useful for networks with multi-element outputs. It ensures that the relative accuracy of output elements with

differing target value ranges are treated as equally important, instead of prioritizing the relative accuracy of the output element with the largest target value range.

You can create a standard network that uses `mse` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `mse`, set `net.performFcn` to '`'mse'`'. This automatically sets `net.performParam` to a structure with the default optional parameter values.

1.3.2 Examples

Here a two-layer feedforward network is created and trained to predict median house prices using the `mse` performance function and a regularization value of 0.01, which is the default performance function for `feedforwardnet`.

```
[x, t] =
```

```
house_dataset;
```

```
net =
```

```
feedforwardnet(10);
```

```
net.performFcn =  
'mse'; % Redundant,  
MSE is default  
  
net.performParam.reg  
= 0.01;  
  
net =  
train(net,x,t);  
  
y = net(x);  
  
perf =
```

```
perform(net,t,y);
```

Alternately, you can call this function directly.

```
perf =
```

```
mse(net,x,t,'regular')
```

1.4 FIT REGRESSION MODEL AND PLOT FITTED VALUES VERSUS TARGETS. EXAMPLES

1.4.1 Description

`[r,m,b] = regression(t,y)` takes these arguments,

t	Target matrix or cell array data with a total of N matrix rows
y	Output matrix or cell array data of the same size

and returns these outputs,

r	Regression values for each of the N matrix rows
m	Slope of regression fit for each of the N matrix rows
b	Offset of regression fit for each of the N matrix rows

`[r,m,b] = regression(t,y,'one')` combines all matrix rows before regressing, and returns single scalar regression, slope, and offset values.

`plotregression(targets,outputs)` plots the

linear regression of targets relative to outputs.

plotregression(targs1,outs1,'name1',targs)
generates multiple plots.

1.4.2 Examples

Train a feedforward network, then calculate and plot the regression between its targets and outputs.

```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(20);  
  
net =  
train(net, x, t);
```

y = net(x);

[r,m,b] =
regression(t,y)

plotregression(t,y)

r =

1.0000

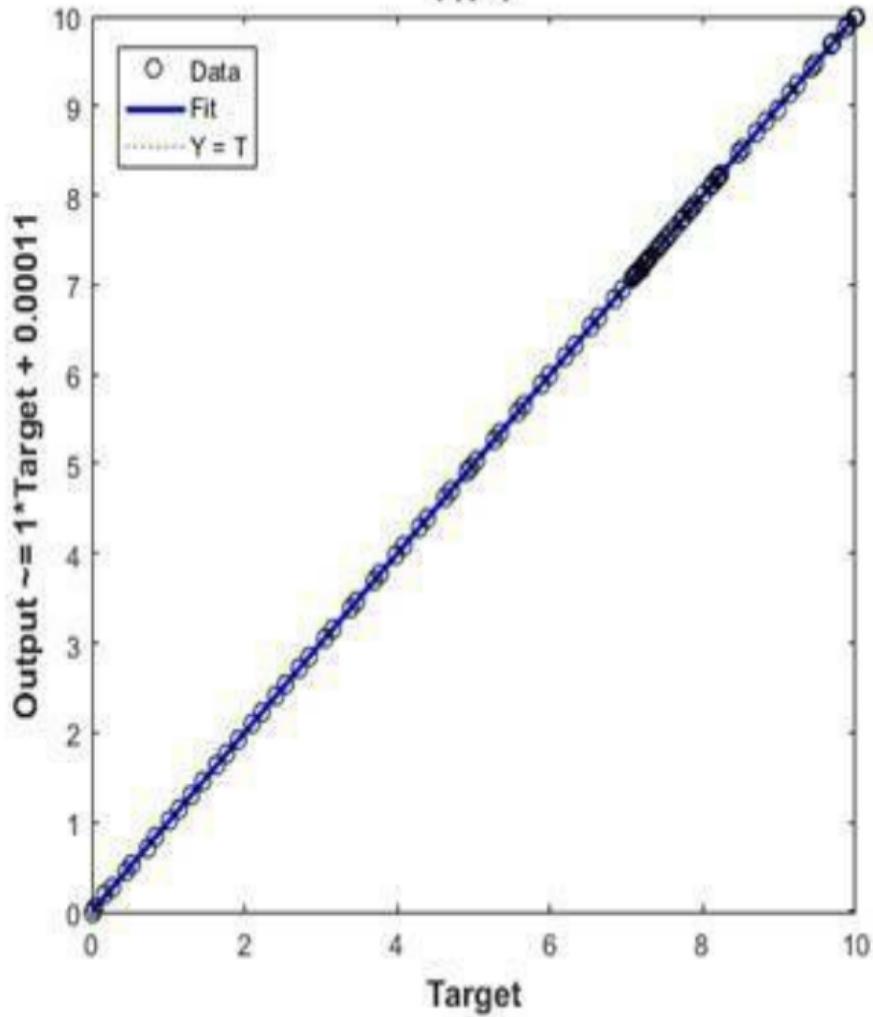
m =

1.0000

b =

1.0878e-04

: R=1

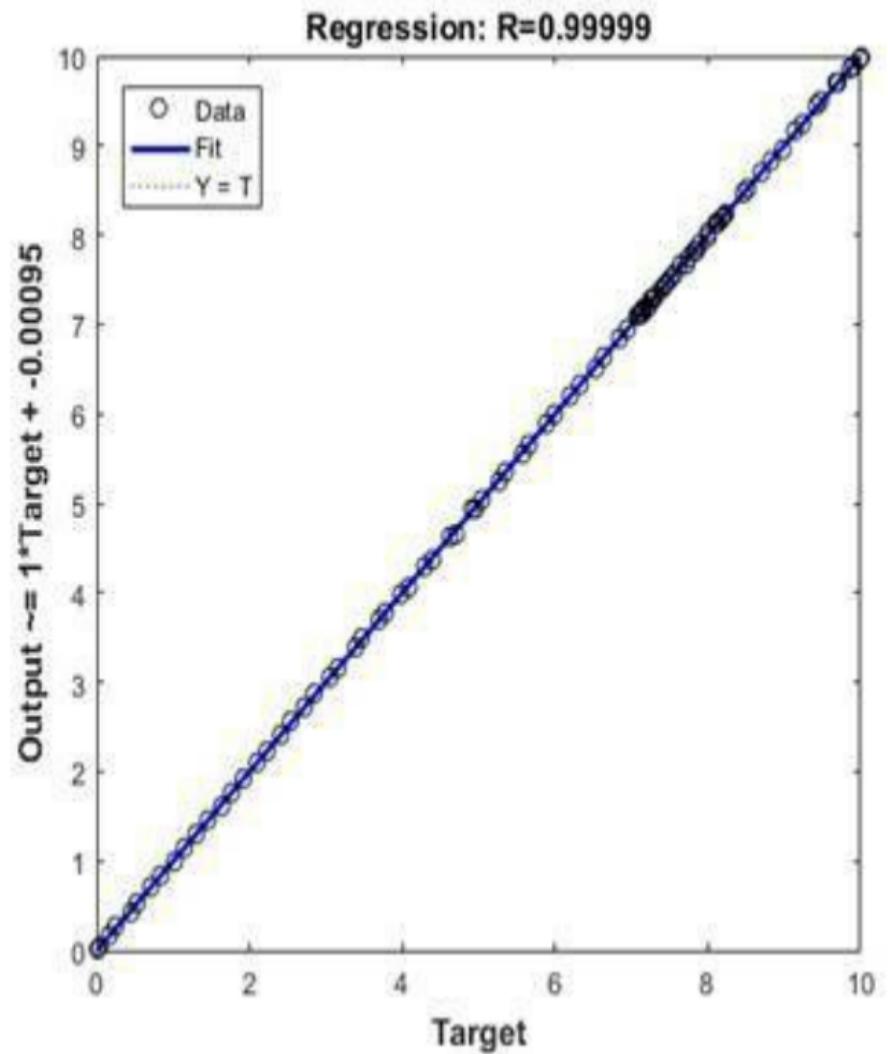


The next example Plot Linear

Regression

```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(10);  
  
net =  
train(net, x, t);  
  
y = net(x);
```

plotregression(t,y,'



1.5 PLOT OUTPUT AND TARGET VALUES. EXAMPLES

1.5.1 Description

`plotfit(net,inputs,targets)` plots the output function of a network across the range of the inputs `inputs` and also plots target `targets` and output data points associated with values in `inputs`. Error bars show the difference between outputs and `targets`.

The plot appears only for networks with one input.

Only the first output/targets appear if the network has more than one output.

`plotfit(targets1,inputs1,'name1')`, a series of plots.

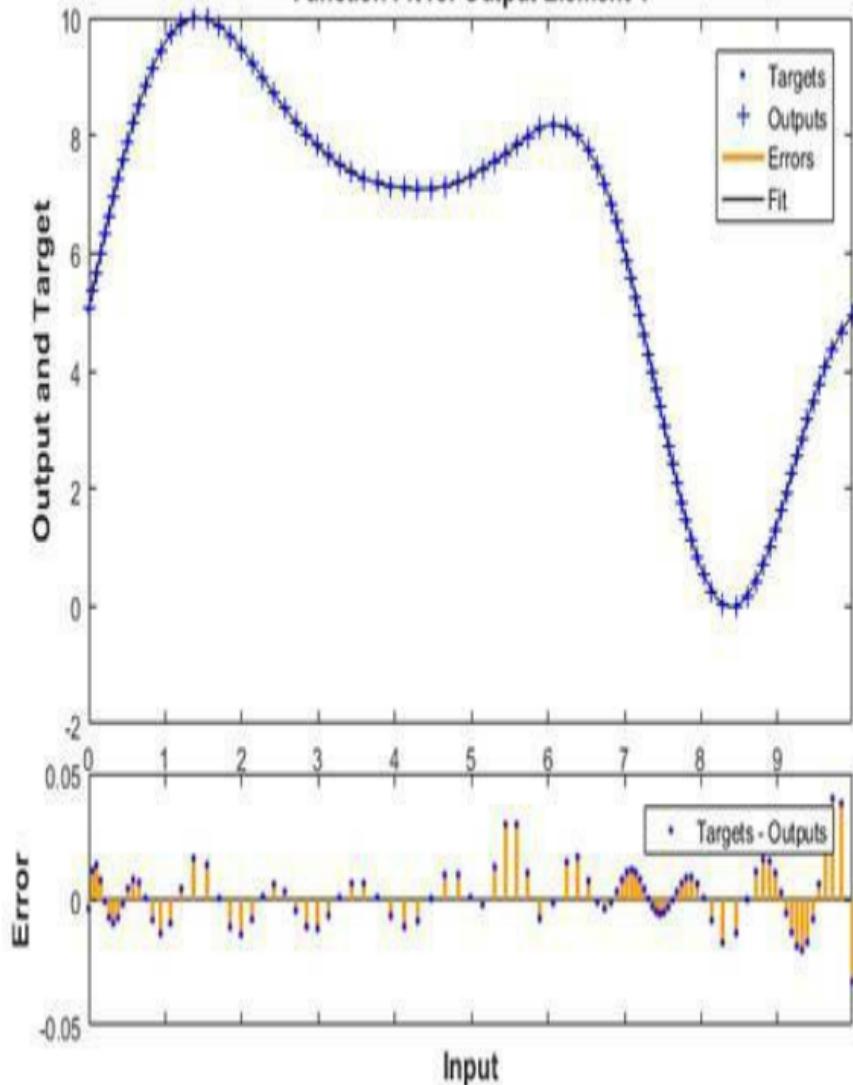
1.5.2 Examples

This example shows how to use a feed-forward network to solve a simple fitting problem.

```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(10);  
  
net =  
train(net, x, t);
```

plotfit (net, x, t)

Function Fit for Output Element 1



1.6 PLOT TRAINING STATE VALUES. EXAMPLES

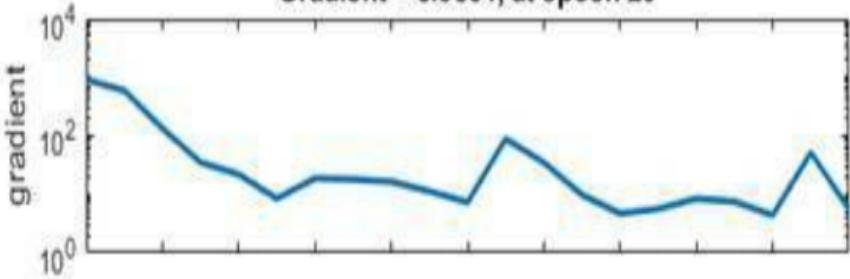
`plottrainstate(tr)` plots the training state from a training record `tr` returned by `train`.

Below is an example:

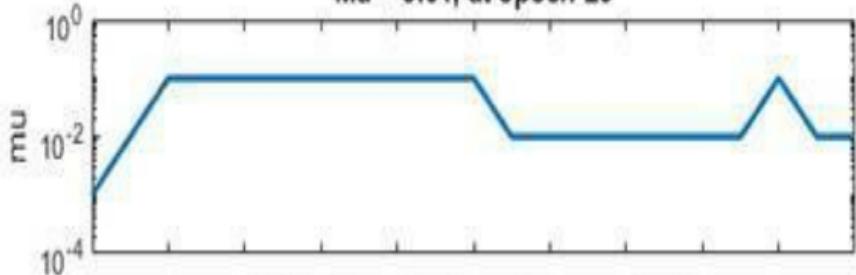
```
[x, t] =  
house_dataset;  
  
net =  
feedforwardnet(10);
```

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

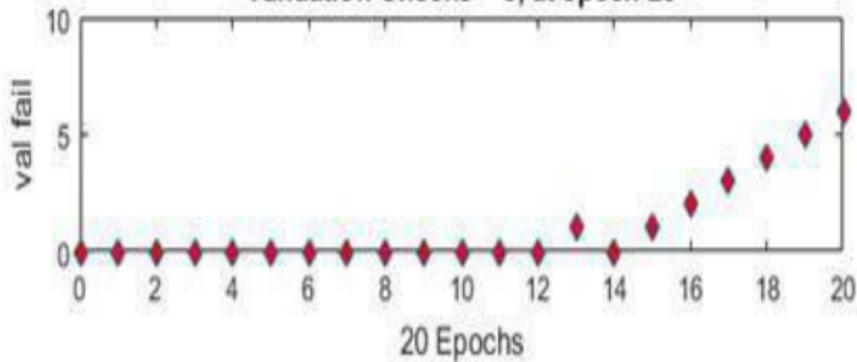
Gradient = 5.5864, at epoch 20



Mu = 0.01, at epoch 20



Validation Checks = 6, at epoch 20



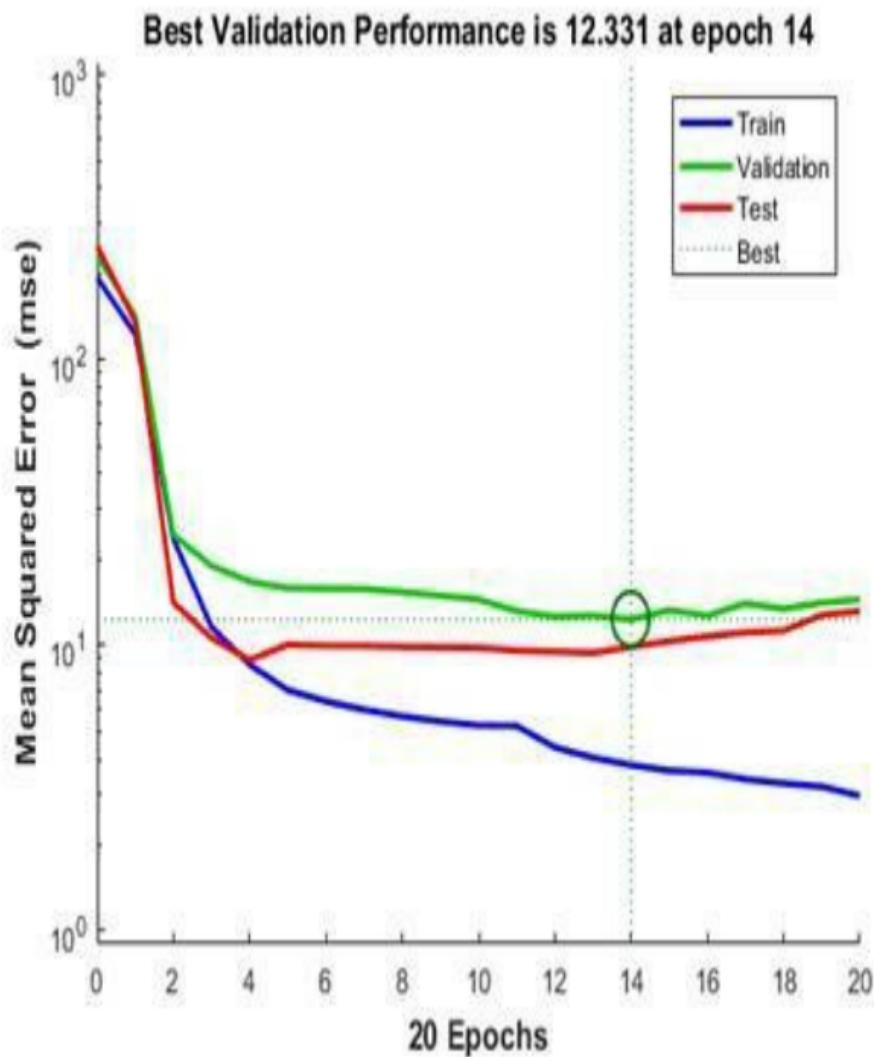
1.7 PLOT PERFORMANCES. EXAMPLES

`plotperform(TR)` plots error vs. epoch for the training, validation, and test performances of the training record `TR` returned by the function [train](#).

This example shows how to use `plotperform` to obtain a plot of training record error values against the number of training epochs.

```
[x,t] = house_dataset;  
net = feedforwardnet(10);  
[net,tr] = train(net,x,t);
```

plotperform(tr)



Generally, the error reduces after more

epochs of training, but might start to increase on the validation data set as the network starts overfitting the training data. In the default setup, the training stops after six consecutive increases in validation error, and the best performance is taken from the epoch with the lowest validation error.

1.8 PLOT HISTOGRAM OF ERROR VALUES. EXAMPLES

1.8.1 Syntax

```
ploterrhist(e)
```

```
ploterrhist(e1,'name1',e2,'na
```

```
ploterrhist(...,'bins',bins)
```

1.8.2 Description

`ploterrhist(e)` plots a histogram of error values `e`.

`ploterrhist(e1, 'name1', e2, 'name2')`
any number of errors and names and plots each pair.

`ploterrhist(..., 'bins', bins)` takes an optional property name/value pair which defines the number of bins to use in the histogram plot. The default is 20.

1.8.3 Examples

Here a feedforward network is used to solve a simple fitting problem:

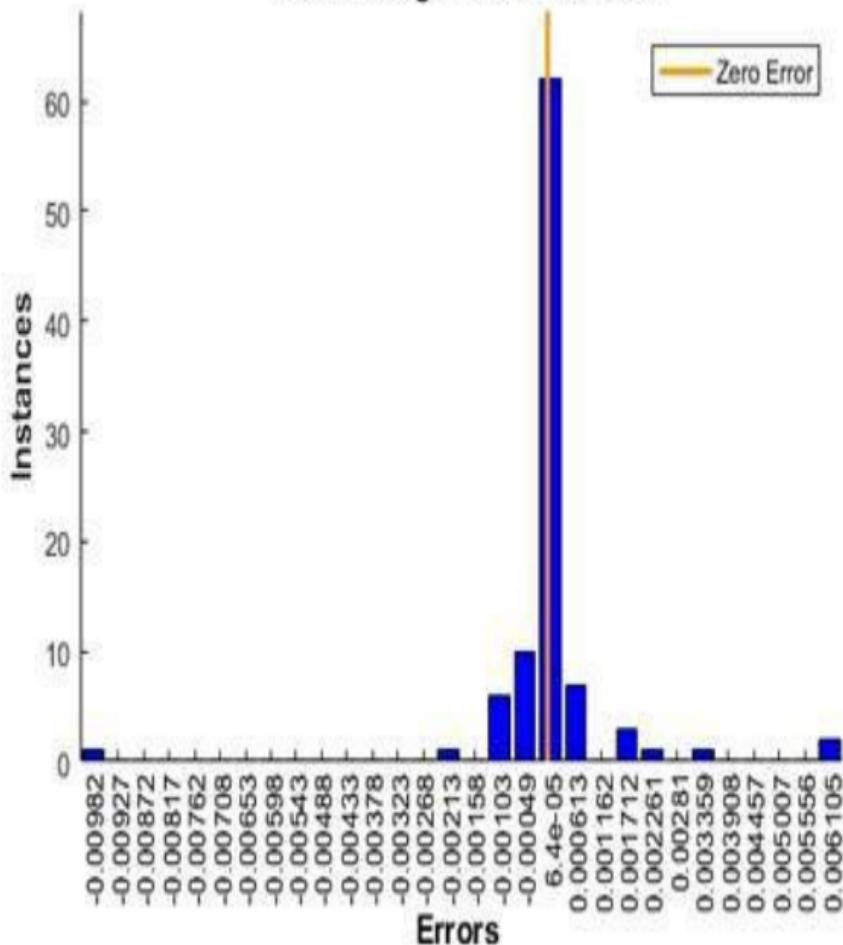
```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(20);  
  
net =  
train(net, x, t);
```

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

```
e = t - y;
```

```
ploterrhist(e, 'bins'
```

Error Histogram with 30 Bins



1.9 GENERATE MATLAB FUNCTION FOR SIMULATING NEURAL NETWORK. EXAMPLES

`genFunction` ([net](#), [pathname](#)) generates a complete stand-alone MATLAB function for simulating a neural network including all settings, weight and bias values, module functions, and calculations in one file. The result is a standalone MATLAB function file. You can also use this function with MATLAB Compiler™ and MATLAB Coder™ tools.

`genFunction(____, 'MatrixOnly', 'yes')`
the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks, the matrix columns are interpreted as independent samples. For dynamic networks, the matrix columns are interpreted as a series of time steps. The default value is 'no'.

`genFunction(____, 'ShowLinks', 'no')`
the default behavior of displaying links to generated help and source code. The default is 'yes'.

1.9.1 Create Functions from Static Neural Network

This example shows how to create a MATLAB function and a MEX-function from a static neural network.

First, train a static network and calculate its outputs for the training data.

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

```
houseNet =  
feedforwardnet(10);
```

```
houseNet =  
train(houseNet,x,t);  
  
y = houseNet(x);
```

Next, generate and test a MATLAB function. Then the new function is compiled to a shared/dynamically linked library with `mcc`.

```
genFunction(houseNet)
```

```
y2 = houseFcn(x);
```

```
accuracy2 =  
max(abs(y-y2))
```

```
mcc -W lib:libHouse  
-T link:lib houseFcn
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB

Coder tool codegen to generate a MEX-function, which is also tested.

```
genFunction(houseNet
```

```
y3 = houseFcn(x);
```

```
accuracy3 =  
max(abs(y-y3))
```

```
x1Type =
```

```
coder.typeof(double([13 Inf])); % Coder  
type of input 1
```

```
codegen houseFcn.m -  
config:mex -o  
houseCodeGen -args  
{x1Type}
```

```
y4 =  
houseCodeGen(x);
```

```
accuracy4 =
```

max (abs (y-y4))

1.9.2 Create Functions from Dynamic Neural Network

This example shows how to create a MATLAB function and a MEX-function from a dynamic neural network.

First, train a dynamic network and calculate its outputs for the training data.

```
[x, t] =  
maglev_dataset;  
  
maglevNet =  
narxnet(1:2, 1:2, 10);
```

```
[X,Xi,Ai,T] =  
preparets(maglevNet,  
{ },t);  
  
maglevNet =  
train(maglevNet,X,T,:  
[y,xf,af] =  
maglevNet(X,Xi,Ai);
```

Next, generate and test a MATLAB

function. Use the function to create a shared/dynamically linked library with mcc.

```
genFunction(maglevNe
```

```
[y2,xf,af] =  
maglevFcn(X,Xi,Ai);
```

```
accuracy2 =  
max(abs(cell2mat(y) -  
cell2mat(y2)))
```

```
mcc -W lib:libMaglev  
-T link:lib  
maglevFcn
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB Coder tool `codegen` to generate a MEX-function, which is also tested.

```
genFunction(maglevNe...
```

```
x1 =  
cell2mat(X(1,:)); %  
Convert each input  
to matrix
```

```
x2 =  
cell2mat(X(2,:));  
  
xil =  
cell2mat(Xi(1,:)); %  
Convert each input
```

state to matrix

xi2 =

cell2mat(Xi(2,:));

[y3,xf1,xf2] =

maglevFcn(x1,x2,xi1,:)

accuracy3 =

max(abs(cell2mat(y) -
y3)))

```
x1Type =  
coder.typeof(double([1 Inf])); % Coder  
type of input 1
```

```
x2Type =  
coder.typeof(double([1 Inf])); % Coder  
type of input 2
```

```
xi1Type =  
coder.typeof(double([
```

```
[1 2]); % Coder type  
of input 1 states
```

```
xi2Type =  
coder.typeof(double([1 2])); % Coder type  
of input 2 states
```

```
codegen maglevFcn.m  
-config:mex -o  
maglevNetCodeGen -  
args {x1Type x2Type  
xi1Type xi2Type}
```

```
[y4,xf1,xf2] =  
maglevNetCodeGen(x1,:  
  
dynamic_codegen_accu:  
=  
max(abs(cell2mat(y) -  
y4)))
```

Chapter 2

CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK.

EXAMPLES

2.1 CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK

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.

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:

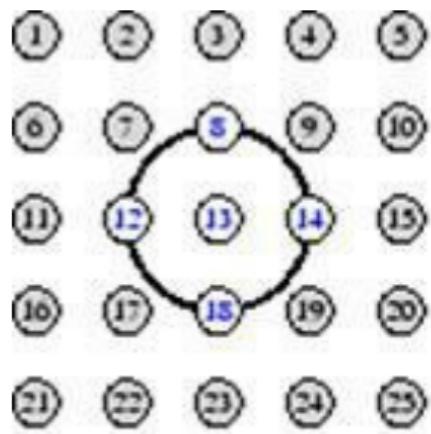
or

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^* .

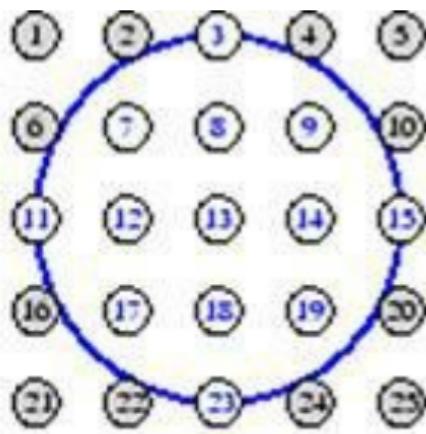
Thus, when a vector \mathbf{p} is presented, the weights of the winning neuron *and* its close neighbors move toward \mathbf{p} . Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.



$$N_{13}(1)$$



$$N_{13}(2)$$

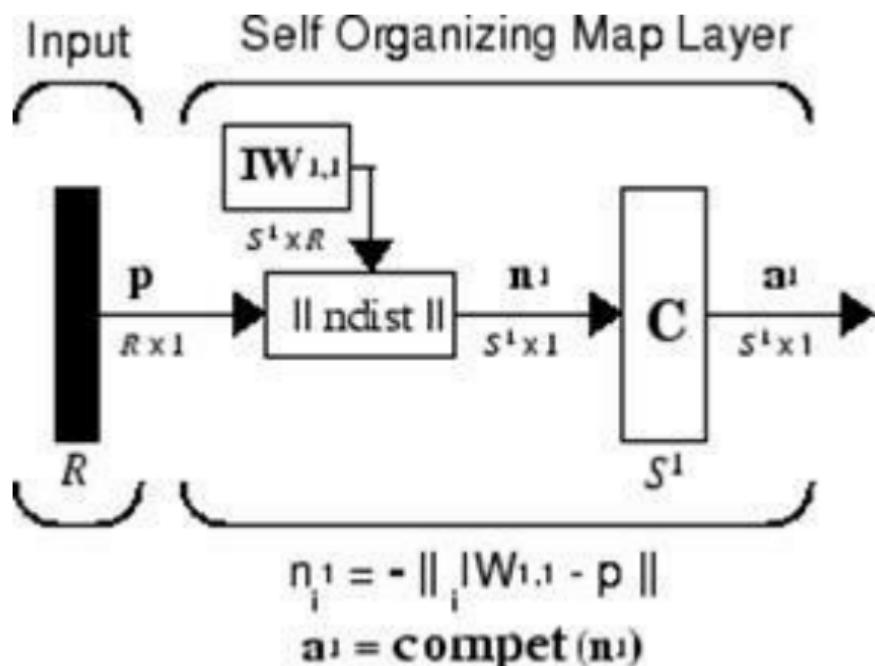
These neighborhoods could be written as $N_{13}(1) = \{8, 12, 13, 14, 18\}$ and $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$.

The neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional

arrangement, or three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

2.2 ARCHITECTURE

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element a^1_i 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.

2.3 CREATE A SELF-ORGANIZING MAP NEURAL NETWORK (SELFORGMAP). EXAMPLES

You can create a new SOM network with the function [selforgmap](#). This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that you want to create a

network having input vectors with two elements, and that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is:

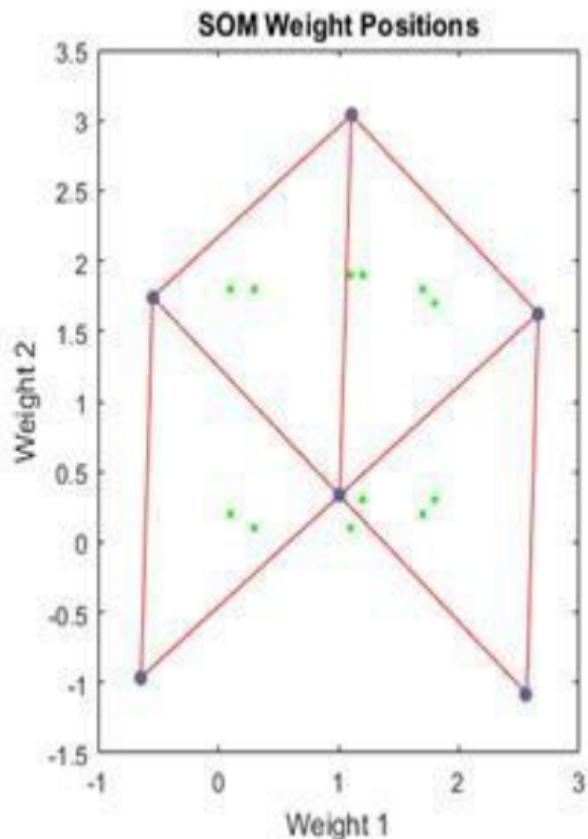
```
net = selforgmap([2,3]);
```

Suppose that the vectors to train on are:

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1  
1.8 1.7;...  
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9  
1.7 1.8];
```

You can configure the network to input the data and plot all of this with:

```
net = configure(net,P);  
plotsompos(net,P)
```



The green spots are the training vectors. The initialization for selforgmap spreads the initial weights across the input space. Note that they are initially some distance from the training vectors.

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated ([negdist](#)) to get the weighted inputs. The weighted inputs are also the net inputs ([netsum](#)). The net inputs compete ([compet](#)) so that only the neuron with the most positive net input will output a 1.

2.4 TRAINING (LEARN SOMB). EXAMPLES

The default learning in a self-organizing feature map occurs in the batch mode (trainbu). The weight learning function

for the self-organizing map is [learnsomb](#).

First, the network identifies the winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner or for which it is in the neighborhood of a winner. The distance that defines the size of the neighborhood is altered during training through two phases.

Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts at a given initial distance, and decreases to the tuning neighborhood distance (1.0). As the neighborhood distance

decreases over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood size has decreased below 1 so only the winning neuron learns for each sample.

Now take a look at some of the specific values commonly used in these networks.

Learning occurs according to the learnsomb learning parameter, shown

here with its default value.

Learning Parameter	Default Value	Pu
LP.init_neighborhood	3	Ini
LP.steps	100	Or

The neighborhood size NS is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as LP.steps. During this phase, the algorithm adjusts ND from the initial neighborhood size LP.init_neighborhood down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, ND is less than 1. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered. The training continues in order to give the neurons time to spread out evenly across

the input vectors.

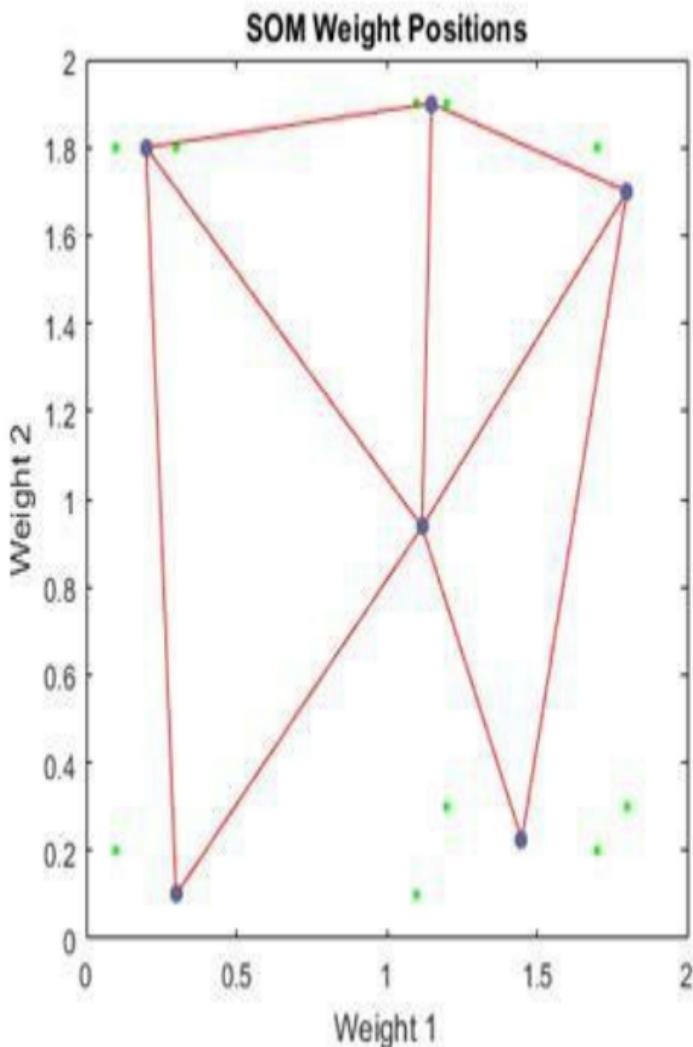
As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

You can train the network for 1000

epochs with

```
net.trainParam.epochs = 1000;  
net = train(net,P);  
plotsompos(net,P)
```



You can see that the neurons have started

to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

2.5 EXAMPLES

Two examples are described briefly below.

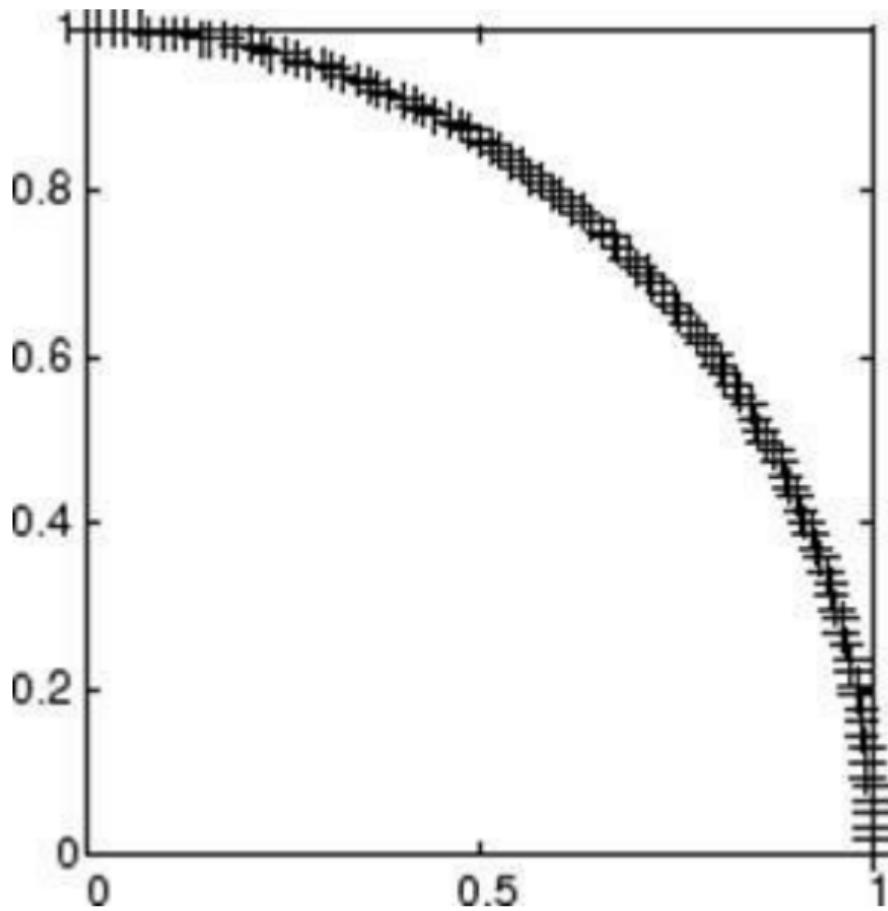
2.5.1 One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90° .

```
angles = 0:0.5*pi/99:0.5*pi;
```

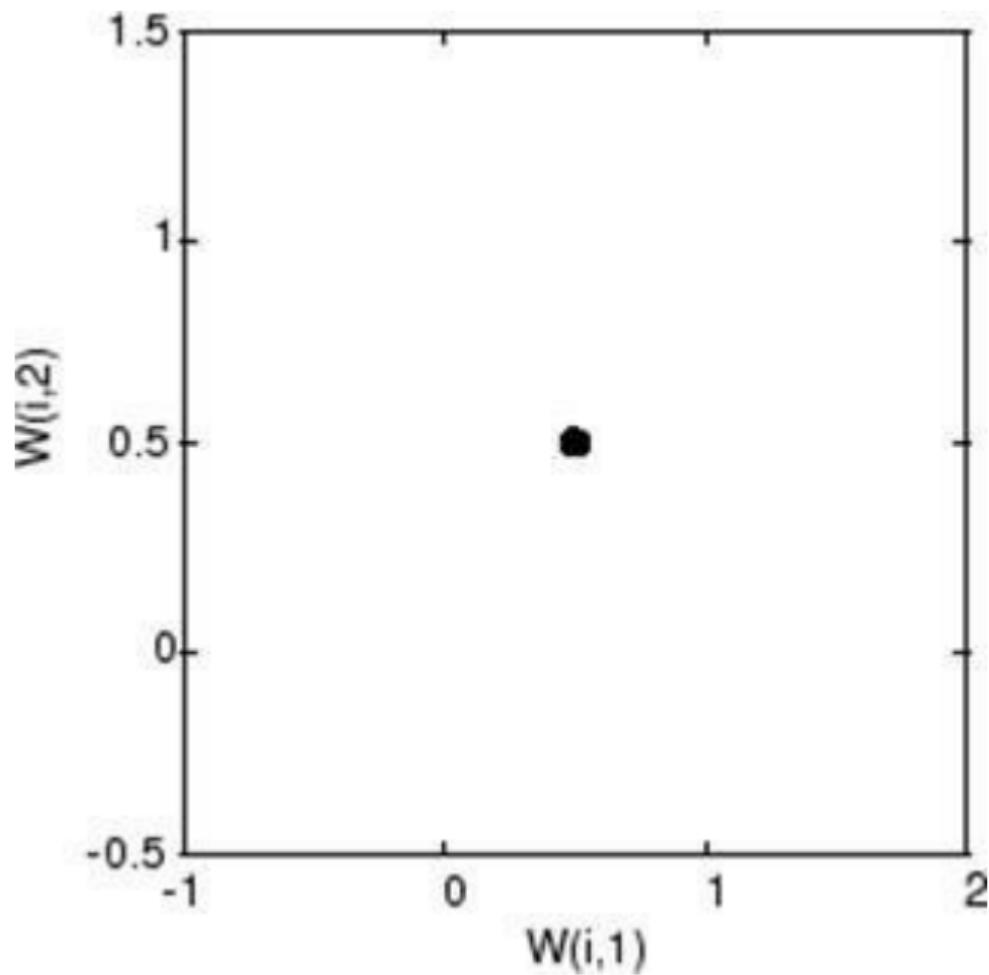
Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```



A self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these

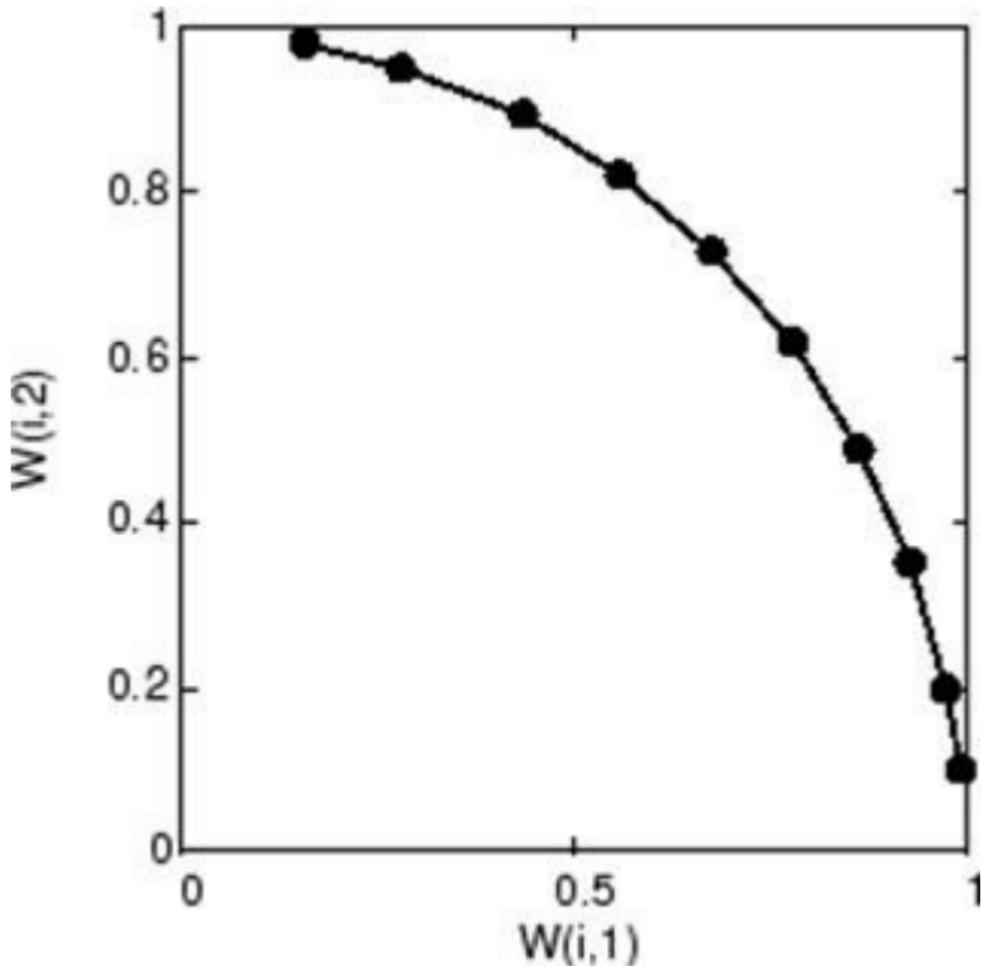
neurons are at the center of the figure.



Of course, because all the weight

vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial

vectors does not guarantee identical training results.

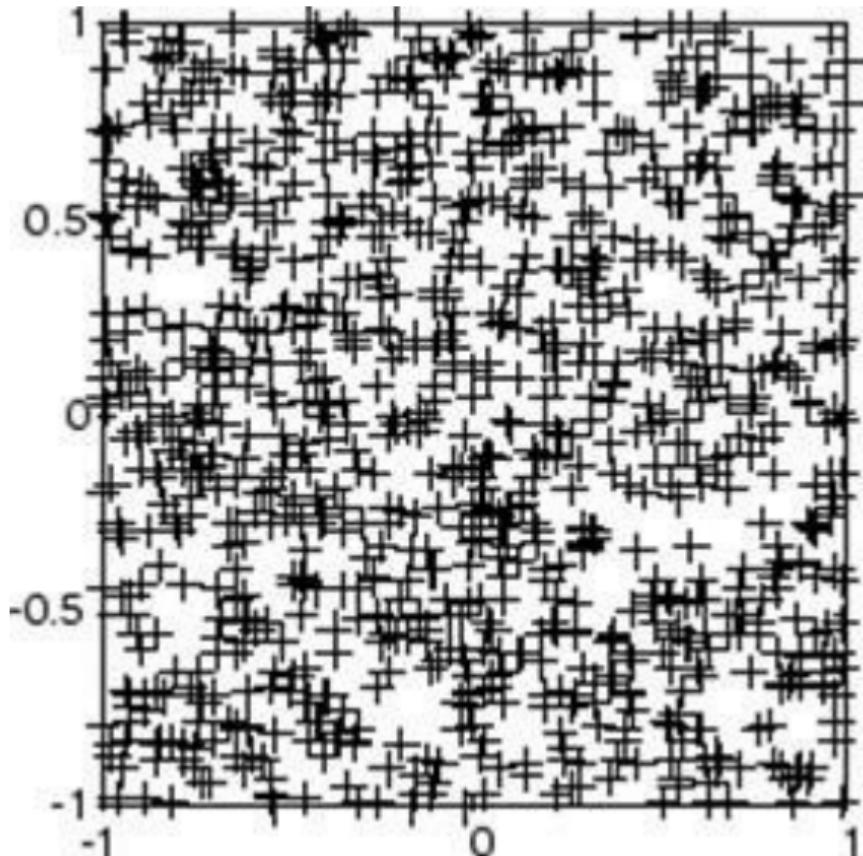
2.5.2 Two-Dimensional Self-Organizing Map

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.

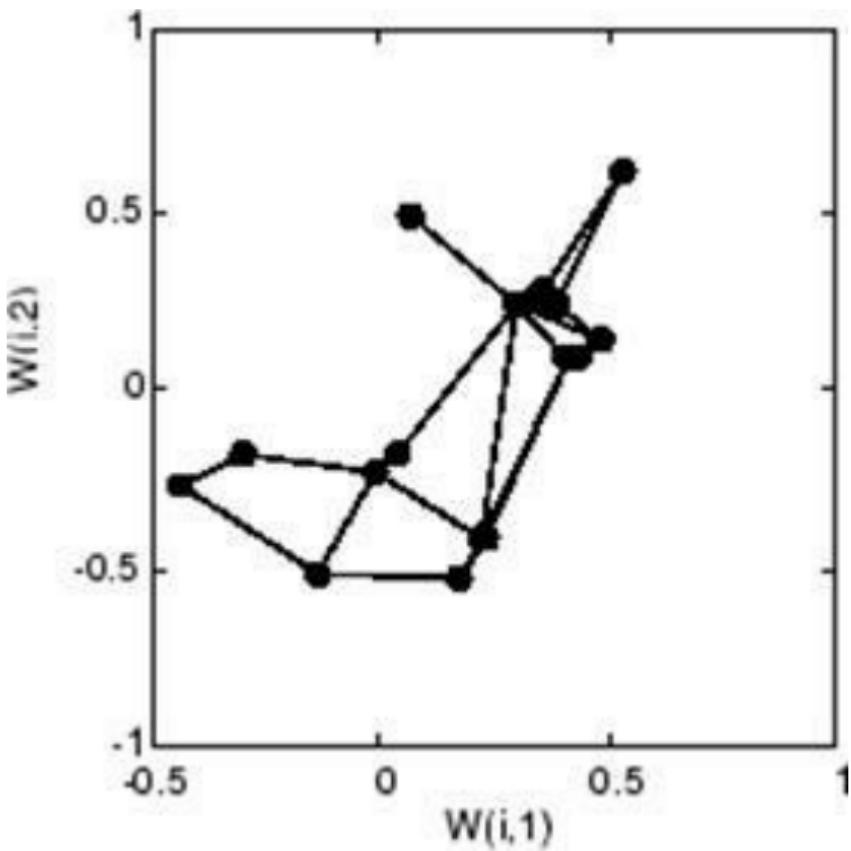


A 5-by-6 two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with

distances calculated according to the Manhattan distance neighborhood function [mandist](#).

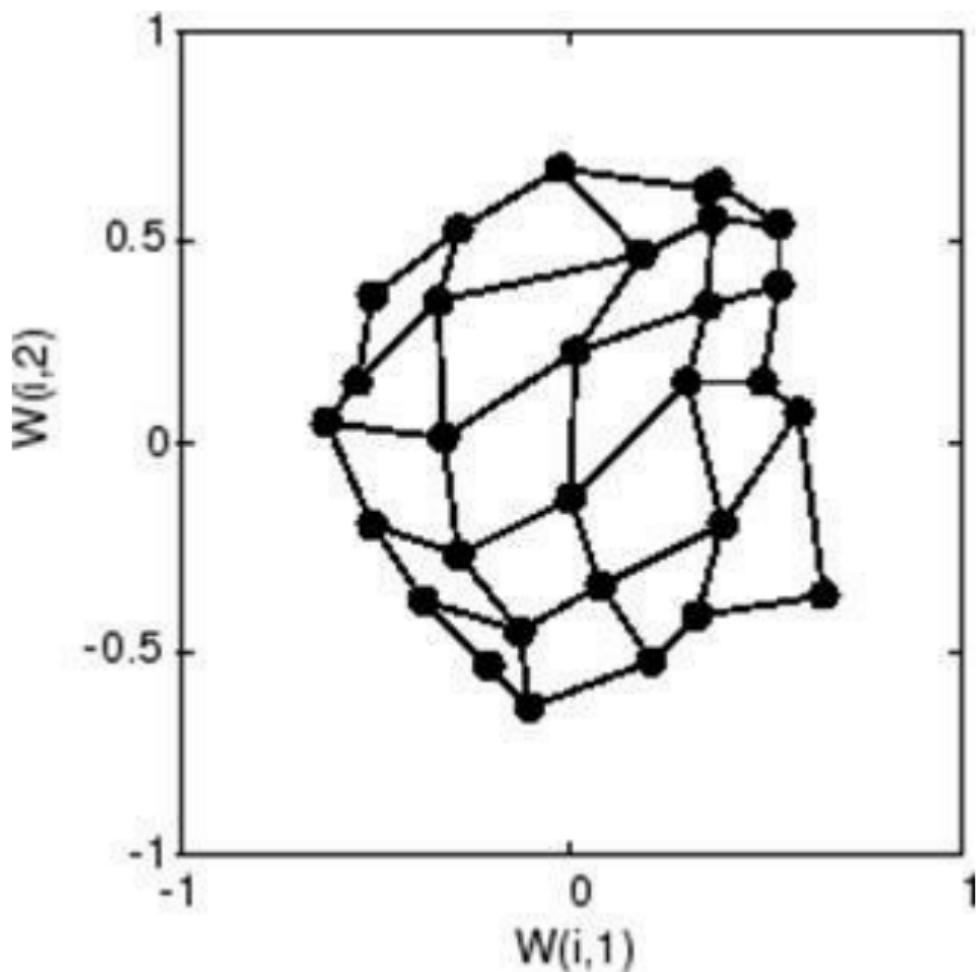
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

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.

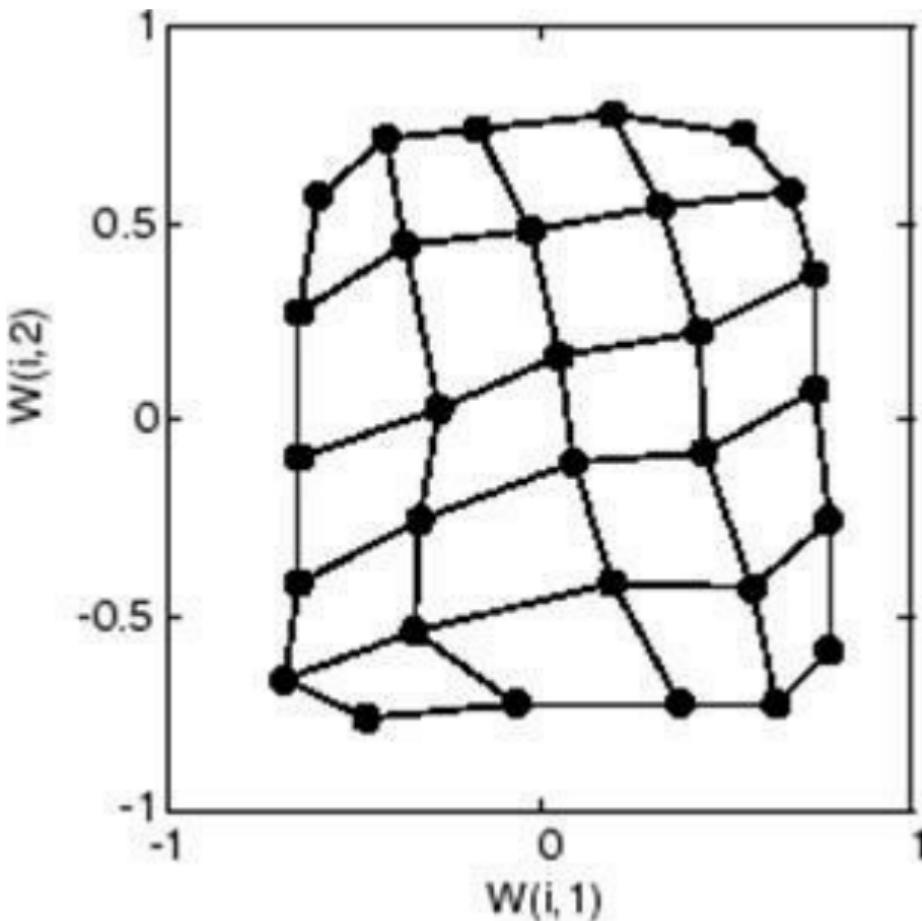
Here is the map after 120 cycles.



After 120 cycles, the map has begun to organize itself according to the topology

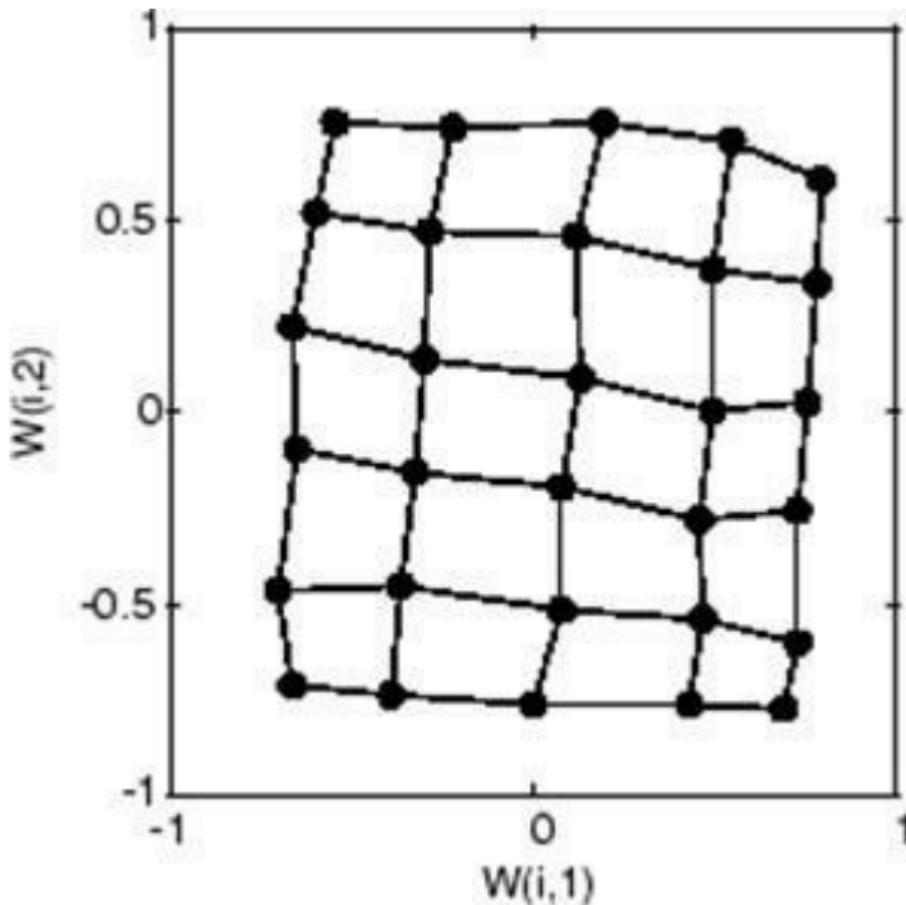
of the input space, which constrains input vectors.

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.

2.5.3 Training with the Batch Algorithm

The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

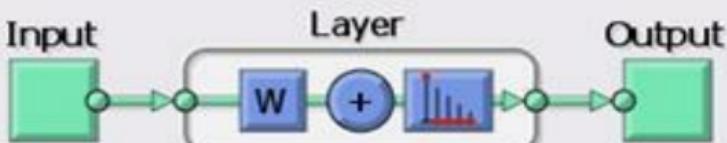
```
x = simplecluster_dataset  
net = selforgmap([6 6]);  
net = train(net,x);
```

This command sequence creates and trains a 6-by-6 two-dimensional map of 36 neurons. During training, the

following figure appears.

Neural Network Training (nntraintool)

Neural Network



Algorithms

Training: Batch Unsupervised Weight/Bias Training (trainbuwb)

Progress

Epoch: 0 200 Iterations 200
Time: 0:00:09

Plots

SOM Topology (plotsomtop)

SOM Neighbor Connections (plotsomnc)

SOM Neighbor Distances (plotsomnd)

SOM Weight Planes (plotsomplanes)

SOM Sample Hits (plotsomhits)

SOM Weight Positions (plotsompos)

Plot Interval: 1 epochs

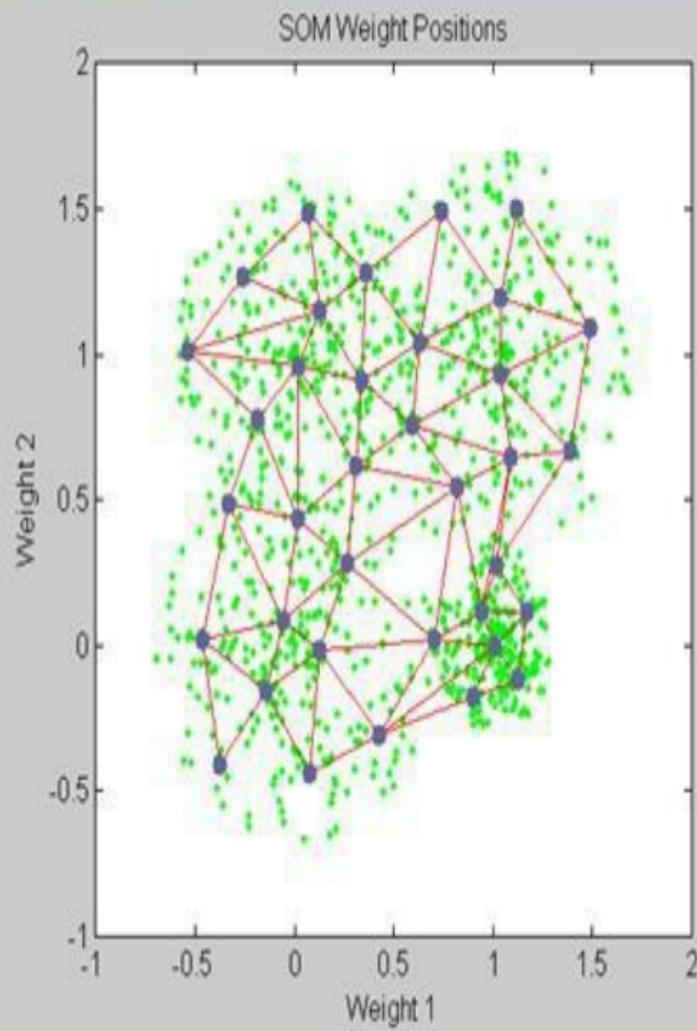
✓ Opening SOM Neighbor Distances Plot

Stop Training

Cancel

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.

SOM Weight Positions (plotsompos)



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

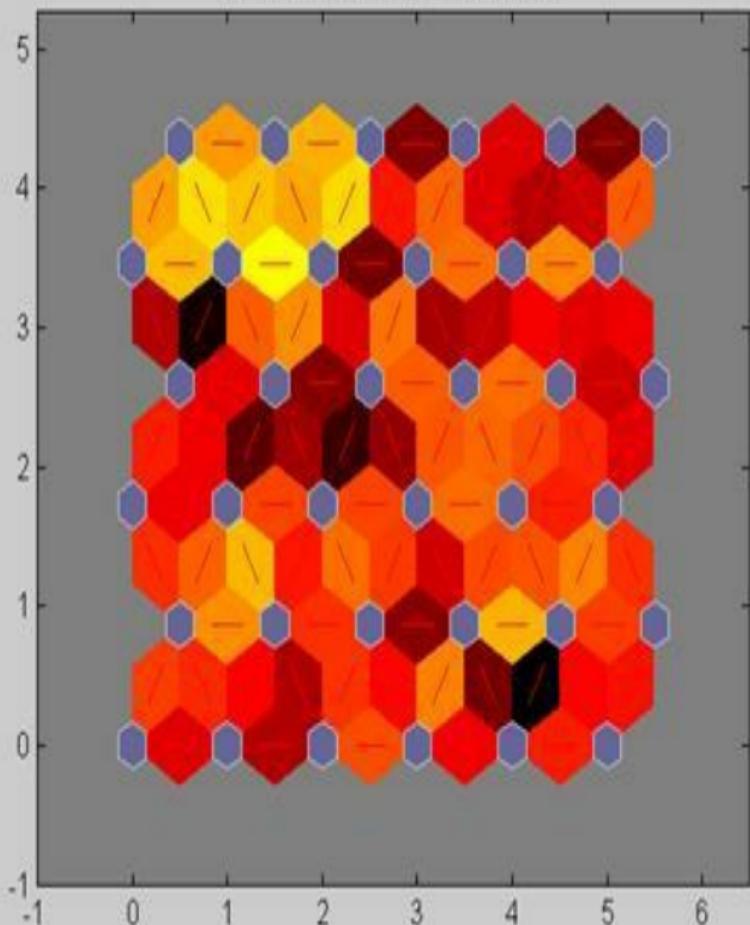
distances between neurons.

- 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

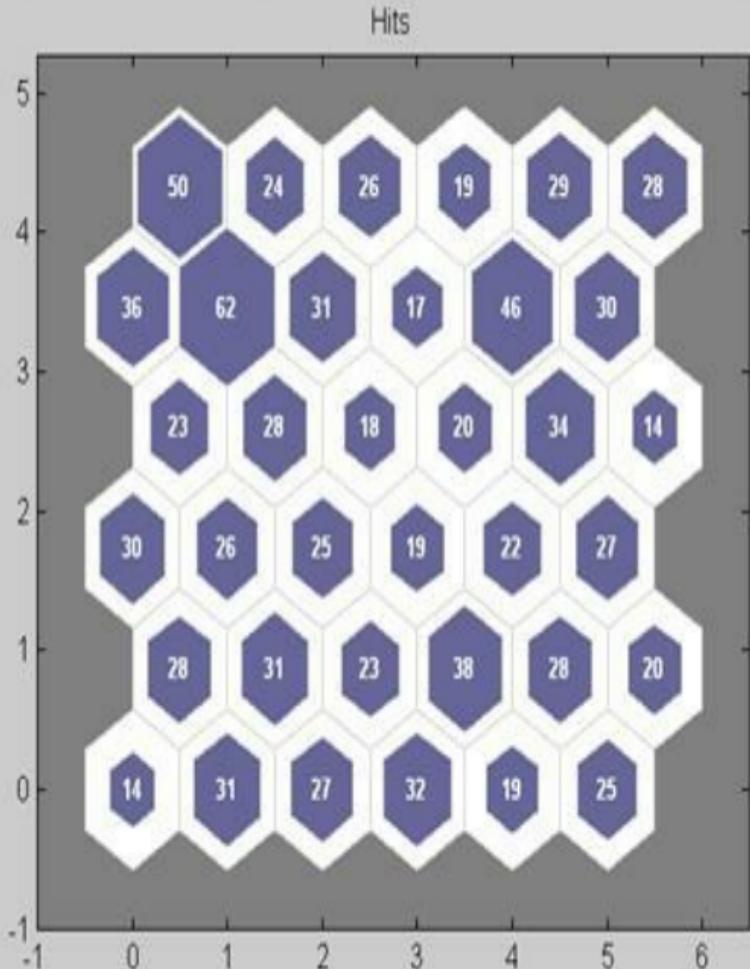
colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor distance figure. The segments in the lower-right region of the neighbor distance figure are darker than those in the upper left. This color difference indicates that data points in this region are farther apart. This distance is confirmed in the weight positions figure.

SOM Neighbor Weight Distances



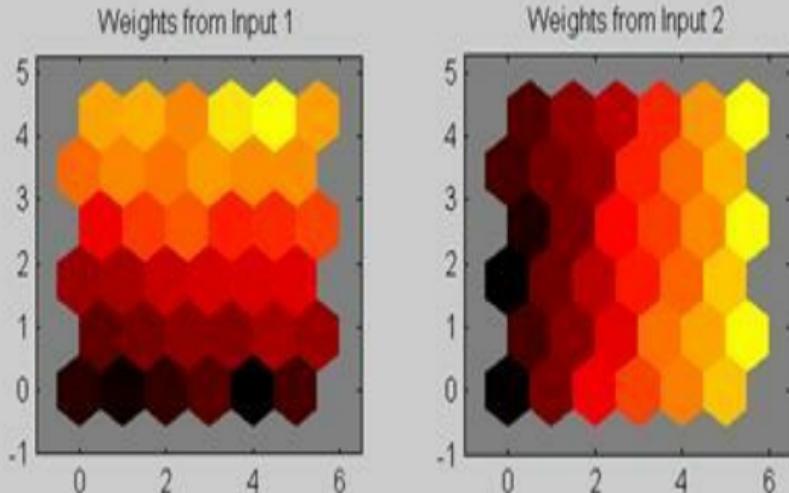
Another useful figure can tell you how

many data points are associated with each neuron. Click **SOM Sample Hits** to see the following figure. It is best if the data are fairly evenly distributed across the neurons. In this example, the data are concentrated a little more in the upper-left neurons, but overall the distribution is fairly even.



You can also visualize the weights

themselves using the weight plane figure. Click **SOM Weight Planes** in the training window to obtain the next figure. There is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2.



You can also produce all of the previous figures from the command line. Try these plotting commands: [plotsomhits](#), [plotsomnc](#), [plotsompos](#), and [plotsomtop](#).

2.6 SELFORGMAP

Self-organizing map

Syntax

```
selforgmap(dimensions,coverSteps,ii)
```

Description

Self-organizing maps learn to cluster data based on similarity, topology, with a preference (but no guarantee) of assigning the same number of instances to each class.

Self-organizing maps are used both to cluster data and to reduce the

dimensionality of data. They are inspired by the sensory and motor mappings in the mammal brain, which also appear to automatically organizing information topologically.

`selforgmap(dimensions,coverSteps,initN`
these arguments,

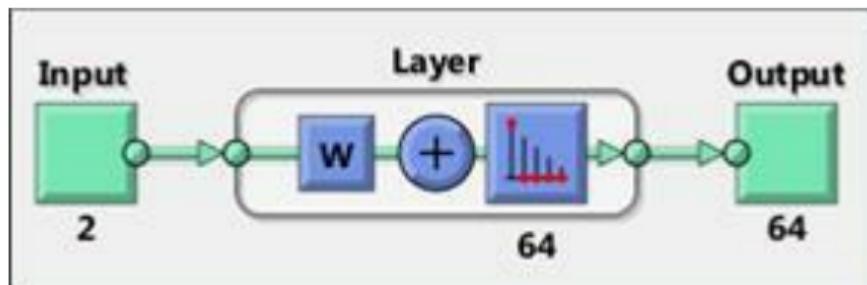
<code>dimensions</code>	Row vector of dimension sizes (default = [8 8])
<code>coverSteps</code>	Number of training steps for initial covering of the input data (default = 100)
<code>initNeighbor</code>	Initial neighborhood size (default = 3)
<code>topologyFcn</code>	Layer topology function (default = 'hextop')
<code>distanceFcn</code>	Neuron distance function (default = 'linkdist')

and returns a self-organizing map.

Examples. Use Self-Organizing Map to Cluster Data

Here a self-organizing map is used to cluster a simple set of data.

```
x = simplecluster_dataset;  
net = selforgmap([8 8]);  
net = train(net,x);  
view(net)  
y = net(x);  
classes = vec2ind(y);
```



2.7 FUNCTIONS FOR SELF-ORGANIZNG MAPS AND EXAMPLES

2.7.1 plotsomhits

Plot self-organizing map sample hits

Syntax

```
plotsomhits(net, inputs)
```

Description

plotsomhits(net, inputs) plots
a SOM layer, with each neuron showing

the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples. Plot SOM Sample Hits

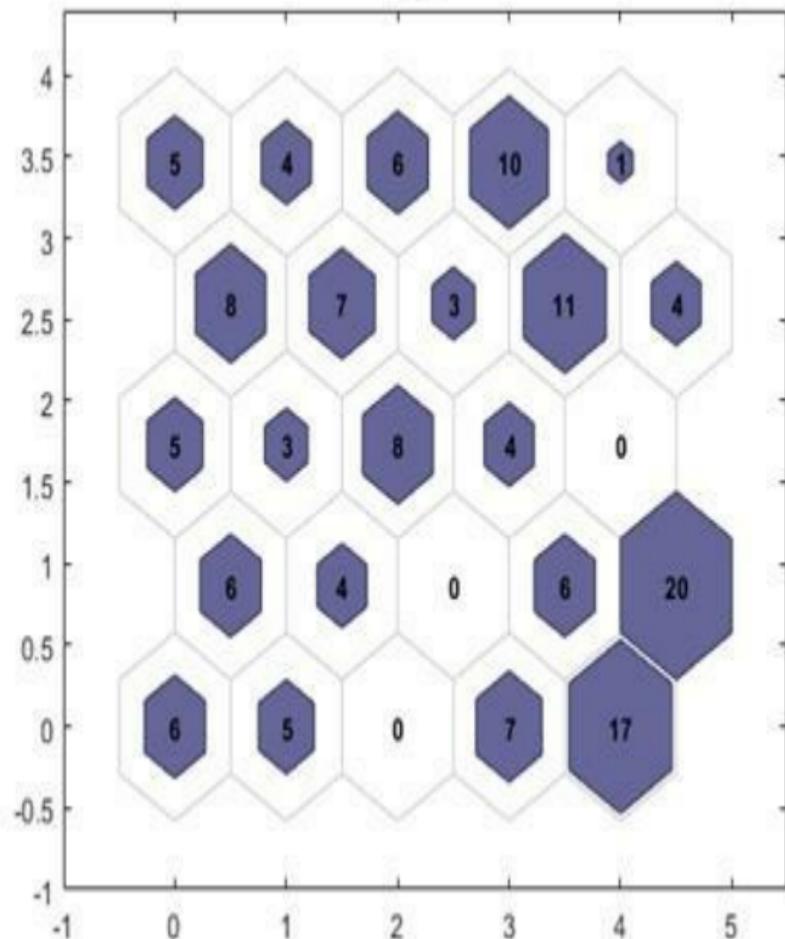
```
x = iris_dataset;
```

```
net = selforgmap([5  
5]);
```

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

```
plotsomhits(net,x)
```

Hits



2.7.2 plotsomnc

Plot self-organizing map neighbor connections

Syntax

```
plotsomnc(net)
```

Description

`plotsomnc(net)` plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

This plot supports SOM networks with hextop and gridtop topologies,

but not tritop or randtop.

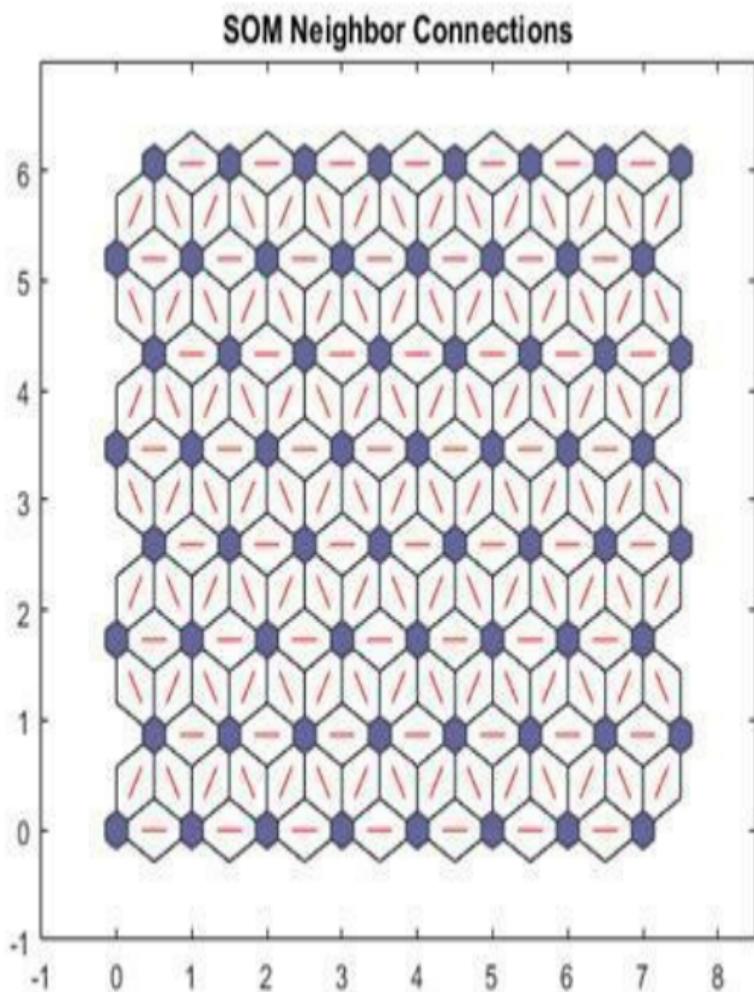
Examples. Plot SOM Neighbor Connections

```
x = iris_dataset;
```

```
net = selforgmap([8  
8]);
```

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

plotsomnc (net)



2.7.3 plotsomnd

Plot self-organizing map neighbor distances

Syntax

```
plotsomnd(net)
```

Description

`plotsomnd(net)` plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines. The neighbor patches are colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

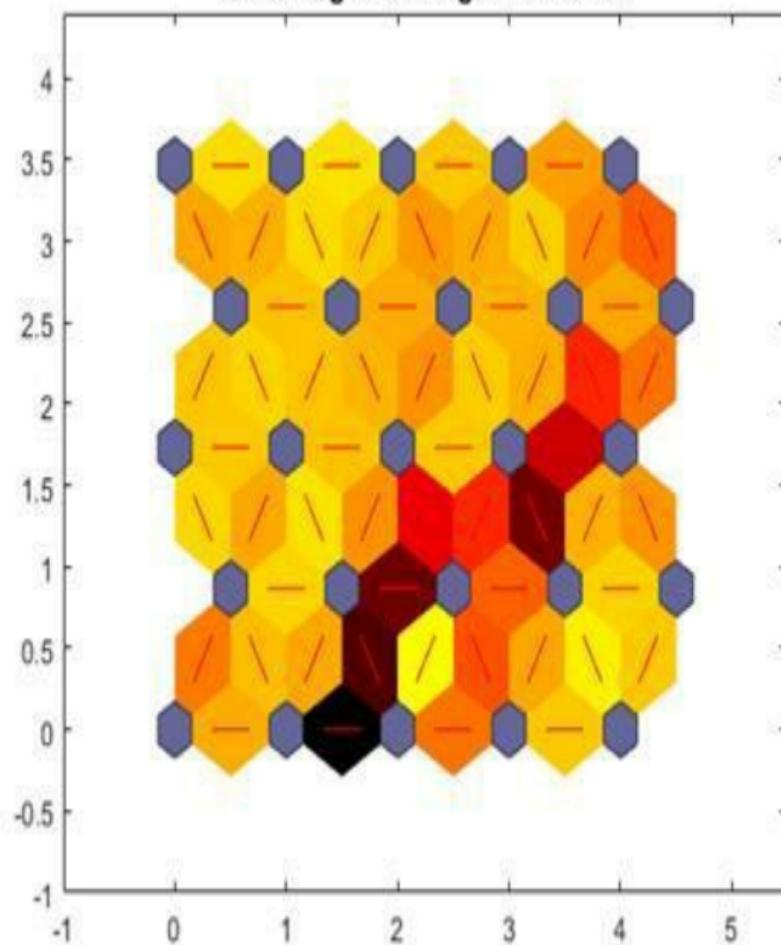
Examples. Plot SOM Neighbor Distances

```
x = iris_dataset;  
  
net = selforgmap([5  
5]);
```

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

```
plotsomnd(net)
```

SOM Neighbor Weight Distances



2.7.4 plotsomplanes

Plot self-organizing map weight planes

Syntax

```
plotsomplanes (net)
```

Description

`plotsomplanes (net)` generates a set of subplots. Each i th subplot shows the weights from the i th input to the layer's neurons, with the most negative connections shown as blue, zero connections as black, and the strongest positive connections as red.

The plot is only shown for layers organized in one or two dimensions.

This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

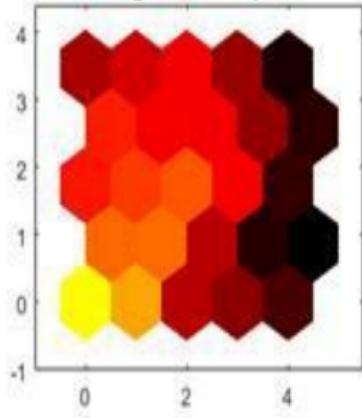
This function can also be called with standardized plotting function arguments used by the function [`train`](#).

Examples. Plot SOM Weight Planes

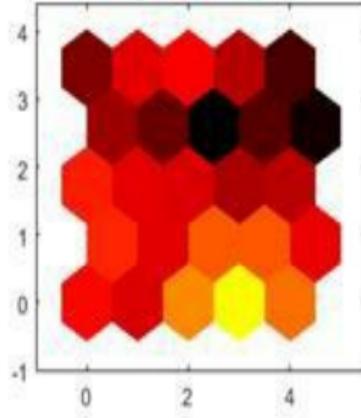
```
x = iris_dataset;
```

```
net = selforgmap([5  
5]);  
  
net = train(net,x);  
  
plotsomplanes(net)
```

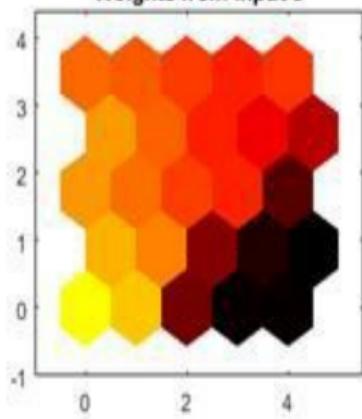
Weights from Input 1



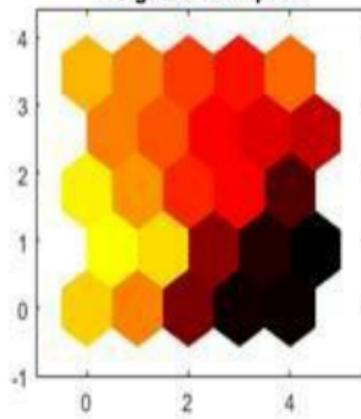
Weights from Input 2



Weights from Input 3



Weights from Input 4



2.7.5 plotsompos

Plot self-organizing map weight positions

Syntax

`plotsompos(net)`

`plotsompos(net, inputs)`

Description

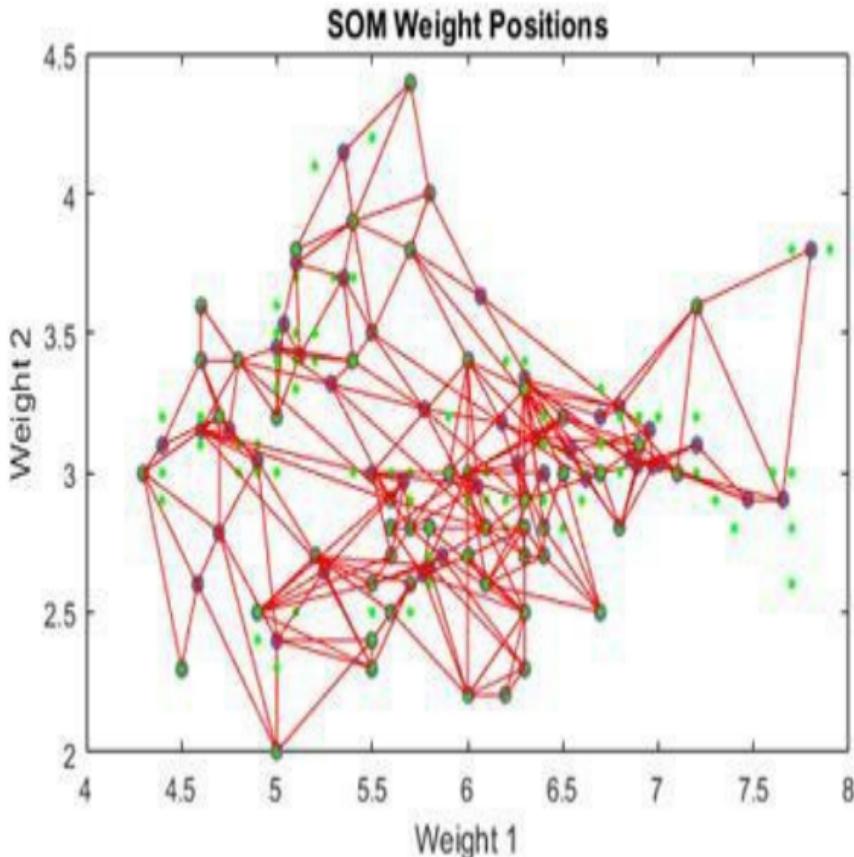
`plotsompos(net)` plots the input vectors as green dots and shows how the SOM classifies the input space by showing blue-gray dots for each neuron's weight vector and connecting neighboring neurons with red lines.

`plotsompos(net, inputs)` plots the input data alongside the weights.

Examples. Plot SOM Weight Positions

```
x = iris_dataset;  
net = selforgmap([10  
10]);  
net = train(net,x);
```

`plotsompos(net, x)`



2.7.6 plotsomtop

Plot self-organizing map topology

Syntax

```
plotsomtop(net)
```

Description

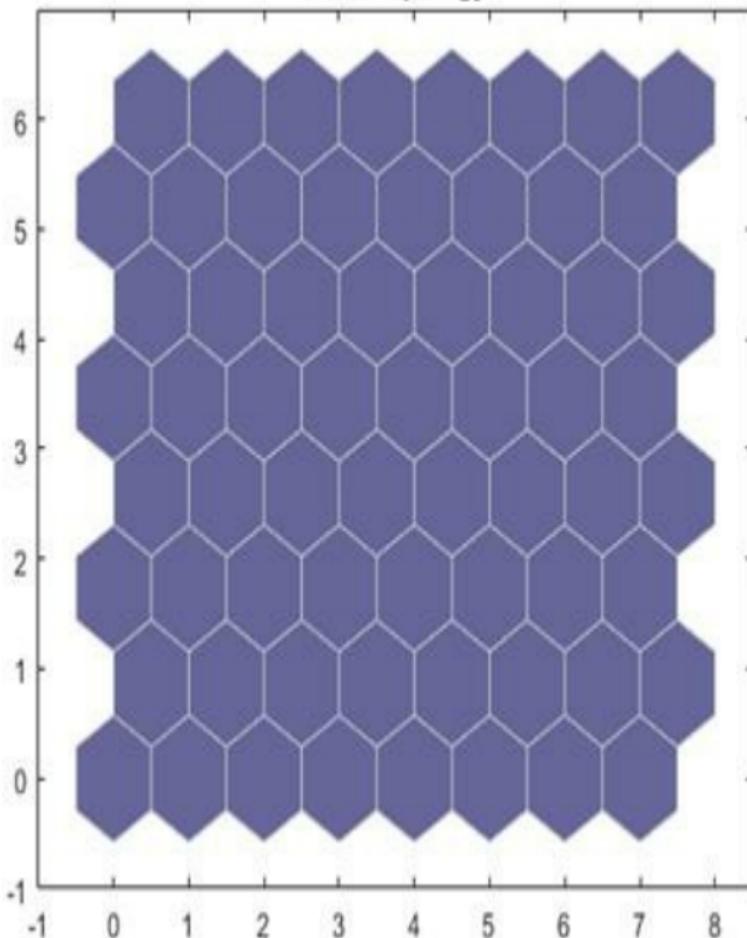
`plotsomtop(net)` plots the topology of a SOM layer.

This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples. Plot SOM Topology

```
x = iris_dataset;  
  
net = selforgmap([8  
8]);  
  
plotsomtop(net)
```

SOM Topology



2.8 A COMPLETE EXAMPLE. IRIS CLUSTERING

This example illustrates how a self-organizing map neural network can cluster iris flowers into classes topologically, providing insight into the types of flowers and a useful tool for further analysis.

In this example we attempt to build a neural network that clusters iris flowers into natural classes, such that similar classes are grouped together. Each iris is described by four features:

- Sepal length in cm

- Sepal width in cm
- Petal length in cm
- Petal width in cm

This is an example of a clustering problem, where we would like to group samples into classes based on the similarity between samples. We would like to create a neural network which not only creates class definitions for the known inputs, but will let us classify unknown inputs accordingly.

2.8.1 Why Self-Organizing Map Neural Networks?

Self-organizing maps (SOMs) are very good at creating classifications. Further, the classifications retain topological information about which classes are most similar to others. Self-organizing maps can be created with any desired level of detail. They are particularly well suited for clustering data in many dimensions and with complexly shaped and connected feature spaces. They are well suited to cluster iris flowers.

The four flower attributes will act as inputs to the SOM, which will map them

onto a 2-dimensional layer of neurons.

2.8.2 Preparing the Data

Data for clustering problems are set up for a SOM by organizing the data into an input matrix X.

Each ith column of the input matrix will have four elements representing the four measurements taken on a single flower.

Here such a dataset is loaded.

```
x = iris_dataset;
```

We can view the size of inputs X.

Note that X has 150 columns. These represent 150 sets of iris flower attributes. It has four rows, for the four

measurements.

size(x)

ans =

4 150

2.8.3 Clustering with a Neural Network

The next step is to create a neural network that will learn to cluster.

selforgmap creates self-organizing maps for classify samples with as much detailed as desired by selecting the number of neurons in each dimension of the layer.

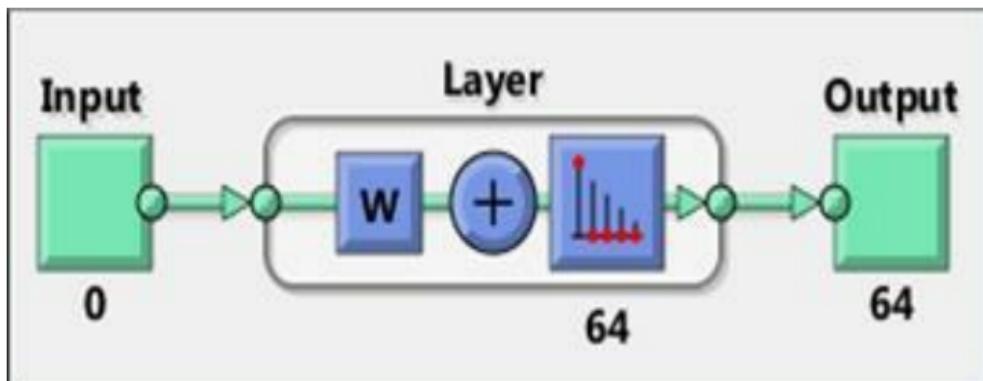
We will try a 2-dimension layer of 64 neurons arranged in an 8x8 hexagonal grid for this example. In general, greater detail is achieved with more neurons, and more dimensions allows for the modelling the topology of more complex

feature spaces.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([8  
8]);
```

```
view(net)
```



Now the network is ready to be optimized with **train**.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

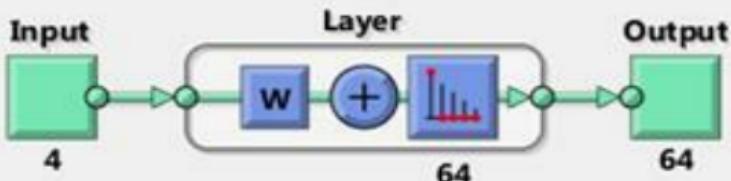
The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net, tr] =
```

```
train(net,x);
```

```
nntraintool
```

Neural Network



Algorithms

Training: Batch Weight/Bias Rules (trainbu)
Performance: Mean Squared Error (mse)
Calculations: MATLAB

Progress

Epoch: 0 200 iterations 200
Time: 0:00:01

Plots

SOM Topology [\(plotsomtop\)](#)

SOM Neighbor Connections [\(plotsomnc\)](#)

SOM Neighbor Distances [\(plotsomnd\)](#)

SOM Input Planes [\(plotsomplanes\)](#)

SOM Sample Hits [\(plotsomhits\)](#)

SOM Weight Positions [\(plotsompos\)](#)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training



Cancel

Here the self-organizing map is used to compute the class vectors of each of the training inputs. These classifications cover the feature space populated by the known flowers, and can now be used to classify new flowers accordingly. The network output will be a 64×150 matrix, where each i th column represents the j th cluster for each i th input vector with a 1 in its j th element.

The function **vec2ind** returns the index of the neuron with an output of 1, for each vector. The indices will range between 1 and 64 for the 64 clusters represented by the 64 neurons.

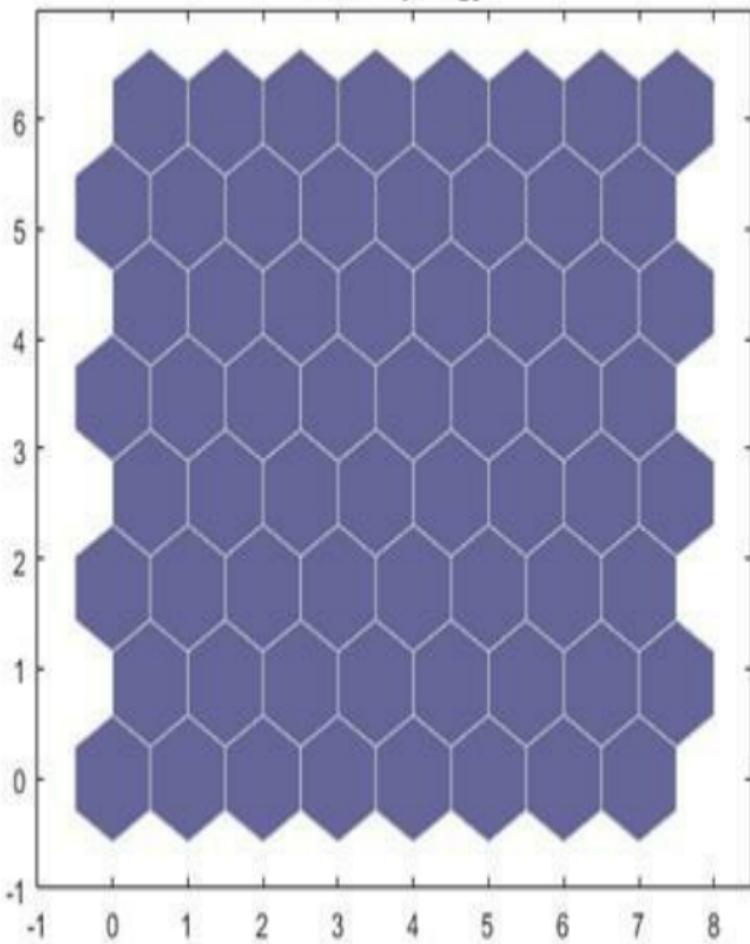
```
y = net(x);
```

```
cluster_index =  
vec2ind(y);
```

plotsomtop plots the self-organizing maps topology of 64 neurons positioned in an 8x8 hexagonal grid. Each neuron has learned to represent a different class of flower, with adjacent neurons typically representing similar classes.

```
plotsomtop(net)
```

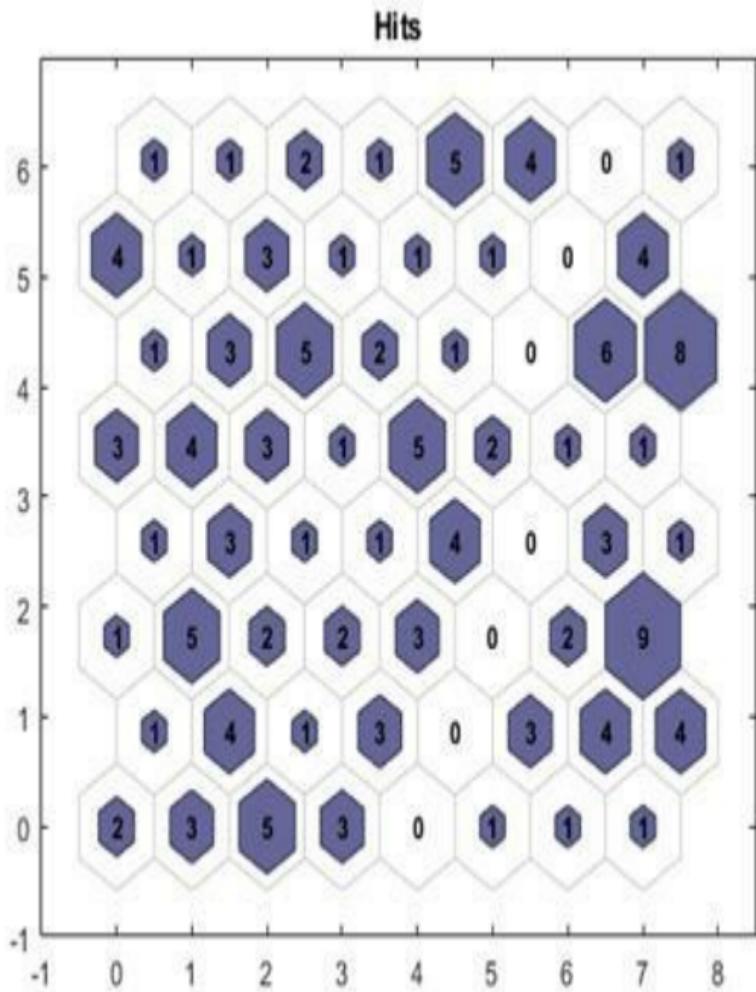
SOM Topology



plotsomhits calculates the classes for each flower and shows the number of

flowers in each class. Areas of neurons with large numbers of hits indicate classes representing similar highly populated regions of the feature space. Whereas areas with few hits indicate sparsely populated regions of the feature space.

plotsomhits(net, x)

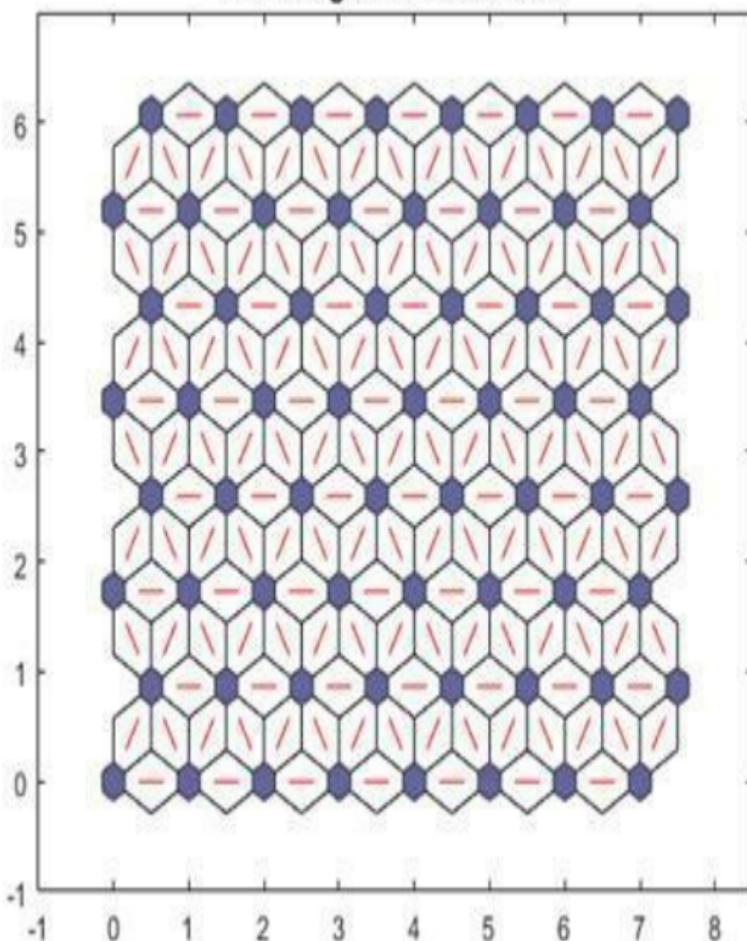


plotsomnc shows the neuron neighbor connections. Neighbors typically

classify similar samples.

`plotsomnc(net)`

SOM Neighbor Connections

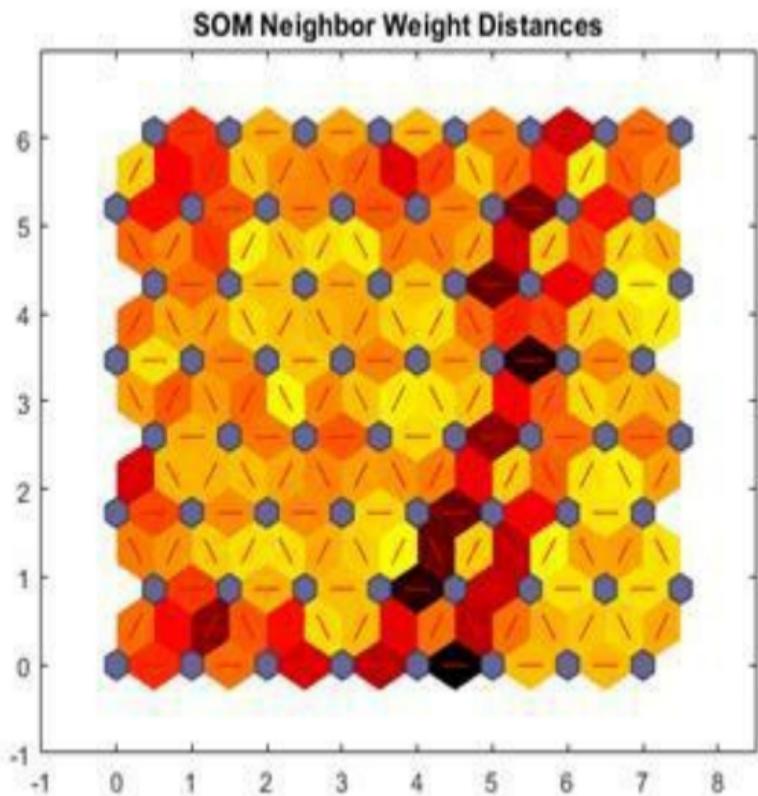


plotsomnd shows how distant (in terms of Euclidian distance) each neuron's

class is from its neighbors. Connections which are bright indicate highly connected areas of the input space. While dark connections indicate classes representing regions of the feature space which are far apart, with few or no flowers between them.

Long borders of dark connections separating large regions of the input space indicate that the classes on either side of the border represent flowers with very different features.

plotsomnd(net)

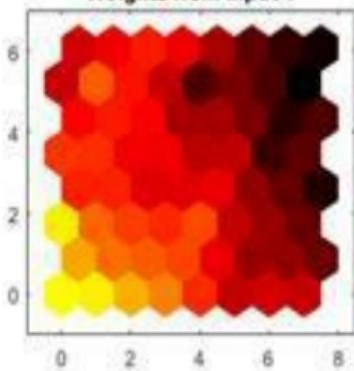


plotsomplanes shows a weight plane for each of the four input features. They are visualizations of the weights that connect each input to each of the 64 neurons in the 8x8 hexagonal grid. Darker colors

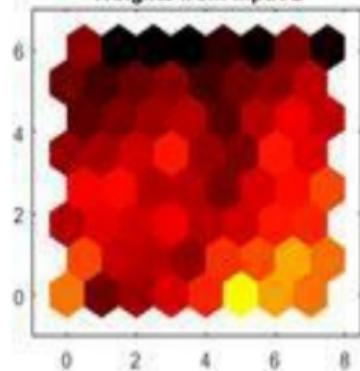
represent larger weights. If two inputs have similar weight planes (their color gradients may be the same or in reverse) it indicates they are highly correlated.

`plotsomplanes (net)`

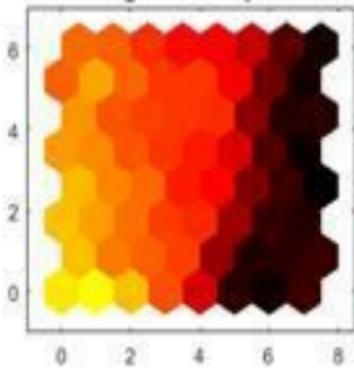
Weights from Input 1



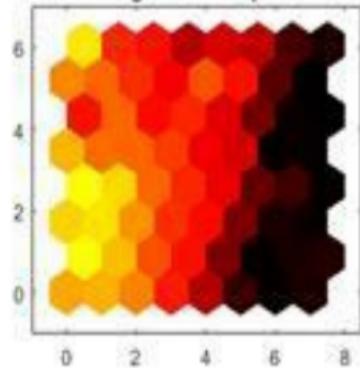
Weights from Input 2



Weights from Input 3



Weights from Input 4



This example illustrated how to design a

neural network that clusters iris flowers based on four of their characteristics.

2.9 GENE EXPRESSION ANALYSIS. CLUSTER ANALYSIS AND PRINCIPAL COMPONENTS

This example demonstrates looking for patterns in gene expression profiles in baker's yeast using neural networks.

2.9.1 The Problem: Analyzing Gene Expressions in Baker's Yeast (*Saccharomyces cerevisiae*)

The goal is to gain some understanding of gene expressions in *Saccharomyces cerevisiae*, which is commonly known as baker's yeast or brewer's yeast. It is the fungus that is used to bake bread and ferment wine from grapes.

Saccharomyces cerevisiae, when introduced in a medium rich in glucose, can convert glucose to ethanol. Initially, yeast converts glucose to ethanol by a

metabolic process called "fermentation". However once supply of glucose is exhausted yeast shifts from anaerobic fermentation of glucose to aerobic respiraton of ethanol. This process is called diauxic shift. This process is of considerable interest since it is accompanied by major changes in gene expression.

The example uses DNA microarray data to study temporal gene expression of almost all genes in *Saccharomyces cerevisiae* during the diauxic shift.

You need Bioinformatics Toolbox™ to run this example.

if ~nnDependency.bioInfoAvailable

```
errordlg('This example requires  
Bioinformatics Toolbox.');
```

return;

```
end
```

2.9.2 The Data

This example uses data from DeRisi, JL, Iyer, VR, Brown, PO. "Exploring the metabolic and genetic control of gene expression on a genomic scale." Science. 1997 Oct 24;278(5338):680-6. PMID: 9381177

The full data set can be downloaded from the Gene Expression Omnibus website: <http://www.yeastgenome.org>

Start by loading the data into MATLAB®.

```
load yeastdata.mat
```

Gene expression levels were measured

at seven time points during the diauxic shift. The variable times contains the times at which the expression levels were measured in the experiment. The variable genes contains the names of the genes whose expression levels were measured.

The

variable yeastvalues contains the "VALUE" data or LOG_RAT2N_MEAN, or log2 of ratio of CH2DN_MEAN and CH1DN_MEAN from the seven time steps in the experiment.

To get an idea of the size of the data you can use **numel(genes)** to show how many genes there are in the data set.

numel(genes)

ans =

6400

genes is a cell array of the gene names.
You can access the entries using
MATLAB cell array indexing:

genes{15}

ans =

YAL054C

This indicates that the 15th row of the
variable **yeastvalues** contains
expression levels for the
ORF YAL054C. You can use the web

command to access information about this ORF in the *Saccharomyces* Genome Database (SGD).

```
url = sprintf(...  
    'http://www.yeastgenome.org/cgi-  
bin/locus.fpl?locus=%s',...  
    genes{15});  
web(url);
```

2.9.3 Filtering the Genes

The data set is quite large and a lot of the information corresponds to genes that do not show any interesting changes during the experiment. To make it easier to find the interesting genes, the first thing to do is to reduce the size of the data set by removing genes with expression profiles that do not show anything of interest. There are 6400 expression profiles. You can use a number of techniques to reduce this to some subset that contains the most significant genes.

If you look through the gene list you will see several spots marked as 'EMPTY'.

These are empty spots on the array, and while they might have data associated with them, for the purposes of this example, you can consider these points to be noise. These points can be found using the **strcmp** function and removed from the data set with indexing commands.

```
emptySpots = strcmp('EMPTY',genes);  
yeastvalues(emptySpots,:)=[];  
genes(emptySpots) = [];  
numel(genes)
```

ans =

6314

In the yeastvalues data you will also see several places where the expression level is marked as NaN. This indicates that no data was collected for this spot at the particular time step. One approach to dealing with these missing values would be to impute them using the mean or median of data for the particular gene over time. This example uses a less rigorous approach of simply throwing away the data for any genes where one or more expression level was not measured.

The function **isnan** is used to identify the genes with missing data and indexing commands are used to remove the genes with missing data.

```
nanIndices = any(isnan(yeastvalues),2);  
yeastvalues(nanIndices,:) = [];  
genes(nanIndices) = [];  
numel(genes)
```

ans =

6276

If you were to plot the expression profiles of all the remaining profiles, you would see that most profiles are flat and not significantly different from the others. This flat data is obviously of use as it indicates that the genes associated with these profiles are not significantly affected by the diauxic shift; however, in this example, you are interested in the

genes with large changes in expression accompanying the diauxic shift. You can use filtering functions in the Bioinformatics Toolbox™ to remove genes with various types of profiles that do not provide useful information about genes affected by the metabolic change.

You can use the **genevarfilter** function to filter out genes with small variance over time. The function returns a logical array of the same size as the variable genes with ones corresponding to rows of yeastvalues with variance greater than the 10th percentile and zeros corresponding to those below the threshold.

```
mask = genevarfilter(yeastvalues);
```

```
% Use the mask as an index into the  
values to remove the filtered genes.  
yeastvalues = yeastvalues(mask,:);  
genes = genes(mask);  
numel(genes)
```

ans =

5648

The function **genelowvalfilter** removes genes that have very low absolute expression values. Note that the gene filter functions can also automatically calculate the filtered data and names.

[mask, yeastvalues, genes] = ...

```
genelowvalfilter(yeastvalues,genes,'absv  
numel(genes)
```

ans =

822

Use **geneentropyfilter** to remove genes whose profiles have low entropy:

```
[mask, yeastvalues, genes] = ...
```

```
geneentropyfilter(yeastvalues,genes,'prct'  
numel(genes)
```

ans =

614

2.9.4 Principal Component Analysis

Now that you have a manageable list of genes, you can look for relationships between the profiles.

Normalizing the standard deviation and mean of data allows the network to treat each input as equally important over its range of values.

Principal-component analysis (PCA) is a useful technique that can be used to reduce the dimensionality of large data sets, such as those from microarray analysis. This technique isolates the principal components of the dataset

eliminating those components that contribute the least to the variation in the data set.

The two settings variables can be used to apply **mapstd** and **processpca** to other data to consistently when the network is applied to new data.

```
[x,std_settings] = mapstd(yeastvalues');  
% Normalize data
```

```
[x,pca_settings] = processpca(x,0.15);  
% PCA
```

The input vectors are first normalized, using **mapstd**, so that they have zero mean and unity variance. **processpca** is the function that implements the PCA algorithm. The second argument passed

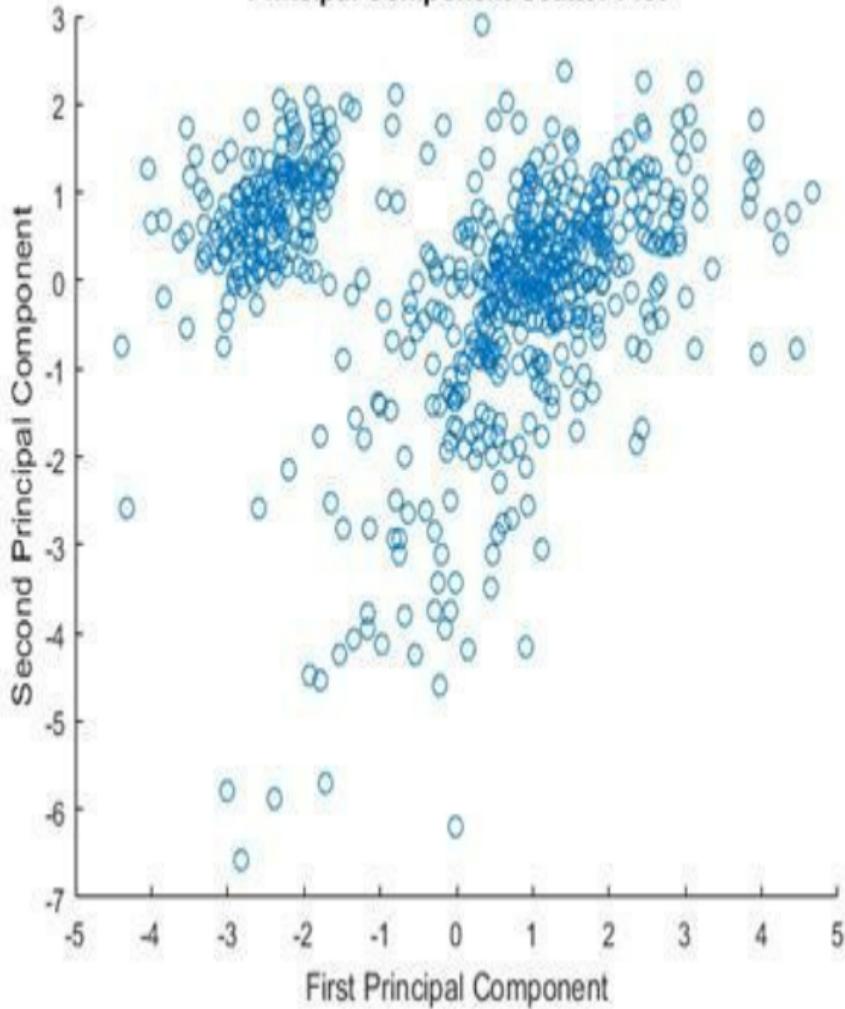
to processpca is 0.15. This means that processpca eliminates those principal components that contribute less than 15% to the total variation in the data set. The variable pc now contains the principal components of the yeastvalues data.

The principal components can be visualized using the **scatter** function.

figure

```
scatter(x(1,:),x(2,:));  
xlabel('First Principal Component');  
ylabel('Second Principal Component');  
title('Principal Component Scatter Plot');
```

Principal Component Scatter Plot



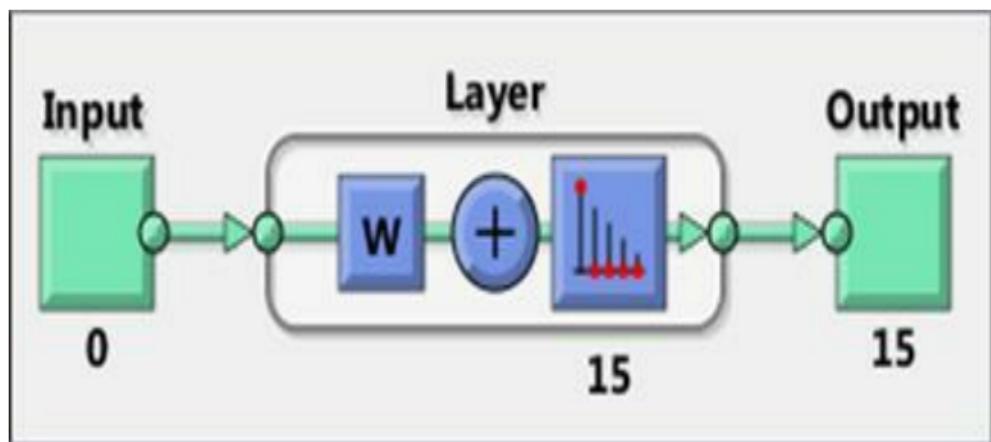
2.9.5 Cluster Analysis using principal components: Self-Organizing Maps

The principal components can be now be clustered using the Self-Organizing map (SOM) clustering algorithm available in Neural Network Toolbox software.

The **selforgmap** function creates a Self-Organizing map network which can then be trained with the **train** function.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([5 3]);  
view(net)
```



Now the network is ready to be trained.

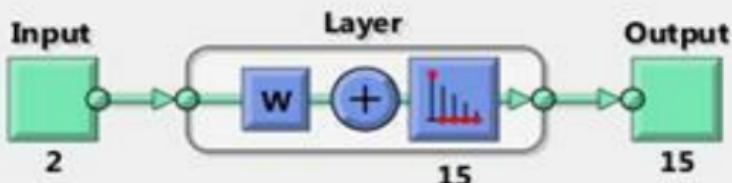
The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful

plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
net = train(net,x);  
nntraintool
```

Neural Network



Algorithms

Training: Batch Weight/Bias Rules (trainbu)
Performance: Mean Squared Error (mse)
Calculations: MATLAB

Progress

Epoch: 0 200 iterations 200
Time: 0:00:00

Plots

SOM Topology [\(plot somtop\)](#)

SOM Neighbor Connections [\(plot somnc\)](#)

SOM Neighbor Distances [\(plot somnd\)](#)

SOM Input Planes [\(plot somplanes\)](#)

SOM Sample Hits [\(plot somhits\)](#)

SOM Weight Positions [\(plot sompos\)](#)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training

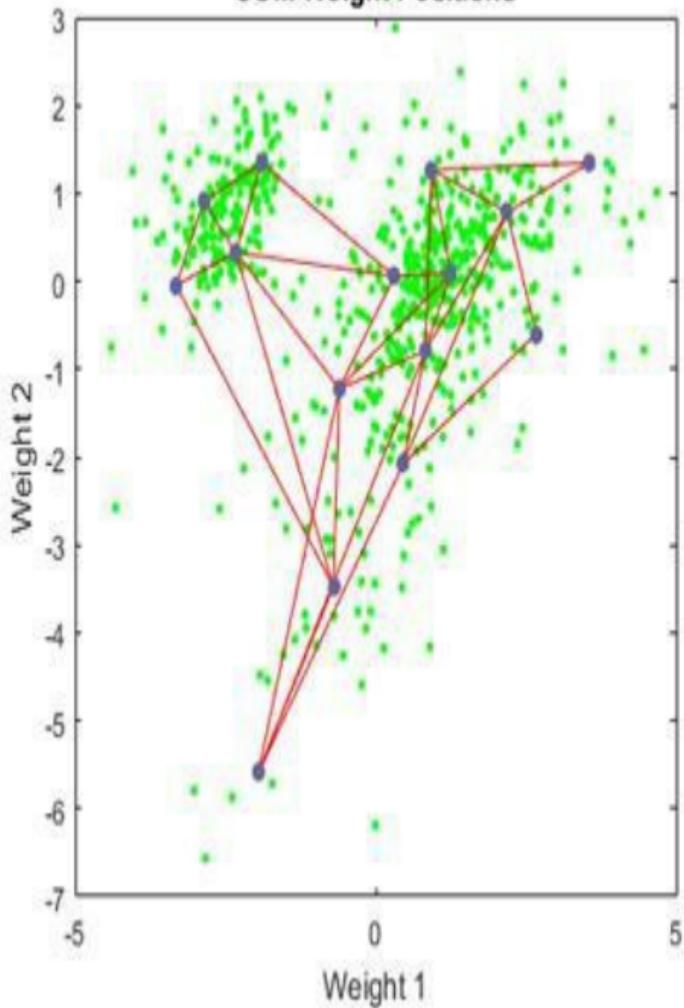


Cancel

Use **plotsompos** to display the network over a scatter plot of the first two dimensions of the data.

```
figure  
plotsompos(net,x);
```

SOM Weight Positions



You can assign clusters using the SOM by finding the nearest node to each point

in the data set.

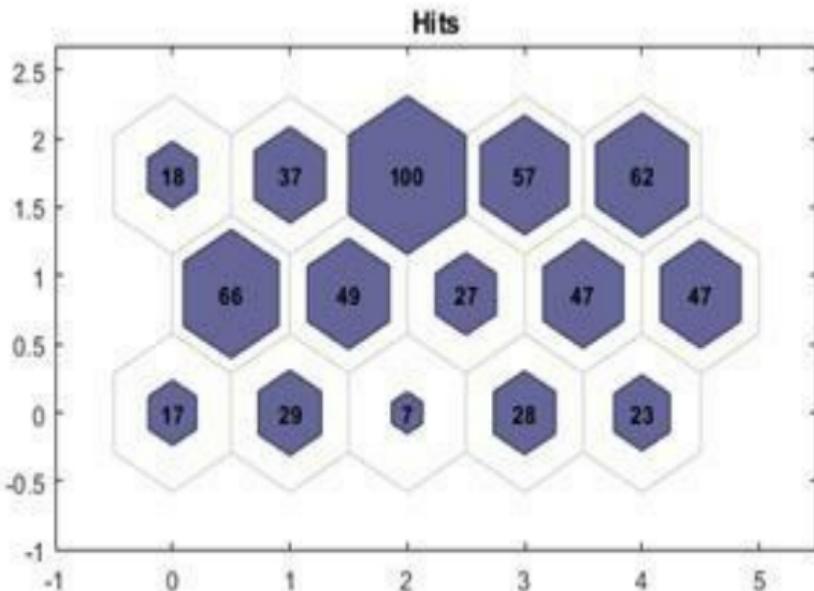
```
y = net(x);
```

```
cluster_indices = vec2ind(y);
```

Use **plotsomhits** to see how many vectors are assigned to each of the neurons in the map.

```
figure
```

```
plotsomhits(net,x);
```



2.10 COMPETITIVE LEARNING

Neurons in a competitive layer learn to represent different regions of the input space where input vectors occur.

P is a set of randomly generated but clustered test data points. Here the data points are plotted.

A competitive network will be used to classify these points into natural classes.

% Create inputs X.

bounds = [0 1; 0 1]; % Cluster centers to be in these bounds.

clusters = 8; % This many clusters.
points = 10; % Number of points

in each cluster.

std_dev = 0.05; % Standard deviation of each cluster.

x =

nngenc(bounds,clusters,points,std_dev);

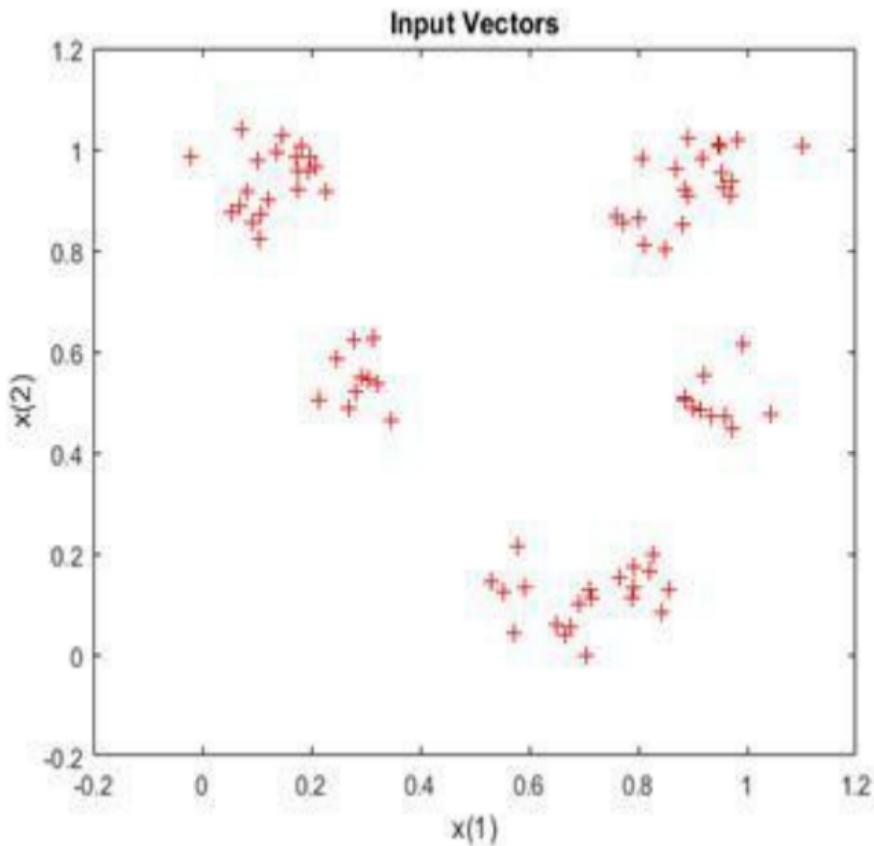
% Plot inputs X.

plot(x(1,:),x(2,:),'+r');

title('Input Vectors');

xlabel('x(1)');

ylabel('x(2)');



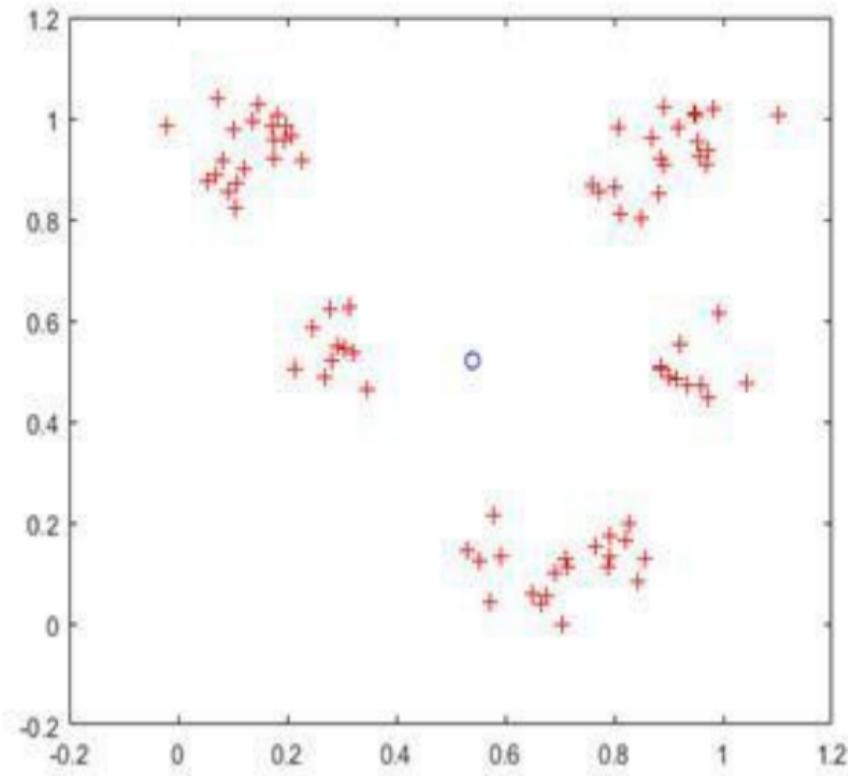
Here COMPETLAYER takes two arguments, the number of neurons and the learning rate.

We can configure the network inputs (normally done automatically by

TRAIN) and plot the initial weight vectors to see their attempt at classification.

The weight vectors (o's) will be trained so that they occur centered in clusters of input vectors (+'s).

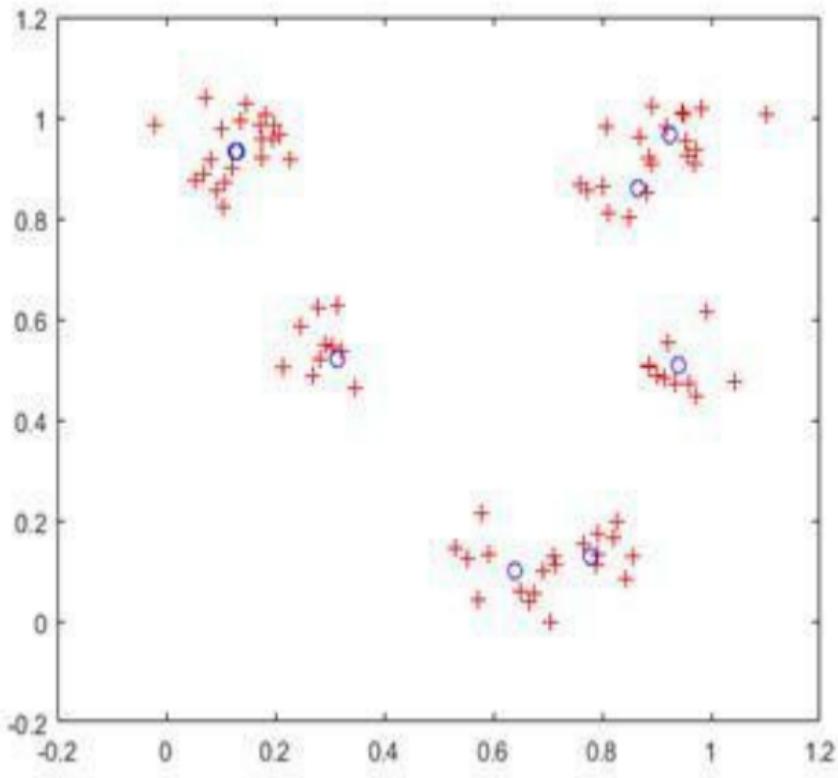
```
net = competlayer(8,.1);
net = configure(net,x);
w = net.IW{1};
plot(x(1,:),x(2,:),'+r');
hold on;
circles = plot(w(:,1),w(:,2),'ob');
```



Set the number of epochs to train before stopping and train this competitive layer (may take several seconds).

Plot the updated layer weights on the same graph.

```
net.trainParam.epochs = 7;  
net = train(net,x);  
w = net.IW{1};  
delete(circles);  
plot(w(:,1),w(:,2),'ob');
```



Now we can use the competitive layer as a classifier, where each neuron corresponds to a different category. Here we define an input vector X_1 as $[0; 0.2]$.

The output Y , indicates which neuron is responding, and thereby which class the input belongs.

$$x_1 = [0; 0.2];$$

$$y = \text{net}(x_1)$$

$$y =$$

0

1

0

0

0

0

0

0

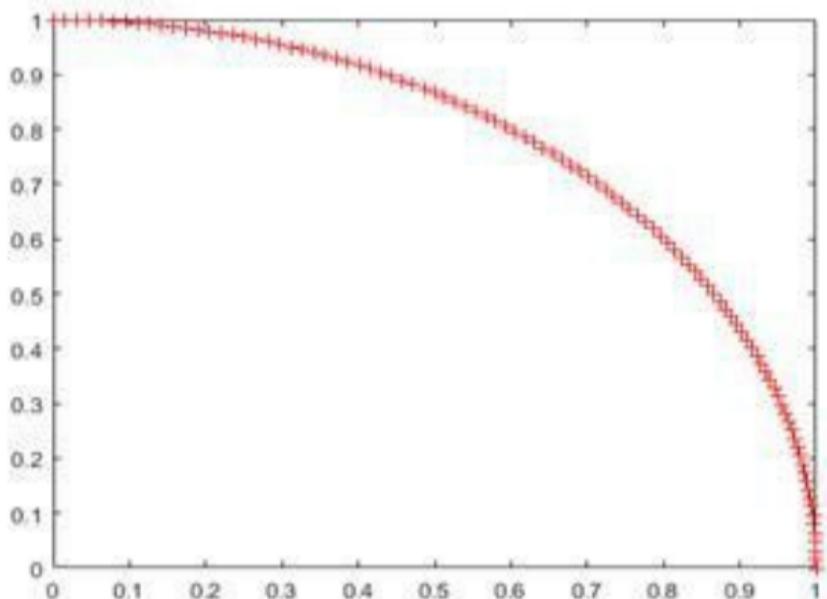
2.11 ONE-DIMENSIONAL SELF-ORGANIZING MAP

Neurons in a 2-D layer learn to represent different regions of the input space where input vectors occur. In addition, neighboring neurons learn to respond to similar inputs, thus the layer learns the topology of the presented input space.

Here 100 data points are created on the unit circle.

A competitive network will be used to classify these points into natural classes.

```
angles = 0:0.5*pi/99:0.5*pi;  
X = [sin(angles); cos(angles)];  
plot(X(1,:),X(2,:),'+r')
```



The map will be a 1-dimensional layer of 10 neurons.

```
net = selforgmap(10);
```

Specify the network is to be trained for

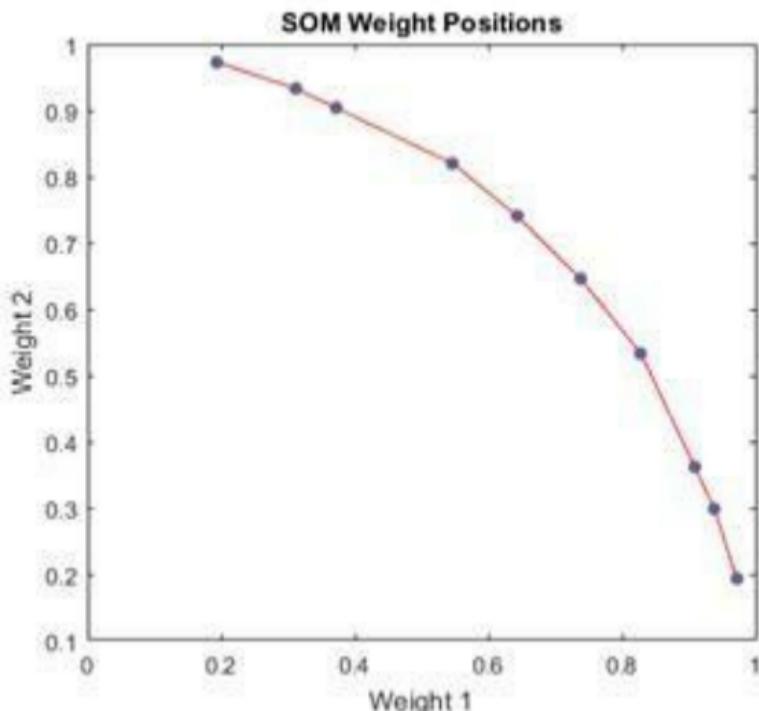
10 epochs and use TRAIN to train the network on the input data P:

```
net.trainParam.epochs = 10;  
net = train(net,X);
```

Now plot the trained network's weight positions with PLOTSOMPOS.

The red dots are the neuron's weight vectors, and the blue lines connect each pair within a distance of 1.

```
plotsompos(net)
```



The map can now be used to classify inputs, like [1; 0]:

Either neuron 1 or 10 should have an output of 1, as the above input vector was at one end of the presented input space. The first pair of numbers indicate the neuron, and the single number

indicates its output.

$x = [1;0];$

$a = \text{net}(x)$

$a =$

1

0

0

0

0

0

0

0

0

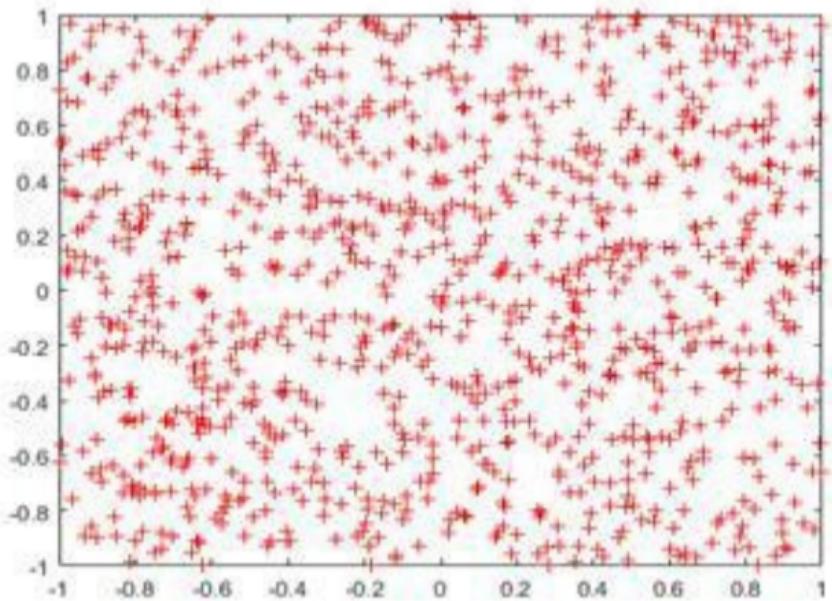
0

2.12 TWO-DIMENSIONAL SELF-ORGANIZING MAP

As in DEMOSM1, this self-organizing map will learn to represent different regions of the input space where input vectors occur. In this example, however, the neurons will arrange themselves in a two-dimensional grid, rather than a line.

We would like to classify 1000 two-element vectors occurring in a rectangular shaped vector space.

```
X = rands(2,1000);  
plot(X(1,:),X(2,:),'r')
```



We will use a 5 by 6 layer of neurons to classify the vectors above. We would like each neuron to respond to a different region of the rectangle, and neighboring neurons to respond to adjacent regions.

The network is configured to match the dimensions of the inputs. This step is

required here because we will plot the initial weights. Normally configuration is performed automatically by TRAIN.

```
net = selforgmap([5 6]);
```

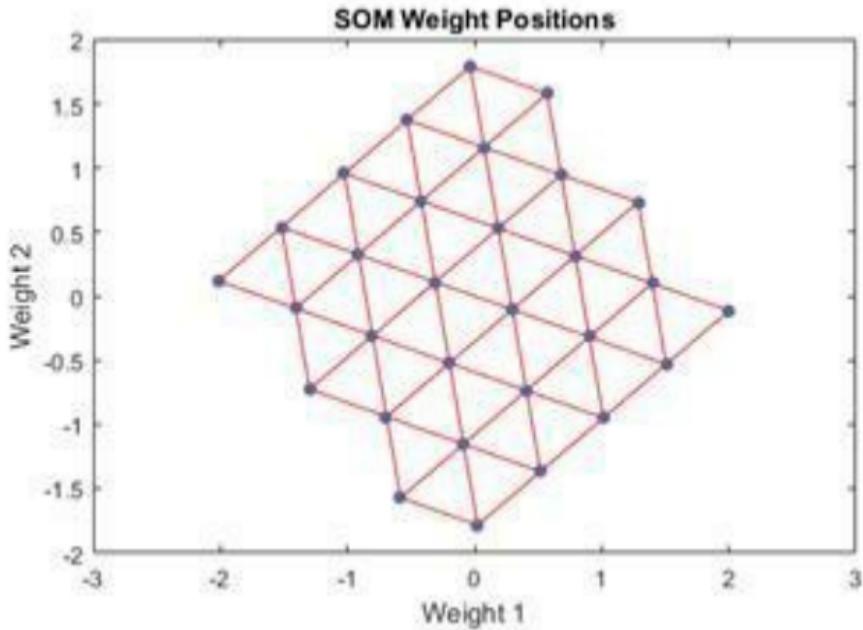
```
net = configure(net,X);
```

We can visualize the network we have just created with PLOTSOMPOS.

Each neuron is represented by a red dot at the location of its two weights.

Initially all the neurons have the same weights in the middle of the vectors, so only one dot appears.

```
plotsompos(net)
```



Now we train the map on the 1000 vectors for 1 epoch and replot the network weights.

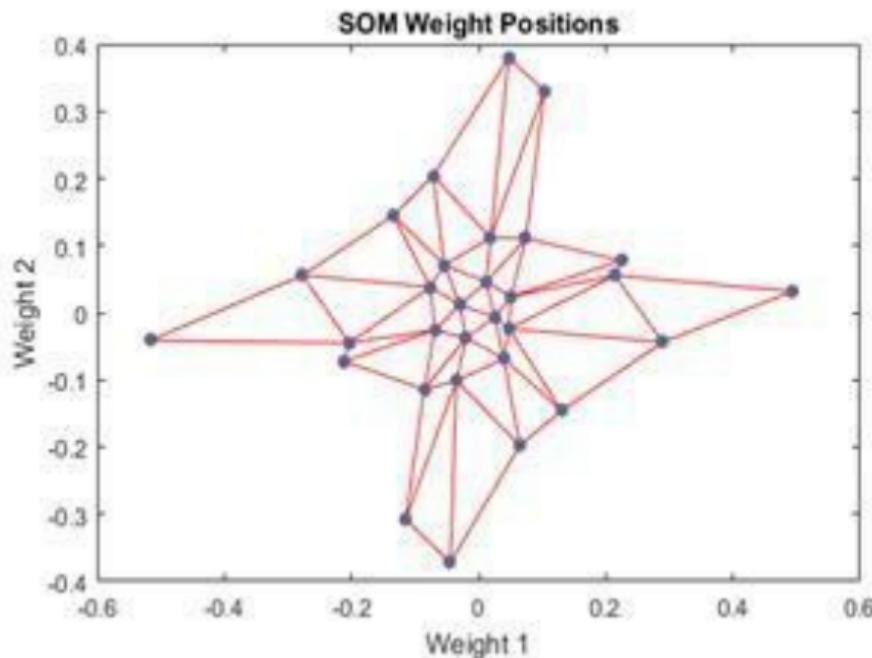
After training, note that the layer of neurons has begun to self-organize so that each neuron now classifies a

different region of the input space, and adjacent (connected) neurons respond to adjacent regions.

```
net.trainParam.epochs = 1;
```

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

```
plotsompos(net)
```



We can now use SIM to classify vectors by giving them to the network and seeing which neuron responds.

The neuron indicated by "a" responded with a "1", so x belongs to that class.

$$x = [0.5; 0.3];$$

$$y = \text{net}(x)$$

$$y =$$

1

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

2.13 CREATE A COMPETITIVE NEURAL NETWORK. BIAS AND KOHONEN LEARNING RULE

You can create a competitive neural network with the function [competlayer](#). A simple example shows how this works.

Suppose you want to divide the following four two-element vectors into two classes.

p = [.1 .8 .1 .9; .2
.9 .1 .8]

p =

0.1000

0.8000 0.1000
0.9000

0.2000

0.9000 0.1000
0.8000

There are two vectors near the origin and two vectors near (1,1).

First, create a two-neuron competitive layer.:

```
net =  
competlayer(2);
```

Now you have a network, but you need to train it to do the classification job.

The first time the network is trained, its weights will initialized to the centers of the input ranges with the function [midpoint](#). You can check see these initial values using the number of

neurons and the input data:

```
wts = midpoint(2,p)
```

```
wts =
```

```
0.5000 0.5000
```

```
0.5000 0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs.

The initial biases are computed by initcon, which gives

```
biases = initcon(2)
```

```
biases =
```

```
5.4366
```

```
5.4366
```

Recall that each neuron competes to respond to an input vector p . If the

biases are all 0, the neuron whose weight vector is closest to \mathbf{p} gets the highest net input and, therefore, wins the competition, and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

2.13.1 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.

2.13.2 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.

2.13.3 Training

Now train the network for 500 epochs.
You can use either [train](#) or [adapt](#).

```
net.trainParam.epoch  
= 500;
```

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

Note that [train](#) for competitive networks uses the training function [trainru](#). You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

```
ans =
```

```
trainru
```

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated after each individual presentation.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

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

```
ac = vec2ind(a)
```

```
ac =
```

```
1 2
```

```
1 2
```

You see that the network is trained to classify the input vectors into two groups, those near the origin, class 1,

and those near (1,1), class 2.

It might be interesting to look at the final weights and biases.

net.IW{1,1}

ans =

0.1000 0.1500

0.8500 0.8500

net.b{1}

ans =

5.4367

5.4365

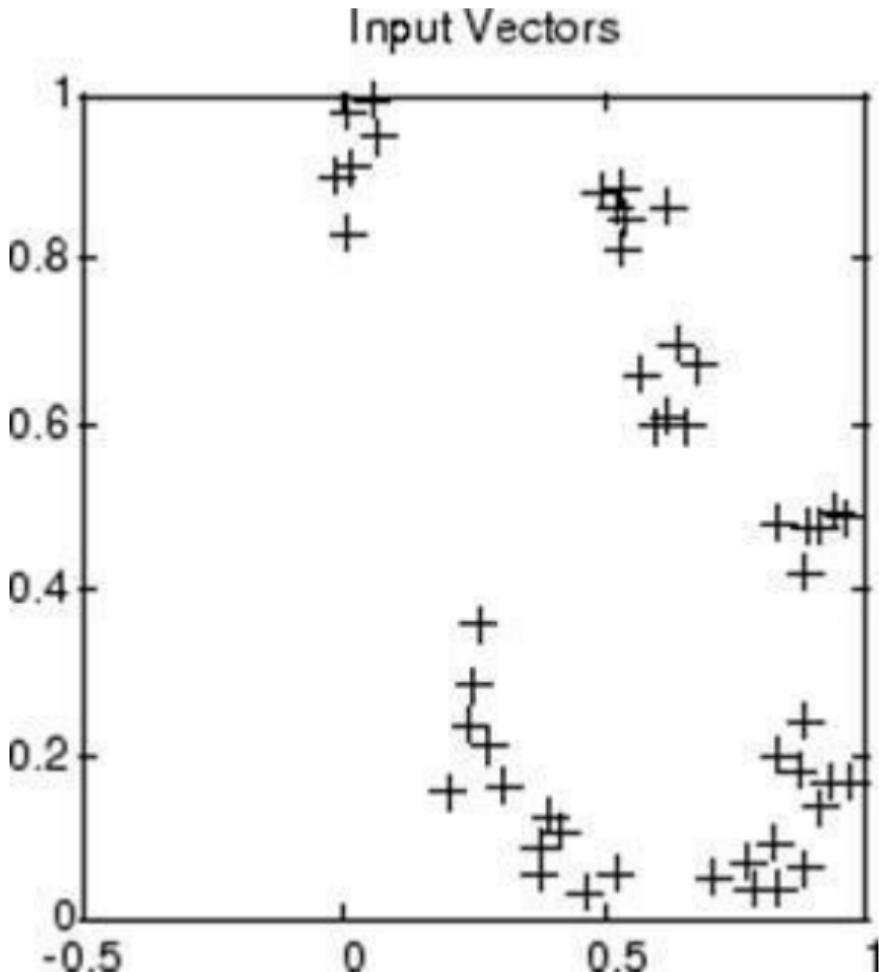
(You might get different answers when you run this problem, because a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to

the origin, while the vector formed from the second row of the weight matrix is close to the input vectors near (1,1). Thus, the network has been trained—just by exposing it to the inputs—to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

2.13.4 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.

2.14 COMPETITIVE LAYERS FUNCTIONS

Identify prototype vectors for clusters of examples using a simple neural network

<u>competlayer</u>	Competitive layer
<u>view</u>	View neural network
<u>train</u>	Train neural network
<u>trainru</u>	Unsupervised random order weight/
<u>learnk</u>	Kohonen weight learning function
<u>learncon</u>	Conscience bias learning function
<u>genFunction</u>	Generate MATLAB function for sim

2.14.1 competlayer

Competitive layer

Syntax

`competlayer(numClasses,kohonenLR)`

Description

Competitive layers learn to classify input vectors into a given number of classes, according to similarity between vectors, with a preference for equal numbers of vectors per class.

`competlayer(numClasses,kohonenLR,con`

these arguments,

numClasses	Number of classes to classify inputs (default = 1)
kohonenLR	Learning rate for Kohonen weights (default = 0.01)
conscienceLR	Learning rate for conscience bias (default = 0.01)

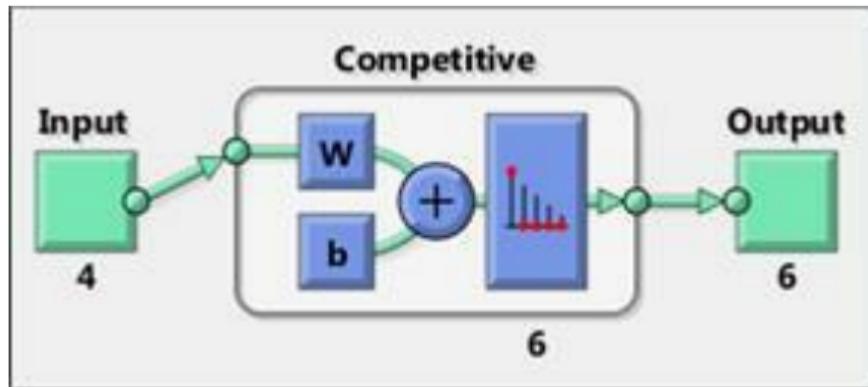
and returns a competitive layer with numClasses neurons.

Examples. Create and Train a Competitive Layer

Here a competitive layer is trained to classify 150 iris flowers into 6 classes.

```
inputs = iris_dataset;
net = competlayer(6);
net = train(net,inputs);
view(net)
```

```
outputs = net(inputs);  
classes = vec2ind(outputs);
```



2.14.2 view

View neural network

Syntax

`view(net)`

Description

`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

Example. View Neural Network

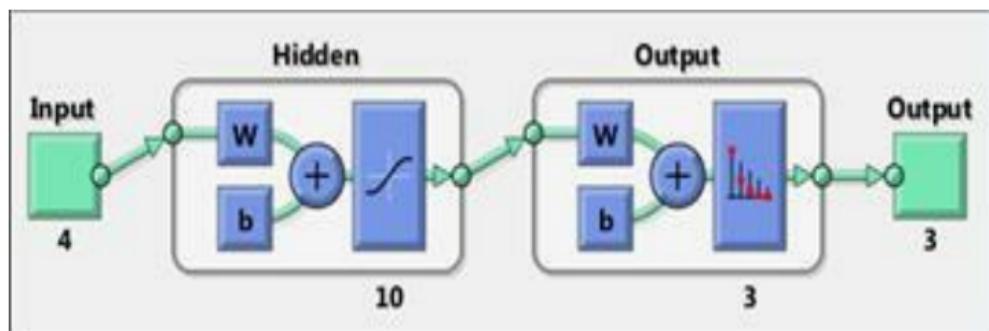
This example shows how to view the diagram of a pattern recognition network.

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

```
net = patternnet;
```

```
net = configure(net,x,t);
```

```
view(net)
```



2.14.3 trainru

Unsupervised random order weight/bias training

Syntax

```
net.trainFcn = 'trainru'  
[net,tr] = train(net,...)
```

Description

trainru is not called directly. Instead it is called by train for networks whose net.trainFcn property is set to 'trainru', thus:

`net.trainFcn = 'trainru'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainru`.

`trainru` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainru` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum num
<code>net.trainParam.show</code>	25	Epochs between
<code>net.trainParam.showCommandLine</code>	false	Generate comm
<code>net.trainParam.showWindow</code>	true	Show training C
<code>net.trainParam.time</code>	Inf	Maximum time

Network Use

To prepare a custom network to be trained with trainru,

1. Set net.trainFcn to 'trainru'. This sets net.trainParam to trainru's default parameters.
2. Set each net.inputWeights {i,j}.learnFcn a learning function.
3. Set each net.layerWeights {i,j}.learnFcn a learning function.
4. Set each net.biases {i}.learnFcn to a learning function. (Weight and bias

learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

Algorithms

For each epoch, all training vectors (or sequences) are each presented once in a

different random order, with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.

2.14.4 learnk

Kohonen weight learning function

Syntax

[dW,LS] =

learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,L
info = learnk('code')

Description

learnk is the Kohonen weight learning function.

[dW,LS] =

learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

W	S-by-R weight matrix (or S-by-1 bias vector)
P	R-by-Q input vectors (or ones(1,Q))
Z	S-by-Q weighted input vectors
N	S-by-Q net input vectors
A	S-by-Q output vectors
T	S-by-Q layer target vectors
E	S-by-Q layer error vectors
gW	S-by-R gradient with respect to performance
gA	S-by-Q output gradient with respect to performance
D	S-by-S neuron distances
LP	Learning parameters, none, LP = []
LS	Learning state, initially should be = []

and returns

dW	S-by-R weight (or bias) change matrix
LS	New learning state

Learning occurs according to learnk's learning parameter, shown here with its

default value.

LP.lr - 0.01	Learning rate
--------------	---------------

`info = learnk('code')` returns useful information for each *code* string:

'pnames'	Names of learning parameters
'pdefaults'	Default learning parameters
'needg'	Returns 1 if this function uses gW or gA

Examples

Here you define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. Also define the learning rate LR.

`p = rand(2,1);`

```
a = rand(3,1);
```

```
w = rand(3,2);
```

```
lp.lr = 0.5;
```

Because learnk only needs these values to calculate a weight change (see "Algorithm" below), use them to do so.

```
dW = learnk(w,p,[],[],a,[],[],[],[],[],[],[],lp,[])
```

Network Use

To prepare the weights of layer i of a custom network to learn with learnk,

1. Set net.trainFcn to 'trainr'.
(net.trainParam automatically

becomes trainr's default parameters.)

2. Set net.adaptFcn to 'trains'.
(net.adaptParam automatically becomes trains's default parameters.)
3. Set
each net.inputWeights {i,j}.learnFcn
4. Set
each net.layerWeights {i,j}.learnFcn
(Each weight learning parameter property is automatically set to learnk's default parameters.)

To train the network (or enable it to adapt),

1. Set net.trainParam (or net.adaptPara properties as desired.
2. Call train (or adapt).

Algorithms

learnk calculates the weight change dW for a given neuron from the neuron's input P , output A , and learning rate LR according to the Kohonen learning rule:

$$dw = lr * (p' - w), \text{ if } a \approx 0; = 0, \text{ otherwise}$$

2.14.5 learncon

Conscience bias learning function

Syntax

[dB,LS] =

learncon(B,P,Z,N,A,T,E,gW,gA,D,LP)
info = learncon('code')

Description

learncon is the conscience bias learning function used to increase the net input to neurons that have the lowest average output until each neuron responds approximately an equal percentage of the time.

[dB,LS] =
learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)
several inputs,

B	S-by-1 bias vector
P	1-by-Q ones vector
Z	S-by-Q weighted input vectors
N	S-by-Q net input vectors
A	S-by-Q output vectors
T	S-by-Q layer target vectors
E	S-by-Q layer error vectors
gW	S-by-R gradient with respect to performance
gA	S-by-Q output gradient with respect to performance
D	S-by-S neuron distances
LP	Learning parameters, none, LP = []
LS	Learning state, initially should be = []

and returns

dB	S-by-1 weight (or bias) change matrix
LS	New learning state

Learning occurs according to learncon's

learning parameter, shown here with its default value.

LP.lr - 0.001	Learning rate
---------------	---------------

`info = learncon('code')` returns useful information for each supported *code* string:

'pnames'	Names of learning parameters
'pdefaults'	Default learning parameters
'needg'	Returns 1 if this function uses gW or gA

Neural Network Toolbox™ 2.0 compatibility: The LP.lr described above equals 1 minus the bias time constant used by trainc in the Neural Network Toolbox 2.0 software.

Examples

Here you define a random output A and bias vector W for a layer with three neurons. You also define the learning rate LR.

```
a = rand(3,1);
```

```
b = rand(3,1);
```

```
lp.lr = 0.5;
```

Because learncon only needs these values to calculate a bias change (see "Algorithm" below), use them to do so.

```
dW = learncon(b,[],[],[],a,[],[],[],[],[],[],[],lp,[])
```

Network Use

To prepare the bias of layer i of a custom network to learn with learncon,

1. Set net.trainFcn to 'trainr'.
(net.trainParam automatically becomes trainr's default parameters.)
2. Set net.adaptFcn to 'trains'.
(net.adaptParam automatically becomes trains's default parameters.)
3. Set net.inputWeights {i}.learnFcn to
4. Set
each net.layerWeights {i,j}.learnFcn.
(Each weight learning parameter

property is automatically set to learncon's default parameters.)

To train the network (or enable it to adapt),

Set net.trainParam (or net.adaptParam) properties as desired.

Call train (or adapt).

Algorithms

learncon calculates the bias change db for a given neuron by first updating each neuron's *conscience*, i.e., the running average of its output:

$$c = (1-lr)*c + lr*a$$

The conscience is then used to compute

a bias for the neuron that is greatest for smaller conscience values.

$$b = \exp(1 - \log(c)) - b$$

(learncon recovers C from the bias values each time it is called.)

Chapter 3

**PATTERN RECOGNITION
AND CLASSIFICATION
WITH NEURAL NETWORKS.
DEEP LEARNING.**

EXAMPLES

3.1 INTRODUCTION

In addition to clustering for classification, neural networks are also good at recognizing patterns with the purpose of classifying.. For example, suppose you want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis, etc. You have 699 example cases for which you have 9 items of data and the correct classification as benign or malignant.

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

- Use the nprtool GUI, as described

in [Using the Neural Network Pattern Recognition Tool](#).

• Use a command-line solution, as described in [Using Command-Line Functions](#).

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. The next section describes the data format. 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 two-class exclusive-or classification problem as follows:

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

Target vectors 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];

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.

3.2 FUNCTIONS FOR PATTERNR RECOGNITION AND CLASSIFICATION. EXAMPLES

The more important functions for pattern recognition and classification are de following:

<u>Autoencoder</u>	Autoencoder class
<u>nnstart</u>	Neural network getting started GL
<u>view</u>	View neural network
<u>trainAutoencoder</u>	Train an autoencoder
<u>trainSoftmaxLayer</u>	Train a softmax layer for classifica
<u>decode</u>	Decode encoded data
<u>encode</u>	Encode input data
<u>predict</u>	Reconstruct the inputs using train
<u>stack</u>	Stack encoders from several autoe
<u>network</u>	Convert Autoencoder object into 1
<u>patternnet</u>	Pattern recognition network

lvqnet	Learning vector quantization neur
train	Train neural network
trainlm	Levenberg-Marquardt backpropaga
trainbr	Bayesian regularization backpropa
trainscg	Scaled conjugate gradient backpro
trainrp	Resilient backpropagation
mse	Mean squared normalized error pe
regression	Linear regression
roc	Receiver operating characteristic
plotconfusion	Plot classification confusion matri
ploterrhist	Plot error histogram
plotperform	Plot network performance
plotregression	Plot linear regression
plotroc	Plot receiver operating characterist
plottrainstate	Plot training state values
crossentropy	Neural network performance
genFunction	Generate MATLAB function for si

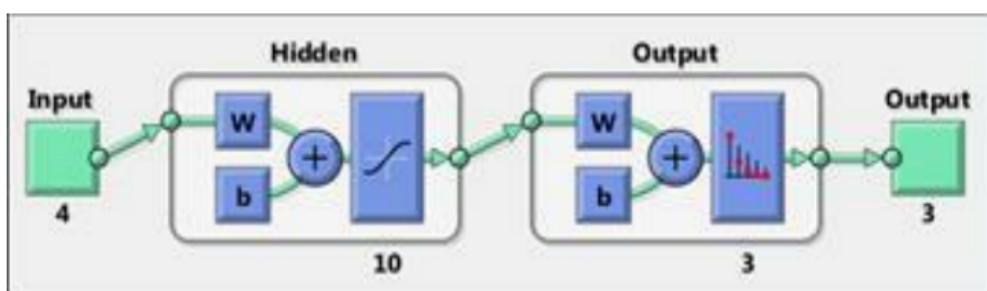
3.3 VIEW NEURAL NETWORK

`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

This example shows how to view the diagram of a pattern recognition network.

```
[x, t] =  
iris_dataset;  
  
net = patternnet;
```

```
net =  
configure(net,x,t);  
  
view(net)
```



3.4 PATTERN RECOGNITION AND LEARNING VECTOR QUANTIZATION

3.4.1 Pattern recognition network: patternnet

Syntax

```
patternnet(hiddenSizes, trainF
```

Description

Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes. The target data for pattern recognition networks should consist of vectors of all zero values except for a 1 in element i , where i is the class they are to represent.

`patternnet(hiddenSizes, trainFcn, ...)`
these arguments,

<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (default = 10)
<code>trainFcn</code>	Training function (default = 'trainscg')
<code>performFcn</code>	Performance function (default = 'crossentropy')

and returns a pattern recognition neural network.

Example of Pattern Recognition

This example shows how to design a pattern recognition network to classify iris flowers.

`[x, t] =`

```
iris_dataset;
```

```
net =  
patternnet(10);
```

```
net =  
train(net,x,t);
```

```
view(net)
```

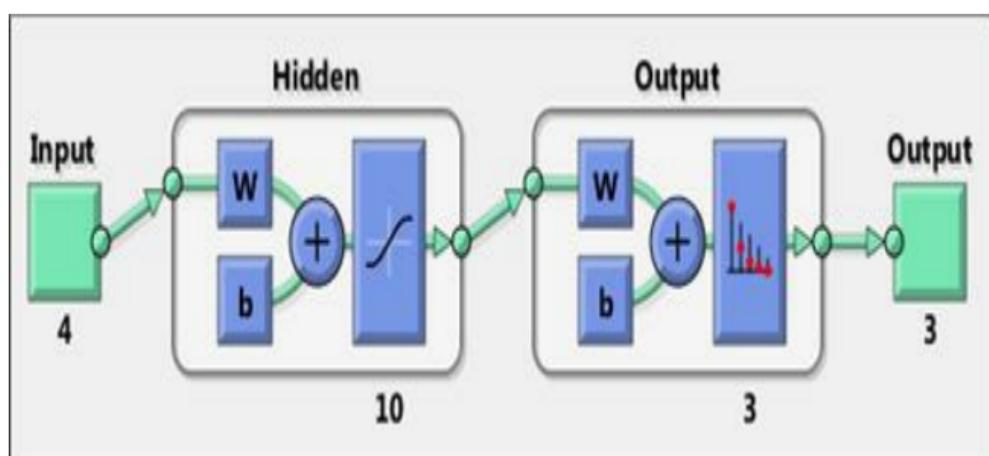
```
y = net(x);
```

```
perf =
```

```
perform(net,t,y);
```

```
classes =
```

```
vec2ind(y);
```



```
net = fitnet(hiddenSizes)
```

```
net =
```

```
fitnet(hiddenSizes,trainFcn)
```

`net = fitnet(hiddenSizes)` returns a function fitting neural network with a hidden layer size of `hiddenSizes` (default=10).

The argument `hiddenSizes` represents the size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network. For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: [10, 8, 5]

`net` =
`fitnet(hiddenSizes, trainFcn)` return a function fitting neural network with a

hidden layer size of `hiddenSizes` and training function, specified by `trainFcn` (`deafut='trainlm'`). The training functions are the following:

Training Function	Algorithm
'trainlm'	Levenberg-Marquardt
'trainbr'	Bayesian Regularization
'trainbfg'	BFGS Quasi-Newton
'trainrp'	Resilient Backpropagation
'trainscg'	Scaled Conjugate Gradient
'traincgb'	Conjugate Gradient with Powell/Beale I
'traincfgf'	Fletcher-Powell Conjugate Gradient
'traincgp'	Polak-Ribière Conjugate Gradient
'trainoss'	One Step Secant
'traingdx'	Variable Learning Rate Gradient Descent
'traingdm'	Gradient Descent with Momentum
'traingd'	Gradient Descent

3.4.2 Learning vector quantization neural network: lvqnet

Syntax

```
lvqnet(hiddenSize, lvqLR, lvqLF)
```

Description

LVQ (learning vector quantization) neural networks consist of two layers. The first layer maps input vectors into clusters that are found by the network during training. The second layer merges groups of first layer clusters into the classes defined by the target data.

The total number of first layer clusters is determined by the number of hidden neurons. The larger the hidden layer the more clusters the first layer can learn, and the more complex mapping of input to target classes can be made. The relative number of first layer clusters assigned to each target class are determined according to the distribution of target classes at the time of network initialization. This occurs when the network is automatically configured the first time [train](#) is called, or manually configured with the function [configure](#), or manually initialized with the function [init](#) is called.

```
lvqnet(hiddenSize, lvqLR, lvqLF) tal
```

these arguments,

hiddenSize	Size of hidden layer (default = 10)
lvqLR	LVQ learning rate (default = 0.01)
lvqLF	LVQ learning function (default = ' learnlv1)

and returns an LVQ neural network.

The other option for the `lvq` learning function is [learnlv2](#).

Example: Train a Learning Vector Quantization Network

Here, an LVQ network is trained to classify iris flowers.

[x, t] =

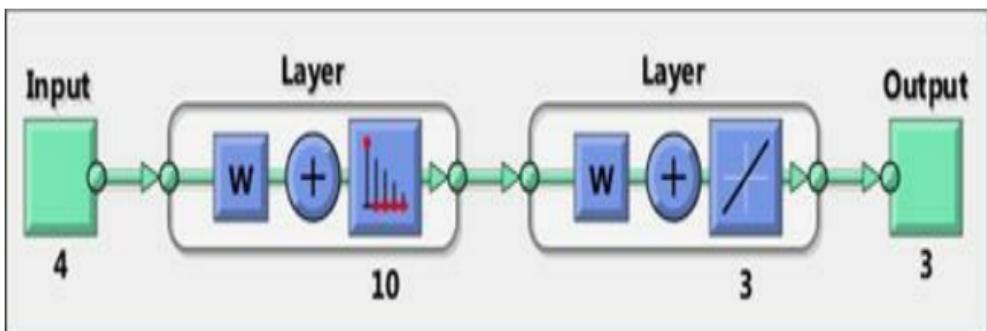
```
iris_dataset;  
  
net = lvqnet(10);  
  
net.trainParam.epoch  
= 50;  
  
net =  
train(net,x,t);  
  
view(net)  
  
y = net(x);
```

```
perf =  
perform(net,y,t)
```

```
classes =  
vec2ind(y);
```

```
perf =
```

0.0489



3.5 TRAINING OPTIONS AND NETWORK PERFORMANCE

The following functions are used to training and network performance.

<u>train</u>	Train neural network
<u>trainlm</u>	Levenberg-Marquardt backpropagation
<u>trainbr</u>	Bayesian regularization backpropagation
<u>trainscg</u>	Scaled conjugate gradient backpropagation
<u>trainrp</u>	Resilient backpropagation
<u>mse</u>	Mean squared normalized error performance
<u>regression</u>	Linear regression
<u>roc</u>	Receiver operating characteristic
<u>plotconfusion</u>	Plot classification confusion matrix
<u>ploterrhist</u>	Plot error histogram
<u>plotperform</u>	Plot network performance
<u>plotregression</u>	Plot linear regression
<u>plotroc</u>	Plot receiver operating characteristic
<u>plottrainstate</u>	Plot training state values
<u>crossentropy</u>	Neural network performance
<u>genFunction</u>	Generate MATLAB function for simulation

3.5.1 Receiver operating characteristic: roc

Syntax

```
[tpr, fpr, thresholds] =  
roc(targets, outputs)
```

Description

The *receiver operating characteristic* is a metric used to check the quality of classifiers. For each class of a classifier, `roc` applies threshold values across the interval $[0, 1]$ to outputs. For each threshold, two values are calculated, the True Positive Ratio

(TPR) and the False Positive Ratio (FPR). For a particular class i , TPR is the number of outputs whose actual and predicted class is class i , divided by the number of outputs whose predicted class is class i . FPR is the number of outputs whose actual class is not class i , but predicted class is class i , divided by the number of outputs whose predicted class is not class i .

You can visualize the results of this function with `plotroc`.

`[tpr, fpr, thresholds] = roc(targets, outputs)` takes these arguments:

targets	S-by-Q matrix, where each column vector contains a single 1 value. The index of the 1 indicates which of S categories that vector represents.
	S-by-Q matrix, where each column contains values in the range [0, 1].

outputs

element in the column indicates which of S categories that vector belongs to. Where values greater or equal to 0.5 indicate class membership, nonmembership.

and returns these values:

tpr	1-by-S cell array of 1-by-N true-positive/positive ratios.
fpr	1-by-S cell array of 1-by-N false-positive/negative ratios.
thresholds	1-by-S cell array of 1-by-N thresholds over interval [0, 1].

`roc(targets, outputs)` takes these arguments:

targets	1-by-Q matrix of Boolean values indicating class membership.
outputs	S-by-Q matrix, of values in [0, 1] interval, where values greater than 0.5 indicate class membership.

and returns these values:

tpr	1-by-N vector of true-positive/positive ratios.
fpr	1-by-N vector of false-positive/negative ratios.
thresholds	1-by-N vector of thresholds over interval [0, 1].

Examples

```
load iris_dataset  
  
net =  
patternnet(20);  
  
net =  
train(net,irisInputs  
irisOutputs =  
sim(net,irisInputs);
```

```
[tpr, fpr, thresholds]
```

```
=
```

```
roc(irisTargets, iris)
```

3.5.2 Plot receiver operating characteristic: plotroc

Syntax

```
plotroc(targets,outputs)
plotroc(targets1,outputs2,'name1')
```

Description

`plotroc(targets,outputs)` plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

```
plotroc(targets1,outputs2,'name1')
```

multiple plots.

Examples: Plot Receiver Operating Characteristic

```
load
```

```
simplecluster_database
```

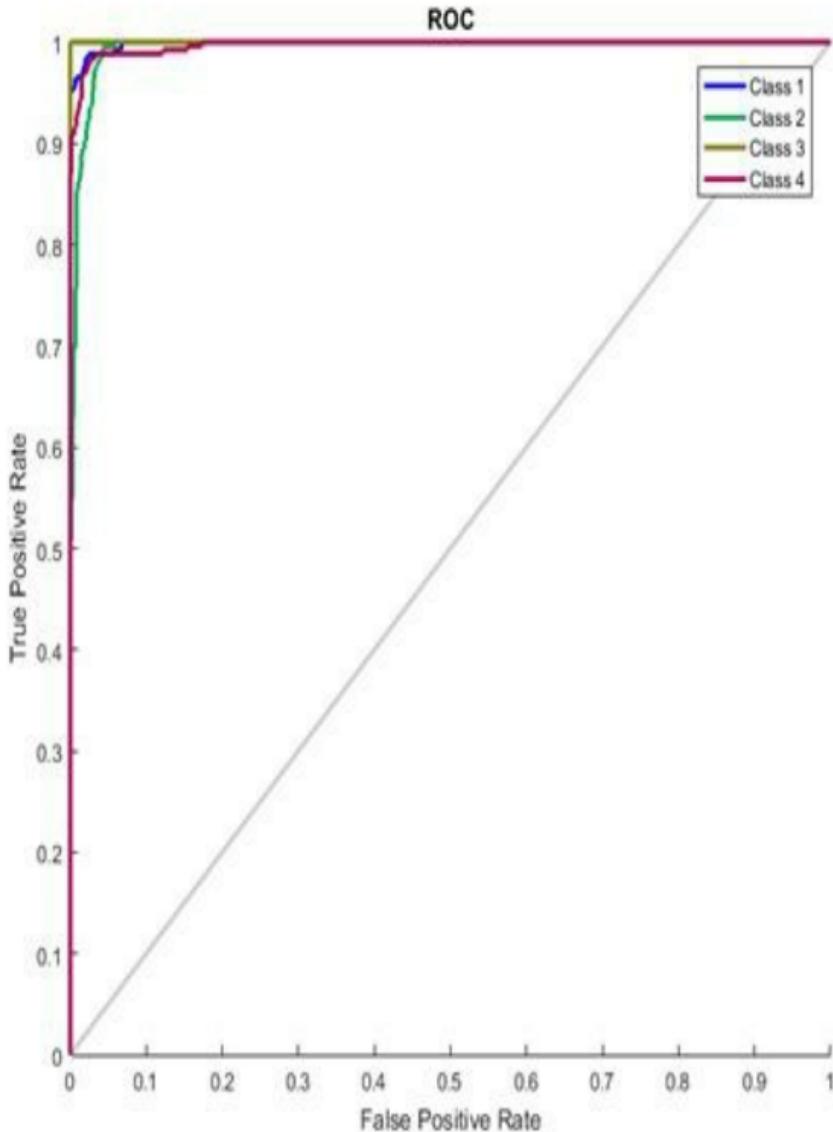
```
net =
```

```
patternnet(20);
```

```
net =
```

```
train(net, simpleclus
```

```
simpleclusterOutputs  
=  
sim(net, simplecluste:  
plotroc(simplecluste:
```



3.5.3 Plot classification confusion matrix: plotconfusion

Syntax

- `plotconfusion(targets, outputs)`
- [example](#)
- `plotconfusion(targets, outputs, title)`
- `plotconfusion(targets1, outputs1, targets2, outputs2)`

Description

`plotconfusion(targets, outputs)` re¹
a confusion matrix plot for the target and

output data in targets and outputs, respectively.

On the confusion matrix plot, the rows correspond to the predicted class (Output Class), and the columns show the true class (Target Class). The diagonal cells show for how many (and what percentage) of the examples the trained network correctly estimates the classes of observations. That is, it shows what percentage of the true and predicted classes match. The off diagonal cells show where the classifier has made mistakes. The column on the far right of the plot shows the accuracy for each predicted class, while the row at the bottom of the plot shows the

accuracy for each true class. The cell in the bottom right of the plot shows the overall accuracy.

`plotconfusion(targets, outputs, name)` creates a confusion matrix plot with the title starting with `name`.

`plotconfusion(targets1, outputs1, ..., name1)` creates several confusion plots in one figure, and prefixes the `name` arguments to the titles of the appropriate plots.

Examples: Plot Confusion Matrix

This example shows how to train a pattern recognition network and plot its

accuracy.

Load the sample data.

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

cancerInputs is a 9x699 matrix defining nine attributes of 699 biopsies. cancerTargets is a 2x966 matrix where each column indicates a correct category with a one in either element 1 (benign) or element 2 (malignant). For more information on this dataset, type help cancer_dataset in the

command line.

Create a pattern recognition network and train it using the sample data.

```
net =  
patternnet(10);
```

```
net =  
train(net,x,t);
```

Estimate the cancer status using the trained network, net .

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

Plot the confusion matrix.

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

Confusion Matrix

		1	2	
		1	2	
Output Class	1	446 63.8%	5 0.7%	98.9% 1.1%
	2	12 1.7%	236 33.8%	95.2% 4.8%
	1	97.4% 2.6%	97.9% 2.1%	97.6% 2.4%
	2			
Target Class				

In this figure, the first two diagonal cells show the number and percentage of correct classifications by the trained network. For example 446 biopsies are correctly classified as benign. This corresponds to 63.8% of all 699 biopsies. Similarly, 236 cases are correctly classified as malignant. This corresponds to 33.8% of all biopsies.

5 of the malignant biopsies are incorrectly classified as benign and this corresponds to 0.7% of all 699 biopsies in the data. Similarly, 12 of the benign biopsies are incorrectly classified as malignant and this corresponds to 1.7% of all data.

Out of 451 benign predictions, 98.9% are correct and 1.1% are wrong. Out of 248 malignant predictions, 95.2% are correct and 4.8% are wrong. Out of 458 benign cases, 97.4% are correctly predicted as benign and 2.6% are predicted as malignant. Out of 241 malignant cases, 97.9% are correctly classified as malignant and 2.1% are classified as benign.

Overall, 97.6% of the predictions are correct and 2.4% are wrong classifications.

3.5.4 Neural network performance: crossentropy

Syntax

- `perf = crossentropy(net,targets,outputs)`
- `perf = crossentropy(__,Name,Value)`

Description

perf =
`crossentropy(net,targets,outputs)`
a network performance given targets

and outputs, with optional performance weights and other parameters. The function returns a result that heavily penalizes outputs that are extremely inaccurate (y near $1-t$), with very little penalty for fairly correct classifications (y near t). Minimizing cross-entropy leads to good classifiers.

The cross-entropy for each pair of output-target elements is calculated as:

$$ce = -t .* \log(y).$$

The aggregate cross-entropy performance is the mean of the individual values:

$$perf = \text{sum}(ce(:)) / \text{numel}(ce).$$

Special case ($N = 1$): If an output consists of only one element, then the

outputs and targets are interpreted as binary encoding. That is, there are two classes with targets of 0 and 1, whereas in 1-of-N encoding, there are two or more classes. The binary cross-entropy expression is:

$$ce = -t \cdot \log(y) - (1-t) \cdot \log(1-y).$$

perf =

`crossentropy(__, Name, Value)` supports customization according to the specified name-value pair arguments.

Examples: Calculate Network Performance

This example shows how to design a

classification network with cross-entropy and 0.1 regularization, then calculation performance on the whole dataset.

```
[x, t] =
```

```
iris_dataset;
```

```
net =
```

```
patternnet(10);
```

```
net.performParam.reg  
= 0.1;
```

```
net =  
train(net,x,t);
```

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

```
perf =  
crossentropy(net,t,y  
{1}, 'regularization'
```

```
perf =
```

0 . 0278

3.6 AUTOENCODER CLASS. DEEP LEARNING

Description

An Autoencoder object contains an autoencoder network, which consists of an encoder and a decoder. The encoder maps the input to a hidden representation. The decoder attempts to map this representation back to the original input.

Construction

```
autoenc =
```

trainAutoencoder (X) returns an autoencoder trained using the training data in x.

```
autoenc =
```

trainAutoencoder (X, hiddenSize) creates an autoencoder with the hidden representation size of hiddenSize.

```
autoenc =
```

trainAutoencoder (, Name, Value) creates any of the above input arguments with additional options specified by one or more Name, Value pair arguments.

Input Arguments

x — training data

matrix | cell array of image data

Hiddensize — size of hidden representation of the autoencoder
10 (default) | positive integer value

Methods

decode	Decode encoded data
encode	Encode input data
generateFunction	Generate a MATLAB function to run the autoencoder
generateSimulink	Generate a Simulink model for the autoencoder
network	Convert Autoencoder object into network
plotWeights	Plot a visualization of the weights for the encoder
predict	Reconstruct the inputs using trained autoencoder
stack	Stack encoders from several autoencoders together
view	View autoencoder

3.6.1 trainAutoencoder

Train an autoencoder

Syntax

- `autoenc = trainAutoencoder(X)`
- `autoenc = trainAutoencoder(X,hiddenSize)`
- `autoenc = trainAutoencoder(__,Name,Value)`

Description

`autoenc` = `trainAutoencoder(X)` returns an

autoencoder, `autoenc`, trained using the training data in `x`.

`autoenc` =

`trainAutoencoder(x, hiddenSize)` re1
an autoencoder `autoenc`, with the hidden representation size of `hiddenSize`.

`autoenc` =

`trainAutoencoder(_____, Name, Value)` r
an autoencoder `autoenc`, for any of the above input arguments with additional options specified by one or more `Name`, `Value` pair arguments.

For example, you can specify the sparsity proportion or the maximum number of training iterations.

Examples. Train Sparse Autoencoder

Load the sample data.

```
X = abalone_dataset;
```

x is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in

the command line.

Train a sparse autoencoder with default settings.

```
autoenc =
```

```
trainAutoencoder(X);
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed =
```

```
predict(autoenc, X);
```

Compute the mean squared reconstruction error.

```
mseError = mse(X -  
XReconstructed)
```

```
mseError =
```

```
0.0167
```

Train Autoencoder with Specified Options

Load the sample data.

```
X = abalone_dataset;  
x is an 8-by-4177 matrix defining eight  
attributes for 4177 different abalone  
shells: sex (M, F, and I (for infant)),  
length, diameter, height, whole weight,  
shucked weight, viscera weight, shell  
weight. For more information on the  
dataset, type help abalone_dataset in  
the command line.
```

Train a sparse autoencoder with hidden
size 4, 400 maximum epochs, and linear
transfer function for the decoder.

```
autoenc =  
trainAutoencoder(X, 4  
  
'DecoderTransferFunc'
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed =  
predict(autoenc, X);  
Compute the mean squared  
reconstruction error.
```

```
mseError = mse(X-  
XReconstructed)
```

```
mseError =
```

0.0056

**Reconstruct Observations
Using Sparse Autoencoder**

Generate the training data.

```
rng(0, 'twister'); %  
For reproducibility  
  
n = 1000;  
  
r =  
linspace(-10, 10, n)';  
  
x = 1 + r*5e-2 +  
sin(r)./r +
```

```
0.2*randn(n,1);
```

Train autoencoder using the training data.

```
hiddenSize = 25;
```

```
autoenc =
```

```
trainAutoencoder(x',
```

```
'EncoderTransferFunc'
```

```
' DecoderTransferFunc'
' L2WeightRegularizat.
' SparsityRegularizat.
' SparsityProportion'
Generate the test data.
```

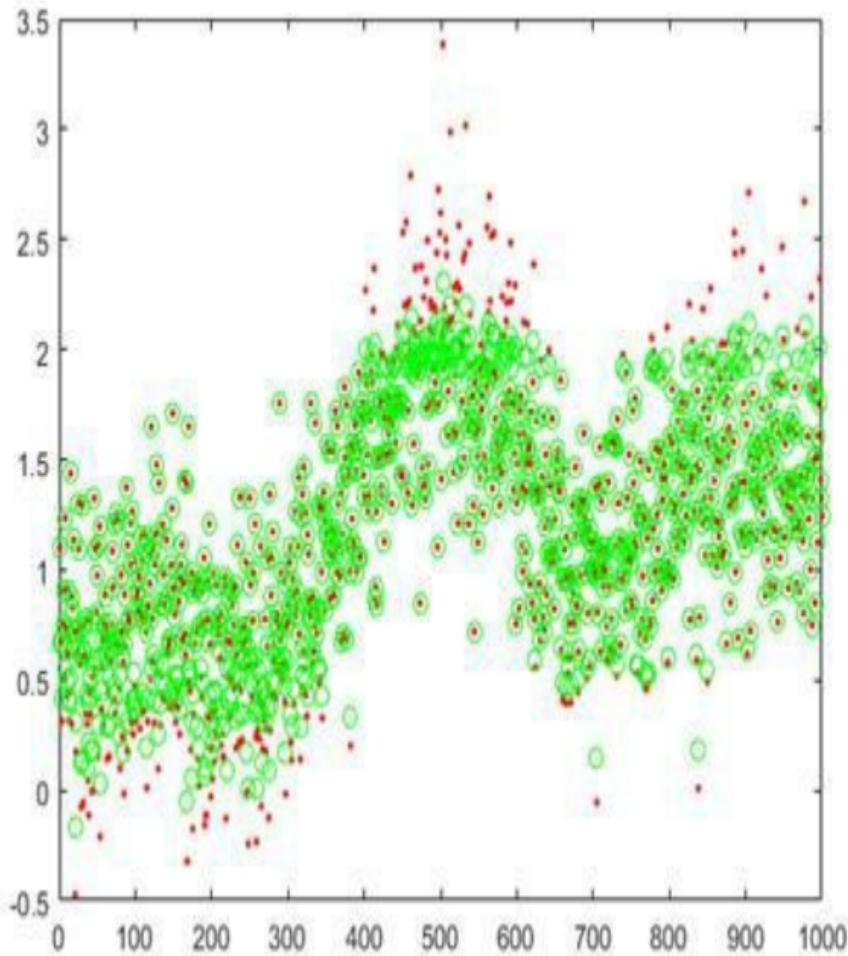
```
n = 1000;
```

```
r = sort(-10 +  
20*rand(n,1));
```

```
xtest = 1 + r*5e-2 +  
sin(r)./r +  
0.4*randn(n,1);
```

Predict the test data using the trained autoencoder, autoenc.

```
xReconstructed =  
predict(autoenc,xtes);  
Plot the actual test data and the  
predictions.  
  
figure;  
  
plot(xtest,'r.');//  
hold on  
  
plot(xReconstructed,
```



**Reconstruct Handwritten
Digit Images Using Sparse**

Autoencoder

Load the training data.

```
X =  
digittrain_dataset;
```

The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.

```
hiddenSize = 25;
```

```
autoenc =
```

```
trainAutoencoder(X, h
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

'SparsityProportion'

Load the test data.

```
x =  
digittest_dataset;
```

The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the

trained autoencoder, autoenc.

```
xReconstructed =  
predict(autoenc,x);  
View the actual test data.  
  
figure;  
  
for i = 1:20  
  
    subplot(4,5,i);
```

```
imshow(X{i});
```

```
end
```

View the reconstructed test data.

```
figure;
```

```
for i = 1:20
```

```
    subplot(4,5,i);
```

```
    imshow(xReconstructed{i});
```

```
end
```

3 .6 .2 Construct Deep Network Using Autoencoders

Load the sample data.

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

Train an autoencoder with a hidden layer of size 10 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
hiddenSize = 10;
```

```
autoenc1 =
```

```
trainAutoencoder(X, h
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

```
'SparsityProportion'  
'DecoderTransferFunc'
```

Extract the features in the hidden layer.

```
features1 =  
encode(autoenc1, X);
```

Train a second autoencoder using the features from the first autoencoder. Do not scale the data.

```
hiddenSize = 10;
```

```
autoenc2 =
```

```
trainAutoencoder (fea
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

```
'SparsityProportion'  
'DecoderTransferFunc'  
  
'ScaleData', false);  
Extract the features in the hidden layer.
```

```
features2 =  
encode(autoenc2, feat  
Train a softmax layer for classification  
using the features, features2, from the  
second autoencoder, autoenc2.
```

```
softnet =
```

```
trainSoftmaxLayer(fe,
```

Stack the encoders and the softmax layer
to form a deep network.

```
deepnet =
```

```
stack(autoencl, autoe:
```

Train the deep network on the wine data.

```
deepnet =
```

```
train(deepnet, X, T);
```

Estimate the wine types using the deep
network, deepnet.

```
wine_type =  
deepnet(X);
```

Plot the confusion matrix.

```
plotconfusion(T,wine_
```

Confusion Matrix

		Target Class			
		1	2	3	4
Output Class	1	59 33.1%	0 0.0%	0 0.0%	100% 0.0%
	2	0 0.0%	71 39.9%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	48 27.0%	100% 0.0%	
4	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	

3.6.3 decode

Decode encoded data

Syntax

- $\text{Y} = \text{decode}(\text{autoenc}, \text{Z})$

Description

Y = `decode(autoenc, Z)` returns the decoded data Y , using the autoencoder object `autoenc`.

Trained autoencoder, returned by the `trainAutoencoder` function as an object of the `Autoencoder` class.

Data encoded by `autoenc`, specified as a matrix. Each column of `z` represents an encoded sample (observation).

Decoded data, returned as a matrix or a cell array of image data.

If the autoencoder `autoenc` was trained on a cell array of image data, then `y` is also a cell array of images.

If the autoencoder `autoenc` was trained on a matrix, then `y` is also a matrix, where each column of `y` corresponds to one sample or observation.

Example: Decode Encoded

Data For New Images

Load the training data.

X =

digitTrainCellArrayD.

x is a 1-by-5003 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder using the training data with a hidden size of 15.

```
hiddenSize = 15;
```

```
autoenc =
```

```
trainAutoencoder(X, h)
```

Extract the encoded data for new images
using the autoencoder.

```
Xnew =
```

```
digitTestCellArrayData
```

```
features =  
encode(autoenc, Xnew)
```

Decode the encoded data from the autoencoder.

```
Y =  
decode(autoenc, features)  
Y is a 1-by-4997 cell array, where each  
cell contains a 28-by-28 matrix  
representing a synthetic image of a  
handwritten digit.
```

3.6.4 encode

Encode input data

Syntax

```
Z = encode(autoenc, Xnew)
```

Description

`Z = encode(autoenc, Xnew)` returns the encoded data, `z`, for the input data `Xnew`, using the autoencoder, `autoenc`.

Example. Encode Decoded Data for New Images

Load the sample data.

X =

digitTrainCellArrayD

x is a 1-by-5003 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden size of 50 using the training data.

```
autoenc =  
trainAutoencoder(X, 5)  
Encode decoded data for new image  
data.
```

```
Xnew =  
digitTestCellArrayDa
```

```
Z =  
encode(autoenc, Xnew)
```

xnew is a 1-by-4997 cell array. z is a 50-by-4997 matrix, where each column

represents the image data of one handwritten digit in the new data x_{new} .

3.6.5 predict

Reconstruct the inputs using trained autoencoder

Syntax

- `Y = predict(autoenc, X)`

Description

`Y = predict(autoenc, X)` returns the predictions `Y` for the input data `X`, using the autoencoder `autoenc`. The result `Y` is a reconstruction of `X`.

Examples:

Predict

Continuous Measurements Using Trained Autoencoder

Load the training data.

```
X = iris_dataset;
```

The training data contains measurements on four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

Train an autoencoder on the training data using the positive saturating linear transfer function in the encoder and

linear transfer function in the decoder.

```
autoenc =  
trainAutoencoder(X, ':  
'satlin', 'DecoderTra:
```

Reconstruct the measurements using the trained network, autoenc.

```
xReconstructed =  
predict(autoenc, X);
```

Plot the predicted measurement values along with the actual values in the training dataset.

```
for i = 1:4
```

```
h(i) =
```

```
subplot(1,4,i);
```

```
plot(X(i,:),'r.');
```

hold on

plot(xReconstructed(

hold off;

end

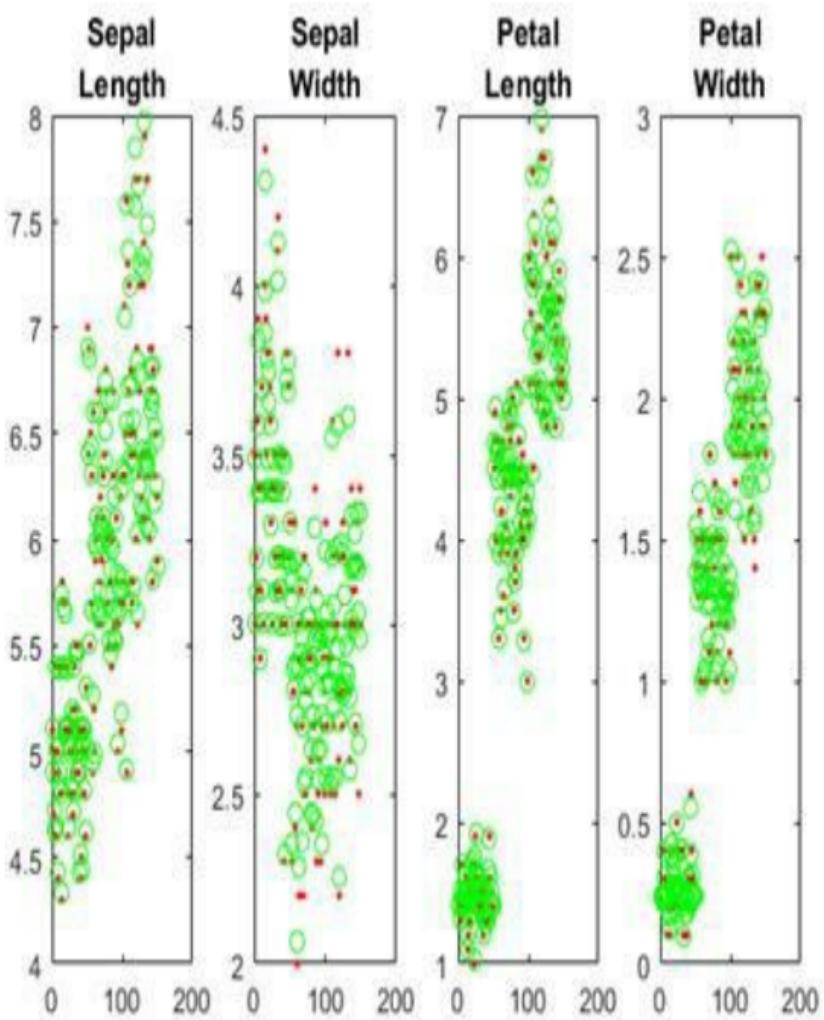
title(h(1),

{'Sepal'; 'Length'}) ;

title(h(2),

{'Sepal'; 'Width'}) ;

```
title(h(3),  
{'Petal';'Length'}) ;  
  
title(h(4),  
{'Petal';'Width'}) ;
```



The red dots represent the training data and the green circles represent the

reconstructed data.

3.6.6 stack

Stack encoders from several autoencoders together

Syntax

- ```
stackednet =
stack(autoenc1,autoenc2,...)
```
- ```
stackednet =  
stack(autoenc1,autoenc2,...,ne
```

Description

```
stackednet =  
stack(autoenc1,autoenc2,...) return
```

a network object created by stacking the encoders of the autoencoders, [autoenc1](#), autoenc2, and so on.

```
stackednet =  
stack(autoenc1, autoenc2, ..., net1)  
a network object created by stacking the  
encoders of the autoencoders and the  
network object net1.
```

The autoencoders and the network object can be stacked only if their dimensions match.

Tips

- The size of the hidden

representation of one autoencoder must match the input size of the next autoencoder or network in the stack.

The first input argument of the stacked network is the input argument of the first autoencoder. The output argument from the encoder of the first autoencoder is the input of the second autoencoder in the stacked network. The output argument from the encoder of the second autoencoder is the input argument to the third autoencoder in the stacked network, and so on.

The stacked network object `stacknet` inherits its training parameters from the final input argument [`net1`](#).

Examples. Create a Stacked Network

Load the training data.

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

Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer

to 4 and sparsity proportion to 0.05.

```
hiddenSize = 5;
```

```
autoenc =
```

```
trainAutoencoder(X,  
hiddenSize, ...)
```

```
'L2WeightRegularizat.  
0.001, ...
```

```
'SparsityRegularizat.  
4, ...  
  
'SparsityProportion'  
0.05, ...  
  
'DecoderTransferFunc'
```

Extract the features in the hidden layer.

```
features =  
encode(autoenc,X);  
Train a softmax layer for classification  
using the features.
```

```
softnet =  
trainSoftmaxLayer(fe...
```

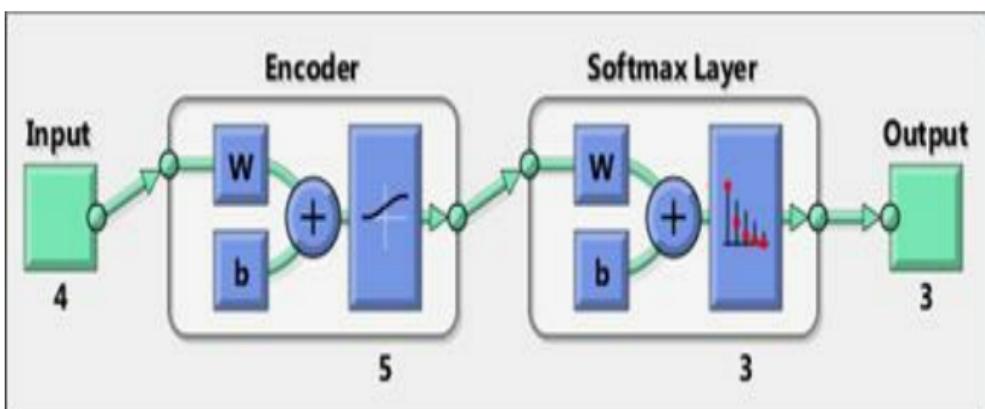
Stack the encoder and the softmax layer
to form a deep network.

```
stackednet =
```

stack (autoenc, softmax)

View the stacked network.

view (stackednet) ;



3.7 TRAIN STACKED AUTOENCODERS FOR IMAGE CLASSIFICATION. DEEP NEURAL NETWORK

This example shows how to use Neural Network Toolbox autoencoders functionality for training a deep neural network to classify images of digits.

Neural networks with multiple hidden layers can be useful for solving classification problems with complex data, such as images. Each layer can learn features at a different level of abstraction. However, training neural

networks with multiple hidden layers can be difficult in practice.

One way to effectively train a neural network with multiple layers is by training one layer at a time. You can achieve this by training a special type of network known as an autoencoder for each desired hidden layer.

This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final softmax layer, and join the layers together to form a deep network, which you train one final time in a supervised

fashion.

3.7.1 Data set

This example uses synthetic data throughout, for training and testing. The synthetic images have been generated by applying random affine transformations to digit images created using different fonts.

Each digit image is 28-by-28 pixels, and there are 5,000 training examples. You can load the training data, and view some of the images.

```
% Load the training data into memory  
[xTrainImages,tTrain] =  
digitTrainCellArrayData;
```

```
% Display some of the training images
clf
for i = 1:20
    subplot(4,5,i);
    imshow(xTrainImages{i});
end
```



The labels for the images are stored in a 10-by-5000 matrix, where in every

column a single element will be 1 to indicate the class that the digit belongs to, and all other elements in the column will be 0. It should be noted that if the tenth element is 1, then the digit image is a zero.

3.7.2 Training the first autoencoder

Begin by training a sparse autoencoder on the training data without using the labels.

An autoencoder is a neural network which attempts to replicate its input at its output. Thus, the size of its input will be the same as the size of its output. When the number of neurons in the hidden layer is less than the size of the input, the autoencoder learns a compressed representation of the input.

Neural networks have weights randomly initialized before training. Therefore the

results from training are different each time. To avoid this behavior, explicitly set the random number generator seed.

```
rng('default')
```

Set the size of the hidden layer for the autoencoder. For the autoencoder that you are going to train, it is a good idea to make this smaller than the input size.

```
hiddenSize1 = 100;
```

The type of autoencoder that you will train is a sparse autoencoder. This autoencoder uses regularizers to learn a sparse representation in the first layer. You can control the influence of these regularizers by setting various parameters:

- L2WeightRegularization controls the impact of an L2 regularizer for the weights of the network (and not the biases). This should typically be quite small.

- SparsityRegularization controls the impact of a sparsity regularizer, which attempts to enforce a constraint on the sparsity of the output from the hidden layer. Note that this is different from applying a sparsity regularizer to the weights.

- SparsityProportion is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value

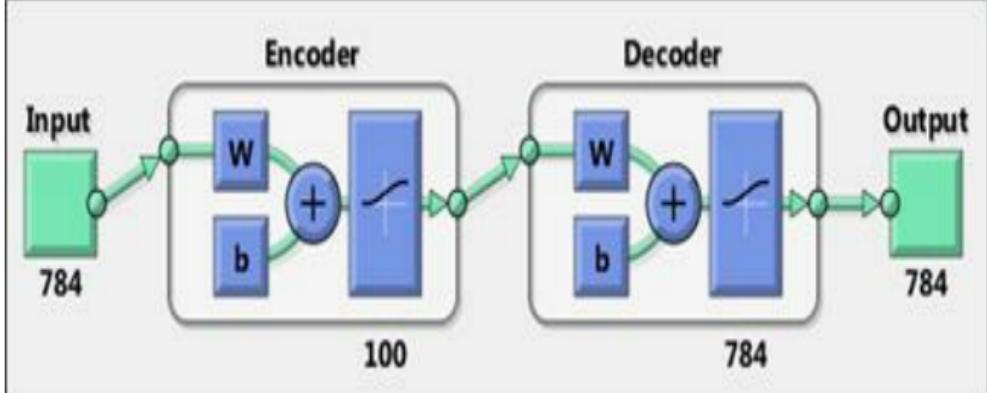
for SparsityProportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. For example, if SparsityProportion is set to 0.1, this is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples. This value must be between 0 and 1. The ideal value varies depending on the nature of the problem.

Now train the autoencoder, specifying the values for the regularizers that are described above.

```
autoenc1 =  
trainAutoencoder(xTrainImages,hiddenSi  
...  
'MaxEpochs',400, ...  
'L2WeightRegularization',0.004, ...  
'SparsityRegularization',4, ...  
'SparsityProportion',0.15, ...  
'ScaleData', false);
```

You can view a diagram of the autoencoder. The autoencoder is comprised of an encoder followed by a decoder. The encoder maps an input to a hidden representation, and the decoder attempts to reverse this mapping to reconstruct the original input.

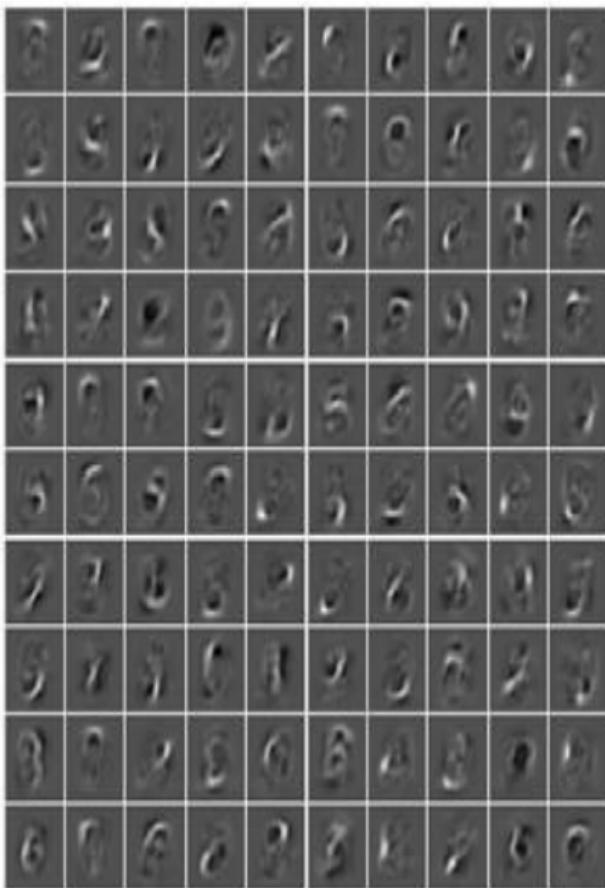
```
view(autoenc1)
```



3.7.3 Visualizing the weights of the first autoencoder

The mapping learned by the encoder part of an autoencoder can be useful for extracting features from data. Each neuron in the encoder has a vector of weights associated with it which will be tuned to respond to a particular visual feature. You can view a representation of these features.

```
figure()  
plotWeights(autoenc1);
```



You can see that the features learned by the autoencoder represent curls and

stroke patterns from the digit images.

The 100-dimensional output from the hidden layer of the autoencoder is a compressed version of the input, which summarizes its response to the features visualized above. Train the next autoencoder on a set of these vectors extracted from the training data. First, you must use the encoder from the trained autoencoder to generate the features.

```
feat1 = encode(autoenc1,xTrainImages);
```

3.7.4 Training the second autoencoder

After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that you use the features that were generated from the first autoencoder as the training data in the second autoencoder. Also, you decrease the size of the hidden representation to 50, so that the encoder in the second autoencoder learns an even smaller representation of the input data.

```
hiddenSize2 = 50;
```

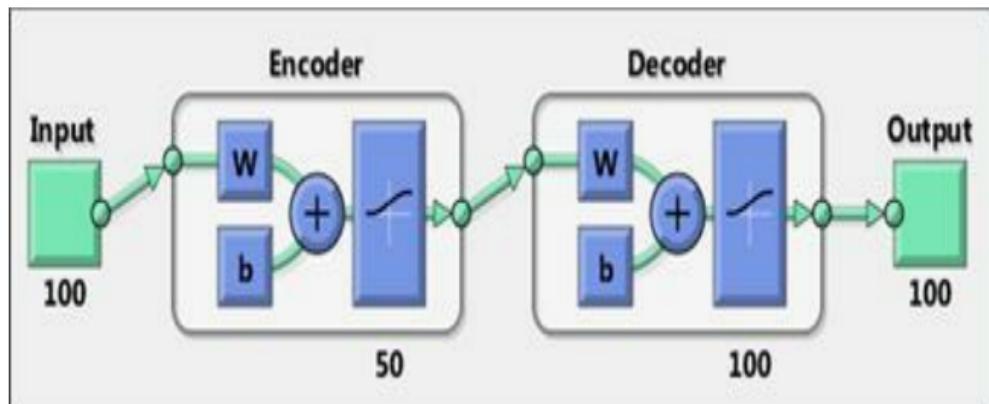
```
autoenc2 =
```

```
trainAutoencoder(feat1,hiddenSize2, ...)
```

```
'MaxEpochs',100, ...
'L2WeightRegularization',0.002, ...
'SparsityRegularization',4, ...
'SparsityProportion',0.1, ...
'ScaleData', false);
```

Once again, you can view a diagram of the autoencoder with the view function.

```
view(autoenc2)
```



You can extract a second set of features by passing the previous set through the

encoder from the second autoencoder.

```
feat2 = encode(autoenc2,feat1);
```

The original vectors in the training data had 784 dimensions. After passing them through the first encoder, this was reduced to 100 dimensions. After using the second encoder, this was reduced again to 50 dimensions. You can now train a final layer to classify these 50-dimensional vectors into different digit classes.

3.7.5 Training the final softmax layer

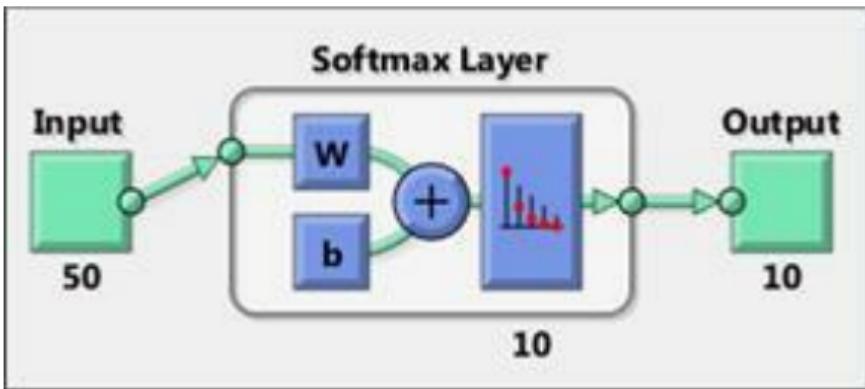
Train a softmax layer to classify the 50-dimensional feature vectors. Unlike the autoencoders, you train the softmax layer in a supervised fashion using labels for the training data.

softnet =

```
trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',10)
```

You can view a diagram of the softmax layer with the `view` function.

```
view(softnet)
```



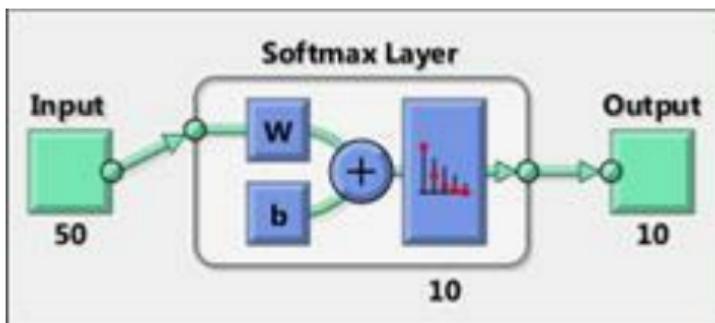
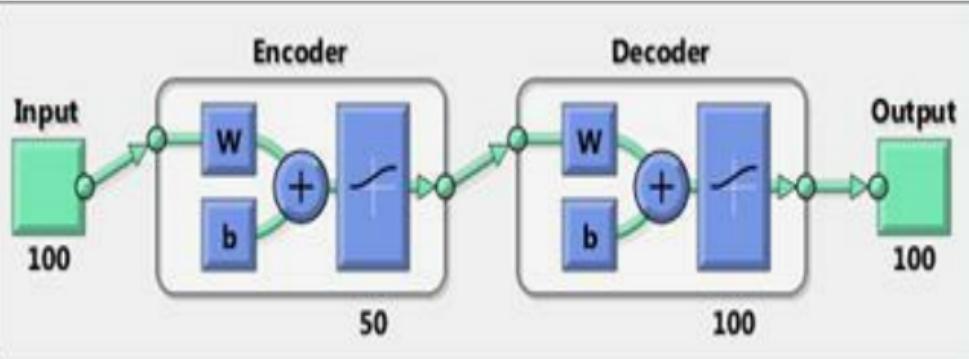
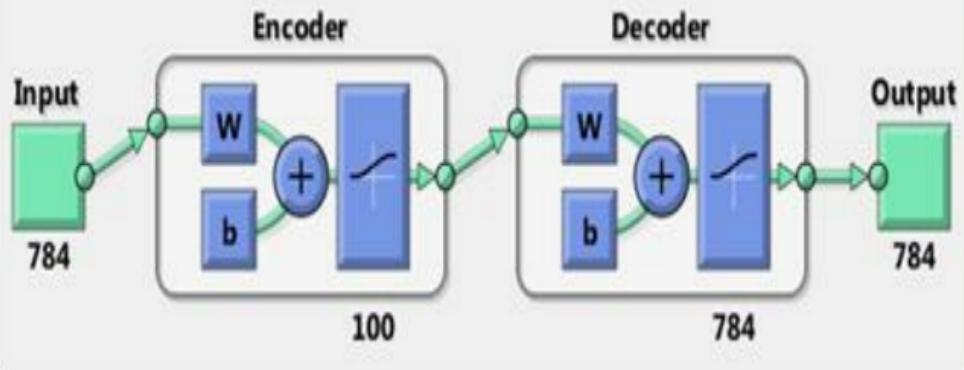
3.7.6 Forming a stacked neural network

You have trained three separate components of a deep neural network in isolation. At this point, it might be useful to view the three neural networks that you have trained. They are autoenc1, autoenc2, and softnet.

`view(autoenc1)`

`view(autoenc2)`

`view(softnet)`



As was explained, the encoders from the

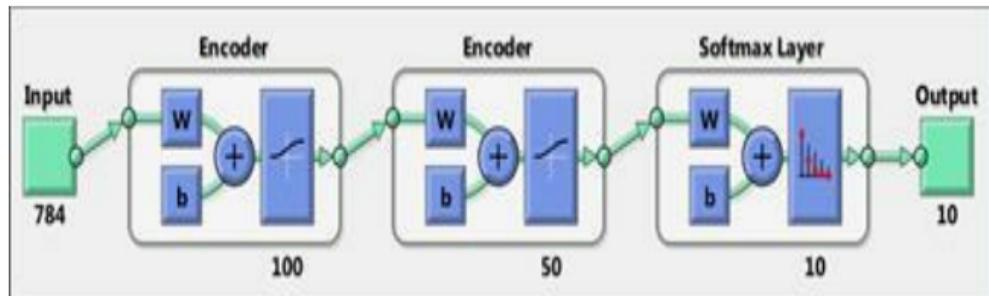
autoencoders have been used to extract features. You can stack the encoders from the autoencoders together with the softmax layer to form a deep network.

deepnet =

```
stack(autoenc1,autoenc2,softnet);
```

You can view a diagram of the stacked network with the view function. The network is formed by the encoders from the autoencoders and the softmax layer.

view(deepnet)



With the full deep network formed, you can compute the results on the test set. To use images with the stacked network, you have to reshape the test images into a matrix. You can do this by stacking the columns of an image to form a vector, and then forming a matrix from these vectors.

```
% Get the number of pixels in each  
image  
imageWidth = 28;  
imageHeight = 28;  
inputSize = imageWidth*imageHeight;  
  
% Load the test images  
[xTestImages,tTest] =  
digitTestCellArrayData;
```

```
% Turn the test images into vectors and  
put them in a matrix  
xTest =  
zeros(inputSize,numel(xTestImages));  
for i = 1:numel(xTestImages)  
    xTest(:,i) = xTestImages{i}(:);  
end
```

You can visualize the results with a confusion matrix. The numbers in the bottom right-hand square of the matrix give the overall accuracy.

```
y = deepnet(xTest);  
plotconfusion(tTest,y);
```

Confusion Matrix

Output Class	Target Class										Overall Accuracy (%)
	1	2	3	4	5	6	7	8	9	10	
1	448 9.0%	9 0.2%	0 0.0%	1 0.0%	3 0.1%	8 0.2%	14 0.3%	11 0.2%	0 0.0%	3 0.1%	90.1% 9.9%
2	3 0.1%	447 8.9%	14 0.3%	4 0.1%	0 0.0%	0 0.0%	14 0.3%	20 0.4%	6 0.1%	17 0.3%	85.1% 14.9%
3	6 0.1%	21 0.4%	338 6.8%	1 0.0%	49 1.0%	2 0.0%	7 0.1%	41 0.8%	3 0.1%	4 0.1%	71.6% 28.4%
4	7 0.1%	1 0.0%	5 0.1%	472 9.4%	1 0.0%	8 0.2%	0 0.0%	1 0.0%	5 0.1%	1 0.0%	94.2% 5.8%
5	0 0.0%	2 0.0%	61 1.2%	2 0.0%	411 8.2%	26 0.5%	0 0.0%	60 1.2%	1 0.0%	2 0.0%	72.7% 27.3%
6	19 0.4%	0 0.0%	5 0.1%	5 0.1%	6 0.1%	409 8.2%	3 0.1%	34 0.7%	9 0.2%	16 0.3%	80.8% 19.2%
7	30 0.6%	10 0.2%	6 0.1%	2 0.0%	0 0.0%	0 0.0%	450 9.0%	5 0.1%	1 0.0%	2 0.0%	88.9% 11.1%
8	0 0.0%	0 0.0%	42 0.8%	2 0.0%	20 0.4%	19 0.4%	7 0.1%	303 6.1%	5 0.1%	27 0.5%	71.3% 28.7%
9	0 0.0%	1 0.0%	16 0.3%	6 0.1%	6 0.1%	9 0.2%	2 0.0%	16 0.3%	461 9.2%	7 0.1%	88.0% 12.0%
10	0 0.0%	10 0.2%	9 0.2%	0 0.0%	12 0.2%	15 0.3%	2 0.0%	10 0.2%	3 0.1%	415 8.3%	87.2% 12.8%
	87.3% 12.7%	89.2% 10.8%	88.1% 31.9%	85.4% 4.6%	80.9% 19.1%	82.5% 17.5%	80.2% 9.8%	80.5% 39.5%	83.3% 6.7%	84.0% 16.0%	83.1% 16.9%

3.7.7 Fine tuning the deep neural network

The results for the deep neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

```
% Turn the training images into vectors  
and put them in a matrix
```

```
xTrain =  
zeros(inputSize,numel(xTrainImages));  
for i = 1:numel(xTrainImages)  
    xTrain(:,i) = xTrainImages{i}(:);  
end
```

```
% Perform fine tuning  
deepnet = train(deepnet,xTrain,tTrain);
```

You then view the results again using a confusion matrix.

```
y = deepnet(xTest);  
plotconfusion(tTest,y);
```

Confusion Matrix

Output Class	Target Class										Accuracy (%)
	1	2	3	4	5	6	7	8	9	10	
1	511 10.2%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	1 0.0%	0 0.0%	0 0.0%	99.6% 0.4%
2	0 0.0%	501 10.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	99.6% 0.4%
3	0 0.0%	0 0.0%	496 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	494 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
5	0 0.0%	0 0.0%	0 0.0%	508 10.2%	1 0.0%	0 0.0%	3 0.1%	0 0.0%	0 0.0%	0 0.0%	99.2% 0.8%
6	0 0.0%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
7	2 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	498 10.0%	0 0.0%	0 0.0%	0 0.0%	99.4% 0.6%
8	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	0 0.0%	496 9.9%	0 0.0%	0 0.0%	99.6% 0.4%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	494 9.9%	0 0.0%	100% 0.0%
10	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	100% 0.0%
	99.6% 0.4%	100% 0.0%	100% 0.0%	99.8% 0.2%	100% 0.0%	99.4% 0.6%	99.8% 0.2%	99.0% 1.0%	100% 0.0%	99.8% 0.2%	99.7% 0.3%

3.7.8 Summary

This example showed how to train a deep neural network to classify digits in images using Neural Network Toolbox™. The steps that have been outlined can be applied to other similar problems, such as classifying images of letters, or even small images of objects of a specific category.

3.8 PERFORM CLASSIFICATION, FEATURE EXTRACTION, AND TRANSFER LEARNING USING

CONVOLUTIONAL NEURAL NETWORKS (CNNs, CONVNets)

Convolution neural networks (CNNs or ConvNets) are essential tools for deep learning, and are especially suited for image recognition. You can construct a CNN architecture, train a network, and use the trained network to predict class labels. You can also extract features from a pre-trained network, and use these features to train a linear classifier. Neural Network Toolbox also enables you to perform transfer learning; that is, retrain the last fully connected layer of an existing CNN on new data.

MATLAB has the following functions:

<u>imageInputLayer</u>	Image input layer
<u>convolution2dLayer</u>	Convolutional layer
<u>reluLayer</u>	Rectified Linear Unit (R
<u>crossChannelNormalizationLayer</u>	Channel-wise local resp
<u>averagePooling2dLayer</u>	Average pooling layer of
<u>maxPooling2dLayer</u>	Max pooling layer
<u>fullyConnectedLayer</u>	Fully connected layer
<u>dropOutLayer</u>	Dropout layer
<u>softmaxLayer</u>	Softmax layer
<u>classificationLayer</u>	Create a classification or

3.9 TRANSFER LEARNING USING CONVOLUTIONAL NEURAL NETWORKS

Fine-tune a convolutional neural network pretrained on digit images to learn the features of letter images. Transfer learning is considered as the transfer of knowledge from one learned task to a new task in machine learning [1]. In the context of neural networks, it is transferring learned features of a pretrained network to a new problem. Training a convolutional neural network from the beginning in each case usually

is not effective when there is not sufficient amount of training data. The common practice in deep learning for such cases is to use a network that is trained on a large data set for a new problem. While the initial layers of the pretrained network can be fixed, the last few layers must be fine-tuned to learn the specific features of the new data set. Transfer learning usually results in faster training times than training a new convolutional neural network because you do not need to estimate all the parameters in the new network.

NOTE: Training a convolutional neural network requires Parallel Computing Toolbox™ and a CUDA®-enabled

NVIDIA® GPU with compute capability 3.0 or higher.

Load the sample data as an ImageDatastore.

```
digitDatasetPath =  
fullfile(matlabroot,'toolbox','nnet','nnDEM  
'nnDatasets','DigitDataset');  
digitData =  
 imageDatastore(digitDatasetPath,...
```

```
'IncludeSubfolders',true,'LabelSource','fo
```

The data store contains 10000 synthetic images of digits 0–9. The images are generated by applying random transformations to digit images created using different fonts. Each digit image is

28-by-28 pixels.

Display some of the images in the datastore.

```
for i = 1:20
```

```
    subplot(4,5,i);  
    imshow(digitData.Files{i});
```

```
end
```



Check the number of images in each digit category.

`digitData.countEachLabel`

`ans =`

Label	Count
-------	-------

0	988
1	1026
2	1003
3	993
4	991
5	1017
6	992
7	999
8	1003
9	988

The data contains an unequal number of images per category.

To balance the number of images for each digit in the training set, first find the minimum number of images in a category.

`minSetCount =`

`min(digitData.countEachLabel {:,2})`

`minSetCount =`

988

Divide the dataset so that each category in the training set has 494 images and the testing set has the remaining images from each label.

`trainingNumFiles =`

`round(minSetCount/2);`

```
rng(1) % For reproducibility  
[trainDigitData,testDigitData] =  
splitEachLabel(digitData,...  
  
trainingNumFiles,'randomize');  
  
splitEachLabel splits the image files  
in digitData into two new  
datastores, trainDigitData and testDigitL  
Create the layers for the convolutional  
neural network.  
  
layers = [imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    reluLayer()  
    maxPooling2dLayer(2,'Stride',2)  
    fullyConnectedLayer(10)]
```

```
softmaxLayer()  
classificationLayer());
```

Create the training options. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.001.

```
options =  
trainingOptions('sgdm','MaxEpochs',20,..  
'InitialLearnRate',0.001);
```

Train the network using the training set and the options you defined in the previous step.

```
convnet =  
trainNetwork(trainDigitData,layers,options)
```

Epoch	Iteration	Time Elapsed	Mini-batch	Mini-
-------	-----------	--------------	------------	-------

batch	Base Learning				
		(seconds)	Loss	Accuracy	
Rate					
0.001000	2	50	0.71	0.2233	92.97%
0.001000	3	100	1.37	0.0182	99.22%
0.001000	4	150	2.02	0.0395	99.22%
0.001000	6	200	2.70	0.0105	99.22%
0.001000	7	250	3.35	0.0026	100.00%
0.001000	8	300	4.00	0.0004	100.00%
0.001000	10	350	4.67	0.0002	100.00%
0.001000	11	400	5.32	0.0001	100.00%
0.001000	12	450	5.95	0.0001	100.00%
0.001000	14	500	6.60	0.0002	100.00%
0.001000	15	550	7.23	0.0001	100.00%
0.001000	16	600	7.87	0.0001	100.00%

	18	650	8.52	0.0001	100.00%
0.001000					
	19	700	9.15	0.0001	100.00%
0.001000					
	20	750	9.79	0.0000	100.00%
0.001000					

Test the network using the testing set and compute the accuracy.

```
YTest = classify(convnet,testDigitData);  
TTest = testDigitData.Labels;  
accuracy = sum(YTest ==  
TTest)/numel(YTest)  
  
accuracy =
```

0.9976

Accuracy is the ratio of the number of true labels in the test data matching the

classifications from classify, to the number of images in the test data. In this case 99.78% of the digit estimations match the true digit values in the test set.

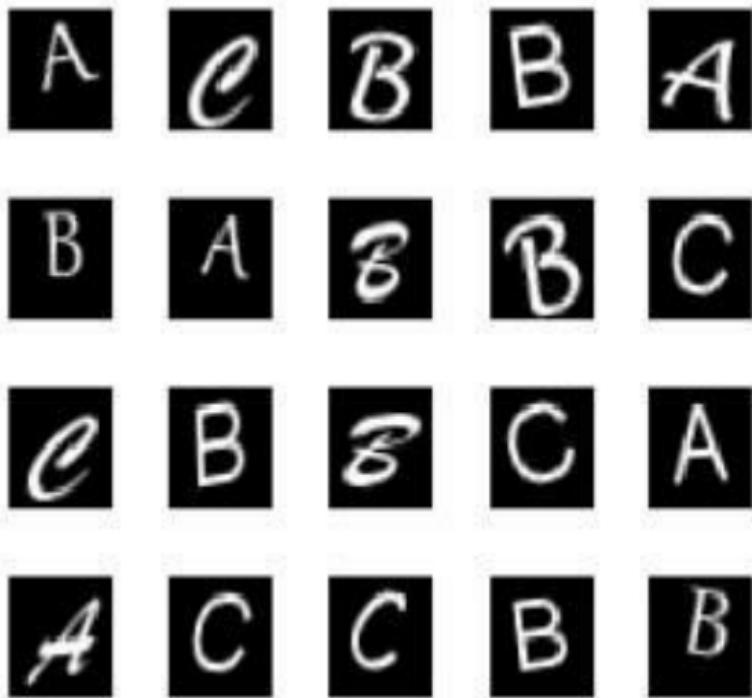
Now, suppose you would like to use the trained network net to predict classes on a new set of data. Load the letters training data.

```
load lettersTrainSet.mat
```

XTrain contains 1500 28-by-28 grayscale images of the letters A, B, and C in a 4-D array. TTrain contains the categorical array of the letter labels.

Display some of the letter images.

```
figure;
for j = 1:20
    subplot(4,5,j);
    selectImage =
datasample(XTrain,1,4);
    imshow(selectImage,[ ]);
end
```



The pixel values in XTrain are in the range [0 1]. The digit data used in training the network net were in [0 255]; scale the letters data between [0 255].

```
XTrain = XTrain*255;
```

The last three layers of the trained network net are tuned for the digit dataset, which has 10 classes. The properties of these layers depend on the classification task. Display the fully connected layer (fullyConnectedLayer).

```
convnet.Layers(end-2)
```

```
ans =
```

FullyConnectedLayer with properties:

Name: 'fc'

Hyperparameters

InputSize: 2880

OutputSize: 10

Learnable Parameters

Weights: [10×2880 single]

Bias: [10×1 single]

Use properties method to see a list of all properties.

Display the last layer
(classificationLayer).

convnet.Layers(end)

ans =

ClassificationOutputLayer with
properties:

Name: 'classoutput'
classNames: {10×1 cell}
OutputSize: 10

Hyperparameters

LossFunction: 'crossentropyex'

These three layers must be fine-tuned for the new classification problem. Extract all the layers but the last three from the trained network, net.

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

The letters data set has three classes. Add a new fully connected layer for three classes, and increase the learning

rate for this layer.

```
layersTransfer(end+1) =  
fullyConnectedLayer(3,...  
    'WeightLearnRateFactor',10,...  
    'BiasLearnRateFactor',20);
```

WeightLearnRateFactor and BiasLearnR
multipliers of the global learning rate for
the fully connected layer.

Add a softmax layer and a classification
output layer.

```
layersTransfer(end+1) = softmaxLayer();  
layersTransfer(end+1) =  
classificationLayer();
```

Create the options for transfer learning.
You do not have to train for many epochs

(MaxEpochs can be lower than before).

Set the InitialLearnRate at a lower rate than used for training net to improve convergence by taking smaller steps.

optionsTransfer =

trainingOptions('sgdm',...

'MaxEpochs',5,...

'InitialLearnRate',0.000005,...

'Verbose',true);

Perform transfer learning.

convnetTransfer =

trainNetwork(XTrain,TTrain,...

layersTransfer,optionsTransfer);

Epoch	Iteration	Time Elapsed	Mini-batch	Mini-batch	Base Learning
		(seconds)		Loss	Accuracy
Rate					
5	50	0.43	0.0011	100.00%	
0.000005					

Load the letters test data. Similar to the letters training data, scale the testing data between [0 255], because the training data were between that range.

```
load lettersTestSet.mat
```

```
XTest = XTest*255;
```

Test the accuracy.

```
YTest =
```

```
classify(convnetTransfer,XTest);
```

```
accuracy = sum(YTest ==
```

TTest)/numel(TTest)

accuracy =

0.9587

3.10 CRAB CLASSIFICATION

This example illustrates using a neural network as a classifier to identify the sex of crabs from physical dimensions of the crab. In this example we attempt to build a classifier that can identify the sex of a crab from its physical measurements. Six physical characteristics of a crab are considered: species, frontallip, rearwidth, length, width and depth. The problem on hand is to identify the sex of a crab given the observed values for each of these 6 physical characteristics.

3.10.1 Why Neural Networks?

Neural networks have proven themselves as proficient classifiers and are particularly well suited for addressing non-linear problems. Given the non-linear nature of real world phenomena, like crab classification, neural networks is certainly a good candidate for solving the problem.

The six physical characterstics will act as inputs to a neural network and the sex of the crab will be target. Given an input, which constitutes the six observed values for the physical characterstics of

a crab, the neural network is expected to identify if the crab is male or female.

This is achieved by presenting previously recorded inputs to a neural network and then tuning it to produce the desired target outputs. This process is called neural network training.

3.10.2 Preparing the Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have six elements representing a crabs species, fontallip, rearwidth, length, width and depth.

Each corresponding column of the target matrix will have two elements. Female crabs are reprented with a one in the first element, male crabs with a one in the second element. (All other elements are zero).

Here such the dataset is loaded.

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

```
size(x)
```

```
size(t)
```

```
ans =
```

6 200

ans =

2 200

3.10.3 Building the Neural Network Classifier

The next step is to create a neural network that will learn to identify the sex of the crabs.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

setdemorandstream(49)

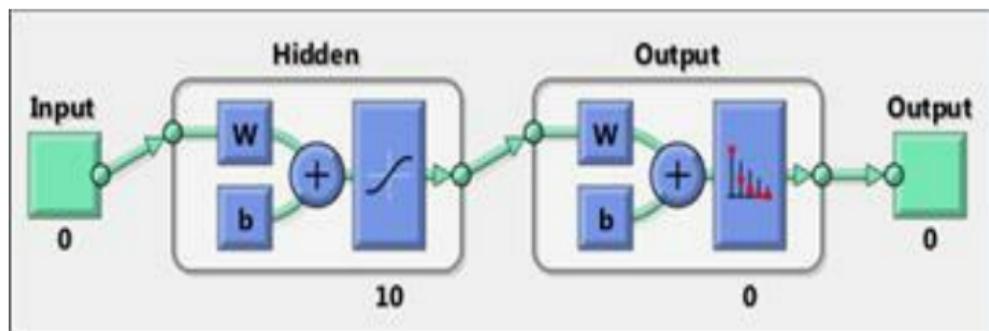
Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target

data. This will happen when the network is trained.

```
net =  
patternnet(10);  
  
view(net)
```



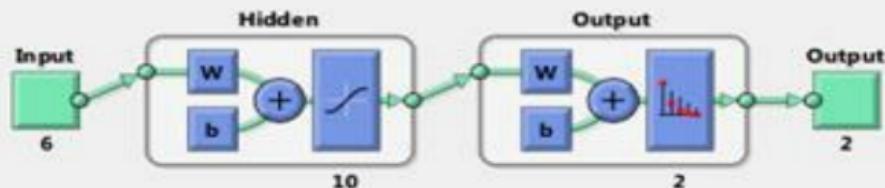
Now the network is ready to be trained.

The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

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

Nntraintool

Neural Network



Algorithms

Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Calculations: MEX

Progress

Epoch:	0	45 iterations	1000
Time:		0:00:00	
Performance:	0.537	0.0170	0.00
Gradient:	0.440	0.0110	1.00e-06
Validation Checks:	0	6	6

Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval:



Validation stop.

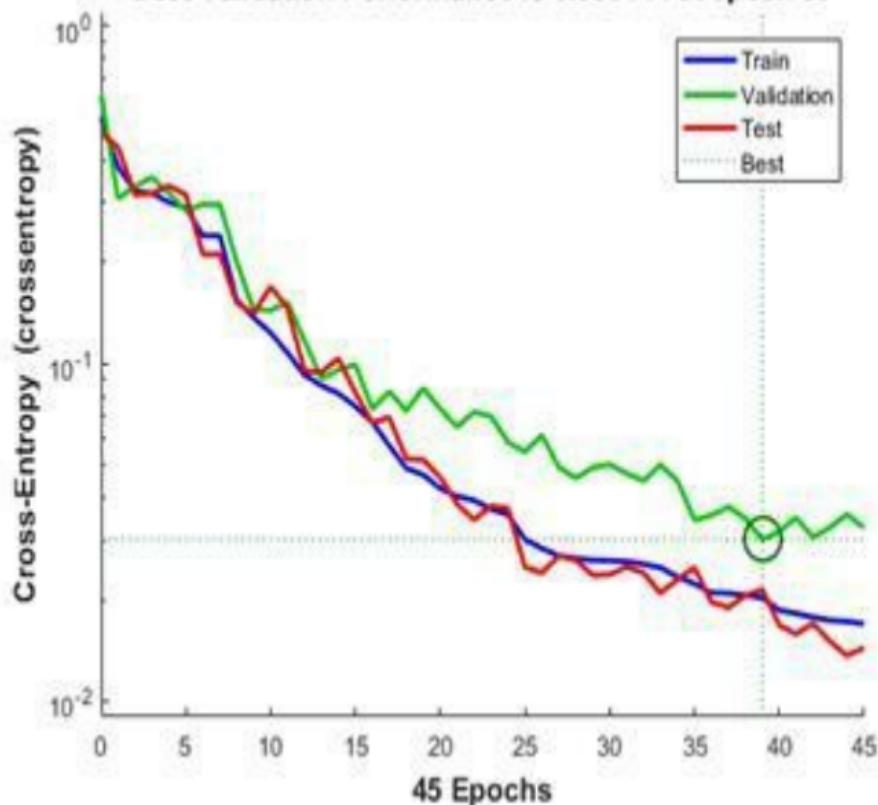
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

Best Validation Performance is 0.030144 at epoch 39



3.10.4 Testing the Classifier

The trained neural network can now be tested with the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX =  
x(:, tr.testInd);
```

```
testT =  
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
testIndices =  
vec2ind(testY)
```

```
testIndices =
```

Columns 1 through 13

2	2	1	2	2	2	2	1	2	2	2	1
---	---	---	---	---	---	---	---	---	---	---	---

Columns 14 through 26

2	1	1	2	2	2	1	2	1	1	1	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---

Columns 27 through 30

1 2 2 1

One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices

diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

plotconfusion(testT, · · ·)



Here are the overall percentages of

correct and incorrect classification.

```
[c, cm] =  
confusion(testT, testC)
```

```
fprintf('Percentage  
Correct  
Classification :  
%f%%\n', 100*(1-c));
```

```
fprintf('Percentage
```

Incorrect
Classification :
%f%%\n', 100*c);

c =

0.0333

cm =

12 1

0 17

Percentage Correct
Classification :

96.66667%

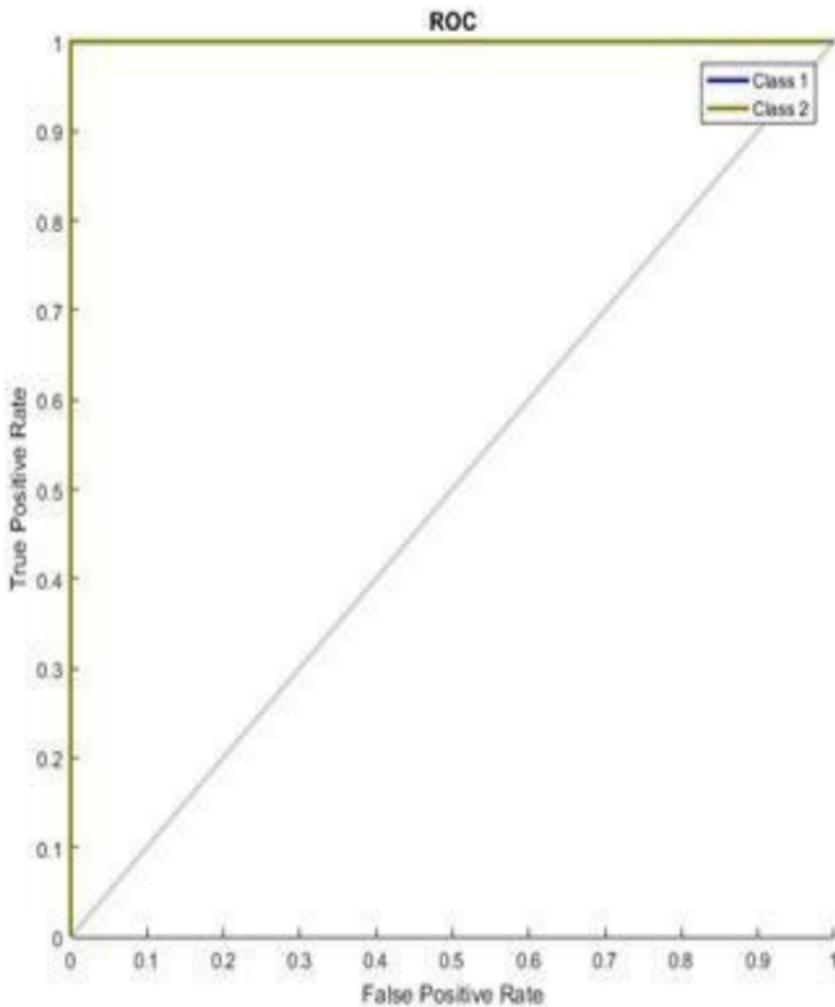
Percentage Incorrect Classification :
3.333333%

Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true

positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

```
plotroc(testT, testY)
```



This example illustrated using a neural

network to classify crabs.

3.11 WINE CLASSIFICATION. PATTERN RECOGNITION

This example illustrates how a pattern recognition neural network can classify wines by winery based on its chemical characteristics.

3.11.1 The Problem: Classify Wines

In this example we attempt to build a neural network that can classify wines from three wineries by thirteen attributes:

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols

- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

This is an example of a pattern recognition problem, where inputs are associated with different classes, and we would like to create a neural network that not only classifies the known wines properly, but can generalize to accurately classify wines that were not used to design the solution.

3.11.2 Why Neural Networks?

Neural networks are very good at pattern recognition problems. A neural network with enough elements (called neurons) can classify any data with arbitrary accuracy. They are particularly well suited for complex decision boundary problems over many variables. Therefore neural networks are a good candidate for solving the wine classification problem.

The thirteen neighborhood attributes will act as inputs to a neural network, and the respective target for each will be

a 3-element class vector with a 1 in the position of the associated winery, #1, #2 or #3.

The network will be designed by using the attributes of neighborhoods to train the network to produce the correct target classes.

3.11.3 Preparing the Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have thirteen elements representing a wine whose winery is already known.

Each corresponding column of the target matrix will have three elements, consisting of two zeros and a 1 in the location of the associated winery.

Here such a dataset is loaded.

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

We can view the sizes of inputs X and targets T.

Note that both X and T have 178 columns. These represent 178 wine sample attributes (inputs) and associated winery class vectors (targets).

Input matrix X has thirteen rows, for the thirteen attributes. Target matrix T has three rows, as for each example we have three possible wineries.

`size(x)`

`size(t)`

`ans =`

13 178

ans =

3 178

3.11.4 Pattern Recognition with a Neural Network

The next step is to create a neural network that will learn to classify the wines.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

```
setdemorandstream(391418381)
```

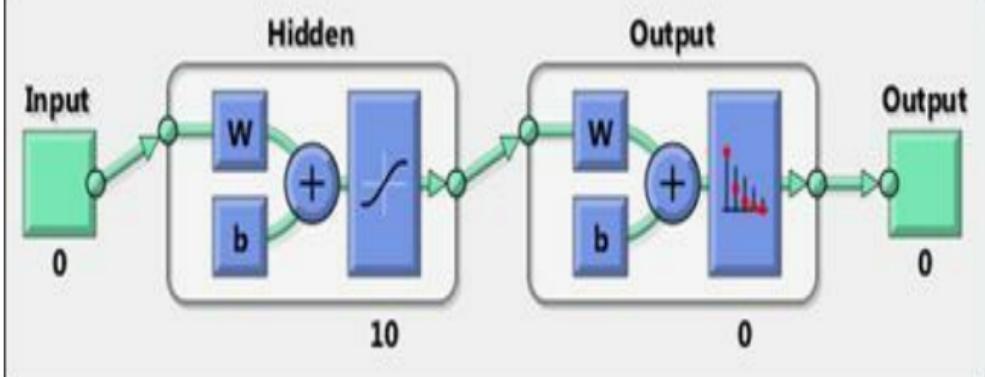
Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough

neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

```
net = patternnet(10);  
view(net)
```



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

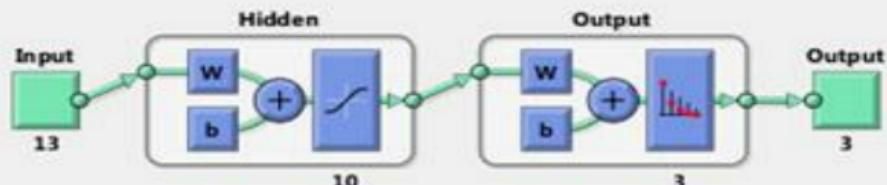
The NN Training Tool shows the network being trained and the algorithms

used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

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

Neural Network



Algorithms

Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Calculations: MEX

Progress

Epoch:	0	37 iterations	1000
Time:		0:00:00	
Performance:	0.635	2.71e-07	0.00
Gradient:	0.383	5.84e-07	1.00e-06
Validation Checks:	0	1	6

Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval: 1 epochs



Minimum gradient reached.



Stop Training



Cancel

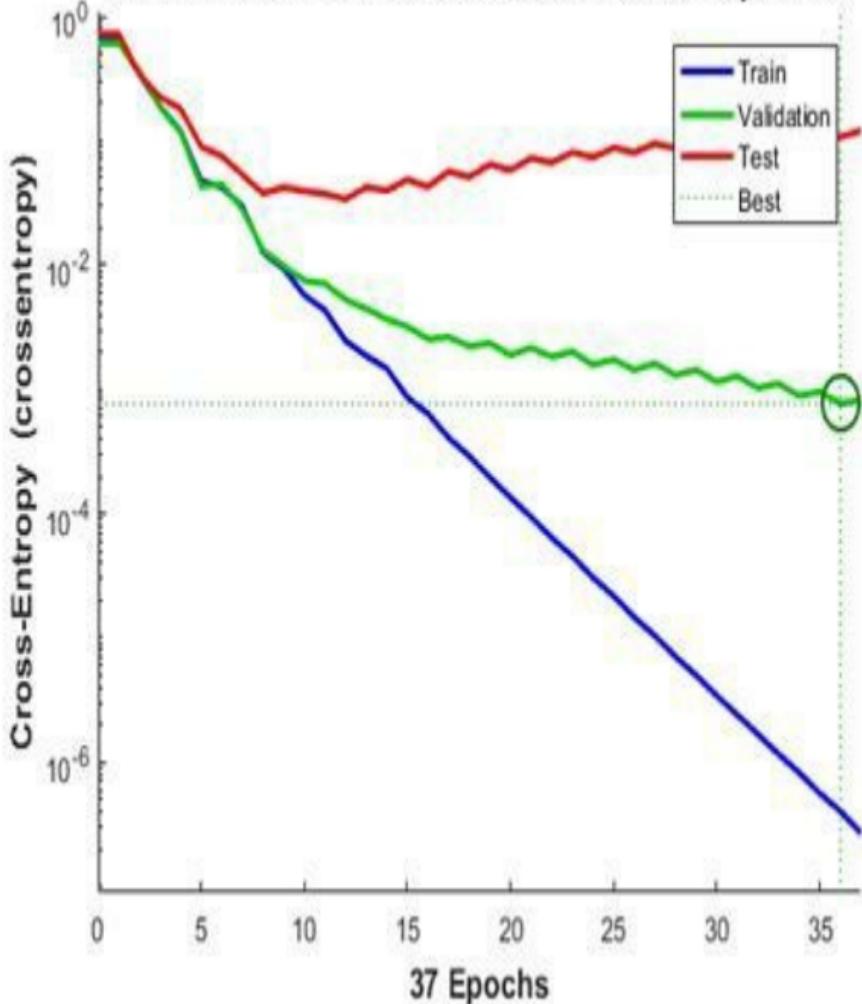
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

Best Validation Performance is 0.00076278 at epoch 36



3.11.5 Testing the Neural Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX = x(:,tr.testInd);
```

```
testT = t(:,tr.testInd);
```

```
testY = net(testX);  
testIndices = vec2ind(testY)  
  
testIndices =
```

Columns 1 through 13

	1	1	1	2	1	1	1	1	1	1	1
1	2	2									

Columns 14 through 26

	2	2	2	2	2	2	3	2	3	3
3	3	3								

Column 27

3

Another measure of how well the neural

network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
plotconfusion(testT,testY)
```

Confusion Matrix

		Target Class			
		1	2	3	4
Output Class	1	10 37.0%	0 0.0%	0 0.0%	100% 0.0%
	2	1 3.7%	8 29.6%	1 3.7%	80.0% 20.0%
3	0 0.0%	0 0.0%	7 25.9%	100% 0.0%	
4	90.9% 9.1%	100% 0.0%	87.5% 12.5%	92.6% 7.4%	

Here are the overall percentages of correct and incorrect classification.

```
[c,cm] = confusion(testT,testY)
```

```
fprintf('Percentage Correct  
Classification : %f%%\n', 100*(1-c));  
fprintf('Percentage Incorrect  
Classification : %f%%\n', 100*c);
```

c =

0.0741

cm =

10 1 0

0	8	0
0	1	7

Percentage Correct Classification :
92.592593%

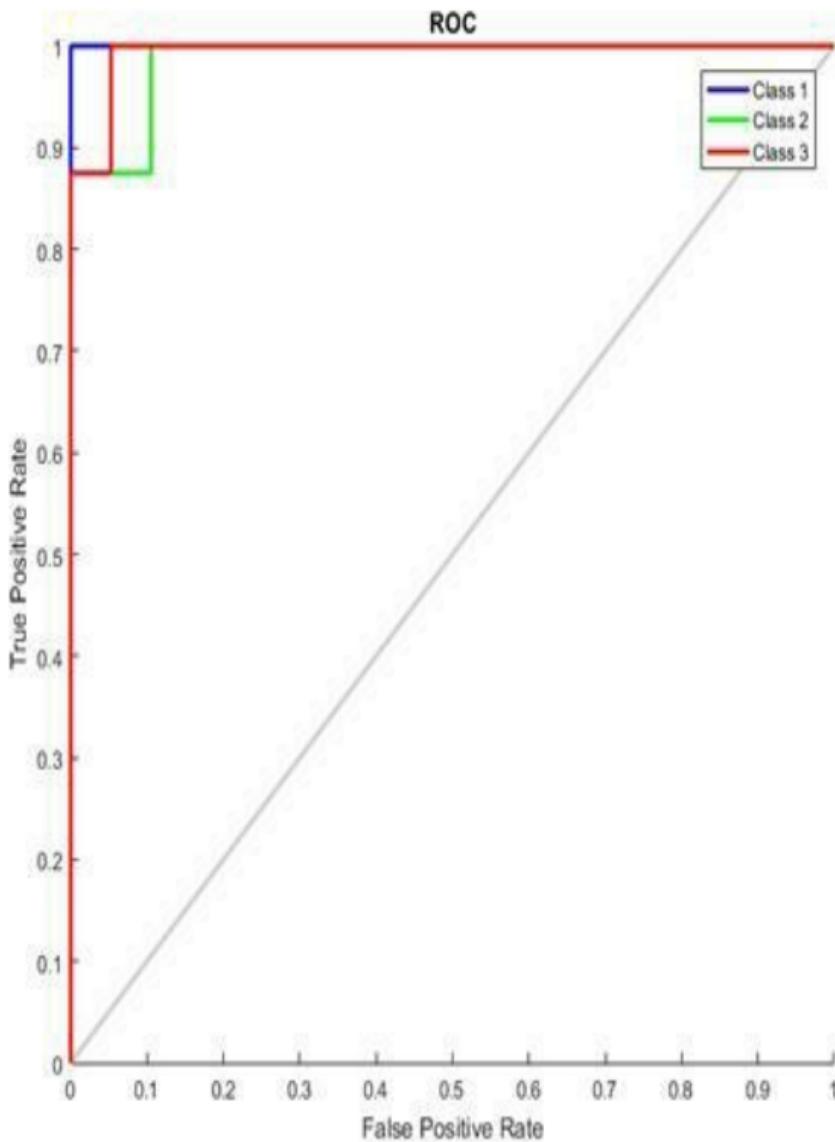
Percentage Incorrect Classification :
7.407407%

A third measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers will

have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

plotroc(testT,testY)



3.12 CANCER DETECTION

This example demonstrates using a neural network to detect cancer from mass spectrometry data on protein profiles.

Serum proteomic pattern diagnostics can be used to differentiate samples from patients with and without disease. Profile patterns are generated using surface-enhanced laser desorption and ionization (SELDI) protein mass spectrometry. This technology has the potential to improve clinical diagnostics tests for cancer pathologies.

The goal is to build a classifier that can

distinguish between cancer and control patients from the mass spectrometry data.

The methodology followed in this example is to select a reduced set of measurements or "features" that can be used to distinguish between cancer and control patients using a classifier.

These features will be ion intensity levels at specific mass/charge values.

3.12.1 Formatting the Data

The data in this example is from the FDA-NCI Clinical Proteomics Program Databank: <http://home.ccr.cancer.gov/nci>

To recreate the data in **ovarian_dataset.mat** used in this example, download and uncompress the raw mass-spectrometry data from the FDA-NCI web site. Create the data file **OvarianCancerQAQCdataset.mat** either running script **msseqprocessing** in Bioinformatics Toolbox (TM) or by following the steps in the example **biodistcompdemo** (Batch processing with parallel computing). The new file contains

variables `Y`, `MZ` and `grp`.

Each column in `Y` represents measurements taken from a patient. There are 216 columns in `Y` representing 216 patients, out of which 121 are ovarian cancer patients and 95 are normal patients.

Each row in `Y` represents the ion intensity level at a specific mass-charge value indicated in `MZ`. There are 15000 mass-charge values in `MZ` and each row in `Y` represents the ion-intesity levels of the patients at that particular mass-charge value.

The variable `grp` holds the index information as to which of these samples

represent cancer patients and which ones represent normal patients.

An extensive description of this data set and excellent introduction to this promising technology can be found in [1] and [2].

3.12.2 Ranking Key Features

This is a typical classification problem in which the number of features is much larger than the number of observations, but in which no single feature achieves a correct classification, therefore we need to find a classifier which appropriately learns how to weight multiple features and at the same time produce a generalized mapping which is not over-fitted.

A simple approach for finding significant features is to assume that each M/Z value is independent and

compute a two-way t-test. **rankfeatures** returns an index to the most significant M/Z values, for instance 100 indices ranked by the absolute value of the test statistic.

To finish recreating the data from **ovarian_dataset.mat**, load the **OvarianCancerQAQCdataset.mat** and Bioinformatics Toolbox to choose 100 highest ranked measurements as inputs **x**.

```
ind =  
rankfeatures(Y,grp,''  
  
x = Y(ind,:);
```

Define the targets t for the two classes as follows:

```
t =  
double(strcmp('Cancel',
```

```
t = [t; 1-t];
```

The preprocessing steps from the script and example listed above are intended to demonstrate a representative set of possible pre-processing and feature selection procedures. Using different steps or parameters may lead to different and possibly improved results of this example.

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

```
whos
```

Name

Size

Bytes Class

Attributes

t

2x216

3456

double

x

100x216

172800

double

Each column in x represents one of 216 different patients.

Each row in \mathbf{x} represents the ion intensity level at one of the 100 specific mass-charge values for each patient.

The variable \mathbf{t} has 2 rows of 216 values each of which are either [1;0], indicating a cancer patient, or [0;1] for a normal patient.

3.12.3 Classification Using a Feed Forward Neural Network

Now that you have identified some significant features, you can use this information to classify the cancer and normal samples.

Since the neural network is initialized with random initial weights, the results after training the network vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. However this is not necessary for your own applications.

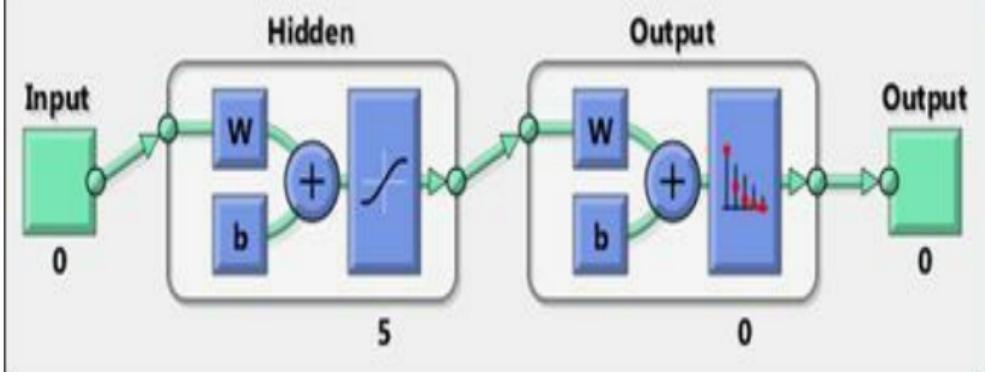
setdemorandstream(67);

A 1-hidden layer feed forward neural network with 5 hidden layer neurons is created and trained. The input and target samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The input and output have sizes of 0

because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

```
net = patternnet(5);  
view(net)
```



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The NN Training Tool shows the network being trained and the algorithms

used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

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

To see how the network's performance

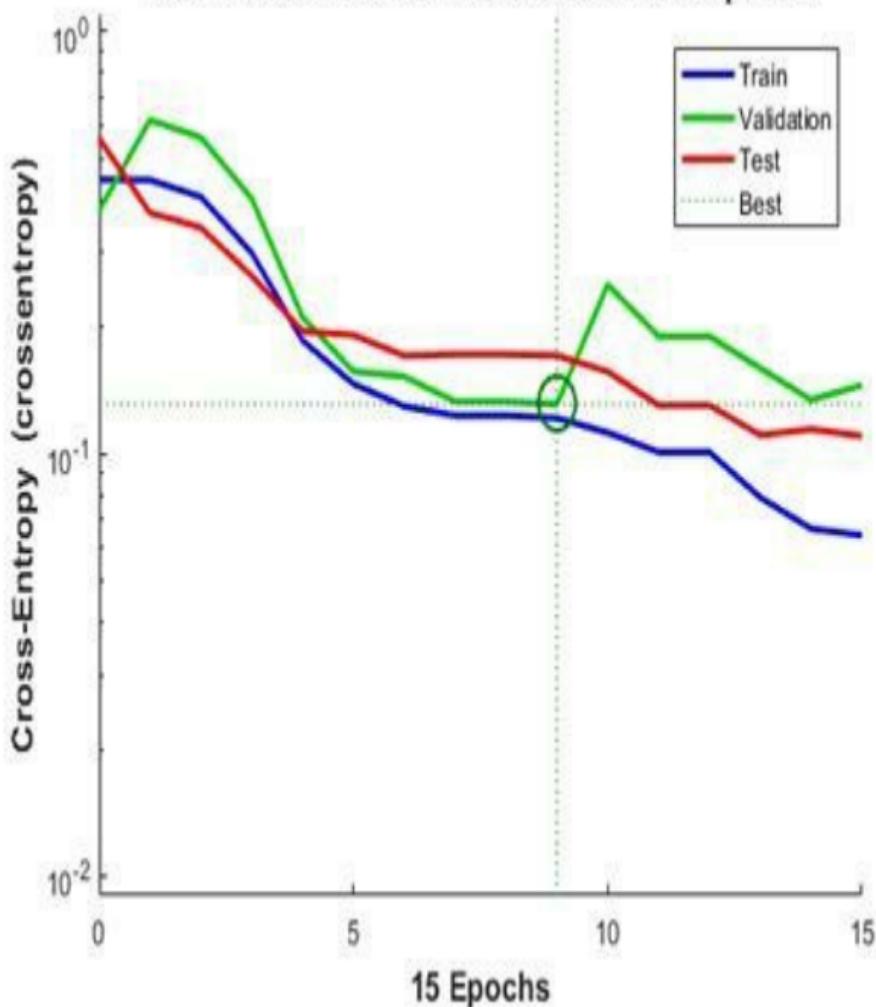
improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is was after training.

```
plotperform(tr)
```

Best Validation Performance is 0.13105 at epoch 9



The trained neural network can now be tested with the testing samples we

partitioned from the main dataset. The testing data was not used in training in any way and hence provides an "out-of-sample" dataset to test the network on. This will give us a sense of how well the network will do when tested with data from the real world.

The network outputs will be in the range 0 to 1, so we threshold them to get 1's and 0's indicating cancer or normal patients respectively.

```
testX =  
x(:, tr.testInd);
```

```
testT =  
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
testClasses = testY  
> 0.5
```

```
testClasses =
```

2×32 logical array

Columns 1 through 19

Columns 20 through 32

0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0								
1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	1								

One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications

are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

plotconfusion(testT, · · ·)

Confusion Matrix		
Output Class	Target Class	
	1	2
1	13 40.6%	5 15.6%
2	0 0.0%	14 43.8%
	100% 0.0%	73.7% 26.3%
		84.4% 15.6%

Here are the overall percentages of correct and incorrect classification.

```
[c, cm] =  
confusion(testT, test)
```

```
fprintf('Percentage  
Correct  
Classification :  
%f%%\n', 100*(1-c));
```

```
fprintf('Percentage  
Incorrect  
Classification :
```

%f%%\n', 100*c);

c =

0.0938

cm =

16 2

1 13

Percentage Correct
Classification :
90.625000%

Percentage Incorrect
Classification :

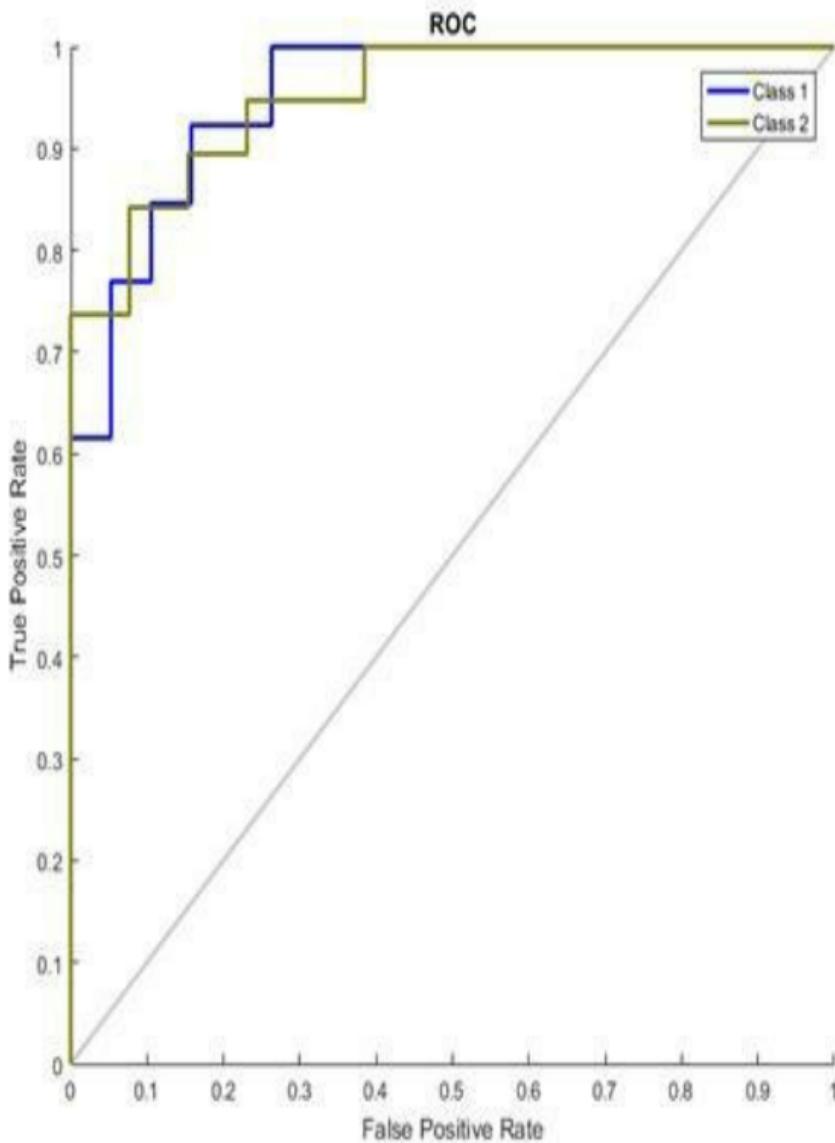
9.37500%

Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

Class 1 indicate cancer patients, class 2 normal patients.

```
plotroc(testT, testY)
```



This example illustrated how neural networks can be used as classifiers for cancer detection. One can also experiment using techniques like principal component analysis to reduce the dimensionality of the data to be used for building neural networks to improve classifier performance.

3.13 CHARACTER RECOGNITION

This example illustrates how to train a neural network to perform simple character recognition.

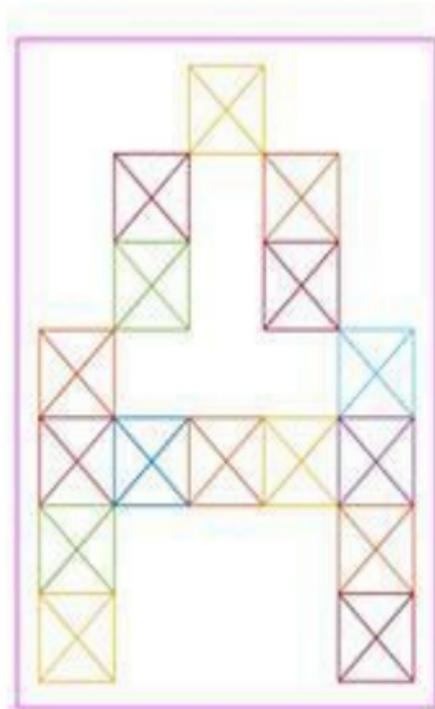
The script **prprob** defines a matrix X with 26 columns, one for each letter of the alphabet. Each column has 35 values which can either be 1 or 0. Each column of 35 values defines a 5x7 bitmap of a letter.

The matrix T is a 26x26 identity matrix which maps the 26 input vectors to the 26 classes.

[X,T] = prprob;

Here A, the first letter, is plotted as a bit map.

`plotchar(X(:,1))`



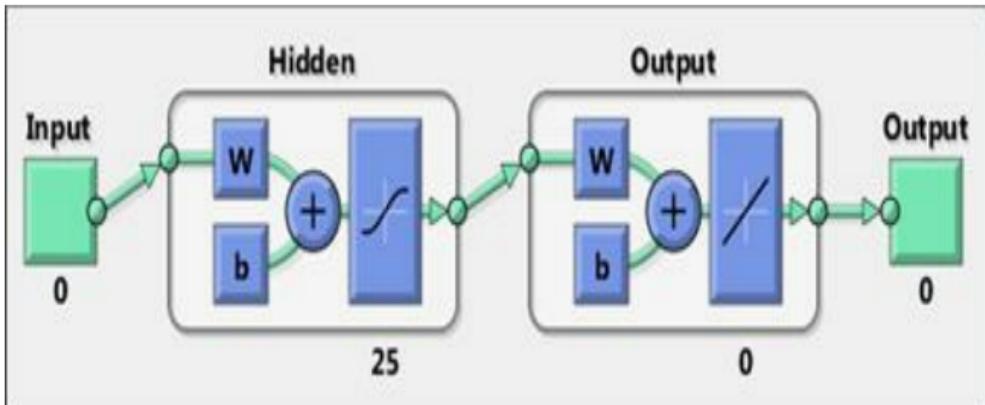
3.13.1 Creating the First Neural Network

To solve this problem we will use a feedforward neural network set up for pattern recognition with 25 hidden neurons.

Since the neural network is initialized with random initial weights, the results after training vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. This is not necessary for your own applications.

```
setdemorandstream(pi);
```

```
net1 = feedforwardnet(25);  
view(net1)
```



3.13.2 Training the first Neural Network

The function **train** divides up the data into training, validation and test sets. The training set is used to update the network, the validation set is used to stop the network before it overfits the training data, thus preserving good generalization. The test set acts as a completely independent measure of how well the network can be expected to do on new samples.

Training stops when the network is no longer likely to improve on the training or validation sets.

```
net1.divideFcn = "';  
net1 = train(net1,X,T,nnMATLAB);
```

Computing Resources: MATLAB on GLNXA64

3.13.3 Training the Second Neural Network

We would like the network to not only recognize perfectly formed letters, but also noisy versions of the letters. So we will try training a second network on noisy data and compare its ability to generalize with the first network.

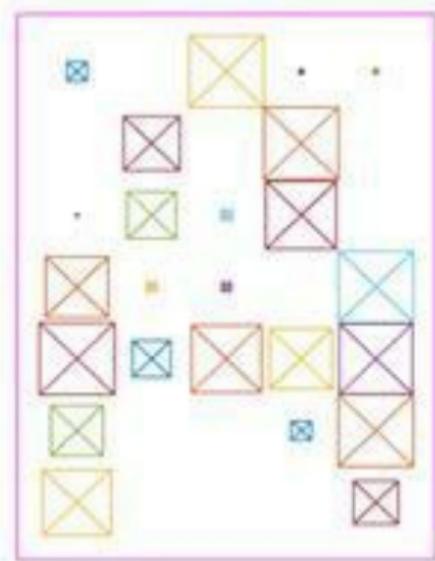
Here 30 noisy copies of each letter X_n are created. Values are limited by **min** and **max** to fall between 0 and 1. The corresponding targets T_n are also defined.

```
numNoise = 30;  
Xn =
```

```
min(max(repmat(X,1,numNoise)+randn(1,  
Tn = repmat(T,1,numNoise);
```

Here is a noise version of A.

```
figure  
plotchar(Xn(:,1))
```



Here the second network is created and

trained.

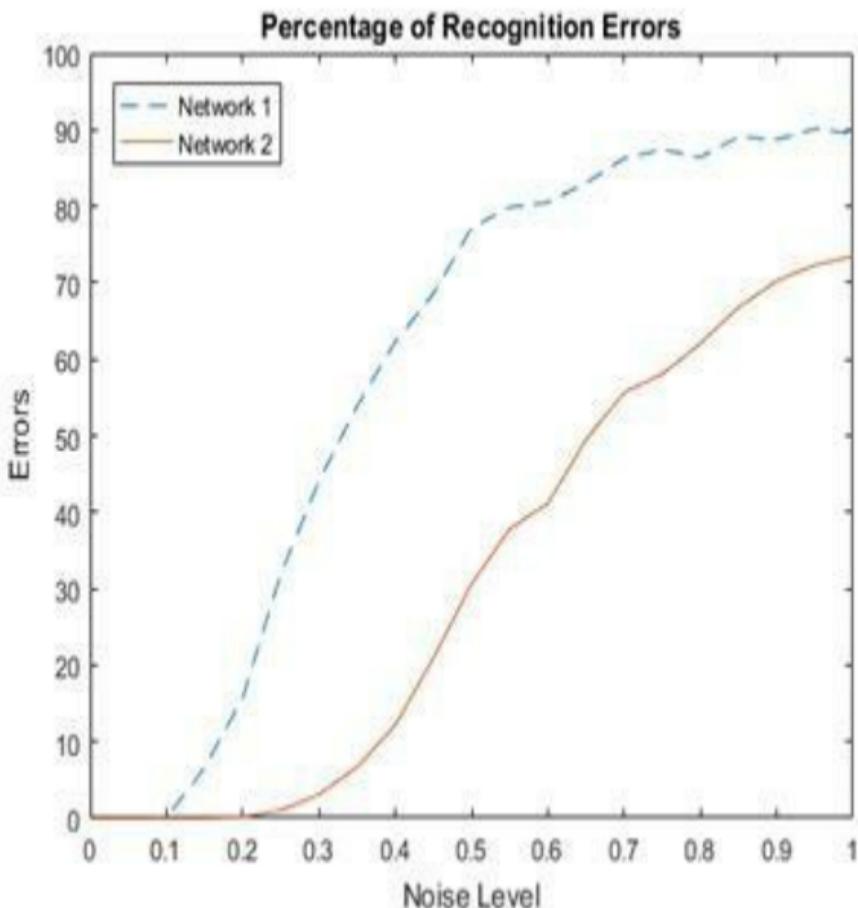
```
net2 = feedforwardnet(25);  
net2 = train(net2,Xn,Tn,nnMATLAB);
```

Computing Resources:
MATLAB on GLNXA64

3.13.4 Testing Both Neural Networks

```
noiseLevels = 0:.05:1;
numLevels = length(noiseLevels);
percError1 = zeros(1,numLevels);
percError2 = zeros(1,numLevels);
for i = 1:numLevels
    Xtest =
        min(max(repmat(X,1,numNoise)+randn(:,1)*noiseLevel));
    Y1 = net1(Xtest);
    percError1(i) = sum(sum(abs(Tn-compet(Y1))))/(26*numNoise^2);
    Y2 = net2(Xtest);
    percError2(i) = sum(sum(abs(Tn-compet(Y2))))/(26*numNoise^2);
end
```

```
figure  
plot(noiseLevels,percError1*100,'--  
' ,noiseLevels,percError2*100);  
title('Percentage of Recognition Errors');  
xlabel('Noise Level');  
ylabel('Errors');  
legend('Network 1','Network  
2','Location','NorthWest')
```



Network 1, trained without noise, has more errors due to noise than does Network 2, which was trained with noise.

3.14 LEARNING VECTOR QUANTIZATION (LVQ). EXAMPLE

An LVQ network is trained to classify input vectors according to given targets.

Let X be 10 2-element example input vectors and C be the classes these vectors fall into. These classes can be transformed into vectors to be used as targets, T, with IND2VEC.

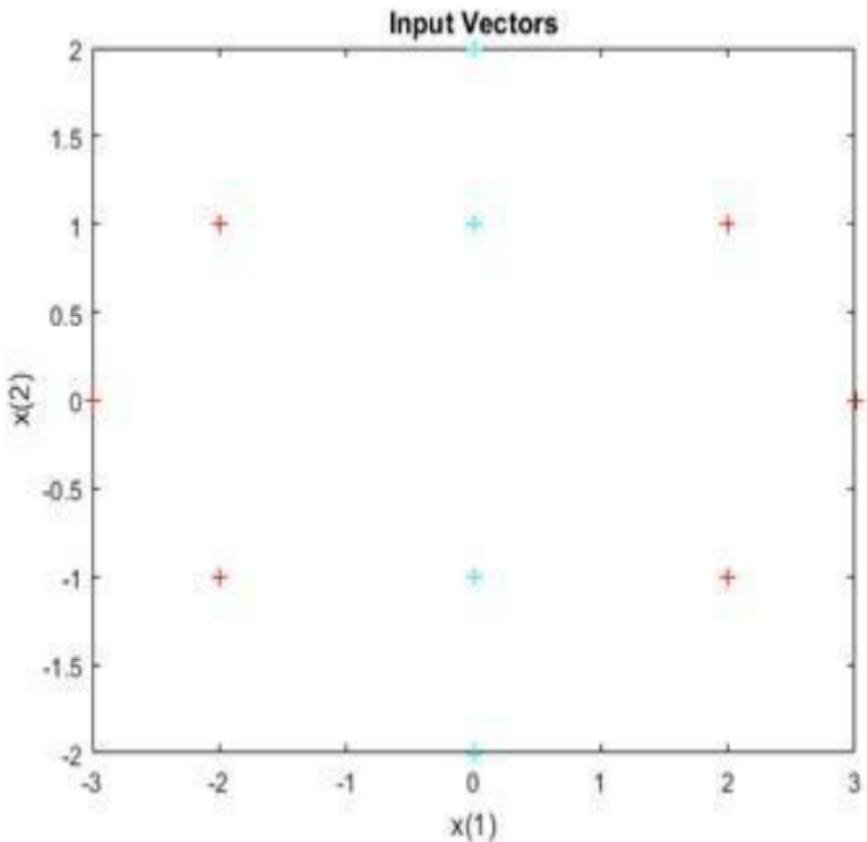
$x = [-3 -2 -2 \ 0 \ 0 \ 0 \ 0 +2 +2 +3;$
 $\quad \quad \quad 0 +1 -1 +2 +1 -1 -2 +1 -1 \ 0];$

$c = [1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2 \ 1 \ 1 \ 1];$

$t = \text{ind2vec}(c);$

Here the data points are plotted. Red = class 1, Cyan = class 2. The LVQ network represents clusters of vectors with hidden neurons, and groups the clusters with output neurons to form the desired classes.

```
colormap(hsv);
plotvec(x,c)
title('Input Vectors');
xlabel('x(1)');
ylabel('x(2)');
```



Here LVQNET creates an LVQ layer with four hidden neurons and a learning rate of 0.1. The network is then configured for inputs X and targets T. (Configuration normally an unnecessary

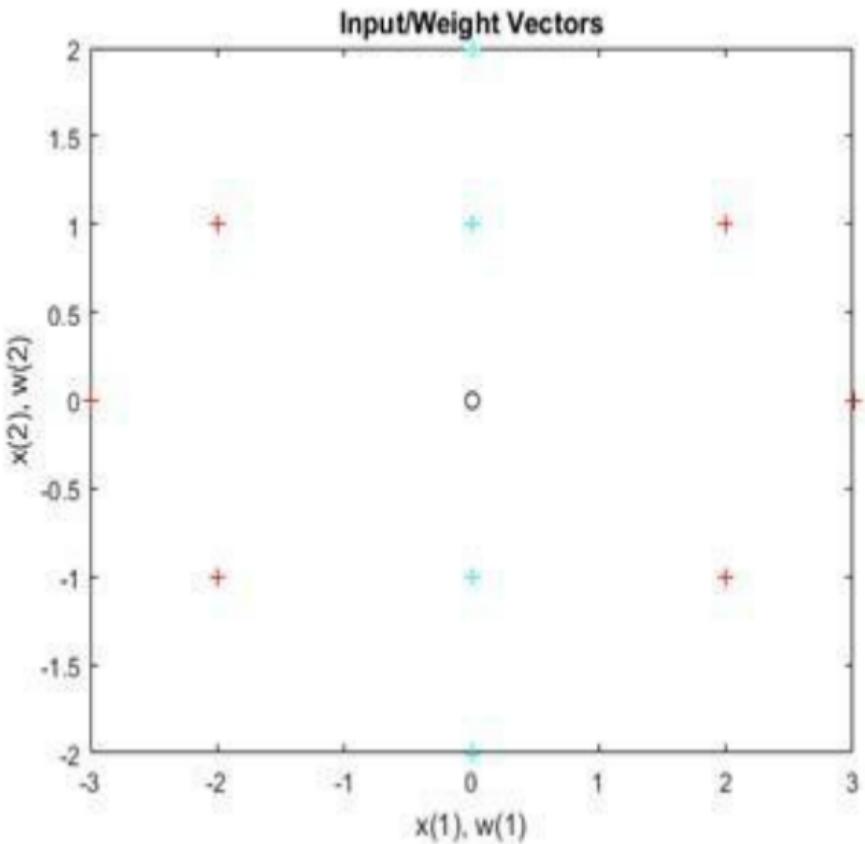
step as it is done automatically by TRAIN.)

```
net = lvqnet(4,0.1);
net = configure(net,x,t);
```

The competitive neuron weight vectors are plotted as follows.

hold on

```
w1 = net.IW{1};
plot(w1(1,1),w1(1,2),'ow')
title('Input/Weight Vectors');
xlabel('x(1), w(1)');
ylabel('x(2), w(2)');
```



To train the network, first override the default number of epochs, and then train the network. When it is finished, replot the input vectors '+' and the competitive neurons' weight vectors 'o'. Red = class

1, Cyan = class 2.

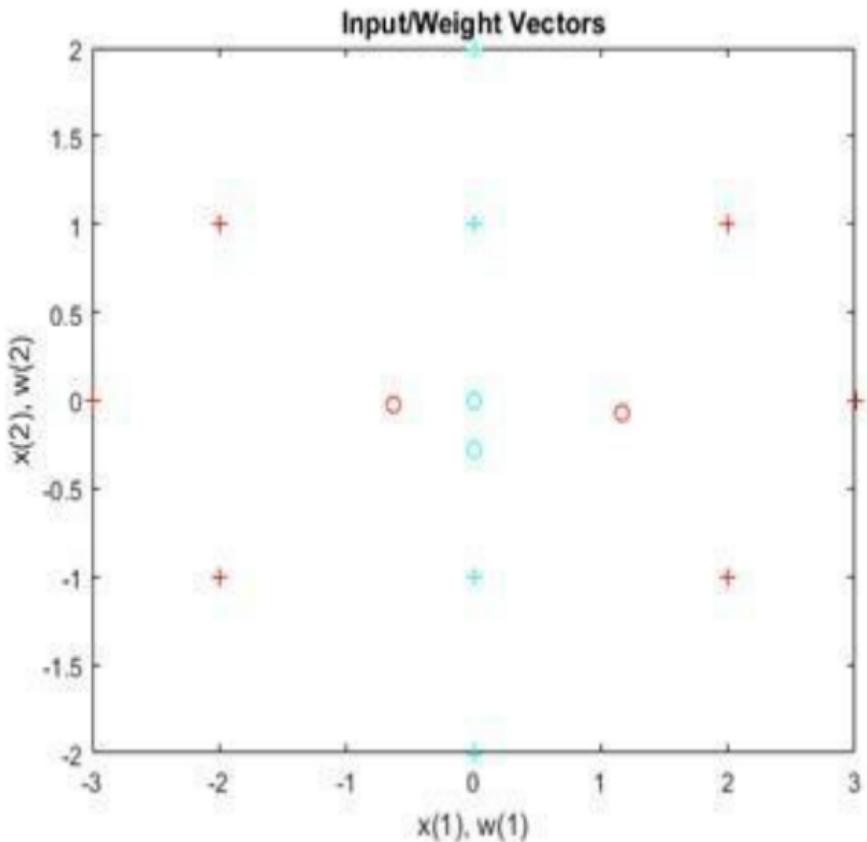
```
net.trainParam.epochs=150;  
net=train(net,x,t);
```

```
cla;
```

```
plotvec(x,c);
```

```
hold on;
```

```
plotvec(net.IW{1}',vec2ind(net.LW{2})),
```



Now use the LVQ network as a classifier, where each neuron corresponds to a different category. Present the input vector $[0.2; 1]$. Red = class 1, Cyan = class 2.

x1 = [0.2; 1];

y1 = vec2ind(net(x1))

y1 =

2

Chapter 4

ADAPTATIVE LINEAR FILTERS

4.1 ADAPTATIVE FILTERS

An **adaptive filter** is a system with a linear **filter** that has a **transfer function** controlled by variable parameters and a means to adjust those parameters according to an **optimization algorithm**. Because of the complexity of the optimization algorithms, almost all adaptive filters are **digital filters**. Adaptive filters are required for some applications because some parameters of the desired processing operation (for instance, the locations of reflective surfaces in a **reverberant space**) are not known in advance or are changing. The closed loop adaptive filter uses

feedback in the form of an error signal to refine its transfer function.

Generally speaking, the closed loop adaptive process involves the use of a **cost function**, which is a criterion for optimum performance of the filter, to feed an algorithm, which determines how to modify filter transfer function to minimize the cost on the next iteration. The most common cost function is the mean square of the error signal.

As the power of **digital signal processors** has increased, adaptive filters have become much more common and are now routinely used in devices such as mobile phones and other communication devices, camcorders and

digital cameras, and medical monitoring equipment.

4.2 PATTERN ASSOCIATION SHOWING ERROR SURFACE

A linear neuron is designed to respond to specific inputs with target outputs.

X defines two 1-element input patterns (column vectors). T defines the associated 1-element targets (column vectors).

$$X = [1.0 \ -1.2]; \\ T = [0.5 \ 1.0];$$

ERRSURF calculates errors for y neuron with y range of possible weight and bias

values. PLOTES plots this error surface with y contour plot underneath. The best weight and bias values are those that result in the lowest point on the error surface.

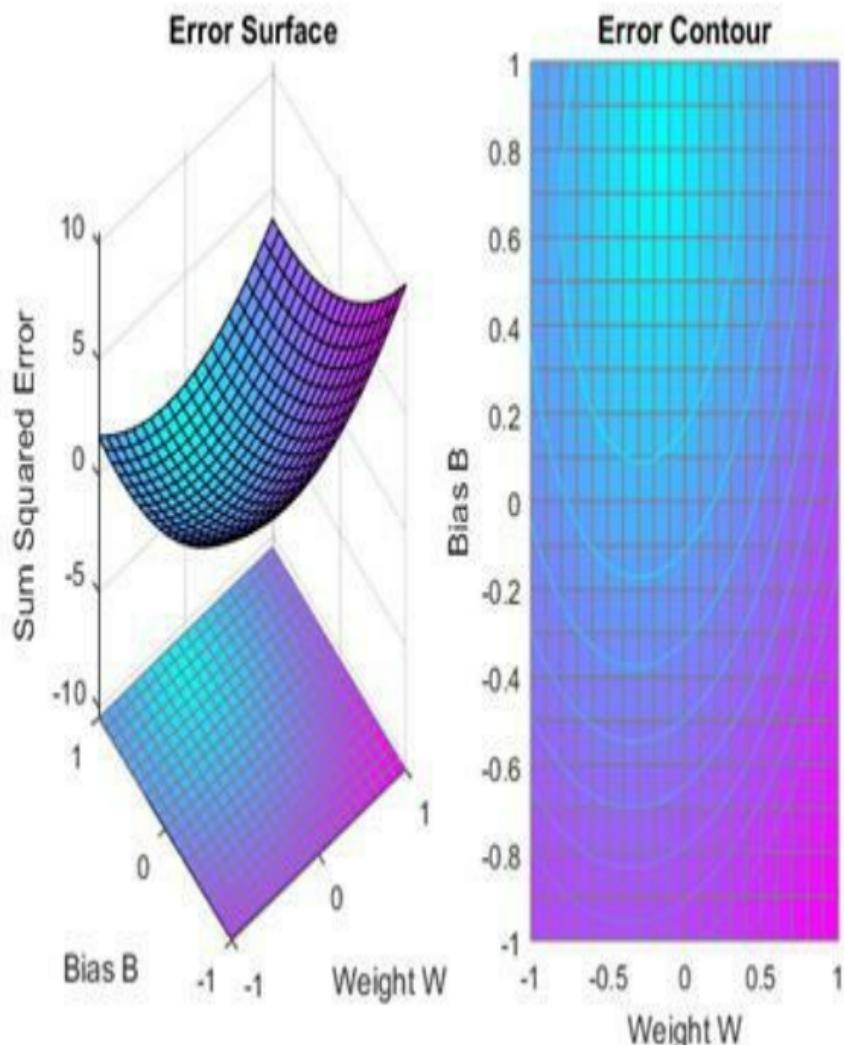
```
w_range = -1:0.1:1;
```

```
b_range = -1:0.1:1;
```

```
ES =
```

```
errsurf(X,T,w_range,b_range,'purelin');
```

```
plotes(w_range,b_range,ES);
```



The function **NEWLIND** will design a network that performs with the minimum

error.

```
net = newlind(X,T);
```

SIM is used to simulate the network for inputs X. We can then calculate the neurons errors. SUMSQR adds up the squared errors.

```
A = net(X)
```

```
E = T - A
```

```
SSE = sumsqr(E)
```

```
A =
```

```
0.5000 1.0000
```

```
E =
```

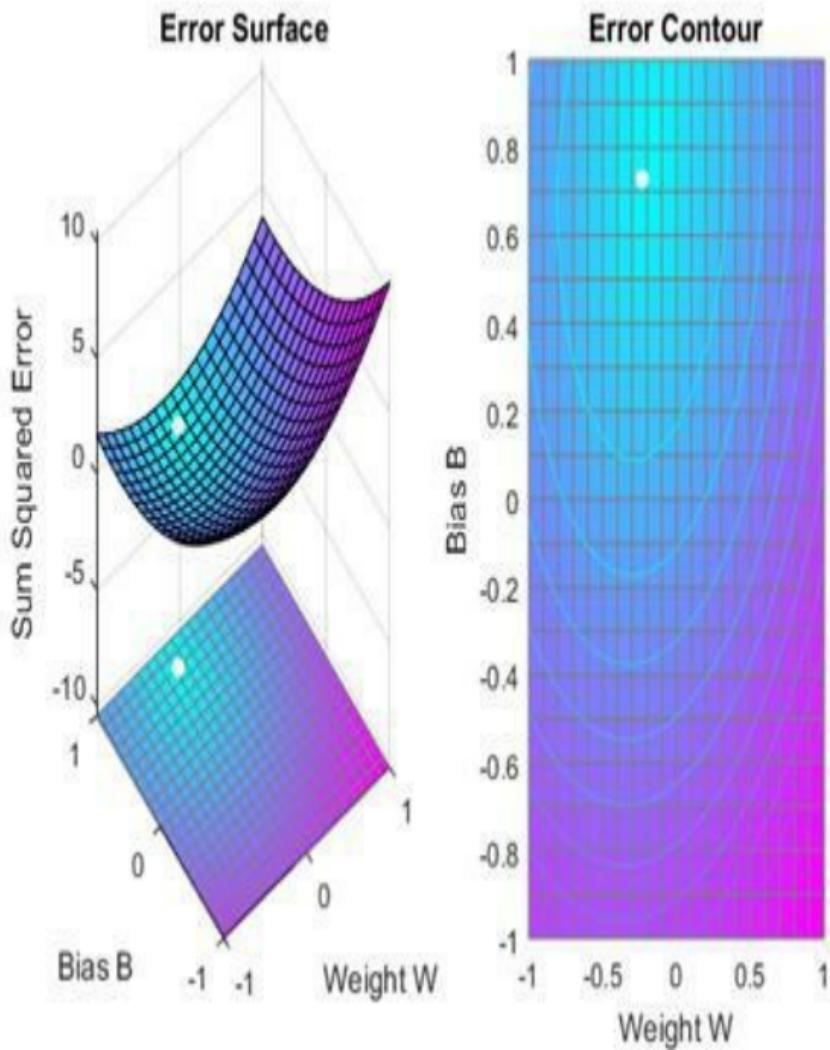
0 0

SSE =

0

PLOTES replots the error surface.
PLOTEP plots the "position" of the
network using the weight and bias values
returned by SOLVELIN. As can be seen
from the plot, SOLVELIN found the
minimum error solution.

```
plotes(w_range,b_range,ES);
plotep(net.IW{1,1},net.b{1},SSE);
```



We can now test the associator with one of the original inputs, -1.2, and see if it

returns the target, 1.0.

x = -1.2;

y = net(x)

y =

1

4.3 TRAINING A LINEAR NEURON

A linear neuron is trained to respond to specific inputs with target outputs.

X defines two 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors). A single input linear neuron with y bias can be used to solve this problem.

$$X = [1.0 \ -1.2];$$

$$T = [0.5 \ 1.0];$$

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath. The best

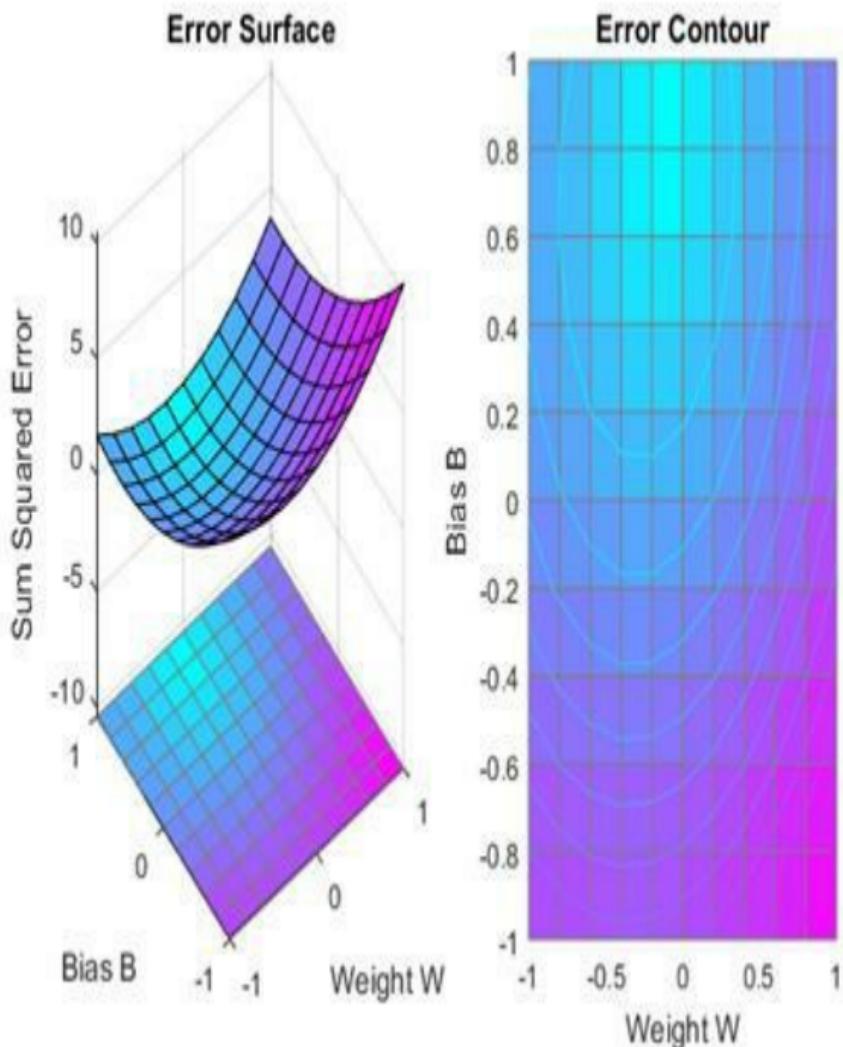
weight and bias values are those that result in the lowest point on the error surface.

w_range = -1:0.2:1; b_range = -1:0.2:1;

ES =

errsurf(X,T,w_range,b_range,'purelin');

plotes(w_range,b_range,ES);



MAXLINLR finds the fastest stable learning rate for training y linear

network. For this example, this rate will only be 40% of this maximum. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

```
maxlr = 0.40*maxlinlr(X,'bias');  
net = newlin([-2 2],1,[0],maxlr);
```

Override the default training parameters by setting the performance goal.

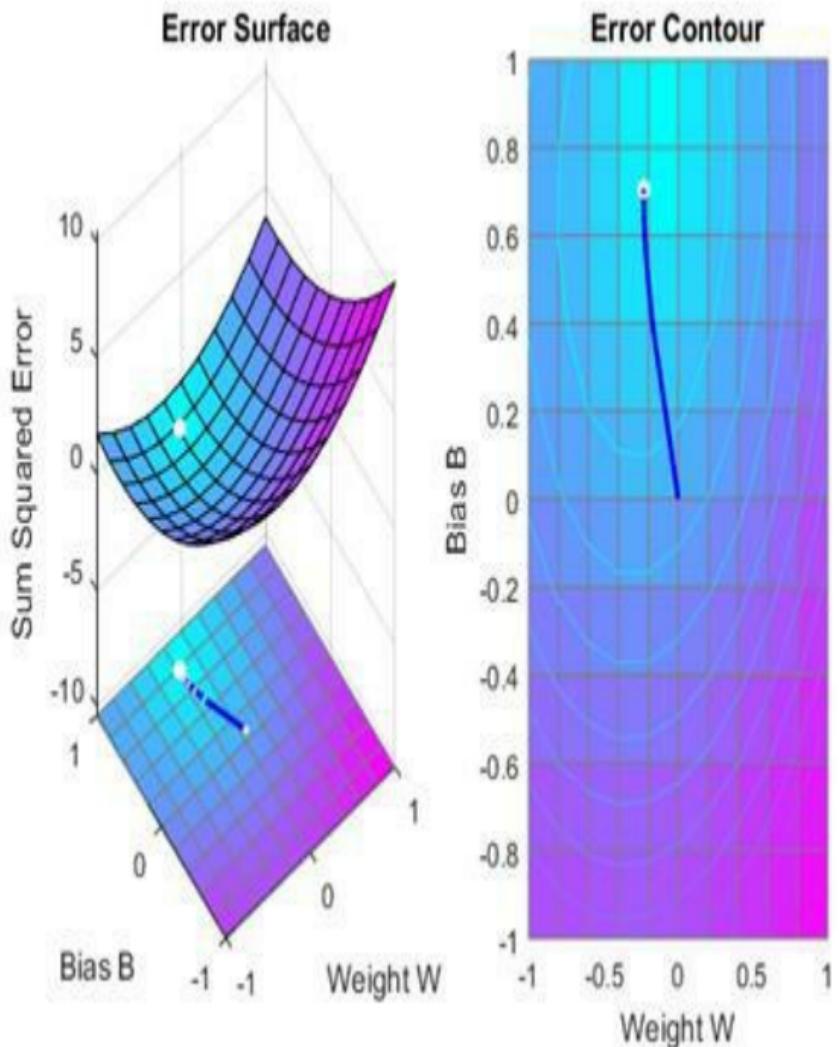
```
net.trainParam.goal = .001;
```

To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch. The plot shows y

history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-
net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while true
    epoch = epoch+1;
    [net,tr] = train(net,X,T);
    if length(tr.epoch) > 1
        h =
```

```
plotep(net.IW{1,1},net.b{1},tr.perf(2),h)
r.epoch=[r.epoch epoch];
r.perf=[r.perf tr.perf(2)];
r.vperf=[r.vperf NaN];
r.tperf=[r.tperf NaN];
else
    break
end
end
tr=r;
```

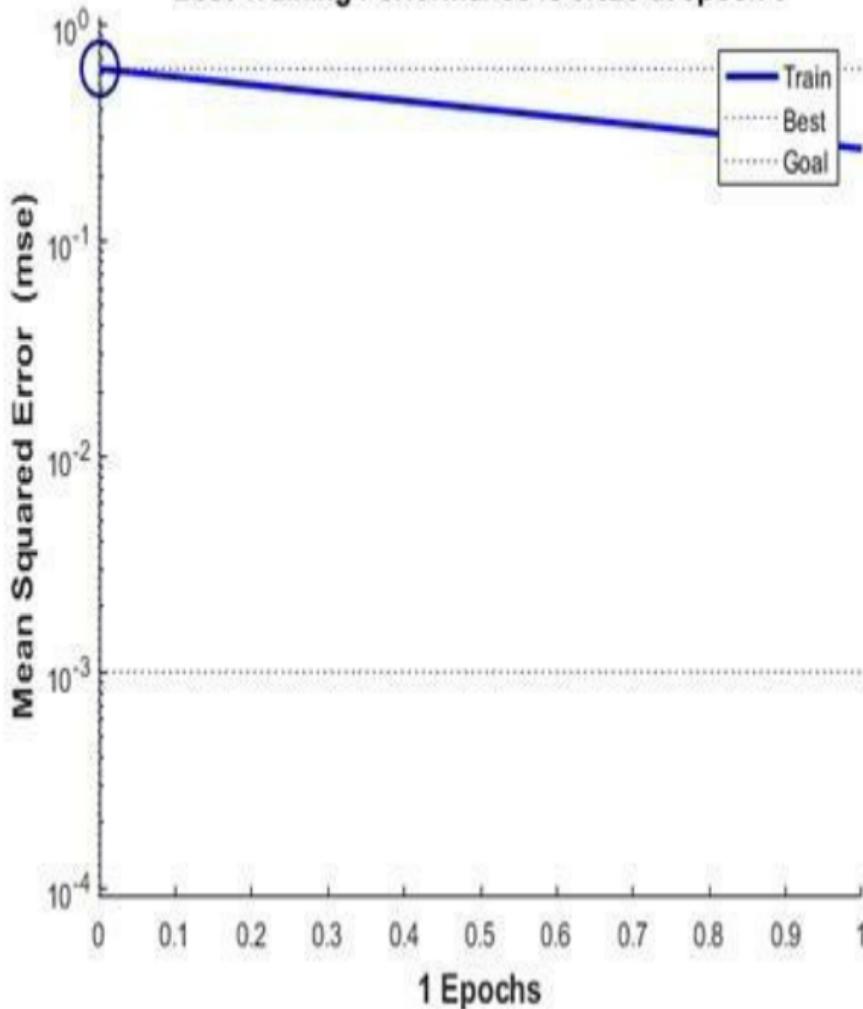


The train function outputs the trained network and y history of the training

performance (tr). Here the errors are plotted with respect to training epochs: The error dropped until it fell beneath the error goal (the black line). At that point training stopped.

```
plotperform(tr);
```

Best Training Performance is 0.625 at epoch 0



Now use SIM to test the associator with one of the original inputs, -1.2, and see

if it returns the target, 1.0. The result is very close to 1, the target. This could be made even closer by lowering the performance goal.

$x = -1.2;$

$y = \text{net}(x)$

$y =$

0.9817

4.4 ADAPTIVE NOISE CANCELLATION

A linear neuron is allowed to adapt so that given one signal, it can predict a second signal.

TIME defines the time steps of this simulation. P defines a signal over these time steps. T is a signal derived from P by shifting it to the left, multiplying it by 2 and adding it to itself.

time = 1:0.01:2.5;

X = sin(sin(time).*time*10);

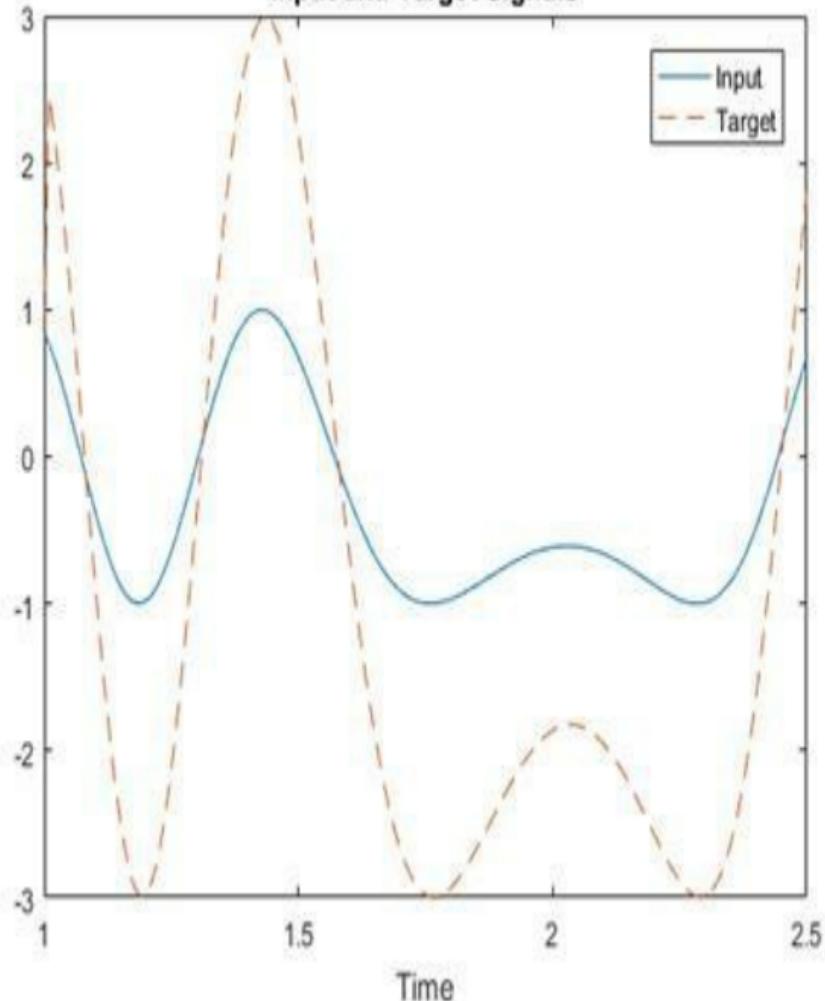
P = con2seq(X);

T = con2seq(2*[0 X(1:(end-1))] + X);

Here is how the two signals are plotted:

```
plot(time,cat(2,P{:}),time,cat(2,T{:}),'--')
title('Input and Target Signals')
xlabel('Time')
legend({'Input','Target'})
```

Input and Target Signals



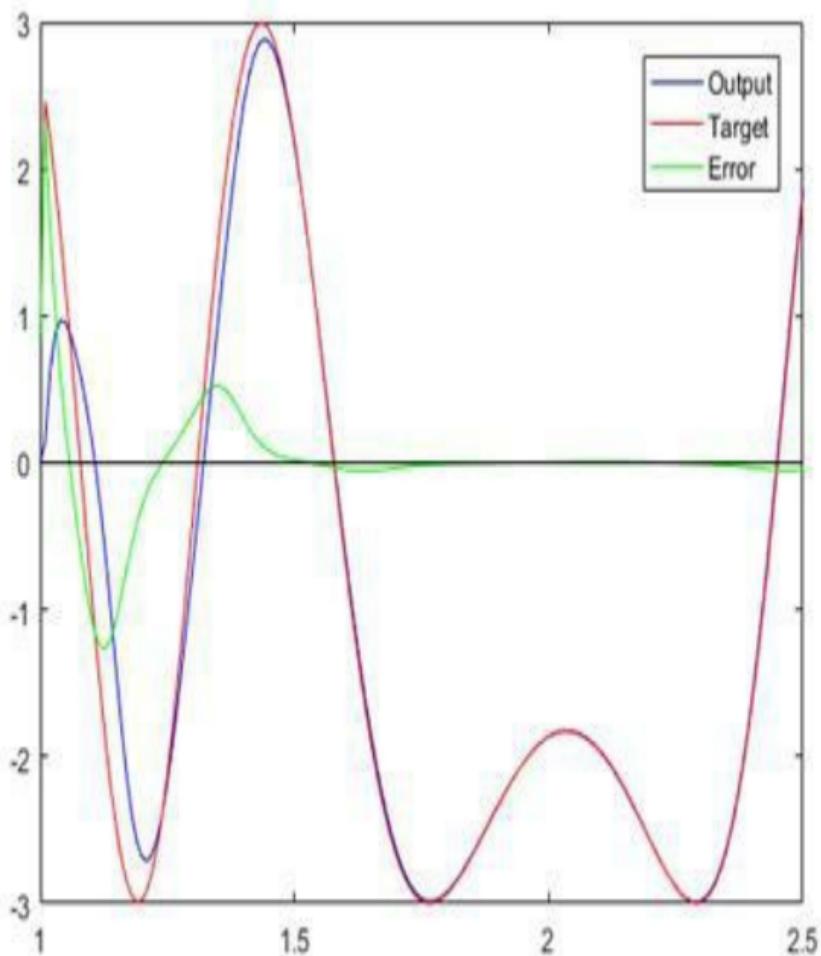
The linear network must have tapped delay in order to learn the time-shifted

correlation between P and T. NEWLIN creates a linear layer. [-3 3] is the expected input range. The second argument is the number of neurons in the layer. [0 1] specifies one input with no delay and one input with a delay of one. The last argument is the learning rate.

```
net = newlin([-3 3],1,[0 1],0.1);
```

ADAPT simulates adaptive networks. It takes a network, a signal, and a target signal, and filters the signal adaptively. Plot the output Y in blue, the target T in red and the error E in green. By t=2 the network has learned the relationship between the input and the target and the error drops to near zero.

```
[net,Y,E,Pf]=adapt(net,P,T);
plot(time,cat(2,Y{:}),'b', ...
      time,cat(2,T{:}),'r', ...
      time,cat(2,E{:}),'g',[1 2.5],[0 0],'k')
legend({'Output','Target','Error'})
```



4.5 LINEAR FIT OF NONLINEAR PROBLEM

A linear neuron is trained to find the minimum sum-squared error linear fit to y nonlinear input/output problem.

X defines four 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors). Note that the relationship between values in X and in T is nonlinear. I.e. No W and B exist such that $X^*W+B = T$ for all of four sets of X and T values above.

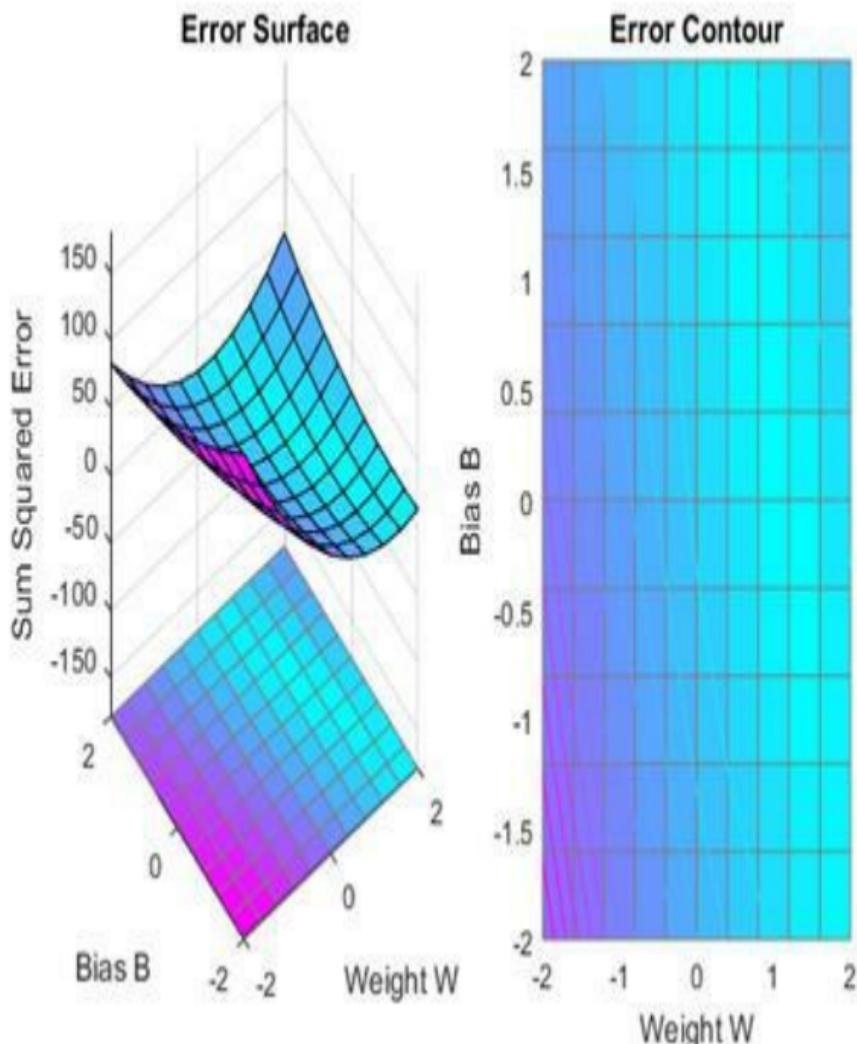
```
X = [+1.0 +1.5 +3.0  
-1.2];
```

```
T = [+0.5 +1.1 +3.0  
-1.0];
```

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath.

The best weight and bias values are those that result in the lowest point on the error surface. Note that because y perfect linear fit is not possible, the minimum has an error greater than 0.

```
w_range = -2:0.4:2;  
b_range = -2:0.4:2;  
  
ES =  
errsurf(X,T,w_range,]  
  
plotes(w_range,b_ran
```



MAXLINLR finds the fastest stable learning rate for training y linear

network. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

```
maxlr =  
maxlinlr(X, 'bias');
```

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

Override the default training parameters by setting the maximum number of epochs. This ensures that training will

stop.

```
net.trainParam.epoch  
= 15;
```

To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch (code not shown here). The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [net,tr] =  
train(net,X,T);
```

```
net.trainParam.epoch  
= 1;
```

```
net.trainParam.show  
= NaN;
```

```
h=plotep (net.IW{1}, ne  
net(X)) );
```

```
[net, tr] =  
train(net, X, T);
```

r = tr;

epoch = 1;

while epoch < 15

epoch = epoch+1;

[net, tr] =
train(net, X, T);

if
length(tr.epoch) > 1

```
h =
plotep(net.IW{1,1},n
r.epoch=
[r.epoch epoch];
r.perf=[r.perf
tr.perf(2)];
r.vperf=
[r.vperf NaN];
```

```
r.tperf= [r.tperf NaN];
```

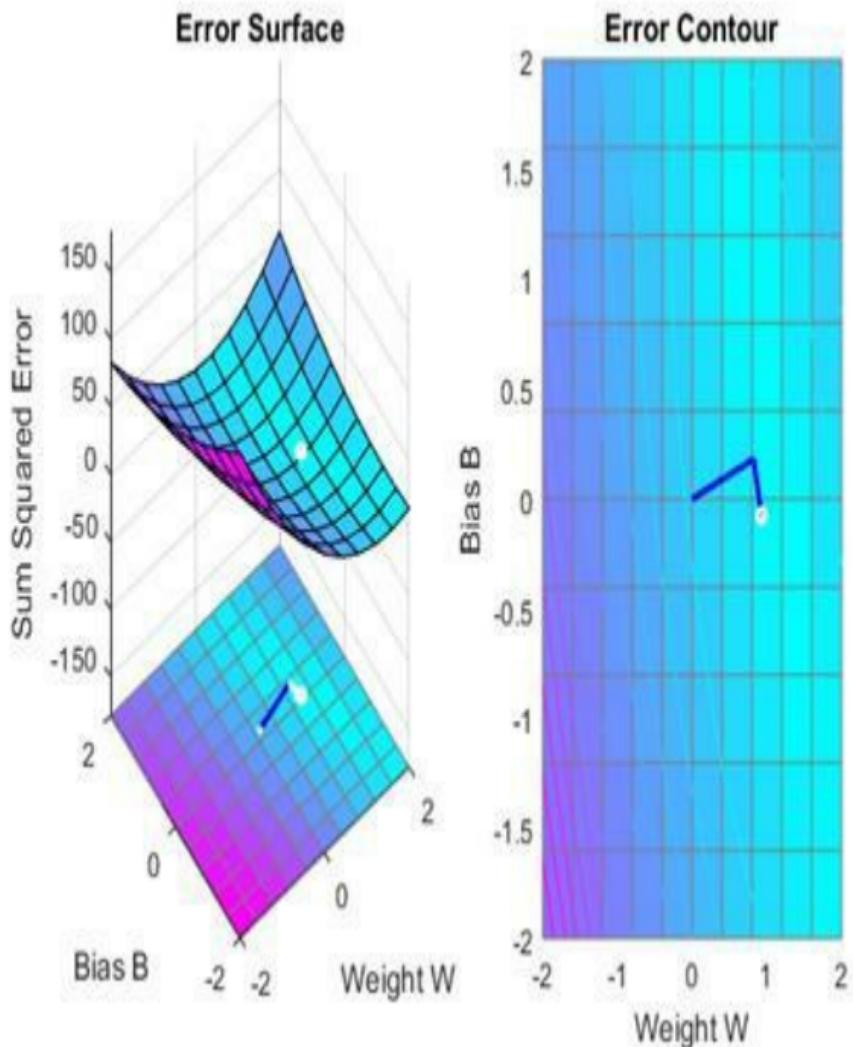
```
else
```

```
break
```

```
end
```

```
end
```

```
tr=r;
```



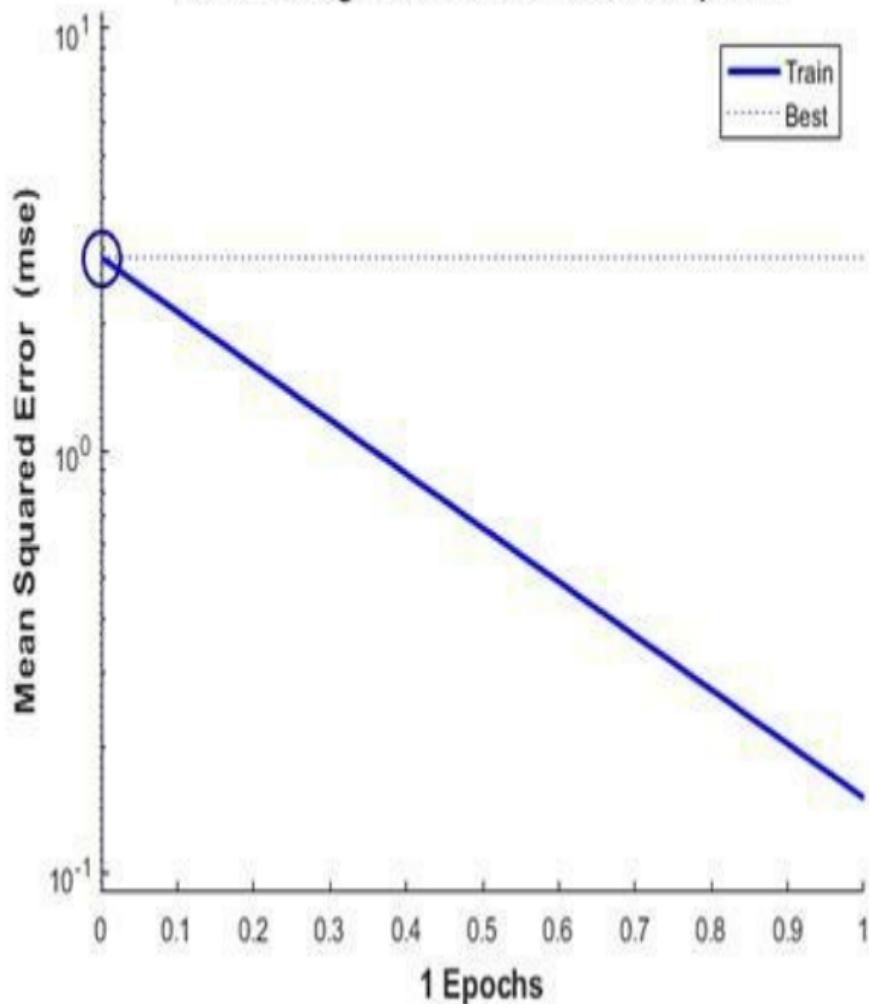
The train function outputs the trained network and y history of the training

performance (tr). Here the errors are plotted with respect to training epochs.

Note that the error never reaches 0. This problem is nonlinear and therefore zero error linear solution is not possible.

```
plotperform(tr);
```

Best Training Performance is 2.865 at epoch 0



Now use SIM to test the associator with one of the original inputs, -1.2, and see

if it returns the target, 1.0.

The result is not very close to 0.5! This is because the network is the best linear fit to y nonlinear problem.

```
x = -1.2;
```

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

```
y =
```

-1.1803

4.6 UNDERDETERMINED PROBLEM

A linear neuron is trained to find y non-unique solution to an undetermined problem.

X defines one 1-element input patterns (column vectors). T defines an associated 1-element target (column vectors). Note that there are infinite values of W and B such that the expression $W^*X+B = T$ is true.
Problems with multiple solutions are called underdetermined.

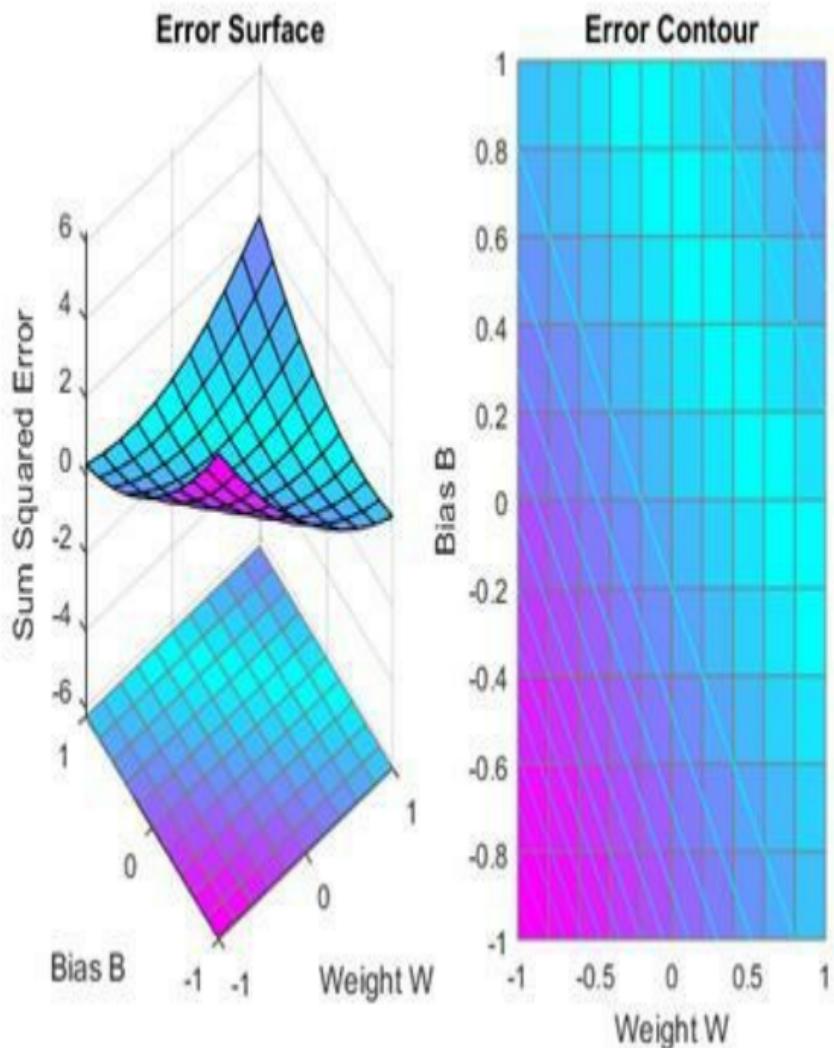
X = [+1.0];

T = [+0.5];

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath. The bottom of the valley in the error surface corresponds to the infinite solutions to this problem.

w_range = -1:0.2:1;

```
b_range = -1:0.2:1;  
  
ES =  
errsurf(X,T,w_range,]  
  
plotes(w_range,b_range)
```



MAXLINLR finds the fastest stable learning rate for training y linear

network. NEWLIN creates y linear neuron. NEWLIN takes these arguments:
1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

```
maxlr =  
maxlinlr(X, 'bias');
```

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

Override the default training parameters by setting the performance goal.

```
net.trainParam.goal  
= 1e-10;
```

To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch. The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [net,tr] =  
train(net,X,T);
```

```
net.trainParam.epoch  
= 1;  
  
net.trainParam.show  
= NaN;  
  
h=plotep(net.IW{1},ne  
net(X)) );  
  
[net,tr] =  
train(net,X,T);  
  
r = tr;
```

```
epoch = 1;
```

```
while true
```

```
    epoch = epoch+1;
```

```
    [net,tr] =  
    train(net,X,T);
```

```
    if
```

```
        length(tr.epoch) > 1
```

```
h =
plotep(net.IW{1,1},n
r.epoch=
[r.epoch epoch];
r.perf=[r.perf
tr.perf(2)];
r.vperf=
[r.vperf NaN];
r.tperf=
```

[r.tperf NaN];

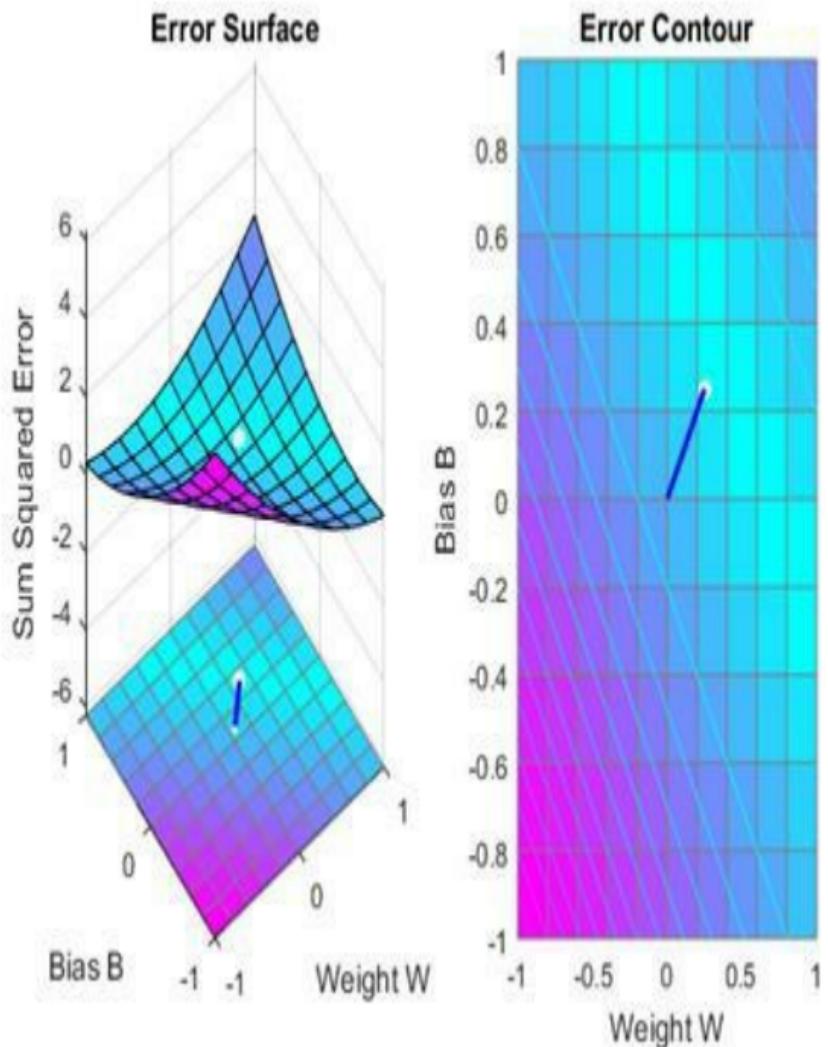
else

break

end

end

tr=r;



Here we plot the NEWLIND solution.
Note that the TRAIN (white dot) and

SOLVELIN (red circle) solutions are not the same. In fact, TRAINWH will return y different solution for different initial conditions, while SOLVELIN will always return the same solution.

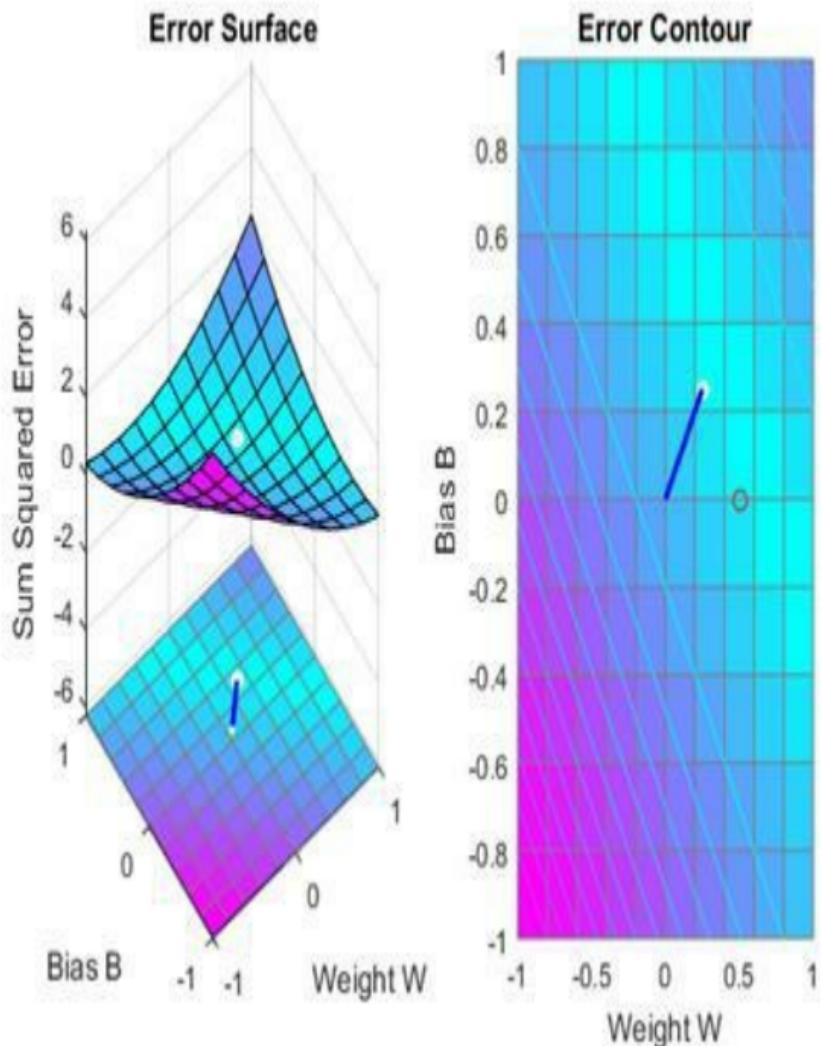
```
solvednet =
```

```
newlind(X, T);
```

```
hold on;
```

```
plot(solvednet.IW{1,
```

```
hold off;
```

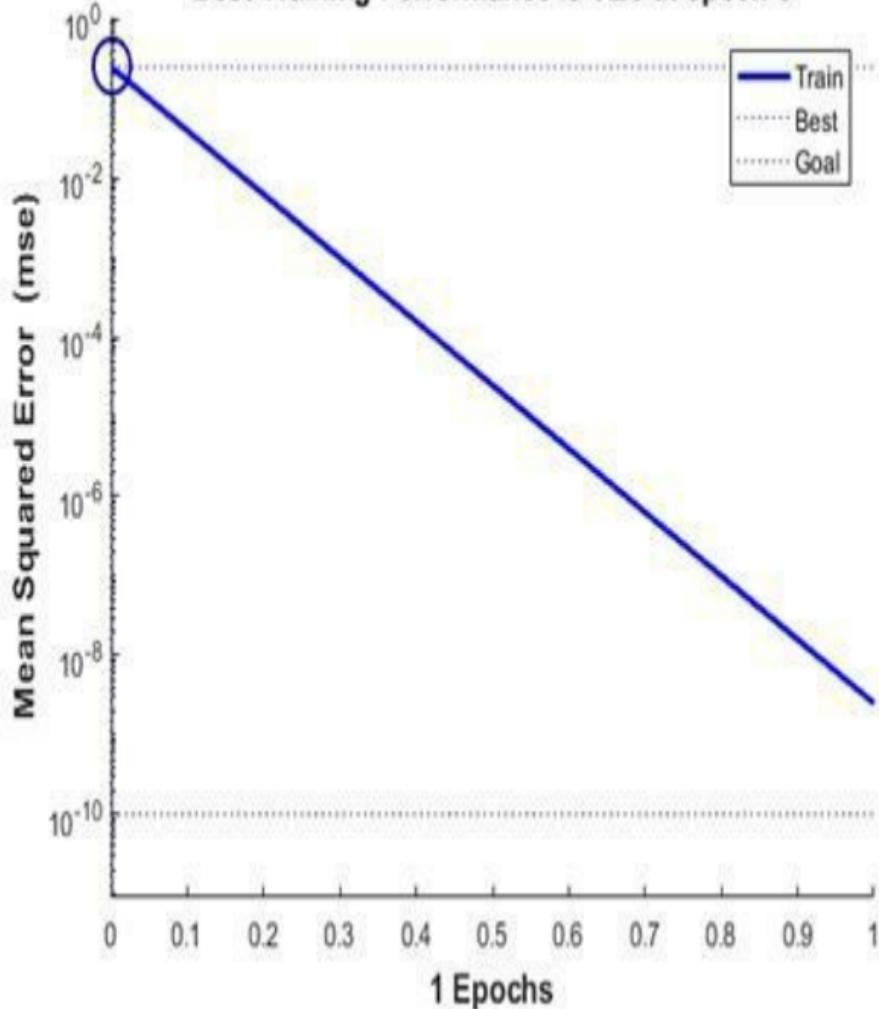


The train function outputs the trained network and y history of the training

performance (tr). Here the errors are plotted with respect to training epochs: Once the error reaches the goal, an adequate solution for W and B has been found. However, because the problem is underdetermined, this solution is not unique.

```
subplot(1,2,1);  
plotperform(tr);
```

Best Training Performance is 0.25 at epoch 0



We can now test the associator with one of the original inputs, 1.0, and see if it

returns the target, 0.5. The result is very close to 0.5. The error can be reduced further, if required, by continued training with TRAINWH using a smaller error goal.

```
x = 1.0;
```

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

```
y =
```

0 . 5000

4.7 LINEARLY DEPENDENT PROBLEM

A linear neuron is trained to find the minimum error solution for y problem with linearly dependent input vectors. If y linear dependence in input vectors is not matched in the target vectors, the problem is nonlinear and does not have y zero error linear solution.

X defines three 2-element input patterns (column vectors). Note that 0.5 times the sum of (column) vectors 1 and 3 results in vector 2. This is called linear dependence.

```
X = [ 1.0  2.0  3.0; ...
      4.0  5.0  6.0];
```

T defines an associated 1-element target (column vectors). Note that 0.5 times the sum of -1.0 and 0.5 does not equal 1.0. Because the linear dependence in X is not matched in T this problem is nonlinear and does not have y zero error linear solution.

```
T = [0.5 1.0 -1.0];
```

MAXLINLR finds the fastest stable learning rate for TRAINWH. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector,

3) Input delay vector, and 4) Learning rate.

```
maxlr = maxlinlr(X,'bias');  
net = newlin([0 10;0 10],1,[0],maxlr);
```

TRAIN uses the Widrow-Hoff rule to train linear networks by default. We will display each 50 epochs and train for a maximum of 500 epochs.

```
net.trainParam.show = 50; %
```

Frequency of progress displays (in epochs).

```
net.trainParam.epochs = 500; %
```

Maximum number of epochs to train.

```
net.trainParam.goal = 0.001; % Sum-squared error goal.
```

Now the network is trained on the inputs

X and targets T. Note that, due to the linear dependence between input vectors, the problem did not reach the error goal represented by the black line.

[net,tr] = train(net,X,T);

We can now test the associator with one of the original inputs, [1; 4], and see if it returns the target, 0.5. The result is not 0.5 as the linear network could not fit the nonlinear problem caused by the linear dependence between input vectors.

p = [1.0; 4];

y = net(p)

y =

0.8971

4.8 TOO LARGE A LEARNING RATE

A linear neuron is trained to find the minimum error solution for a simple problem. The neuron is trained with the learning rate larger than the one suggested by MAXLINLR.

X defines two 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors).

$$X = [+1.0 \ -1.2];$$

$$T = [+0.5 \ +1.0];$$

ERRSURF calculates errors for a neuron with a range of possible weight and bias values. PLOTES plots this error surface

with a contour plot underneath. The best weight and bias values are those that result in the lowest point on the error surface.

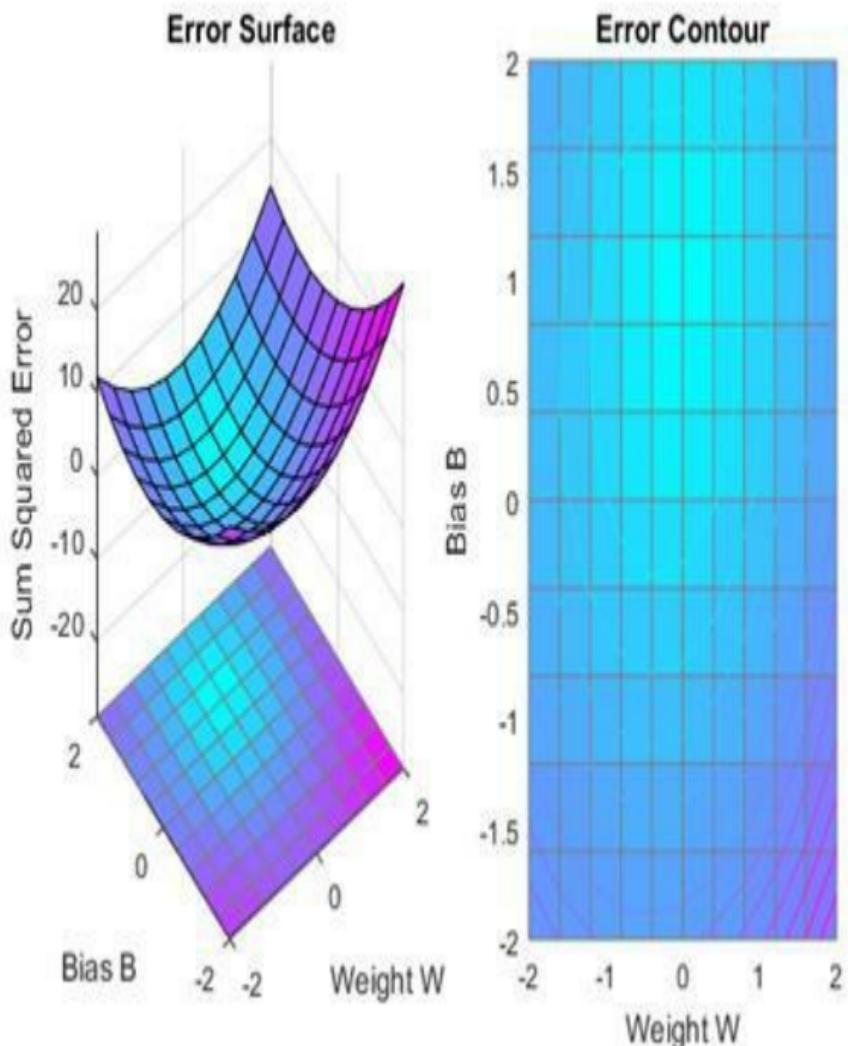
```
w_range = -2:0.4:2;
```

```
b_range = -2:0.4:2;
```

```
ES =
```

```
errsurf(X,T,w_range,b_range,'purelin');
```

```
plotes(w_range,b_range,ES);
```



MAXLINLR finds the fastest stable learning rate for training a linear

network. NEWLIN creates a linear neuron. To see what happens when the learning rate is too large, increase the learning rate to 225% of the recommended value. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

```
maxlr = maxlinlr(X,'bias');  
net = newlin([-2 2],1,[0],maxlr*2.25);
```

Override the default training parameters by setting the maximum number of epochs. This ensures that training will stop:

```
net.trainParam.epochs = 20;
```

To show the path of the training we will train only one epoch at a time and call PLOTEP every epoch (code not shown here). The plot shows a history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
%[net,tr] = train(net,X,T);
```

```
net.trainParam.epochs = 1;
```

```
net.trainParam.show = NaN;
```

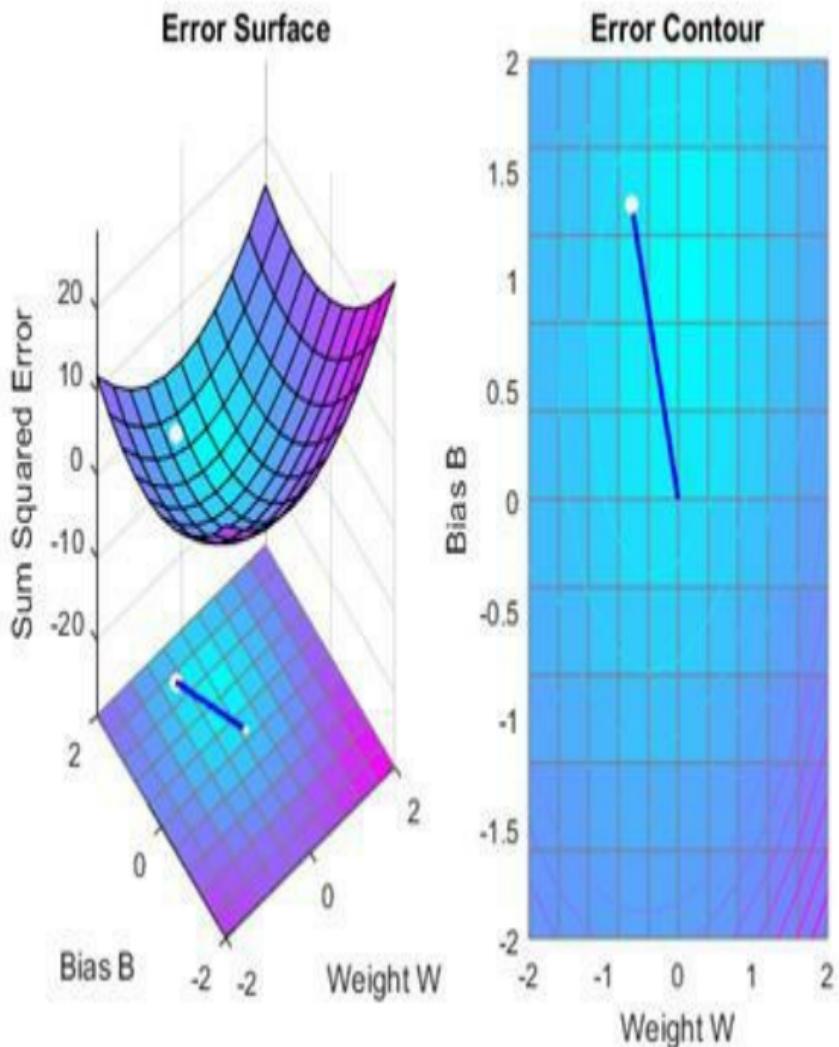
```
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
```

```
[net,tr] = train(net,X,T);
```

```
r = tr;
```

```
epoch = 1;
```

```
while epoch < 20
    epoch = epoch+1;
    [net,tr] = train(net,X,T);
    if length(tr.epoch) > 1
        h =
        plotep(net.IW{1,1},net.b{1},tr.perf(2),h)
        r.epoch=[r.epoch epoch];
        r.perf=[r.perf tr.perf(2)];
        r.vperf=[r.vperf NaN];
        r.tperf=[r.tperf NaN];
    else
        break
    end
end
tr=r;
```

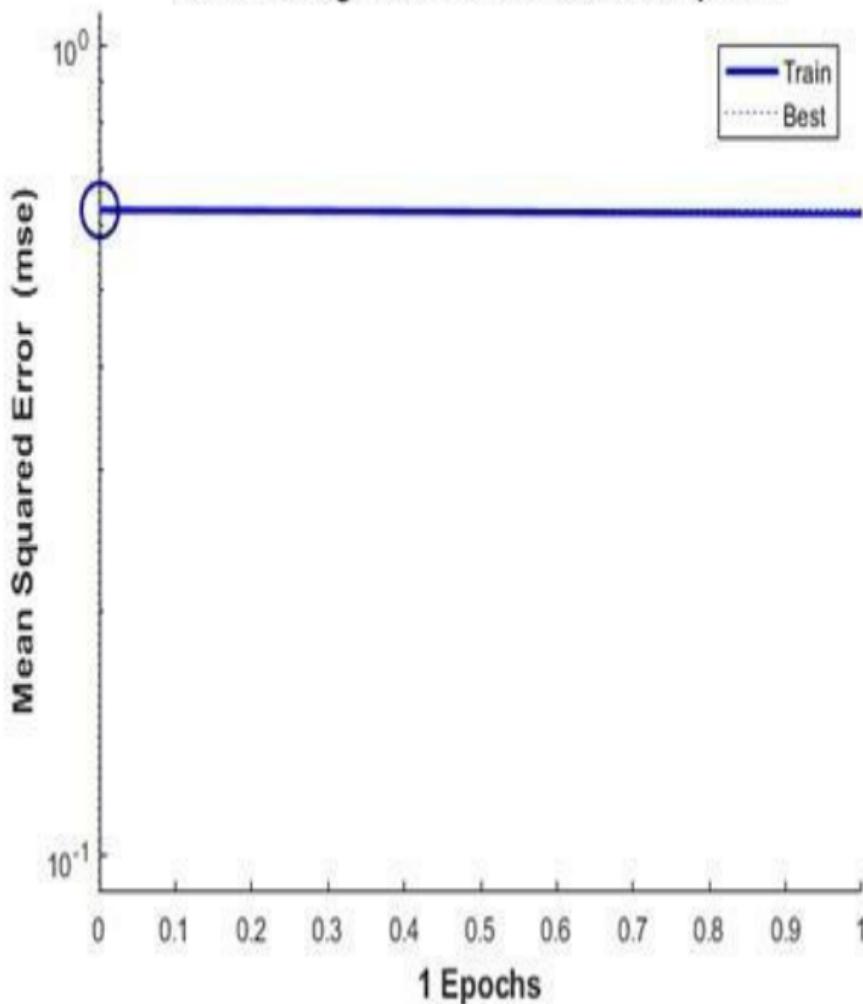


The train function outputs the trained network and a history of the training

performance (tr). Here the errors are plotted with respect to training epochs.

```
plotperform(tr);
```

Best Training Performance is 0.625 at epoch 0



We can now use SIM to test the associator with one of the original

inputs, -1.2, and see if it returns the target, 1.0. The result is not very close to 0.5! This is because the network was trained with too large a learning rate.

$x = -1.2;$

$y = \text{net}(x)$

$y =$

2.0913

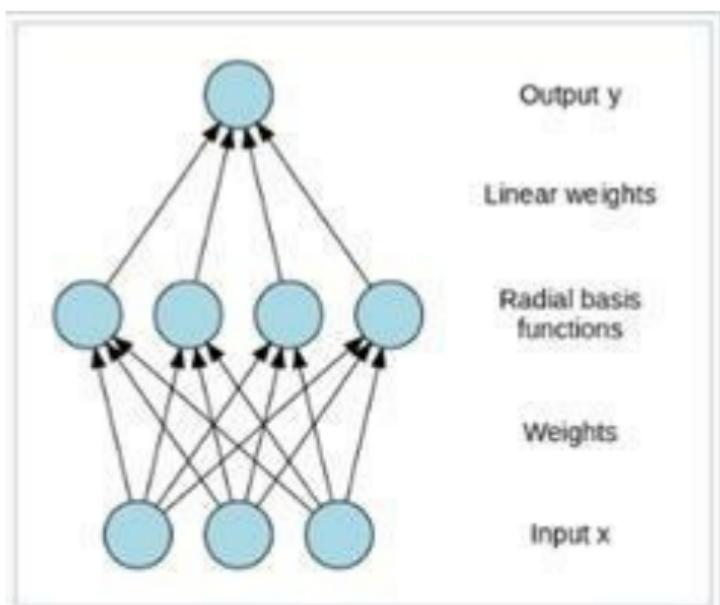
Chapter 5

RADIAL BASIS NETWORKS

5.1 RADIAL BASIS FUNCTION NETWORK

In the field of mathematical modeling, a **radial basis function network** is an artificial neural network that uses radial basis functions as activation functions. The output of the network is a linear combination of radial basis functions of the inputs and neuron parameters. Radial basis function networks have many uses, including function approximation, time series prediction, classification, and system control. They were first formulated in a 1988 paper by

Broomhead and Lowe, both researchers at the Royal Signals and Radar Establishment.



Architecture of a radial basis function network. An input vector \mathbf{x} is used as input to all radial basis functions, each with different parameters. The output of the network is a linear combination of the outputs from radial basis functions.

RBF networks are typically trained by a

two-step algorithm. In the first step, the center vectors c_i of the RBF functions in the hidden layer are chosen. This step can be performed in several ways; centers can be randomly sampled from some set of examples, or they can be determined using k-means clustering. Note that this step is unsupervised. A third backpropagation step can be performed to fine-tune all of the RBF net's parameters

If the purpose is not to perform strict interpolation but instead more general function approximation or classification the optimization is somewhat more complex because there is no obvious choice for

the centers. The training is typically done in two phases first fixing the width and centers and then the weights. This can be justified by considering the different nature of the non-linear hidden neurons versus the linear output neuron.

5.2 RADIAL BASIS APPROXIMATION

This example uses the NEWRB function to create a radial basis network that approximates a function defined by a set of data points.

Define 21 inputs P and associated targets T.

```
X = -1:.1:1;
```

```
T = [-.9602 -.5770  
-.0729 .3771
```

.6405 .6600 .4609

...

.1336 -.2013

-.4344 -.5000 -.3930

-.1647 .0988 ...

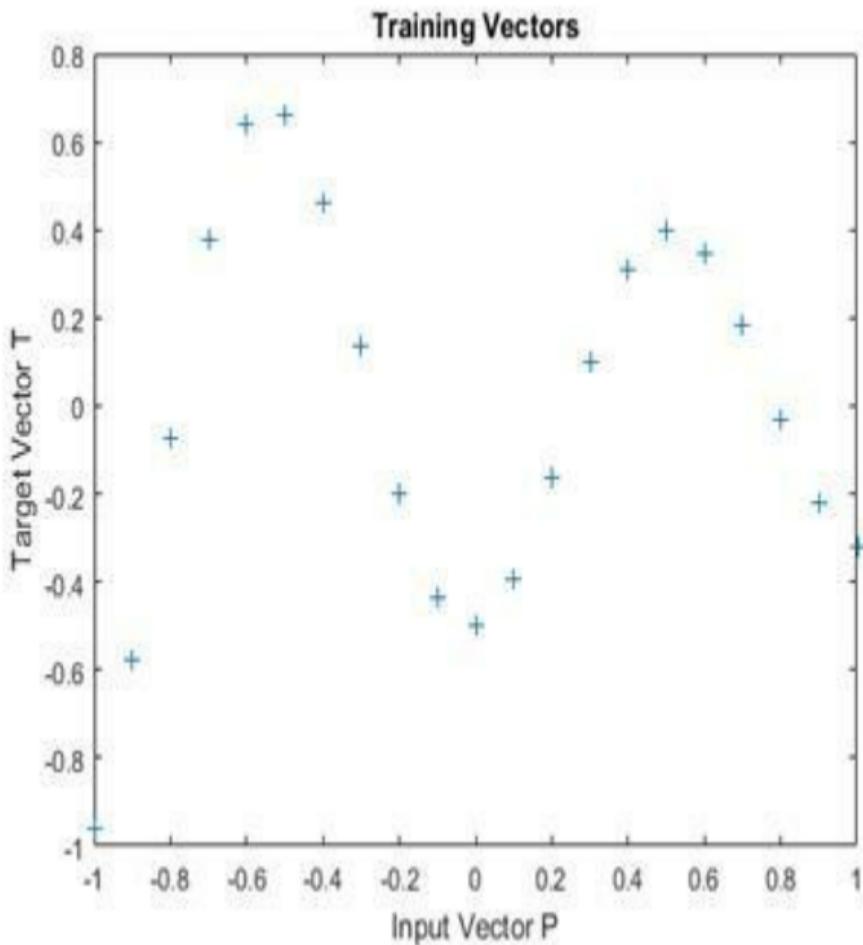
.3072 .3960

.3449 .1816 -.0312

-.2189 -.3201];

plot(X,T,'+') ;

```
title('Training  
Vectors');  
  
xlabel('Input Vector  
P');  
  
ylabel('Target  
Vector T');
```



We would like to find a function which fits the 21 data points. One way to do this is with a radial basis network. A radial basis network is a network with

two layers. A hidden layer of radial basis neurons and an output layer of linear neurons. Here is the radial basis transfer function used by the hidden layer.

```
x = -3:.1:3;
```

```
a = radbas(x);
```

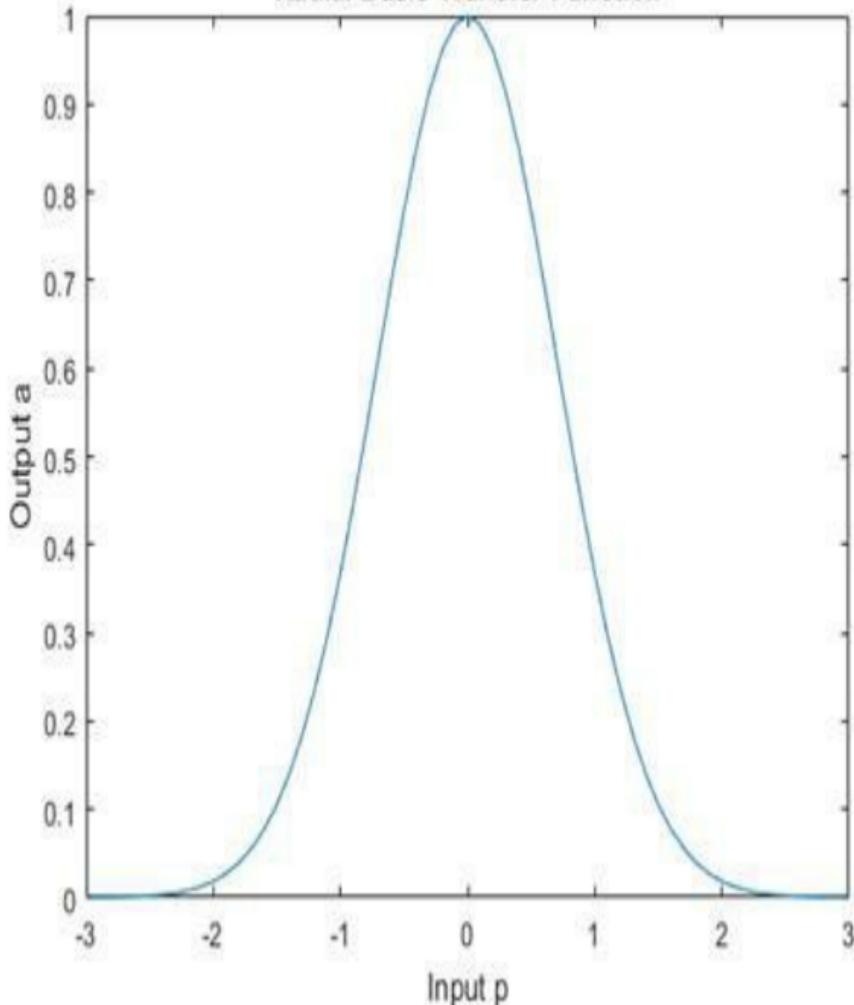
```
plot(x,a)
```

```
title('Radial Basis  
Transfer Function');
```

```
xlabel('Input p');
```

```
ylabel('Output a');
```

Radial Basis Transfer Function



The weights and biases of each neuron in the hidden layer define the position

and width of a radial basis function. Each linear output neuron forms a weighted sum of these radial basis functions. With the correct weight and bias values for each layer, and enough hidden neurons, a radial basis network can fit any function with any desired accuracy. This is an example of three radial basis functions (in blue) are scaled and summed to produce a function (in magenta).

```
a2 = radbas(x-1.5);
```

```
a3 = radbas(x+2);
```

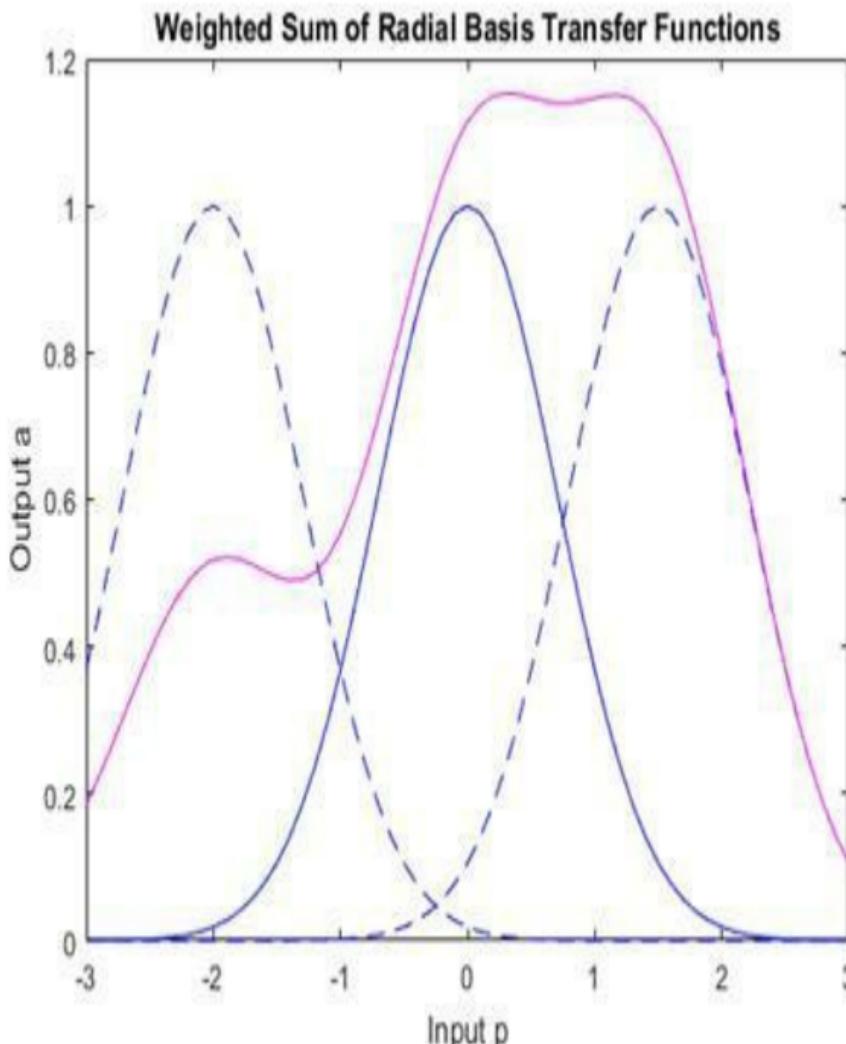
```
a4 = a + a2*1 +  
a3*0.5;
```

```
plot(x,a,'b-'  
' ,x,a2,'b--'  
' ,x,a3,'b--'  
' ,x,a4,'m-' )
```

```
title('Weighted Sum  
of Radial Basis  
Transfer  
Functions') ;
```

```
xlabel('Input p');
```

```
ylabel('Output a');
```



The function NEWRB quickly creates a radial basis network which

approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant.

```
eg = 0.02; % sum-  
squared error goal
```

```
sc = 1; % spread  
constant
```

```
net =  
newrb(X,T,eg,sc);
```

```
NEWRB, neurons = 0,  
MSE = 0.176192
```

To see how the network performs, replot the training set. Then simulate the network response for inputs over the same range. Finally, plot the results on the same graph.

```
plot(X, T, '+') ;
```

```
xlabel('Input') ;
```

```
X = -1:.01:1;
```

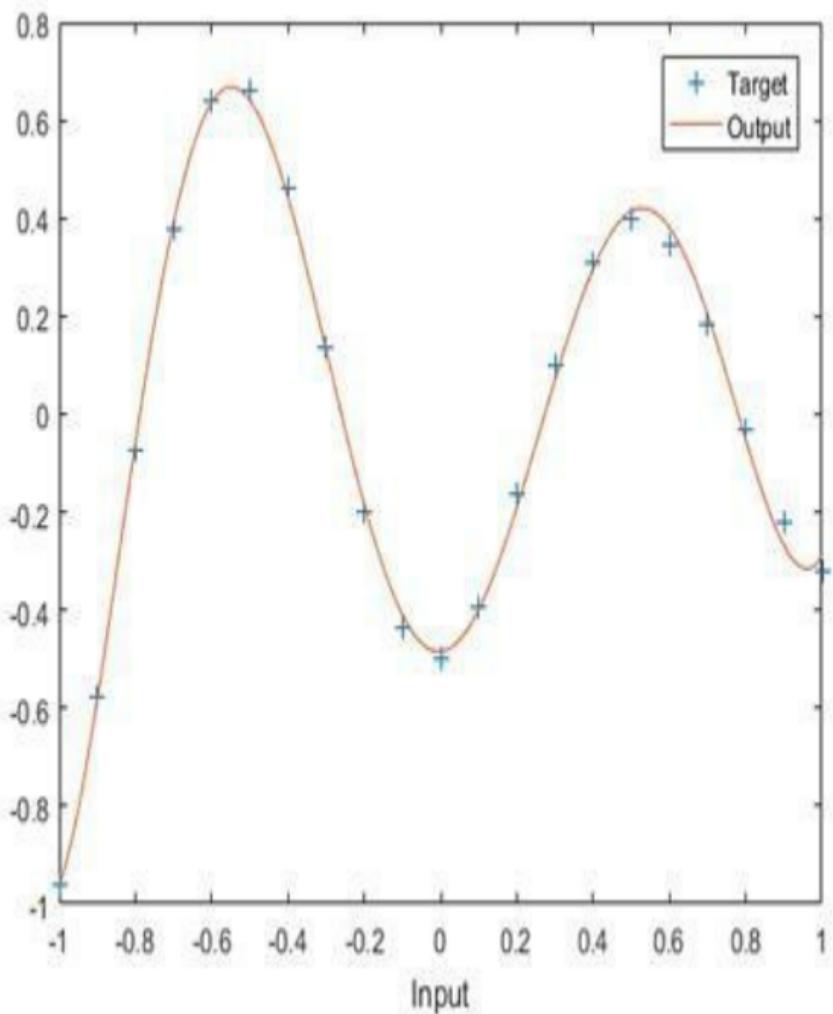
```
Y = net(X);
```

```
hold on;
```

```
plot(X,Y);
```

```
hold off;
```

legend({ 'Target' , 'Out



5.3 RADIAL BASIS UNDERLAPPING NEURONS

A radial basis network is trained to respond to specific inputs with target outputs. However, because the spread of the radial basis neurons is too low, the network requires many neurons.

Define 21 inputs P and associated targets T.

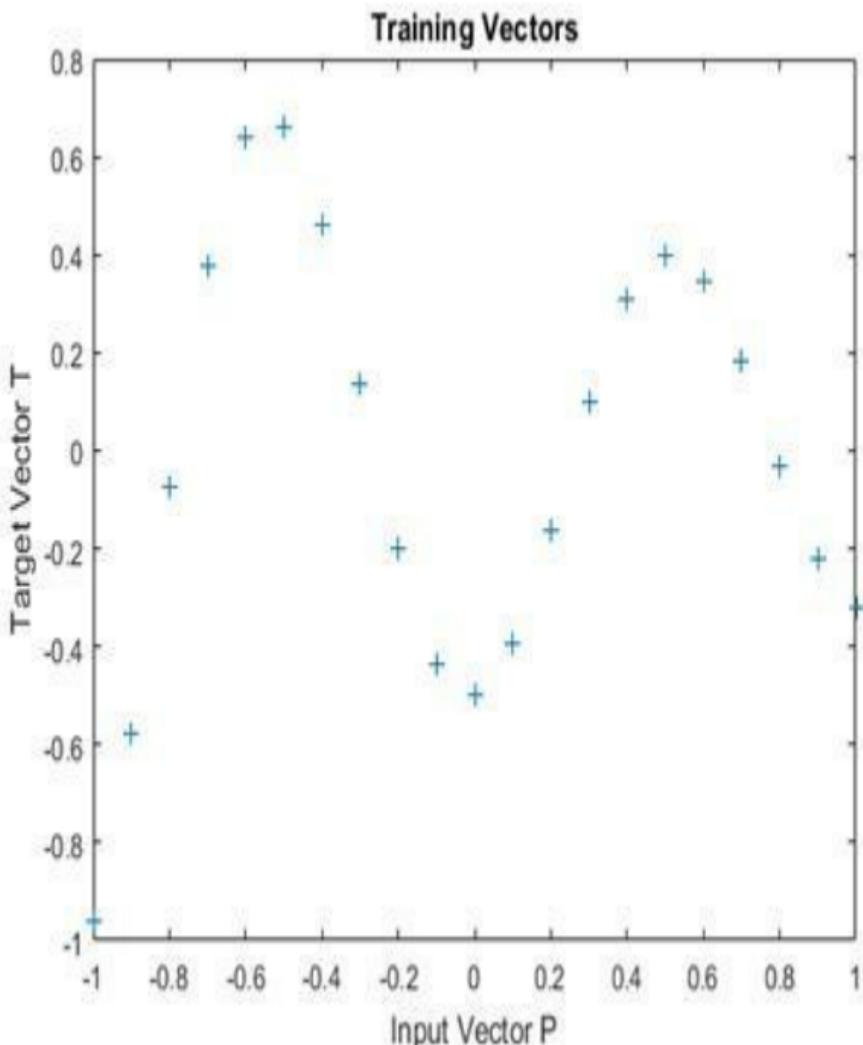
$$P = -1 : .1 : 1 ;$$

T = [-.9602 -.5770
-.0729 .3771
.6405 .6600 .4609
...

 .1336 -.2013
-.4344 -.5000 -.3930
-.1647 .0988 ...

 .3072 .3960
.3449 .1816 -.0312
-.2189 -.3201];

```
plot(P,T,'+') ;  
  
title('Training  
Vectors') ;  
  
xlabel('Input Vector  
P') ;  
  
ylabel('Target  
Vector T') ;
```



The function NEWRB quickly creates a radial basis network which

approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant. The spread of the radial basis neurons B is set to a very small number.

eg = 0.02; % sum-squared error goal

sc = .01; % spread constant

```
net =  
newrb(P,T,eg,sc);
```

NEWRB, neurons = 0,
MSE = 0.176192

To check that the network fits the function in a smooth way, define another set of test input vectors and simulate the network with these new inputs. Plot the results on the same graph as the training set. The test vectors reveal that the function has been overfit! The network could have done better with a higher spread constant.

```
X = -1:.01:1;
```

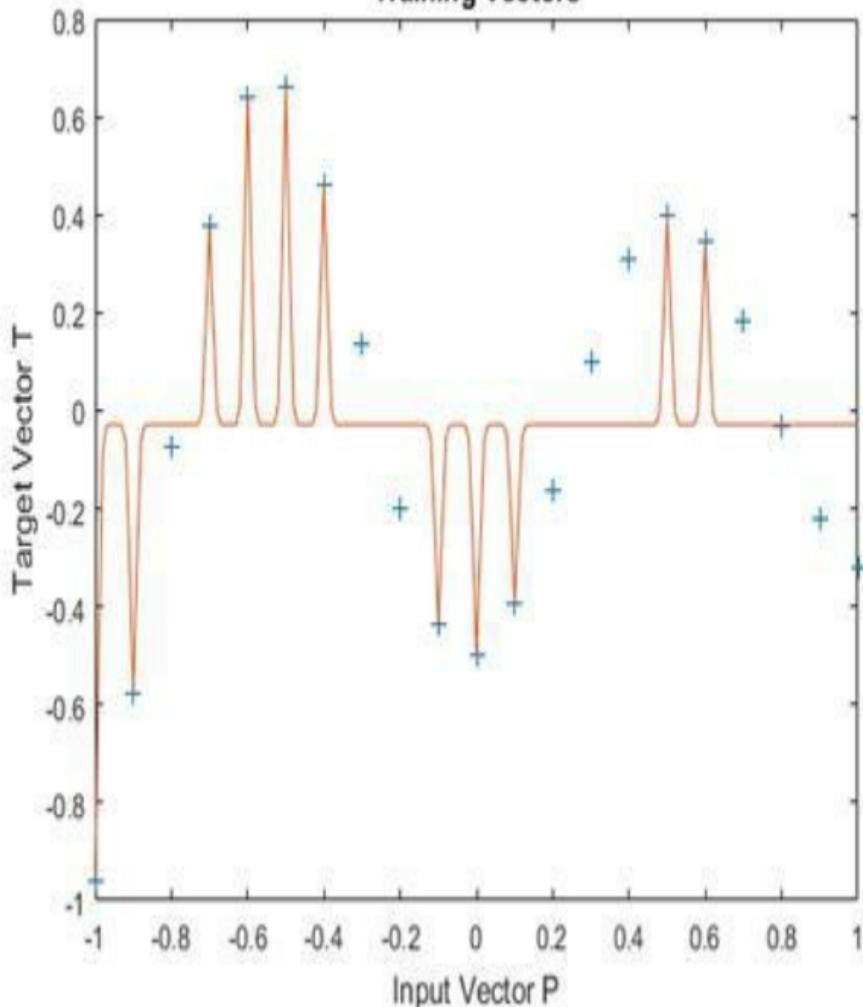
```
Y = net(X);
```

```
hold on;
```

```
plot(X,Y);
```

```
hold off;
```

Training Vectors



5.4 GRNN FUNCTION APPROXIMATION

This example uses functions NEWGRNN and SIM.

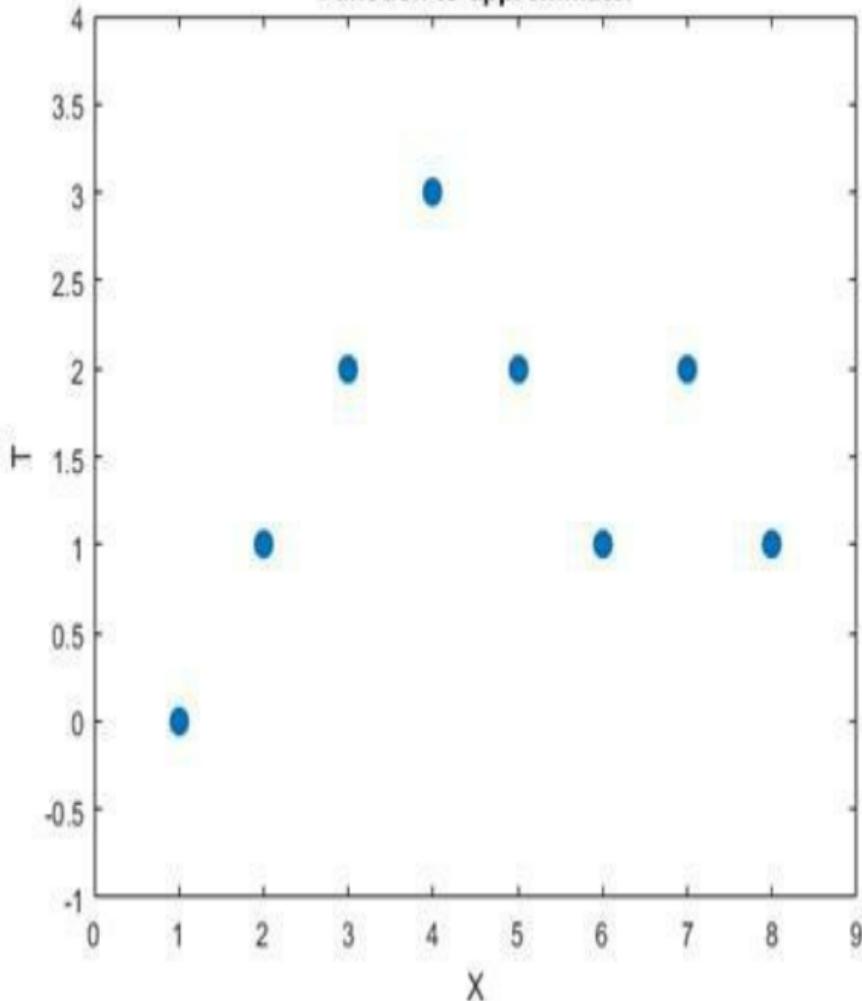
Here are eight data points of y function we would like to fit. The functions inputs X should result in target outputs T.

```
X = [1 2 3 4 5 6 7  
8];
```

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

```
plot(X,T,'.', 'marker');
axis([0 9 -1 4])
title('Function to
approximate.')
xlabel('X')
ylabel('T')
```

Function to approximate.



We use NEWGRNN to create y generalized regression network. We use

y SPREAD slightly lower than 1, the distance between input values, in order, to get y function that fits individual data points fairly closely. A smaller spread would fit data better but be less smooth.

```
spread = 0.7;
```

```
net =  
newgrnn(X, T, spread);
```

```
A = net(X);
```

```
hold on
```

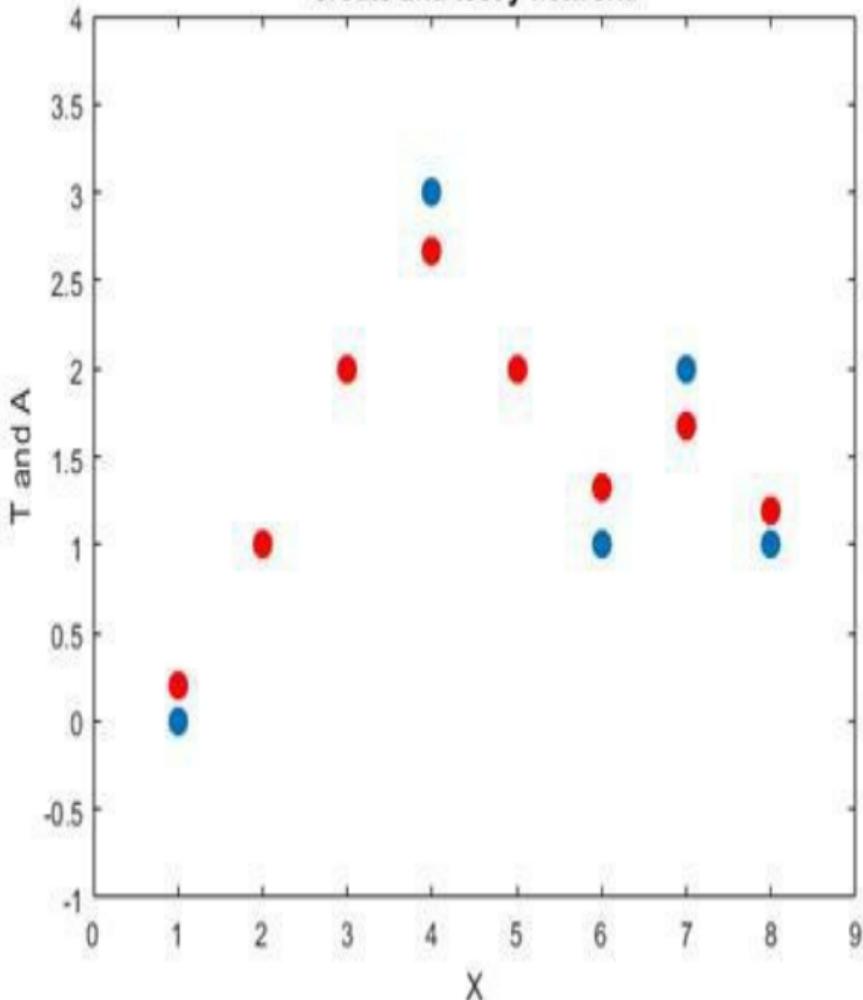
```
outputline =  
plot(X,A,'.', 'marker'  
[1 0 0]);
```

```
title('Create and  
test y network.')
```

```
xlabel('X')
```

```
ylabel('T and A')
```

Create and test y network.



We can use the network to approximate the function at y new input value.

```
x = 3.5;
```

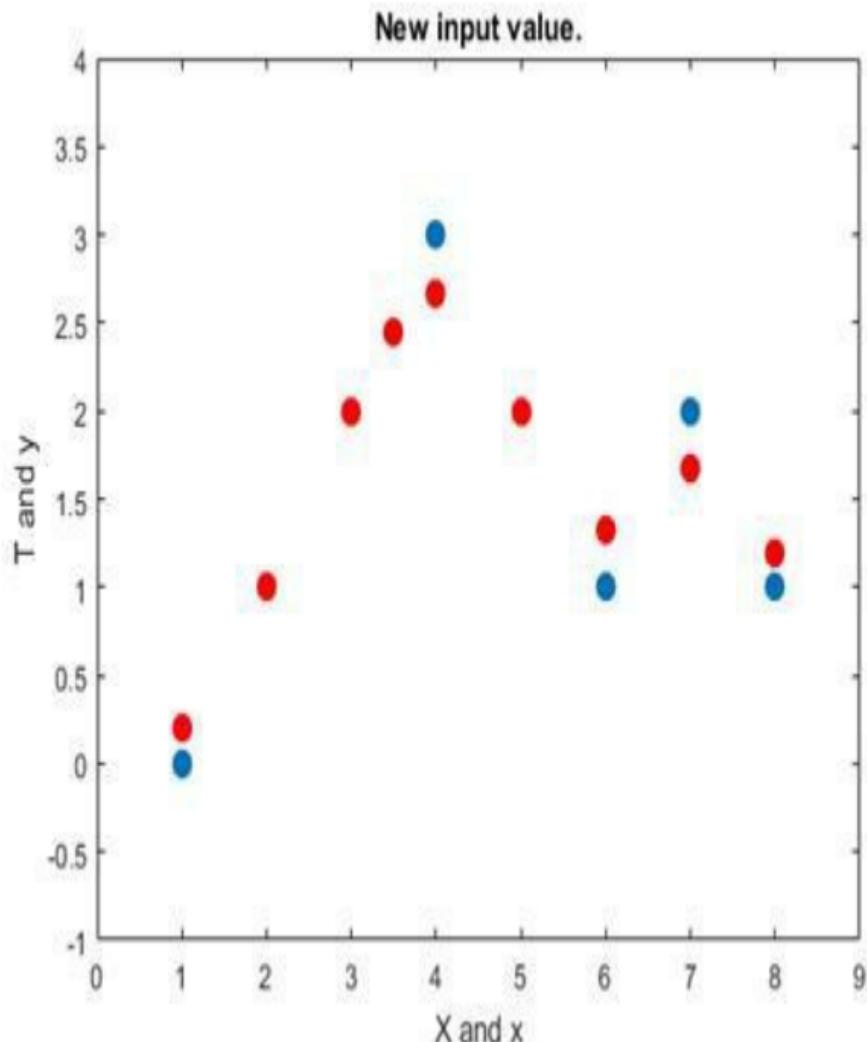
```
y = net(x);
```

```
plot(x,y,'.', 'marker',  
[1 0 0]);
```

```
title('New input  
value.')
```

```
xlabel('X and x')
```

ylabel('T and y')



Here the network's response is simulated

for many values, allowing us to see the function it represents.

```
X2 = 0:.1:9;
```

```
Y2 = net(X2);
```

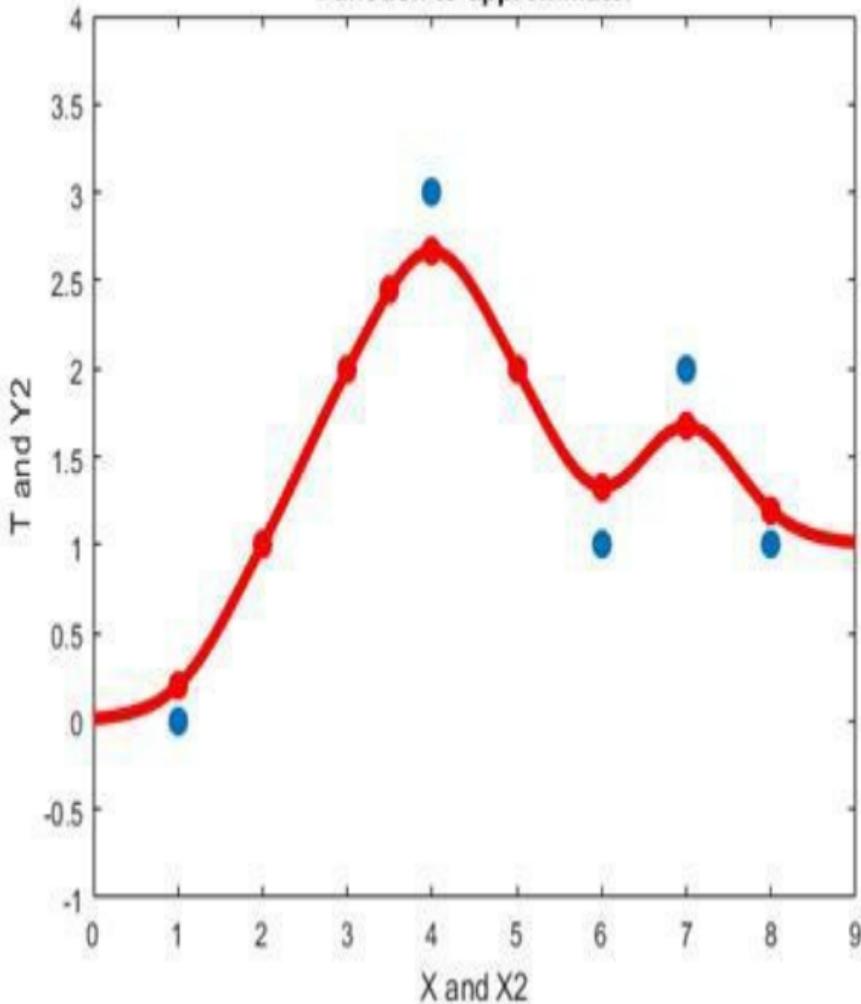
```
plot(X2,Y2,'linewidt:  
[1 0 0])
```

```
title('Function to  
approximate.')
```

```
xlabel('X and X2')
```

```
ylabel('T and Y2')
```

Function to approximate.



5.5 PNN CLASSIFICATION

This example uses functions NEWPNN and SIM.

Here are three two-element input vectors X and their associated classes Tc. We would like to create y probabilistic neural network that classifies these vectors properly.

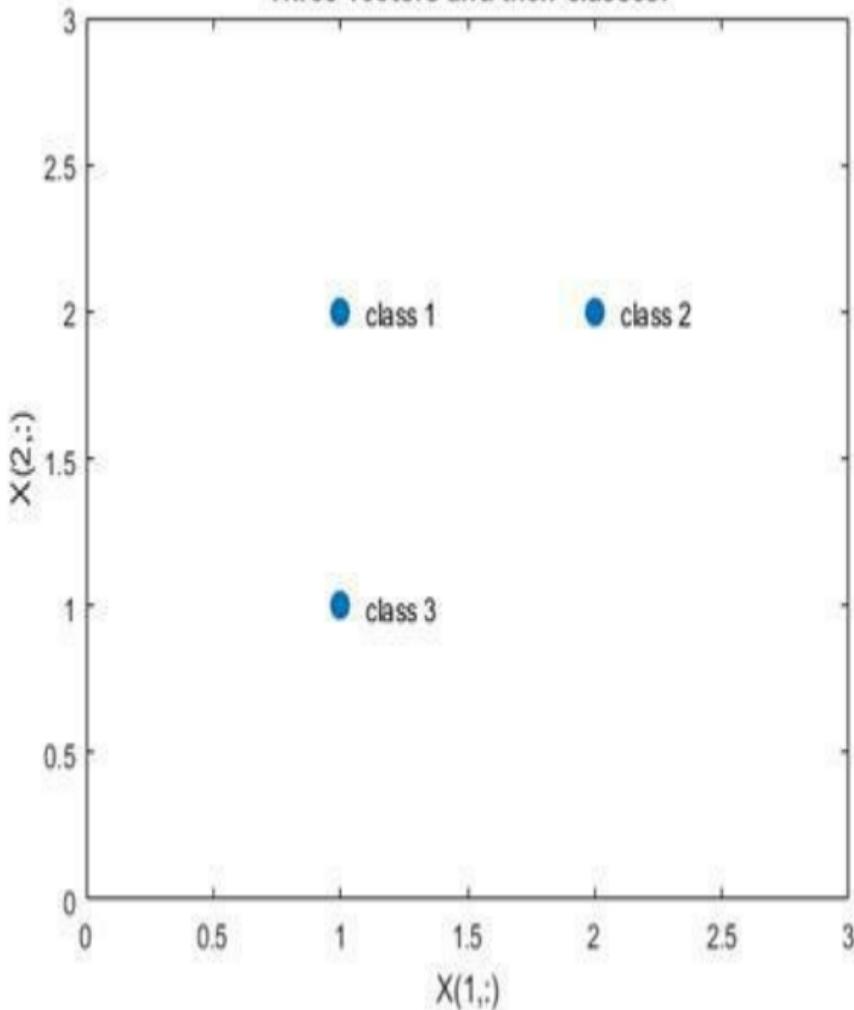
```
X = [1 2; 2 2; 1  
1]';
```

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

```
plot(X(1,:),X(2,:),'  
for i = 1:3,  
text(X(1,i)+0.1,X(2,  
%g',Tc(i))), end  
axis([0 3 0 3])  
title('Three vectors  
and their classes.')  
xlabel('X(1,:)')
```

```
ylabel('X(2,:')
```

Three vectors and their classes.



First we convert the target class indices T_c to vectors T . Then we design a probabilistic neural network with NEWPNN. We use a SPREAD value of 1 because that is a typical distance between the input vectors.

```
T = ind2vec(Tc);
```

```
spread = 1;
```

```
net =
```

```
newpnn(X, T, spread);
```

Now we test the network on the design input vectors. We do this by simulating

the network and converting its vector outputs to indices.

```
Y = net(X);
```

```
YC = vec2ind(Y);
```

```
plot(X(1,:), X(2,:), '^')
```

```
axis([0 3 0 3])
```

```
for i =
```

```
1:3, text(X(1,i)+0.1,
```

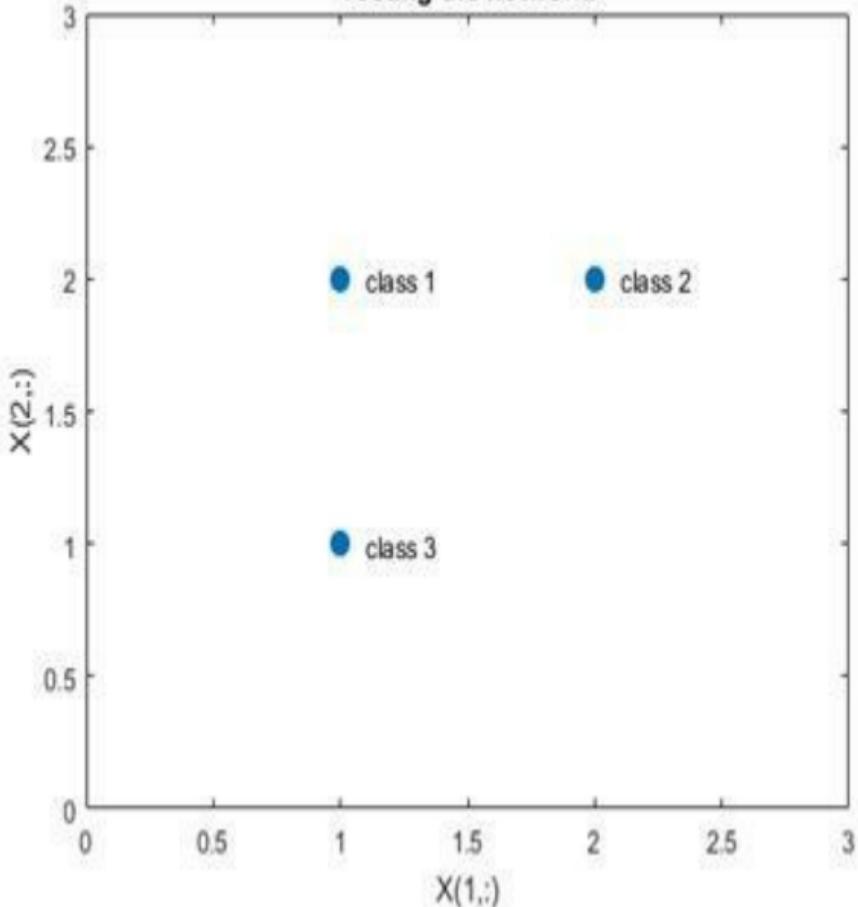
```
%g', Yc(i))), end
```

```
title('Testing the  
network.')
```

```
xlabel('X(1,:') )
```

```
ylabel('X(2,:') )
```

Testing the network.



Let's classify y new vector with our network.

```
x = [2; 1.5];
```

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

```
ac = vec2ind(y);
```

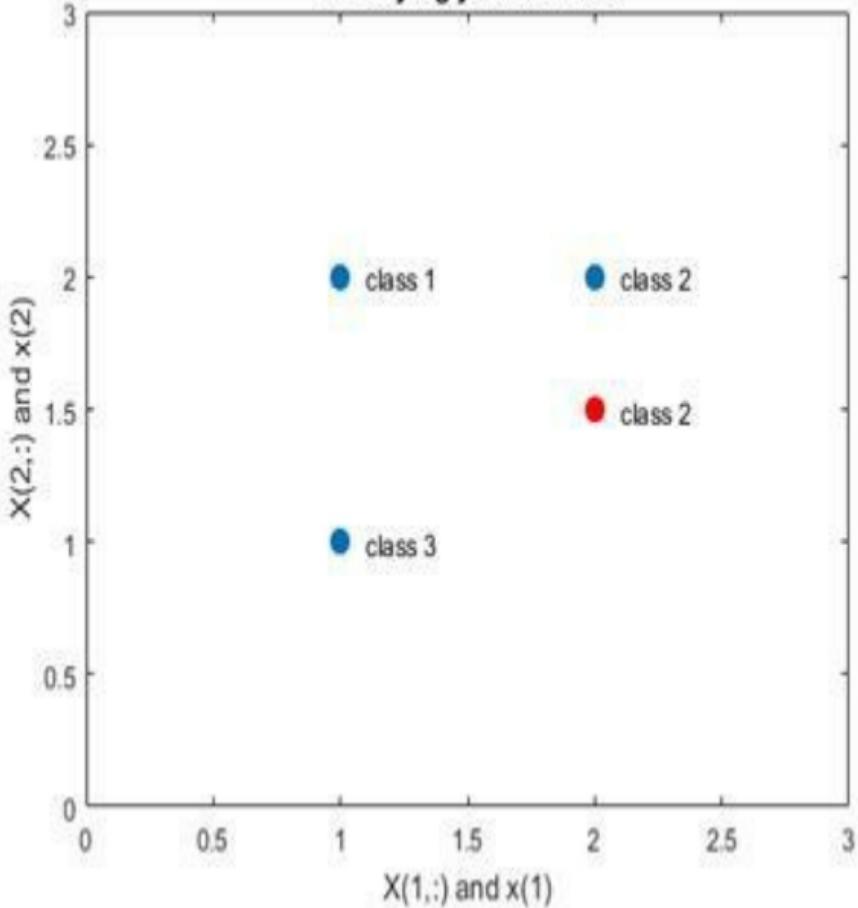
```
hold on
```

```
plot(x(1), x(2), '.*', [1 0 0])
```

```
text(x(1)+0.1, x(2), sprintf('a%d', ac))
```

```
hold off  
  
title('Classifying y  
new vector.')  
  
xlabel('X(1,:) and  
x(1)')  
  
ylabel('X(2,:) and  
x(2)')
```

Classifying y new vector.



This diagram shows how the probabilistic neural network divides the input space into the three classes.

```
x1 = 0:.05:3;
```

```
x2 = x1;
```

```
[X1,X2] =
```

```
meshgrid(x1,x2);
```

```
xx = [X1(:) X2(:)]';
```

```
yy = net(xx);
```

```
yy = full(yy);
```

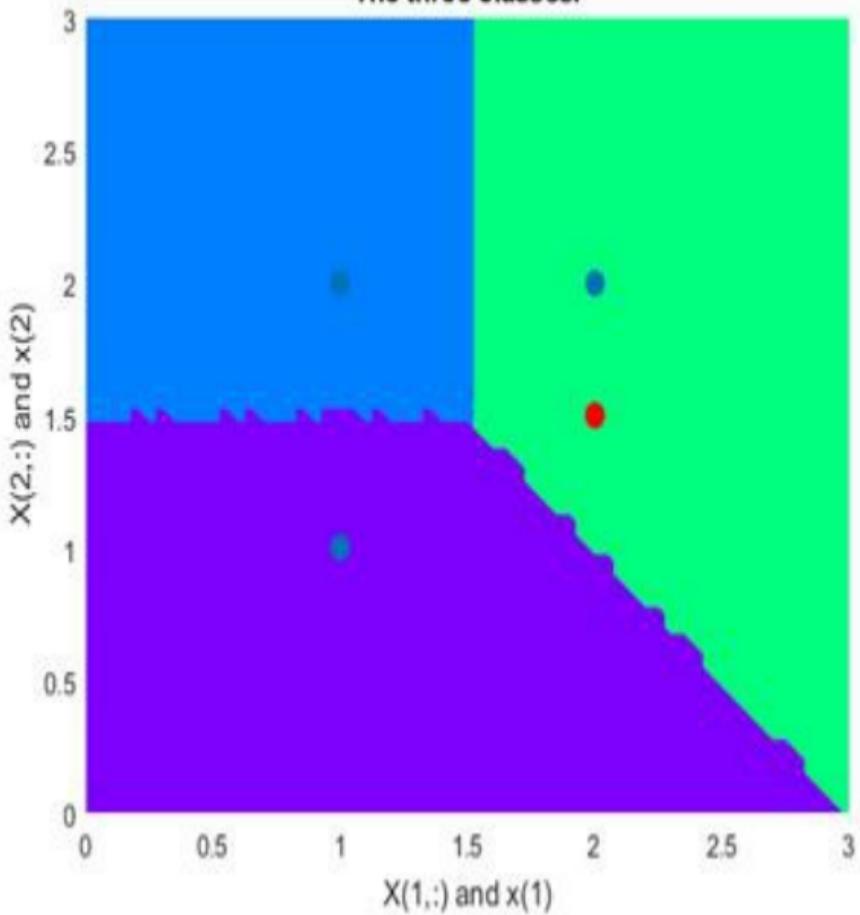
```
m =  
mesh(X1,X2,reshape(y:  
  
m.FaceColor = [0 0.5  
1];  
  
m.LineStyle =  
'none';  
  
hold on  
  
m =
```

```
mesh(X1,X2,reshape(y  
m.FaceColor = [0 1.0  
0.5];  
  
m.LineStyle =  
'none';  
  
m =  
mesh(X1,X2,reshape(y  
m.FaceColor = [0.5 0  
1];
```

```
m.LineStyle =  
'none';  
  
plot3(X(1,:),X(2,:),  
[1 1  
1]+0.1,'.', 'markersi'  
  
plot3(x(1),x(2),1.1,  
[1 0 0])  
  
hold off
```

```
view(2)  
  
title('The three  
classes.')  
  
xlabel('X(1,:) and  
x(1)')  
  
ylabel('X(2,:) and  
x(2)')
```

The three classes.



Chapter 6

SIMPLE APPLICATIONS AND HOLFIELD NETWORKS

6.1 LINEAR PREDICTION DESIGN

This example illustrates how to design a linear neuron to predict the next value in a time series given the last five values.

6.1.1 Defining a Wave Form

Here time is defined from 0 to 5 seconds in steps of 1/40 of a second.

```
time = 0:0.025:5;
```

We can define a signal with respect to time.

```
signal =  
sin(time*4*pi);
```

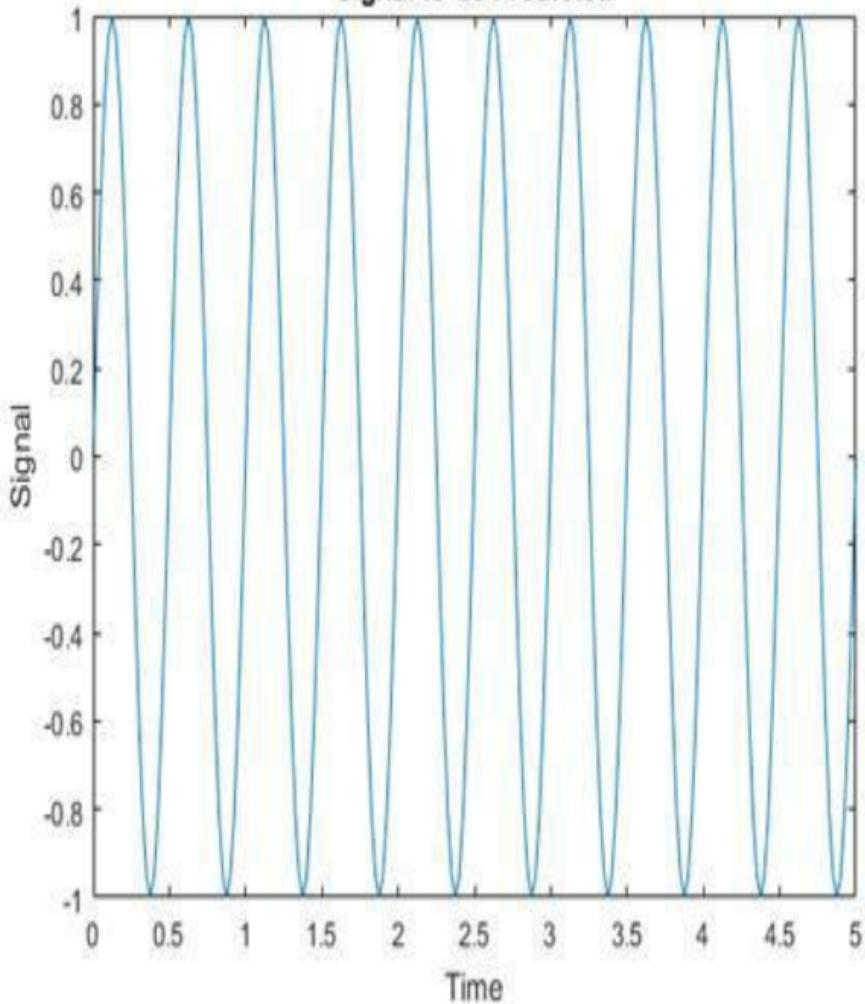
```
plot(time, signal)
```

```
xlabel('Time');
```

```
ylabel('Signal');
```

```
title('Signal to be  
Predicted');
```

Signal to be Predicted



6.1.2 Setting up the Problem for a Neural Network

The signal convert is then converted to a cell array. Neural Networks represent timesteps as columns of a cell array, do distinguish them from different samples at a given time, which are represented with columns of matrices.

```
signal =  
con2seq(signal);
```

To set up the problem we will use the first four values of the signal as initial input delay states, and the rest except for

the last step as inputs.

```
Xi = signal(1:4);
```

```
X = signal(5:(end-1));
```

```
timex = time(5:(end-1));
```

The targets are now defined to match the inputs, but shifted earlier by one timestep.

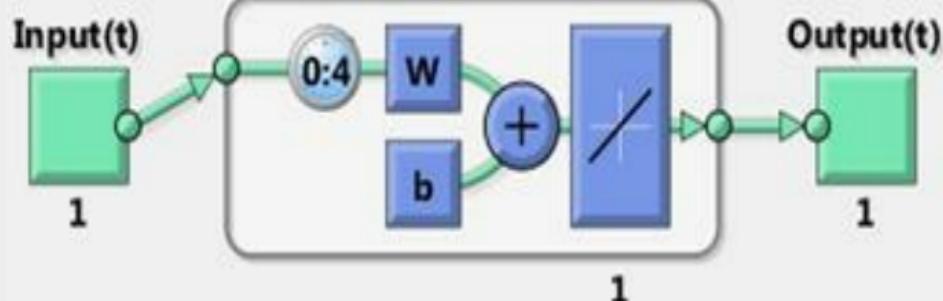
```
T = signal(6:end);
```

6.1.3 Designing the Linear Layer

The function **newlind** will now design a linear layer with a single neuron which predicts the next timestep of the signal given the current and four past values.

```
net =  
newlind(X, T, Xi);  
  
view(net)
```

Layer



6.1.4 Testing the Linear Layer

The network can now be called like a function on the inputs and delayed states to get its time response.

$Y = \text{net}(X, X_i);$

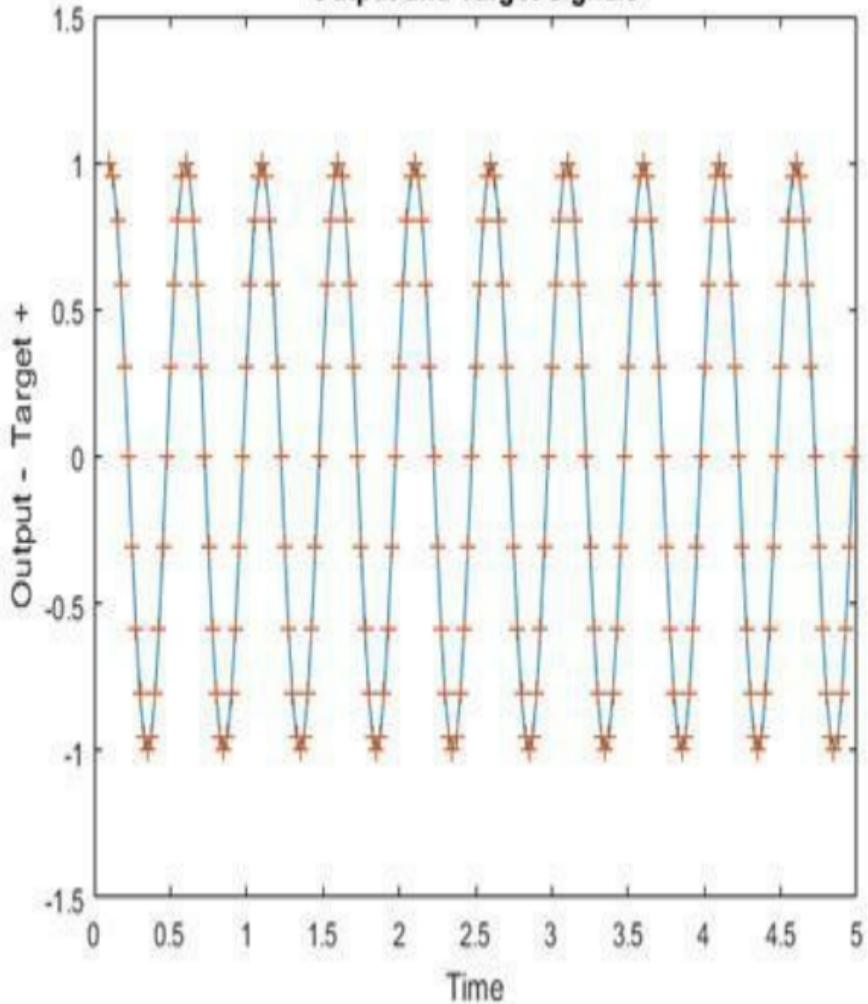
The output signal is plotted with the targets.

figure

`plot(timex, cell2mat(`

```
xlabel('Time');  
  
ylabel('Output -  
Target +');  
  
title('Output and  
Target Signals');
```

Output and Target Signals



The error can also be plotted.

```
figure
```

```
E = cell2mat(T) -  
cell2mat(Y);
```

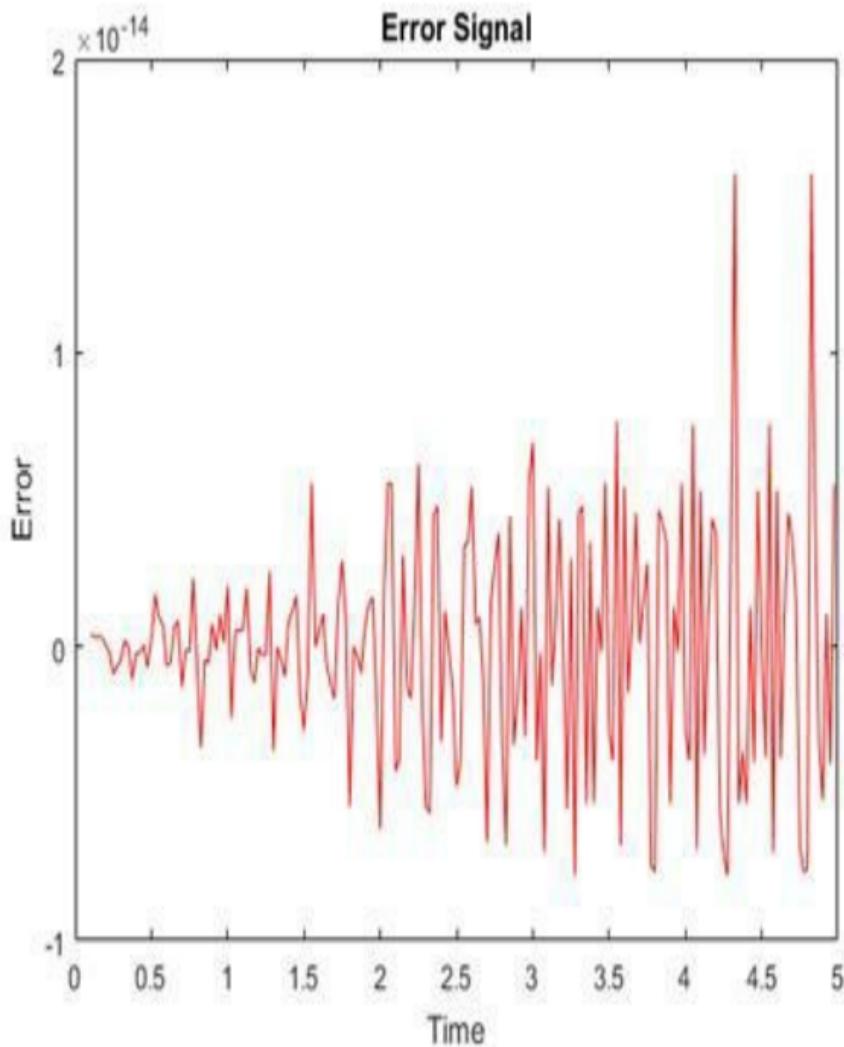
```
plot(timex, E, 'r')
```

```
hold off
```

```
xlabel('Time');
```

```
ylabel('Error');
```

```
title('Error  
Signal');
```



Notice how small the error is!

This example illustrated how to design a

dynamic linear network which can predict a signal's next value from current and past values.

6.2 ADAPTIVE LINEAR PREDICTION

This example illustrates how an adaptive linear layer can learn to predict the next value in a signal, given the current and last four values.

6.2.1 Defining a Wave Form

Here two time segments are defined from 0 to 6 seconds in steps of 1/40 of a second.

```
time1 =  
0:0.025:4; %  
from 0 to 4 seconds
```

```
time2 =  
4.025:0.025:6; %  
from 4 to 6 seconds
```

```
time = [time1  
time2]; % from 0 to  
6 seconds
```

Here is a signal which starts at one frequency but then transitions to another frequency.

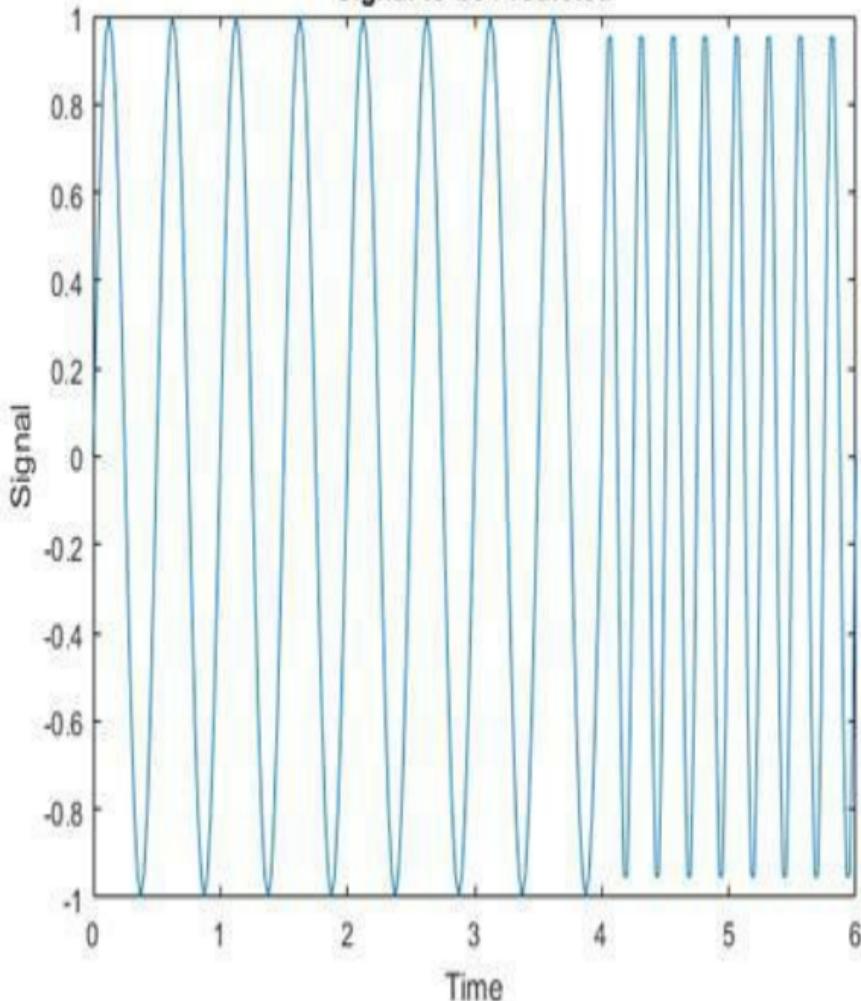
```
signal =  
[sin(time1*4*pi)  
sin(time2*8*pi)];  
  
plot(time, signal)
```

```
xlabel('Time');
```

```
ylabel('Signal');
```

```
title('Signal to be  
Predicted');
```

Signal to be Predicted



6.2.2 Setting up the Problem for a Neural Network

The signal convert is then converted to a cell array. Neural Networks represent timesteps as columns of a cell array, do distinguish them from different samples at a given time, which are represented with columns of matrices.

```
signal =  
con2seq(signal);
```

To set up the problem we will use the first five values of the signal as initial input delay states, and the rest for inputs.

```
Xi = signal(1:5);
```

```
X = signal(6:end);
```

```
timex = time(6:end);
```

The targets are now defined to match the inputs. The network is to predict the current input, only using the last five values.

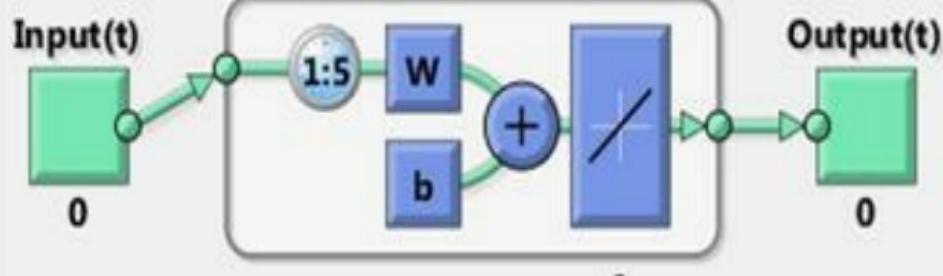
```
T = signal(6:end);
```

6.2.3 Creating the Linear Layer

The function **linearlayer** creates a linear layer with a single neuron with a tap delay of the last five inputs.

```
net =  
linearlayer(1:5, 0.1)  
  
view(net)
```

Linear



6.2.4 Adapting the Linear Layer

The function `*adapt*` simulates the network on the input, while

adjusting its weights and biases after each timestep

in response
to how closely its
output matches the
target.

It returns the
update networks, its
outputs, and its
errors.

```
[net, Y] =  
adapt(net, X, T, Xi);
```

The output signal is plotted with the targets.

```
figure
```

```
plot(timex,cell2mat(
```

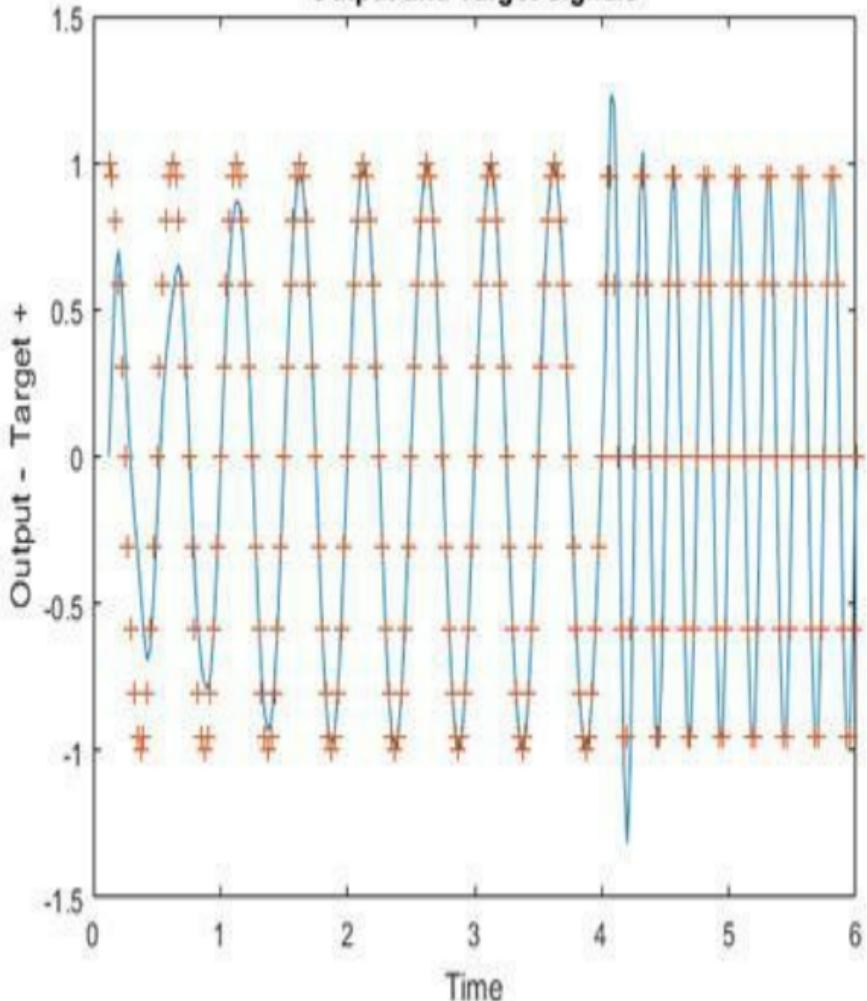
```
xlabel('Time');
```

```
ylabel('Output -  
Target +');
```

```
title('Output and
```

Target Signals');

Output and Target Signals



The error can also be plotted.

```
figure
```

```
E = cell2mat(T) -  
cell2mat(Y);
```

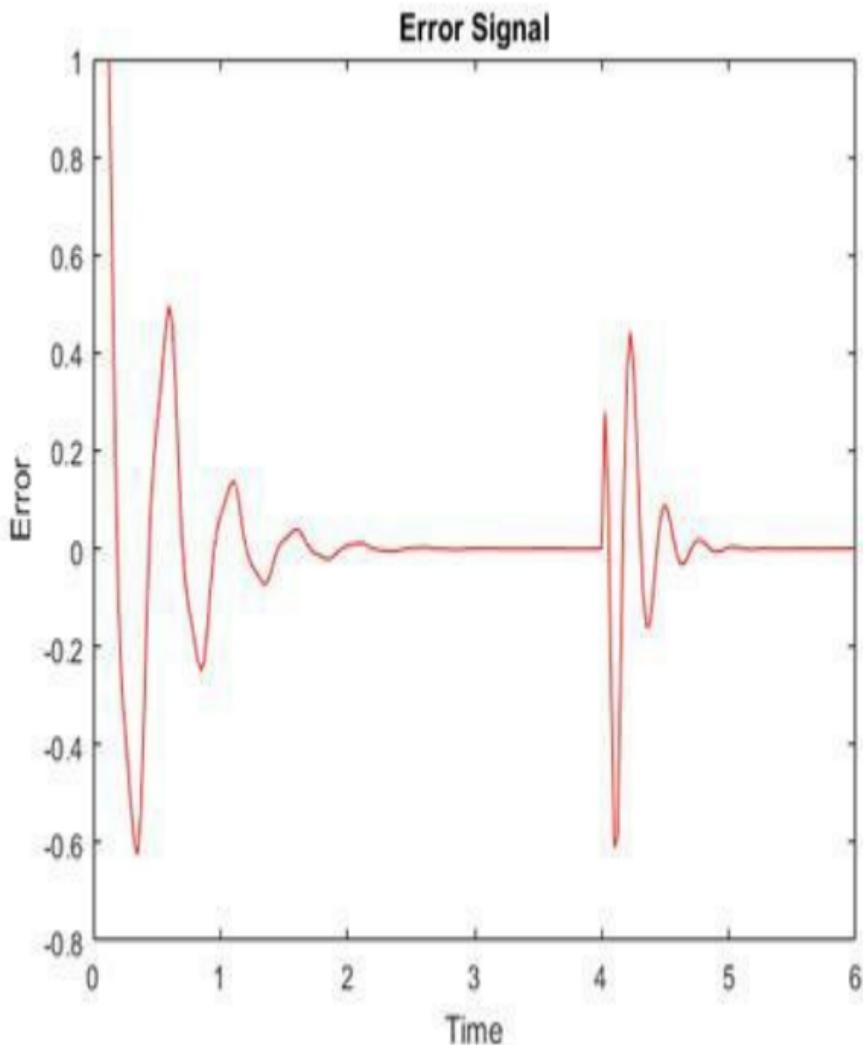
```
plot(timex, E, 'r')
```

```
hold off
```

```
xlabel('Time');
```

```
ylabel('Error');
```

```
title('Error  
Signal');
```



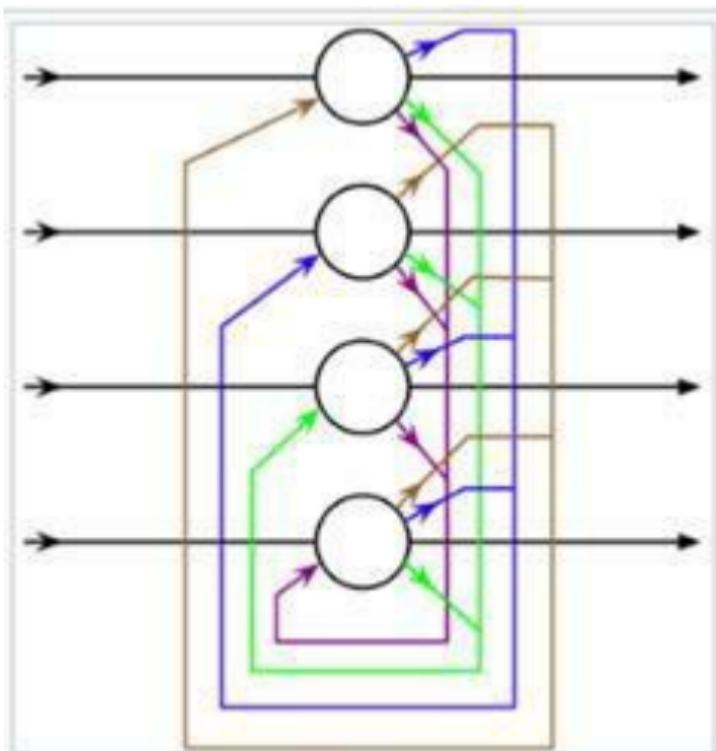
Notice how small the error is except for initial errors and the network learns the

systems behavior at the beginning and after the system transition.

This example illustrated how to simulate an adaptive linear network which can predict a signal's next value from current and past values despite changes in the signals behavior.

6.3 HOPFIELD NETWORK

A Hopfield network is a form of recurrent artificial neural network popularized by John Hopfield in 1982, but described earlier by Little in 1974. Hopfield nets serve as content-addressable memory systems with binary threshold nodes. They are guaranteed to converge to a local minimum, but will sometimes converge to a false pattern (wrong local minimum) rather than the stored pattern (expected local minimum). Hopfield networks also provide a model for understanding human memory.



A Hopfield net with four nodes.



Initialization of the Hopfield Networks is done by setting the values of the units to the desired start pattern. Repeated updates are then performed until the

network converges to an attractor pattern. Convergence is generally assured, as Hopfield proved that the attractors of this nonlinear dynamical system are stable, not periodic or chaotic as in some other systems. Therefore, in the context of Hopfield Networks, an attractor pattern is a final stable state, a pattern that cannot change any value within it under updating.

Training a Hopfield net involves lowering the energy of states that the net should "remember". This allows the net to serve as a content addressable memory system, that is to say, the network will converge to a "remembered" state if it is given only

part of the state. The net can be used to recover from a distorted input to the trained state that is most similar to that input. This is called associative memory because it recovers memories on the basis of similarity. For example, if we train a Hopfield net with five units so that the state $(1, 0, 1, 0, 1)$ is an energy minimum, and we give the network the state $(1, 0, 0, 0, 1)$ it will converge to $(1, 0, 1, 0, 1)$. Thus, the network is properly trained when the energy of states which the network should remember are local minima.

There are various different learning rules that can be used to store information in the memory of the

Hopfield Network. It is desirable for a learning rule to have both of the following two properties:

- *Local*: A learning rule is *local* if each weight is updated using information available to neurons on either side of the connection that is associated with that particular weight.
- *Incremental*: New patterns can be learned without using information from the old patterns that have been also used for training. That is, when a new pattern is used for training, the new values for the weights only depend on the old values and on

the new pattern.

These properties are desirable, since a learning rule satisfying them is more biologically plausible. For example, since the human brain is always learning new concepts, one can reason that human learning is incremental. A learning system that were not incremental would generally be trained only once, with a huge batch of training data.

The Network capacity of the Hopfield network model is determined by neuron amounts and connections within a given network. Therefore, the number of memories that are able to be stored is dependent on neurons and connections. Furthermore, it was shown that the recall

accuracy between vectors and nodes was 0.138 (approximately 138 vectors can be recalled from storage for every 1000 nodes) (Hertz et al., 1991). Therefore, it is evident that many mistakes will occur if one tries to store a large number of vectors. When the Hopfield model does not recall the right pattern, it is possible that an intrusion has taken place, since semantically related items tend to confuse the individual, and recollection of the wrong pattern occurs. Therefore, the Hopfield network model is shown to confuse one stored item with that of another upon retrieval. Perfect recalls and high capacity, >0.14 , can be loaded in the

network by Hebbian learning method.

The Hopfield model accounts for associative memory through the incorporation of memory vectors.

Memory vectors can be slightly used, and this would spark the retrieval of the most similar vector in the network.

However, we will find out that due to this process, intrusions can occur. In associative memory for the Hopfield network, there are two types of operations: auto-association and hetero-association. The first being when a vector is associated with itself, and the latter being when two different vectors are associated in storage. Furthermore, both types of operations are possible to

store within a single memory matrix, but only if that given representation matrix is not one or the other of the operations, but rather the combination (auto-associative and hetero-associative) of the two. It is important to note that Hopfield's network model utilizes the same learning rule as [Hebb's \(1949\) learning rule](#), which basically tried to show that learning occurs as a result of the strengthening of the weights by when activity is occurring.

Rizzuto and Kahana (2001) were able to show that the neural network model can account for repetition on recall accuracy by incorporating a probabilistic-learning algorithm. During the retrieval process,

no learning occurs. As a result, the weights of the network remain fixed, showing that the model is able to switch from a learning stage to a recall stage. By adding contextual drift we are able to show the rapid forgetting that occurs in a Hopfield model during a cued-recall task. The entire network contributes to the change in the activation of any single node.

McCullough and Pitts' (1943) dynamical rule, which describes the behavior of neurons, does so in a way that shows how the activations of multiple neurons map onto the activation of a new neuron's firing rate, and how the weights of the neurons strengthen the synaptic

connections between the new activated neuron (and those that activated it). Hopfield would use McCullough-Pitts's dynamical rule in order to show how retrieval is possible in the Hopfield network. However, it is important to note that Hopfield would do so in a repetitious fashion. Hopfield would use a nonlinear activation function, instead of using a linear function. This would therefore create the Hopfield dynamical rule and with this, Hopfield was able to show that with the nonlinear activation function, the dynamical rule will always modify the values of the state vector in the direction of one of the stored patterns

6.4 HOPFIELD TWO NEURON DESIGN

A Hopfield network consisting of two neurons is designed with two stable equilibrium points and simulated using the above functions.

We would like to obtain a Hopfield network that has the two stable points defined by the two target (column) vectors in T.

$$T = [+1 \ -1 ; \ \dots]$$

$$\quad \quad \quad -1 \ +1] ;$$

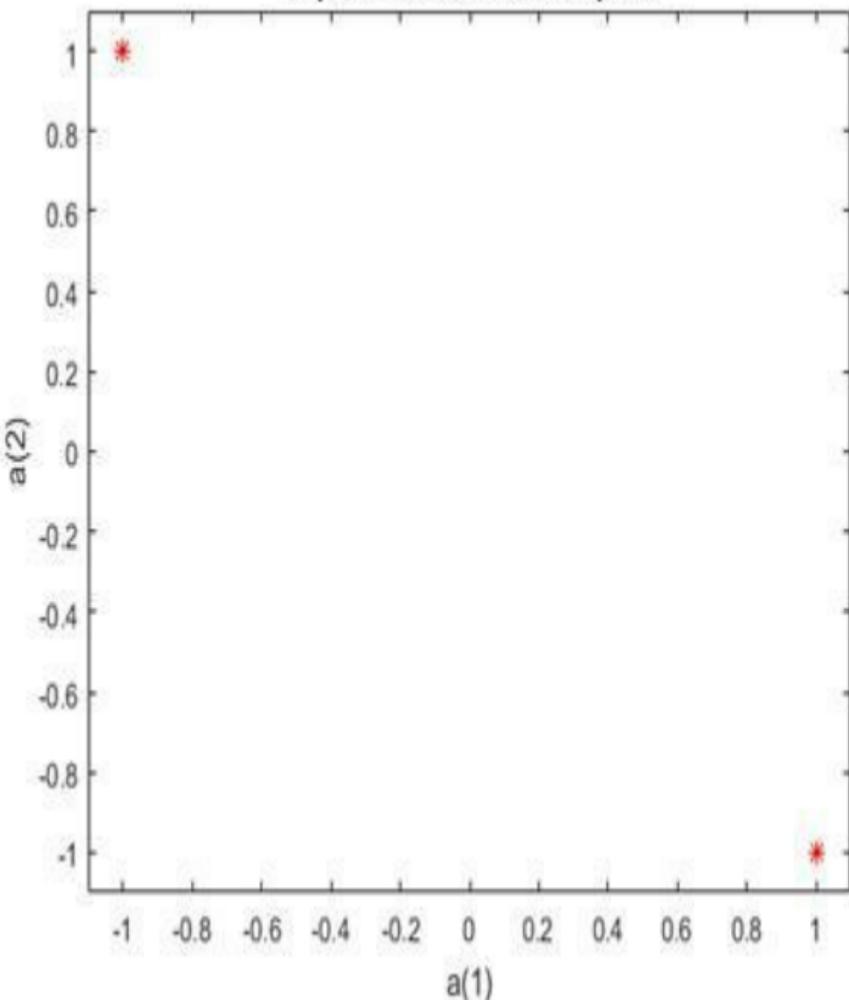
Here is a plot where the stable points are shown at the corners. All possible states of the 2-neuron Hopfield network are contained within the plots boundaries.

```
plot(T(1, :), T(2, :), '  
axis([-1.1 1.1 -1.1  
1.1])  
title('Hopfield  
Network State  
Space')
```

```
xlabel('a(1)');
```

```
ylabel('a(2)');
```

Hopfield Network State Space



The function NEWHOP creates Hopfield networks given the stable points T.

```
net = newhop(T);
```

First we check that the target vectors are indeed stable. We check this by giving the target vectors to the Hopfield network. It should return the two targets unchanged, and indeed it does.

```
[Y, Pf, Af] = net([],[],T);
```

Y

Y =

1 -1

-1 1

Here we define a random starting point and simulate the Hopfield network for 20 steps. It should reach one of its stable points.

a = {rands(2,1)};

```
[y, Pf, Af] =  
net({20}, {}, a);
```

We can make a plot of the Hopfield networks activity.

Sure enough, the network ends up in either the upper-left or lower right corners of the plot.

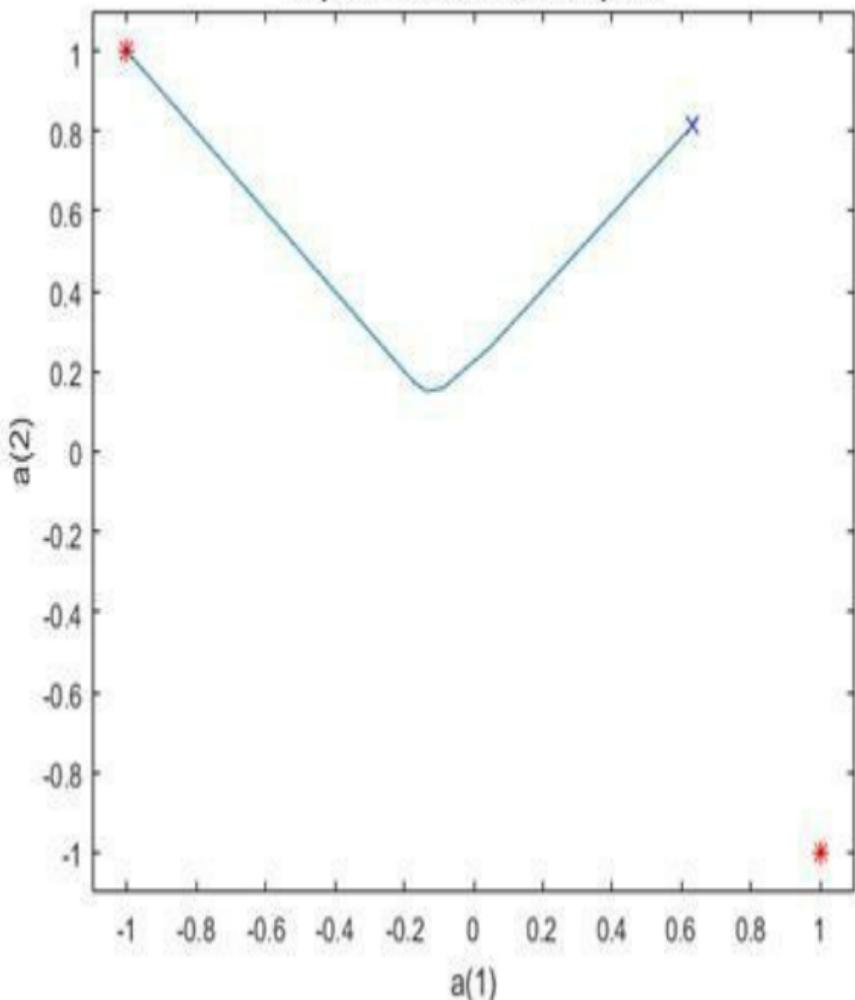
```
record =  
[cell2mat(a)  
cell2mat(y)];
```

```
start = cell2mat(a);
```

hold on

plot(start(1,1), star

Hopfield Network State Space



We repeat the simulation for 25 more initial conditions.

Note that if the Hopfield network starts out closer to the upper-left, it will go to the upper-left, and vice versa. This ability to find the closest memory to an initial input is what makes the Hopfield network useful.

```
color = 'rgbmy';  
for i=1:25  
    a = {rands(2,1)};  
  
    [y,Pf,Af] =
```

```
net( {20} , { } , a) ;
```

```
record=
```

```
[cell2mat(a)
```

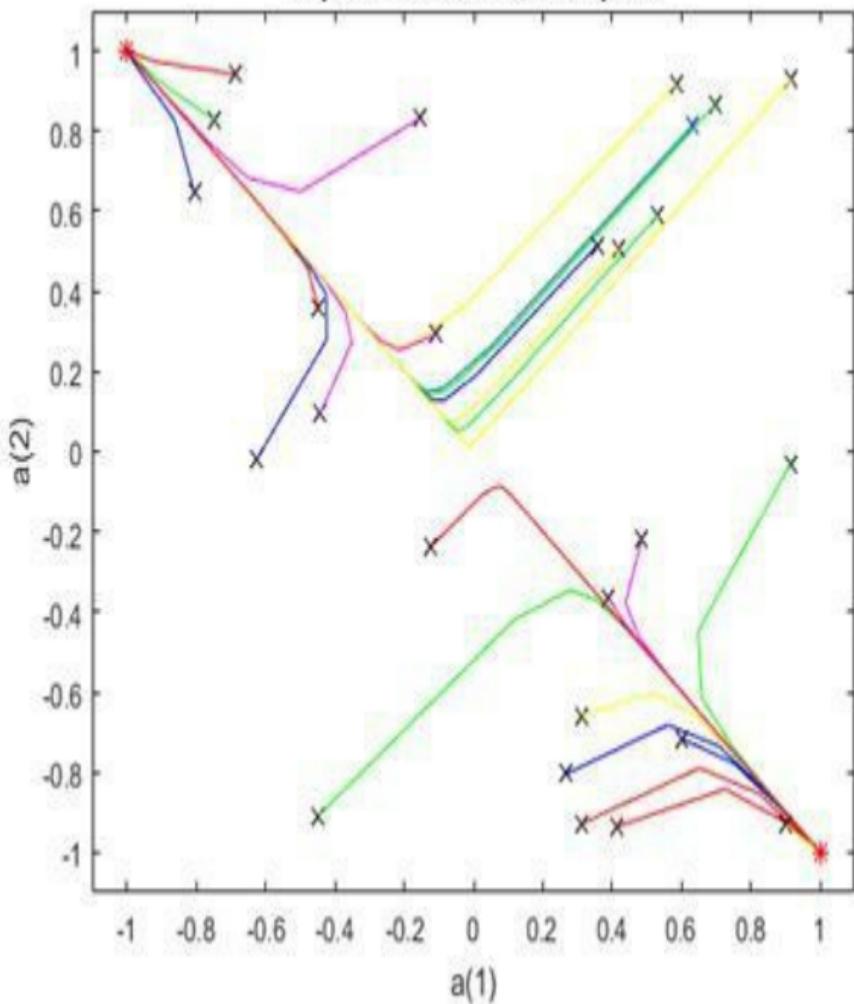
```
cell2mat(y) ] ;
```

```
start=cell2mat(a) ;
```

```
plot(start(1,1),star
```

```
end
```

Hopfield Network State Space



6.5 HOPFIELD UNSTABLE EQUILIBRIA

A Hopfield network is designed with target stable points. However, while NEWHOP finds a solution with the minimum number of unspecified stable points, they do often occur. The Hopfield network designed here is shown to have an undesired equilibrium point. However, these points are unstable in that any noise in the system will move the network out of them.

We would like to obtain a Hopfield network that has the two stable points define by the two target (column)

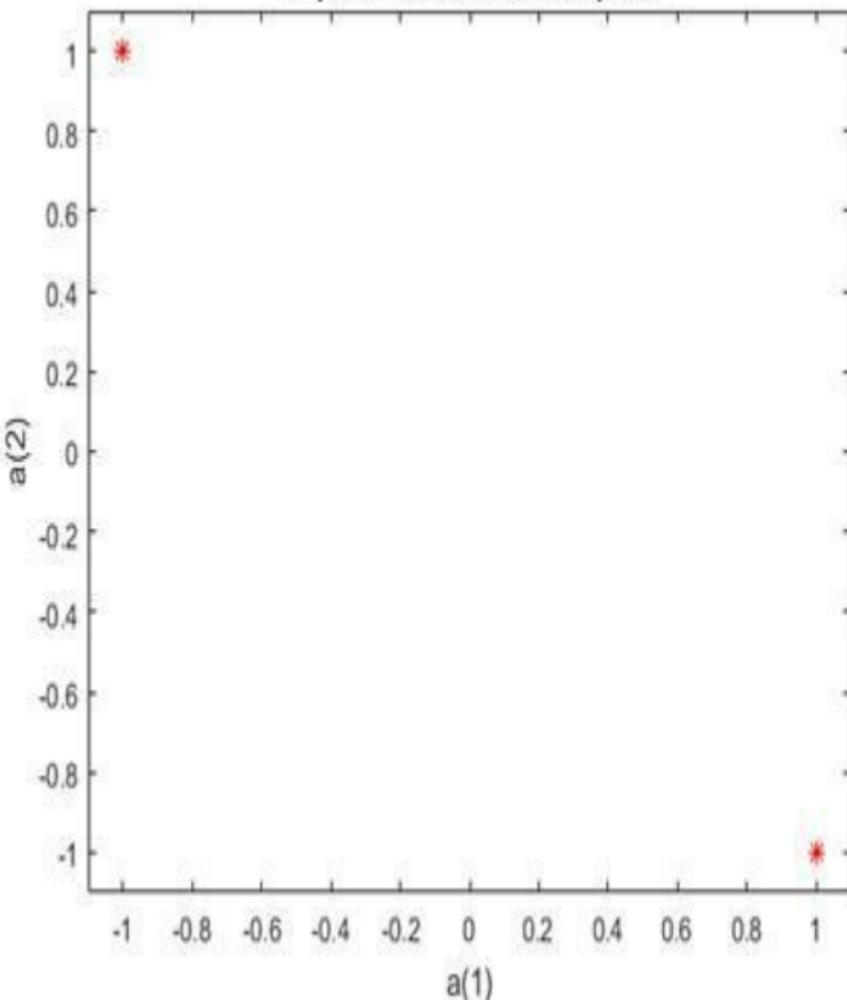
vectors in T.

```
T = [+1 -1; ...  
      -1 +1];
```

Here is a plot where the stable points are shown at the corners. All possible states of the 2-neuron Hopfield network are contained within the plots boundaries.

```
plot(T(1,:),T(2,:),'r*')  
axis([-1.1 1.1 -1.1 1.1])  
title('Hopfield Network State Space')  
xlabel('a(1)');  
ylabel('a(2)');
```

Hopfield Network State Space



The function NEWHOP creates Hopfield networks given the stable points T.

```
net = newhop(T);
```

Here we define a random starting point and simulate the Hopfield network for 50 steps. It should reach one of its stable points.

```
a = {rands(2,1)};
```

```
[y,Pf,Af] = net({1 50},[],a);
```

We can make a plot of the Hopfield networks activity.

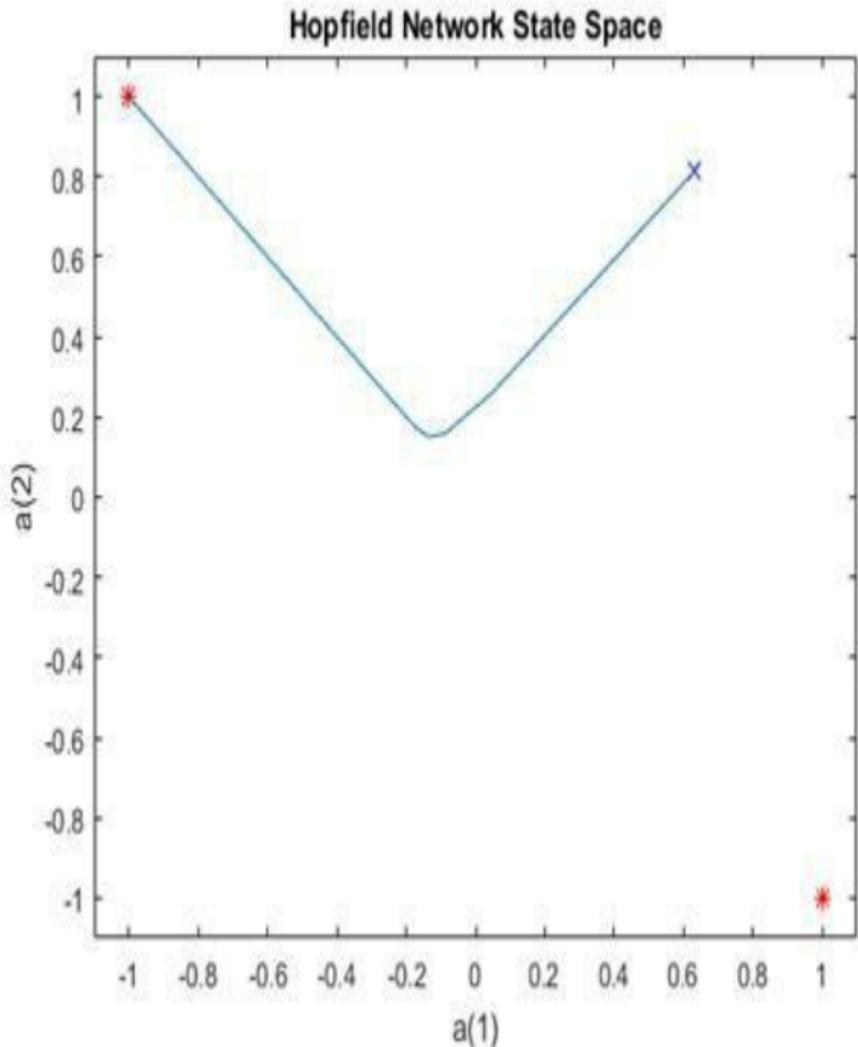
Sure enough, the network ends up in either the upper-left or lower right corners of the plot.

```
record = [cell2mat(a) cell2mat(y)];
```

```
start = cell2mat(a);
```

```
hold on
```

```
plot(start(1,1),start(2,1),'bx',record(1,:),r
```



Unfortunately, the network has undesired

stable points at places other than the corners. We can see this when we simulate the Hopfield for the five initial weights, P.

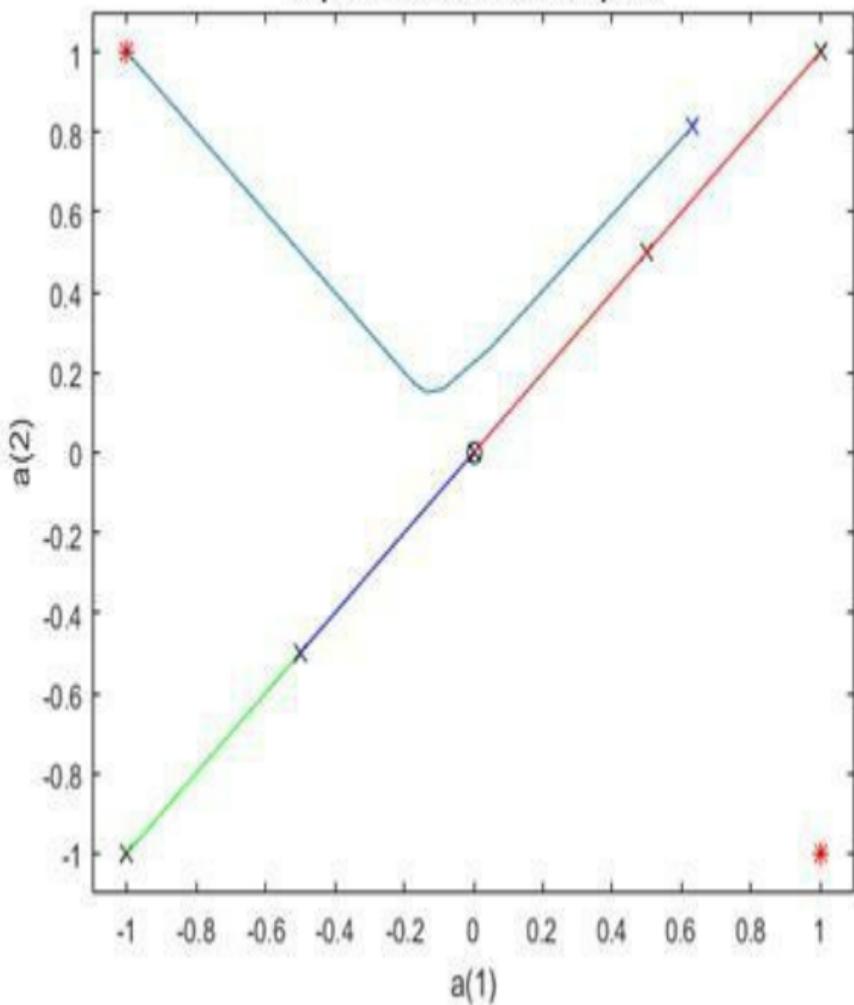
These points are exactly between the two target stable points. The result is that they all move into the center of the state space, where an undesired stable point exists.

```
plot(0,0,'ko');
P = [-1.0 -0.5 0.0 +0.5 +1.0;
      -1.0 -0.5 0.0 +0.5 +1.0];
color = 'rgbmy';
for i=1:5
    a = {P(:,i)};
    [y,Pf,Af] = net({1 50}, {}, a);
    record=[cell2mat(a) cell2mat(y)];
```

```
start = cell2mat(a);
```

```
plot(start(1,1),start(2,1),'kx',record(1,:),r  
drawnow  
end
```

Hopfield Network State Space



6.6 HOPFIELD THREE NEURON DESIGN

A Hopfield network is designed with target stable points. The behavior of the Hopfield network for different initial conditions is studied.

We would like to obtain a Hopfield network that has the two stable points defined by the two target (column) vectors in T.

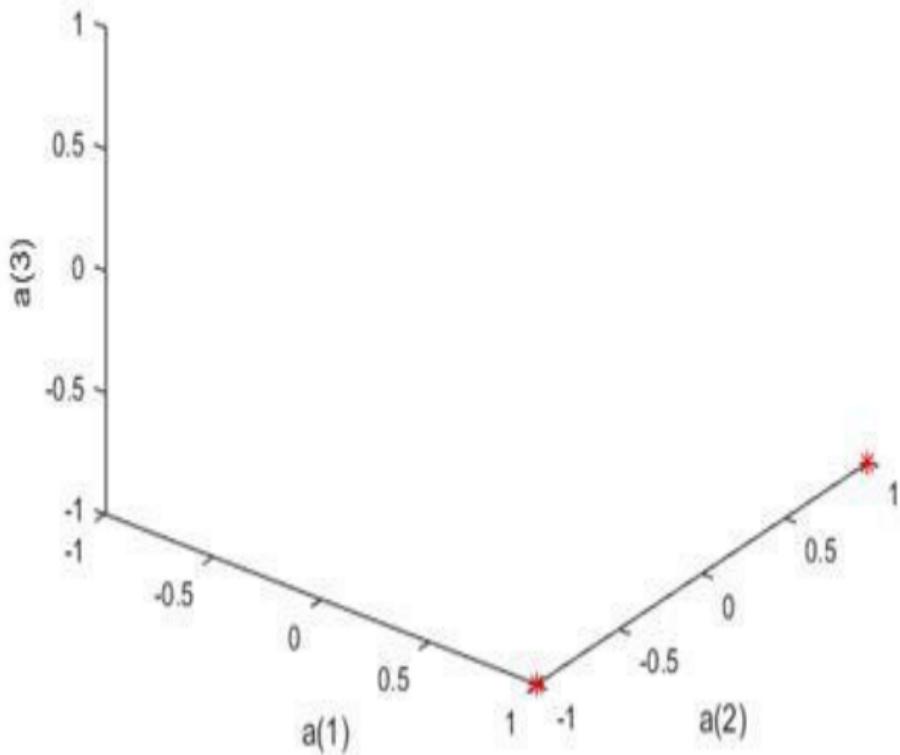
$$T = [+1 \ +1; \dots \\ -1 \ +1; \dots \\ -1 \ -1];$$

Here is a plot where the stable points are shown at the corners. All possible

states of the 2-neuron Hopfield network
are contained within the plots
boundaries.

```
axis([-1 1 -1 1 -1 1])
gca.box = 'on';
axis manual;
hold on;
plot3(T(1,:),T(2,:),T(3,:),'r*')
title('Hopfield Network State Space')
xlabel('a(1)');
ylabel('a(2)');
zlabel('a(3)');
view([37.5 30]);
```

Hopfield Network State Space



The function NEWHOP creates Hopfield networks given the stable points T.

```
net = newhop(T);
```

Here we define a random starting point and simulate the Hopfield network for 50 steps. It should reach one of its stable points.

```
a = {rands(3,1)};
```

```
[y,Pf,Af] = net({1 10},[],a);
```

We can make a plot of the Hopfield networks activity.

Sure enough, the network ends up at a designed stable point in the corner.

```
record = [cell2mat(a) cell2mat(y)];
```

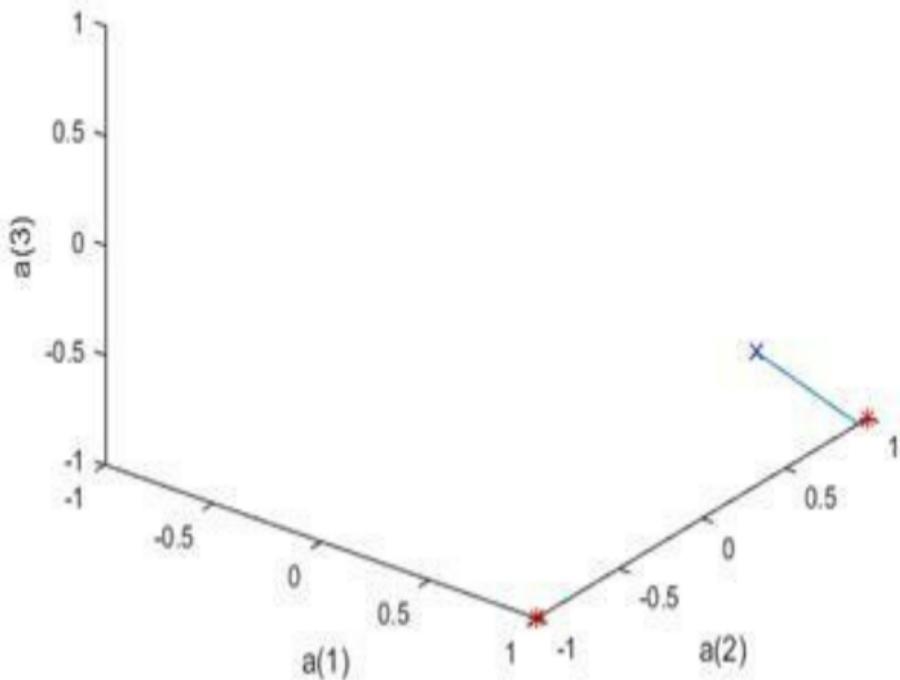
```
start = cell2mat(a);
```

```
hold on
```

```
plot3(start(1,1),start(2,1),start(3,1),'bx',
```

...
record(1,:),record(2,:),record(3,:))

Hopfield Network State Space



We repeat the simulation for 25 more

randomly generated initial conditions.

```
color = 'rgbmy';
```

```
for i = 1:25
```

```
    a = {rands(3,1)};
```

```
    [y,Pf,Af] = net([1 10],{},a);
```

```
    record = [cell2mat(a) cell2mat(y)];
```

```
    start = cell2mat(a);
```

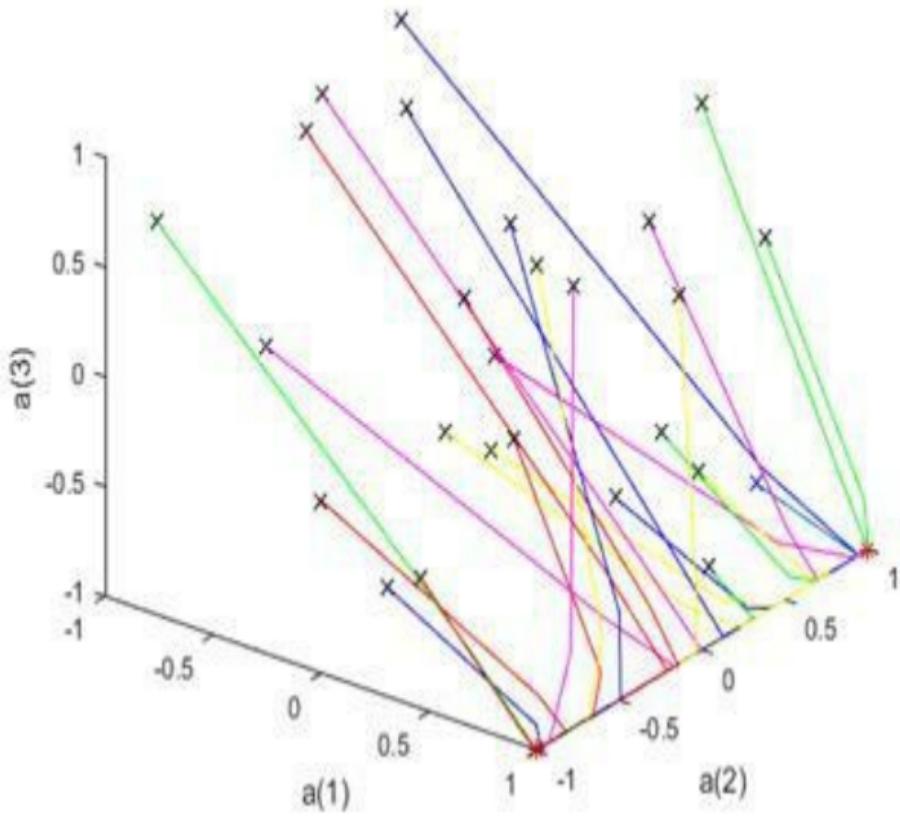
```
plot3(start(1,1),start(2,1),start(3,1),'kx',
```

```
...
```

```
record(1,:),record(2,:),record(3,:),color(
```

```
end
```

Hopfield Network State Space



Now we simulate the Hopfield for the following initial conditions, each a column vector of P .

These points were exactly between the two target stable points. The result is that they all move into the center of the state space, where an undesired stable point exists.

```
P = [ 1.0 -1.0 -0.5 1.00 1.00 0.0; ...
      0.0 0.0 0.0 0.00 0.00 -0.0; ...
      -1.0 1.0 0.5 -1.01 -1.00 0.0];
```

```
cla
```

```
plot3(T(1,:),T(2,:),T(3,:),'r*')
```

```
color = 'rgbmy';
```

```
for i = 1:6
```

```
    a = {P(:,i)};
```

```
[y,Pf,Af] = net( {1 10}, {}, a);
```

```
record = [cell2mat(a) cell2mat(y)];
```

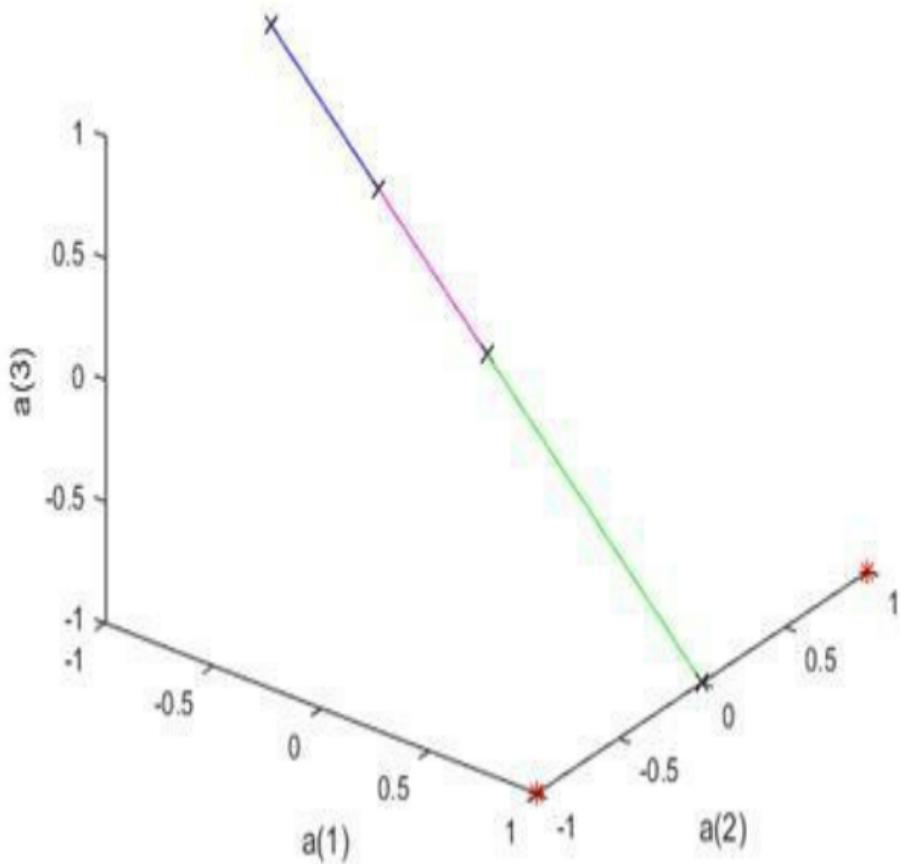
```
start = cell2mat(a);
```

```
plot3(start(1,1),start(2,1),start(3,1),'kx',
```

```
...
```

```
record(1,:),record(2,:),record(3,:),color(  
end
```

Hopfield Network State Space



6.7 HOPFIELD SPURIOUS STABLE POINTS

A Hopfield network with five neurons is designed to have four stable equilibria. However, unavoidably, it has other undesired equilibria.

We would like to obtain a Hopfield network that has the four stable points defined by the two target (column) vectors in T .

$$T = [+1 \quad +1 \quad -1 \quad +1 ; \\ \dots]$$

-1 +1 +1 -1 ;

...

-1 -1 -1 +1 ;

...

+1 +1 +1 +1 ;

...

-1 -1 +1 +1] ;

The function NEWHOP creates Hopfield

networks given the stable points T.

```
net = newhop(T);
```

Here we define 4 random starting points and simulate the Hopfield network for 50 steps.

Some initial conditions will lead to desired stable points. Others will lead to undesired stable points.

```
P = {rands(5, 4)};
```

```
[Y, Pf, Af] = net({4  
50}, {}, P);
```

Y { end }

ans =

1 -1

1 1

1 -1

1 -1

-1 -1

1 1

1 1

1 1

-1 1

1 1

Chapter 7

PERCEPTRONS

7.1 PERCEPTRON

In machine learning, the **perceptron** is an algorithm for supervised learning of binary classifiers (functions that can decide whether an input, represented by a vector of numbers, belongs to some specific class or not).^[1] It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time. The perceptron algorithm dates back to

the late 1950s; its first implementation, in custom hardware, was one of the first artificial neural networks to be produced.

The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt, funded by the United States Office of Naval Research.^[5] The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron". This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to

the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.

In a 1958 press conference organized by the US Navy, Rosenblatt made statements about the perceptron that caused a heated controversy among the fledgling [AI](#) community; based on Rosenblatt's statements, *The New York Times* reported the perceptron to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

Although the perceptron initially seemed promising, it was quickly proved that

perceptrons could not be trained to recognise many classes of patterns. This caused the field of neural network research to stagnate for many years, before it was recognised that a feedforward neural network with two or more layers (also called a multilayer perceptron) had far greater processing power than perceptrons with one layer (also called a single layer perceptron). Single layer perceptrons are only capable of learning linearly separable patterns; in 1969 a famous book entitled *Perceptrons* by Marvin Minsky and Seymour Papert showed that it was impossible for these classes of network to learn an XOR function. It is

often believed that they also conjectured (incorrectly) that a similar result would hold for a multi-layer perceptron network. However, this is not true, as both Minsky and Papert already knew that multi-layer perceptrons were capable of producing an XOR function. (See the page on *Perceptrons (book)* for more information.) Three years later Stephen Grossberg published a series of papers introducing networks capable of modelling differential, contrast-enhancing and XOR functions. (The papers were published in 1972 and 1973, see e.g.: *Grossberg (1973). "Contour enhancement, short-term memory, and constancies in*

reverberating neural networks" (PDF). *Studies in Applied Mathematics*. 52: 213–257.).

Nevertheless, the often-misquoted Minsky/Papert text caused a significant decline in interest and funding of neural network research. It took ten more years until neural network research experienced a resurgence in the 1980s. This text was reprinted in 1987 as "Perceptrons - Expanded Edition" where some errors in the original text are shown and corrected.

The kernel perceptron algorithm was already introduced in 1964 by Aizerman et al. Margin bounds guarantees were given for the Perceptron algorithm in the

general non-separable case first by Freund and Schapire (1998), and more recently by Mohri and Rostamizadeh (2013) who extend previous results and give new L1 bounds.

The perceptron is a linear classifier, therefore it will never get to the state with all the input vectors classified correctly if the training set D is not linearly separable, i.e. if the positive examples can not be separated from the negative examples by a hyperplane. In this case, no "approximate" solution will be gradually approached under the standard learning algorithm, but instead learning will fail completely. Hence, if

linear separability of the training set is not known *a priori*, one of the training variants below should be used.

But if the training set *is* linearly separable, then the perceptron is guaranteed to converge, and there is an upper bound on the number of times the perceptron will adjust its weights during the training.

While the perceptron algorithm is guaranteed to converge on *some* solution in the case of a linearly separable training set, it may still pick *any* solution and problems may admit many solutions of varying quality. The *perceptron of optimal stability*, nowadays better known as the linear support vector

machine, was designed to solve this problem.

7.2 CLASSIFICATION WITH A 2-INPUT PERCEPTRON

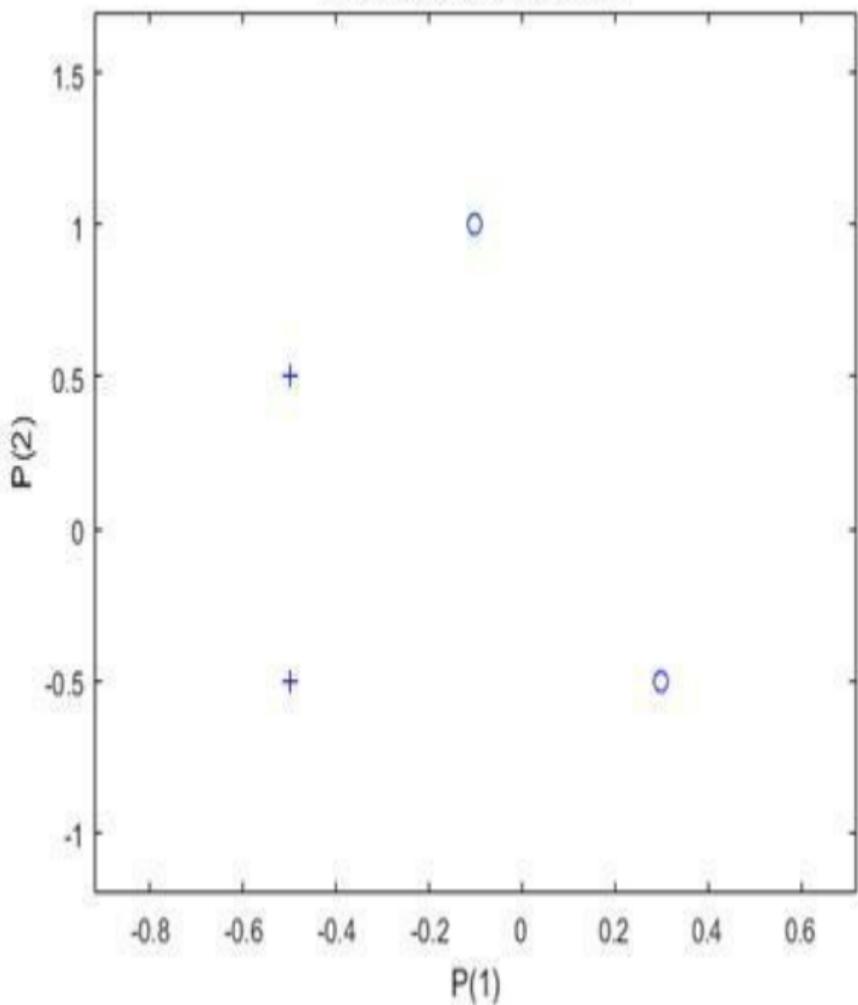
A 2-input hard limit neuron is trained to classify 5 input vectors into two categories.

Each of the five column vectors in X defines a 2-element input vectors and a row vector T defines the vector's target categories. We can plot these vectors with PLOTPV.

$$X = \begin{bmatrix} -0.5 & -0.5 & +0.3 \\ -0.1 & \dots \end{bmatrix}$$

```
-0.5 +0.5 -0.5  
+1.0] ;  
  
T = [1 1 0 0] ;  
  
plotpv(X,T) ;
```

Vectors to be Classified



The perceptron must properly classify the 5 input vectors in X into the two

categories defined by T. Perceptrons have HARDLIM neurons. These neurons are capable of separating an input space with a straight line into two categories (0 and 1).

Here PERCEPTRON creates a new neural network with a single neuron. The network is then configured to the data, so we can examine its initial weight and bias values. (Normally the configuration step can be skipped as it is automatically done by ADAPT or TRAIN.)

```
net = perceptron;
```

```
net =  
configure(net,X,T);
```

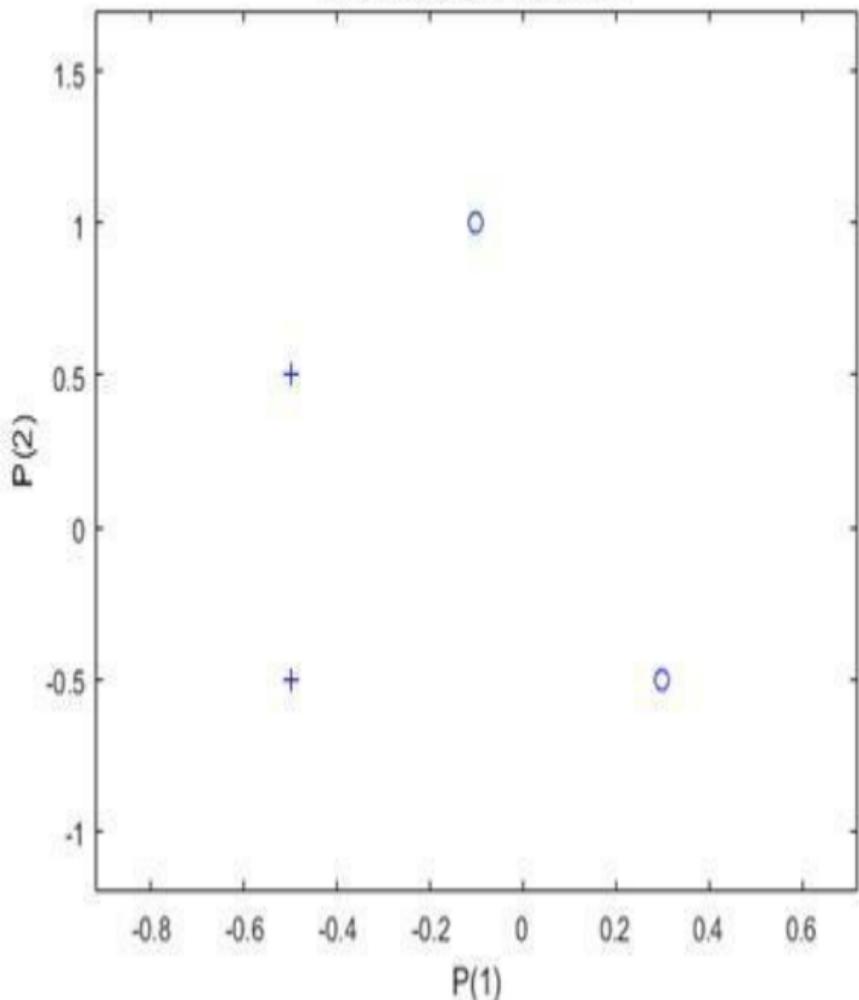
The input vectors are replotted with the neuron's initial attempt at classification.

The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
plotpv(X,T);
```

```
plotpc(net.IW{1},net
```

Vectors to be Classified



Here the input and target data are

converted to sequential data (cell array where each column indicates a timestep) and copied three times to form the series XX and TT.

ADAPT updates the network for each timestep in the series and returns a new network object that performs as a better classifier.

XX =

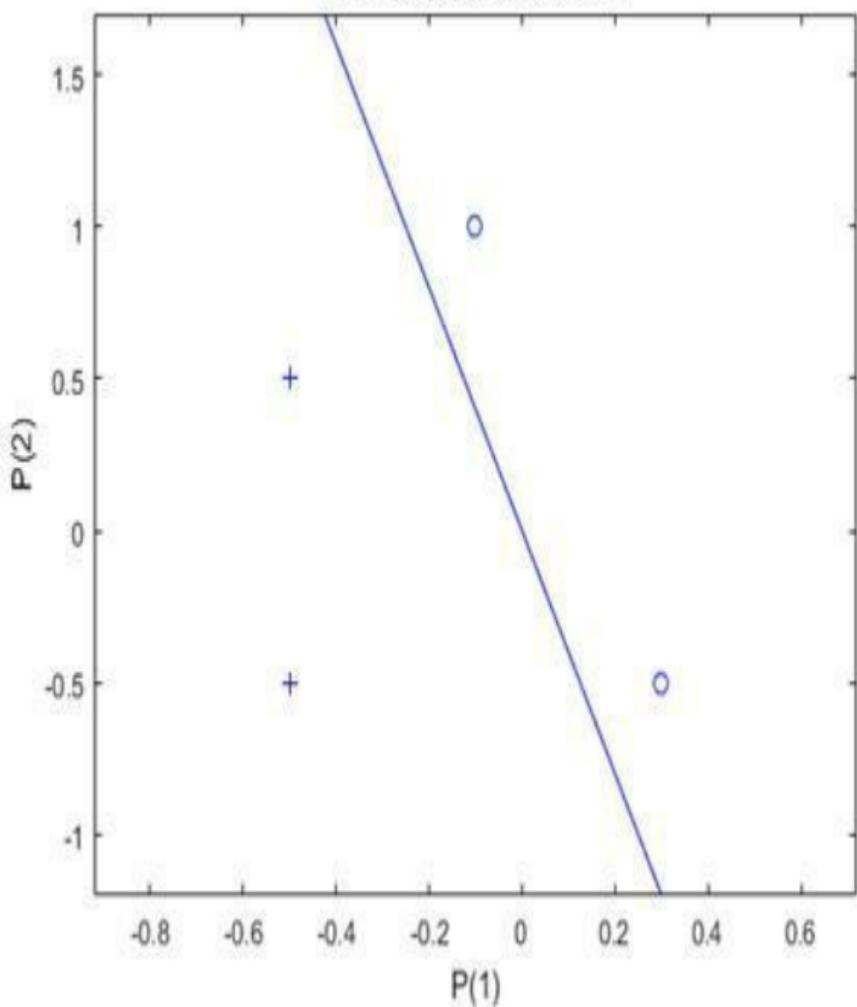
repmat (con2seq (X) , 1 ,

TT =

repmat (con2seq (T) , 1 ,

```
net =  
adapt(net, XX, TT);  
  
plotpc(net.IW{1}, net
```

Vectors to be Classified



Now SIM is used to classify any other input vector, like $[0.7; 1.2]$. A plot of

this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
x = [0.7; 1.2];
```

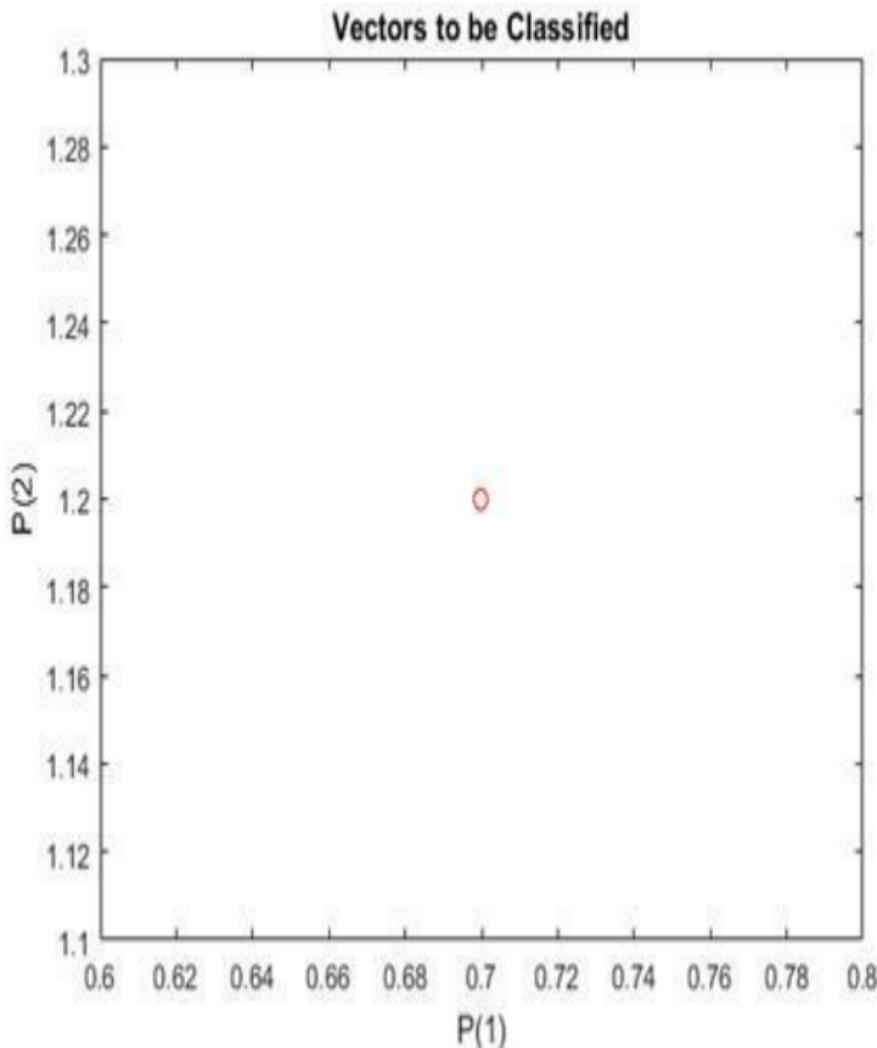
```
y = net(x);
```

```
plotpv(x,y);
```

```
point =
```

```
findobj(gca,'type','
```

```
point.Color = 'red';
```



Turn on "hold" so the previous plot is

not erased and plot the training set and the classification line.

The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus).

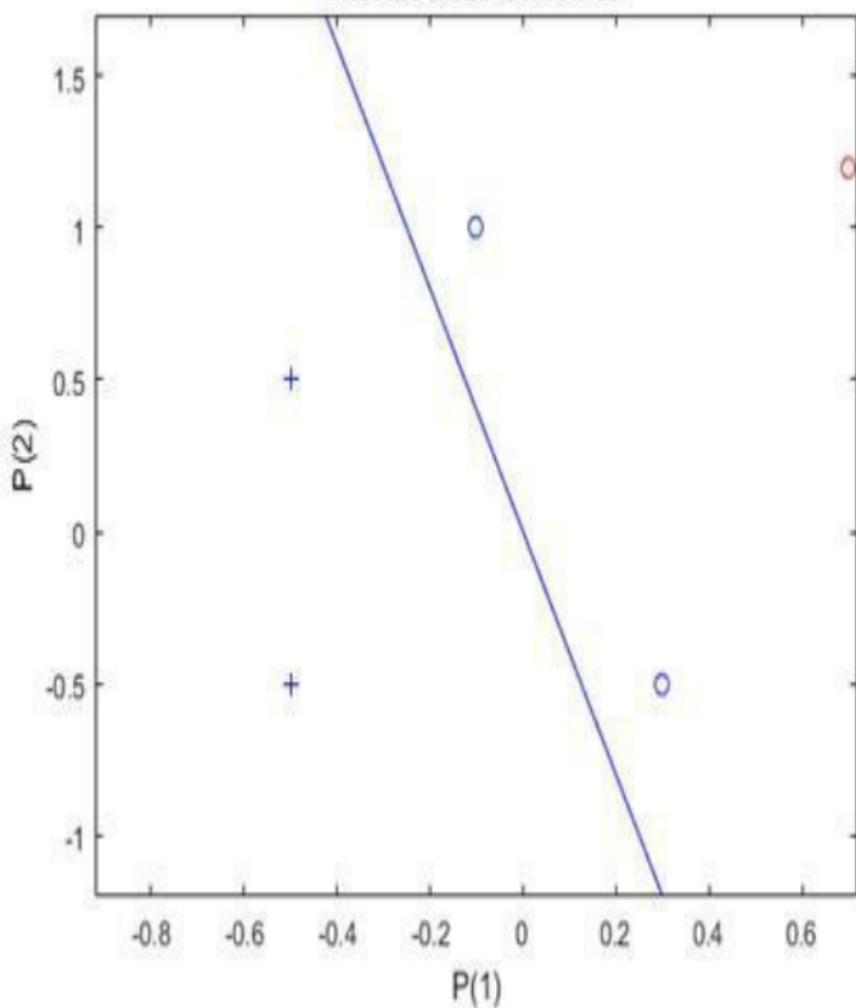
```
hold on;
```

```
plotpv(X, T);
```

```
plotpc(net.IW{1}, net
```

```
hold off;
```

Vectors to be Classified



7.3 OUTLIER INPUT VECTORS

A 2-input hard limit neuron is trained to classify 5 input vectors into two categories. However, because 1 input vector is much larger than all of the others, training takes a long time.

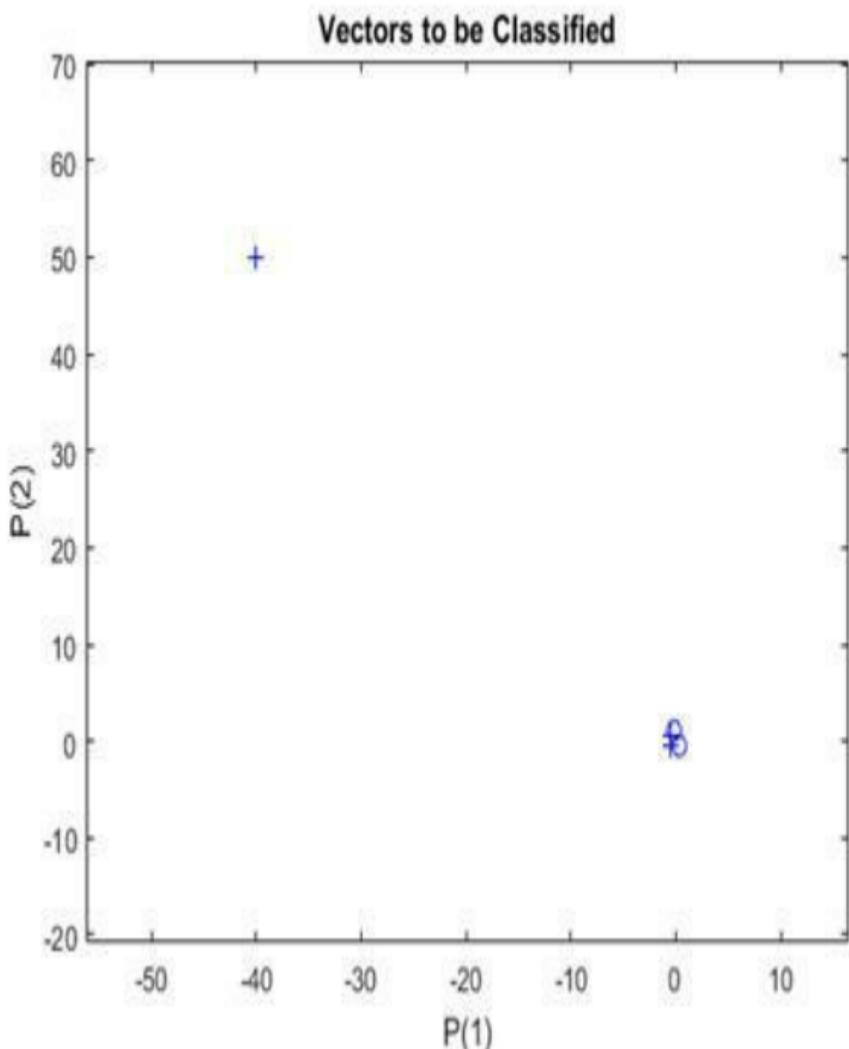
Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

$$X = [-0.5 \ -0.5 \ +0.3$$

```
-0.1 -40; -0.5 +0.5  
-0.5 +1.0 50];
```

```
T = [1 1 0 0 1];
```

```
plotpv(X,T);
```



Note that 4 input vectors have much smaller magnitudes than the fifth vector

in the upper left of the plot. The perceptron must properly classify the 5 input vectors in X into the two categories defined by T.

PERCEPTRON creates a new network which is then configured with the input and target data which results in initial values for its weights and bias.
(Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

```
net = perceptron;
```

```
net =
```

```
configure(net,X,T);
```

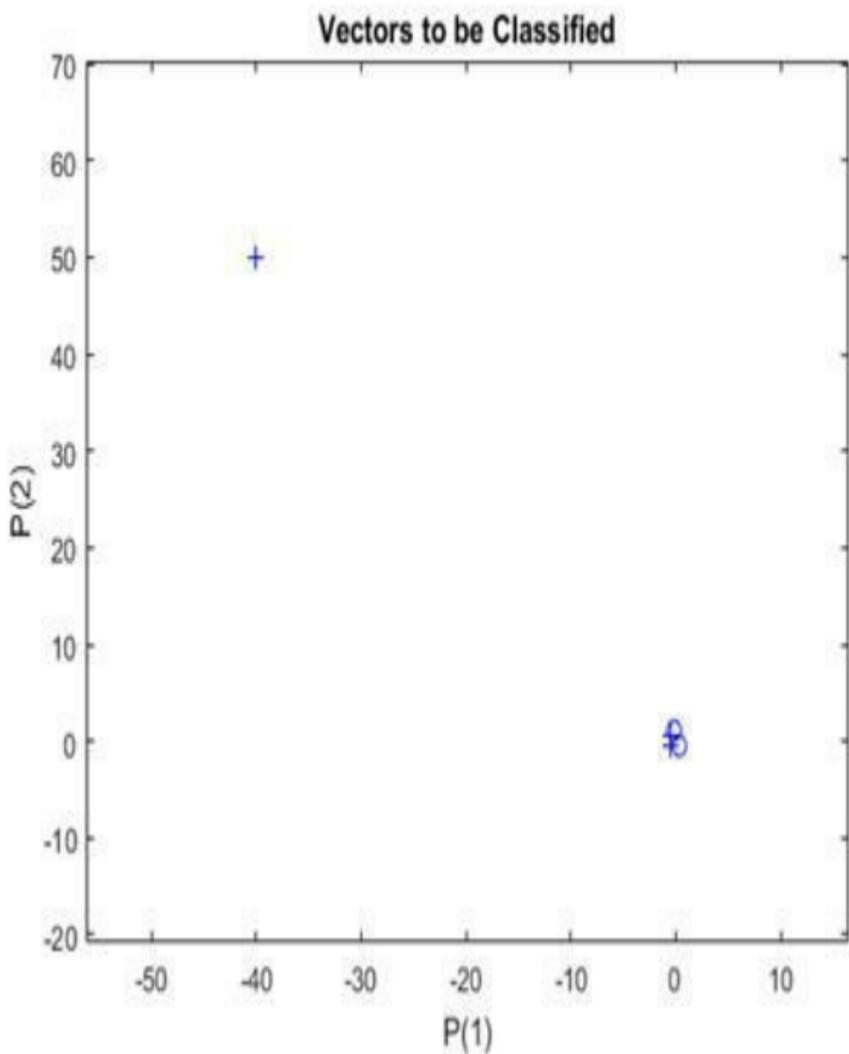
Add the neuron's initial attempt at classification to the plot.

The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

hold on

```
linehandle =
```

```
plotpc(net.IW{1},net
```



ADAPT returns a new network object that performs as a better classifier, the

network output, and the error. This loop adapts the network and plots the classification line, until the error is zero.

```
E = 1;
```

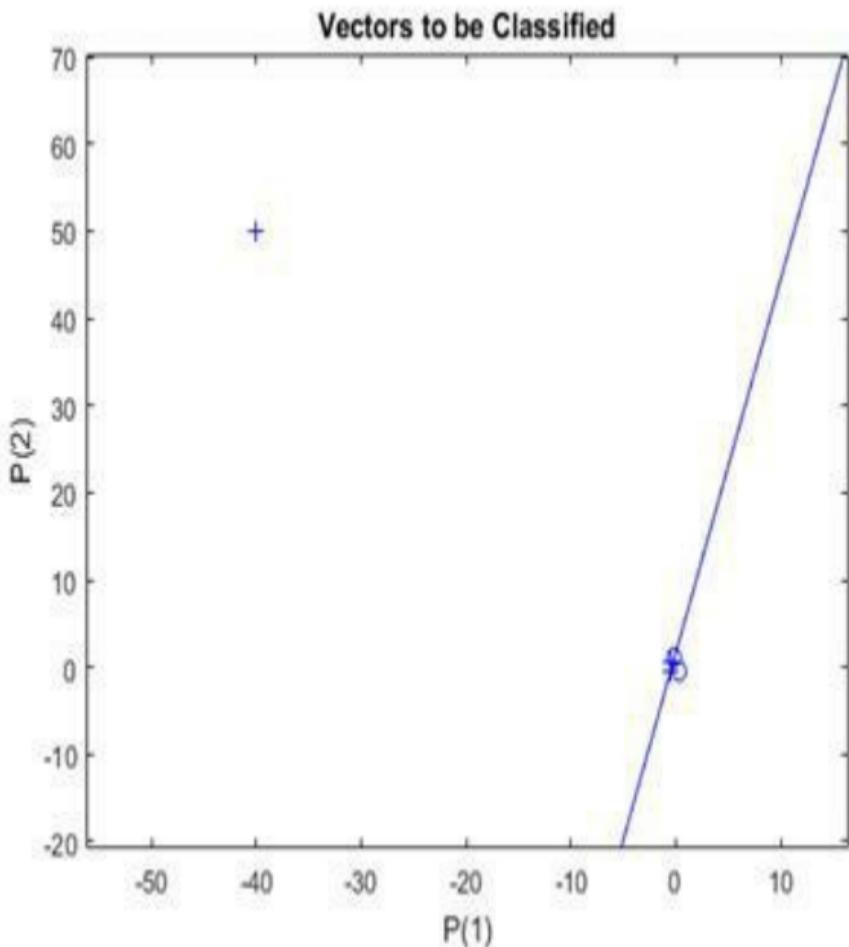
```
while (sse(E))
```

```
[net, Y, E] =  
adapt(net, X, T);
```

```
linehandle =  
plotpc(net.IW{1}, net
```

drawnow;

end



Note that it took the perceptron three passes to get it right. This a long time for such a simple problem. The reason for the long training time is the outlier

vector. Despite the long training time, the perceptron still learns properly and can be used to classify other inputs.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

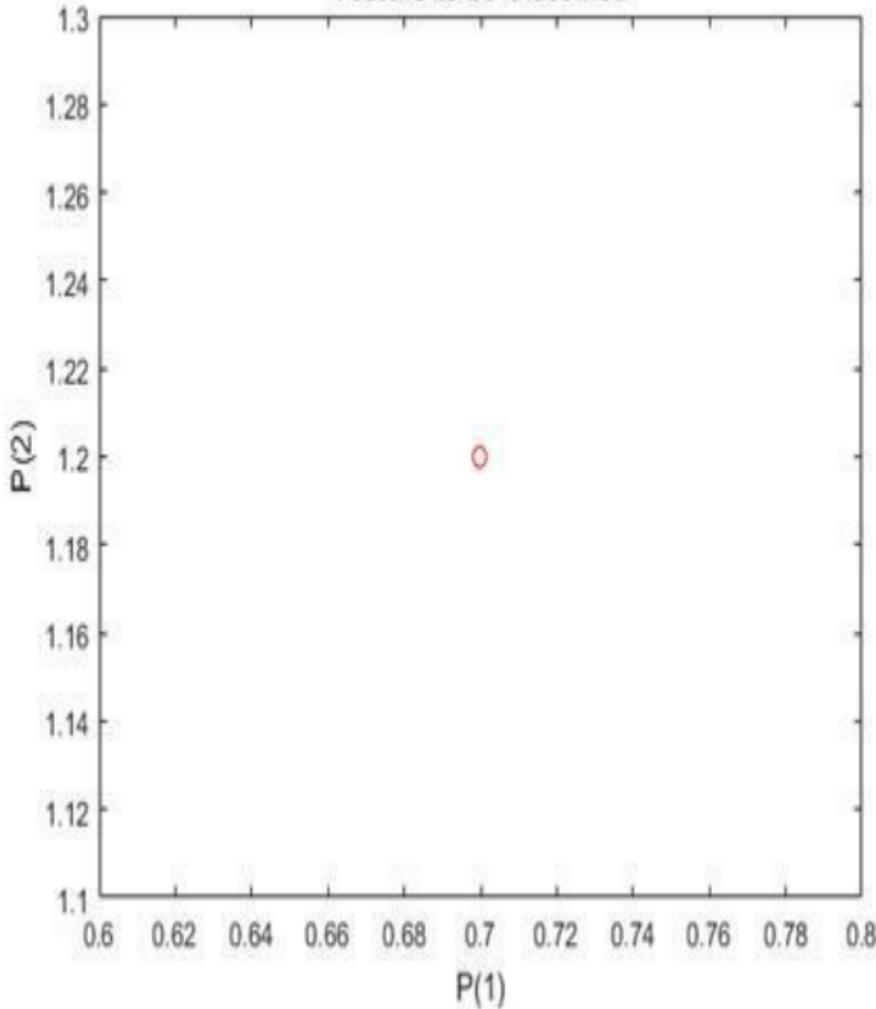
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
x = [0.7; 1.2];
```

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

```
plotpv(x,y);  
  
circle =  
findobj(gca,'type','  
  
circle.Color =  
'red';
```

Vectors to be Classified



Turn on "hold" so the previous plot is not erased. Add the training set and the

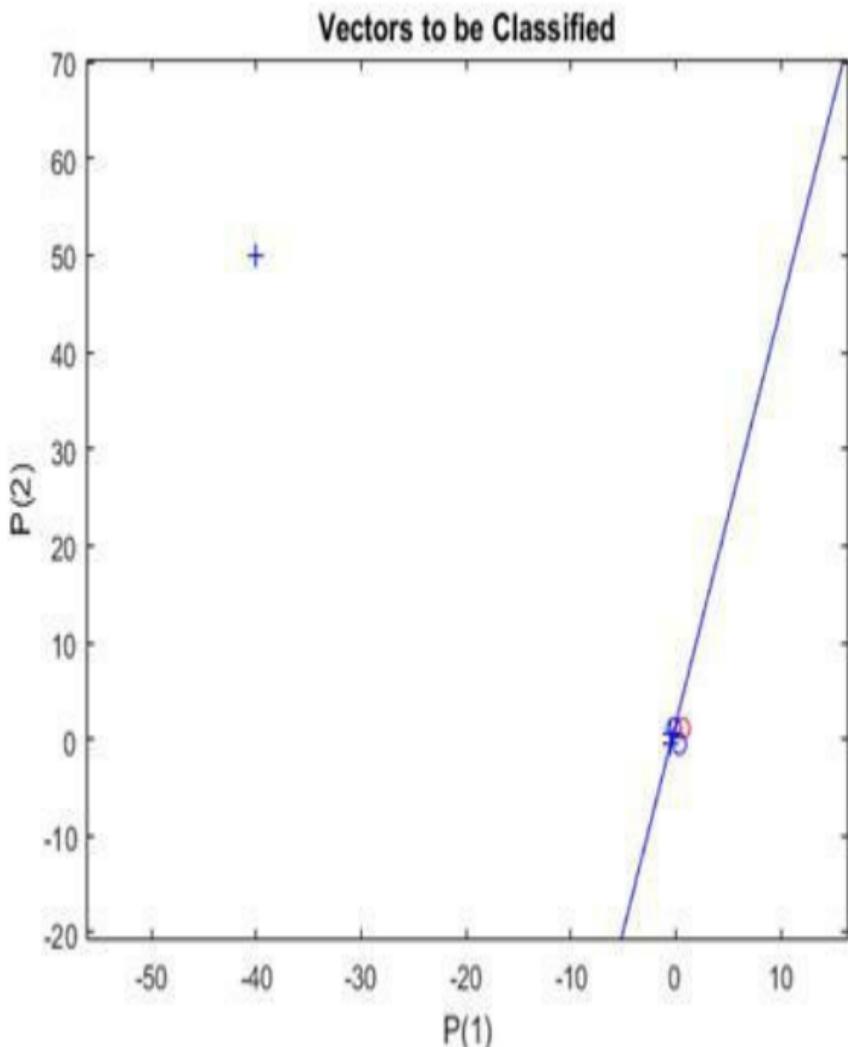
classification line to the plot.

```
hold on;
```

```
plotpv(X,T);
```

```
plotpc(net.IW{1},net
```

```
hold off;
```

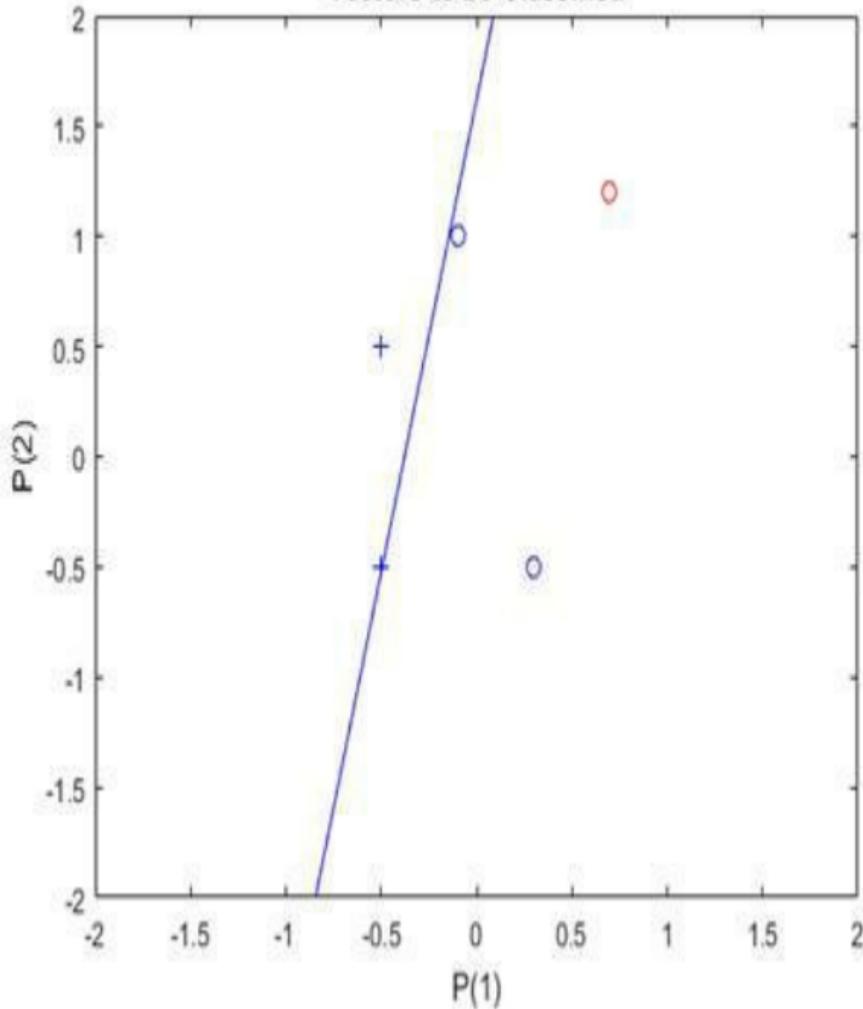


Finally, zoom into the area of interest.
The perceptron correctly classified our

new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus). Despite the long training time, the perceptron still learns properly. To see how to reduce training times associated with outlier vectors, see the "Normalized Perceptron Rule" example.

```
axis( [-2 2 -2 2] );
```

Vectors to be Classified

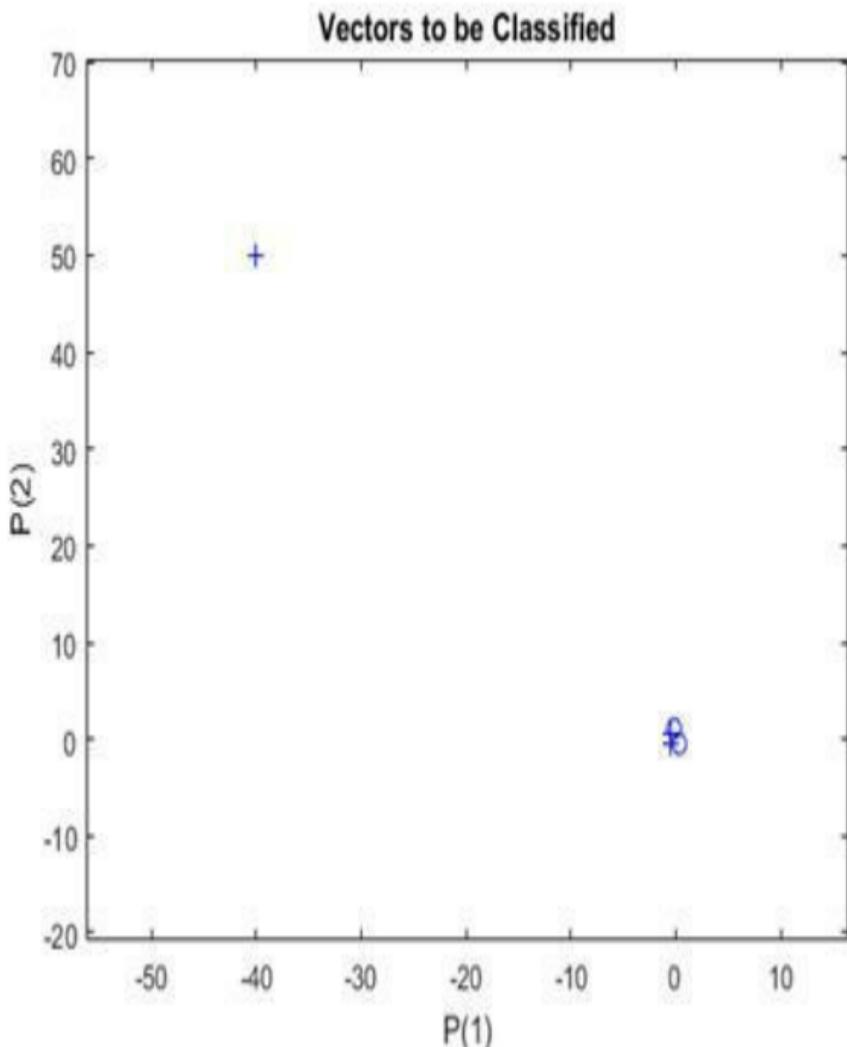


7.4 NORMALIZED PERCEPTRON RULE

A 2-input hard limit neuron is trained to classify 5 input vectors into two categories. Despite the fact that one input vector is much bigger than the others, training with LEARNPN is quick.

Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

```
X = [ -0.5 -0.5 +0.3  
-0.1 -40; ...  
      -0.5 +0.5 -0.5  
+1.0 50];  
  
T = [1 1 0 0 1];  
  
plotpv(X,T);
```



Note that 4 input vectors have much smaller magnitudes than the fifth vector

in the upper left of the plot. The perceptron must properly classify the 5 input vectors in X into the two categories defined by T.

PERCEPTRON creates a new network with LEARPN learning rule, which is less sensitive to large variations in input vector size than LEARNP (the default).

The network is then configured with the input and target data which results in initial values for its weights and bias. (Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

net =

```
perceptron('hardlim'
```

```
net =
```

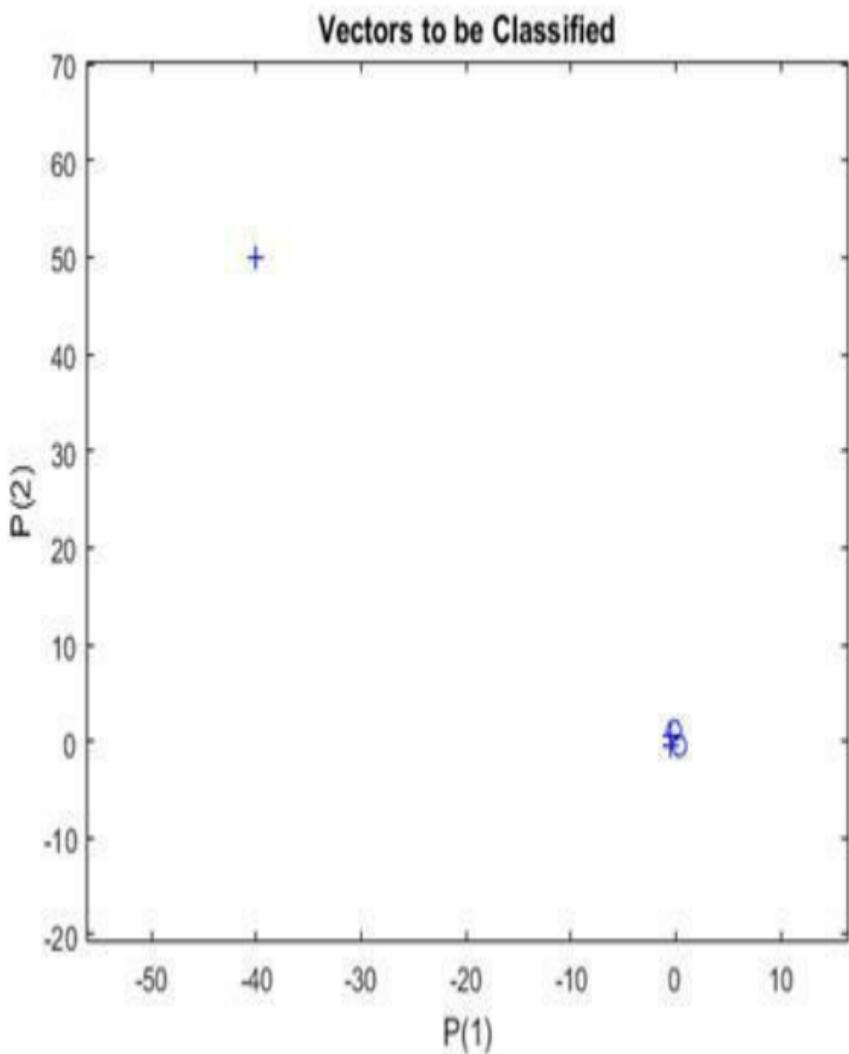
```
configure(net,X,T);
```

Add the neuron's initial attempt at classification to the plot.

The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
hold on
```

```
linehandle =  
plotpc(net.IW{1},net
```



ADAPT returns a new network object that performs as a better classifier, the

network output, and the error. This loop allows the network to adapt, plots the classification line, and continues until the error is zero.

```
E = 1;
```

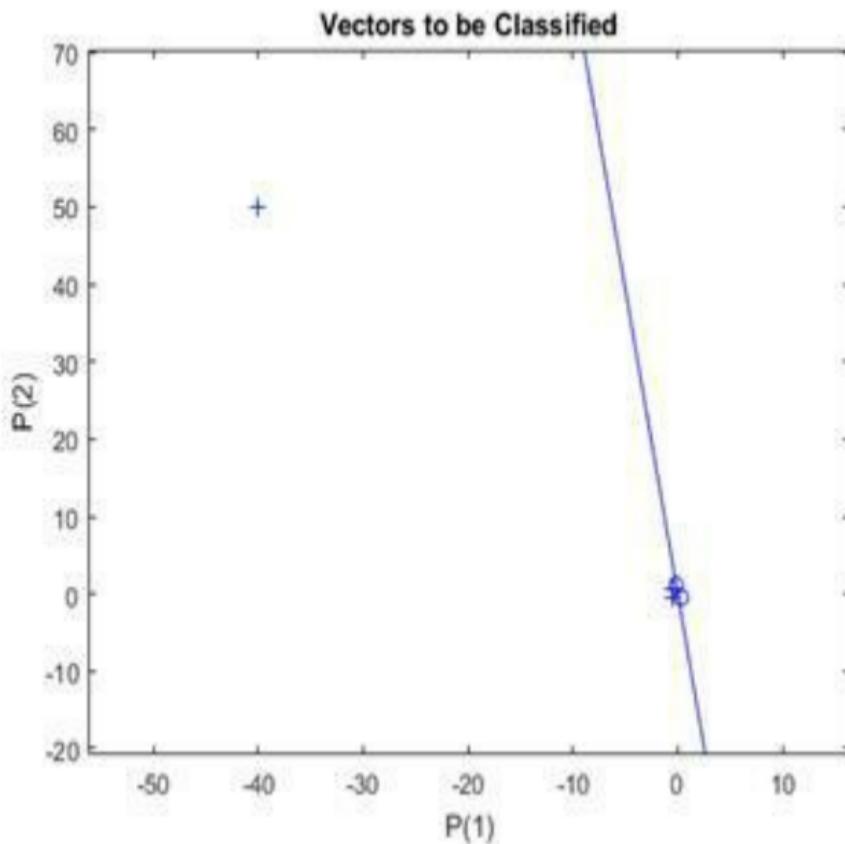
```
while (sse(E))
```

```
[net, Y, E] =  
adapt(net, X, T);
```

```
linehandle =  
plotpc(net.IW{1}, net
```

```
drawnow;
```

```
end
```



Note that training with LEARNP took only 3 epochs, while solving the same problem with LEARNPN required 32 epochs. Thus, LEARNPN does much better job than LEARNP when there are large variations in input vector size.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

$$x = [0.7; 1.2];$$

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

```
plotpv(x,y);
```

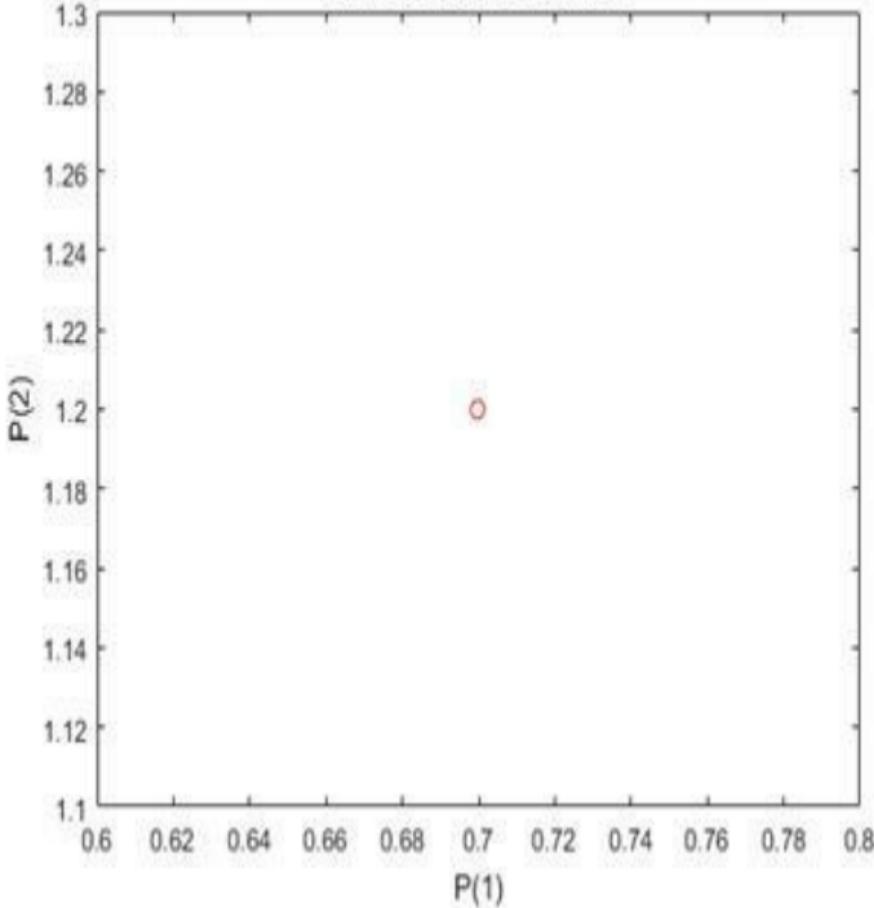
```
circle =
```

```
findobj(gca,'type','..
```

```
circle.Color =
```

```
'red';
```

Vectors to be Classified



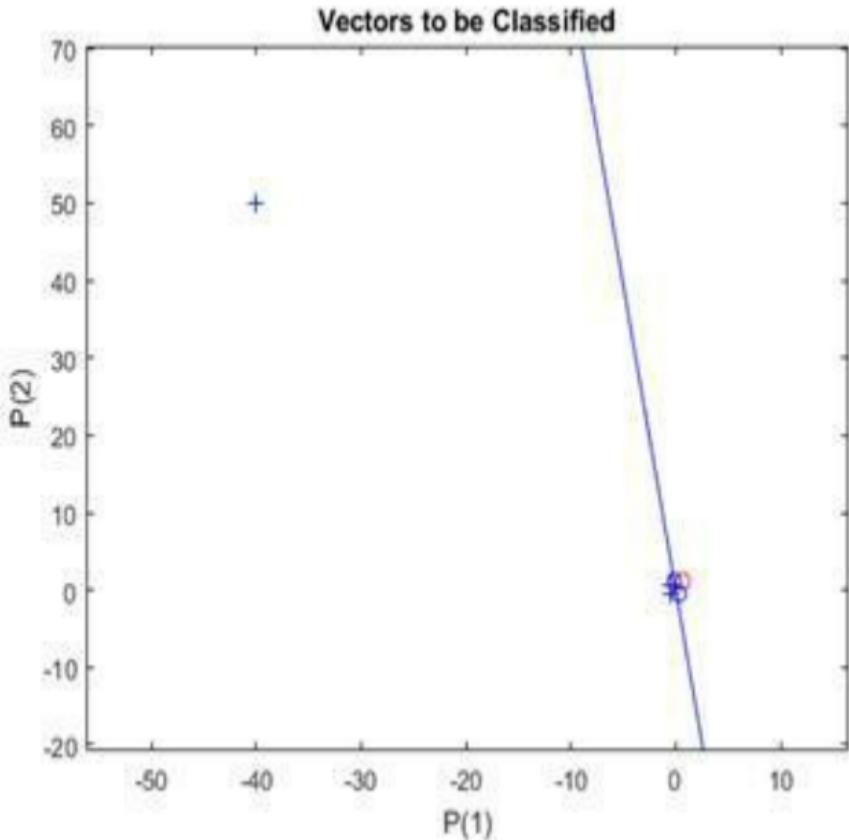
Turn on "hold" so the previous plot is not erased. Add the training set and the classification line to the plot.

```
hold on;
```

```
plotpv(X,T);
```

```
plotpc(net.IW{1},net
```

```
hold off;
```

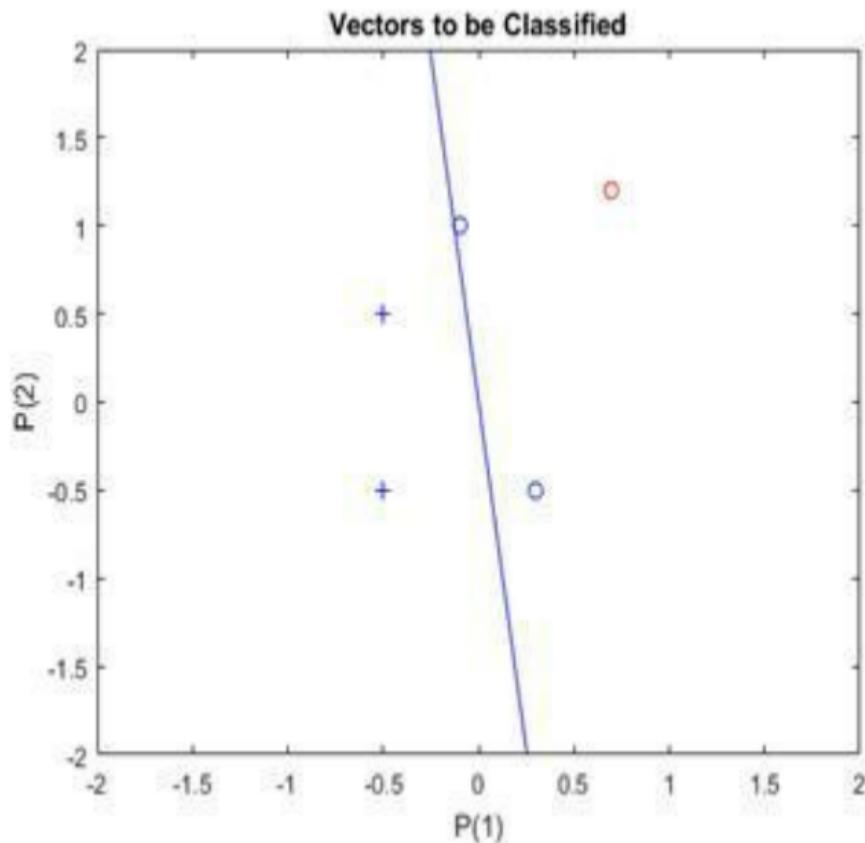


Finally, zoom into the area of interest.

The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus). The perceptron

learns properly in much shorter time in spite of the outlier (compare with the "Outlier Input Vectors" example).

```
axis( [-2 2 -2 2] );
```



7.5 LINEARLY NON- SEPARABLE VECTORS

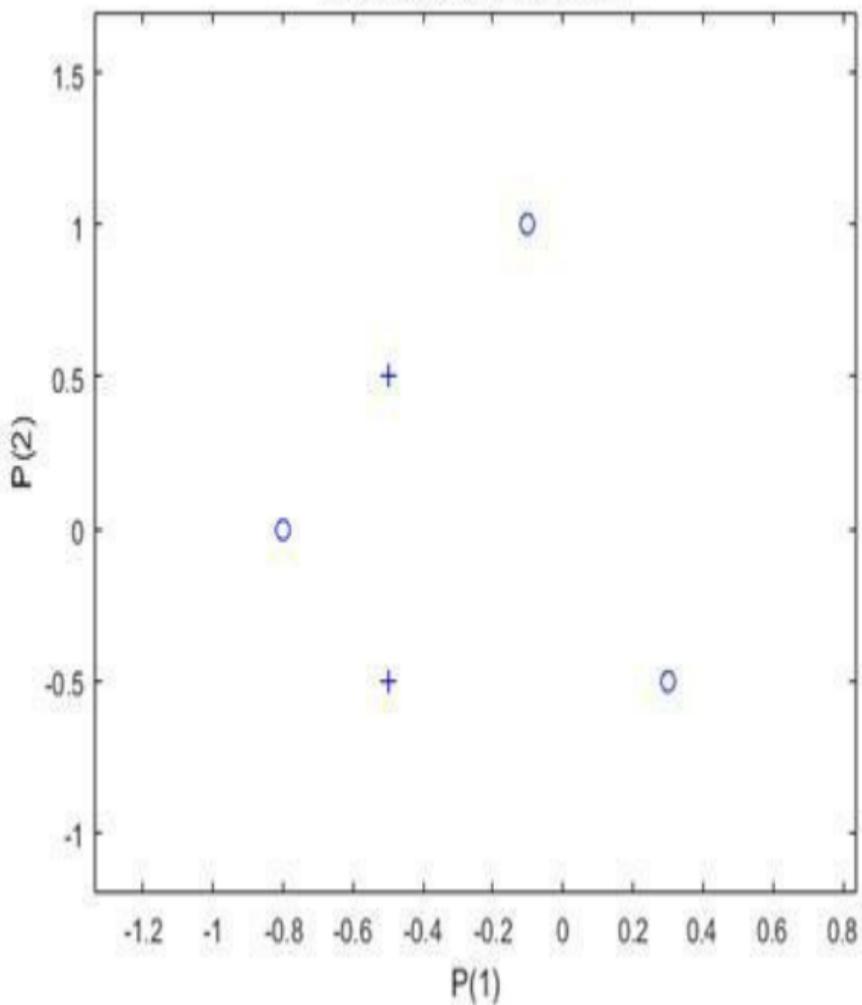
A 2-input hard limit neuron fails to properly classify 5 input vectors because they are linearly non-separable.

Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

```
X = [ -0.5 -0.5 +0.3  
-0.1 -0.8; ... ]
```

```
-0.5 +0.5 -0.5  
+1.0 +0.0 ] ;  
  
T = [1 1 0 0 0] ;  
  
plotpv(X,T) ;
```

Vectors to be Classified



The perceptron must properly classify the input vectors in X into the categories

defined by T. Because the two kinds of input vectors cannot be separated by a straight line, the perceptron will not be able to do it.

Here the initial perceptron is created and configured. (The configuration step is normally optional, as it is performed automatically by ADAPT and TRAIN.)

```
net = perceptron;
```

```
net =  
configure(net, X, T);
```

Add the neuron's initial attempt at classification to the plot. The initial

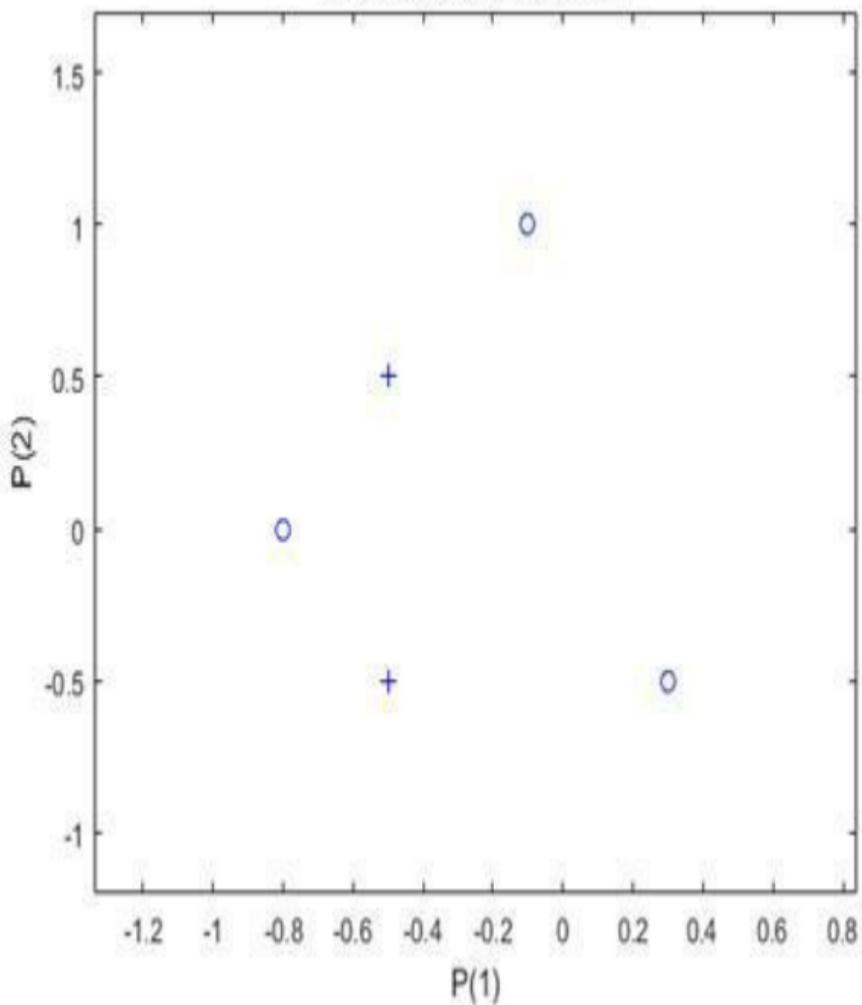
weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot.

hold on

plotpv(X, T);

linehandle =
plotpc(net.IW{1}, net

Vectors to be Classified



ADAPT returns a new network after learning on the input and target data, the

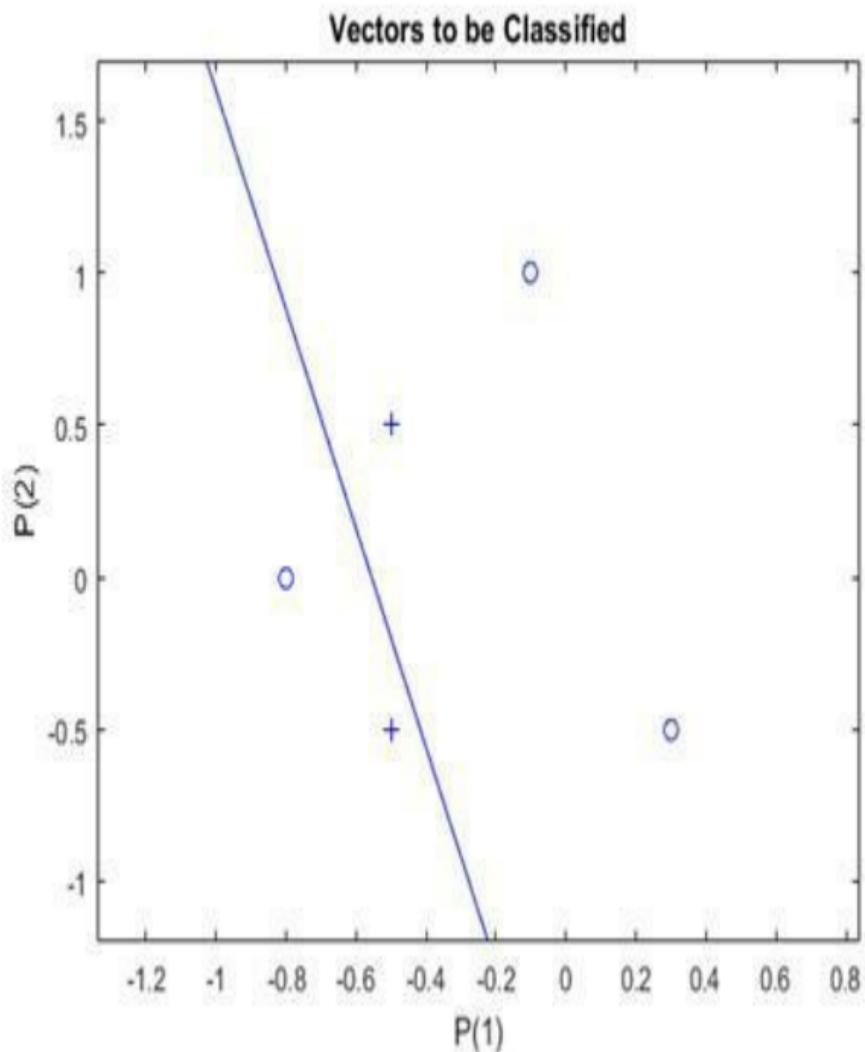
outputs and error. The loop allows the network to repeatedly adapt, plots the classification line, and stops after 25 iterations.

```
for a = 1:25
```

```
[net,Y,E] =  
adapt(net,X,T);
```

```
linehandle =  
plotpc(net.IW{1},net  
drawnow;
```

end;



Note that zero error was never obtained.

Despite training, the perceptron has not become an acceptable classifier. Only being able to classify linearly separable data is the fundamental limitation of perceptrons.

Chapter 8

TIME SERIES NEURAL NETWORKS. EXAMPLES

8.1 FUNCTIONS FOR MODELING AND PREDICTION

The more important functions for modeling and prediction with NARX and time delay networks are the following:

nnstart	Neural network getting started GUI
view	View neural network
timedelaynet	Time delay neural network
narxnet	Nonlinear autoregressive neural network
narnet	Nonlinear autoregressive neural network
layrecnet	Layer recurrent neural network
distdelaynet	Distributed delay network
train	Train neural network
gensim	Generate Simulink block for neural network
adddelay	Add delay to neural network response
removedelay	Remove delay to neural network's response
closeloop	Convert neural network open-loop function to closed-loop
openloop	Convert neural network closed-loop function to open-loop
ploterrhist	Plot error histogram

<u>plotinerrcorr</u>	Plot input to error time-series cross-
<u>plotregression</u>	Plot linear regression
<u>plotresponse</u>	Plot dynamic network time series re-
<u>plotterrcorr</u>	Plot autocorrelation of error time ser
<u>genFunction</u>	Generate MATLAB function for sim

8.2 TIMEDELAYNET

Time delay neural network

Syntax

`timedelaynet(inputDelays,hiddenSize`

Description

Time delay networks are similar to feedforward networks, except that the input weight has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the distributed delay neural network ([distdelaynet](#)),

which has delays on the layer weights in addition to the input weight.

`timedelaynet(inputDelays,hiddenSizes,trainFcn)`
these arguments,

<code>inputDelays</code>	Row vector of increasing 0 or positive delays (default = 1)
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (default = [1 1])
<code>trainFcn</code>	Training function (default = 'trainlm')

and returns a time delay neural network.

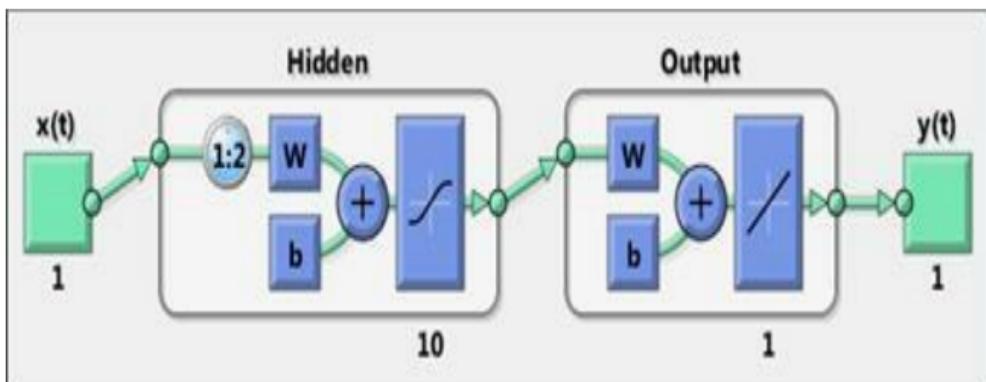
Examples. Time Delay Network

Here a time delay neural network is used to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;  
net = timedelaynet(1:2,10);
```

```
[Xs,Xi,Ai,Ts] = prepares(net,X,T);  
net = train(net,Xs,Ts,Xi,Ai);  
view(net)  
Y = net(Xs,Xi,Ai);  
perf = perform(net,Ts,Y)  
perf =
```

0.0225



8.3 NARXNET

Nonlinear autoregressive neural network
with external input

Syntax

```
narxnet(inputDelays, feedbackDelays,
```

Description

NARX (Nonlinear autoregressive with external input) networks can learn to predict one time series given past values of the same time series, the feedback input, and another time series, called the external or exogenous time series.

`narxnet(inputDelays,feedbackDelays,hiddenSizes)`
these arguments,

<code>inputDelays</code>	Row vector of increasing 0 or positive delays (
<code>feedbackDelays</code>	Row vector of increasing 0 or positive delays (
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (
<code>trainFcn</code>	Training function (default = ' <code>'trainlm'</code> ')

and returns a NARX neural network.

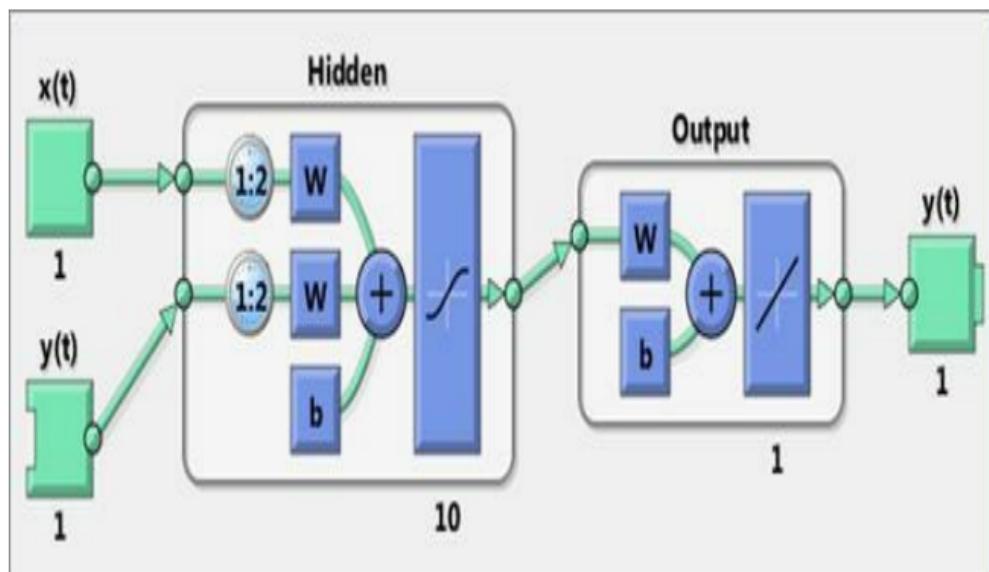
Examples. Use NARX Network For Time Series Problem

Here a NARX neural network is used to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;  
net = narxnet(1:2,1:2,10);
```

```
[Xs,Xi,Ai,Ts] = prepares(net,X,{},T);  
net = train(net,Xs,Ts,Xi,Ai);  
view(net)  
Y = net(Xs,Xi,Ai);  
perf = perform(net,Ts,Y)  
perf =
```

0.0192



Here the NARX network is simulated in

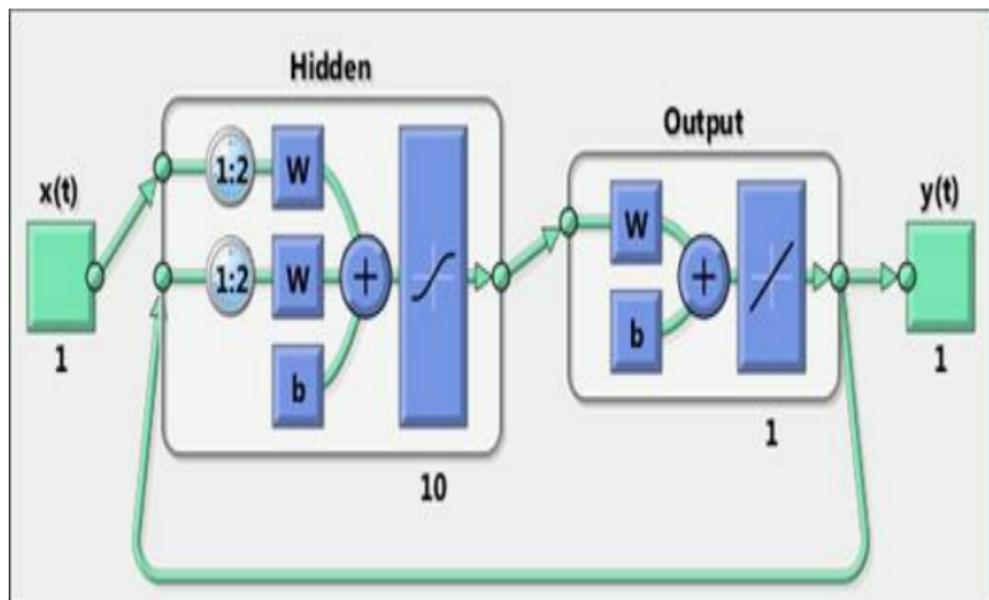
closed loop form.

```
netc = closeloop(net);
```

```
view(netc)
```

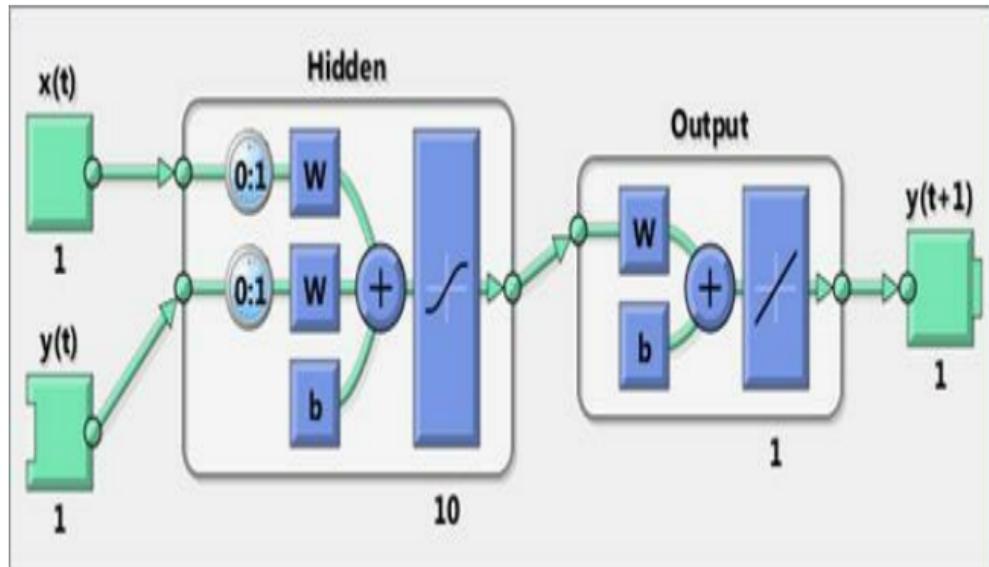
```
[Xs,Xi,Ai,Ts] = prepares(netc,X,{},T);
```

```
y = netc(Xs,Xi,Ai);
```



Here the NARX network is used to predict the next output, a timestep ahead of when it will actually appear.

```
netp = removedelay(net);  
view(netp)  
[Xs,Xi,Ai,Ts] = preparets(netp,X,{},{},T);  
y = netp(Xs,Xi,Ai);
```



8.4 NARNET

Nonlinear autoregressive neural network

Syntax

```
narnet(feedbackDelays,hiddenSizes,t
```

Description

NAR (nonlinear autoregressive) neural networks can be trained to predict a time series from that series past values.

narnet(feedbackDelays,hiddenSizes,train these arguments,

feedbackDelays	Row vector of increasing 0 or positive delays (
hiddenSizes	Row vector of one or more hidden layer sizes (

trainFcn

Training function (default = 'trainlm')

and returns a NAR neural network.

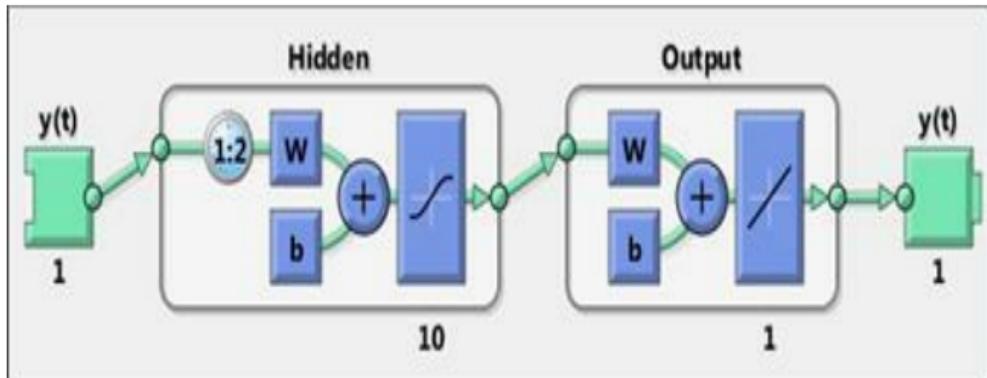
Examples. Nonlinear Autoregressive Neural Network

Here a NAR network is used to solve a simple time series problem.

```
T = simplenar_dataset;  
net = narne(1:2,10);  
[Xs,Xi,Ai,Ts] = preparets(net, {}, {}, T);  
net = train(net,Xs,Ts,Xi,Ai);  
view(net)  
Y = net(Xs,Xi);  
perf = perform(net,Ts,Y)
```

perf =

1.0100e-09



8.5 LAYRECNET

Layer recurrent neural network

Syntax

```
layrecnet(layerDelays,hiddenSizes,tr
```

Description

Layer recurrent neural networks are similar to feedforward networks, except that each layer has a recurrent connection with a tap delay associated with it. This allows the network to have an infinite dynamic response to time series input data. This network is similar

to the time delay ([timedelaynet](#)) and distributed delay ([distdelaynet](#)) neural networks, which have finite input responses.

`layrecnet(layerDelays,hiddenSizes,trainF`
these arguments,

<code>layerDelays</code>	Row vector of increasing 0 or positive delays (defau
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (defau
<code>trainFcn</code>	Training function (default = ' <code>trainlm</code> ')

and returns a layer recurrent neural network.

Examples. Recurrent Neural Network

Use a layer recurrent neural network to

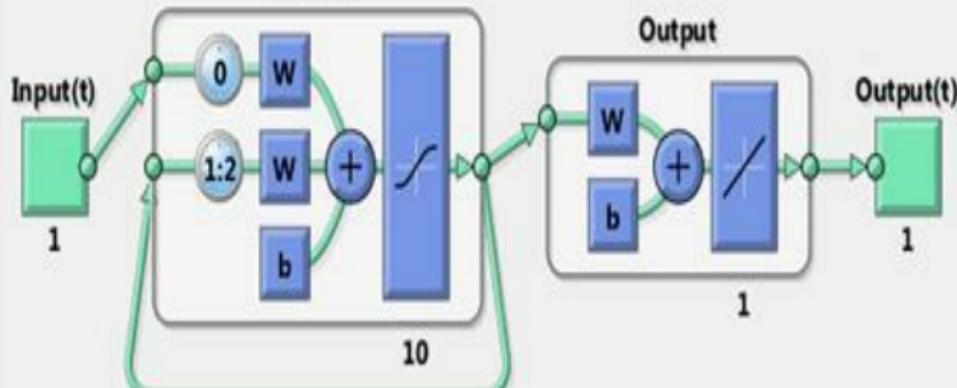
solve a simple time series problem.

```
[X,T] = simpleseries_dataset;
net = layrecnet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
```

perf =

6.1239e-11

Hidden



8.6 DISTDELAYNET

Distributed delay network

Syntax

```
distdelaynet(delays,hiddenSizes,train
```

Description

Distributed delay networks are similar to feedforward networks, except that each input and layer weights has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the

time delay neural network ([timedelaynet](#)), which only has delays on the input weight.

`distdelaynet(delays,hiddenSizes,trainFcn)`
these arguments,

<code>delays</code>	Row vector of increasing 0 or positive delays (default = 1)
<code>hiddenSizes</code>	Row vector of one or more hidden layer sizes (default = [1 1])
<code>trainFcn</code>	Training function (default = 'trainlm')

and returns a distributed delay neural network.

Examples. Distributed Delay Network

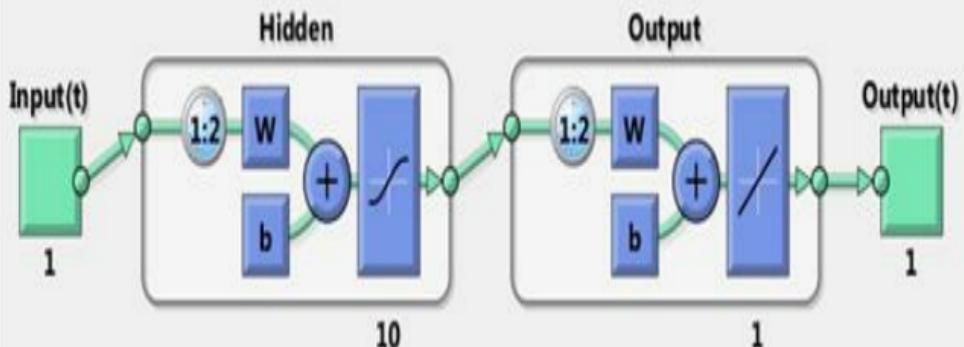
Here a distributed delay neural network is used to solve a simple time series

problem.

```
[X,T] = simpleseries_dataset;
net = distdelaynet( {1:2,1:2},10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
```

perf =

0.0323



8.7 TRAIN

Train neural network

Syntax

[net,tr] = train(net,X,T,Xi,Ai,EW)

[net,__] = train(__,'useParallel',__)

[net,__] = train(__,'useGPU',__)

[net,__] =

train(__,'showResources',__)

[net,__] =

train(Xcomposite,Tcomposite,__)

[net,__] = train(Xgpu,Tgpu,__)

net =

train(__,'CheckpointFile','path/name')

Description

train trains a network net according to net.trainFcn and net.trainParam.

[net,tr] = train(net,X,T,Xi,Ai,EW) takes

net Network

X Network inputs

T Network targets (default = zeros)

Xi Initial input delay conditions
(default = zeros)

Ai Initial layer delay conditions
(default = zeros)

EW Error weights

and returns

net Newly trained network

tr Training record (epoch and perf)

Note that T is optional and need only be used for networks that require targets. Xi is also optional and need only be used for networks that have input or layer delays.

train arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

8.8 USING COMMAND-LINE FUNCTIONS

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

You can solve this problem in two ways:

- Use a graphical user interface, [ntstool](#).
- Use command-line functions.

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox. If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

To define a time-series problem for the toolbox, arrange a set of TS input vectors as columns in a cell array. Then, arrange another set of TS target vectors

(the correct output vectors for each of the input vectors) into a second cell array. However, there are cases in which you only need to have a target data set. For example, you can define the following time-series problem, in which you want to use previous values of a series to predict the next value:

targets = {1 2 3 4 5};

The next section shows how to train a network to fit a time-series data set, using the neural network time-series functions.

```
% Solve an Autoregression Problem  
with External  
% Input with a NARX Neural Network
```

```
% Script generated by NTSTOOL
%
% This script assumes the variables on
the right of
% these equalities are defined:
%
% phInputs - input time series.
% phTargets - feedback time series.
```

```
inputSeries = phInputs;
targetSeries = phTargets;
```

```
% Create a Nonlinear Autoregressive
Network with External Input
inputDelays = 1:4;
feedbackDelays = 1:4;
hiddenLayerSize = 10;
```

```
net =  
narxnet(inputDelays,feedbackDelays,hiddenNodes)
```

% Prepare the Data for Training and
Simulation

% The function PREPARETS prepares
time series data

% for a particular network, shifting time
by the minimum

% amount to fill input states and layer
states.

% Using PREPARETS allows you to
keep your original

% time series data unchanged, while
easily customizing it

% for networks with differing numbers
of delays, with

% open loop or closed loop feedback modes.

[inputs,inputStates,layerStates,targets] =

...

 prepares(net,inputSeries,
 {},targetSeries);

% Set up Division of Data for Training,
Validation, Testing

net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

net.divideParam.testRatio = 15/100;

% Train the Network

[net,tr] =

train(net,inputs,targets,inputStates,layerS

```
% Test the Network  
outputs =  
net(inputs,inputStates,layerStates);  
errors = gsubtract(targets,outputs);  
performance =  
perform(net,targets,outputs)  
  
% View the Network  
view(net)  
  
% Plots  
% Uncomment these lines to enable  
various plots.  
% figure, plotperform(tr)  
% figure, plottrainstate(tr)  
% figure, plotregression(targets,outputs)  
% figure, plotresponse(targets,outputs)
```

```
% figure, ploterrcorr(errors)
```

```
% figure, plotinerrcorr(inputs,errors)
```

8.9

```
% Closed Loop Network
% Use this network to do multi-step
prediction.
% The function CLOSELOOP replaces
the feedback input with a direct
% connection from the output layer.
netc = closeloop(net);
netc.name = [net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] =
prepares(netc,inputSeries,
{},targetSeries);
yc = netc(xc,xic,aic);
closedLoopPerformance =
perform(netc,tc,yc)
```

% Early Prediction Network
% For some applications it helps to get
the prediction a
% timestep early.
% The original network returns
predicted $y(t+1)$ at the same
% time it is given $y(t+1)$.
% For some applications such as
decision making, it would
% help to have predicted $y(t+1)$ once
 $y(t)$ is available, but
% before the actual $y(t+1)$ occurs.
% The network can be made to return its
output a timestep early
% by removing one delay so that its
minimal tap delay is now
% 0 instead of 1. The new network

returns the same outputs as
% the original network, but outputs are
shifted left one timestep.

```
nets = removedelay(net);
```

```
nets.name = [net.name ' - Predict One  
Step Ahead'];
```

```
view(nets)
```

```
[xs,xis,ais,ts] =
```

```
preparets(nets,inputSeries,  
{},targetSeries);
```

```
ys = nets(xs,xis,ais);
```

```
earlyPredictPerformance =
```

```
perform(nets,ts,ys)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the

training process. In this case, follow each of the steps in the script.

The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
load ph_dataset  
inputSeries = phInputs;  
targetSeries = phTargets;
```

Create a network. The NARX network, [narxnet](#), is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This network has two inputs. One is an external input, and the other is a

feedback connection from the network output. (After the network has been trained, this feedback connection can be closed, as you will see at a later step.) For each of these inputs, there is a tapped delay line to store previous values. To assign the network architecture for a NARX network, you must select the delays associated with each tapped delay line, and also the number of hidden layer neurons. In the following steps, you assign the input delays and the feedback delays to range from 1 to 4 and the number of hidden neurons to be 10.

```
inputDelays = 1:4;  
feedbackDelays = 1:4;
```

```
hiddenLayerSize = 10;  
net =  
narxnet(inputDelays,feedbackDelays,hiddenLayerSize)
```

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

Prepare the data for training. When training a network containing tapped delay lines, it is necessary to fill the delays with initial values of the inputs and outputs of the network. There is a toolbox command that facilitates this process - [preparet](#)s. This function has three input arguments: the network, the input sequence and the target sequence. The function returns the initial conditions that are needed to fill the tapped delay lines in the network, and modified input and target sequences, where the initial conditions have been removed. You can call the function as follows:

```
[inputs,inputStates,layerStates,targets] =
```

...

```
prepares(net,inputSeries,  
{},targetSeries);
```

Set up the division of data.

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

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

Train the network. The network uses the default Levenberg-Marquardt algorithm ([trainlm](#)) for training. For problems in which Levenberg-Marquardt does not produce as accurate results as desired,

or for large data problems, consider setting the network training function to Bayesian Regularization ([trainbr](#)) or Scaled Conjugate Gradient ([trainscg](#)), respectively, with either

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

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

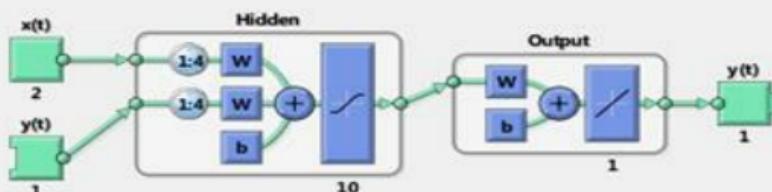
To train the network, enter:

```
[net,tr] =
```

```
train(net,inputs,targets,inputStates,layerS
```

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.

Neural Network



Algorithms

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

Progress

Epoch:	0	44 iterations	1000
Time:		0:00:00	
Performance:	79.1	0.00219	0.00
Gradient:	134	0.0187	1.00e-07
Mu:	0.00100	1.00e-05	1.00e+10
Validation Checks:	0	6	6

Plots

- Performance** (plotperform)
- Training State** (plottrainstate)
- Error Histogram** (ploterrhist)
- Regression** (plotregression)
- Time-Series Response** (plotresponse)
- Error Autocorrelation** (ploterrcorr)
- Input-Error Cross-correlation** (plotinerrcorr)

Plot Interval: 1 epochs



Validation stop.

Stop Training

Cancel

This training stopped when the validation error increased for six iterations, which occurred at iteration 44.

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

`outputs =`

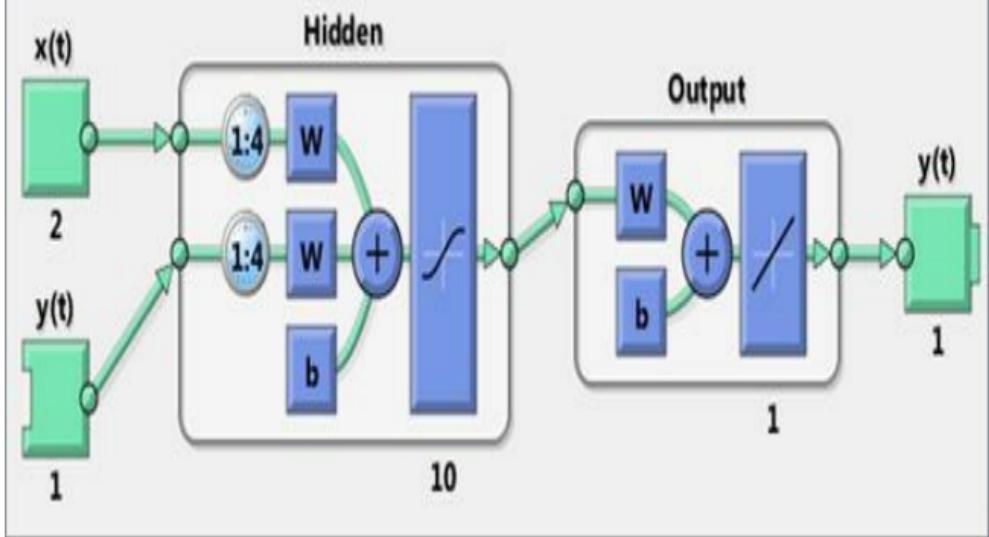
```
net(inputs,inputStates,layerStates);  
errors = gsubtract(targets,outputs);  
performance =  
perform(net,targets,outputs)
```

performance =

0.0042

View the network diagram.

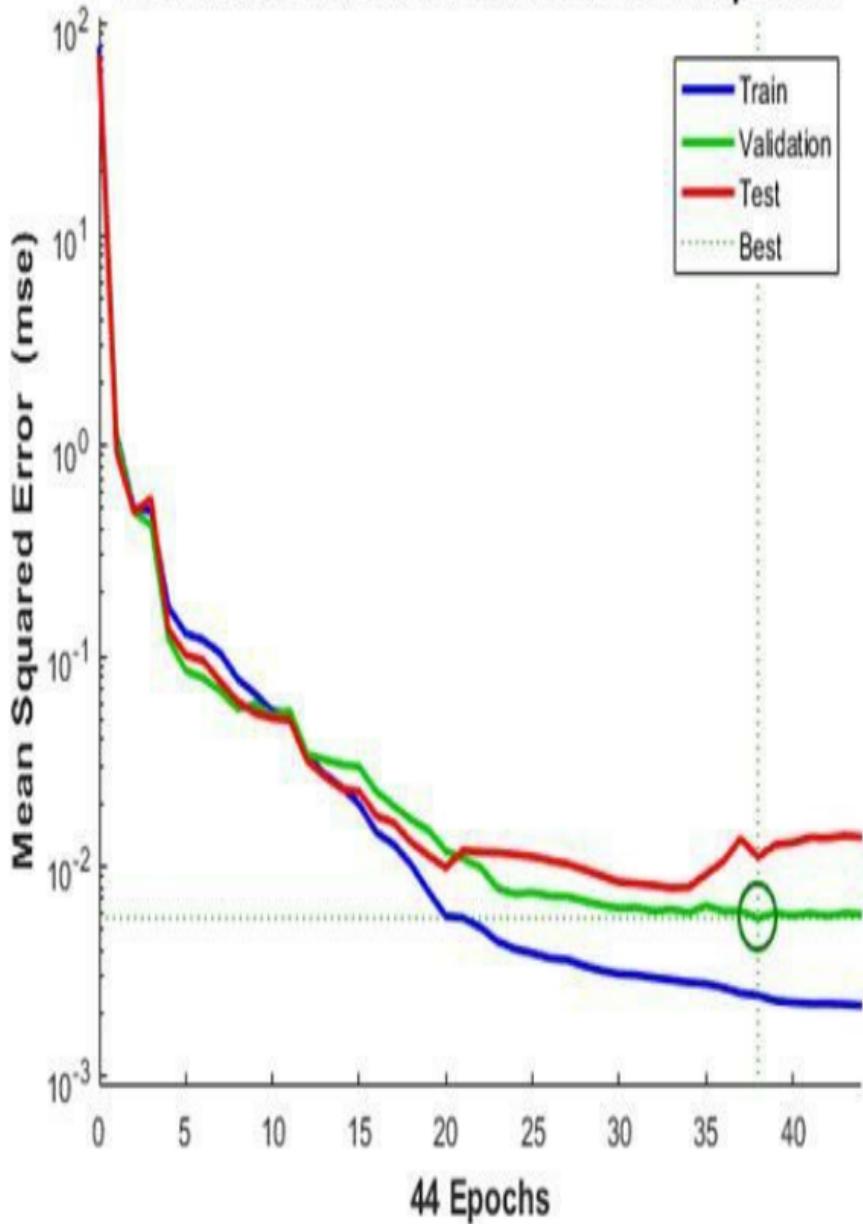
```
view(net)
```



Plot the performance training record to check for potential overfitting.

`figure, plotperform(tr)`

Best Validation Performance is 0.0056917 at epoch 38



This figure shows that training, validation and testing errors all decreased until iteration 64. It does not appear that any overfitting has occurred, because neither testing nor validation error increased before iteration 64.

All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) is it transformed to closed loop for multistep-ahead prediction.

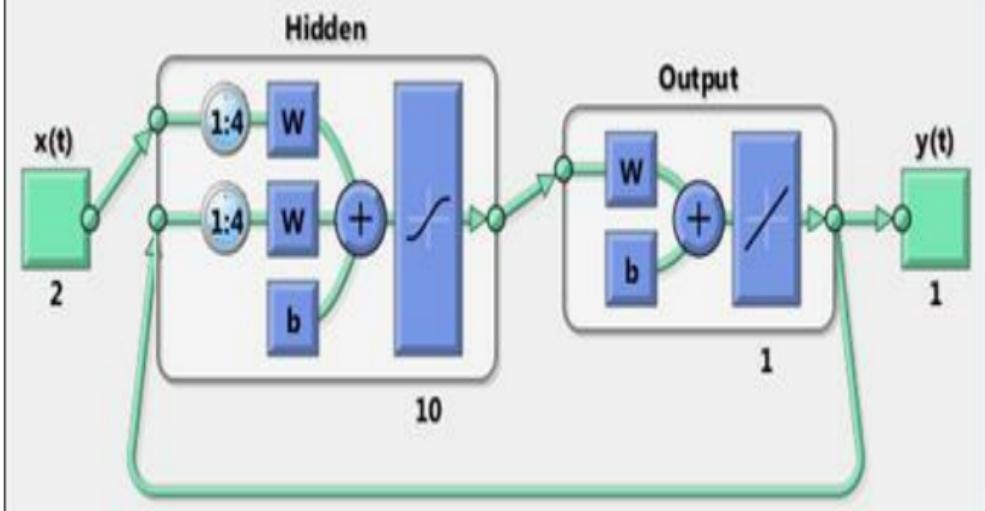
Likewise, the Rvalues in the GUI are computed based on the open-loop training results.

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

```
netc = closeloop(net);
netc.name = [ net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] =
prepares(netc,inputSeries,
{},targetSeries);
yc = netc(xc,xic,aic);
perf = perform(netc,tc,yc)
```

perf =

2.8744



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

```

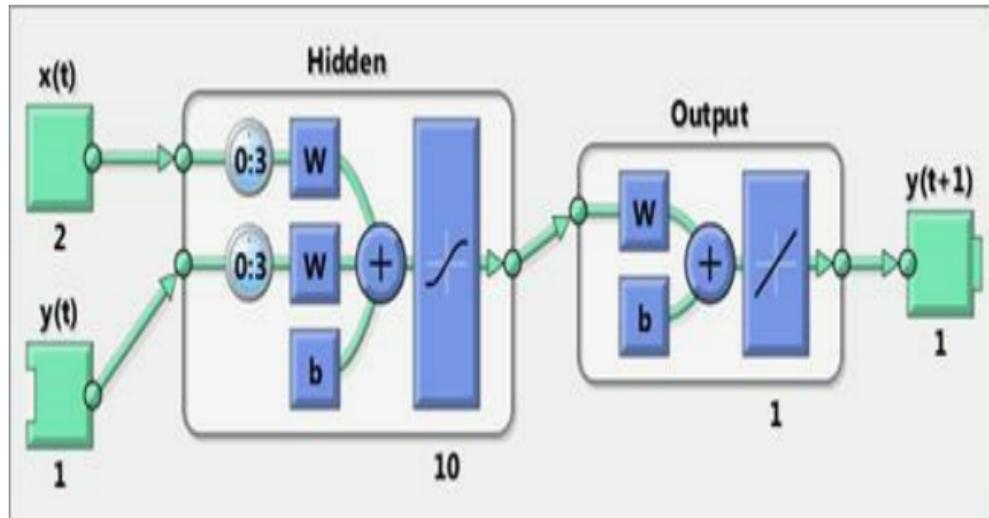
nets = removedelay(net);
nets.name = [net.name ' - Predict One
Step Ahead'];
view(nets)
[xs,xis,ais,ts] =
prepares(nets,inputSeries,

```

```
{},targetSeries);  
ys = nets(xs,xis,ais);  
earlyPredictPerformance =  
perform(nets,ts,ys)
```

earlyPredictPerformance =

0.0042



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

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

- Reset the initial network weights and biases to new values with [init](#) and train again

- Increase the number of hidden neurons or the number of delays.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm.

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

- During training, open a plot window (such as the error correlation plot), and watch it

animate.

- Plot from the command line with functions such as plotresponse, ploterrcorr and plotperform.

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

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

a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired.

8.10 A COMPLTE EXAMPLE. MAGLEV MODELING

This example illustrates how a NARX
(Nonlinear AutoRegressive with
eXternal input) neural network can
model a magnet levitation dynamical
system.

8.10.1 The Problem: Model a Magnetic Levitation System

In this example we attempt to build a neural network that can predict the dynamic behavior of a magnet levitated using a control current.

The system is characterized by the magnet's position and a control current, both of which determine where the magnet will be an instant later.

This is an example of a time series problem, where past values of a feedback time series (the magnet position) and an external input series

(the control current) are used to predict future values of the feedback series.

8.10.2 Why Neural Networks?

Neural networks are very good at time series problems. A neural network with enough elements (called neurons) can model dynamic systems with arbitrary accuracy. They are particularly well suited for addressing non-linear dynamic problems. Neural networks are a good candidate for solving this problem.

The network will be designed by using recordings of an actual levitated magnet's position responding to a control current.

8.10.3 Preparing the Data

Data for function fitting problems are set up for a neural network by organizing the data into two matrices, the input time series X and the target time series T.

The input series X is a row cell array, where each element is the associated timestep of the control current.

The target series T is a row cell array, where each element is the associated timestep of the levitated magnets position.

Here such a dataset is loaded.

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

We can view the sizes of inputs X and targets T.

Note that both X and T have 4001 columns. These represent 4001 timesteps of the control current and magnet position.

`size(x)`

`size(t)`

`ans =`

1 4001

`ans =`

1 4001

8.10.4 Time Series Modelling with a Neural Network

The next step is to create a neural network that will learn to model how the magnet changes position.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

```
setdemorandstream(491218381)
```

Two-layer (i.e. one-hidden-layer) NARX neural networks can fit any dynamical input-output relationship

given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

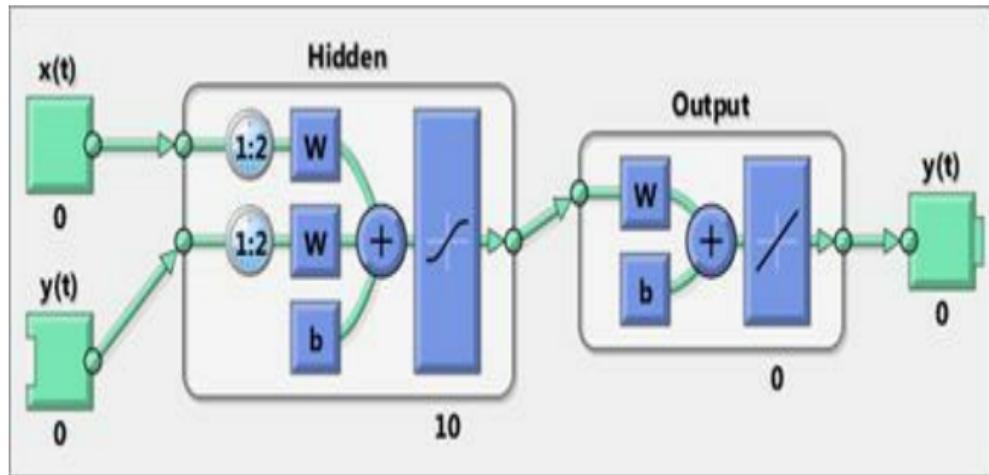
We will also try using tap delays with two delays for the external input (control current) and feedback (magnet position). More delays allow the network to model more complex dynamic systems.

The input and output have sizes of 0 because the network has not yet been

configured to match our input and target data. This will happen when the network is trained.

The output $y(t)$ is also an input, whose delayed v

```
net = narxnet(1:2,1:2,10);  
view(net)
```



Before we can train the network, we must use the first two timesteps of the

external input and feedback time series to fill the two tap delay states of the network.

Furthermore, we need to use the feedback series both as an input series and target series.

The function PREPARETS prepares time series data for simulation and training for us. Xs will consist of shifted input and target series to be presented to the network. Xi is the initial input delay states. Ai is the layer delay states (empty in this case as there are no layer-to-layer delays), and Ts is the shifted feedback series.

[Xs,Xi,Ai,Ts] = prepares(net,x,{},t);

Now the network is ready to be trained. The timesteps are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

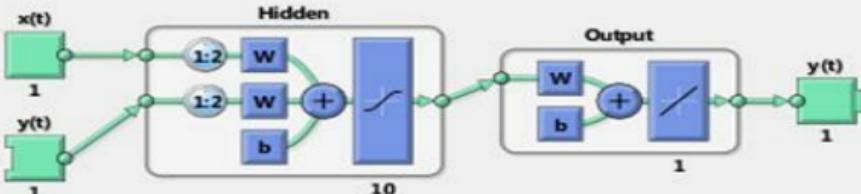
The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and

after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,Xs,Ts,Xi,Ai);  
nntraintool
```

Neural Network



Algorithms

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

Progress

Epoch:	0	198 iterations	1000
Time:		0:00:05	
Performance:	36.3	4.75e-07	0.00
Gradient:	73.2	2.96e-05	1.00e-07
Mu:	0.00100	1.00e-07	1.00e+10
Validation Checks:	0	6	6

Plots

- Performance** (plotperform)
- Training State (plottrainstate)
- Error Histogram (ploterrhist)
- Regression (plotregression)
- Time-Series Response (plotresponse)
- Error Autocorrelation (ploterrcorr)
- Input-Error Cross-correlation (plotinerrcorr)

Plot Interval:



Validation stop.

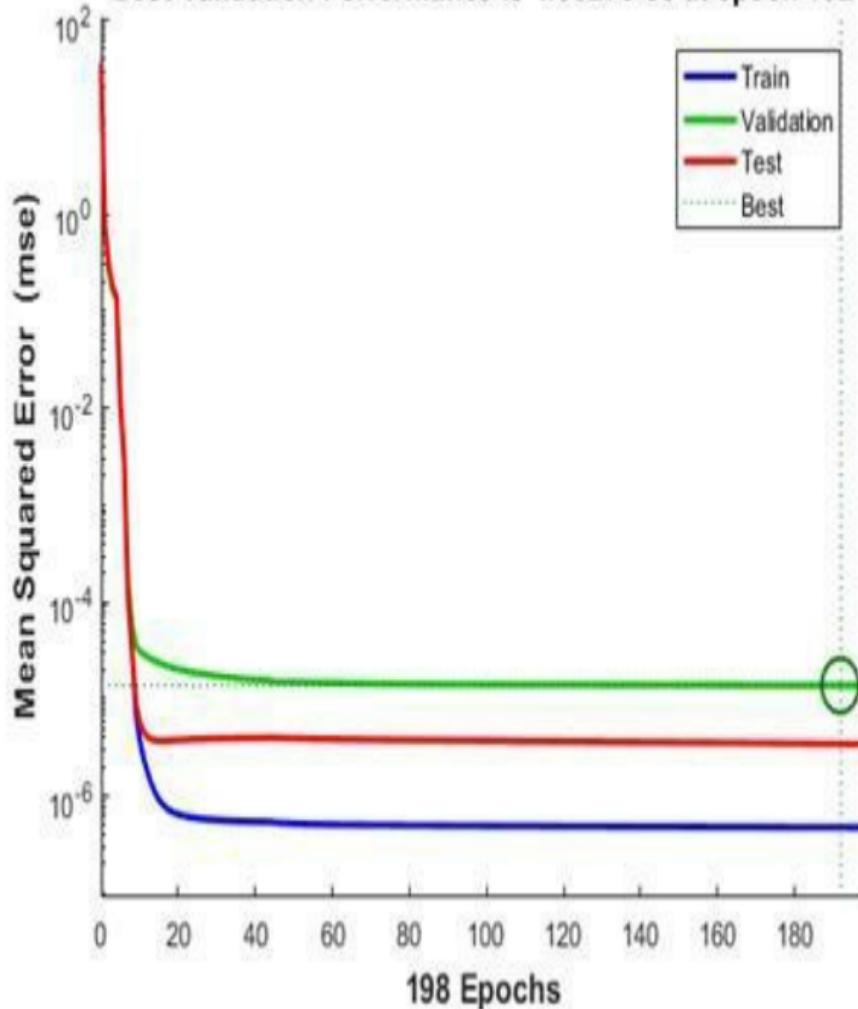
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

`plotperform(tr)`

Best Validation Performance is 1.3827e-05 at epoch 192



8.10.5 Testing the Neural Network

The mean squared error of the trained neural network for all timesteps can now be measured.

```
Y = net(Xs,Xi,Ai);
```

```
perf = mse(net,Ts,Y)
```

```
perf =
```

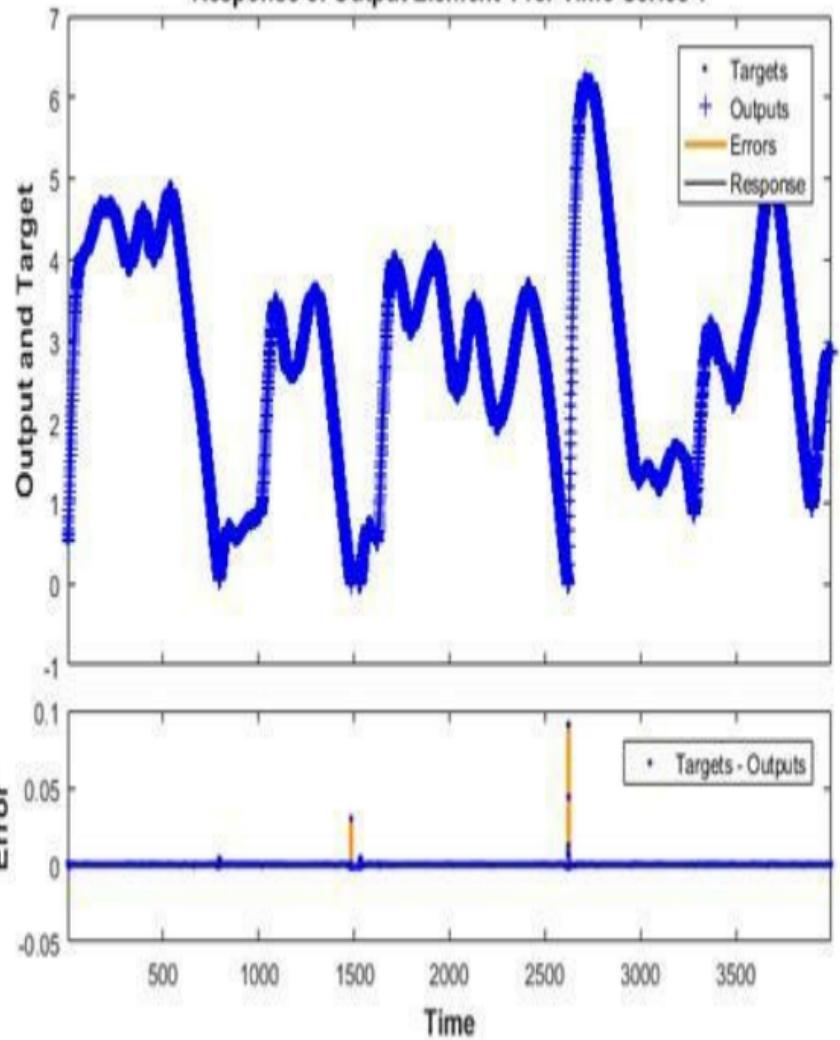
2.9245e-06

PLOTRESPONSE will show us the network's response in comparison to the actual magnet position. If the model is

accurate the '+' points will track the diamond points, and the errors in the bottom axis will be very small.

plotresponse(Ts,Y)

Response of Output Element 1 for Time-Series 1



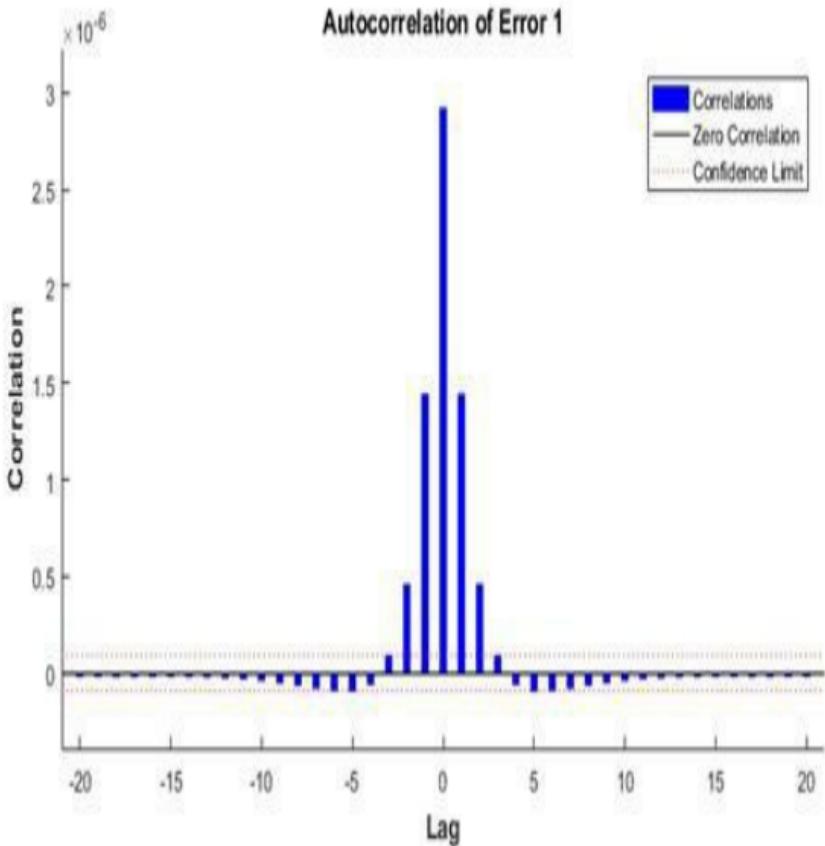
PLOTERRCORR shows the correlation of error at time t , $e(t)$ with errors over

varying lags, $e(t+lag)$. The center line shows the mean squared error. If the network has been trained well all the other lines will be much shorter, and most if not all will fall within the red confidence limits.

The function GSUBTRACT is used to calculate the error. This function generalizes subtraction to support differences between cell array data.

```
E = gsubtract(Ts,Y);
```

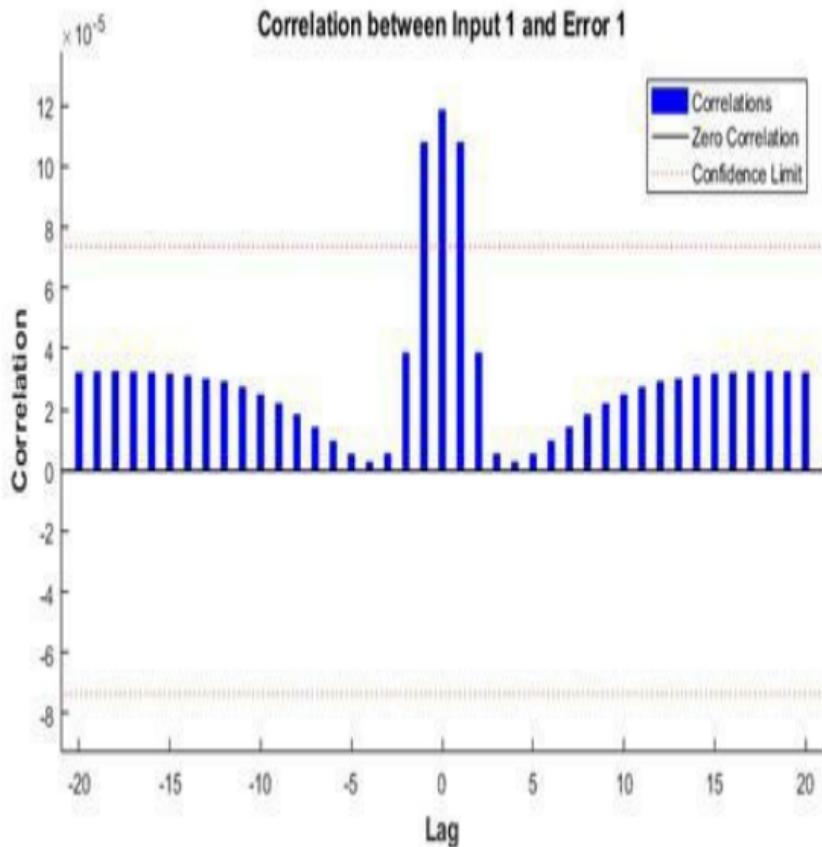
```
ploterrcorr(E)
```



Similarly, PLOTINERRCORR shows the correlation of error with respect to the inputs, with varying degrees of lag. In this case, most or all the lines should fall within the confidence limits,

including the center line.

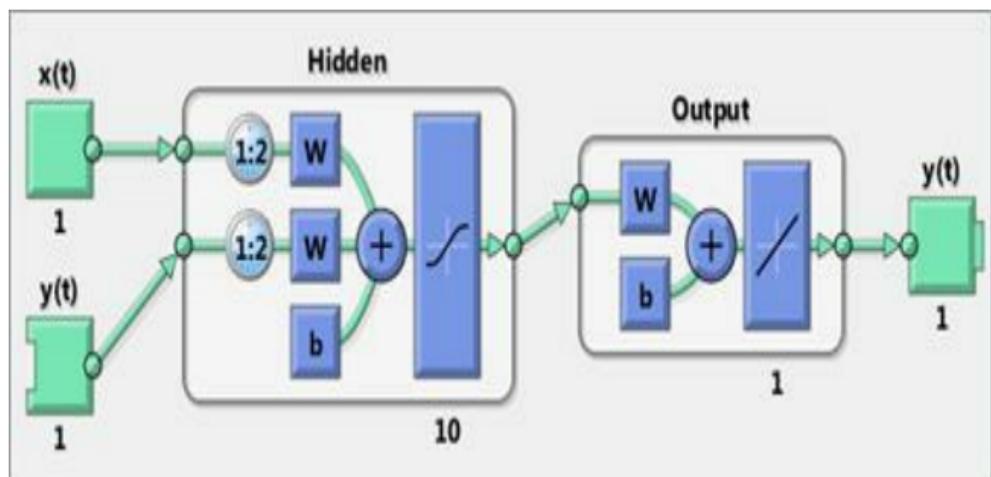
plotinerrcorr(Xs,E)



The network was trained in open loop form, where targets were used as

feedback inputs. The network can also be converted to closed loop form, where its own predictions become the feedback inputs.

```
net2 = closeloop(net);  
view(net)
```



We can simulate the network in closed loop form. In this case the network is only given initial magnet positions, and then must use its own predicted

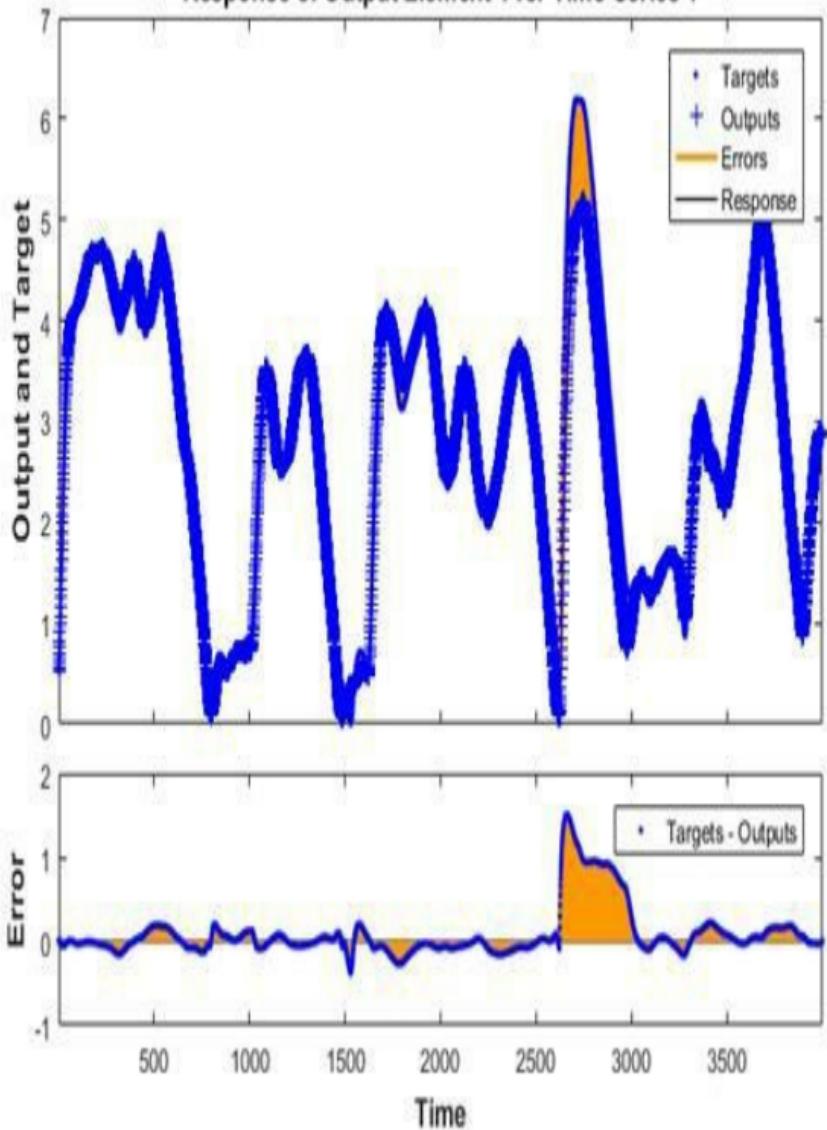
positions recursively to predict new positions.

This quickly results in a poor fit between the predicted and actual response. This will occur even if the model is very good. But it is interesting to see how many steps they match before separating.

Again, PREPARETS does the work of preparing the time series data for us taking into account the altered network.

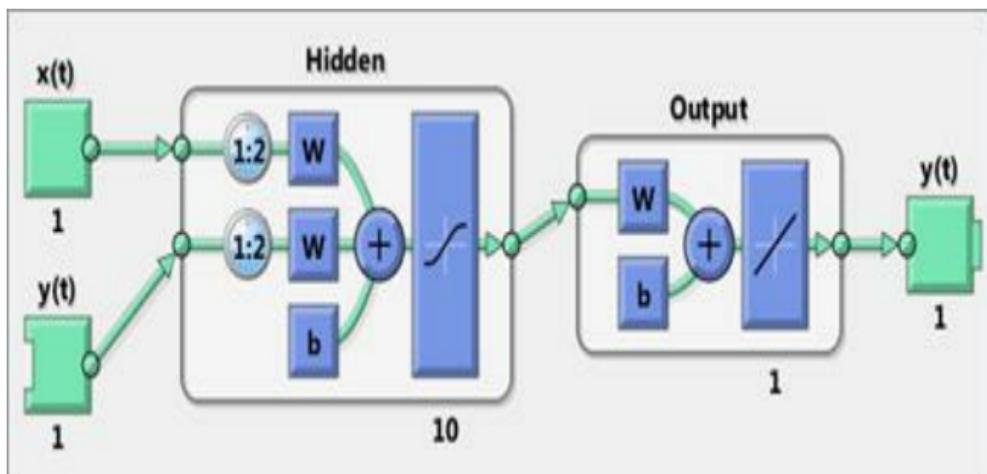
```
[Xs,Xi,Ai,Ts] = preparets(net2,x,{},t);  
Y = net2(Xs,Xi,Ai);  
plotresponse(Ts,Y)
```

Response of Output Element 1 for Time-Series 1



If the application required us to access the predicted magnet position a timestep ahead of when it actually occur, we can remove a delay from the network so at any given time t , the output is an estimate of the position at time $t+1$.

```
net3 = removedelay(net);  
view(net)
```

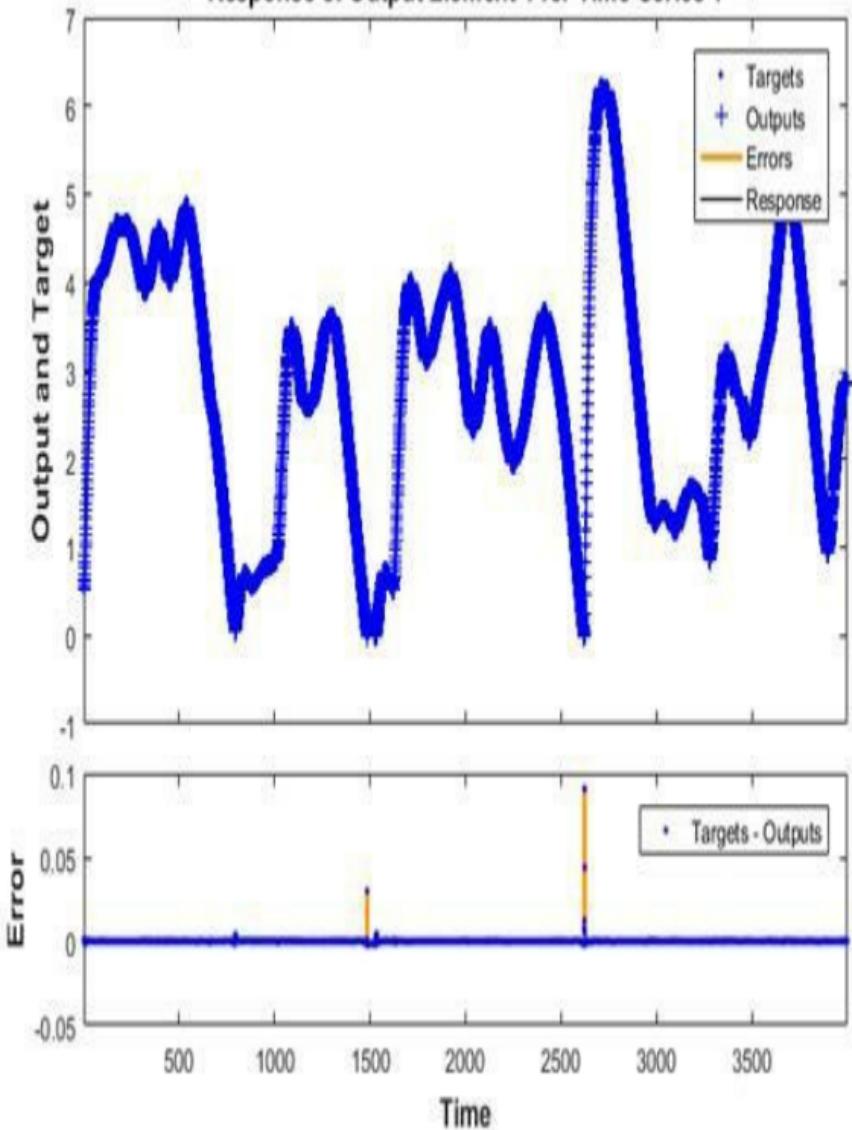


Again we use PREPARETS to prepare the time series for simulation. This time

the network is again very accurate as it is doing open loop prediction, but the output is shifted one timestep.

```
[Xs,Xi,Ai,Ts] = preparets(net3,x,{},t);  
Y = net3(Xs,Xi,Ai);  
plotresponse(Ts,Y)
```

Response of Output Element 1 for Time-Series 1



This example illustrated how to design a neural network that models the behavior of a dynamical magnet levitation system.

Chapter 9

**FIT DATA WITH A NEURAL
NETWORK. GRAPHICAL
INTERFACE**

9.1 INTRODUCTION

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

Suppose, for instance, that you have data from a housing application. 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 two ways:

- Use a graphical user interface, *nftool*, as described in [Using the Neural Network Fitting Tool](#).

- Use command-line functions, as described in [Using Command-Line Functions](#).

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox (see [Neural Network Toolbox Sample Data Sets](#)). If

you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

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 (see "[Data Structures](#)" for a detailed description of data formatting for static and time-series data). 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 shows how to train a network to fit a data set, using the neural network fitting tool GUI, [*nftool*](#). This example uses the housing data set provided with the toolbox.

9.2 USING THE NEURAL NETWORK FITTING TOOL

- Open the Neural Network Start GUI with this command: *nnstart*



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Getting Started Wizards

More Information

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

Input-output and curve fitting.



Fitting app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npctool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

- Click *Fitting Tool* to open the Neural Network Fitting Tool. (You can also use the command [nftool](#).)



Welcome to the Neural Fitting app.

Solve an input-output fitting problem with a two-layer feed-forward neural network.

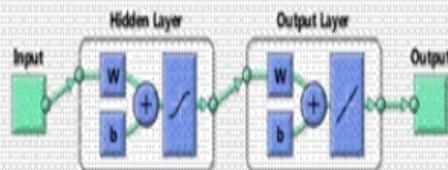
Introduction

In fitting problems, you want a neural network to map between a data set of numeric inputs and a set of numeric targets.

Examples of this type of problem include estimating house prices from such input variables as tax rate, pupil/teacher ratio in local schools and crime rate ([house dataset](#)), estimating engine emission levels based on measurements of fuel consumption and speed ([engine dataset](#)), or predicting a patient's bodyfat level based on body measurements ([bodyfat dataset](#)).

The Neural Fitting app will help you select data, create and train a network, and evaluate its performance using mean square error and regression analysis.

Neural Network



A two-layer feed-forward network with sigmoid hidden neurons and linear output neurons ([neuron](#)), can fit multi-dimensional mapping problems arbitrarily well, given consistent data and enough neurons in its hidden layer.

The network will be trained with Levenberg-Marquardt backpropagation algorithm ([trainlm](#)), unless there is not enough memory, in which case scaled conjugate gradient backpropagation ([trainscg](#)) will be used.

To continue, click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

- Click *Next* to proceed.

Select Data

What inputs and targets define your fitting problem?

Get Data from Workspace

Input data to present to the network.

Inputs:

(none)

Target data defining desired network output.

Targets:

(none)

Samples are

Matrix columns Matrix rows

Want to try out this tool with an example data set?

Summary

No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

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

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



Select a data set:

Simple Fitting Problem

House Pricing

Abalone Rings

Body Fat

Building Energy

Chemical

Engine

Description

Filename: chemical dataset

Function fitting is the process of training a neural network on a set of inputs in order to produce an associated set of target outputs. Once the neural network has fit the data, it forms a generalization of the input-output relationship and can be used to generate outputs for inputs it was not trained on.

This dataset can be used to train a neural network to estimate one sensor signal from eight other sensor signals.

LOAD chemical dataset.MAT loads these two variables:

chemicalInputs - a 8x498 matrix defining measurements taken from eight sensors during a chemical process.

chemicalTargets - a 1x498 matrix of a ninth sensor's measurements, to be estimated from the first eight.

A good estimator for the ninth sensor will allow it to be removed and estimations used in its place.

Import

Cancel

- Select **Chemical**, and click **Import**. This returns you to the Select Data window.
- Click **Next** to display the Validation and Test Data window, shown in the following figure.

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



Validation and Test Data

Set aside some samples for validation and testing.

Select Percentages

- Randomly divide up the 498 samples:

Training:	70%	348 samples
Validation:	15% ▾	75 samples
Testing:	15% ▾	75 samples

Explanation

- Three Kinds of Samples:

- Training:

These are presented to the network during training, and the network is adjusted according to its error.

- Validation:

These are used to measure network generalization, and to halt training when generalization stops improving.

- Testing:

These have no effect on training and so provide an independent measure of network performance during and after training.

Restore Defaults

Change percentages if desired, then click [Next] to continue.

Neural Network Start

Welcome

Back

Next

Cancel

With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

1. 70% will be used for training.
2. 15% will be used to validate that the network is generalizing and to stop training before overfitting.
3. The last 15% will be used as a completely independent test of network generalization.

Click **Next**.

The standard network that is used for function fitting is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. The default number of hidden neurons is set to 10. You might want to increase this number later, if the network training performance is poor.



Network Architecture

Set the number of neurons in the fitting network's hidden layer.

Hidden Layer

Define a fitting neural network... (fitnet)

Number of Hidden Neurons:

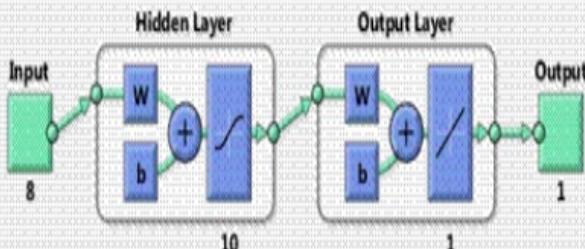
10

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

[Restore Defaults](#)

Neural Network



Change settings if desired, then click [Next] to continue.

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

Click Next.

Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm:

Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt... (trainlm)

Train

Samples

MSE

R

Results

Training:

348

Validation:

75

Testing:

75

Plot Fit

Plot Error Histogram

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].

Neural Network Start

Welcome

Back

Next

Cancel

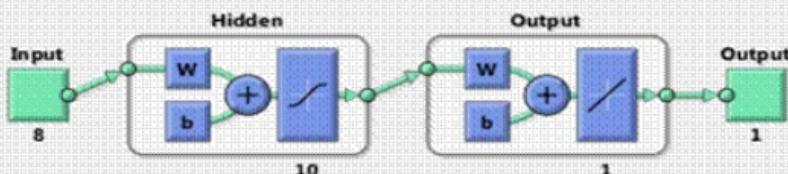
Select a training algorithm, then click **Train..**. Levenberg-Marquardt (`trainlm`) is recommended for most problems, but for some noisy and small problems Bayesian Regularization (`trainbr`) can take longer but obtain a better solution. For large problems, however, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use. This example uses the default Levenberg-Marquardt.

The training continued until the

validation error failed to decrease for six iterations (validation stop).

Neural Network Training (nntraintool)

Neural Network



Algorithms

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

Progress

Epoch:	0	19 iterations	1000
Time:		0:00:00	
Performance:	908	2.34	0.00
Gradient:	$3.07e+03$	9.55	$1.00e-07$
Mu:	0.00100	0.00100	$1.00e+10$
Validation Checks:	0	6	6

Plots

Performance

(plotperform)

Training State

(plottrainstate)

Error Histogram

(ploterrhist)

Regression

(plotregression)

Fit

(plotfit)

Plot Interval:

1 epochs



Validation stop.



Stop Training



Cancel

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

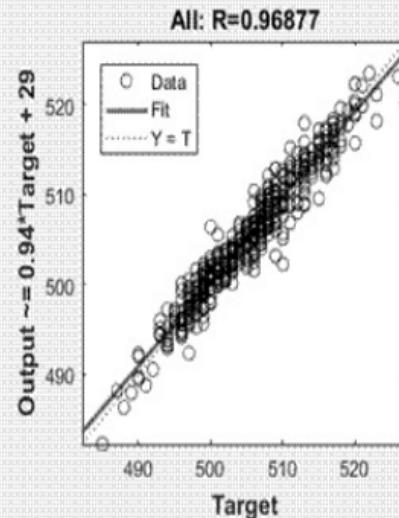
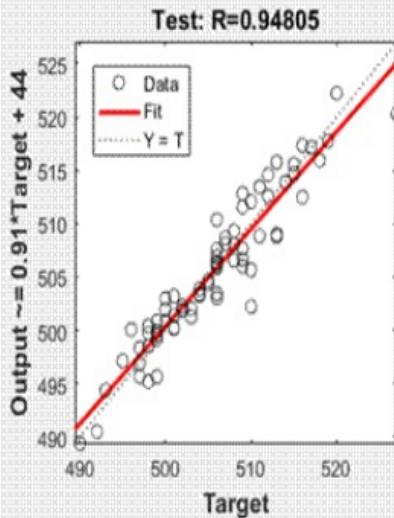
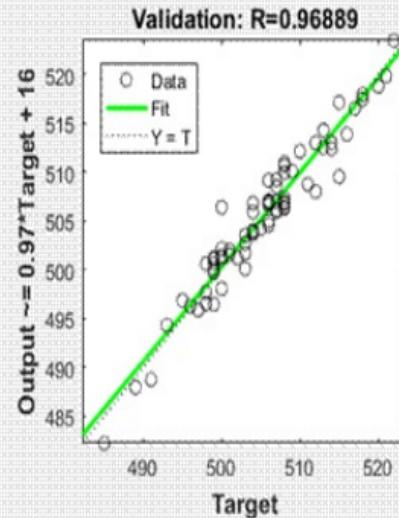
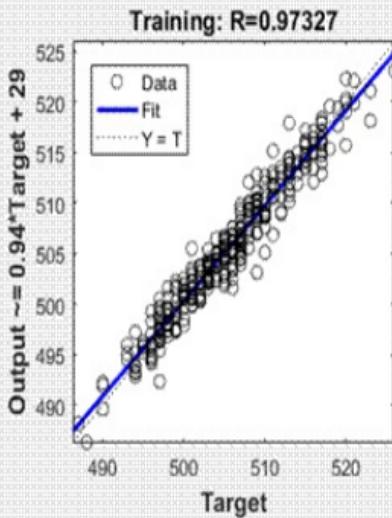
This is used to validate the network performance.

The following regression plots display the network outputs with respect to targets for training, validation, and test sets. For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the targets. For this problem, the fit is reasonably good for all data sets, with R values in each case of 0.93 or above. If even more accurate results were required, you could retrain the network by clicking **Retrain** in

[nftool](#). This will change the initial weights and biases of the network, and may produce an improved network after retraining. Other options are provided on the following pane.

Regression (plotregression)

File Edit View Insert Tools Desktop Window Help

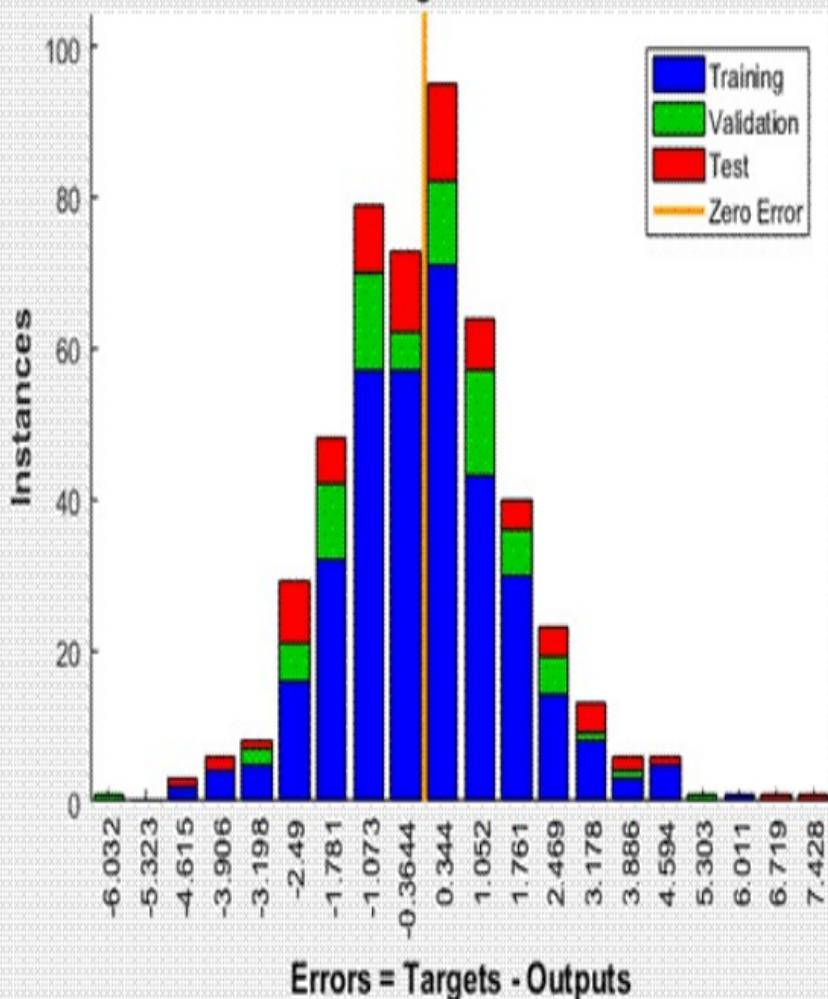


View the error histogram to obtain additional verification of network performance. Under the **Plots** pane, click **Error Histogram**.

Error Histogram (ploterrhist)

File Edit View Insert Tools Desktop Window Help

Error Histogram with 20 Bins



The blue bars represent training data, the green bars represent validation data, and the red bars represent testing data. The histogram can give you an indication of outliers, which are data points where the fit is significantly worse than the majority of data. In this case, you can see that while most errors fall between -5 and 5, there is a training point with an error of 17 and validation points with errors of 12 and 13. These outliers are also visible on the testing regression plot. The first corresponds to the point with a target of 50 and output near 33. It is a

good idea to check the outliers to determine if the data is bad, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points. You should collect more data that looks like the outlier points, and retrain the network.

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



Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

Try training again if a first try did not generate good results
or you require marginal improvement.

Train Again

Increase network size if retraining did not help.

Adjust Network Size

Not working? You may need to use a larger data set.

Import Larger Data Set

Select inputs and targets, click an improvement button, or click [Next].

Optionally perform additional tests

Inputs:

(none)



Targets:

(none)



Samples are:

Matrix columns

Matrix rows

No inputs selected.

No targets selected.

MSE

R

Test Network

Plot Fit

Plot Error Histogram

Plot Regression

Neural Network Start

Welcome

Back

Next

Cancel

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 do one of the following:

1. Train it again.
2. Increase the number of neurons.
3. Get a larger training data set.

If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the number of neurons can improve your results. If training performance is poor,

then you may want to increase the number of neurons.

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

Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs, or deploy the network with MATLAB Compiler tools and other MATLAB code generation tools.



Deploy Solution

Generate deployable versions of your trained neural network.

Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

(gensim)



Graphics

Generate a graphical diagram of the neural network:

(network/view)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

Use the buttons on this screen to generate scripts or to save your results.

Save Results



Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool



Generate a script with additional options and example code:



Save Data to Workspace

Save network to MATLAB network object named:

net

Save performance and data set information to MATLAB struct named:

info

Save outputs to MATLAB matrix named:

output

Save errors to MATLAB matrix named:

error

Save inputs to MATLAB matrix named:

input

Save targets to MATLAB matrix named:

target

Save ALL selected values above to MATLAB struct named:

results

Restore Defaults



Save results and click [Finish].

Neural Network Start

Welcome

Back

Next

Finish

You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

You can also have the network saved as net in the workspace. You can perform additional tests on it or put it to work on new inputs.

When you have created the MATLAB code and saved your results, click **Finish**.

9.3 USING COMMAND-LINE FUNCTIONS

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created at step 14 of the previous section.

```
% Solve an Input-Output Fitting problem  
with a Neural Network  
  
% Script generated by NFTOOL  
  
%  
  
% This script assumes these variables
```

are defined:

%

% houseInputs - input data.

% houseTargets - target data.

inputs = houseInputs;

targets = houseTargets;

% Create a Fitting Network

hiddenLayerSize = 10;

net = fitnet(hiddenLayerSize);

% Set up Division of Data for Training,
Validation, Testing

net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

net.divideParam.testRatio = 15/100;

% Train the Network

[net,tr] = train(net,inputs,targets);

% Test the Network

outputs = net(inputs);

errors = gsubtract(outputs,targets);

performance =

```
perform(net,targets,outputs)
```

```
% View the Network
```

```
view(net)
```

```
% Plots
```

```
% Uncomment these lines to enable  
various plots.
```

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

```
% figure, plottrainstate(tr)
```

```
% figure, plotfit(targets,outputs)
```

```
% figure, plotregression(targets,outputs)
```

```
% figure, ploterrhist(errors)
```

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

The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

1. `load house_dataset`
2. `inputs = houseInputs;`
3. `targets = houseTargets;`

This data set is one of the sample data

sets that is part of the toolbox (see [Neural Network Toolbox Sample Data Sets](#)). You can see a list of all available data sets by entering the command help nndatasets. The load command also allows you to load the variables from any of these data sets using your own variable names. For example, the command

```
[inputs,targets] = house_dataset;
```

will load the housing inputs into the array inputs and the housing targets into the array targets.

Create a network. The default network for function fitting (or regression) problems, [fitnet](#), is a feedforward

network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. You assigned ten neurons (somewhat arbitrary) to the one hidden layer in the previous section. The network has one output neuron, because there is only one target value associated with each input vector.

```
hiddenLayerSize = 10;
```

```
net = fitnet(hiddenLayerSize);
```

Note More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More

layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the [fitnet](#) command.

Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
```

```
net.divideParam.valRatio = 15/100;
```

```
net.divideParam.testRatio = 15/100;
```

With these settings, the input vectors and target vectors will be randomly divided, with 70% used for training, 15%

validation and 15% for testing.

Train the network. The network uses the default Levenberg-Marquardt algorithm ([trainlm](#)) for training. For problems in which Levenberg-Marquardt does not produce as accurate results as desired, or for large data problems, consider setting the network training function to Bayesian Regularization ([trainbr](#)) or Scaled Conjugate Gradient ([trainscg](#)), respectively, with either

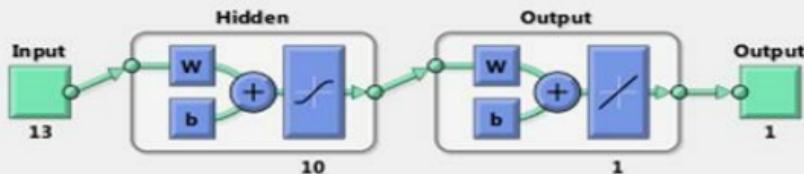
```
net.trainFcn = 'trainbr';
```

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

To train the network, enter:

```
[net,tr] = train(net,inputs,targets);
```

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**.

Neural Network**Algorithms**

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

Progress

Epoch:	0	20 iterations	1000
Time:		0:00:00	
Performance:	194	2.97	0.00
Gradient:	914	5.59	1.00e-07
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

Plots

- Performance** (plotperform)
- Training State** (plottrainstate)
- Error Histogram** (ploterrhist)
- Regression** (plotregression)
- Fit** (plotfit)

Plot Interval: 1 epochs

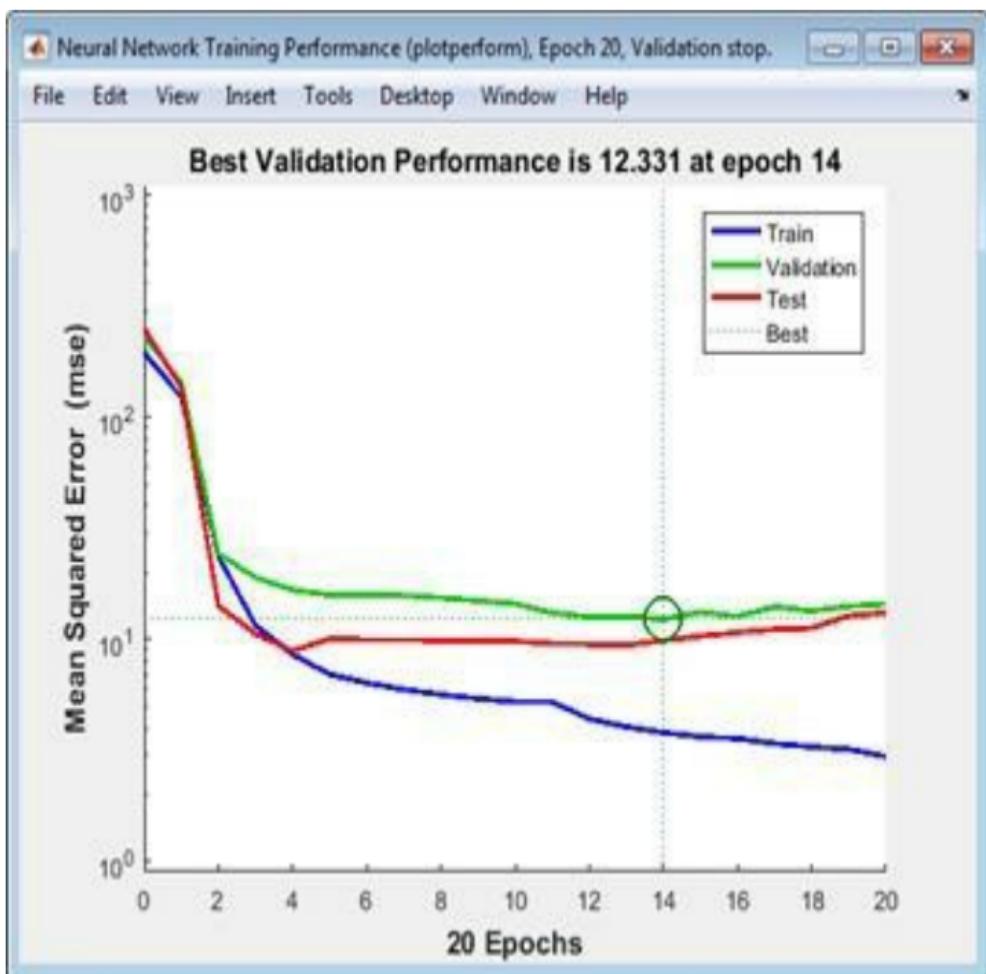


Opening Regression Plot

This training stopped when the validation error increased for six iterations, which occurred at iteration 20. 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:

1. The final mean-square error is small.
2. The test set error and the validation set error have similar characteristics.
3. No significant overfitting has

occurred by iteration 14 (where the best validation performance occurs).



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

```
outputs = net(inputs);
```

```
errors = gsubtract(targets,outputs);
```

```
performance =
```

```
perform(net,targets,outputs)
```

```
performance =
```

6.0023

It is also possible to calculate the network performance only on the test set,

by using the testing indices, which are located in the training record.

```
tInd = tr.testInd;
```

```
tstOutputs = net(inputs(:,tInd));
```

```
tstPerform =
```

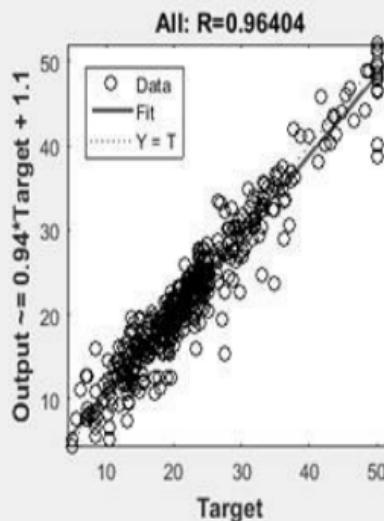
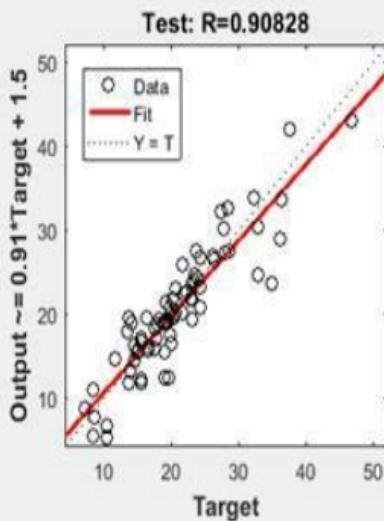
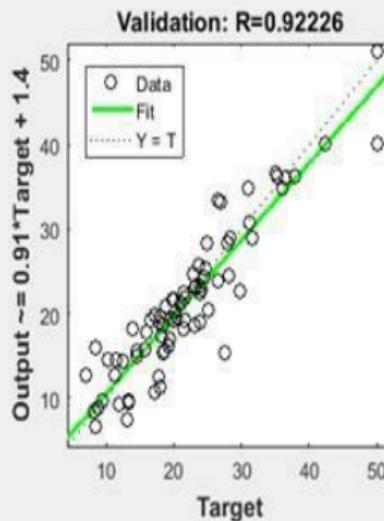
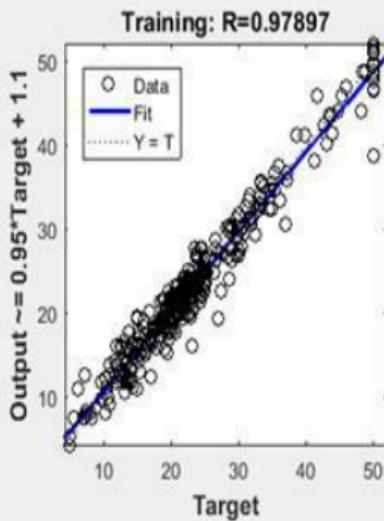
```
perform(net,targets(tInd),tstOutputs)
```

```
tstPerform =
```

9.8912

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 0.96 for the total response. If even more accurate results were required, you could try any of these approaches:

1. Reset the initial network weights and biases to new values with [init](#) and train again .
2. Increase the number of hidden neurons.
3. Increase the number of training vectors.
4. Increase the number of input values,

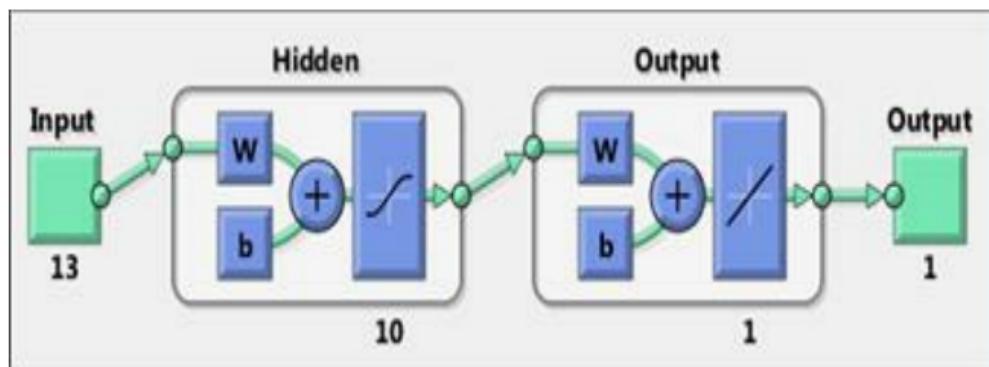
if more relevant information is available.

5. Try a different training algorithm.

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

View the network diagram.

`view(net)`



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

1. During training, open a plot window (such as the regression plot), and watch it animate.
2. 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.)

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

line.

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

Chapter 10

**CLASSIFY PATTERNS WITH
A NEURAL NETWORK.
GRAPHICAL INTERFACE**

10.1 INTRODUCTION

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

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

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

- Use the nprtool GUI, as described in [Using the Neural Network Pattern](#)

Recognition Tool.

- Use a command-line solution, as described in [Using Command-Line Functions](#).

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. The next section describes the data format.

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 two-class exclusive-or classification problem as follows:

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

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

Target vectors 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 shows how to

train a network to recognize patterns, using the neural network pattern recognition tool GUI, [nprtool](#). This example uses the cancer data set provided with the toolbox. This data set consists of 699 nine-element input vectors and two-element target vectors. There are two elements in each target vector, because there are two categories (benign or malignant) associated with each input vector.

10.2 USING THE NEURAL NETWORK PATTERN RECOGNITION TOOL

If needed, open the Neural Network Start GUI with this command:

- nnstart



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Getting Started Wizards

More Information

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

Input-output and curve fitting.



Fitting app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npctool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

- **Click Pattern Recognition Tool** to open the Neural Network Pattern Recognition Tool. (You can also use the command [nprtool](#).)



Welcome to the Neural Pattern Recognition app.

Solve a pattern-recognition problem with a two-layer feed-forward network.

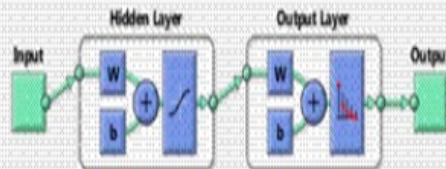
Introduction

In pattern recognition problems, you want a neural network to classify inputs into a set of target categories.

For example, recognize the vineyard that a particular bottle of wine came from, based on chemical analysis ([wine dataset](#)) ; or classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis ([cancer dataset](#)).

The Neural Pattern Recognition app will help you select data, create and train a network, and evaluate its performance using cross-entropy and confusion matrices.

Neural Network



A two-layer feed-forward network, with sigmoid hidden and softmax output neurons ([pattern](#)), can classify vectors arbitrarily well, given enough neurons in its hidden layer.

The network will be trained with scaled conjugate gradient backpropagation ([theory](#)).

To continue, click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

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

Select Data

What inputs and targets define your pattern recognition problem?

Get Data from Workspace

Input data to present to the network.

Inputs:

(none)

Target data defining desired network output.

Targets:

(none)

Samples are

Matrix columns Matrix rows

Want to try out this tool with an example data set?

Summary

No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

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



Select a data set:

Simple Classes

Iris Flowers

Breast Cancer

Types of Glass

Thyroid

Wine Vintage

Description

Filename: [cancer dataset](#)

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

This dataset can be used to design a neural network that classifies cancers as either benign or malignant depending on the characteristics of sample biopsies.

LOAD [cancer dataset](#).MAT loads these two variables:

cancerInputs - a 9x699 matrix defining nine attributes of 699 biopsies.

1. Clump thickness
2. Uniformity of cell size
3. Uniformity of cell shape
4. Marginal Adhesion
5. Single epithelial cell size
6. Bare nuclei
7. Bland chromatin
8. Normal nucleoli

Import

Cancel

- Select **Breast Cancer** and click **Import**. You return to the Select Data window.
- Click **Next** to continue to the Validation and Test Data window.



Validation and Test Data

Set aside some samples for validation and testing.

Select Percentages

Randomly divide up the 699 samples:

Training:	70%	489 samples
Validation:	15%	105 samples
Testing:	15%	105 samples

Explanation

Three Kinds of Samples:

Training:

These are presented to the network during training, and the network is adjusted according to its error.

Validation:

These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:

These have no effect on training and so provide an independent measure of network performance during and after training.

Restore Defaults

Change percentages if desired, then click [Next] to continue.

Neural Network Start

Welcome

Back

Next

Cancel

Validation and test data sets are each set to 15% of the original data. With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

1. 70% are used for training.
2. 15% are used to validate that the network is generalizing and to stop training before overfitting.
3. The last 15% are used as a completely independent

test of network generalization.

• Click Next.

The standard network that is used for pattern recognition is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer. The default number of hidden neurons is set to 10. You might want to come back and increase this number if the network does not perform as well as you expect. The number of output neurons is set to 2, which is equal to the number of elements in the target vector (the number of categories).

Network Architecture

Set the number of neurons in the pattern recognition network's hidden layer.

Hidden Layer

Define a pattern recognition neural network... (patternnet)

Number of Hidden Neurons:

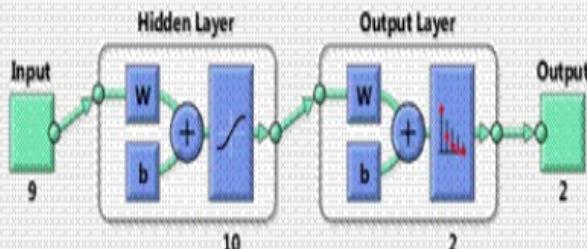
10

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

[Restore Defaults](#)

Neural Network



Change settings if desired, then click [Next] to continue.

[Neural Network Start](#)

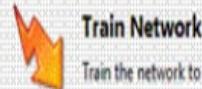
[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

- Click **Next**.



Train Network

Train the network to classify the inputs according to the targets.

Train Network

Train using scaled conjugate gradient backpropagation. (trainscg)



Training automatically stops when generalization stops improving, as indicated by an increase in the cross-entropy error of the validation samples.

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

Results

	Samples	CE	%
Training:	489		
Validation:	105		
Testing:	105		



Minimizing Cross-Entropy results in good classification. 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].

Neural Network Start

Welcome

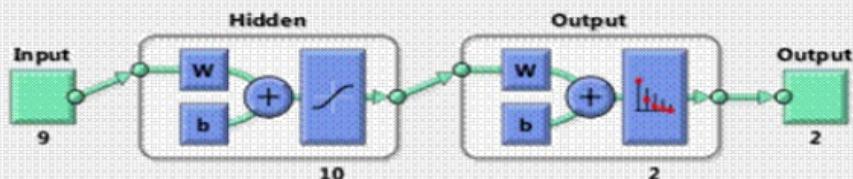
Back

Next

Cancel

· Click Train.

Neural Network



Algorithms

Data Division: Random (dividerand)
 Training: Scaled Conjugate Gradient (trainscg)
 Performance: Cross-Entropy (crossentropy)
 Calculations: MEX

Progress

Epoch:	0	20 iterations	1000
Time:		0:00:00	
Performance:	0.424	0.0236	0.00
Gradient:	0.442	0.0188	1.00e-06
Validation Checks:	0	6	6

Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval: 1 epochs



Validation stop.



Stop Training



Cancel

The training continues for 55 iterations.

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 outputs are very accurate, 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.

Confusion (plotconfusion)

File Edit View Insert Tools Desktop Window Help

Training Confusion Matrix

Output Class	1	2	3
1	312 63.8%	1 0.2%	99.7% 0.3%
2	8 1.6%	168 34.4%	95.5% 4.5%
3	97.5% 2.5%	99.4% 0.6%	98.2% 1.8%

Target Class

Validation Confusion Matrix

Output Class	1	2	3
1	66 62.9%	3 2.9%	95.7% 4.3%
2	3 2.9%	33 31.4%	91.7% 8.3%
3	95.7% 4.3%	91.7% 8.3%	94.3% 5.7%

Target Class

Test Confusion Matrix

Output Class	1	2	3
1	66 62.9%	3 2.9%	95.7% 4.3%
2	3 2.9%	33 31.4%	91.7% 8.3%
3	95.7% 4.3%	91.7% 8.3%	94.3% 5.7%

Target Class

All Confusion Matrix

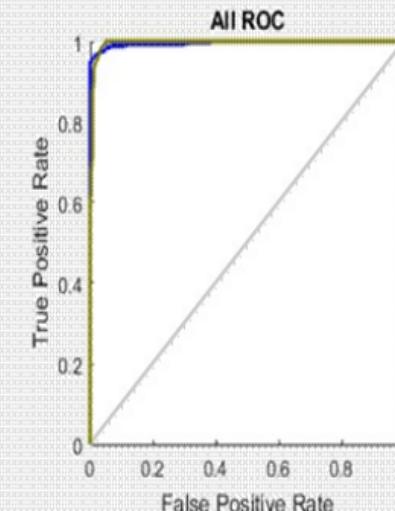
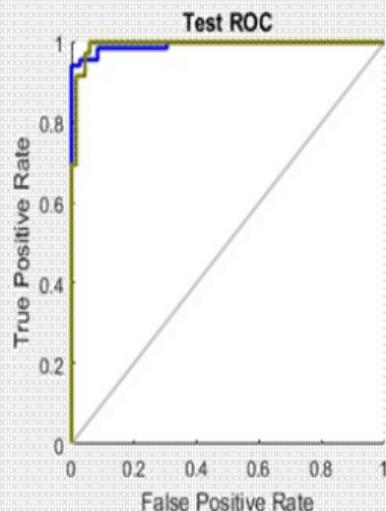
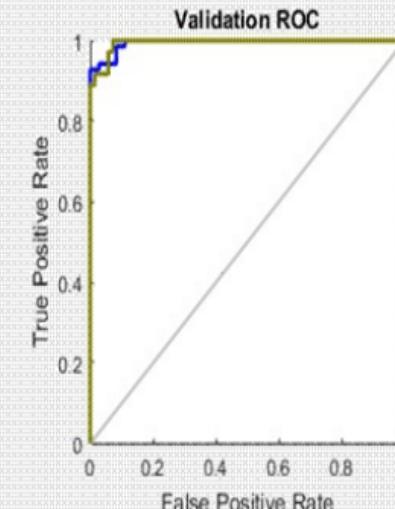
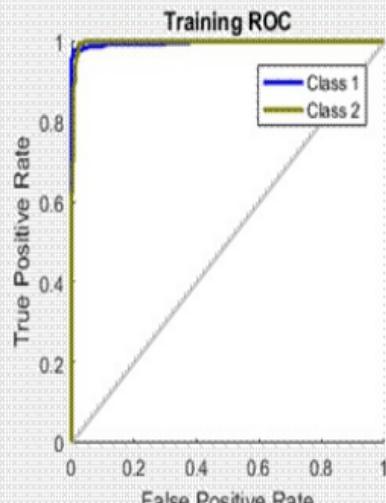
Output Class	1	2	3
1	444 63.5%	7 1.0%	98.4% 1.6%
2	14 2.0%	234 33.5%	94.4% 5.6%
3	96.9% 3.1%	97.1% 2.9%	97.0% 3.0%

Target Class

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

Receiver Operating Characteristic (plotroc)

File Edit View Insert Tools Desktop Window Help



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



Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

Try training again if a first try did not generate good results
or you require marginal improvement.

Train Again

Increase network size if retraining did not help.

Adjust Network Size

Not working? You may need to use a larger data set.

Import Larger Data Set

Test Network

CE

PE

Plot Confusion

Plot ROC

Select inputs and targets, click an improvement button, or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

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. If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the number of neurons can improve your results.

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

Use this panel to generate a MATLAB

function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs or deploy the network with MATLAB Compiler tools and other MATLAB code generation tools.



Deploy Solution

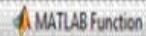
Generate deployable versions of your trained neural network.

Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

(gensim)



Graphics

Generate a graphical diagram of the neural network:

(network/view)



- ! Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

- Click **Next**. Use the buttons on this screen to save your results.

Save Results



Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool



Generate a script with additional options and example code:



Save Data to Workspace

Save network to MATLAB network object named:

net

Save performance and data set information to MATLAB struct named:

info

Save outputs to MATLAB matrix named:

output

Save errors to MATLAB matrix named:

error

Save inputs to MATLAB matrix named:

input

Save targets to MATLAB matrix named:

target

Save ALL selected values above to MATLAB struct named:

results

Restore Defaults



Save results and click [Finish].

Neural Network Start

Welcome

Back

Next

Finish

- You can click **Simple Script** or **Advanced Script** to create MATLAB[®] code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.
- You can also save the network as net in the workspace. You can perform additional tests on it or put it to work on new inputs.
- When you have saved your results, click **Finish**.

10.3 USING COMMAND-LINE FUNCTIONS

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. For example, look at the simple script that was created at step 14 of the previous section.

```
% Solve a Pattern Recognition Problem  
with a Neural Network
```

```
% Script generated by NPRTOOL
```

%

% This script assumes these variables
are defined:

%

% cancerInputs - input data.

% cancerTargets - target data.

inputs = cancerInputs;

targets = cancerTargets;

% Create a Pattern Recognition Network

hiddenLayerSize = 10;

net = patternnet(hiddenLayerSize);

% Set up Division of Data for Training,
Validation, Testing

net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

net.divideParam.testRatio = 15/100;

% Train the Network

[net,tr] = train(net,inputs,targets);

% Test the Network

outputs = net(inputs);

errors = gsubtract(targets,outputs);

```
performance =  
perform(net,targets,outputs)
```

```
% View the Network
```

```
view(net)
```

```
% Plots
```

```
% Uncomment these lines to enable  
various plots.
```

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

```
% figure, plottrainstate(tr)
```

```
% figure, plotconfusion(targets,outputs)
```

```
% figure, ploterrhist(errors)
```

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

The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
[inputs,targets] = cancer_dataset;
```

Create the network. The default network for function fitting (or regression) problems, [patternnet](#), is a feedforward network with the default tan-sigmoid transfer function in the hidden layer, and

a softmax transfer function in the output layer. You assigned ten neurons (somewhat arbitrary) to the one hidden layer in the previous section.

The network has two output neurons, because there are two target values (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 the network, enter these

commands:

```
hiddenLayerSize = 10;
```

```
net = patternnet(hiddenLayerSize);
```

Note The choice of network architecture for pattern recognition problems follows similar guidelines to function fitting problems. More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes

as elements of an array in the [patternnet](#) command.

Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
```

```
net.divideParam.valRatio = 15/100;
```

```
net.divideParam.testRatio = 15/100;
```

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

Train the network. The pattern recognition network uses the default Scaled Conjugate Gradient ([trainscg](#)) algorithm for training. To train the

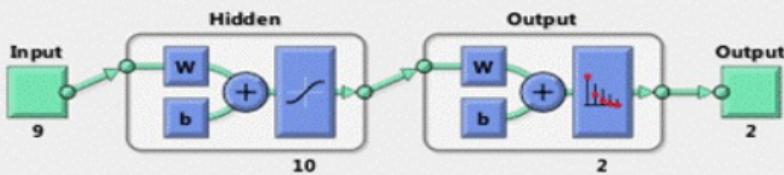
network, enter this command:

```
[net,tr] = train(net,inputs,targets);
```

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

Neural Network Training (nntraintool)

Neural Network



Algorithms

Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Derivative: Default (defaultderiv)

Progress

Epoch:	0	9 iterations	1000
Time:		0:00:00	
Performance:	0.909	0.0427	0.00
Gradient:	1.40	0.0243	1.00e-06
Validation Checks:	0	6	6

Plots

- Performance** (plotperform)
- Training State (plottrainstate)
- Error Histogram (ploterrhist)
- Confusion (plotconfusion)
- Receiver Operating Characteristic (plotroc)

Plot Interval: 1 epochs



Validation stop.

Stop Training

Cancel

This training stopped when the validation error increased for six iterations, which occurred at iteration 24.

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

```
outputs = net(inputs);
```

```
errors = gsubtract(targets,outputs);
```

```
performance =
```

```
perform(net,targets,outputs)
```

```
performance =
```

0.0307

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

```
tInd = tr.testInd;
```

```
tstOutputs = net(inputs(:,tInd));
```

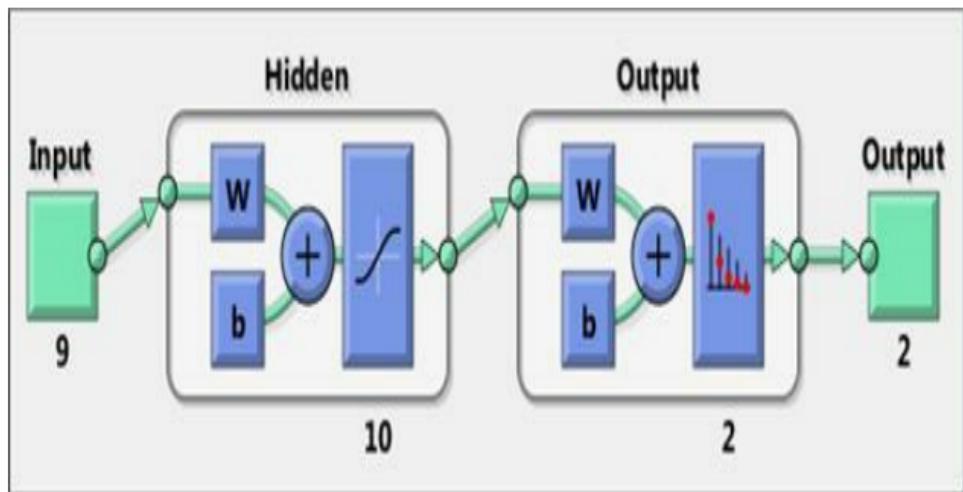
```
tstPerform =  
perform(net,targets(:,tInd),tstOutputs)
```

```
tstPerform =
```

0.0257

View the network diagram.

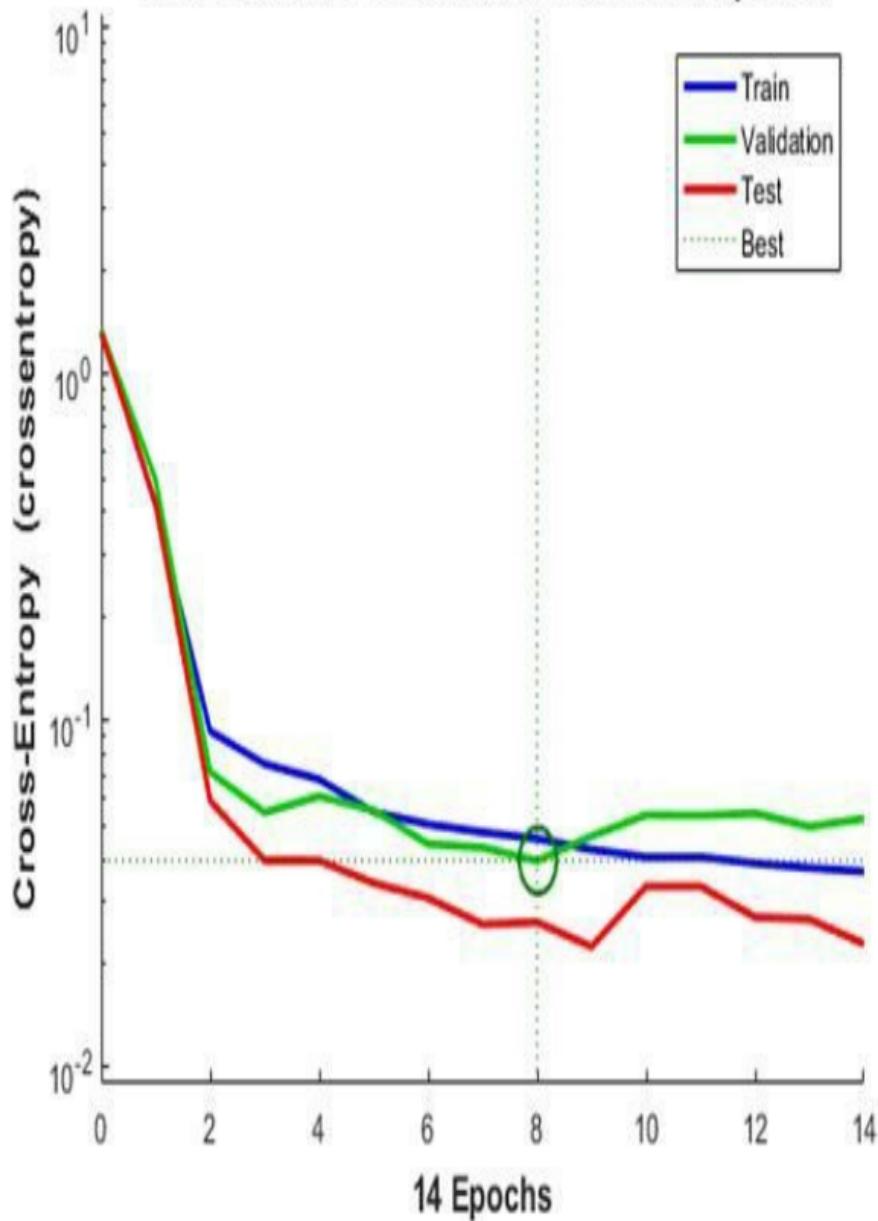
`view(net)`



Plot the training, validation,
and test performance.

`figure, plotperform(tr)`

Best Validation Performance is 0.039514 at epoch 8



Use the [plotconfusion](#) function to plot the confusion matrix. It shows the various types of errors that occurred for the final trained network.

```
figure, plotconfusion(targets,outputs)
```

Confusion Matrix

		Target Class
Output Class	1	2
	1	2
1	446 63.8%	5 0.7%
2	12 1.7%	236 33.8%
	97.4% 2.6%	97.9% 2.1%
		97.6% 2.4%

The diagonal cells 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 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 "[Training Algorithms](#)").

In this case, the network response is satisfactory, and you can now 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 [plotroc](#) and [plottrainstate](#).

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

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

been found, retrain several times.

Chapter 11

**CLUSTER DATA WITH A
SELF-ORGANIZING MAP.
GRAPHICAL INTERFACE**

11.1 INTRODUCTION

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

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

expression patterns

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

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

- Use the [nctool](#) GUI.
- Use a command-line solution.

To define a clustering problem, simply arrange Q input vectors to be clustered as columns in an input matrix (see ["Data](#)

[Structures](#)" for a detailed description of data formatting for static and time-series data). 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 shows how to train a network using the [nctool](#) GUI.

11.2 USING THE NEURAL NETWORK CLUSTERING TOOL

If needed, open the Neural Network Start GUI with this command:

```
nnstart
```



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Getting Started Wizards

More Information

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

Input-output and curve fitting.



Fitting app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npztool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

Click **Clustering Tool** to open
the Neural Network Clustering Tool.
(You can also use the command [nctool](#).)



Welcome to the Neural Clustering app.

Solve a clustering problem with a self-organizing map (SOM) network.

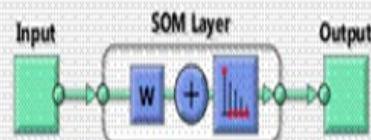
Introduction

In clustering problems, you want a neural network to group data by similarity.

For example: market segmentation done by grouping people according to their buying patterns; data mining can be done by partitioning data into related subsets; or bioinformatic analysis such as grouping genes with related expression patterns.

The Neural Clustering app will help you select data, create and train a network, and evaluate its performance using a variety of visualization tools.

Neural Network



A self-organizing map ([selforgmap](#)) consists of a competitive layer which can classify a dataset of vectors with any number of dimensions into as many classes as the layer has neurons. The neurons are arranged in a 2D topology, which allows the layer to form a representation of the distribution and a two-dimensional approximation of the topology of the dataset.

The network is trained with the SOM batch algorithm ([trainSOM](#), [learnSOM](#)).

To continue, click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

Click Next. The Select Data window appears.



Select Data

What inputs define your clustering problem?

Get Data from Workspace

Input data to be clustered.

Inputs:

(none)



Samples are:

Matrix columns

Matrix rows

Summary

No inputs selected.

Want to try out this tool with an example data set?

[Load Example Data Set](#)



Select inputs, then click [Next].

[Neural Network Start](#)

Welcome

Back

Next

Cancel

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

Clustering Data Set Chooser



Select a data set:

Simple Clusters

Iris Flowers

Description

Filename: [simplecluster dataset](#)

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

This dataset can be used to demonstrate how a neural network can be trained develop its own classification system for a set of examples.

LOAD [simplecluster dataset](#).MAT loads these two variables:

simplecluster/inputs - a 2x1000 matrix of 1000 two-element vectors.

[X,T] = [simplecluster dataset](#) loads the inputs and targets into variables of your own choosing.

For an intro to clustering with the [Neural Clustering app](#), click "Load Example Data Set" in the second panel and pick this dataset.

Here is how to design an 8x8 clustering neural network with this data at the command line. See [selforoma](#) for more details.

Import

Cancel

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

Data window.

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

For clustering problems, the self-organizing feature map (SOM) is the most commonly used network, because after the network has been trained, there are many visualization tools that can be used to analyze the resulting clusters.

This network has one layer, with neurons organized in a grid. When creating the network, you specify the numbers of rows and columns in the grid. Here, the number of rows and columns is set to 10. The total number of neurons is 100. You can change this number in another run if

you want.



Network Architecture

Set the number of neurons in the self-organizing map network.

Self-Organizing Map

Define a self-organizing map. (selforgmap)

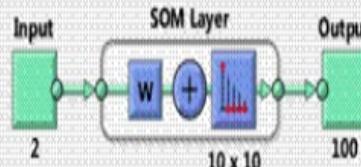
Size of two-dimensional Map:

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

[Restore Defaults](#)

Neural Network



Change settings if desired, then click [Next] to continue.

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

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



Train Network

Train the network to learn the topology and distribution of the input samples.

Train Network

Train using batch SOM algorithm. (trainbu) (learnsomb)

**Train**

Results

[Plot SOM Neighbor Distances](#)[Plot SOM Weight Planes](#)[Plot SOM Sample Hits](#)[Plot SOM Weight Positions](#)

Training automatically stops when the full number of epochs have occurred.

Notes

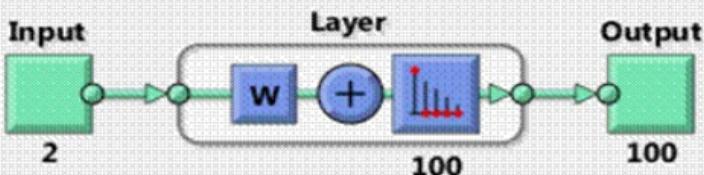
 Training multiple times will generate different results due to different initial conditions and sampling.

 Train network, then click [Next].

[Neural Network Start](#)[Welcome](#)[Back](#)[Next](#)[Cancel](#)

Click Train.

Neural Network



Algorithms

Training: Batch Weight/Bias Rules (trainbu)

Performance: Mean Squared Error (mse)

Calculations: MATLAB

Progress

Epoch: 0 200 iterations 200

Time: 0:00:02

Plots

SOM Topology (plotsomtop)

SOM Neighbor Connections (plotsomnc)

SOM Neighbor Distances (plotsomnd)

SOM Input Planes (plotsomplanes)

SOM Sample Hits (plotsomhits)

SOM Weight Positions (plotsompos)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training



Cancel

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

For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. Investigate some of the visualization tools for the SOM.

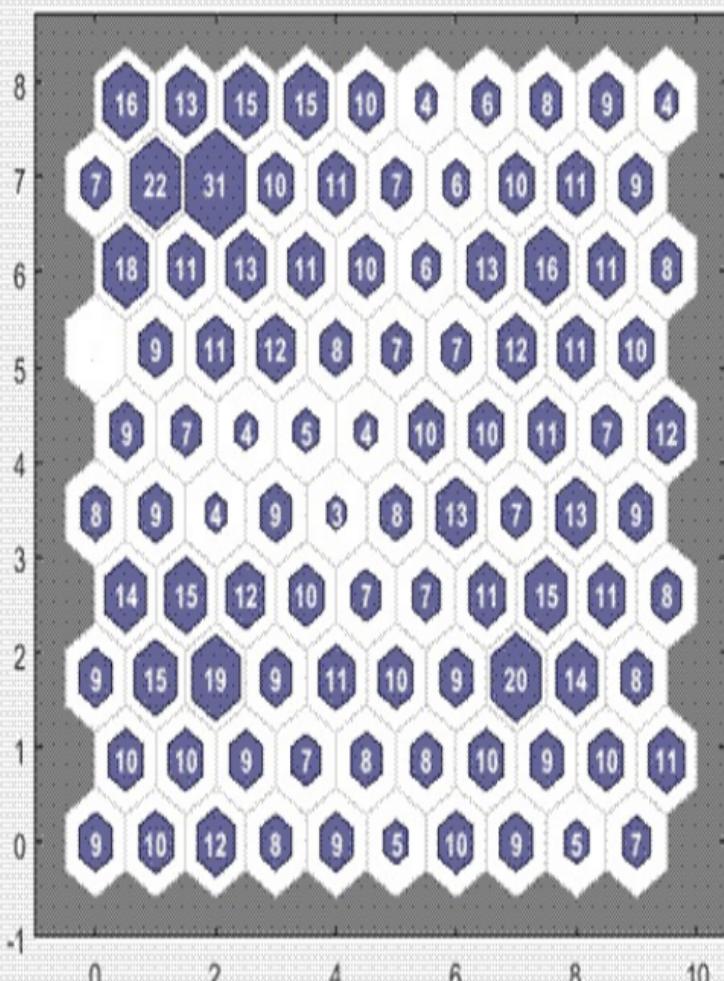
Under the **Plots** pane, click **SOM Sample Hits**.

SOM Sample Hits (plotsomhits)



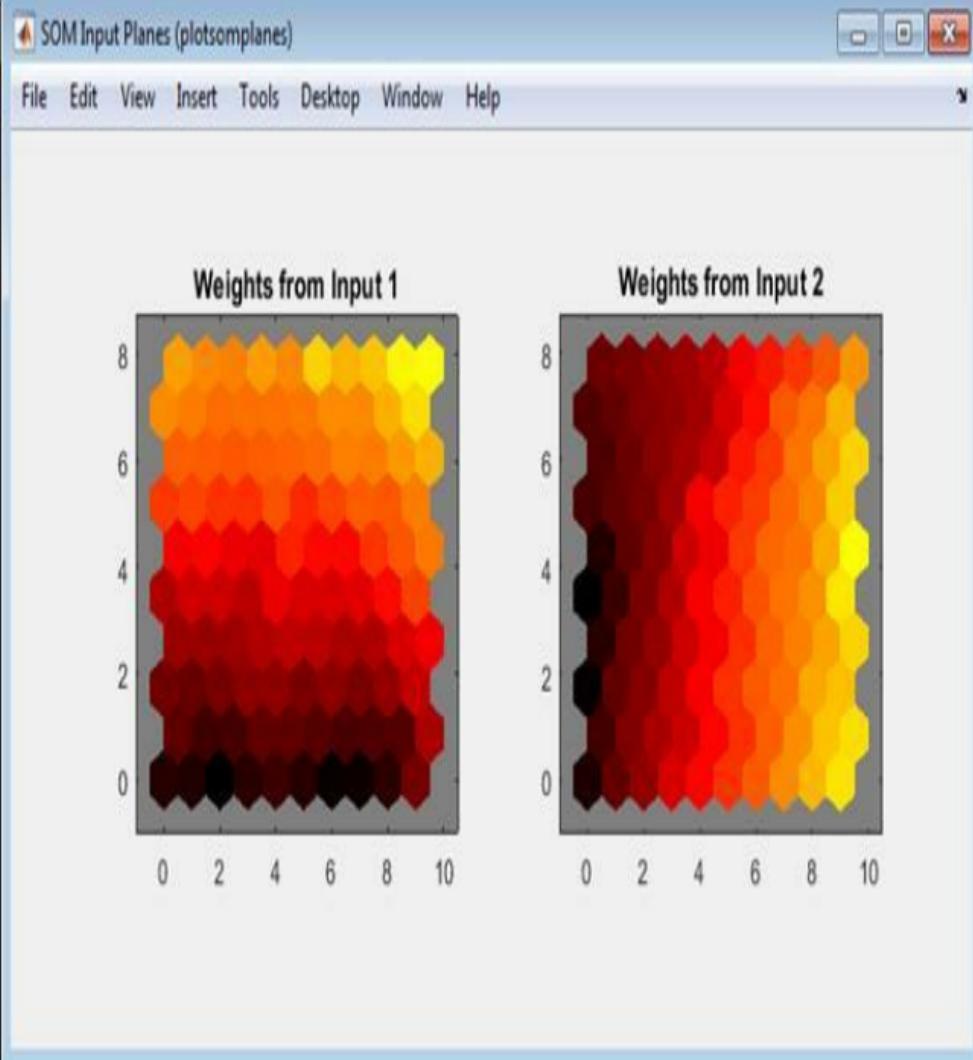
File Edit View Insert Tools Desktop Window Help

Hits



The default topology of the SOM is hexagonal. This figure shows the neuron locations in the topology, and indicates 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.

You can also visualize the SOM by displaying weight planes (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.

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



Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

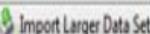
Try training again if a first try did not generate good results or you require marginal improvement.



Increase network size if retraining did not help.



Not working? You may need to use a larger data set.



Optionally perform additional tests



(none)



Samples are:

[Matrix columns] [Matrix rows]

No inputs selected.



Plot SOM Neighbor Distances

Plot SOM Weight Planes

Plot SOM Sample Hits

Plot SOM Weight Positions



Select inputs, click an improvement button, or click [Next].



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.

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

Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs or deploy the network with MATLAB

Compiler tools and other MATLAB and Simulink code generation tools.



Deploy Solution

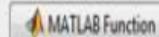
Generate deployable versions of your trained neural network.

Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

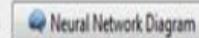
(gensim)



Graphics

Generate a graphical diagram of the neural network:

(network/view)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

Use the buttons on this screen to save your results.



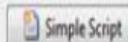
Save Results

Generate MATLAB scripts, save results and generate diagrams.

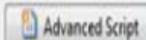
Generate Scripts

Recommended >> Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool:



Generate a script with additional options and example code:



Save Data to Workspace

Save network to MATLAB network object named:

net:

Save outputs to MATLAB matrix named:

output

Save inputs to MATLAB matrix named:

input

Save ALL selected values above to MATLAB struct named:

results

Save results and click [Finish].

You can click **Simple Script** or **Advanced Script** to create MATLAB® code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

You can also save the network as net in the workspace. You can perform additional tests on it or put it to work on new inputs.

When you have generated scripts and saved your results, click **Finish**.

11.3 USING COMMAND-LINE FUNCTIONS

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created in step 14 of the previous section.

```
% Solve a Clustering Problem with a  
Self-Organizing Map  
% Script generated by NCTOOL  
%  
% This script assumes these variables  
are defined:
```

%

% simpleclusterInputs - input data.

inputs = simpleclusterInputs;

% Create a Self-Organizing Map

dimension1 = 10;

dimension2 = 10;

net = selforgmap([dimension1
dimension2]);

% Train the Network

[net,tr] = train(net,inputs);

% Test the Network

outputs = net(inputs);

```
% View the Network  
view(net)
```

```
% Plots
```

```
% Uncomment these lines to enable  
various plots.
```

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

```
% figure, plotsomnc(net)
```

```
% figure, plotsomnd(net)
```

```
% figure, plotsomplanes(net)
```

```
% figure, plotsomhits(net,inputs)
```

```
% figure, plotsompos(net,inputs)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, let's

follow each of the steps in the script.

The script assumes that the input vectors are already loaded into the workspace. To show the command-line operations, you can use a different data set than you used for the GUI operation. Use the flower data set as an example. The iris data set consists of 150 four-element input vectors.

```
load iris_dataset  
inputs = irisInputs;
```

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. When creating the network with [selforgmap](#), you specify the number of rows and columns in the

grid:

```
dimension1 = 10;
```

```
dimension2 = 10;
```

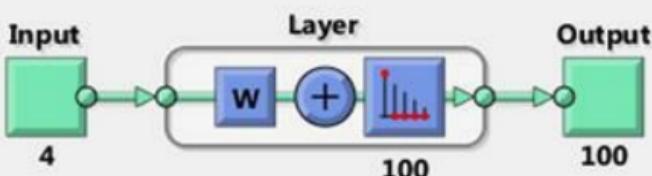
```
net = selforgmap([dimension1  
dimension2]);
```

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

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

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

Neural Network



Algorithms

Training: Batch Weight/Bias Rules (trainbu)

Performance: Mean Squared Error (mse)

Calculations: MATLAB

Progress

Epoch: 0 200

Time: 0:00:00

Plots

SOM Topology

(plotsomtop)

SOM Neighbor Connections

(plotsomnc)

SOM Neighbor Distances

(plotsomnd)

SOM Input Planes

(plotsomplanes)

SOM Sample Hits

(plotsomhits)

SOM Weight Positions

(plotsompos)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training



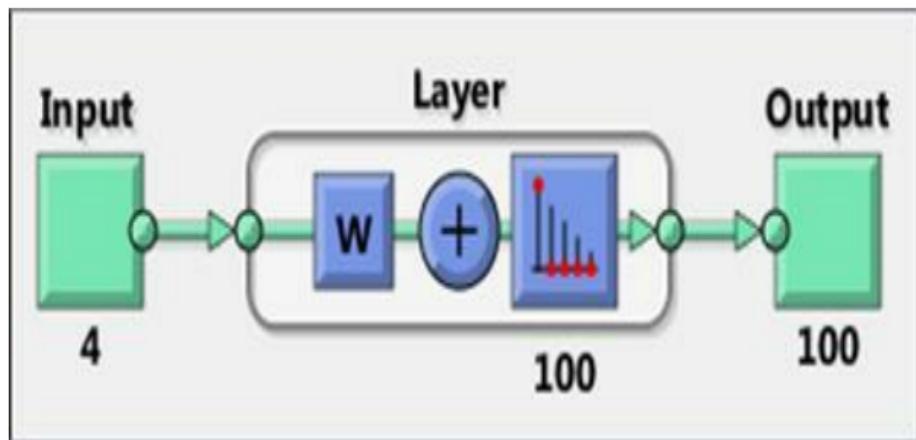
Cancel

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

```
outputs = net(inputs);
```

View the network diagram.

```
view(net)
```

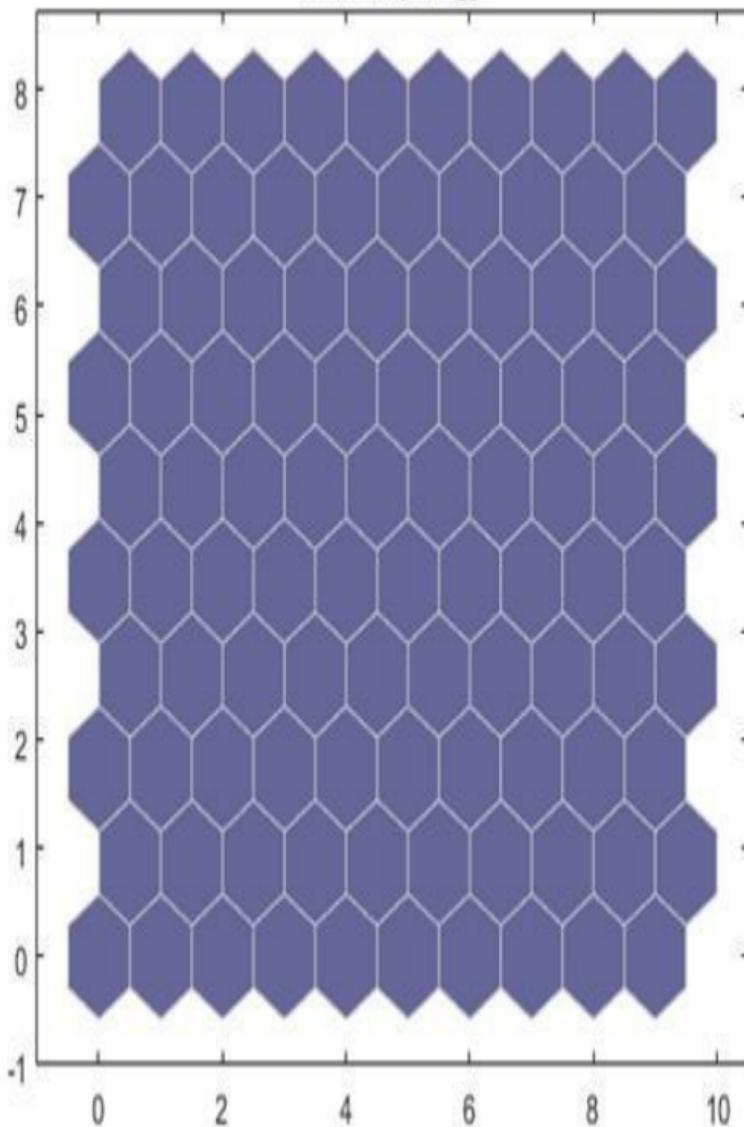


For SOM training, the weight vector

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

figure, plotsomtop(net)

SOM Topology



In this figure, each of the hexagons represents a neuron. The grid is 10-by-10, so there are a total of 100 neurons in this network. There are four 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*).

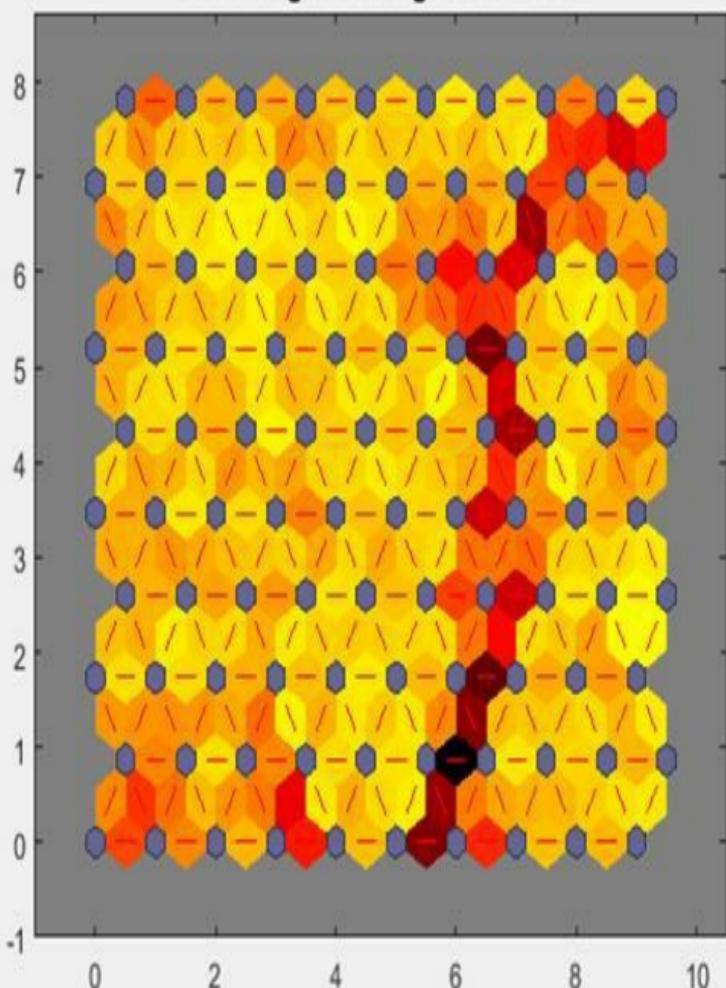
To view the U-matrix, click **SOM Neighbor Distances** in the training window.

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.

Neural Network Training SOM Neighbor Distances (plotsomnd), Epoch 200, Maxi...

File Edit View Insert Tools Desktop Window Help

SOM Neighbor Weight Distances



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](#), [pl](#) and [plotsomtop](#).

Chapter 12

NEURAL NETWORK TIME-SERIES PREDICTION AND MODELING. GRAPHICAL

INTERFACE

12.1 INTRODUCTION

Dynamic neural networks are good at time-series prediction.

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

You can solve this problem in two ways:

- Use a graphical user

interface, [ntstool](#).

- Use command-line functions.

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox. If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

To define a time-series problem for the toolbox, arrange a set of TS input

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

targets = {1 2 3 4 5};

The next section shows how to train a network to fit a time-series data set, using the neural network time-series tool GUI, [*ntstool*](#). This example uses the pH neutralization data set provided with the toolbox.

12.2 USING THE NEURAL NETWORK TIME SERIES TOOL

If needed, open the Neural Network Start GUI with this command:

`nnstart`



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Getting Started Wizards

More Information

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

Input-output and curve fitting.



Fitting app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npctool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

Click Time Series Tool to open
the Neural Network Time Series Tool.
(You can also use the command [ntstool](#).)



Welcome to the Neural Network Time Series Tool.

Solve a nonlinear time series problem with a dynamic neural network.

Introduction

Prediction is a kind of dynamic filtering, in which past values of one or more time series are used to predict future values. Dynamic neural networks, which include tapped delay lines are used for nonlinear filtering and prediction.

There are many applications for prediction. For example, a financial analyst might want to predict the future value of a stock, bond or other financial instrument. An engineer might want to predict the impending failure of a jet engine.

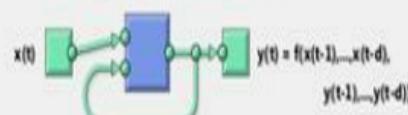
Predictive models are also used for system identification (or dynamic modelling), in which you build dynamic models of physical systems. These dynamic models are important for analysis, simulation, monitoring and control of a variety of systems, including manufacturing systems, chemical processes, robotics and aerospace systems.

This tool allows you to solve three kinds of nonlinear time series problems shown in the right panel. Choose one and click [Next].

Select a Problem

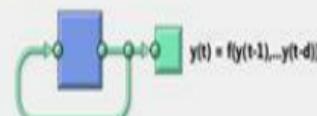
Nonlinear Autoregressive with External (Exogenous) Input (NARX)

Predict series $y(t)$ given d past values of $y(t)$ and another series $x(t)$.



Nonlinear Autoregressive (NAR)

Predict series $y(t)$ given d past values of $y(t)$.



Nonlinear Input-Output

Predict series $y(t)$ given d past values of series $x(t)$.

Important Note: NARX solutions are more accurate than this solution. Only use this solution if past values of $y(t)$ will not be available when deployed.



Choose a problem, then click [Next].

Notice that this opening pane is different than the opening panes for the other GUIs. This is because [ntstool](#) can be used to solve three different kinds of time-series problems.

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

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

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

In the second type of time-series problem, there is only one series involved. The future values of a time series $y(t)$ are predicted only from past

values of that series. This form of prediction is called nonlinear autoregressive, or NAR, and can be written as follows:

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

This model could also be used to predict financial instruments, but without the use of a companion series.

The third time-series problem is similar to the first type, in that two series are involved, an input series $x(t)$ and an output/target series $y(t)$. Here you want to predict values of $y(t)$ from previous values of $x(t)$, but without knowledge of

previous values of $y(t)$. This input/output model can be written as follows:

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

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

For this example, select the NARX

model and click **Next** to proceed.



Select Data

What inputs and targets define your nonlinear autoregressive problem?

Get Data from Workspace

Input time series $x(t)$:

Inputs:

(none)

Summary

No inputs selected.

Target time series, defining the desired output $y(t)$:

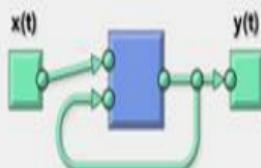
Targets:

(none)

No targets selected.

Select the time series format. (tonndata)

Time step: Cell column Matrix column Matrix row



Want to try out this tool with an example data set?

Load Example Data Set



Select inputs and targets, then click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

Click Load Example Data Set in the Select Data window. The Time Series Data Set Chooser window opens.

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



Select a data set:

Simple NARX Problem

Heat Exchanger

Magnetic Levitation

pH Neutralization Process

Pollution Mortality

Fluid Flow in Pipe

Description

Filename: [simpnarx dataset](#)

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

This dataset can be used to demonstrate how a neural network can be trained to make predictions.

LOAD [simpnarx dataset](#).MAT loads these two variables:

simpnarxInputs - a 1x100 cell array of scalar values representing a 100 timestep time-series.

simpnarxTargets - a 1x100 cell array of scalar values representing a 100 timestep time-series to be predicted.

[X,T] = [simpnarx dataset](#) loads the inputs and targets into variables of your own choosing.

For an intro to prediction with the [Neural Time Series app](#)

Import

Cancel

Select pH Neutralization Process, and

click **Import**. This returns you to the Select Data window.

Click **Next** to open the Validation and Test Data window, shown in the following figure.

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



Validation and Test Data

Set aside some target timesteps for validation and testing.

Select Percentages

Randomly divide up the 2001 target timesteps:

Training:	70%	1401 target timesteps
Validation:	<input type="button" value="15% ▾"/>	300 target timesteps
Testing:	<input type="button" value="15% ▾"/>	300 target timesteps

Explanation

Three Kinds of Target Timesteps:

Training:

These are presented to the network during training, and the network is adjusted according to its error.

Validation:

These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:

These have no effect on training and so provide an independent measure of network performance during and after training.

Change percentages if desired, then click [Next] to continue.

Neural Network Start

Welcome

Back

Next

Cancel

With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

- 70% will be used for training.
- 15% will be used to validate that the network is generalizing and to stop training before overfitting.
- The last 15% will be used as a completely independent test of network generalization.

Click Next.



Network Architecture

Choose the number of neurons and input/feedback delays.

Architecture Choices

Define a NARX neural network. (narxnet)

Number of Hidden Neurons:

Number of delays d:

Problem definition:

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

[Restore Defaults](#)

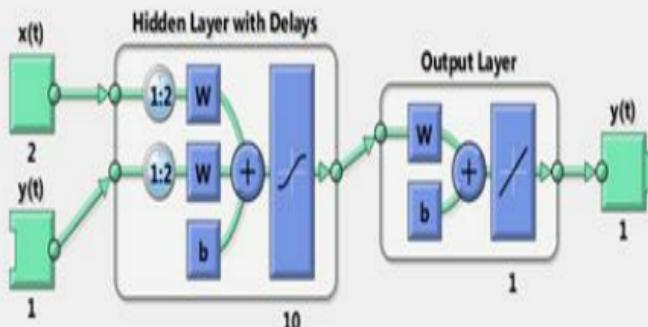
Recommendation

Return to this panel and change the number of neurons or delays if the network does not perform well after training.

The network will be created and trained in open loop form as shown below. Open loop (single-step) is more efficient than closed loop (multi-step) training. Open loop allows us to supply the network with correct past outputs as we train it to produce the correct current outputs.

After training, the network may be converted to closed loop form, or any other form, that the application requires.

Neural Network



Change settings if desired, then click [Next] to continue.

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

The standard NARX network is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This network also uses tapped delay lines to store previous values of the $x(t)$ and $y(t)$ sequences.

Note that the output of the NARX network, $y(t)$, is fed back to the input of the network (through delays), since $y(t)$ is a function of $y(t - 1)$, $y(t - 2)$, ..., $y(t - d)$. However, for efficient training this feedback loop can be opened.

Because the true output is available during the training of the network, you

can use the open-loop architecture shown above, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and therefore a more efficient algorithm can be used for training. This network is discussed in more detail in ["NARX Network"](#) (`narxnet`, `closeloop`).

The default number of hidden neurons is set to 10. The default number of delays is 2. Change this value to 4. You might want to adjust these numbers if the

network training performance is poor.

Click Next.



Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm:

Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)



Results

	Target Values	MSE	R
--	---------------	-----	---

	Training:	1401	-
	Validation:	300	-
	Testing:	300	-

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.

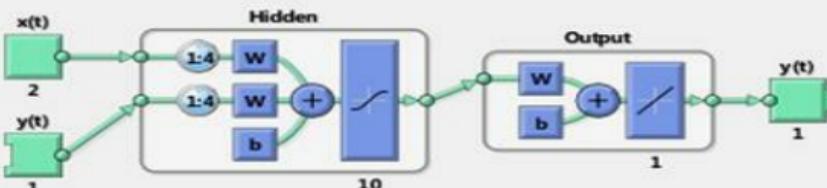
Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Train network, then click [Next].

Select a training algorithm, then click **Train**. Levenberg-Marquardt (`trainlm`) is recommended for most problems, but for some noisy and small problems Bayesian Regularization (`trainbr`) can take longer but obtain a better solution. For large problems, however, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use. This example uses the default Levenberg-Marquardt.

The training continued until the

validation error failed to decrease for six iterations (validation stop).

Neural Network**Algorithms**

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

Progress

Epoch:	0	39 iterations	1000
Time:		0:00:00	
Performance:	33.5	0.00196	0.00
Gradient:	79.8	0.0756	1.00e-07
Mu:	0.00100	1.00e-07	1.00e+10
Validation Checks:	0	6	6

Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Regression (plotregression)

Time-Series Response (plotresponse)

Error Autocorrelation (ploterrcorr)

Input-Error Cross-correlation (plotinerrcorr)

Plot Interval:



Validation stop.



Stop Training



Cancel

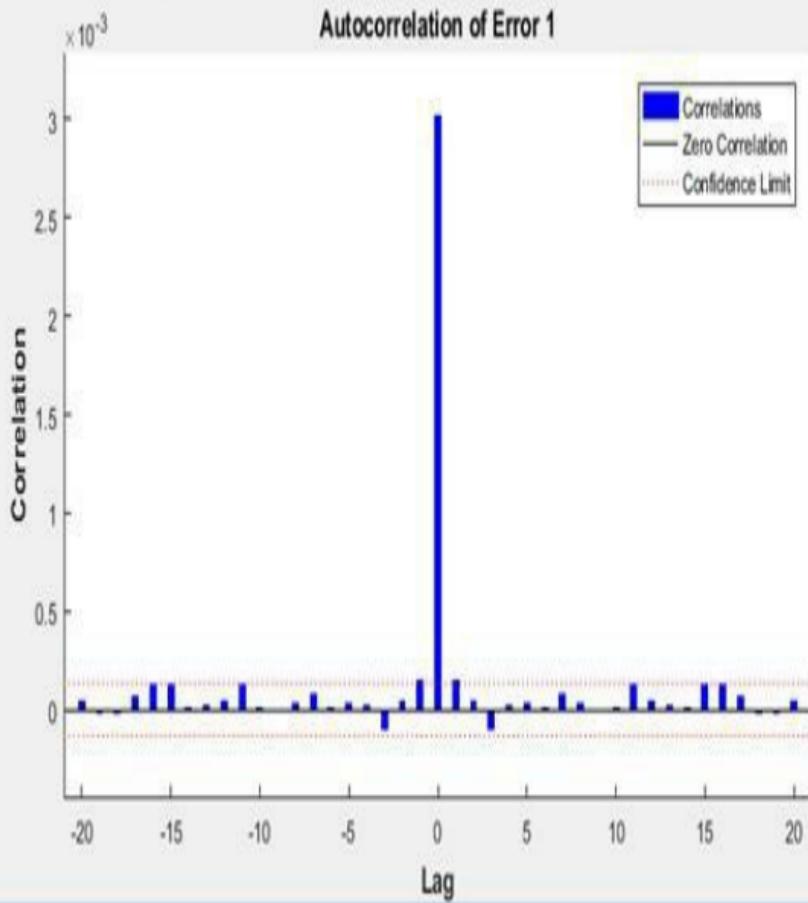
Under Plots, click Error Autocorrelation. This is used to validate the network performance.

The following plot displays the error autocorrelation function. It describes how the prediction errors are related in time. For a perfect prediction model, there should only be one nonzero value of the autocorrelation function, and it should occur at zero lag. (This is the mean square error.) This would mean that the prediction errors were completely uncorrelated with each other (white noise). If there was significant

correlation in the prediction errors, then it should be possible to improve the prediction - perhaps by increasing the number of delays in the tapped delay lines. In this case, the correlations, except for the one at zero lag, fall approximately within the 95% confidence limits around zero, so the model seems to be adequate. If even more accurate results were required, you could retrain the network by clicking **Retrain** in [ntstool](#). This will change the initial weights and biases of the network, and may produce an improved network after retraining.

Error Autocorrelation (plottercorr)

File Edit View Insert Tools Desktop Window Help

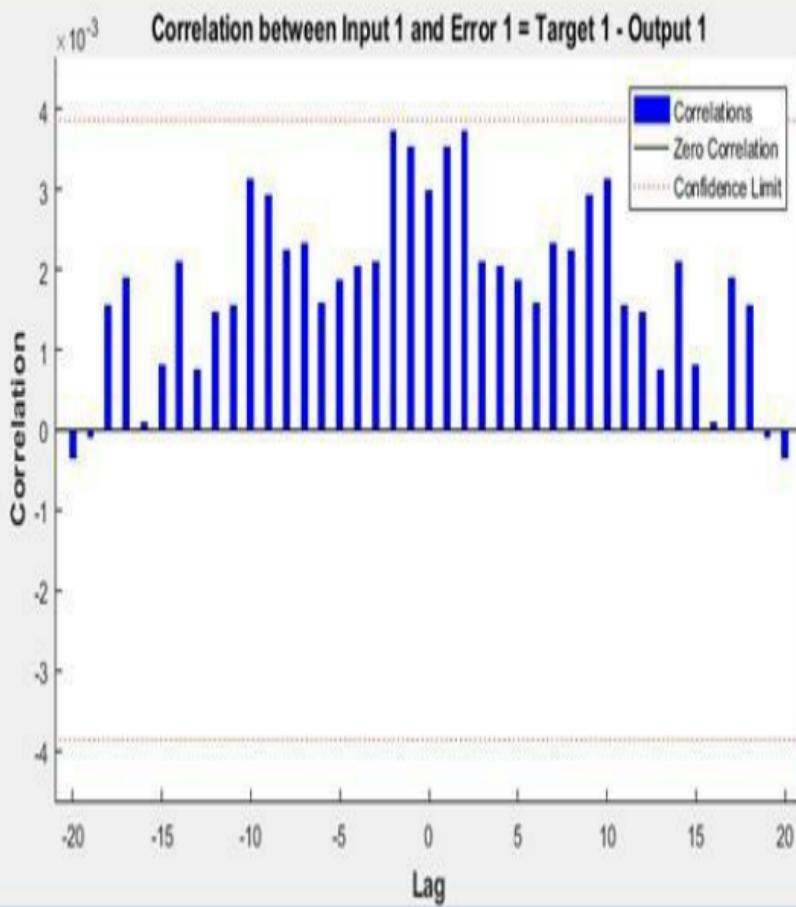


View the input-error cross-correlation

function to obtain additional verification of network performance. Under the **Plots** pane, click **Input-Error Cross-correlation**.

Input-Error Cross-correlation (plotinerrcorr)

File Edit View Insert Tools Desktop Window Help

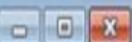


This input-error cross-correlation

function illustrates how the errors are correlated with the input sequence $x(t)$. For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to improve the prediction, perhaps by increasing the number of delays in the tapped delay lines. In this case, all of the correlations fall within the confidence bounds around zero.

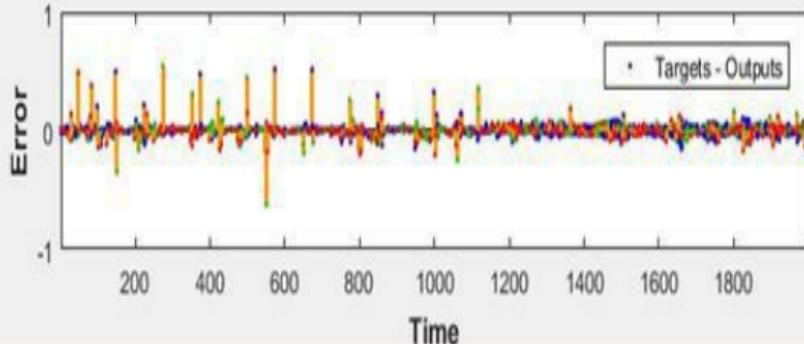
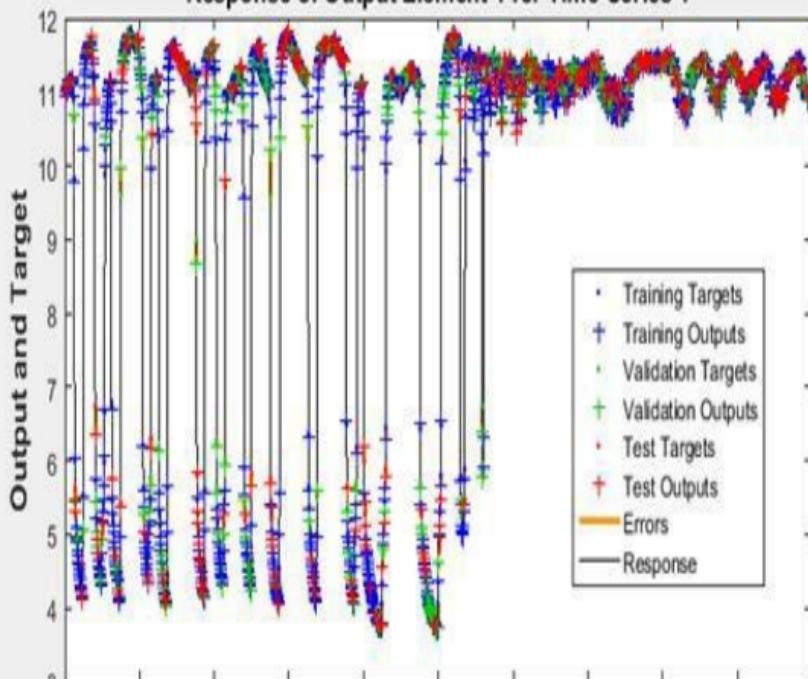
Under **Plots**, click **Time Series Response**. This displays the inputs, targets and errors versus time. It also indicates which time points were selected for training, testing and validation.

Time-Series Response (plotresponse)



File Edit View Insert Tools Desktop Window Help

Response of Output Element 1 for Time-Series 1



Click **Next in the Neural Network Time Series Tool to evaluate the network.**

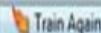


Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

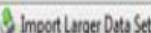
Try training again if a first try did not generate good results or you require marginal improvement.



Increase network size if retraining did not help.



Not working? You may need to use a larger data set.



Optionally perform additional tests



Time step:



[III] Cell column



[III] Matrix column



[III] Matrix row

(none)

(none)

No inputs selected.

No targets selected.



Select inputs and targets, click an improvement button, or click [Next].



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 do any of the following:

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

If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the

number of neurons can improve your results.

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

Use this panel to generate a MATLAB function or Simulink® diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs, or deploy the network with MATLAB Compiler tools and other MATLAB and Simulink code generation tools.



Deploy Solution

Generate deployable versions of your trained neural network.

Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

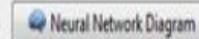
(gensim)



Graphics

Generate a graphical diagram of the neural network:

(network/view)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

Use the buttons on this screen to generate scripts or to save your results.



Save Results

Generate MATLAB scripts, save results and generate diagrams.

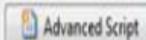
Generate Scripts

Recommended >> Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool:



Generate a script with additional options and example code:



Save Data to Workspace

Save network to MATLAB network object named:

net1

Save performance and data set information to MATLAB struct named:

info

Save outputs to MATLAB matrix named:

output

Save errors to MATLAB matrix named:

error

Save inputs to MATLAB matrix named:

input

Save feedback to MATLAB matrix named:

feedback

Save ALL selected values above to MATLAB struct named:

results

Restore Defaults



Save results and click [Finish].

Neural Network Start

Welcome

Back

Next

Finish

You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

You can also have the network saved as net in the workspace. You can perform additional tests on it or put it to work on new inputs.

After creating MATLAB code and saving your results, click **Finish**.

12.3 USING COMMAND-LINE FUNCTIONS

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created at step 15 of the previous section.

```
% Solve an Autoregression Problem  
with External  
% Input with a NARX Neural Network  
% Script generated by NTSTOOL  
%  
% This script assumes the variables on
```

the right of

% these equalities are defined:

%

% phInputs - input time series.

% phTargets - feedback time series.

inputSeries = phInputs;

targetSeries = phTargets;

% Create a Nonlinear Autoregressive Network with External Input

inputDelays = 1:4;

feedbackDelays = 1:4;

hiddenLayerSize = 10;

net =

narxnet(inputDelays,feedbackDelays,hiddenLayerSize)

% Prepare the Data for Training and
Simulation

% The function PREPARETS prepares
time series data

% for a particular network, shifting time
by the minimum

% amount to fill input states and layer
states.

% Using PREPARETS allows you to
keep your original

% time series data unchanged, while
easily customizing it

% for networks with differing numbers
of delays, with

% open loop or closed loop feedback
modes.

[inputs,inputStates,layerStates,targets] =

...
 prepares(net,inputSeries,
 {},targetSeries);

% Set up Division of Data for Training,
Validation, Testing

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

% Train the Network

[net,tr] =
train(net,inputs,targets,inputStates,layerS

% Test the Network

outputs =
net(inputs,inputStates,layerStates);

```
errors = gsubtract(targets,outputs);  
performance =  
perform(net,targets,outputs)
```

```
% View the Network  
view(net)
```

```
% Plots
```

```
% Uncomment these lines to enable  
various plots.
```

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

```
% figure, plottrainstate(tr)
```

```
% figure, plotregression(targets,outputs)
```

```
% figure, plotresponse(targets,outputs)
```

```
% figure, ploterrcorr(errors)
```

```
% figure, plotinerrcorr(inputs,errors)
```

```
% Closed Loop Network  
% Use this network to do multi-step  
prediction.  
% The function CLOSELOOP replaces  
the feedback input with a direct  
% connection from the output layer.  
netc = closeloop(net);  
netc.name = [net.name ' - Closed Loop'];  
view(netc)  
[xc,xic,aic,tc] =  
preparets(netc,inputSeries,  
{},targetSeries);  
yc = netc(xc,xic,aic);  
closedLoopPerformance =  
perform(netc,tc,yc)  
  
% Early Prediction Network
```

% For some applications it helps to get
the prediction a
% timestep early.
% The original network returns
predicted $y(t+1)$ at the same
% time it is given $y(t+1)$.
% For some applications such as
decision making, it would
% help to have predicted $y(t+1)$ once
 $y(t)$ is available, but
% before the actual $y(t+1)$ occurs.
% The network can be made to return its
output a timestep early
% by removing one delay so that its
minimal tap delay is now
% 0 instead of 1. The new network
returns the same outputs as

% the original network, but outputs are shifted left one timestep.

```
nets = removedelay(net);
```

```
nets.name = [net.name ' - Predict One Step Ahead'];
```

```
view(nets)
```

```
[xs,xis,ais,ts] =
```

```
preparets(nets,inputSeries,
```

```
{},targetSeries);
```

```
ys = nets(xs,xis,ais);
```

```
earlyPredictPerformance =
```

```
perform(nets,ts,ys)
```

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

each of the steps in the script.

The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
load ph_dataset  
inputSeries = phInputs;  
targetSeries = phTargets;
```

Create a network. The NARX network, [narxnet](#), is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This network has two inputs. One is an external input, and the other is a feedback connection from the network

output. (After the network has been trained, this feedback connection can be closed, as you will see at a later step.) For each of these inputs, there is a tapped delay line to store previous values. To assign the network architecture for a NARX network, you must select the delays associated with each tapped delay line, and also the number of hidden layer neurons. In the following steps, you assign the input delays and the feedback delays to range from 1 to 4 and the number of hidden neurons to be 10.

```
inputDelays = 1:4;  
feedbackDelays = 1:4;
```

```
hiddenLayerSize = 10;
```

```
net =
```

```
narxnet(inputDelays,feedbackDelays,hiddenLayerSize)
```

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

Prepare the data for training. When training a network containing tapped delay lines, it is necessary to fill the delays with initial values of the inputs and outputs of the network. There is a toolbox command that facilitates this process - [prepreats](#). This function has three input arguments: the network, the input sequence and the target sequence. The function returns the initial conditions that are needed to fill the tapped delay lines in the network, and modified input and target sequences, where the initial conditions have been removed. You can call the function as follows:

```
[inputs,inputStates,layerStates,targets] =
```

...

```
    prepares(net,inputSeries,  
    {},targetSeries);
```

Set up the division of data.

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

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

Train the network. The network uses the default Levenberg-Marquardt algorithm ([trainlm](#)) for training. For problems in which Levenberg-Marquardt does not produce as accurate results as desired,

or for large data problems, consider setting the network training function to Bayesian Regularization ([trainbr](#)) or Scaled Conjugate Gradient ([trainscg](#)), respectively, with either

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

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

To train the network, enter:

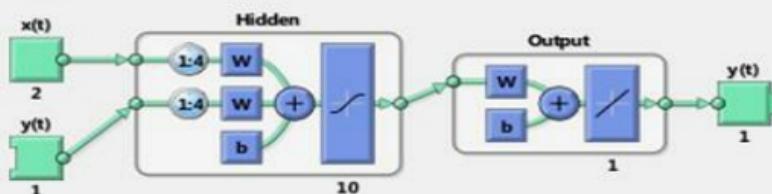
```
[net,tr] =
```

```
train(net,inputs,targets,inputStates,layerS)
```

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**.

Neural Network Training (nntraintool)

Neural Network



Algorithms

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

Progress

Epoch:	0	44 iterations	1000
Time:		0:00:00	
Performance:	79.1	0.00219	0.00
Gradient:	134	0.0187	1.00e-07
Mu:	0.00100	1.00e-05	1.00e+10
Validation Checks:	0	6	6

Plots

- Performance** (plotperform)
- Training State (plottrainstate)
- Error Histogram (ploterrhist)
- Regression (plotregression)
- Time-Series Response (plotresponse)
- Error Autocorrelation (ploterrcorr)
- Input-Error Cross-correlation (plotinerrcorr)

Plot Interval: 1 epochs



Validation stop.

This training stopped when the validation error increased for six iterations, which occurred at iteration 44.

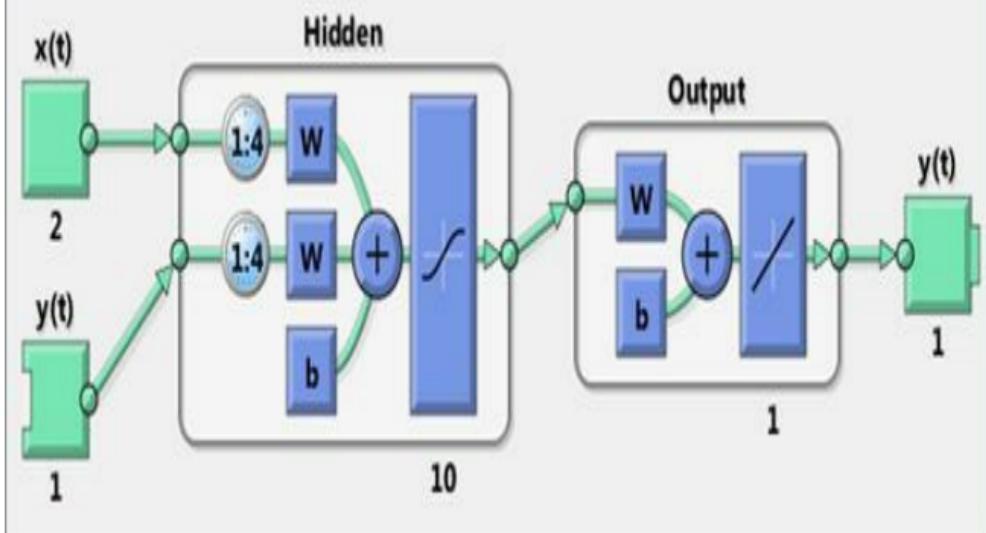
Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors and overall performance. Note that to simulate a network with tapped delay lines, you need to assign the initial values for these delayed signals.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

```
outputs =  
net(inputs,inputStates,layerStates);  
errors = gsubtract(targets,outputs);  
performance =  
perform(net,targets,outputs)
```

performance =

0.0042

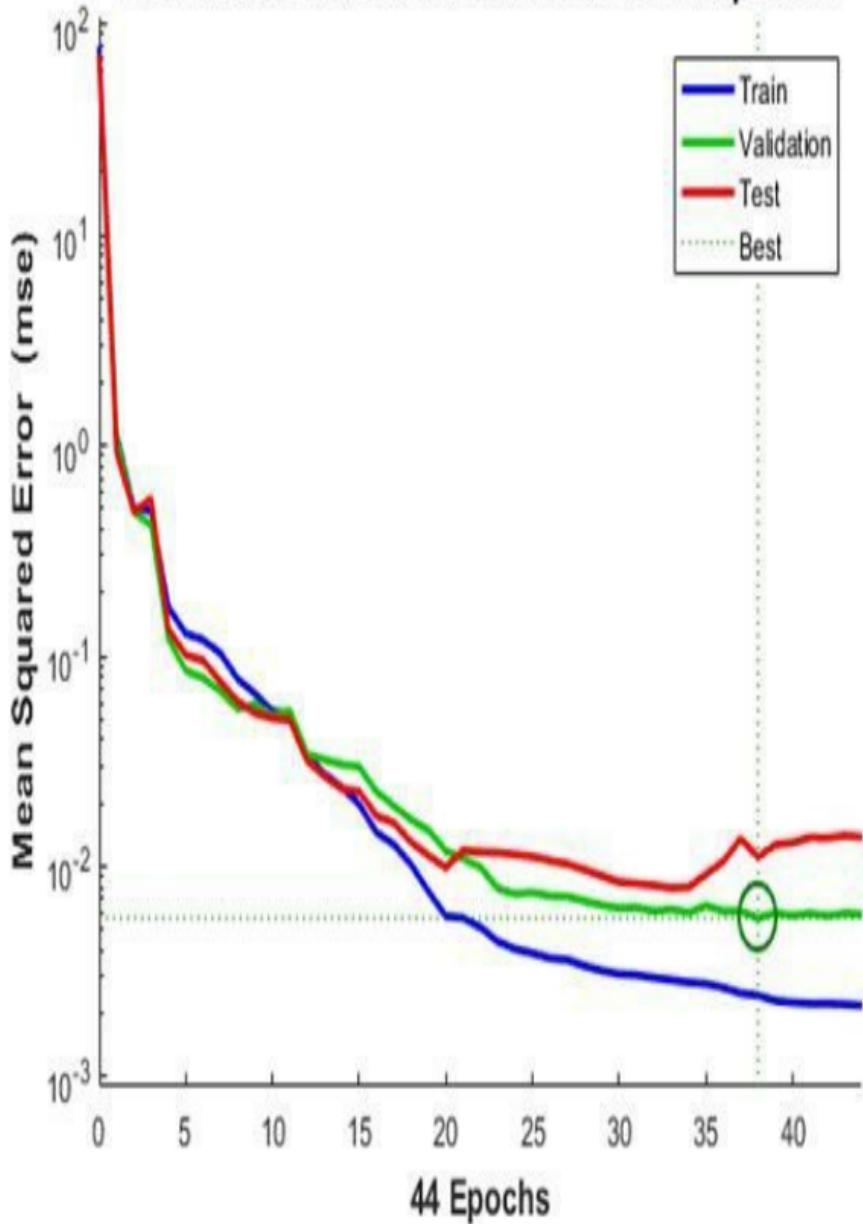
View the network diagram.
view(net)



Plot the performance training record to check for potential overfitting.

figure, plotperform(tr)

Best Validation Performance is 0.0056917 at epoch 38



This figure shows that training, validation and testing errors all decreased until iteration 64. It does not appear that any overfitting has occurred, because neither testing nor validation error increased before iteration 64.

All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) is it transformed to closed loop for multistep-ahead prediction.

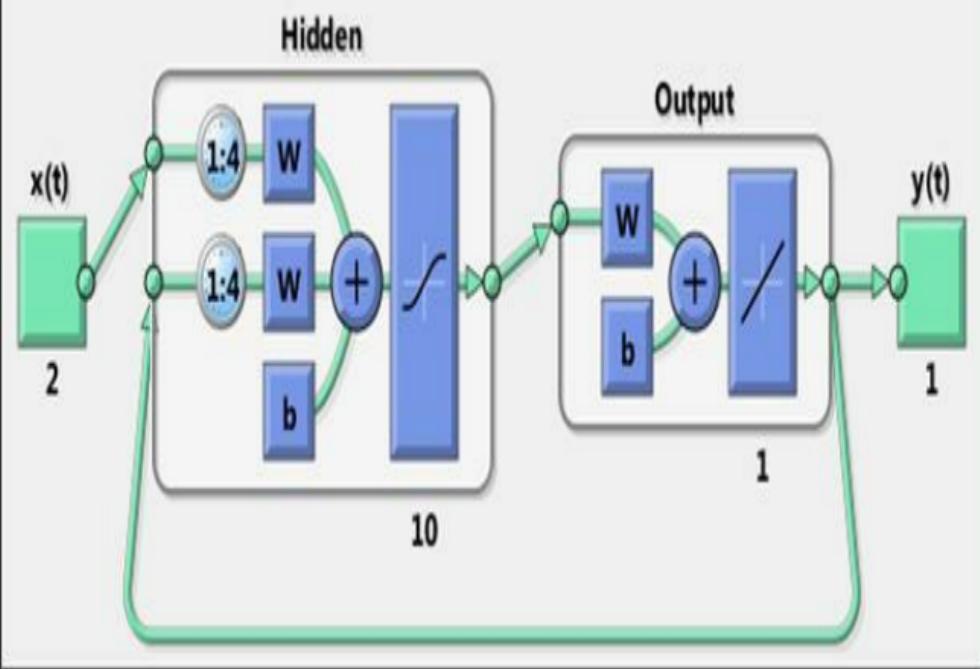
Likewise, the Rvalues in the GUI are computed based on the open-loop training results.

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

```
netc = closeloop(net);
netc.name = [ net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] =
prepares(netc,inputSeries,
{},targetSeries);
yc = netc(xc,xic,aic);
perf = perform(netc,tc,yc)
```

perf =

2.8744

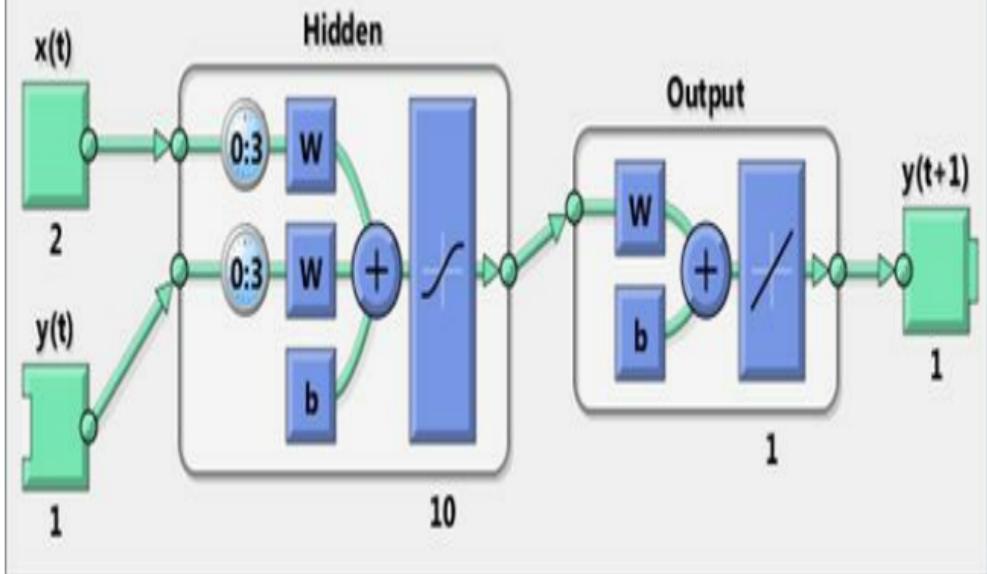


Remove a delay from the network, to get the prediction one time step early.

```
nets = removedelay(net);
nets.name = [net.name ' - Predict One
Step Ahead'];
view(nets)
```

```
[xs,xis,ais,ts] =  
prepares(nets,inputSeries,  
{ },targetSeries);  
ys = nets(xs,xis,ais);  
earlyPredictPerformance =  
perform(nets,ts,ys)  
earlyPredictPerformance =
```

0.0042



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

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

- Reset the initial network weights and biases to new values with [init](#) and train again
- Increase the number of hidden neurons or the number of delays.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training

algorithm.

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

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

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

Each time a neural network is trained,

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

There are several other techniques for improving upon initial solutions if higher accuracy is desired.

Chapter 13

BIBLIOGRAPHY

13.1 NEURAL NETWORK BIBLIOGRAPHY

- [**Batt92**] Battiti, R., "First and second order methods for learning: Between steepest descent and Newton's method," *Neural Computation*, Vol. 4, No. 2, 1992, pp. 141–166.
- [**Beal72**] Beale, E.M.L., "A derivation of conjugate gradients," in F.A. Lootsma, Ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.
- [**Bren73**] Brent, R.P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall,

1973.

[Caud89] Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[CaBu92] Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations*, Vols. 1 and 2, Cambridge, MA: The MIT Press, 1992.

This is a two-volume workbook designed to give students "hands on" experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear, and helpful in understanding a field that traditionally has been buried in mathematics.

[Char92] Charalambous, C., "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEEE Proceedings*, Vol. 139, No. 3, 1992, pp. 301–310.

[ChCo91] Chen, S., C.F.N. Cowan, and

P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, 1991, pp. 302–309.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[ChDa99] Chengyu, G., and K. Danai, "Fault diagnosis of the IFAC Benchmark Problem with a model-based recurrent neural network," *Proceedings of the 1999 IEEE International Conference on Control Applications*, Vol. 2, 1999,

pp. 1755–1760.

[DARP88] *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

[DeHa01a] De Jesús, O., and M.T.

Hagan, "Backpropagation Through Time for a General Class of Recurrent Network," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2638–2642.

[DeHa01b] De Jesús, O., and M.T. Hagan, "Forward Perturbation Algorithm for a General Class of Recurrent Network," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2626–2631.

[DeHa07] De Jesús, O., and M.T. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," *IEEE Transactions on Neural Networks*,

This paper provides detailed algorithms for the calculation of gradients and Jacobians for arbitrarily-connected neural networks. Both the backpropagation-through-time and real-time recurrent learning algorithms are covered.

[DeSc83] Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[DHH01] De Jesús, O., J.M. Horn, and M.T. Hagan, "Analysis of Recurrent Network Training and Suggestions for

Improvements," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2632–2637.

[Elma90] Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

[FeTs03] Feng, J., C.K. Tse, and F.C.M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*,

Vol. 50, No. 7, 2003, pp. 954–957.

[FIRe64] 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*, Dordrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999*

American Control Conference, San Diego, CA, 1999, pp. 1642–1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

[HaMe94] Hagan, M.T., and M. Menhaj, "Training feed-forward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm

for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

[HaRu78] Harrison, D., and Rubinfeld, D.L., "Hedonic prices and the demand for clean air," *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

[HDB96] Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing,

1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor's guide, and transparency overheads for teaching.

[HDH09] Horn, J.M., O. De Jesús and M.T. Hagan, "Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance," IEEE Transactions on Neural Networks, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

[Hebb49] Hebb, D.O., *The Organization of Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

[Himm72] Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

- [HuSb92] Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System — A Survey," *Automatica*, Vol. 28, 1992, pp. 1083–1112.
- [JaRa04] Jayadeva and S.A.Rahman, "A neural network with $O(N)$ neurons for ranking N numbers in $O(1/N)$ time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044–2051.
- [Joll86] Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.
- [KaGr96] Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen,

and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657–664.

[Koho87] Kohonen, T., *Self-Organization and Associative Memory*, 2nd Edition, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

[Koho97] Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A.N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405–1422.

This paper discusses a class of neural networks described by first-order linear

differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User's Guide*.

[Lipp87] Lippman, R.P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, 1987, pp. 4–22.

This paper gives an introduction to the

field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

[MacK92] MacKay, D.J.C., "Bayesian interpolation," *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447.

[Marq63] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

[McPi43] McCulloch, W.S., and W.H.

Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, 1943, pp. 115–133.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

[MeJa00] Medsker, L.R., and L.C. Jain, *Recurrent neural networks: design and applications*, Boca Raton, FL: CRC Press, 2000.

[Moll93] Moller, M.F., "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, Vol. 6, 1993, pp. 525–533.

[MuNe92] Murray, R., D. Neumerkel, and D. Sbarbaro, "Neural Networks for Modeling and Control of a Non-linear Dynamic System," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404–409.

[NaMu97] Narendra, K.S., and S. Mukhopadhyay, "Adaptive Control Using Neural Networks and Approximate Models," *IEEE Transactions on Neural Networks*, Vol. 8, 1997, pp. 475–485.

[NaPa91] Narendra, Kumpati S. and Kannan Parthasarathy, "Learning Automata Approach to Hierarchical Multiobjective Analysis," *IEEE Transactions on Systems, Man and*

Cybernetics, Vol. 20, No. 1,
January/February 1991, pp. 263–272.

[NgWi89] Nguyen, D., and B. Widrow,
"The truck backer-upper: An example of
self-learning in neural
networks," *Proceedings of the
International Joint Conference on
Neural Networks*, Vol. 2, 1989, pp. 357–
363.

This paper describes a two-layer
network that first learned the truck
dynamics and then learned how to back
the truck to a specified position at a
loading dock. To do this, the neural
network had to solve a highly nonlinear
control systems problem.

[NgWi90] Nguyen, D., and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1990, pp. 21–26.

Nguyen and Widrow show that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-

Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

[Powe77] Powell, M.J.D., "Restart procedures for the conjugate gradient method," *Mathematical Programming*, Vol. 12, 1977, pp. 241–254.

[Pulu92] Purdie, N., E.A. Lucas, and M.B. Talley, "Direct measure of total cholesterol and its distribution among major serum lipoproteins," *Clinical Chemistry*, Vol. 38, No. 9, 1992, pp. 1645–1647.

[RiBr93] Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP

algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[Robin94] Robinson, A.J., "An application of recurrent nets to phone probability estimation," *IEEE Transactions on Neural Networks*, Vol. 5 , No. 2, 1994.

[RoJa96] Roman, J., and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, Vol. 2, 1996, pp. 454–460.

[Rose61] Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt's results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

[RuHi86a] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," in D.E. Rumelhart and J.L. McClelland, Eds., *Parallel Data Processing, Vol. 1*, Cambridge, MA: The M.I.T. Press, 1986, pp. 318–362.

This is a basic reference on backpropagation.

[**RuHi86b**] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, 1986, pp. 533–536.

[**RuMc86**] Rumelhart, D.E., J.L. McClelland, and the PDP Research Group, Eds., *Parallel Distributed Processing, Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of

publication.

[Scal85] Scales, L.E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[SoHa96] Soloway, D., and P.J. Haley, "Neural Generalized Predictive Control," *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277–281.

[VoMa88] Vogl, T.P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, Vol. 59, 1988, pp. 256–264.

Backpropagation learning can be

speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

[WaHa89] Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, 1989, pp. 328–339.

[Wass93] Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

[WeGe94] Weigend, A. S., and N. A. Gershenfeld, eds., *Time Series*

Prediction: Forecasting the Future and Understanding the Past, Reading, MA: Addison-Wesley, 1994.

[WiHo60] Widrow, B., and M.E. Hoff,
"Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.

[WiSt85] Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

