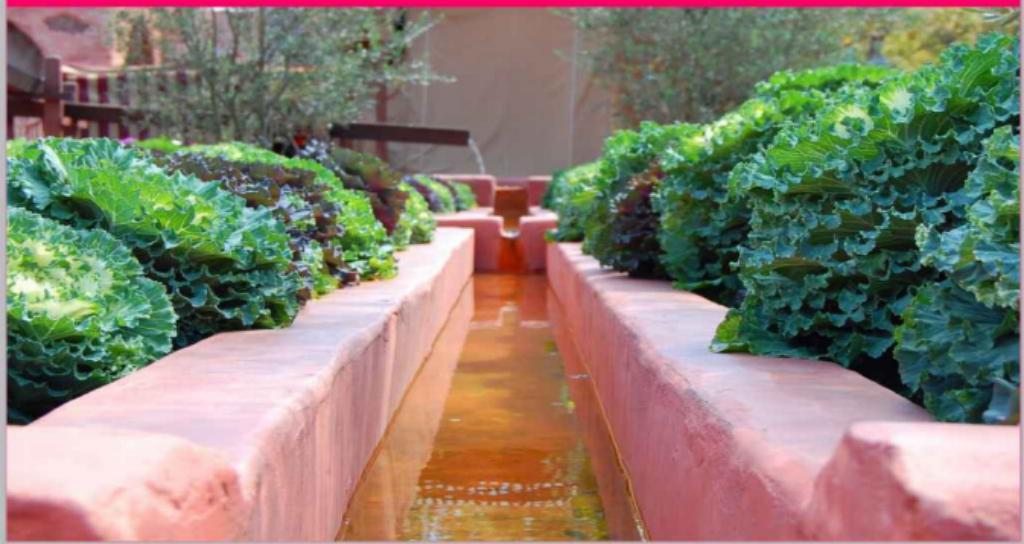


NEURAL NETWORK ARCHITECTURES. EXAMPLES using MATLAB

J. SMITH



NEURAL NETWORK ARCHITECTURES EXAMPLES USING MATLAB

J. SMITH

CONTENTS

NEURAL NETWORKS

1.1 INTRODUCTION

1.2 NEURAL NETWORK STRUCTURE

1.2.1 A simple neuron

1.2.2 A more complicated neuron

1.3 ARCHITECTURE OF NEURAL NETWORKS

1.3.1 Feed-forward networks

1.3.2 Feedback networks

1.3.3 Network layers

1.3.4 Perceptrons

1.4 THE LEARNING PROCESS

1.4.1 Transfer Function

1.4.2 An Example to illustrate the above teaching procedure

1.4.3 The Back-Propagation Algorithm

1.4.4 The back-propagation Algorithm - a mathematical approach

NEURAL NETWORKS WITH MATLAB

2.1 NEURAL NETWORK TOOLBOX

2.2 USING NEURAL NETWORK TOOLBOX

2.3 AUTOMATIC SCRIPT GENERATION

2.4 NEURAL NETWORK TOOLBOX APPLICATIONS

2.5 NEURAL NETWORK DESIGN STEPS

WORKFLOW FOR NEURAL NETWORK DESIGN

3.1 INTRODUCTION

3.2 FOUR LEVELS OF NEURAL NETWORK DESIGN

3.3 NEURAL NETWORK ARCHITECTURES

3.3.1 One Layer of Neurons

3.3.2 Multiple Layers of Neurons

3.3.3 Input and Output Processing Functions

3.4 MULTILAYER NEURAL NETWORKS AND BACKPROPAGATION TRAINING

3.5 MULTILAYER NEURAL NETWORK ARCHITECTURE

3.5.1 Neuron Model (`logsig`, `tansig`, `purelin`)

3.5.2 Feedforward Neural Network

3.6 UNDERSTANDING NEURAL NETWORK TOOLBOX DATA STRUCTURES

3.6.1 Simulation with Concurrent Inputs in a Static Network

3.6.2 Simulation with Sequential Inputs in a Dynamic Network

3.6.3 Simulation with Concurrent Inputs in a Dynamic Network

ADAPTIVE NEURAL NETWORK FILTERS

4.1 INTRODUCTION

4.2 ADAPTIVE FUNCTIONS:
ADAPT

4.3 LINEAR NEURON MODEL

4.4 ADAPTIVE LINEAR NETWORK

ARCHITECTURE

4.4.1 Single ADALINE (linearlayer)

4.4.2 linearlayer

4.5 LEAST MEAN SQUARE ERROR

4.6 LMS ALGORITHM (LEARNWH)

4.6.1 learnwh

4.6.2 maxlinlr

4.7 ADAPTIVE FILTERING
(ADAPT)

4.7.1 Tapped Delay Line

4.7.2 Adaptive Filter

4.7.3 Adaptive Filter Example

4.7.4 Prediction Example

4.7.5 Noise Cancelation Example

4.7.6 Multiple Neuron Adaptive

Filters

4.8 ADAPTATIVE FILTER EXAMPLES

4.8.1 Pattern Association Showing Error Surface

4.8.2 Training a Linear Neuron

4.8.3 Adaptive Noise Cancellation

4.8.4 Linear Fit of Nonlinear Problem

4.8.5 Underdetermined Problem

4.8.6 Linearly Dependent Problem

4.8.7 Too Large a Learning Rate

PERCEPTRON NEURAL NETWORKS

5.1 INTRODUCTION

5.2 NEURON MODEL

5.3 PERCEPTRON ARCHITECTURE

5.4 CREATE A PERCEPTRON

5.5 PERCEPTRON LEARNING RULE (LEARNP)

5.6 TRAINING (TRAIN)

5.7 LIMITATIONS AND CAUTIONS

5.8 PERCEPTRON EXAMPLES

5.8.1 Classification with a 2-Input Perceptron

5.8.2 Outlier Input Vectors

5.8.3 Normalized Perceptron Rule

5.8.4 Linearly Non-separable Vectors

RADIAL BASIS NEURAL NETWORKS

6.1 RADIAL BASIS FUNCTION

NETWORK

6.2 NEURON MODEL

6.3 NETWORK ARCHITECTURE

6.4 EXACT DESIGN (NEWRBE)

6.5 MORE EFFICIENT DESIGN (NEWRB)

6.6 RADIAL BASIS EXAMPLES

6.6.1 Radial Basis Approximation

6.6.2 Radial Basis Underlapping Neurons

6.6.3 GRNN Function Approximation

6.6.4 PNN Classification

PROBABILISTIC, GENERALIZED REGRESSION AND LVQ NEURAL NETWORKS

7.1 PROBABILISTIC NEURAL NETWORKS

7.1.1 Network Architecture

7.1.2 Design (newpnn)

7.2 GENERALIZED REGRESSION NEURAL NETWORKS

7.2.1 Network Architecture

7.2.2 Design (newgrnn)

7.3 LEARNING VECTOR QUANTIZATION (LVQ) NEURAL NETWORKS

7.3.1 Architecture

7.3.2 Creating an LVQ Network

7.3.3 LVQ1 Learning Rule (learnlv1)

7.3.4 Training

7.3.5 Supplemental LVQ2.1 Learning

Rule (learnlv2)

HOPFIELD AND LINEAR NEURAL NETWORKS

8.1 LINEAR NEURAL NETWORKS

8.1.1 Neuron Model

8.1.2 Network Architecture

8.1.3 Create a Linear Neuron (linearlayer)

8.1.4 Least Mean Square Error

8.1.5 Linear System Design (newlind)

8.1.6 Linear Networks with Delays

8.1.7 LMS Algorithm (learnwh)

8.1.8 Linear Classification (train)

8.1.9 Limitations and Cautions

8.2 HOPFIELD NEURAL NETWORK

8.2.1 Fundamentals

8.2.2 Architecture

8.2.3 Design (newhop)

8.2.4 Summary

8.3 LINEAR PREDICTION DESIGN EXAMPLE

8.3.1 Defining a Wave Form

8.3.2 Setting up the Problem for a Neural Network

8.3.3 Designing the Linear Layer

8.3.4 Testing the Linear Layer

8.4 ADAPTIVE LINEAR PREDICTION EXAMPLE

8.4.1 Defining a Wave Form

8.4.2 Setting up the Problem for a Neural Network

8.4.3 Creating the Linear Layer

8.4.4 Adapting the Linear Layer

8.5 HOPFIELD TWO NEURON
DESIGN EXAMPLE

8.6 HOPFIELD UNSTABLE
EQUILIBRIA EXAMPLE

8.7 HOPFIELD THREE NEURON
DESIGN EXAMPLE

8.8 HOPFIELD SPURIOUS STABLE
POINTS EXAMPLE

CUSTOM NEURAL NETWORKS

9.1 CREATE NEURAL NETWORK
OBJECT

9.2 CONFIGURE NEURAL
NETWORK INPUTS AND OUTPUTS

9.3 CREATE AND TRAIN CUSTOM

NEURAL NETWORK ARCHITECTURES

9.3.1 Custom Network

9.3.2 Network Definition

9.3.3 Network Behavior

BIBLIOGRAPHY

10.1 NEURAL NETWORK BIBLIOGRAPHY

Chapter 1

NEURAL NETWORKS

1.1 INTRODUCTION

Neural networks theory is inspired from the natural neural network of human nervous system. Is possible define a neural network as a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel

structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

The human brain is composed of 100 billion nerve cells called neurons. They are connected to

other thousand cells by Axons. Stimuli from external environment or inputs from sensory organs are accepted by dendrites. These inputs create electric impulses, which quickly travel through the neural network. A neuron can then send the message to other neuron to handle the issue or does not send it forward. ANNs are composed of multiple nodes, which imitate biological neurons of human brain. The neurons are connected by links and they interact with each other. The nodes can take input data and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is called its activation or node

value.

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras. Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed

the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer

techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.

Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.

3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

Neural networks take a different approach to problem solving than that of

conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected

processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is solved must be known and stated in small unambiguous instructions. These

instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

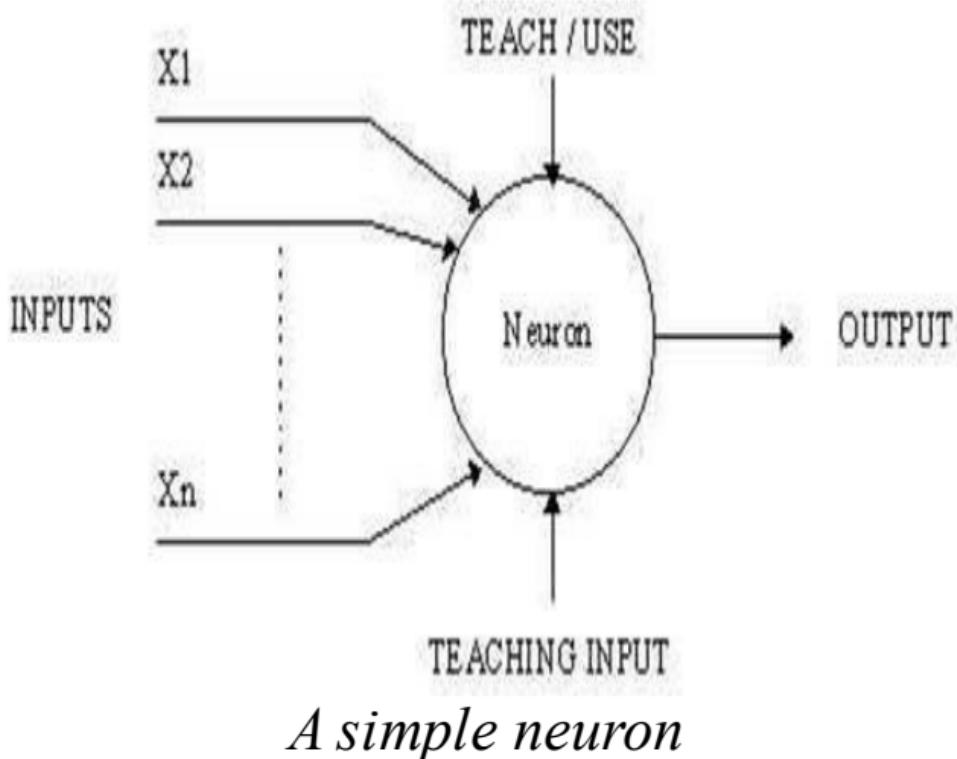
Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is

used to supervise the neural network) in order to perform at maximum efficiency.

1.2 NEURAL NETWORK STRUCTURE

1.2.1 A simple neuron

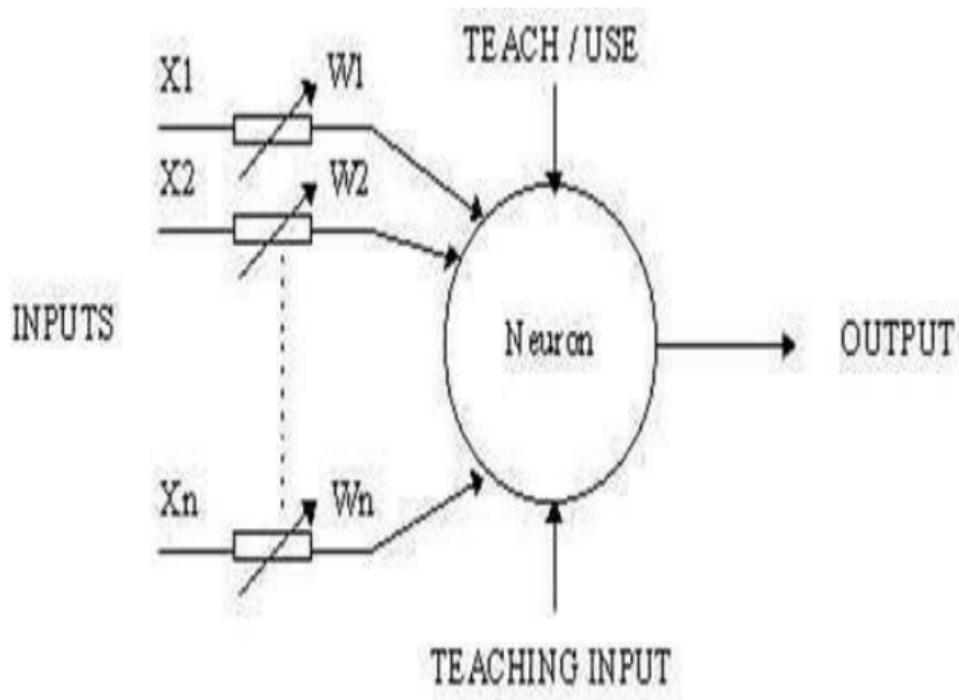
An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



1.2.2 A more complicated neuron

The previous neuron doesn't do anything that conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the

neuron fires. In any other case the neuron does not fire.



A compose neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \dots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The composed neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed-forward networks and the latter in feedback networks.

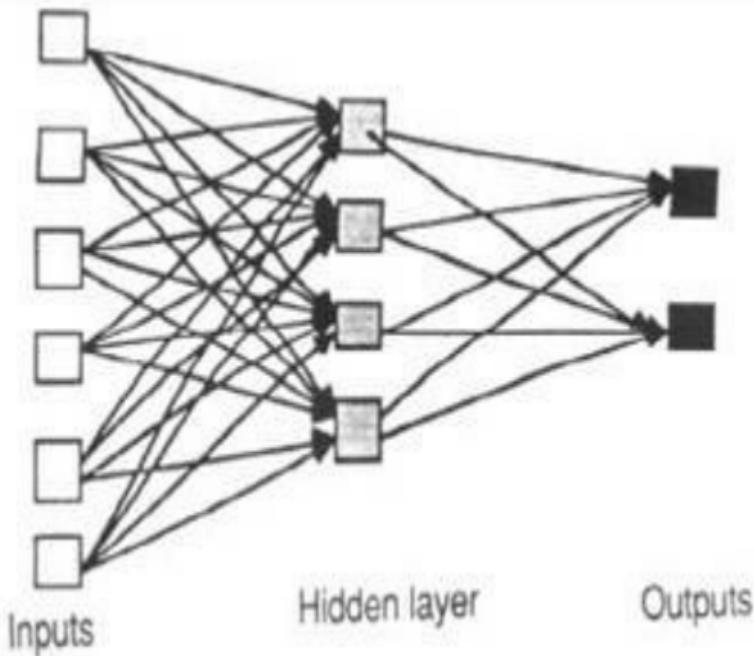
1.3 ARCHITECTURE OF NEURAL NETWORKS

1.3.1 Feed-forward networks

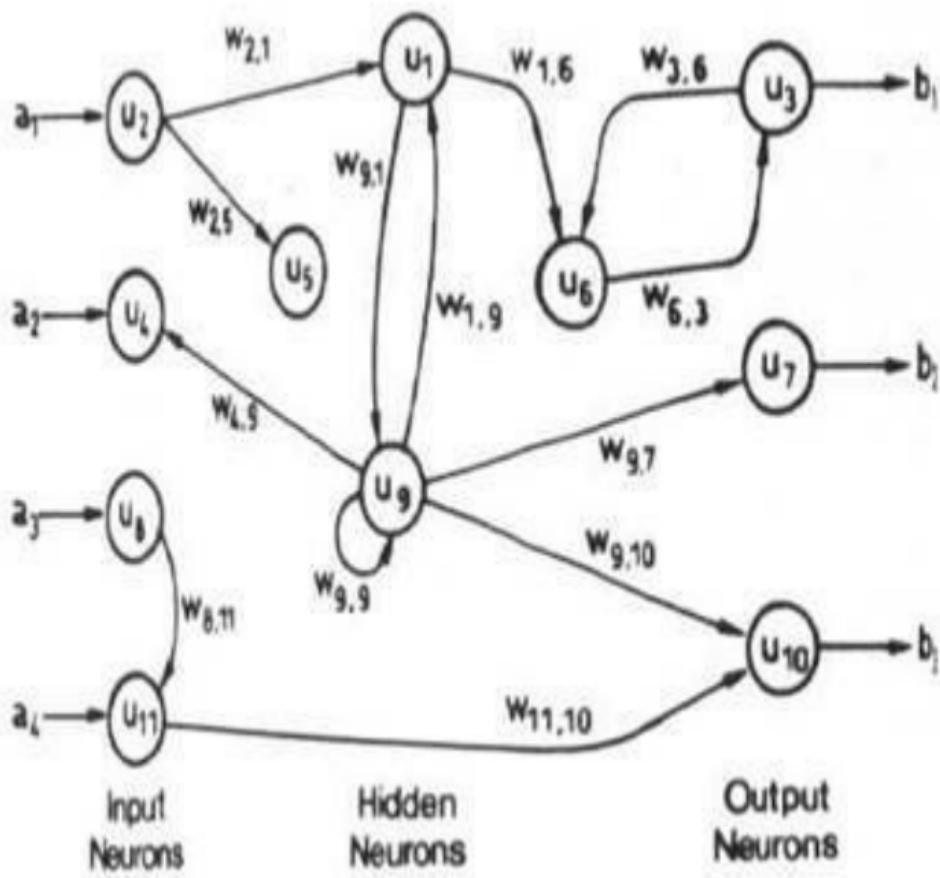
Feed-forward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.

1.3.2 Feedback networks

Feedback networks can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.



An example of a simple feedforward network



An example of a composed network

1.3.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of "**output**" units.

The activity of the input units represents the raw information that is fed into the network.

- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input

and the hidden units.

- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

1.3.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) turns out to be an compose model (neuron with weighted inputs) with some additional, fixed, pre-processing. Units labelled A_1, A_2, A_j, A_p are called association units and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot.

more.

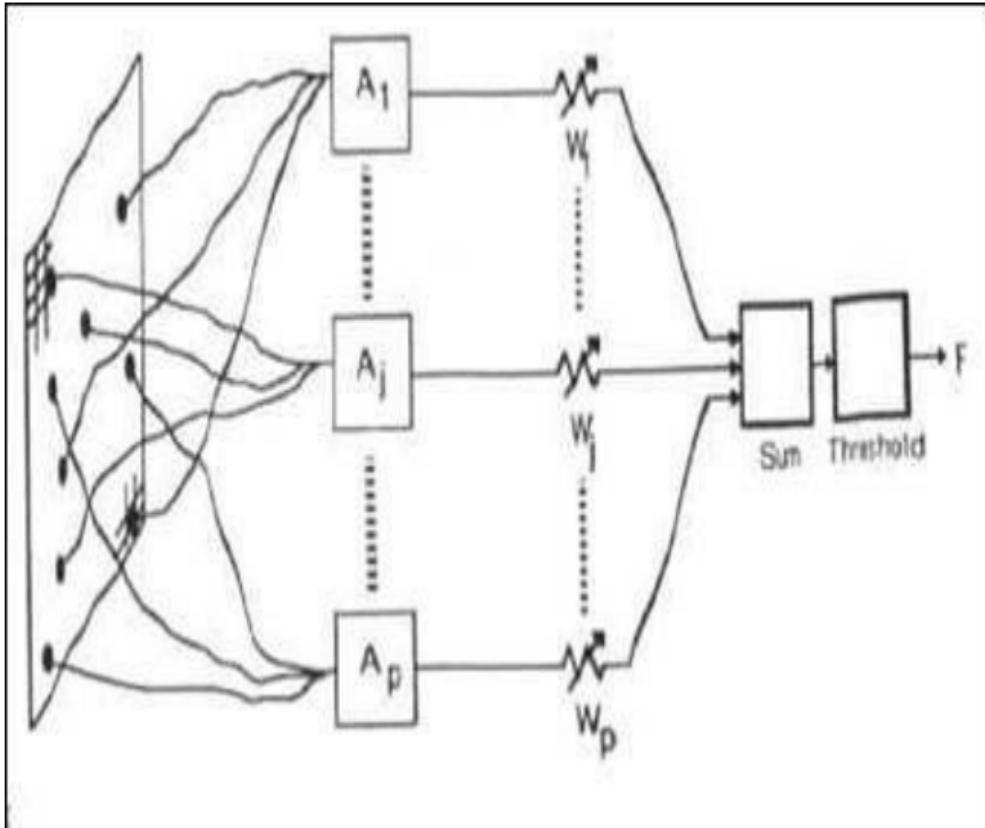


Figure 4.4

In 1969 Minsky and Papert wrote a book in which they described the

limitations of single layer Perceptrons. The impact that the book had was tremendous and caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that single layer perceptrons could not do some basic pattern recognition operations like determining the parity of a shape or determining whether a shape is connected or not. What they did not realised, until the 80's, is that given the appropriate training, multilevel perceptrons can do these operations.

1.4 THE LEARNING PROCESS

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associative mapping can generally be broken down into two mechanisms:

- *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is

used to provide pattern completion, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.

 *hetero-association*: is related to two recall mechanisms:

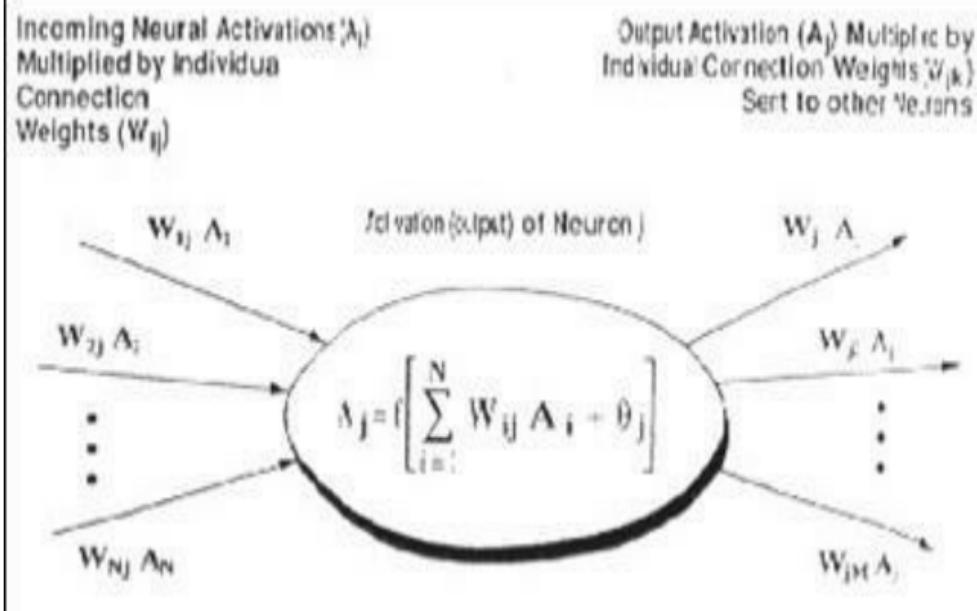
- ➊ *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and

- ➋ *interpolative* recall, where the output pattern is a similarity dependent

interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.

• **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in associative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network possesses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.



Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie $dW/dt=0$. In such networks, the weights are fixed a priori according to the problem to solve.
- **adaptive networks** which are able to change their weights, ie $dW/dt \neq 0$.

All learning methods used for adaptive neural networks can be classified into two major categories:

Supervised learning which

incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.

An important issue concerning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known

method, which is common to many learning paradigms is the least mean square (LMS) convergence.

• **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian learning and competitive learning.

Ano2.2 From Human Neurones to Artificial Neuronesthe aspect of learning concerns the distinction or not of a separate phase, during which the network is trained, and a subsequent

operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas unsupervised learning is performed on-line.

1.4.1 Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)
- threshold
- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at

one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do linear or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections

determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.

3. We change the weight of each connection so that the network produces a better approximation of the desired output.

1.4.2 An Example to illustrate the above teaching procedure

Assume that we want a network to recognise hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink in a small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a number of hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output

units.

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the error, which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce the error. We repeat this training process for many different images of each different images of each kind of digit until the network classifies every image correctly.

To implement this procedure we need to calculate the error derivative for

the weight (EW) in order to change the weight by an amount that is proportional to the rate at which the error changes as the weight is changed. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But that method is inefficient because it requires a separate perturbation for each of the many weights.

Another way to calculate the EW is to use the Back-propagation algorithm which is described below, and has become nowadays one of the most important tools for training neural networks. It was developed independently by two teams, one

(Fogelman-Soulie, Gallinari and Le Cun) in France, the other (Rumelhart, Hinton and Williams) in U.S.

1.4.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (EW). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the EW.

The back-propagation algorithm is

easiest to understand if all the units in the network are linear. The algorithm computes each EW by first computing the EA, the rate at which the error changes as the activity level of a unit is changed. For output units, the EA is simply the difference between the actual and the desired output. To compute the EA for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the EAs of those output units and add the products. This sum equals the EA for the chosen hidden unit. After calculating all the EAs in the hidden

layer just before the output layer, we can compute in like fashion the EAs for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the EA has been computed for a unit, it is straight forward to compute the EW for each incoming connection of the unit. The EW is the product of the EA and the activity through the incoming connection.

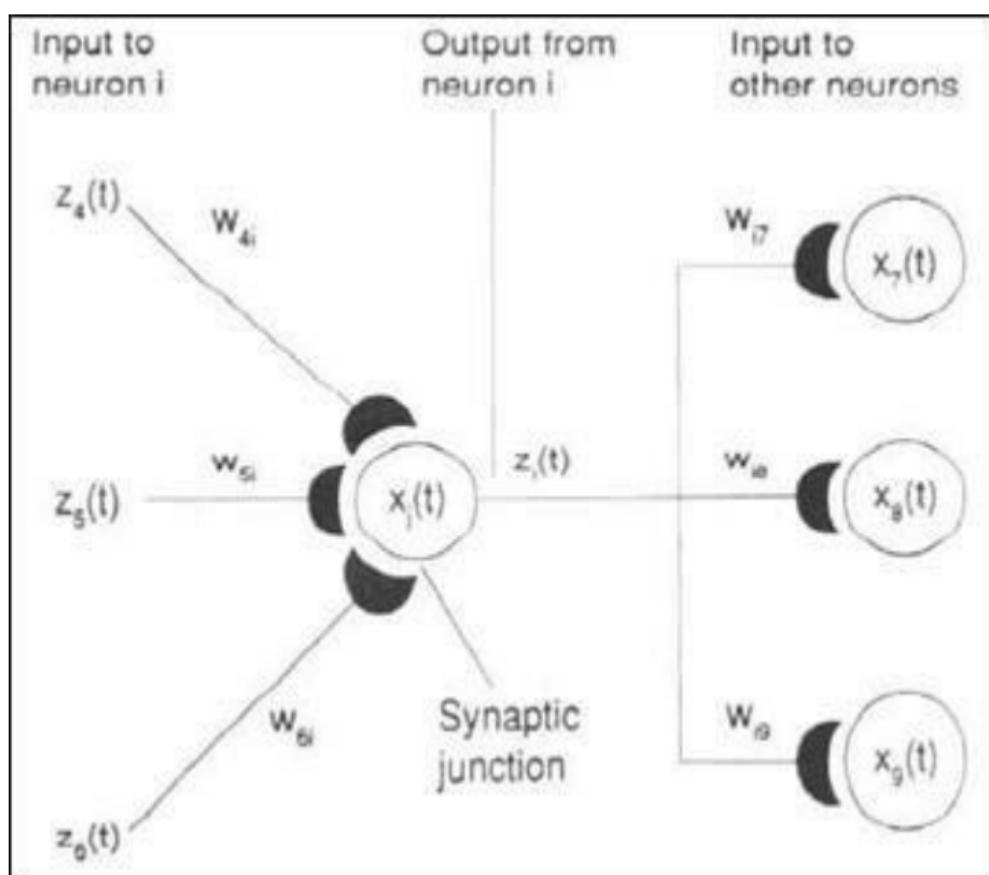
Note that for non-linear units, the back-propagation algorithm includes an extra step. Before back-propagating, the EA must be converted into the EI, the rate at which the error changes as the

total input received by a unit is changed.

1.4.4 The back-propagation Algorithm - a mathematical approach

Units are connected to one another. Connections correspond to the edges of the underlying directed graph. There is a real number associated with each connection, which is called the weight of the connection. We denote by W_{ij} the weight of the connection from unit u_i to unit u_j . It is then convenient to represent the pattern of connectivity in the network by a weight matrix W whose elements are the weights W_{ij} . Two types of connection are usually distinguished: excitatory and inhibitory. A positive weight represents an excitatory

connection whereas a negative weight represents an inhibitory connection. The pattern of connectivity characterises the architecture of the network.



A unit in the output layer determines its activity by following a two step procedure.

- First, it computes the total weighted input x_j , using the formula:

$$X_j = \sum_i y_i W_{ij}$$

where y_i is the activity level of the j th unit in the previous layer and W_{ij} is the weight of the connection between the i th and the j th unit.

- Next, the unit calculates the activity y_j using some function of the total weighted input. Typically we use the sigmoid function:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

Once the activities of all output units have been determined, the network computes the error E, which is defined by the expression:

$$E = \frac{1}{2} \sum_i (y_i - d_i)^2$$

where y_j is the activity level of the j th unit in the top layer and d_j is the desired output of the j th unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes

as the activity of an output unit is changed. This error derivative (EA) is the difference between the actual and the desired activity.

$$EA_j = \frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity (EI) is the answer from step 1 multiplied by the rate at which the output of a unit changes as its total input is changed.

$$EI_j = \frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = EA_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity (EW) is the answer from step 2 multiplied by the activity level of the unit from which the connection emanates.

$$EW_{ij} = \frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta a_j} \times \frac{\delta a_j}{\delta w_{ij}} = EI_j y_i$$

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the

activity of a unit in the previous layer changes, it affects the activities of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these separate effects on output units. But each effect is simple to calculate. It is the answer in step 2 multiplied by the weight on the connection to that output unit.

$$EA_i = \frac{\delta E}{\delta y_i} = \sum_j \frac{\delta E}{\delta x_j} \times \frac{\partial x_j}{\partial y_i} = \sum_j EI_j W_{ij}$$

By using steps 2 and 4, we can convert the EAs of one layer of units into EAs for the previous layer. This procedure can be repeated to get the EAs for as

many previous layers as desired. Once we know the EA of a unit, we can use steps 2 and 3 to compute the EWs on its incoming connections.

Chapter 2

NEURAL NETWORKS WITH MATLAB

2.1 NEURAL NETWORK TOOLBOX

MATLAB has the tool Neural Network Toolbox that provides algorithms, functions, and apps to create, train, visualize, and simulate neural networks. You can perform classification, regression, clustering, dimensionality reduction, time-series forecasting, and dynamic system modeling and control.

The toolbox includes convolutional neural network and autoencoder deep learning algorithms for image classification and feature learning tasks. To speed up training of large data sets,

you can distribute computations and data across multicore processors, GPUs, and computer clusters using Parallel Computing Toolbox.

The more important features are the following:

- Deep learning, including convolutional neural networks and autoencoders
- Parallel computing and GPU support for accelerating training (with Parallel Computing Toolbox)
- Supervised learning algorithms, including multilayer, radial basis, learning vector quantization (LVQ),

time-delay, nonlinear autoregressive (NARX), and recurrent neural network (RNN)

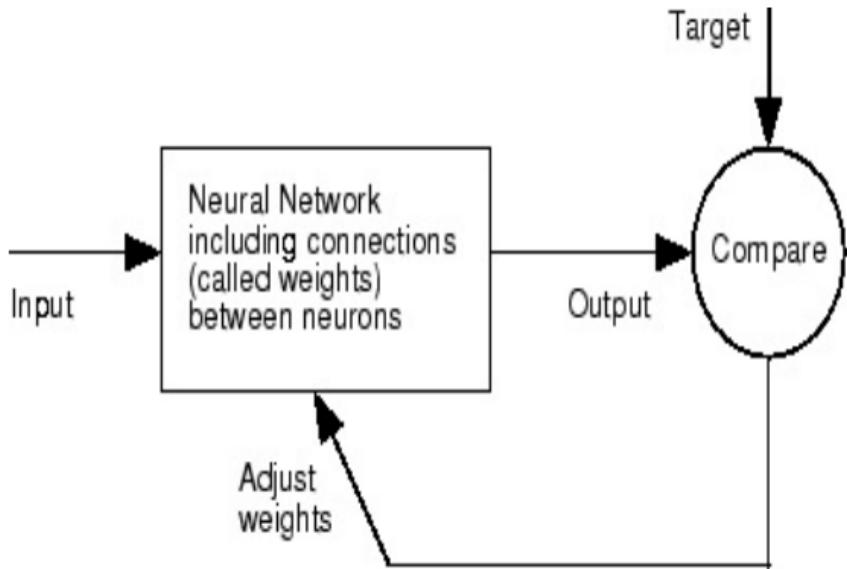
- Unsupervised learning algorithms, including self-organizing maps and competitive layers
- Apps for data-fitting, pattern recognition, and clustering
- Preprocessing, postprocessing, and network visualization for improving training efficiency and assessing network performance
- Simulink® blocks for building and evaluating neural networks and for control systems applications

Neural networks are composed of

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

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

Typically, many such input/target pairs are needed to train a network.



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

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

2.2 USING NEURAL NETWORK TOOLBOX

There are four ways you can use the Neural Network Toolbox software.

- The first way is through its tools. You can open any of these tools from a master tool started by the command `nnstart`. These tools provide a convenient way to access the capabilities of the toolbox for the following tasks:
 - Function fitting ([nftool](#))
 - Pattern recognition ([nprtool](#))

- o Data clustering ([*nctool*](#))
- o Time-series analysis ([*ntstool*](#))

The second way to use the toolbox is through basic command-line operations. The command-line operations offer more flexibility than the tools, but with some added complexity. If this is your first experience with the toolbox, the tools provide the best introduction. In addition, the tools can generate scripts of documented MATLAB code to provide you with templates for creating your own customized command-line functions. The process of using the tools first, and then

generating and modifying MATLAB scripts, is an excellent way to learn about the functionality of the toolbox.

- The third way to use the toolbox is through customization. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox. You can create networks with arbitrary connections, and you still be able to train them using existing toolbox training functions (as long as the network components are differentiable).

- The fourth way to use the toolbox is through the ability to modify any of

the functions contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

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

2.3 AUTOMATIC SCRIPT GENERATION

The tools themselves form an important part of the learning process for the Neural Network Toolbox software. They guide you through the process of designing neural networks to solve problems in four important application areas, without requiring any background in neural networks or sophistication in using MATLAB. In addition, the tools can automatically generate both simple and advanced MATLAB scripts that can reproduce the steps performed by the tool, but with the option to override default settings. These scripts can provide you

with templates for creating customized code, and they can aid you in becoming familiar with the command-line functionality of the toolbox. It is highly recommended that you use the automatic script generation facility of these tools.

2.4 NEURAL NETWORK TOOLBOX APPLICATIONS

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

Industry	Business Applications

Aerospace	High-performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, and aircraft component fault detection
Automotive	Automobile automatic guidance system, and warranty activity analysis
Banking	Check and other document reading and credit application evaluation
Defense	Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, and signal/image identification
Electronics	Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, and nonlinear modeling
Entertainment	Animation, special effects, and market forecasting
Financial	Real estate appraisal, loan advising, mortgage screening, corporate bond rating, credit-line use analysis, credit card activity tracking, portfolio trading program, corporate financial analysis, and currency price prediction
Industrial	Prediction of industrial processes, such as the output gases of furnaces, replacing complex and costly equipment used for this purpose in the past

Insurance	Policy application evaluation and product optimization
Manufacturing	Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, and dynamic modeling of chemical process system
Medical	Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, and emergency-room test advisement
Oil and gas	Exploration
Robotics	Trajectory control, forklift robot, manipulator controllers, and vision systems
Securities	Market analysis, automatic bond rating, and stock trading advisory systems
Speech	Speech recognition, speech compression, vowel classification, and text-to-speech synthesis
Telecommunications	Image and data compression, automated information services, real-time translation of spoken language, and customer payment

processing systems

Transportation Truck brake diagnosis systems, vehicle scheduling, and routing systems

2.5 NEURAL NETWORK DESIGN STEPS

The standard steps for designing neural networks to solve problems are the following:

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

There are four typical neural networks application areas: function fitting, pattern recognition, clustering, and time-series analysis.

Chapter 3

WORK FLOW FOR NEURAL NETWORK DESIGN

3.1 INTRODUCTION

The work flow for the neural network design process has seven primary steps. Referenced topics discuss the basic ideas behind steps 2, 3, and 5.

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

Data collection in step 1 generally

occurs outside the framework of Neural Network Toolbox software. Details of the other steps and discussions of steps 4, 6, and 7, are discussed in topics specific to the type of network.

The Neural Network Toolbox software uses the network object to store all of the information that defines a neural network. This topic describes the basic components of a neural network and shows how they are created and stored in the network object.

After a neural network has been created, it needs to be configured and then trained. Configuration involves arranging

the network so that it is compatible with the problem you want to solve, as defined by sample data. After the network has been configured, the adjustable network parameters (called weights and biases) need to be tuned, so that the network performance is optimized. This tuning process is referred to as training the network. Configuration and training require that the network be provided with example data. This topic shows how to format the data for presentation to the network. It also explains network configuration and the two forms of network training: incremental training and batch training.

3.2 FOUR LEVELS OF NEURAL NETWORK DESIGN

There are four different levels at which the Neural Network Toolbox software can be used. The first level is represented by the GUIs. These provide a quick way to access the power of the toolbox for many problems of function fitting, pattern recognition, clustering and time series analysis.

The second level of toolbox use is through basic command-line operations. The command-line functions use simple argument lists with intelligent default

settings for function parameters. (You can override all of the default settings, for increased functionality.) This topic, and the ones that follow, concentrate on command-line operations.

The GUIs described in Getting Started can automatically generate MATLAB code files with the command-line implementation of the GUI operations. This provides a nice introduction to the use of the command-line functionality.

A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create

your own custom neural networks, while still having access to the full functionality of the toolbox.

The fourth level of toolbox usage is the ability to modify any of the code files contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

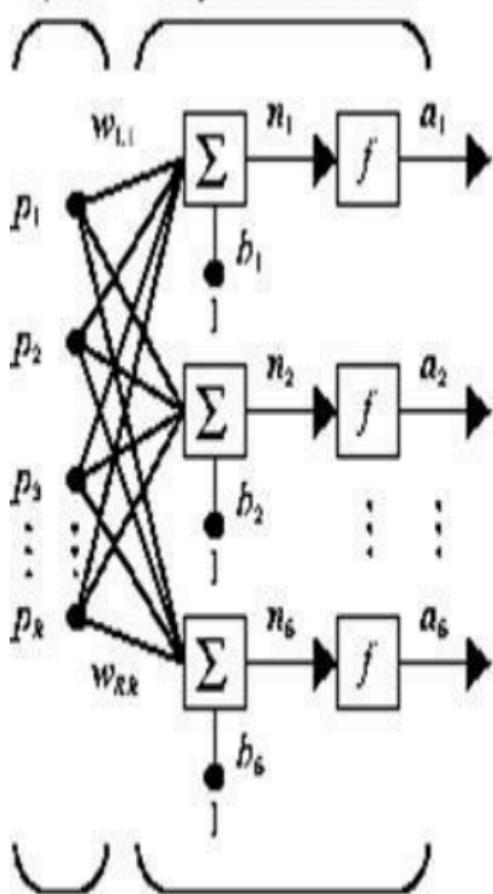
3.3 NEURAL NETWORK ARCHITECTURES

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

3.3.1 One Layer of Neurons

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

Inputs Layer of Neurons



Where

R = number of elements in input vector

S = number of neurons in layer

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

In this network, each element of the input vector \mathbf{p} is connected to each neuron

input through the weight matrix \mathbf{W} . The i th neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$. The various $n(i)$ taken together form an S -element net input vector \mathbf{n} . Finally, the neuron layer outputs form a column vector \mathbf{a} . The expression for \mathbf{a} is shown at the bottom of the figure.

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

You can create a single (composite) layer of neurons having different transfer

functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix \mathbf{W} .

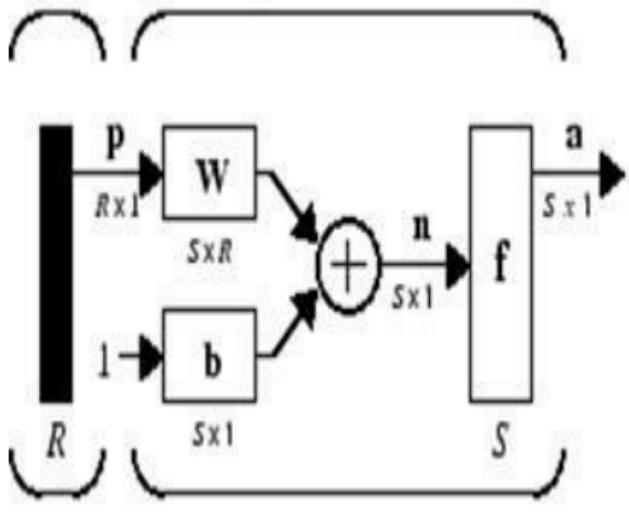
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix \mathbf{W} indicate the destination neuron of the weight, and the column indices indicate which source is

the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

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

Input Layer of Neurons



Where...

R = number of elements in input vector

S = number of neurons in layer 1

$$a = f(Wp + b)$$

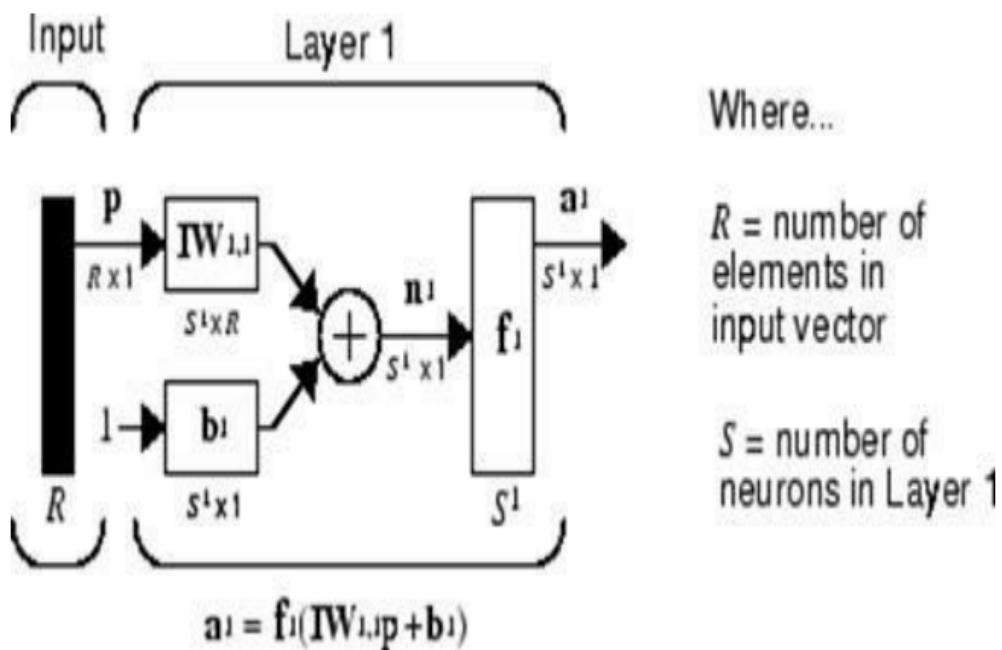
Here p is an R -length input vector, W is an $S \times R$ matrix, a and b are S -length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector b , the summer, and the transfer function blocks.

Inputs and Layers

To describe networks having multiple layers, the notation must be extended. Specifically, it needs to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. It also needs to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs *input weights*; we will call weight matrices connected to layer outputs *layer weights*. Further, superscripts are used to identify the

source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, the one-layer multiple input network shown earlier is redrawn in abbreviated form here.



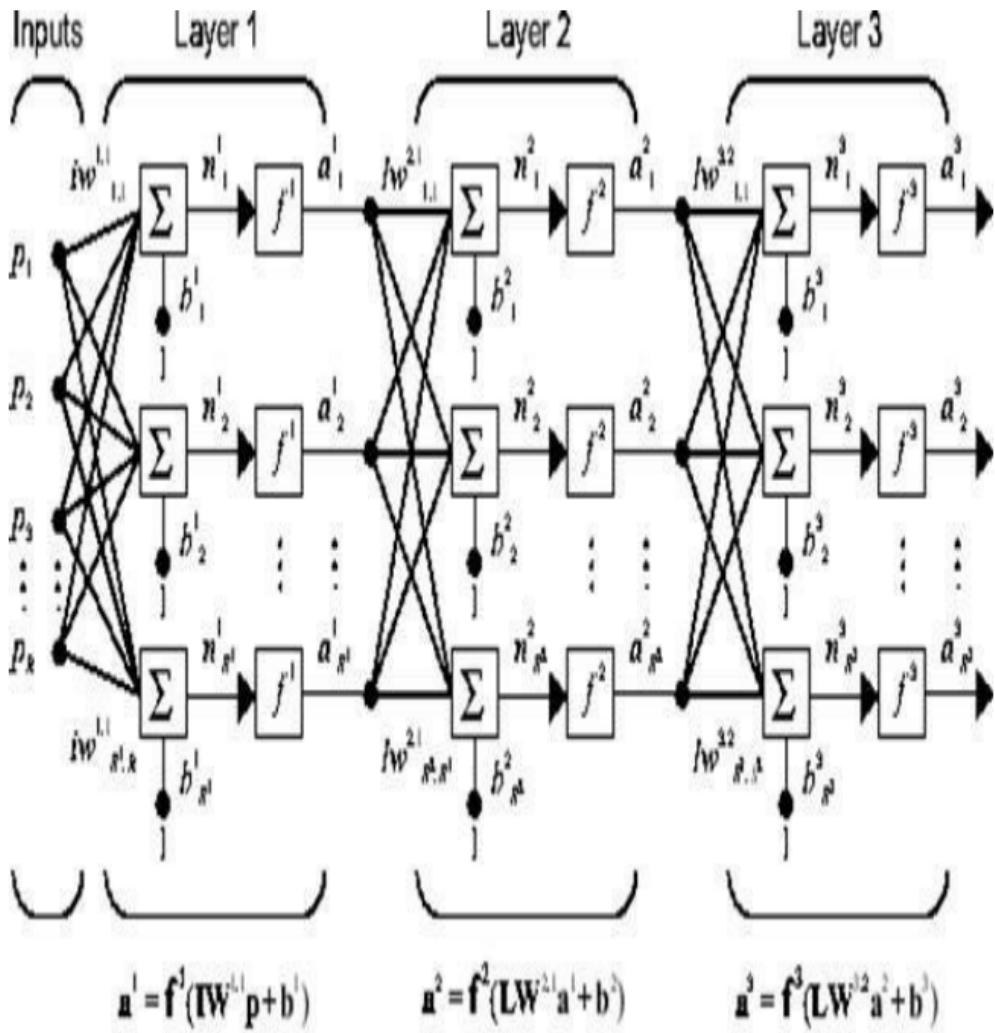
As you can see, the weight matrix

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

Multiple Layers of Neurons uses layer weight (\mathbf{LW}) matrices as well as input weight (\mathbf{IW}) matrices.

3.3.2 Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix \mathbf{W} , a bias vector \mathbf{b} , and an output vector \mathbf{a} . To distinguish between the weight matrices, output vectors, etc., for each of these layers in the figures, the number of the layer is appended as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown next, and in the equations at the bottom of the figure.



The network shown above

has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an $S^2 \times S^1$ weight matrix \mathbf{W}^2 . The input to layer 2 is \mathbf{a}^1 ; the output is \mathbf{a}^2 . Now that all the vectors and matrices of layer 2 have been identified, it can be treated as a single-layer network on its own. This approach can be taken with any

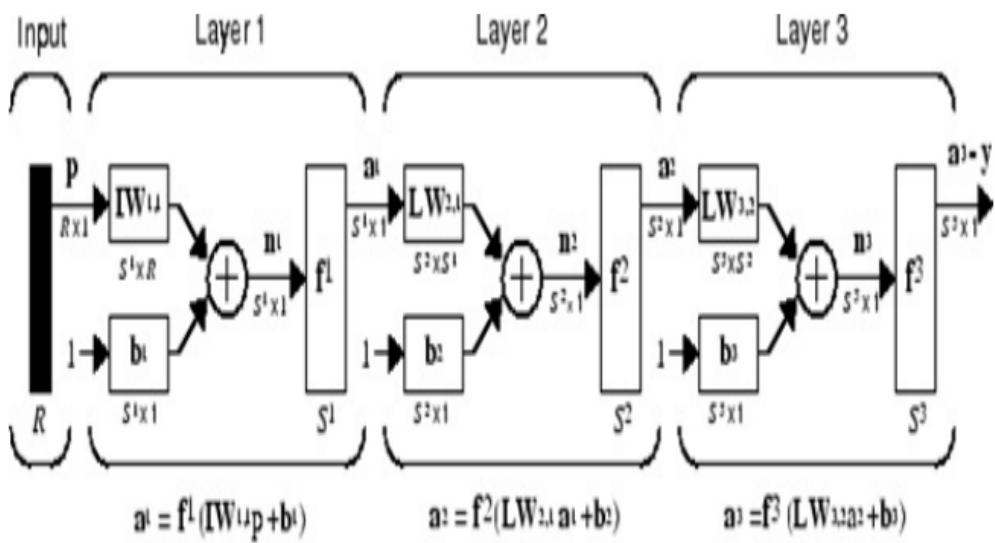
layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. This toolbox does not use that designation.

The architecture of a multilayer network with a single input vector can be specified with the notation $R - S^1 - S^2 - \dots - S^M$, where the number of elements of the input vector

and the number of neurons in each layer are specified.

The same three-layer network can also be drawn using abbreviated notation.



Multiple-layer networks are quite

powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in Multilayer Neural Networks and Backpropagation Training.

Here it is assumed that the output of the third layer, \mathbf{a}^3 , is the network output of interest, and this output is labeled as y . This notation is used to specify the output of multilayer networks.

3.3.3 Input and Output Processing Functions

Network inputs might have associated processing functions. Processing functions transform user input data to a form that is easier or more efficient for a network.

For instance, `mapminmax` transforms input data so that all values fall into the interval $[-1, 1]$. This can speed up learning for many networks. `removeconstantrows` removes the rows of the input vector that correspond to input elements that always have the same value, because these input

elements are not providing any useful information to the network. The third common processing function is `fixunknowns`, which recodes unknown data (represented in the user's data with `NAN` values) into a numerical form for the network. `fixunknowns` preserves information about which values are known and which are unknown.

Similarly, network outputs can also have associated processing functions. Output processing functions are used to transform user-provided target vectors for network use. Then, network outputs are reverse-processed using the same functions to produce output data with the

same characteristics as the original user-provided targets.

Both `mapminmax` and `removeconstantcols` are often associated with network outputs. However, `fixunknowns` is not. Unknown values in targets (represented by `NaN` values) do not need to be altered for network use.

3.4 MULTILAYER NEURAL NETWORKS AND BACKPROPAGATION TRAINING

The multilayer feedforward neural network is the workhorse of the Neural Network Toolbox™ software. It can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network.

Note: The training functions described in this topic are not limited to multilayer networks. They can be used to train arbitrary architectures (even custom networks), as long as their components are differentiable.

The work flow for the general neural network design process has seven primary steps:

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

6. Validate the network (post-training analysis)
7. Use the network

Step 1 might happen outside the framework of Neural Network Toolbox software, but this step is critical to the success of the design process.

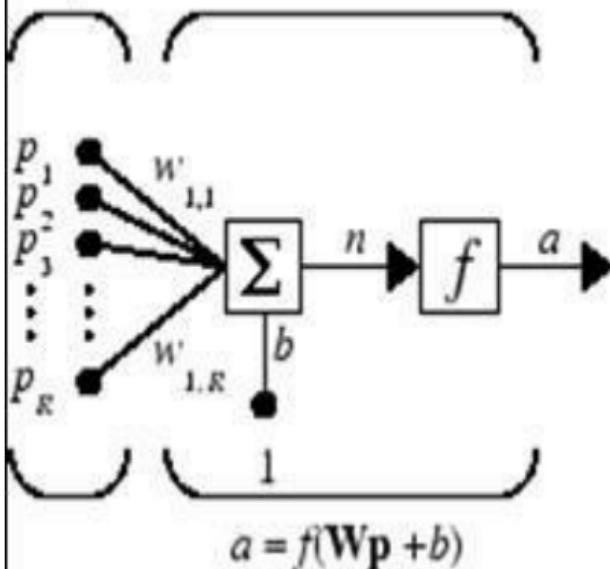
3.5 MULTILAYER NEURAL NETWORK ARCHITECTURE

This topic presents part of a typical multilayer network workflow.

3.5.1 Neuron Model (logsig, tansig, purelin)

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

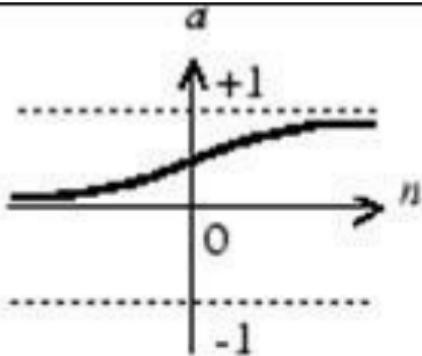
Input General Neuron



Where

R = number of elements in input vector

Multilayer networks often use the log-sigmoid transfer function [logsig](#).

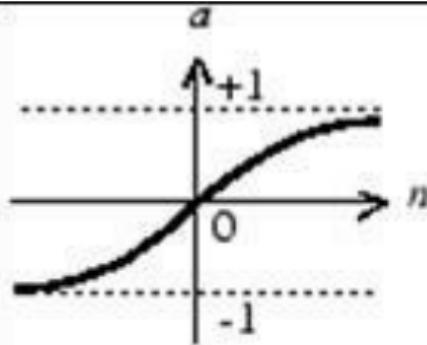


$$a = \text{logsig}(n)$$

Log-Sigmoid Transfer Function

The function [logsig](#) generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

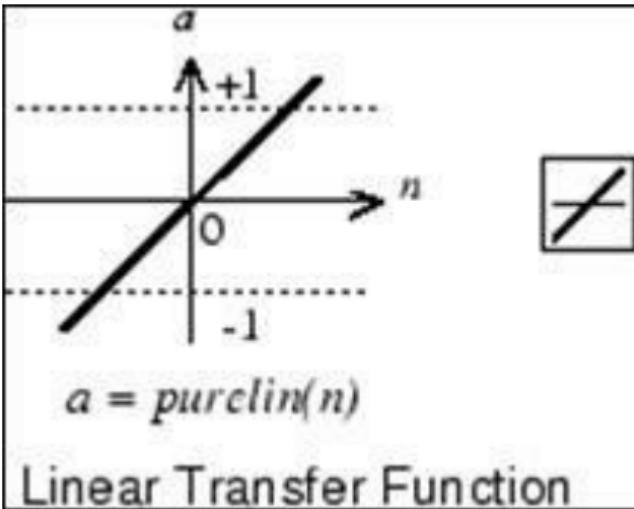
Alternatively, multilayer networks can use the tan-sigmoid transfer function [tansig](#).



$$a = \text{tansig}(n)$$

Tan-Sigmoid Transfer Function

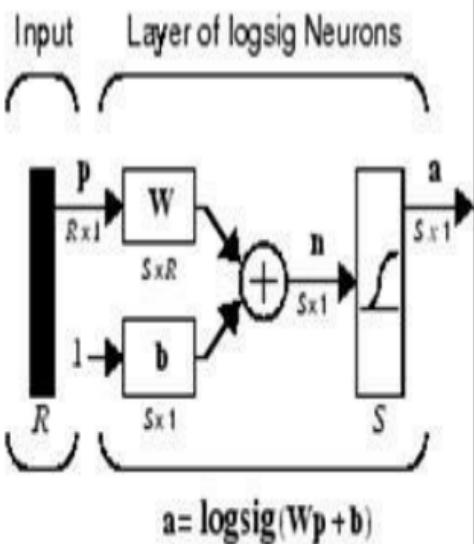
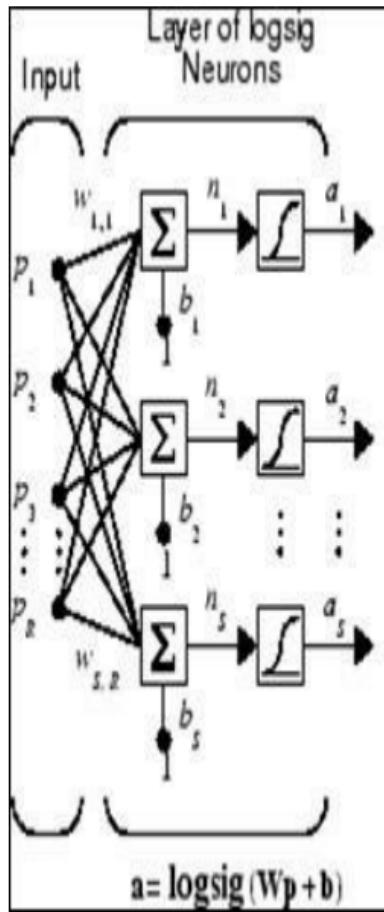
Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function [purelin](#) is shown below.



The three transfer functions described here are the most commonly used transfer functions for multilayer networks, but other differentiable transfer functions can be created and used if desired.

3.5.2 Feedforward Neural Network

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



Where...

R = number of elements in input vector

S = number of neurons in layer

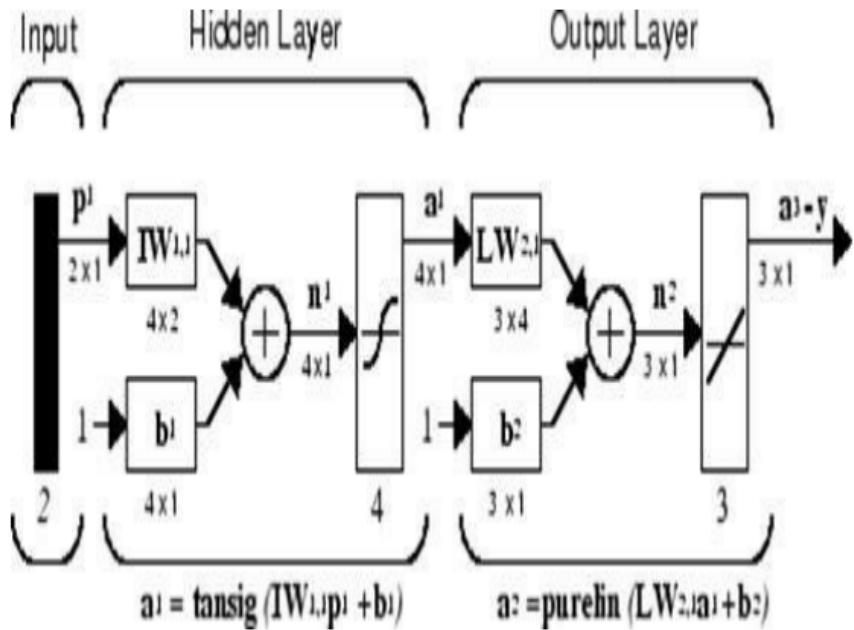
Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of

neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as [logsig](#)). This is the case when the network is used for pattern recognition problems (in which a decision is being made by the network).

For multiple-layer networks the

layer number determines the superscript on the weight matrix. The appropriate notation is used in the two-layer **tansig/purelin** network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite

number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer.

Now that the architecture of the multilayer network has been defined, the design process is described in the following sections.

3.6 UNDERSTANDING NEURAL NETWORK TOOLBOX DATA STRUCTURES

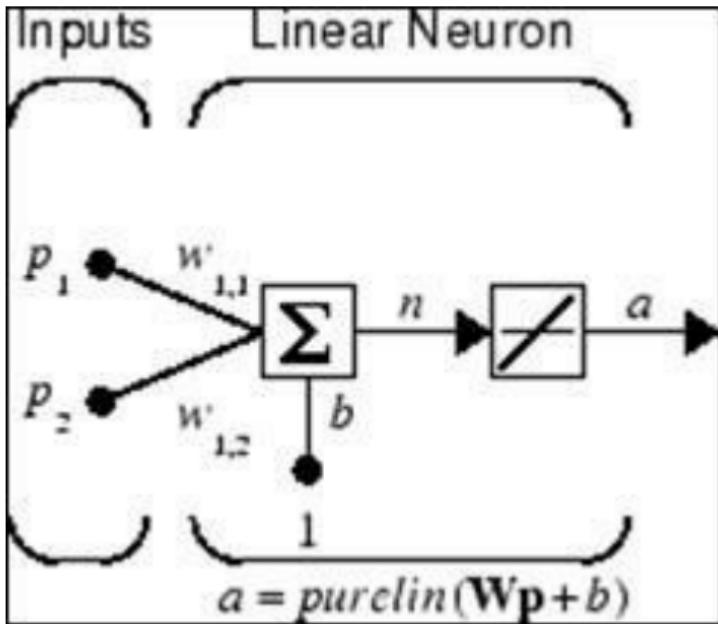
This topic discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks. The following section describes how the format of the data structures affects network training.

There are two basic types of input vectors: those that occur concurrently (at the same time, or in no particular time

sequence), and those that occur sequentially in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, you could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

3.6.1 Simulation with Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, you need not be concerned about whether or not the input vectors occur in a particular time sequence, so you can treat the inputs as concurrent. In addition, the problem is made even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this linear feedforward network, use the following commands:

```
net = linearlayer;
```

```
net.inputs{1}.size = 2;
```

```
net.layers{1}.dimensions = 1;
```

For simplicity, assign the weight matrix and bias to be $W = [1 \ 2]$ and $b = [0]$.

The commands for these assignments are

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

```
net.b{1} = 0;
```

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

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

Concurrent vectors are presented to

the network as a single matrix:

$$P = [1 \ 2 \ 2 \ 3; 2 \ 1 \ 3 \ 1];$$

You can now simulate the network:

$$A = \text{net}(P)$$

$$A =$$

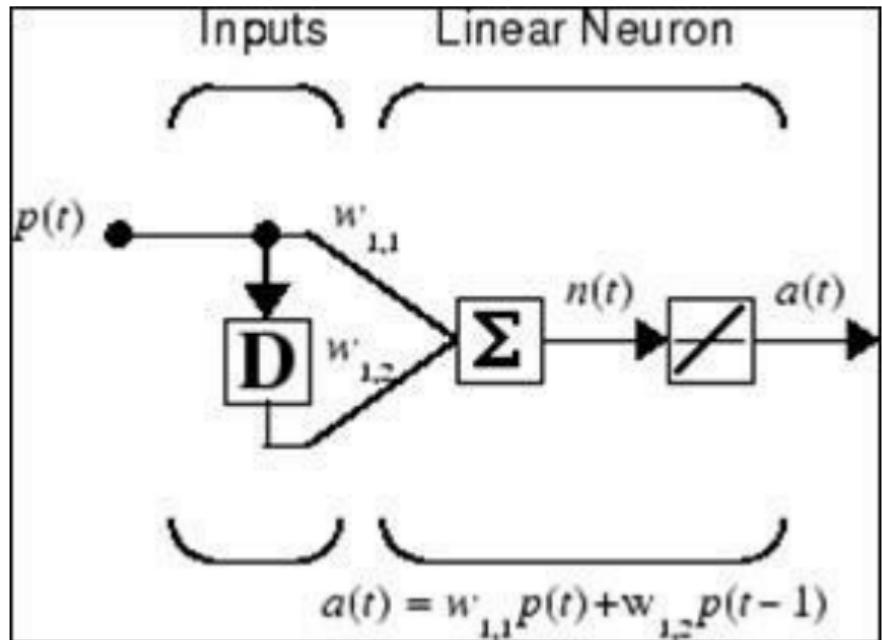
$$\begin{matrix} 5 & 4 & 8 & 5 \end{matrix}$$

A single matrix of concurrent vectors is presented to the network, and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The

ordering of the input vectors is not important, because they do not interact with each other.

3.6.2 Simulation with Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, the next figure shows a simple network that contains one delay.



The following commands create this network:

```
net = linearlayer([0 1]);
```

```
net.inputs{1}.size = 1;
```

```
net.layers{1}.dimensions = 1;
```

```
net.biasConnect = 0;
```

Assign the weight matrix to be $W = [1 \ 2]$.

The command is:

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

Suppose that the input sequence is:

$p_1 = [1], p_2 = [2], p_3 = [3], p_4 = [4]$

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

`P = {1 2 3 4};`

You can now simulate the network:

`A = net(P)`

A =

[1] [4] [7] [10]

You input a cell array containing a sequence of inputs, and the network produces a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If you were to change the order of the inputs, the numbers obtained in the output would change.

3.6.3 Simulation with Concurrent Inputs in a Dynamic Network

If you were to apply the same inputs as a set of concurrent inputs instead of a sequence of inputs, you would obtain a completely different response. (However, it is not clear why you would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

which can be created with the following code:

$$P = [1 \ 2 \ 3 \ 4];$$

When you simulate with concurrent inputs, you obtain

$$A = \text{net}(P)$$

$$A =$$

$$1 \quad 2 \quad 3 \quad 4$$

The result is the same as if you had concurrently applied each one of the inputs to a separate network and computed one output. Note that because you did not assign any initial conditions to the network delays, they were assumed to be 0. For this case the output is simply 1

times the input, because the weight that multiplies the current input is 1.

In certain special cases, you might want to simulate the network response to several different sequences at the same time. In this case, you would want to present the network with a concurrent set of sequences. For example, suppose you wanted to present the following two sequences to the network:

$$p_1(1) = [1], p_1(2) = [2], p_1(3) = [3], p_1(4) = [4]$$
$$p_2(1) = [4], p_2(2) = [3], p_2(3) = [2], p_2(4) = [1]$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur

at the same time:

$$P = \{[1\ 4]\ [2\ 3]\ [3\ 2]\ [4\ 1]\};$$

You can now simulate the network:

$$A = \text{net}(P);$$

The resulting network output would be

$$A = \{[1\ 4]\ [4\ 11]\ [7\ 8]\ [10\ 5]\}$$

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were

each applied to separate networks running in parallel.

The following diagram shows the general format for the network input P when there are Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.

*Q*th Sequence



$\{[\mathbf{p}_1(1), \mathbf{p}_2(1), \dots, \mathbf{p}_Q(1)], [\mathbf{p}_1(2), \mathbf{p}_2(2), \dots, \mathbf{p}_Q(2)], \dots, [\mathbf{p}_1(TS), \mathbf{p}_2(TS), \dots, \mathbf{p}_Q(TS)]\}$

First Sequence

Chapter 4

ADAPTIVE NEURAL NETWORK FILTERS

4.1 INTRODUCTION

The ADALINE (adaptive linear neuron) networks discussed in this topic are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can solve only linearly separable problems. However, here the LMS (least mean squares) learning rule, which is much more powerful than the perceptron learning rule, is used. The LMS, or Widrow-Hoff, learning rule minimizes the mean square error and

thus moves the decision boundaries as far as it can from the training patterns.

In this section, you design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancellation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this

subject is Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985.

The adaptive training of self-organizing and competitive networks is also considered in this section.

4.2 ADAPTIVE FUNCTIONS: ADAPT

This section introduces the function adapt, which changes the weights and biases of a network incrementally during training.

Adapt: Adapt neural network to data as it is simulated

Syntax

$$[net, Y, E, Pf, Af, tr] = \text{adapt}(net, P, T, Pi, Ai)$$

Description

This function calculates network outputs and errors after each presentation of an input.

[net,Y,E,Pf,Af,tr] =
adapt(net,P,T,Pi,Ai) takes

net	Network
P	Network inputs
T	Network targets (default = zeros)
Pi	Initial input delay conditions (default = zeros)
Ai	Initial layer delay conditions (default = zeros)

and returns the following after applying the adapt function net.adaptFcn with the adaption parameters net.adaptParam:

net	Updated network
Y	Network outputs
E	Network errors
Pf	Final input delay conditions
Af	Final layer delay conditions
tr	Training record (epoch and perf)

Note that T is optional and is only needed for networks that require targets. Pi and Pf are also optional and only need to be used for networks that have input or layer delays.

adapt's signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented,

P	Ni-by-TS cell array	Each element $P\{i,ts\}$ is an Ri -by- Q
T	Nt-by-TS cell array	Each element $T\{i,ts\}$ is a Vi -by- Q
Pi	Ni-by-ID cell array	Each element $Pi\{i,k\}$ is an Ri -by- Q
Ai	Nl-by-LD cell array	Each element $Ai\{i,k\}$ is an Si -by- Q
Y	No-by-TS cell array	Each element $Y\{i,ts\}$ is a Ui -by- Q

E	No-by-TS cell array	Each element E{i,ts} is a Ui-by-Q
Pf	Ni-by-ID cell array	Each element Pf{i,k} is an Ri-by-Q
Af	Nl-by-LD cell array	Each element Af{i,k} is an Si-by-Q

where

Ni	=	net.numInputs
Nl	=	net.numLayers
No	=	net.numOutputs
ID	=	net.numInputDelays
LD	=	net.numLayerDelays
TS	=	Number of time steps
Q	=	Batch size
Ri	=	net.inputs{i}.size
Si	=	net.layers{i}.size
Ui	=	net.outputs{i}.size

The columns of Pi, Pf, Ai, and Af are ordered from oldest delay condition to most recent:

Pi{i,k}	=	Input i at time ts = k - ID
Pf{i,k}	=	Input i at time ts = TS + k - ID
Ai{i,k}	=	Layer output i at time ts = k - LD

$Af\{i,k\}$	=	Layer output i at time $ts = TS + k - LD$
-------------	---	---

The matrix format can be used if only one time step is to be simulated ($TS = 1$). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

P	(sum of R_i)-by-Q matrix
T	(sum of V_i)-by-Q matrix
P_i	(sum of R_i)-by-($ID * Q$) matrix
A_i	(sum of S_i)-by-($LD * Q$) matrix
Y	(sum of U_i)-by-Q matrix
E	(sum of U_i)-by-Q matrix
P_f	(sum of R_i)-by-($ID * Q$) matrix
A_f	(sum of S_i)-by-($LD * Q$) matrix

Examples

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.

$$p1 = \{-1 \ 0 \ 1 \ 0 \ 1 \ 1 \ -1 \ 0 \ -1 \ 1 \ 0 \ 1\};$$

$$t1 = \{-1 \ -1 \ 1 \ 1 \ 1 \ 2 \ 0 \ -1 \ -1 \ 0 \ 1 \ 1\};$$

Here linearlayer is used to create a layer with an input range of [-1 1], one neuron, input delays of 0 and 1, and a learning rate of 0.1. The linear layer is then simulated.

```
net = linearlayer([0 1],0.1);
```

Here the network adapts for one pass through the sequence.

The network's mean squared error is displayed. (Because this is the first call to adapt, the default Pi is used.)

```
[net,y,e,pf] = adapt(net,p1,t1);
```

```
mse(e)
```

Note that the errors are quite large. Here the network adapts to another 12 time steps (using the previous Pf as the new initial delay conditions).

```
p2 = {1 -1 -1 1 1 -1 0 0 0 1 -1 -1};
```

```
t2 = {2 0 -2 0 2 0 -1 0 0 1 0 -1};
```

```
[net,y,e,pe] = adapt(net,p2,t2,pe);  
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```
p3 = [p1 p2];
```

```
t3 = [t1 t2];
```

```
for i = 1:100
```

```
[net,y,e] = adapt(net,p3,t3);
```

```
end
```

```
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

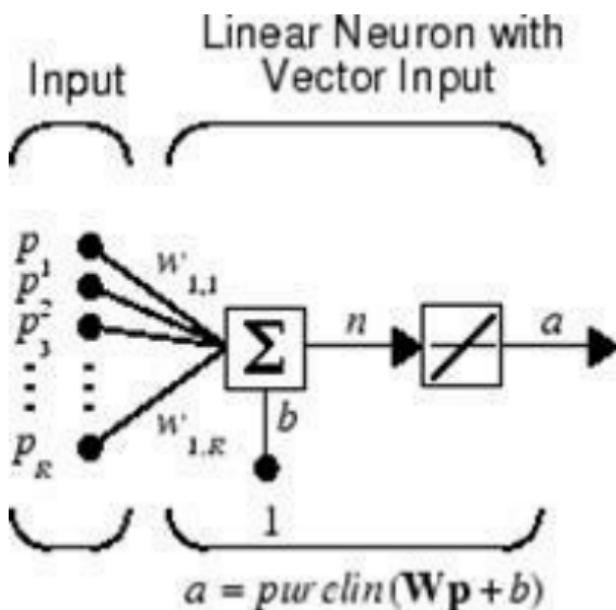
Algorithms

adapt calls the function indicated by net.adaptFcn, using the adaption parameter values indicated by net.adaptParam.

Given an input sequence with TS steps, the network is updated as follows: Each step in the sequence of inputs is presented to the network one at a time. The network's weight and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated TS times.

4.3 LINEAR NEURON MODEL

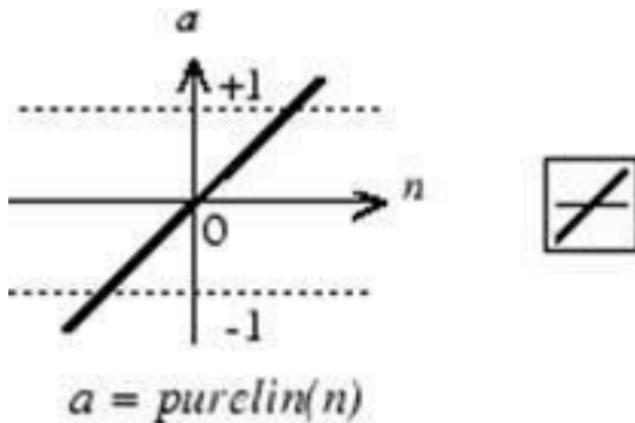
A linear neuron with R inputs is shown below.



Where...

R = number of elements in input vector

This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, named purelin.



Linear Transfer Function

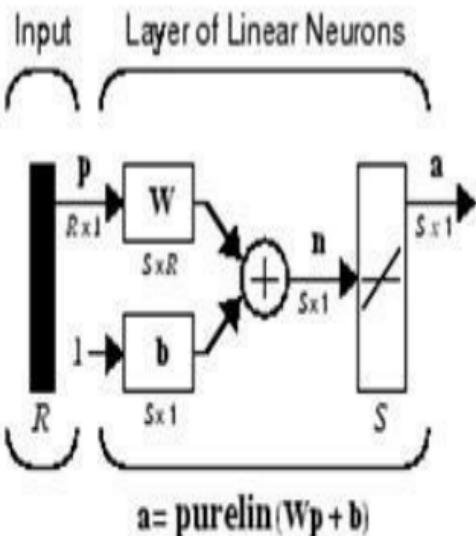
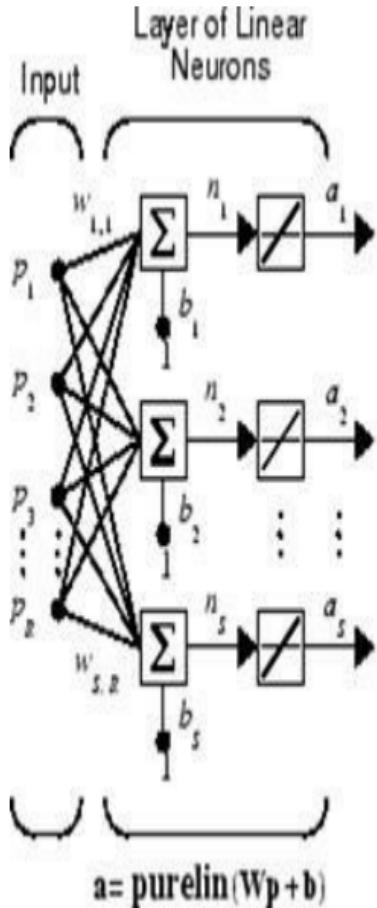
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\begin{aligned} a &= \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) \\ &= \mathbf{W}\mathbf{p} + b \end{aligned}$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

4.4 ADAPTIVE LINEAR NETWORK ARCHITECTURE

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights \mathbf{W} .



Where...

R = number of elements in input vector

S = number of neurons in layer

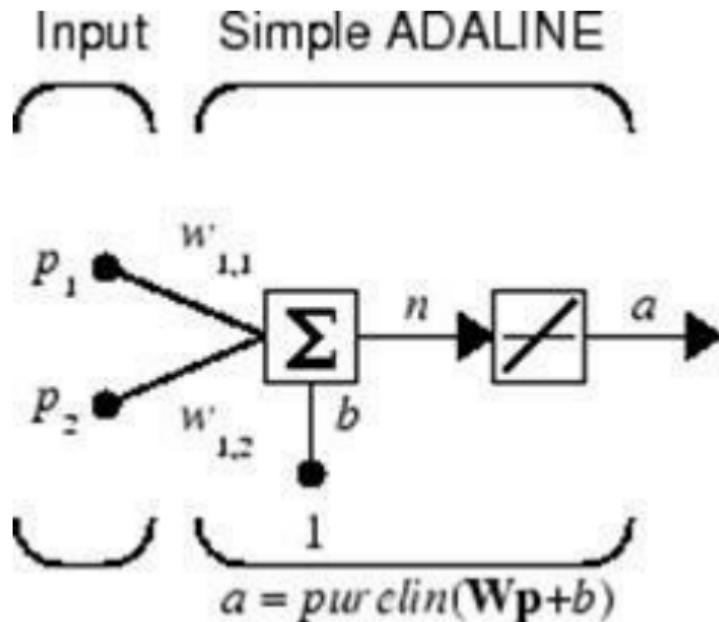
This network is sometimes called a MADALINE for Many ADALINES. Note that the figure on the right defines

an S -length output vector \mathbf{a} .

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

4.4.1 Single ADALINE (linearlayer)

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



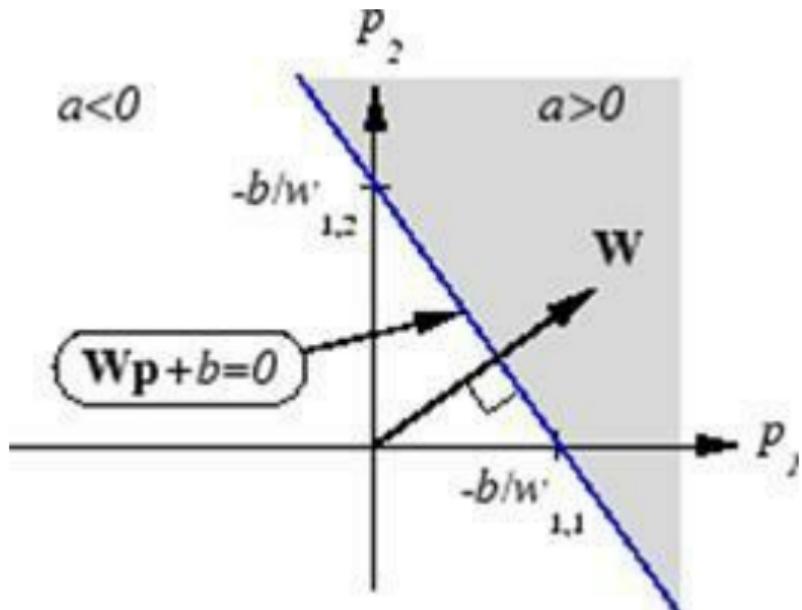
The weight matrix \mathbf{W} in this case has only one row. The network output is

$$\begin{aligned}\alpha &= \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) \\ &= \mathbf{W}\mathbf{p} + b\end{aligned}$$

or

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

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{W}\mathbf{p} + b = 0$ specifies such a decision boundary, as shown below).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

However, ADALINE can classify

objects in this way only when the objects are linearly separable. Thus, ADALINE has the same limitation as the perceptron.

You can create a network similar to the one shown using this command:

```
net = linearlayer;
```

```
net = configure(net,[0;0],[0]);
```

The sizes of the two arguments to `configure` indicate that the layer is to have two inputs and one output. Normally [train](#) does this configuration for you, but this allows us to inspect the weights before training.

The network weights and biases are set

to zero, by default. You can see the current values using the commands:

$W = \text{net.IW}\{1,1\}$

$W =$

0 0

and

$b = \text{net.b}\{1\}$

$b =$

0

You can also assign arbitrary values to the weights and bias, such as 2 and 3 for the weights and -4 for the bias:

$\text{net.IW}\{1,1\} = [2 3];$

```
net.b{1} = -4;
```

You can simulate the ADALINE for a particular input vector.

```
p = [5; 6];
```

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

```
a =
```

24

To summarize, you can create an ADALINE network with linearlayer, adjust its elements as you want, and simulate it with sim.

4.4.2 linearlayer

Linear layer

Syntax

```
linearlayer(inputDelays,widrowHoff
```

Description

Linear layers are single layers of linear neurons. They may be static, with input delays of 0, or dynamic, with input delays greater than 0. They can be trained on simple linear time series problems, but often are used adaptively

to continue learning while deployed so they can adjust to changes in the relationship between inputs and outputs while being used.

If a network is needed to solve a nonlinear time series relationship, then better networks to try include timedelaynet, narxnet, and narnet.

linearlayer(inputDelays,widrowHoffLR)
these arguments,

inputDelays	Row vector of increasing 0 or positive delays (0)
widrowHoffLR	Widrow-Hoff learning rate (default = 0.01)

and returns a linear layer.

If the learning rate is too small, learning

will happen very slowly. However, a greater danger is that it may be too large and learning will become unstable resulting in large changes to weight vectors and errors increasing instead of decreasing. If a data set is available which characterizes the relationship the layer is to learn, the maximum stable learning rate can be calculated with [maxlinlr](#).

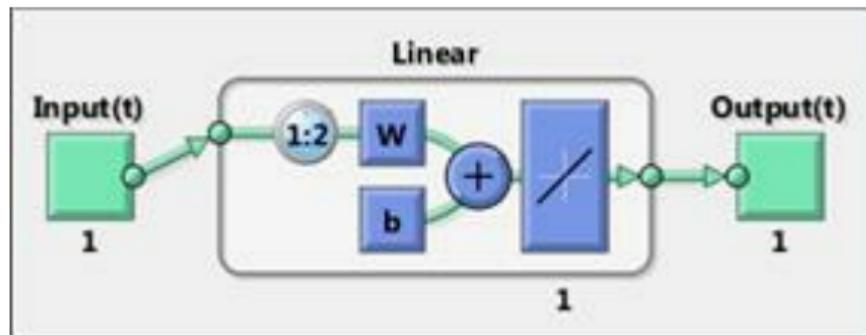
Examples: Create and Train a Linear Layer

Here a linear layer is trained on a simple time series problem.

$$x = \{0 -1 1 1 0 -1 1 0 0 1\};$$

```
t = {0 -1 0 2 1 -1 0 1 0 1};  
net = linearlayer(1:2,0.01);  
[Xs,Xi,Ai,Ts] = preparets(net,x,t);  
net = train(net,Xs,Ts,Xi,Ai);  
view(net)  
Y = net(Xs,Xi);  
perf = perform(net,Ts,Y)  
perf =
```

0.2396



4.5 LEAST MEAN SQUARE ERROR

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

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

Here \mathbf{p}_q is an input to the network, and \mathbf{t}_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and

the network output. The goal is to minimize the average of the sum of these errors.

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

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

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the

input vectors determine whether or not a unique solution exists.

4.6 LMS ALGORITHM (LEARNWH)

Adaptive networks will use the LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown here, is discussed in detail in Linear Neural Networks.

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

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

The LMS algorithm, or Widrow-Hoff

learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

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

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

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

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

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

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

or

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

Here $p_i(k)$ is the i th element of the input vector at the k th iteration.

This can be simplified to

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

and

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

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

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

and

$$2\alpha e(k)$$

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

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

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

Here the error \mathbf{e} and the bias \mathbf{b} are vectors, and α is a *learning rate*. If α is

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

Fortunately, there is a toolbox function, `learnwh`, that does all the calculation for you. It calculates the change in weights as

$$dw = lr * e * p'$$

and the bias change as

$$db = lr * e$$

The constant 2, shown a few lines

above, has been absorbed into the code learning rate lr. The function maxlinlr calculates this maximum stable learning rate lr as $0.999 * P' * P$.

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

4.6.1 learnwh

Widrow-Hoff weight/bias learning function

Syntax

[dW,LS] =

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

Description

learnwh is the Widrow-Hoff weight/bias learning function, and is also known as the delta or least mean squared (LMS) rule.

$[dW, LS] =$
learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
several inputs,

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

and returns

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

Learning occurs according to

the learnwh learning parameter, shown here with its default value.

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

info = learnwh('code') returns useful information for each *code* 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 and error E for a layer with a two-element input and three neurons. You also define the learning rate LR learning parameter.

```
p = rand(2,1);
```

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

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

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

```
dW = learnwh([],p,[],[],[],[],e,[],[],[],lp,[])
```

Network Use

You can create a standard network that uses `learnwh` with `linearlayer`.

To prepare the weights and the bias of layer i of a custom network to learn with `learnwh`,

- Set `net.trainFcn` to '`trainb`'. `net.trainF` becomes `trainb`'s default parameters.
- Set `net.adaptFcn` to '`trains`'. `net.adap` becomes `trains`'s default parameters.
- Set each `net.inputWeights{i,j}.learnFcn`
- Set each `net.layerWeights{i,j}.learnFcn`
- Set `net.biases{i}.learnFcn` to '`learnw`'
Each weight and bias learning parameter property is automatically set to the `learnwh` default parameters.

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

- Set net.trainParam (or net.adaptPara) properties to desired values.
- Call train (or adapt).

Algorithms

learnwh calculates the weight change dW for a given neuron from the neuron's input P and error E , and the weight (or bias) learning rate LR , according to the Widrow-Hoff learning rule:

$$dw = lr * e * pn'$$

4.6.2 maxlinlr

Maximum learning rate for linear layer

Syntax

$lr = \text{maxlinlr}(P)$

$lr = \text{maxlinlr}(P, 'bias')$

Description

`maxlinlr` is used to calculate learning rates for linearlayer.

$lr = \text{maxlinlr}(P)$ takes one argument,

P	R-by-Q matrix of input vectors
---	--------------------------------

and returns the maximum learning rate for a linear layer without a bias that is to be trained only on the vectors in P.

lr = maxlinlr(P,'bias') returns the maximum learning rate for a linear layer with a bias.

Examples

Here you define a batch of four two-element input vectors and find the maximum learning rate for a linear layer with a bias.

```
P = [1 2 -4 7; 0.1 3 10 6];
```

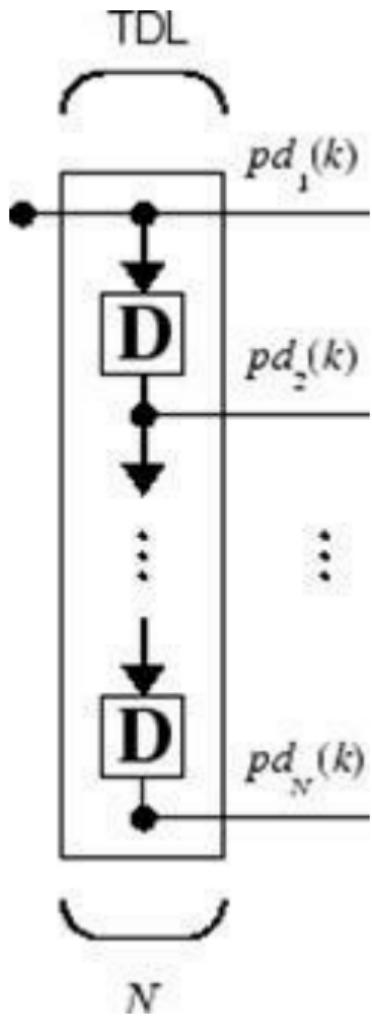
```
lr = maxlinlr(P,'bias')
```

4.7 ADAPTIVE FILTERING (ADAPT)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

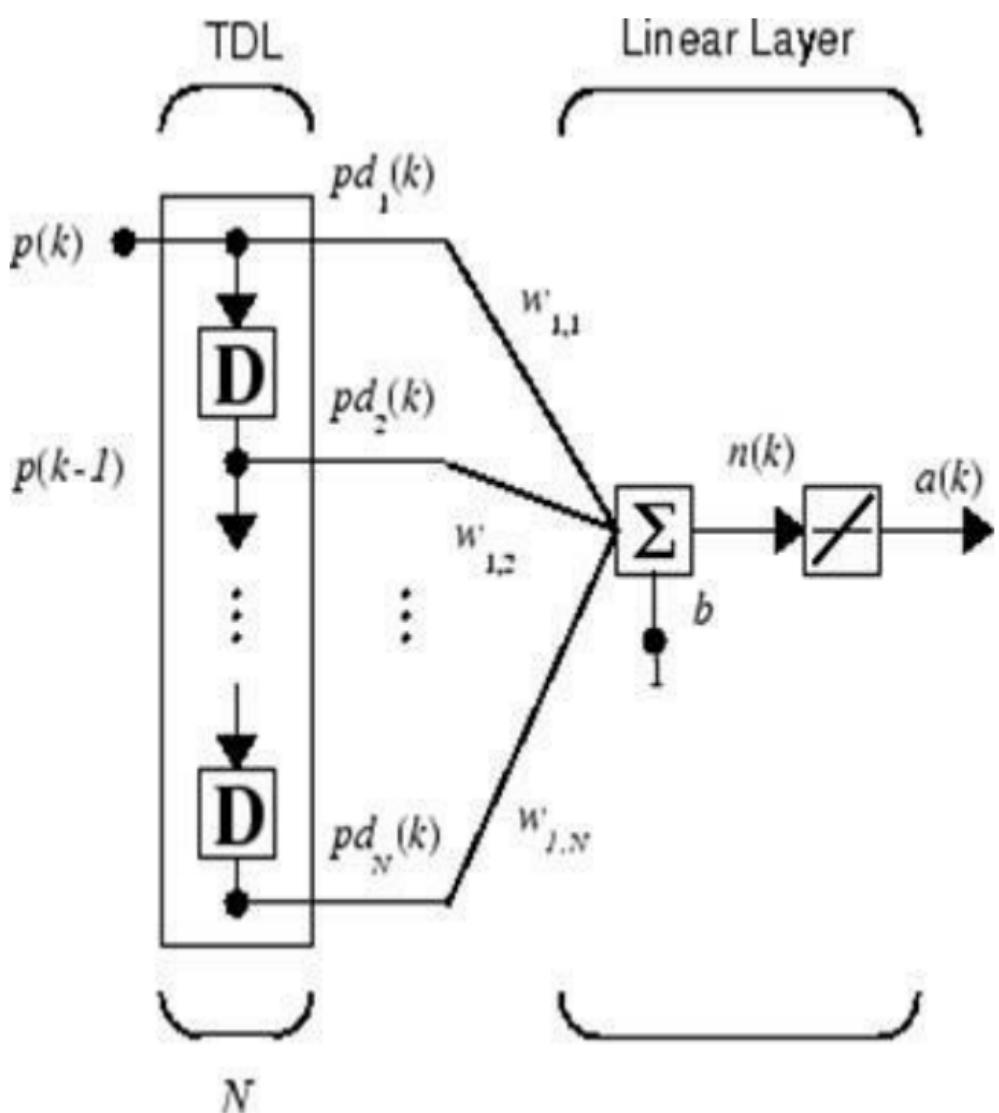
4.7.1 Tapped Delay Line

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



4.7.2 Adaptive Filter

You can combine a tapped delay line with an ADALINE network to create the *adaptive filter* shown in the next figure.



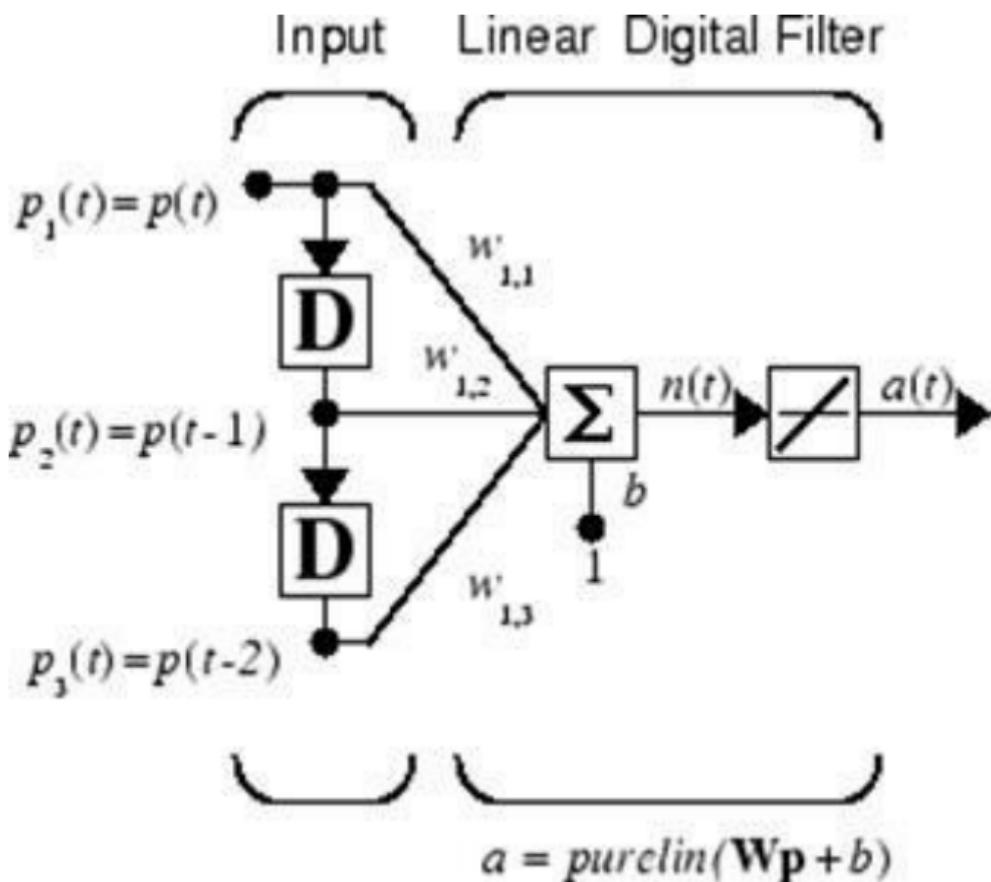
The output of the filter is given by

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

In digital signal processing, this network is referred to as a *finite impulse response (FIR)* filter. Take a look at the code used to generate and simulate such an adaptive network.

4.7.3 Adaptive Filter Example

First, define a new linear network using [linearlayer](#).



Assume that the linear layer has a single neuron with a single input and a tap delay of 0, 1, and 2 delays.

```
net = linearlayer([0 1 2]);
```

```
net = configure(net,0,0);
```

You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.

You can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
```

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

Finally, define the initial values of the

outputs of the delays as

$$\mathbf{p}_i = \{1\ 2\};$$

These are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network.

To set up the input, assume that the input scalars arrive in a sequence: first the value 3, then the value 4, next the value 5, and finally the value 6. You can indicate this sequence by defining the values as elements of a cell array in curly braces.

$$\mathbf{p} = \{3\ 4\ 5\ 6\};$$

Now, you have a network and a

sequence of inputs. Simulate the network to see what its output is as a function of time.

$$[a, pf] = \text{sim}(\text{net}, p, pi)$$

This simulation yields an output sequence

a

[46] [70] [94] [118]

and final values for the delay outputs of

pf

[5] [6]

The example is sufficiently simple that you can check it without a calculator to make sure that you understand the inputs,

initial values of the delays, etc.

The network just defined can be trained with the function adapt to produce a particular output sequence. Suppose, for instance, you want the network to produce the sequence of values 10, 20, 30, 40.

$$t = \{10 \ 20 \ 30 \ 40\};$$

You can train the defined network to do this, starting from the initial delay conditions used above.

Let the network adapt for 10 passes over the data.

for i = 1:10

[net,y,E,rf,af] = adapt(net,p,t,pi);

end

This code returns the final weights, bias, and output sequence shown here.

wts = net.IW{1,1}

wts =

0.5059 3.1053 5.7046

bias = net.b{1}

bias =

-1.5993

y

y =

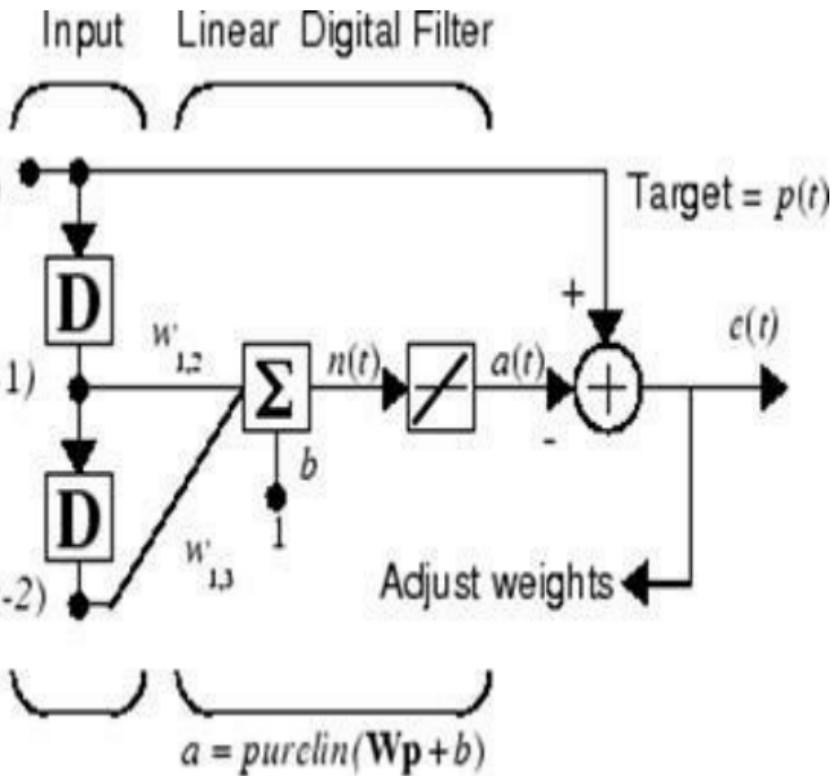
[11.8558] [20.7735]
[29.6679] [39.0036]

Presumably, if you ran additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with adapt. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as prediction or noise cancelation.

4.7.4 Prediction Example

Suppose that you want to use an adaptive filter to predict the next value of a stationary random process, $p(t)$. You can use the network shown in the following figure to do this prediction.



Predictive Filter: $a(t)$ is approximation to $p(t)$

The signal to be predicted, $p(t)$, enters from the left into a tapped delay line. The previous two values of $p(t)$ are available as outputs from the tapped delay line. The network uses adapt to

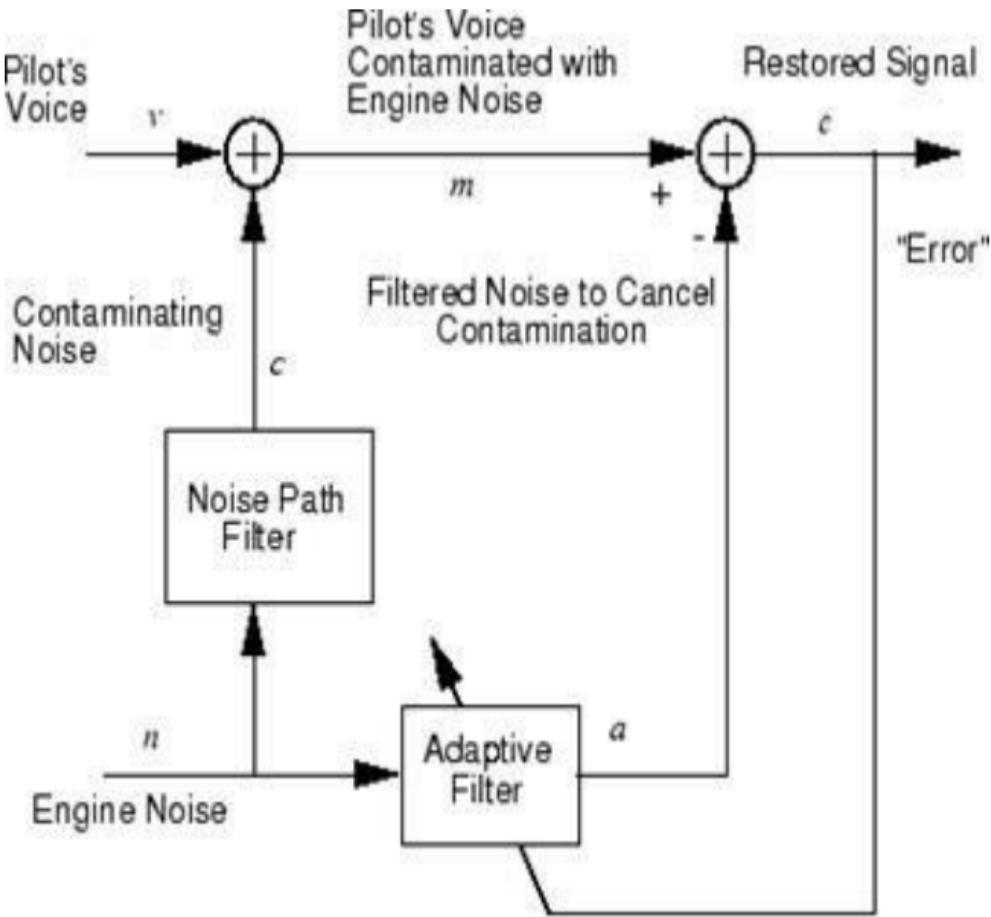
change the weights on each time step so as to minimize the error $e(t)$ on the far right. If this error is 0, the network output $a(t)$ is exactly equal to $p(t)$, and the network has done its prediction properly.

Given the autocorrelation function of the stationary random process $p(t)$, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. This lack does not matter to the network. After it is initialized and operating, the network adapts at each time step to

minimize the error and in a relatively short time is able to predict the input $p(t)$.

4.7.5 Noise Cancelation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit combines with the voice signal. This additional noise makes the resultant signal heard by passengers of low quality. The goal is to obtain a signal that contains the pilot's voice, but not the engine noise. You can cancel the noise with an adaptive filter if you obtain a sample of the engine noise and apply it as the input to the adaptive filter.



Adaptive Filter Adjusts to Minimize Error.
This removes the engine noise from contaminated
signal, leaving the pilot's voice as the "error."

As the preceding figure shows, you

adaptively train the neural linear network to predict the combined pilot/engine signal m from an engine signal n . The engine signal n does not tell the adaptive network anything about the pilot's voice signal contained in m . However, the engine signal n does give the network information it can use to predict the engine's contribution to the pilot/engine signal m .

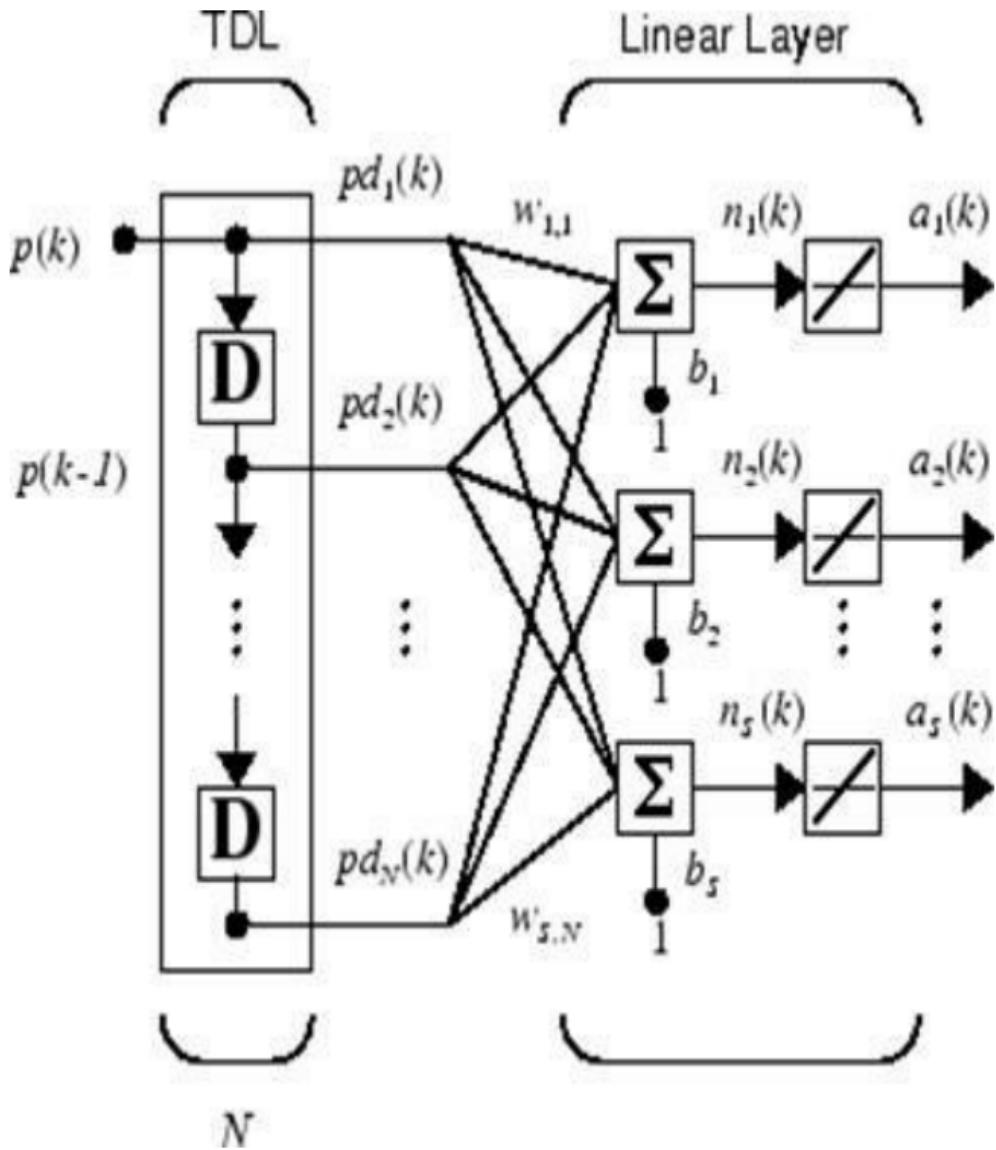
The network does its best to output m adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal m . The network error e is equal to m , the pilot/engine signal, minus the predicted contaminating engine noise

signal. Thus, e contains only the pilot's voice. The linear adaptive network adaptively learns to cancel the engine noise.

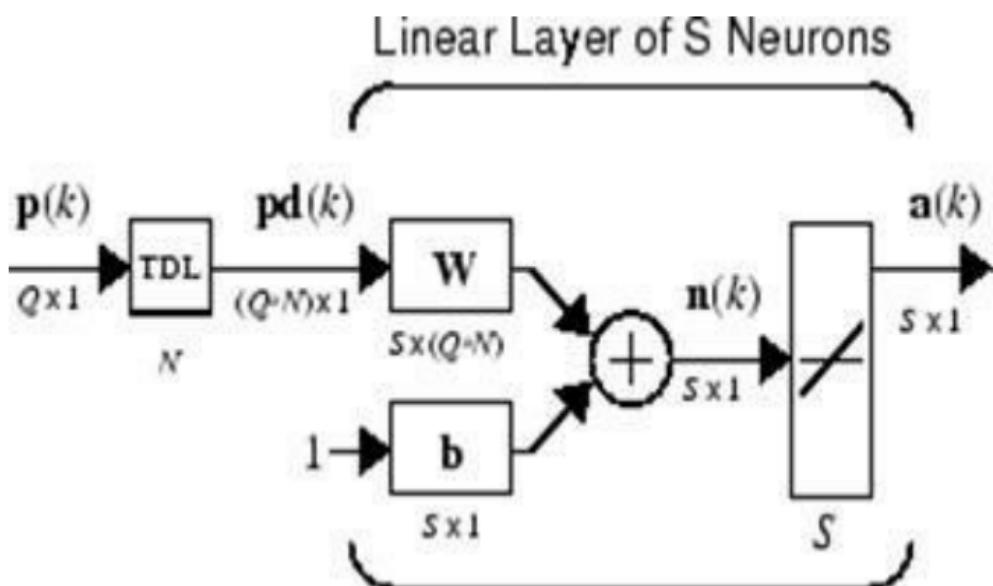
Such adaptive noise canceling generally does a better job than a classical filter, because it subtracts from the signal rather than filtering it out the noise of the signal m .

4.7.6 Multiple Neuron Adaptive Filters

You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with S linear neurons, as shown in the next figure.

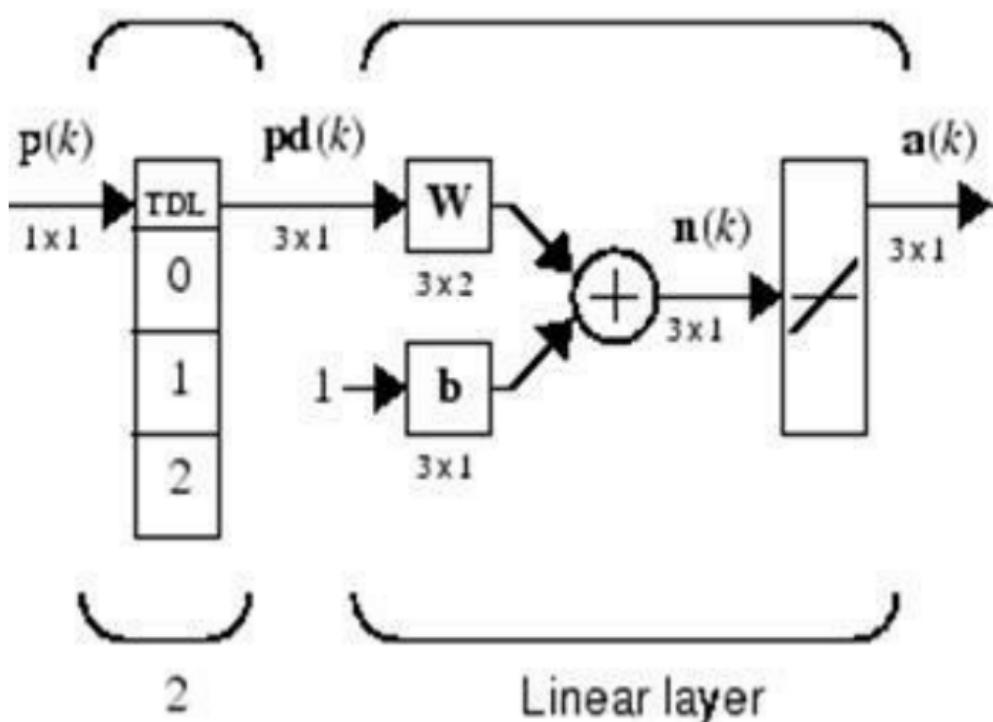


Alternatively, you can represent this same network in abbreviated form.



If you want to show more of the detail of the tapped delay line—and there are not too many delays—you can use the following notation:

Abbreviated Notation



Here, a tapped delay line sends to the weight matrix:

- The current signal
- The previous signal

- The signal delayed before that

You could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays must appear in increasing order as they go from top to bottom.

4.8 ADAPTATIVE FILTER EXAMPLES

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.

4.8.1 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.

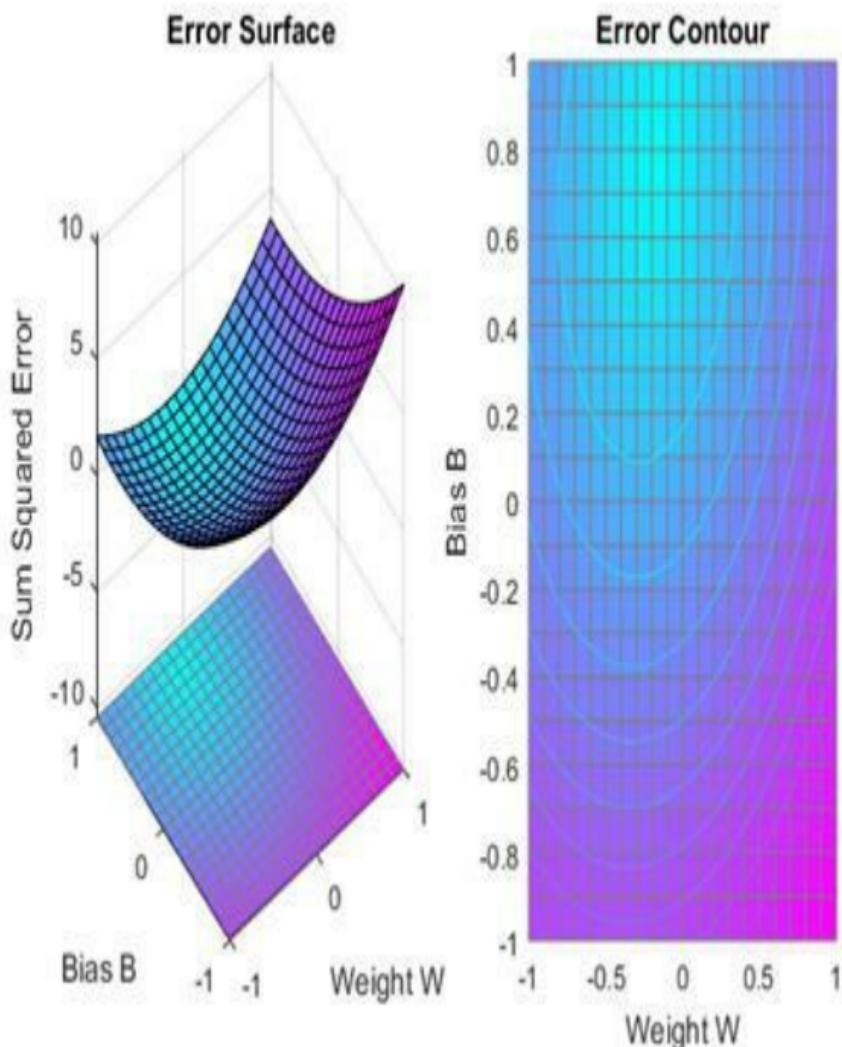
```
w_range = -1:0.1:1;
```

```
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.

$$A = \text{net}(X)$$
$$E = T - A$$
$$\text{SSE} = \text{sumsqr}(E)$$

A =

$$\begin{matrix} 0.5000 & 1.0000 \end{matrix}$$

E =

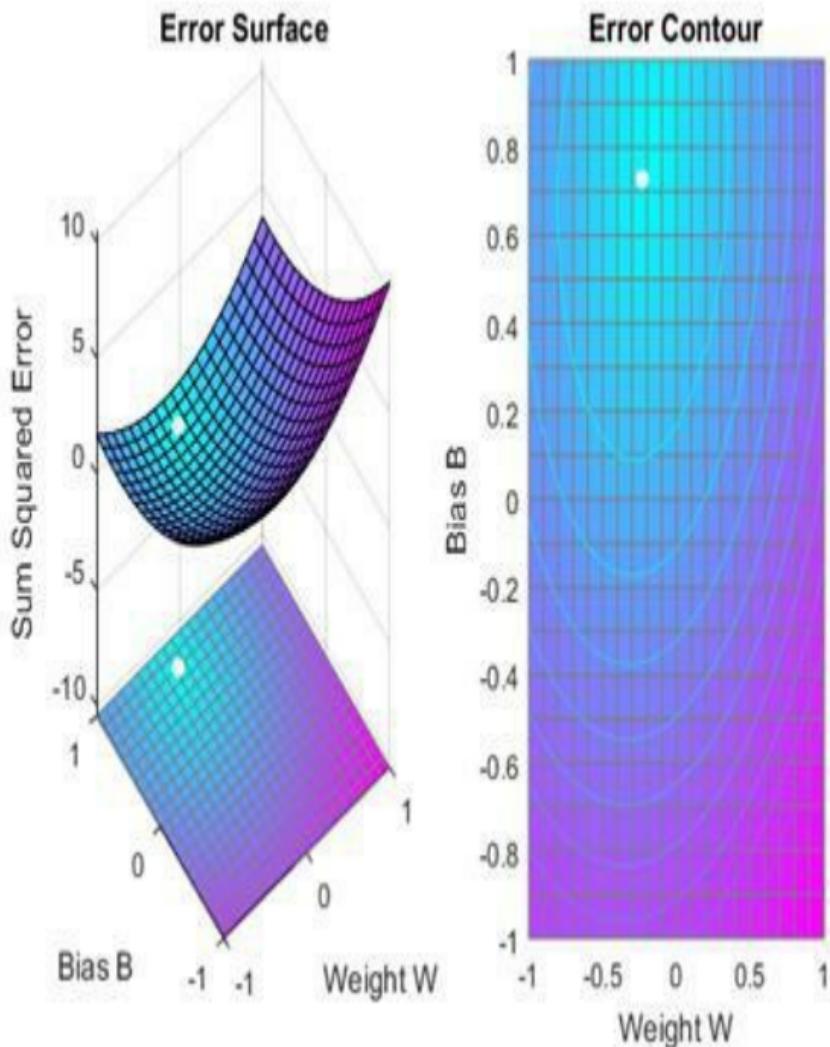
0 0

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

4.8.2 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

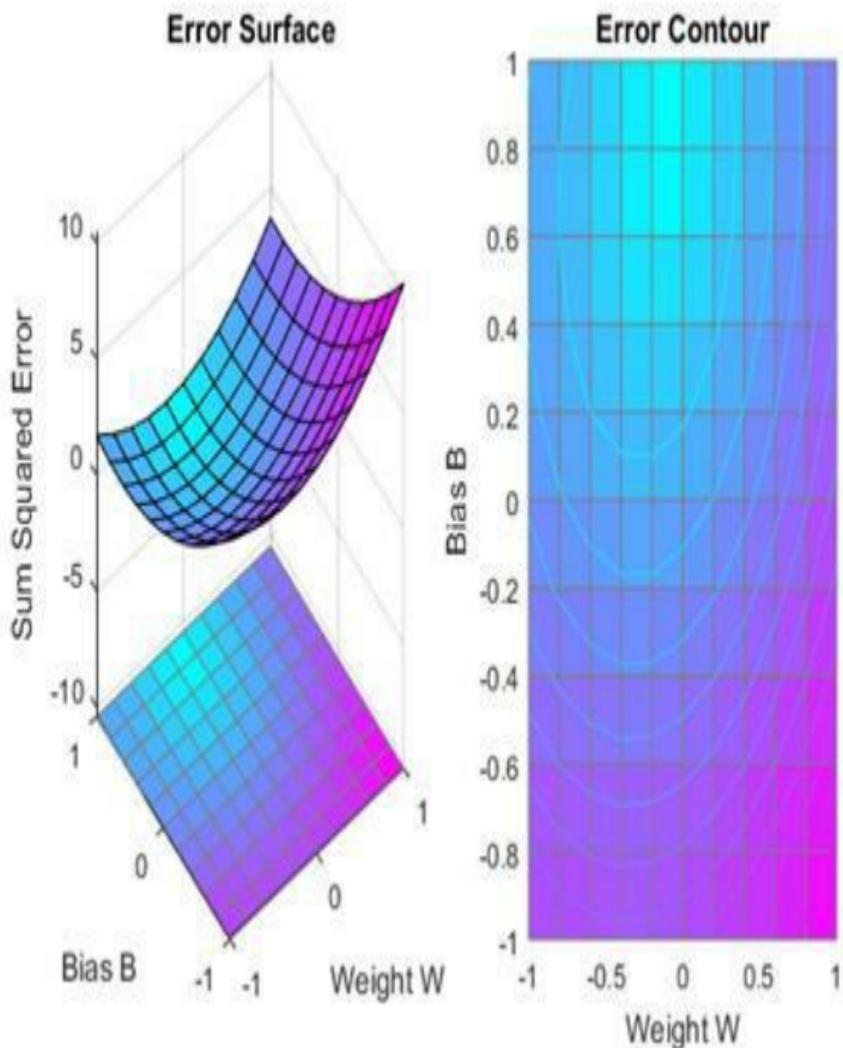
weight and bias values are those that result in the lowest point on the error surface.

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.

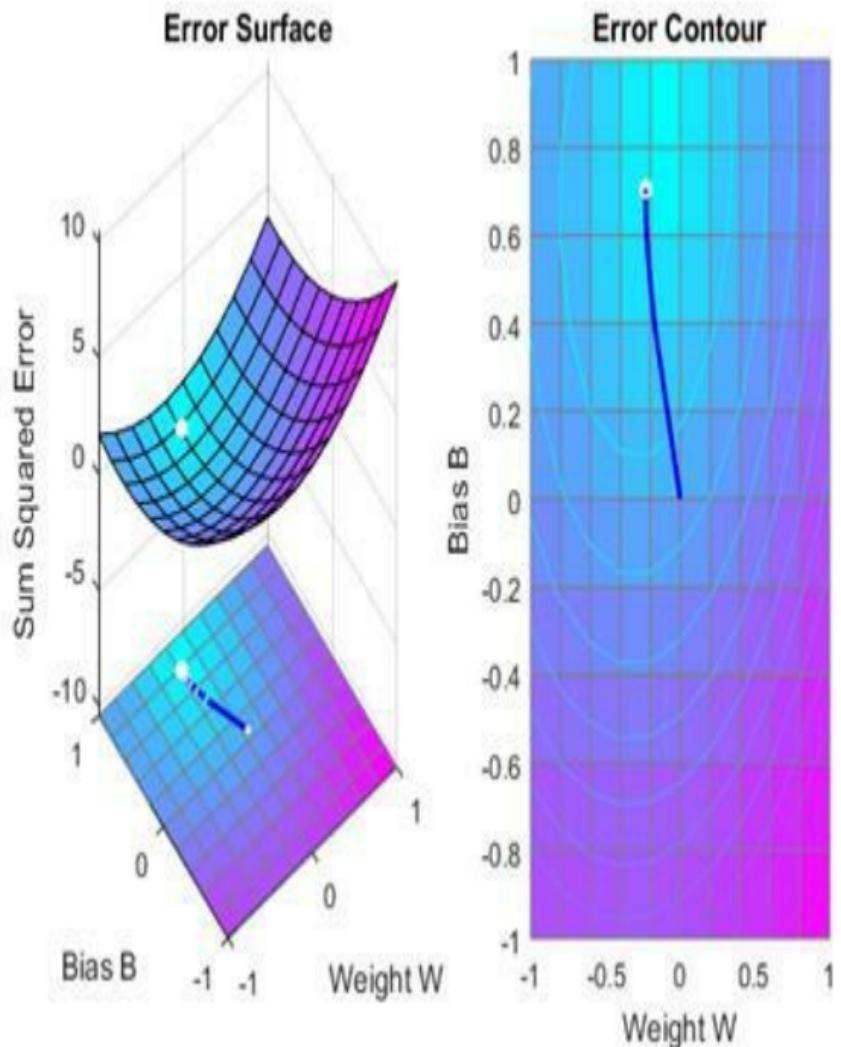
```
net.trainParam.goal = .001;
```

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;
```

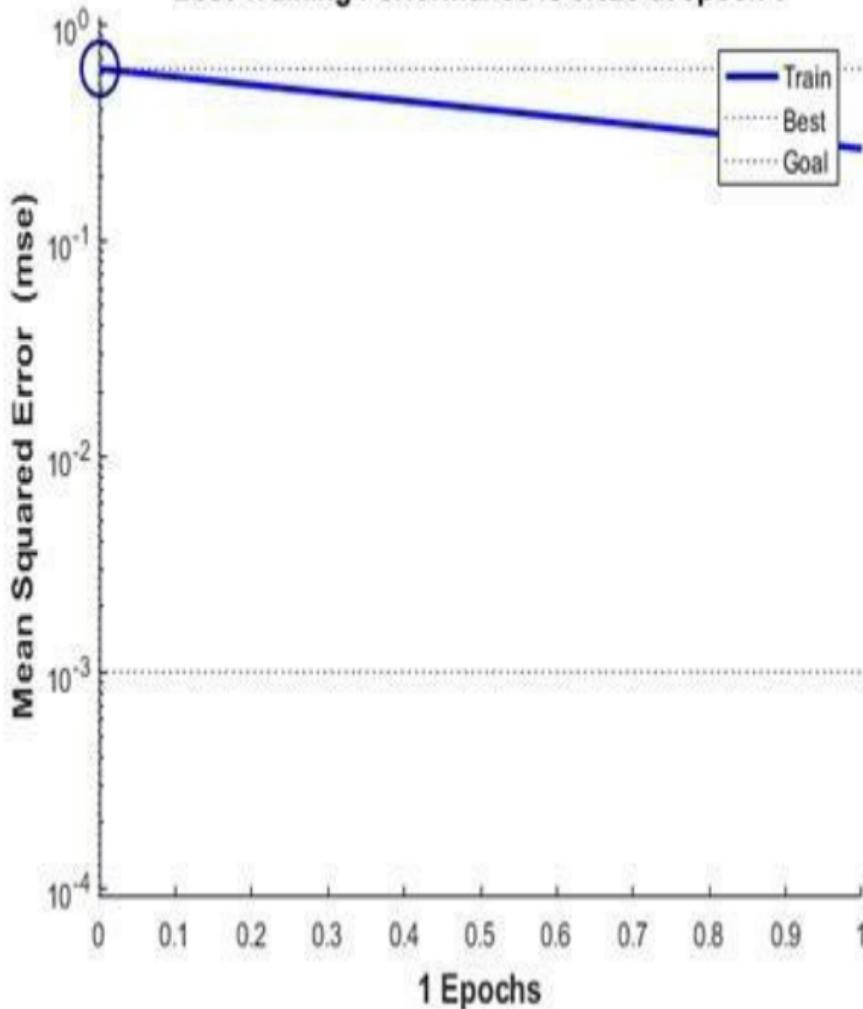


The train function outputs the trained network and y history of the training

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

4.8.3 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;
```

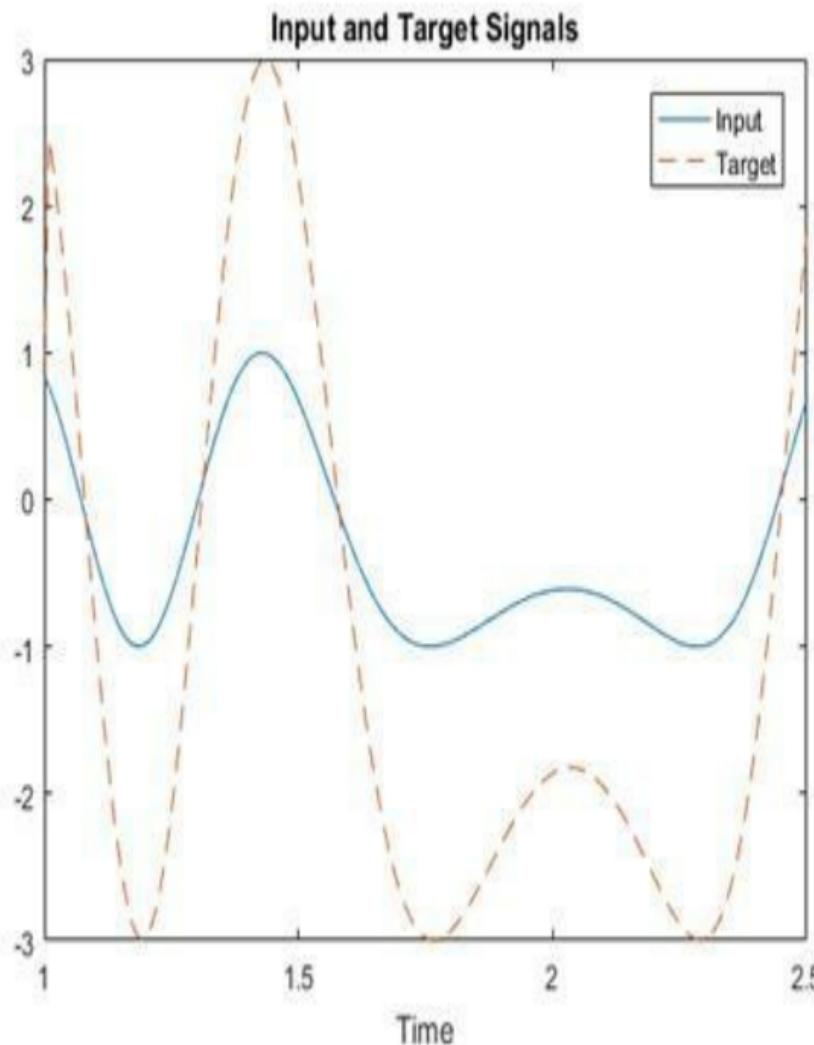
```
X = sin(sin(time).*time*10);
```

```
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'})
```



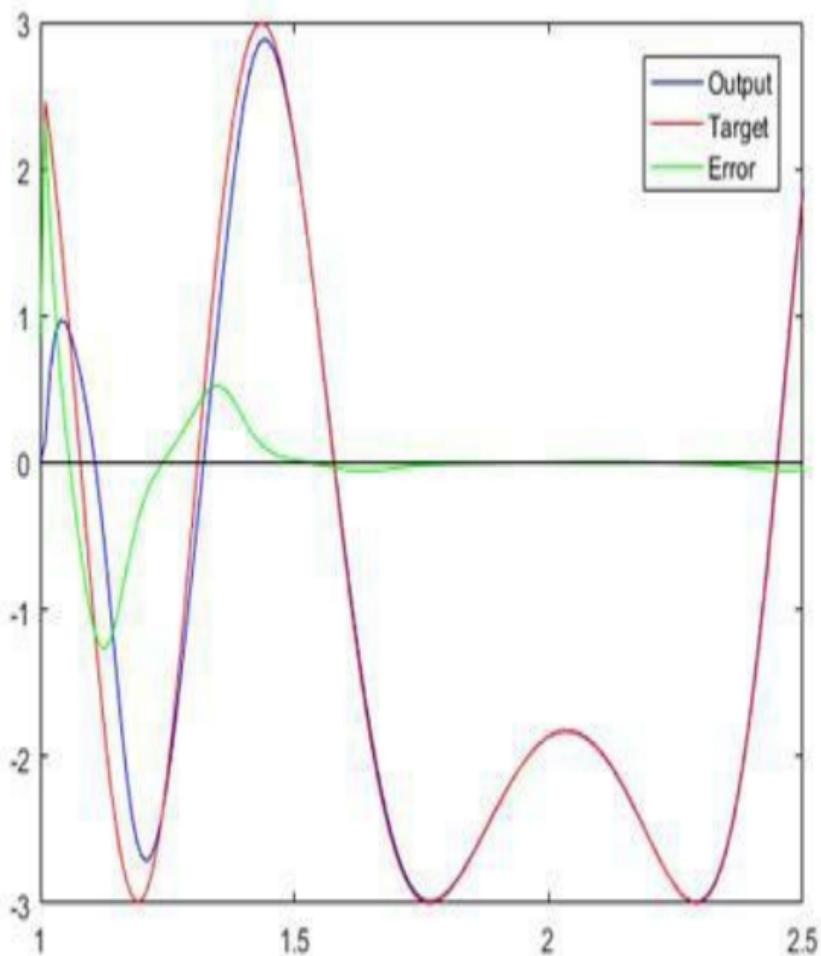
The linear network must have tapped delay in order to learn the time-shifted

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'})
```



4.8.4 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.

$$X = [+1.0 \quad +1.5 \quad +3.0]$$

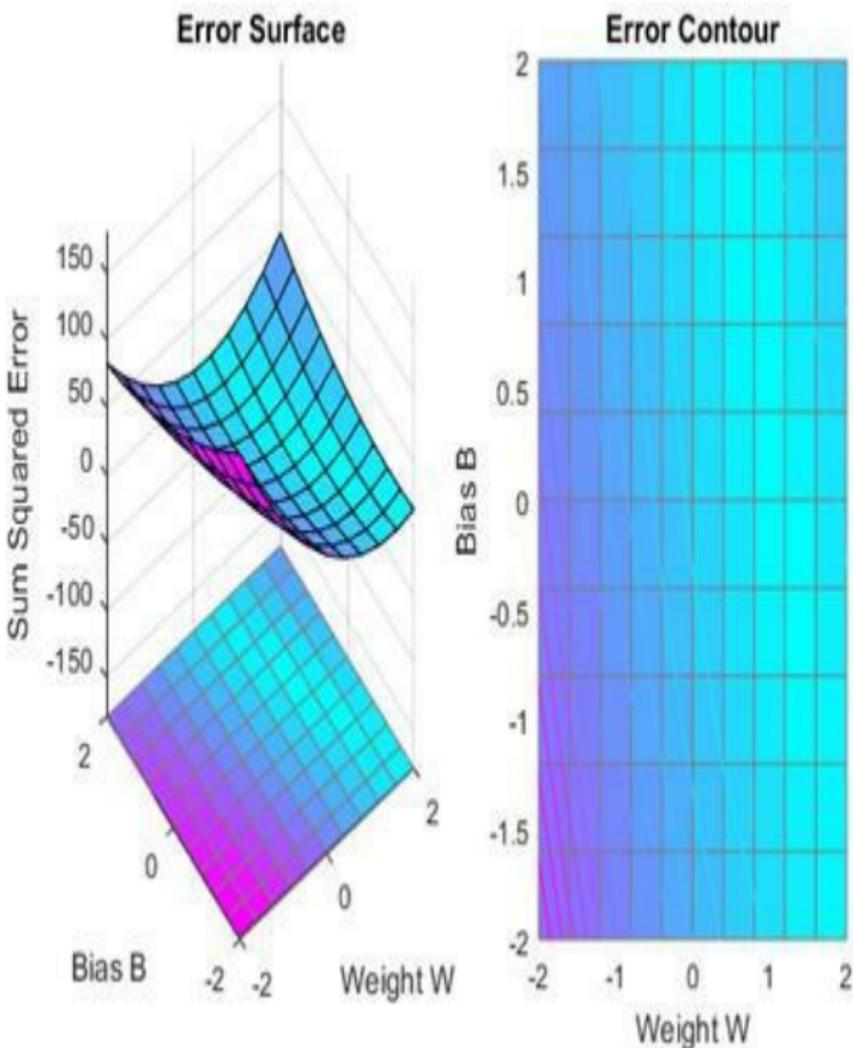
-1.2] ;

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 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.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];
```

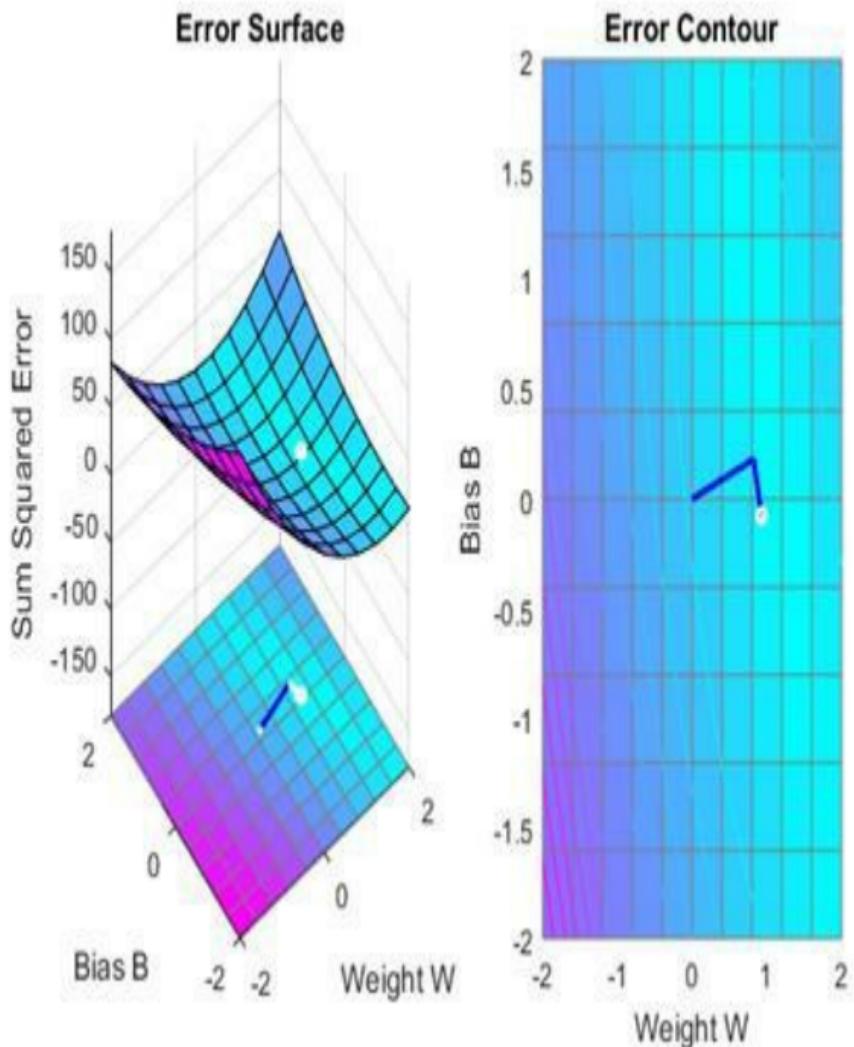
```
else
```

```
break
```

```
end
```

```
end
```

```
tr=r;
```



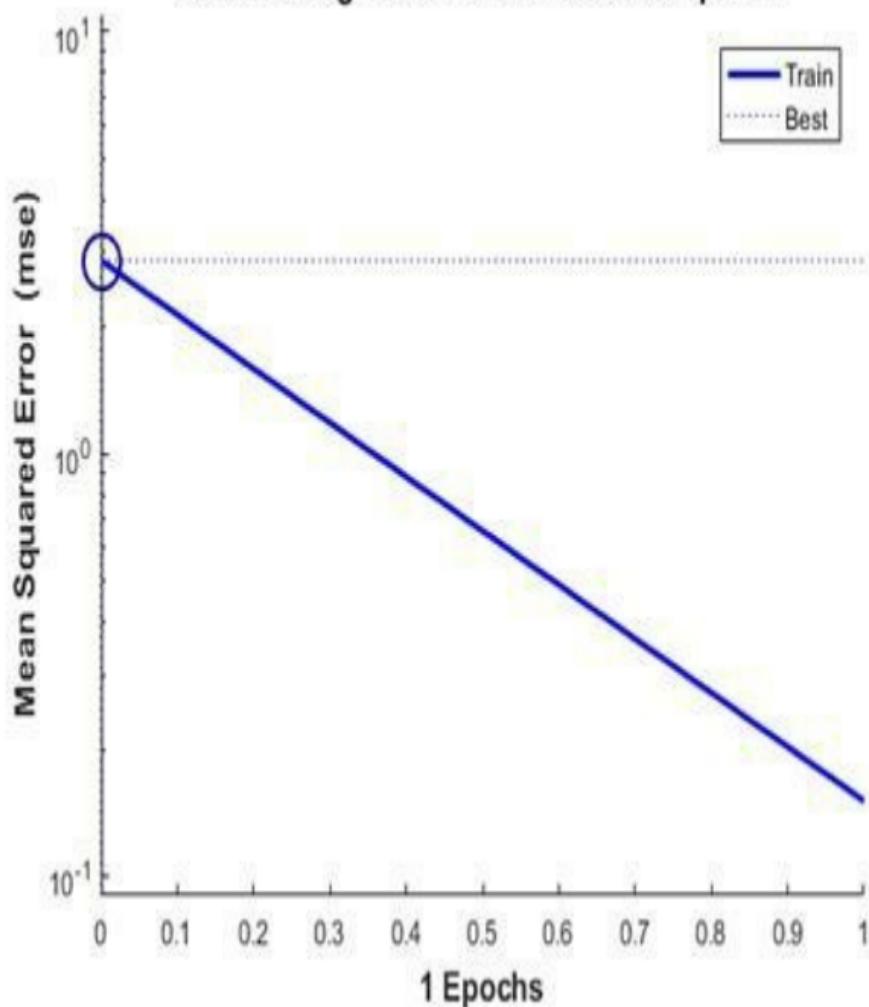
The `train` function outputs the trained network and `y` history of the training

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 y 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

4.8.5 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.

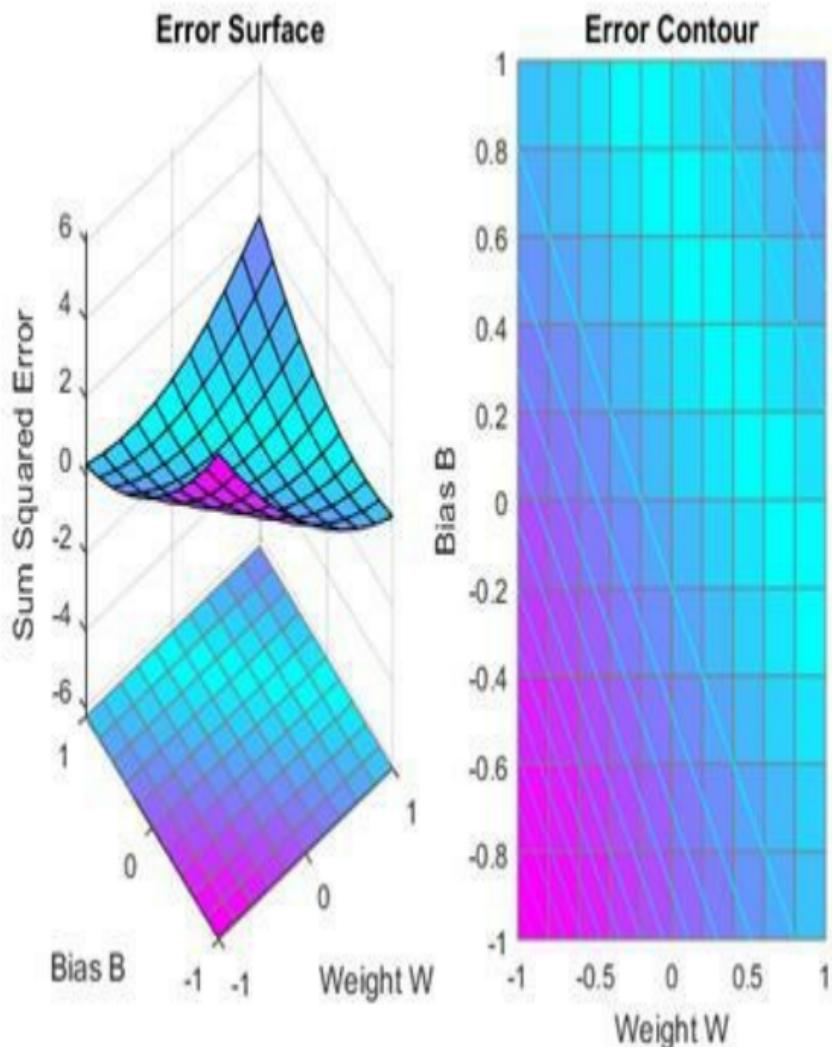
X = [+1.0];

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];

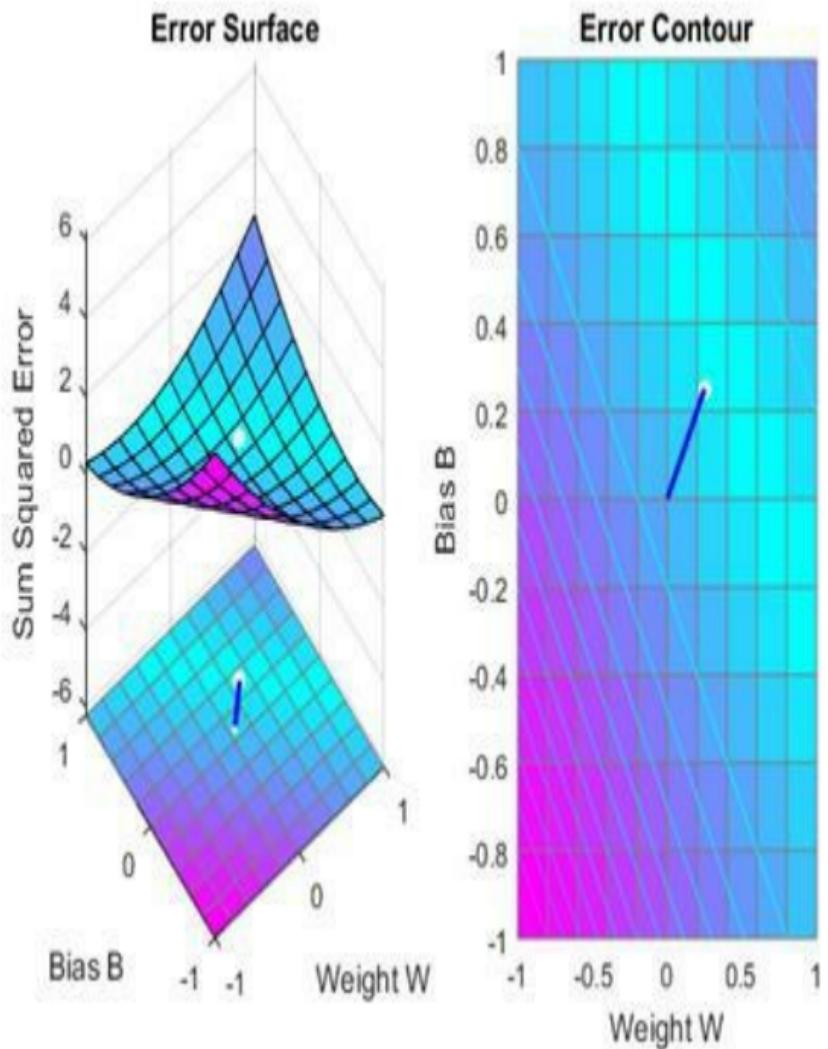
else

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.

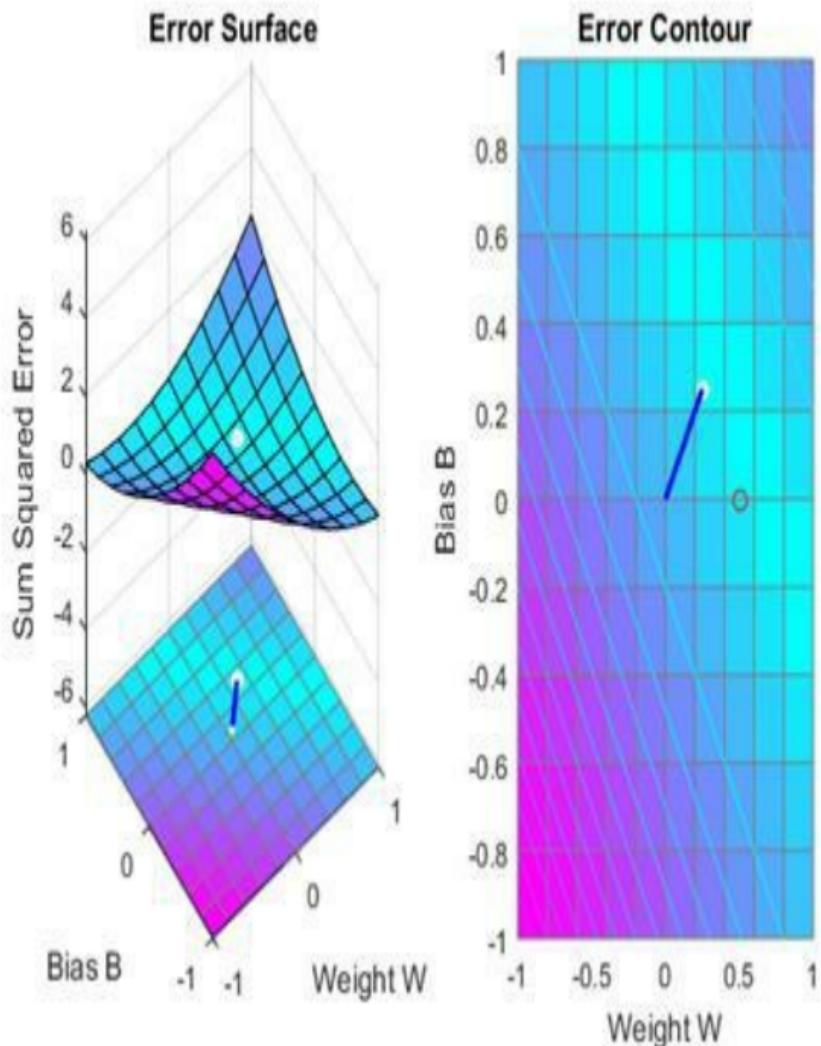
```
solvednet =
```

```
newlind(X, T);
```

```
hold on;
```

```
plot(solvednet.IW{1,
```

```
hold off;
```

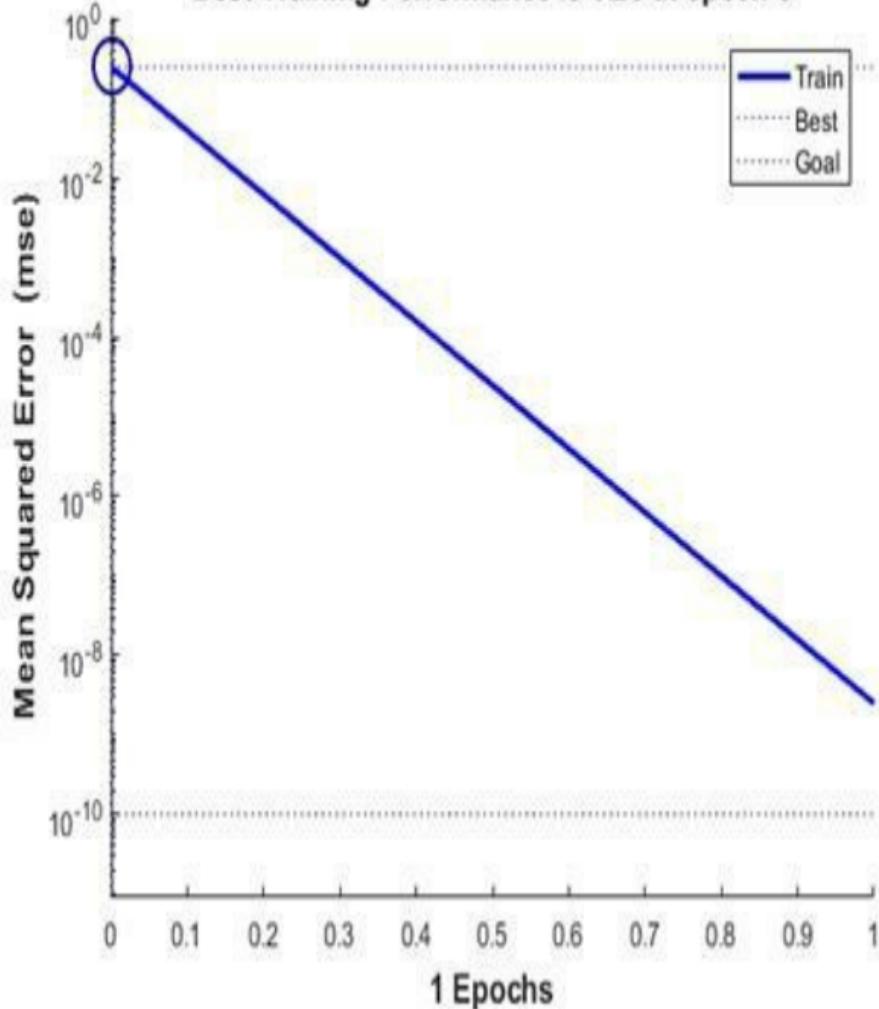


The `train` function outputs the trained network and y history of the training

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

4.8.6 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.

$$X = [\begin{matrix} 1.0 & 2.0 & 3.0 \\ \end{matrix}; \dots]$$

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 y 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

4.8.7 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.

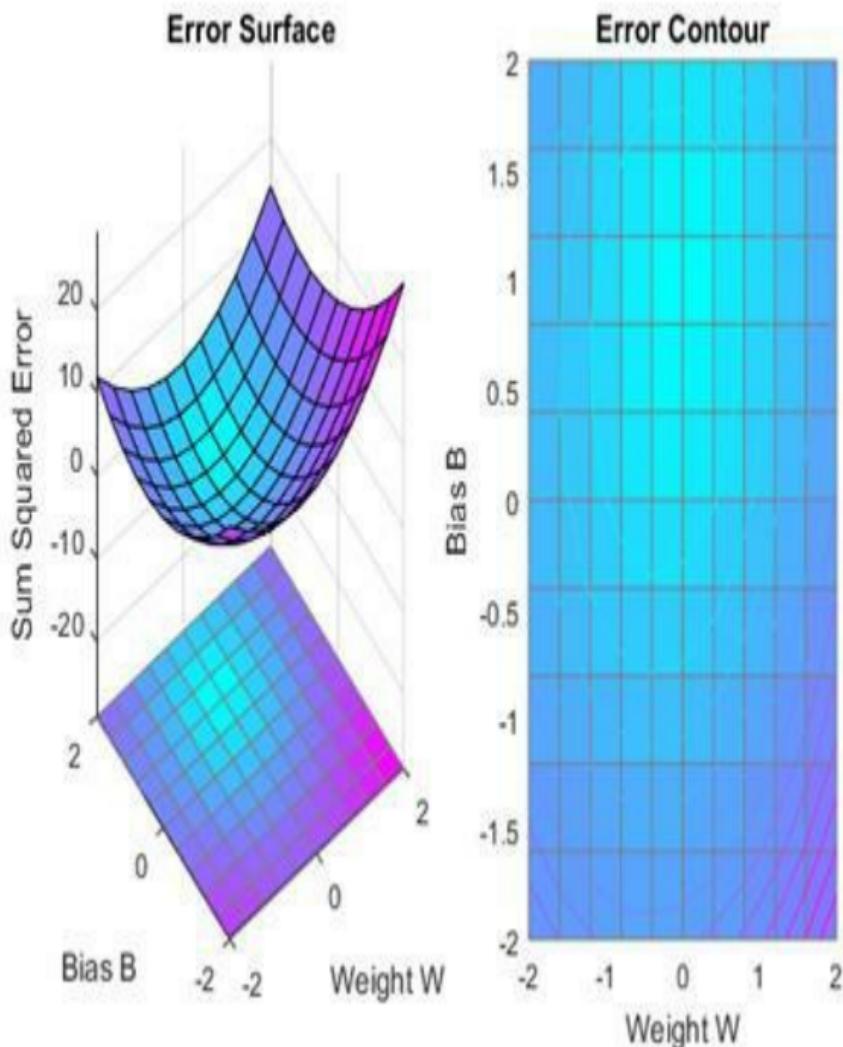
```
w_range = -2:0.4:2;
```

```
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;
```

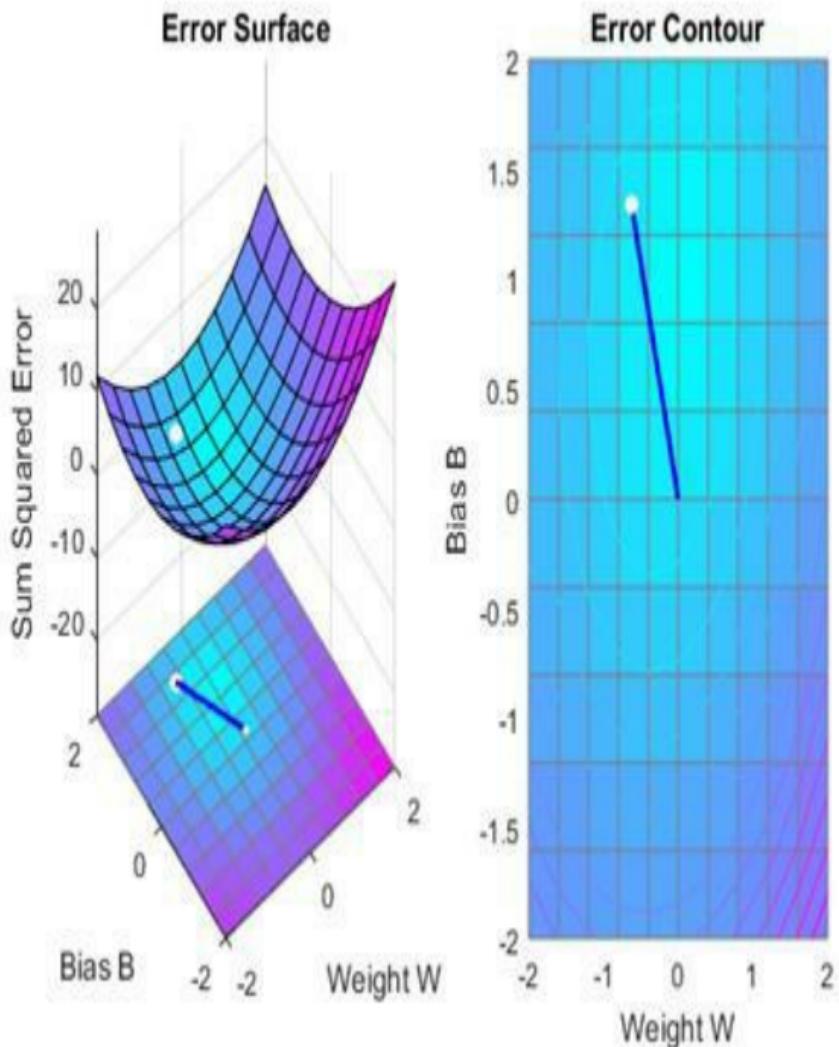
```
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
```

```
[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;
```

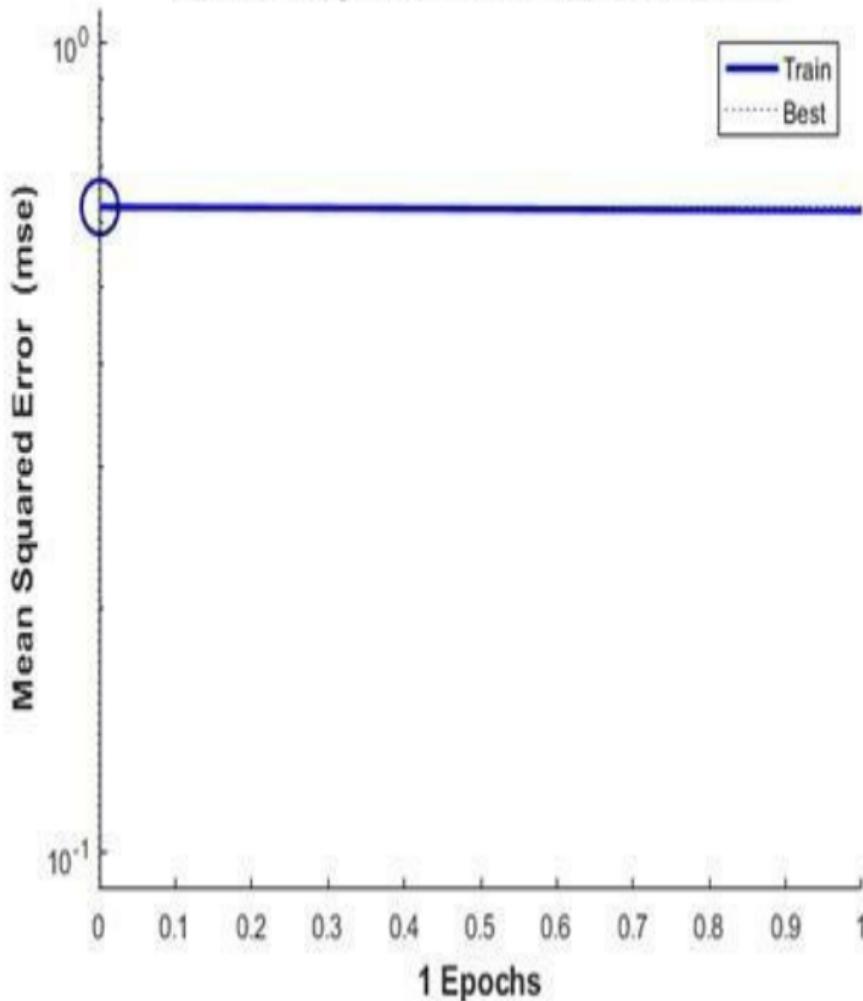


The train function outputs the trained network and a history of the training

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

PERCEPTRON NEURAL NETWORKS

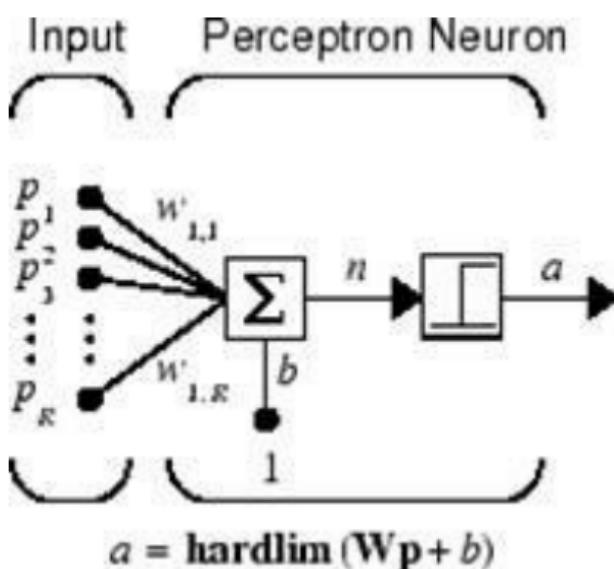
5.1 INTRODUCTION

Rosenblatt created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable

networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

5.2 NEURON MODEL

A perceptron neuron, which uses the hard-limit transfer function [hardlim](#), is shown below.

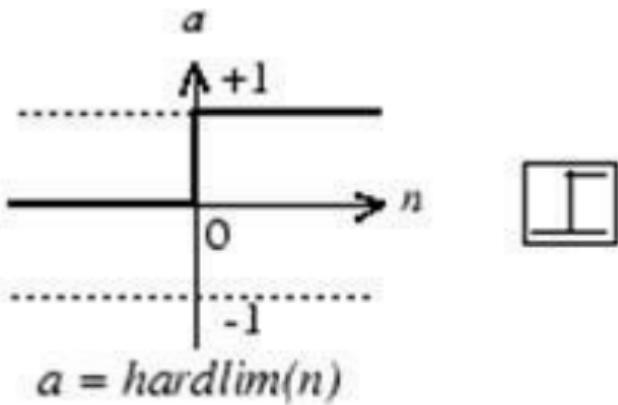


Where

R = number of elements in input vector

Each external input is weighted with an appropriate weight w_{1j} , and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an

input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.

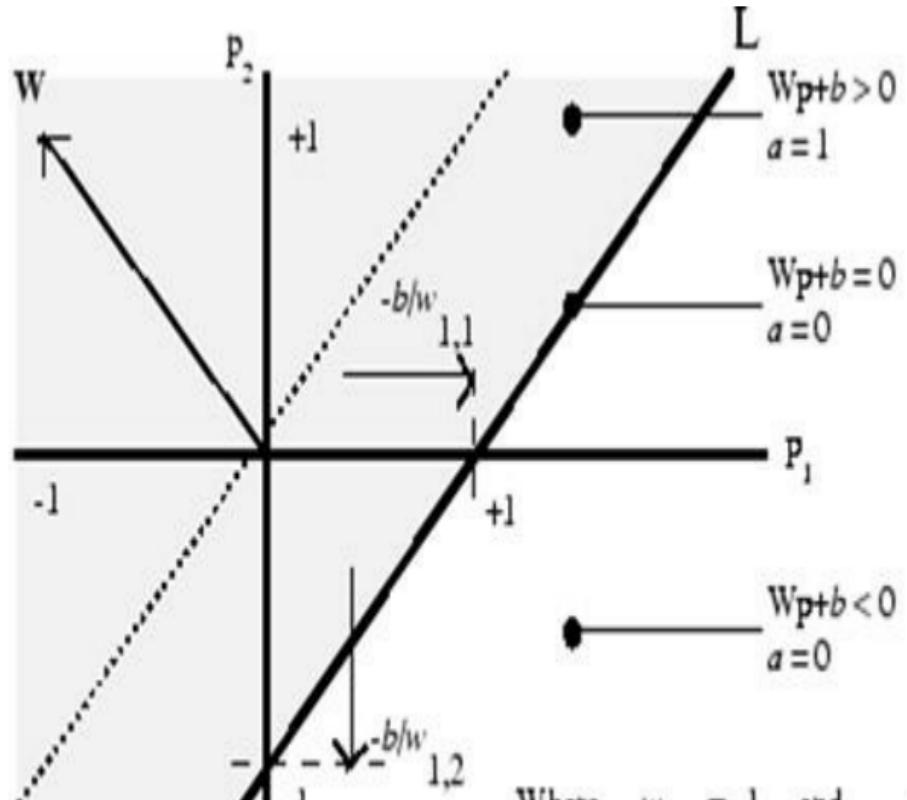


Hard-Limit Transfer Function

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

produces a 0.

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



Where... $w_{1,1} = -1$ and $b = +1$

$$w_{1,2} = +1$$

Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$. This line is perpendicular

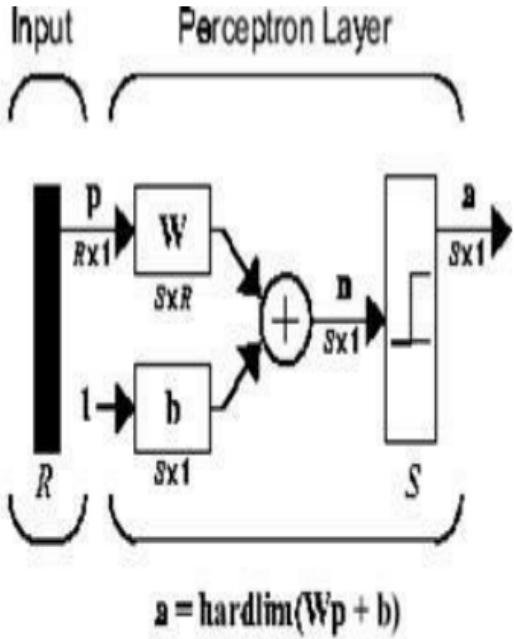
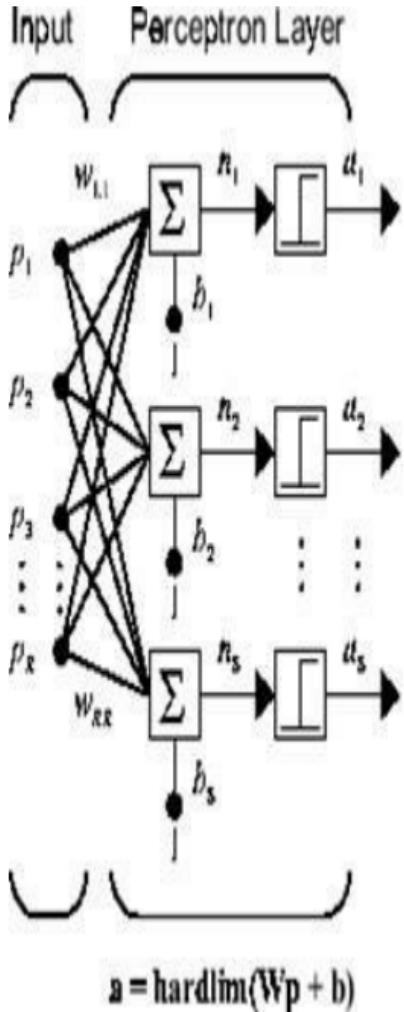
to the weight matrix \mathbf{W} and shifted according to the bias b . Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias

allows the decision boundary to be shifted away from the origin, as shown in the plot above.

5.3 PERCEPTRON ARCHITECTURE

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



Where

R = number of elements in input

S = number of neurons in layer

The perceptron learning rule described shortly is capable of training only a

single layer. Thus only one-layer networks are considered here. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed in Limitations and Cautions.

5.4 CREATE A PERCEPTRON

You can create a perceptron with the following:

```
net = perceptron;
```

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

where input arguments are as follows:

- P is an R-by-Q matrix of Q input vectors of R elements each.
- T is an S-by-Q matrix of Q target vectors of S elements each.

Commonly, the [hardlim](#) function is used in perceptrons, so it is the default.

The following commands create a perceptron network with a single one-element input vector with the values 0 and 2, and one neuron with outputs that can be either 0 or 1:

```
P = [0 2];
```

```
T = [0 1];
```

```
net = perceptron;
```

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

You can see what network has been created by executing the following command:

```
inputweights =  
net.inputweights {1,1}
```

which yields

```
inputweights =  
    delays: 0  
  
    initFcn: 'initzero'  
  
    learn: true  
  
    learnFcn: 'learnp'  
  
    learnParam: (none)  
  
    size: [1 1]  
  
    weightFcn: 'dotprod'  
  
    weightParam: (none)  
  
    userdata: (your custom info)
```

The default learning function is [learnp](#).
The net input to the [hardlim](#) transfer function is [dotprod](#), which generates the

product of the input vector and weight matrix and adds the bias to compute the net input.

The default initialization function [initzero](#) is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
```

```
    initFcn: 'initzero'
```

```
    learn: 1
```

```
    learnFcn: 'learnp'
```

```
    learnParam: []
```

size: 1

userdata: [1x1 struct]

You can see that the default initialization for the bias is also 0.

5.5 PERCEPTRON LEARNING RULE (LEARNP)

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

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

where \mathbf{p} is an input to the network and \mathbf{t} is the corresponding correct (target) output. The objective is to reduce the error e , which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response \mathbf{a} and the target vector \mathbf{t} .

The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector p and the associated error e . The target vector t must contain values of either 0 or 1, because perceptrons (with `hardlim` transfer functions) can only output these values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight

vector \mathbf{w} to point toward input vectors to be classified as 1 and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to \mathbf{w} and that properly classifies the input vectors.

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

CASE 1. If an input vector is presented and the output of the neuron is correct ($\mathbf{a} = \mathbf{t}$ and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$), then the weight vector \mathbf{w} is not altered.

CASE 2. If the neuron output is 0 and should have been 1 ($\mathbf{a} = 0$ and $\mathbf{t} = 1$,

and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$), the input vector \mathbf{p} is added to the weight vector \mathbf{w} . This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

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

The perceptron learning rule can be written more succinctly in terms of the

error $e = t - a$ and the change to be made to the weight vector Δw :

CASE 1. If $e = 0$, then make a change Δw equal to 0.

CASE 2. If $e = 1$, then make a change Δw equal to p^T .

CASE 3. If $e = -1$, then make a change Δw equal to $-p^T$.

All three cases can then be written with a single expression:

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

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

input of 1:

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

For the case of a layer of neurons you have

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

and

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

The perceptron learning rule can be summarized as follows:

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

and

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

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

Now try a simple example. Start with a single neuron having an input vector with just two elements.

net = perceptron;

net = configure(net,[0;0],0);

To simplify matters, set the bias equal to 0 and the weights to 1 and -0.8:

net.b{1} = [0];

w = [1 -0.8];

net.IW{1,1} = w;

The input target pair is given by

p = [1; 2];

$t = [1];$

You can compute the output and error with

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

$a =$

0

$e = t - a$

$e =$

1

and use the function [learnp](#) to find the change in the weights.

$dw = \text{learnp}(w, p, [], [], [], [], e, [], [], [], [], []])$

$$dw =$$

$$\begin{matrix} 1 & 2 \end{matrix}$$

The new weights, then, are obtained as

$$w = w + dw$$

$$w =$$

$$\begin{matrix} 2.0000 & 1.2000 \end{matrix}$$

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

in such cases can be separated by a single line.

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

5.6 TRAINING (TRAIN)

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

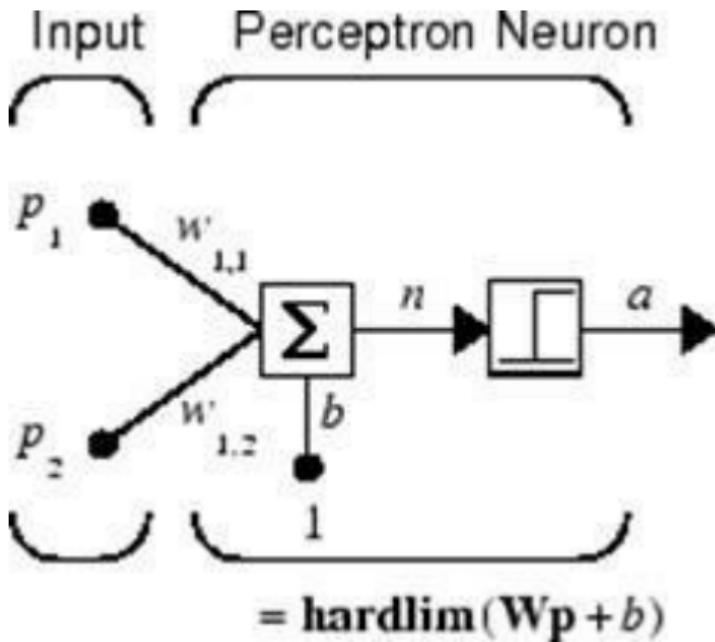
The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as

the inputs are presented.

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

To illustrate the training procedure, work through a simple problem.

Consider a one-neuron perceptron with a single vector input having two elements:



This network, and the problem you are about to consider, are simple enough that you can follow through what is done

with hand calculations if you want.

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

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

Use the initial weights and bias. Denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, the initial values are $\mathbf{W}(0)$ and $b(0)$.

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

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

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(0)\mathbf{p}_1 + b(0)) \\ &= \text{hardlim}\left(\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0\right) = \text{hardlim}(0) = 1\end{aligned}$$

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

$$e = t_1 - \alpha = 0 - 1 = -1$$

$$\Delta \mathbf{W} = e \mathbf{p}_1^T = (-1) [2 \ 2] = [-2 \ -2]$$

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

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

$$\mathbf{W}^{\text{new}} = \mathbf{W}^{\text{old}} + e \mathbf{p}_1^T = \begin{bmatrix} 0 & 0 \end{bmatrix} + [-2 \ -2] = [-2 \ -2] = \mathbf{W}(1)$$

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

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

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(1)\mathbf{p}_2 + b(1)) \\ &= \text{hardlim}\left(\begin{bmatrix} -2 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1\right) = \text{hardlim}(1) = 1\end{aligned}$$

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

You can continue in this fashion, presenting \mathbf{p}_3 next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values $\mathbf{W}(4) = [-3 -1]$ and $b(4) = 0$. To determine whether a satisfactory solution is obtained, make one pass through all input vectors to see if they all produce the desired target values. This is not true for the fourth input, but the algorithm does converge on the sixth presentation of an input. The final values

are

$$\mathbf{W}(6) = [-2 \ -3] \text{ and } b(6) = 1.$$

This concludes the hand calculation.
Now, how can you do this using
the [train](#) function?

The following code defines a
perceptron.

`net = perceptron;`

Consider the application of a single
input

`p = [2; 2];`

having the target

`t = [0];`

Set epochs to 1, so that [train](#) goes

through the input vectors (only one here) just one time.

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

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

The new weights and bias are

$$w = \text{net.iw}\{1,1\}, b = \text{net.b}\{1\}$$

w =

-2 -2

b =

-1

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

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

Apply [train](#) for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = perceptron;
```

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

The input vectors and targets are

$$p = [[2;2] [1;-2] [-2;2] [-1;1]]$$

$$t = [0 \ 1 \ 0 \ 1]$$

Now train the network with

$$\text{net} = \text{train}(\text{net}, p, t);$$

The new weights and bias are

$$w = \text{net.iw}\{1,1\}, b = \text{net.b}\{1\}$$

$$w =$$

$$\begin{matrix} -3 & -1 \end{matrix}$$

$$b =$$

$$0$$

This is the same result as you got previously by hand.

Finally, simulate the trained network for

each of the inputs.

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

$$a =$$

$$0 \quad 0 \quad 1 \quad 1$$

The outputs do not yet equal the targets, so you need to train the network for more than one pass. Try more epochs. This run gives a mean absolute error performance of 0 after two epochs:

```
net.trainParam.epochs = 1000;
```

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

Thus, the network was trained by the time the inputs were presented on the third epoch. (As you know from hand calculation, the network converges on

the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

$$w = \text{net.iw}\{1,1\}, b = \text{net.b}\{1\}$$

$$w =$$

$$\begin{matrix} -2 & -3 \end{matrix}$$

$$b =$$

$$1$$

The simulated output and errors for the various inputs are

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

$$a =$$

0 1 0

1

error = a-t

error =

0 0 0

0

You confirm that the training procedure is successful. The network converges and produces the correct target outputs for the four input vectors.

The default training function for networks created with perceptron is [trainc](#). (You can find this by executing net.trainFcn.) This training function applies the perceptron

learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with [train](#) will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

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

You might want to try various example programs. For instance, `demop1` illustrates

classification and training of a simple perceptron.

5.7 LIMITATIONS AND CAUTIONS

Perceptron networks should be trained with adapt, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of adapt in this way guarantees that any linearly separable problem is solved in a finite number of training presentations.

As noted in the previous pages, perceptrons can also be trained with the function train. Commonly when train is used for perceptrons, it presents the inputs to the network in batches, and

makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of [train](#) for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) because of the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If

the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. However, it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try demop6. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate

them. A two-neuron network can be found such that its two decision boundaries classify the inputs into four categories.

Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to

overcome. You might want to try demop4 to see how an outlier affects the training.

By changing the perceptron learning rule slightly, you can make training times insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

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

As shown above, the larger an input vector \mathbf{p} , the larger its effect on the weight vector \mathbf{w} . Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be

presented many times to have an effect.

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

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

The normalized perceptron rule is implemented with the function [learnpn](#), which is called exactly like [learnp](#). The normalized perceptron rule function [learnpn](#) takes slightly more time to execute, but reduces the number of epochs considerably if there are outlier input vectors. You might try `demop5` to see how this normalized training rule

works.

5.8 PERCEPTRON EXAMPLES

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.

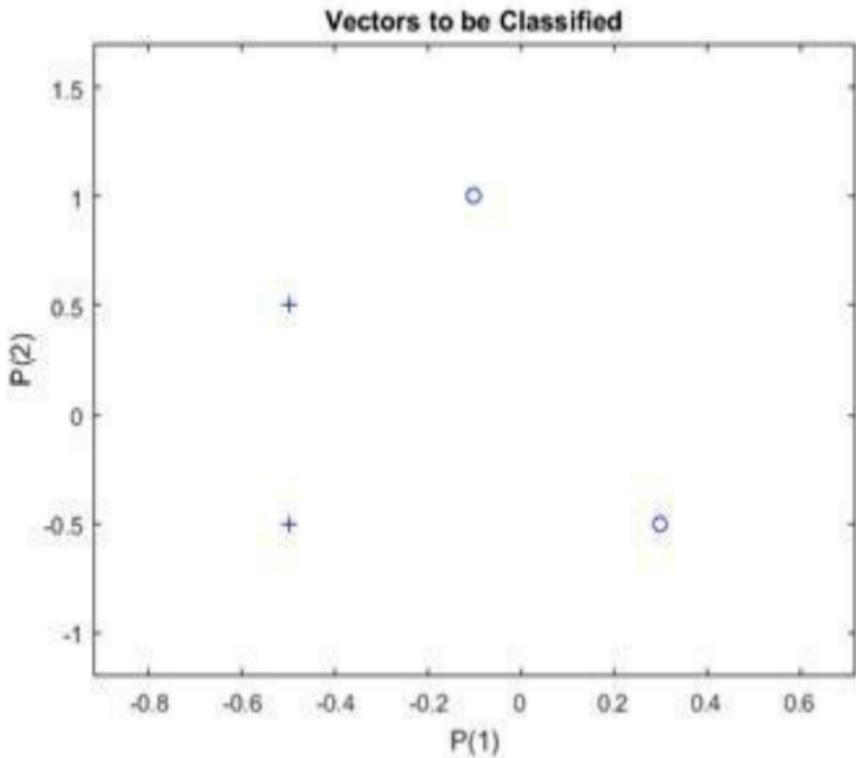
5.8.1 Classification with a 2-Input Perceptron

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 = [ -0.5 -0.5 +0.3  
-0.1; ... ]
```

```
-0.5 +0.5 -0.5  
+1.0] ;  
  
T = [1 1 0 0] ;  
  
plotpv(X,T) ;
```



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 =
```

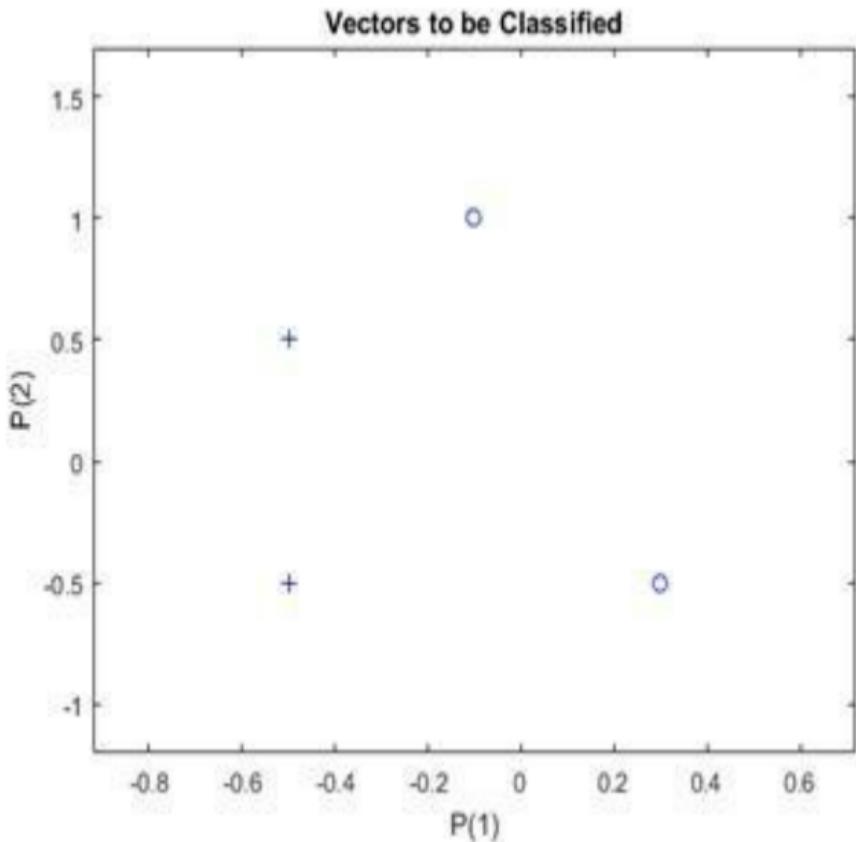
```
configure(net, X, T);
```

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
```



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,
```

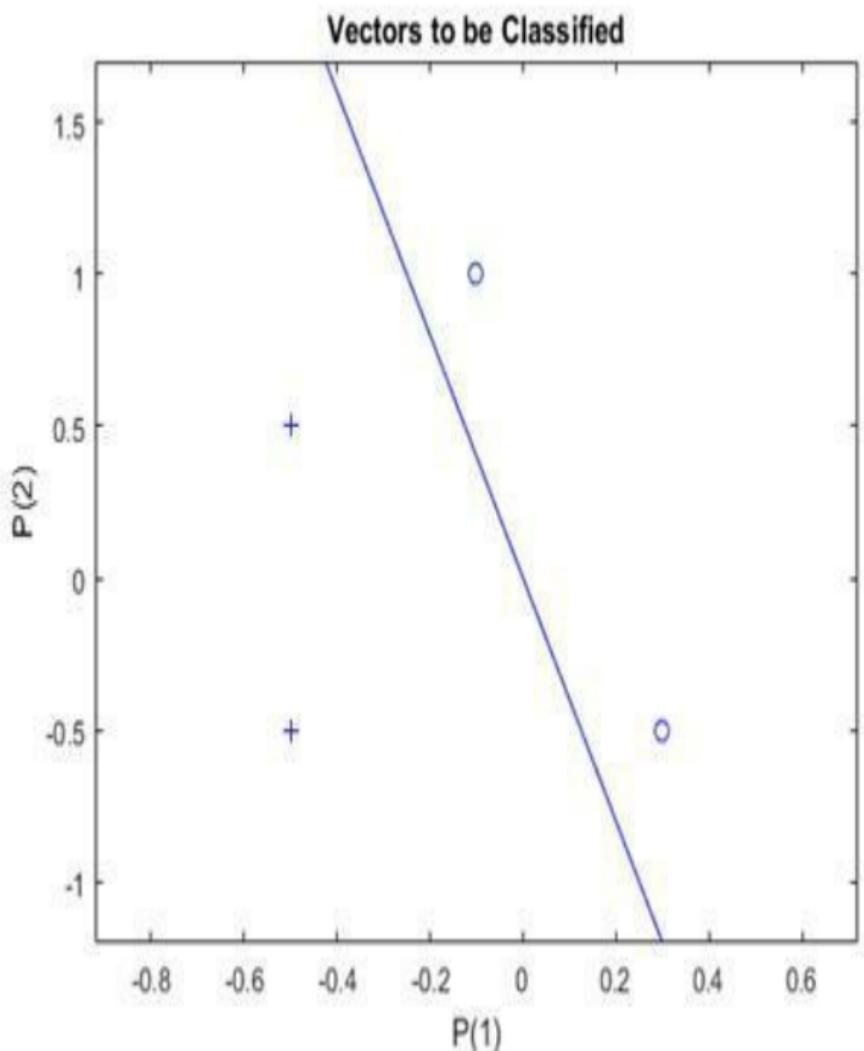
```
TT =
```

```
repmat (con2seq(T), 1,
```

```
net =
```

```
adapt (net, XX, TT);
```

```
plotpc(net.IW{1}, net
```



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.

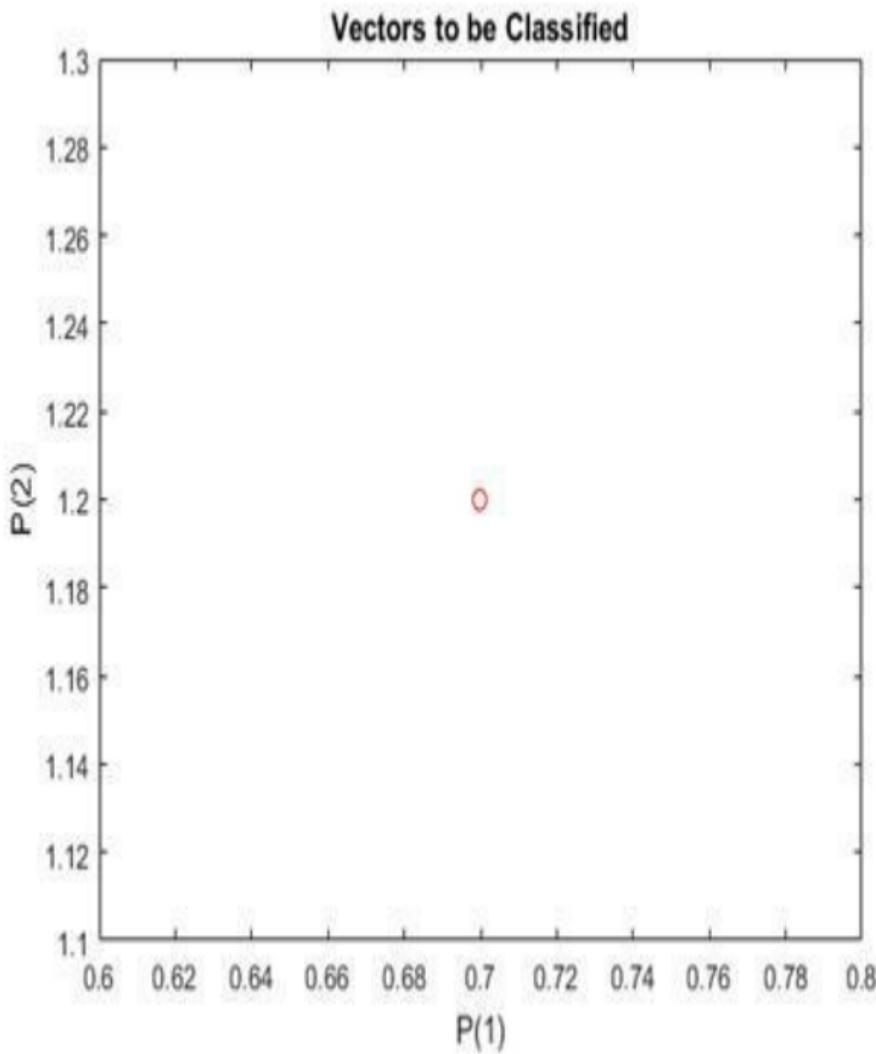
```
x = [0.7; 1.2];
```

```
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.

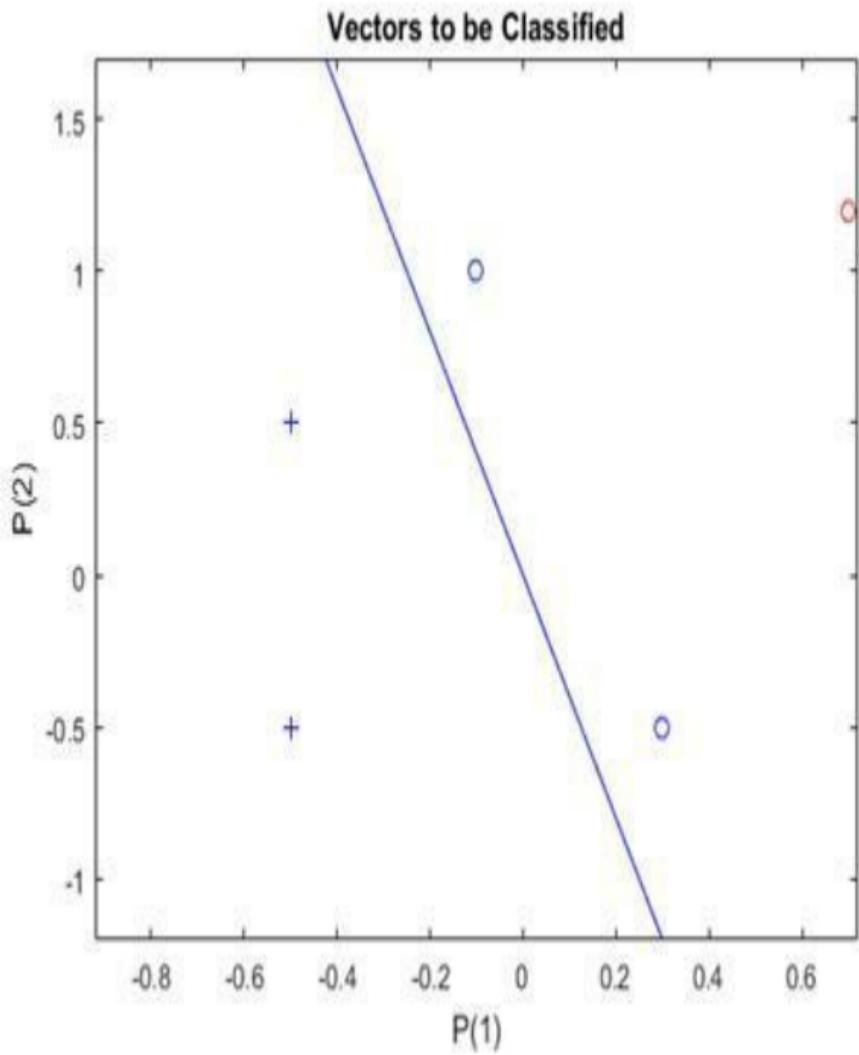
The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus).

hold on;

plotpv(X, T);

plotpc(net.IW{1}, net

hold off;



5.8.2 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.

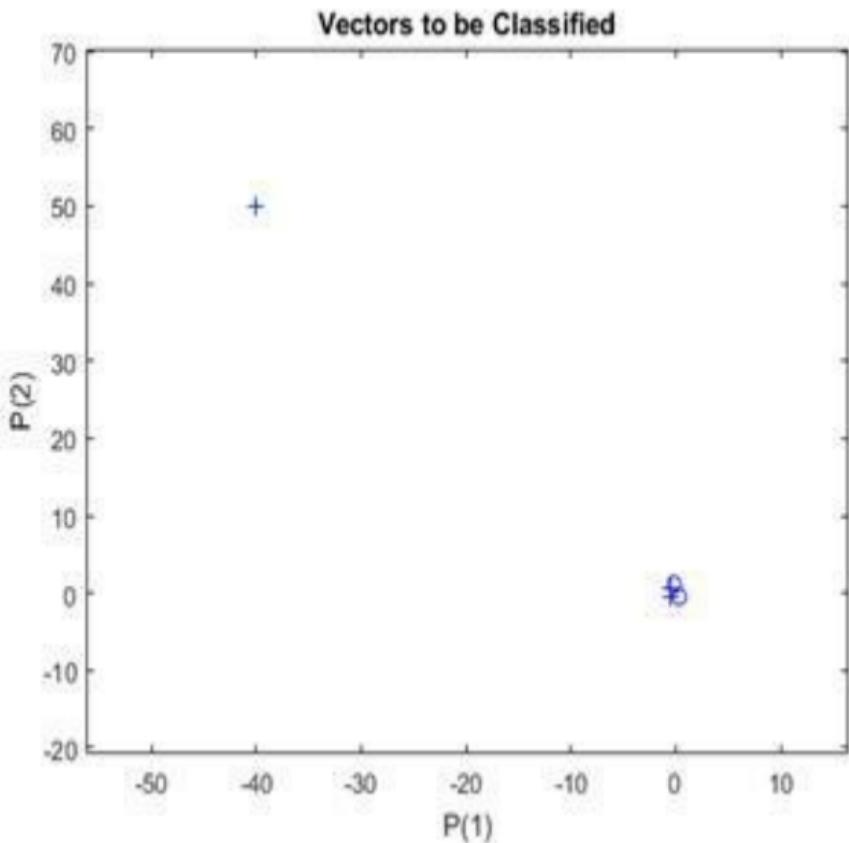
Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

$$X = \begin{bmatrix} -0.5 & -0.5 & +0.3 \\ -0.1 & -40; & -0.5 & +0.5 \end{bmatrix}$$

```
-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 =
```

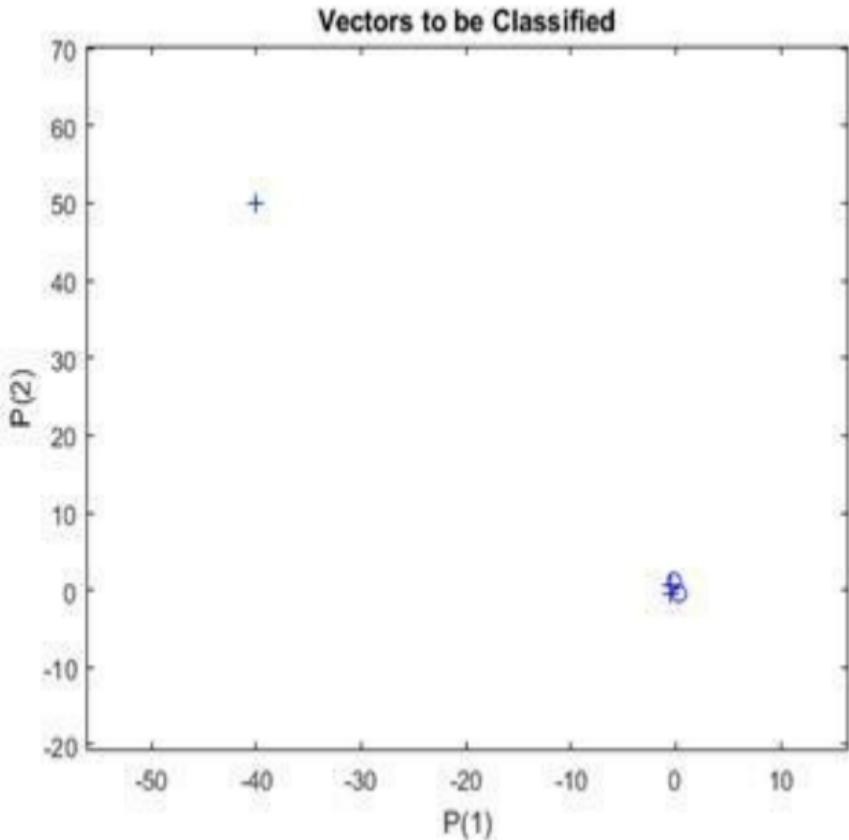
```
configure(net,X,T);
```

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.

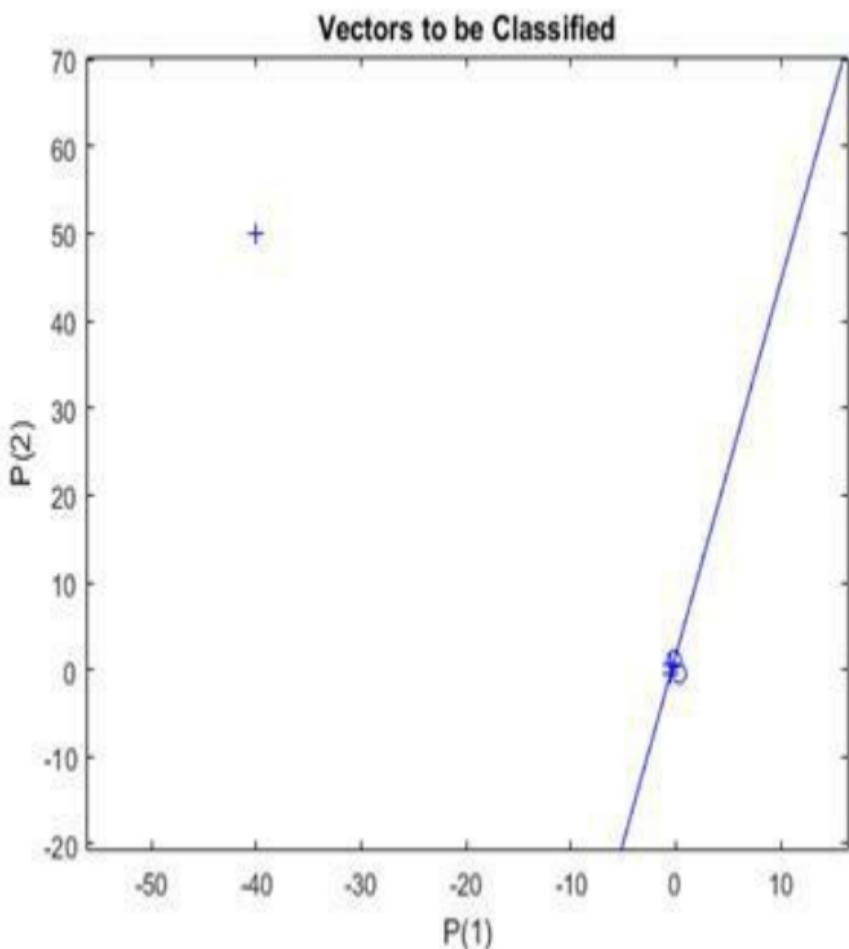
E = 1;

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.

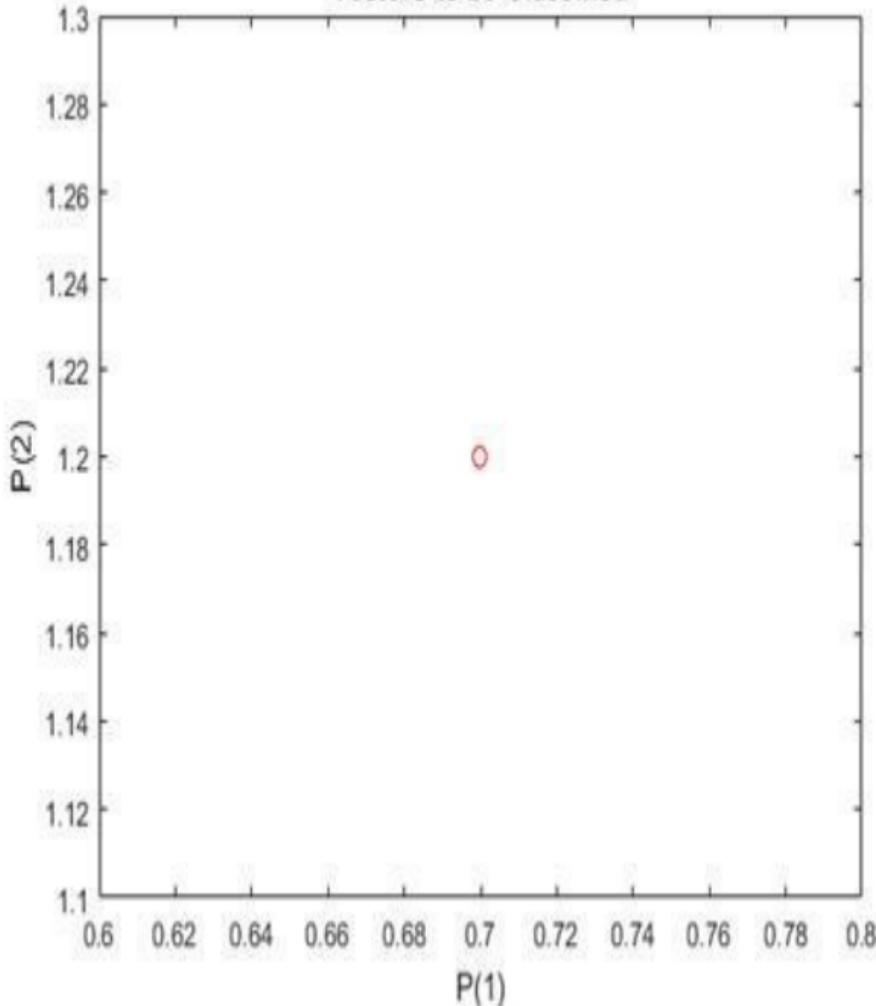
Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

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

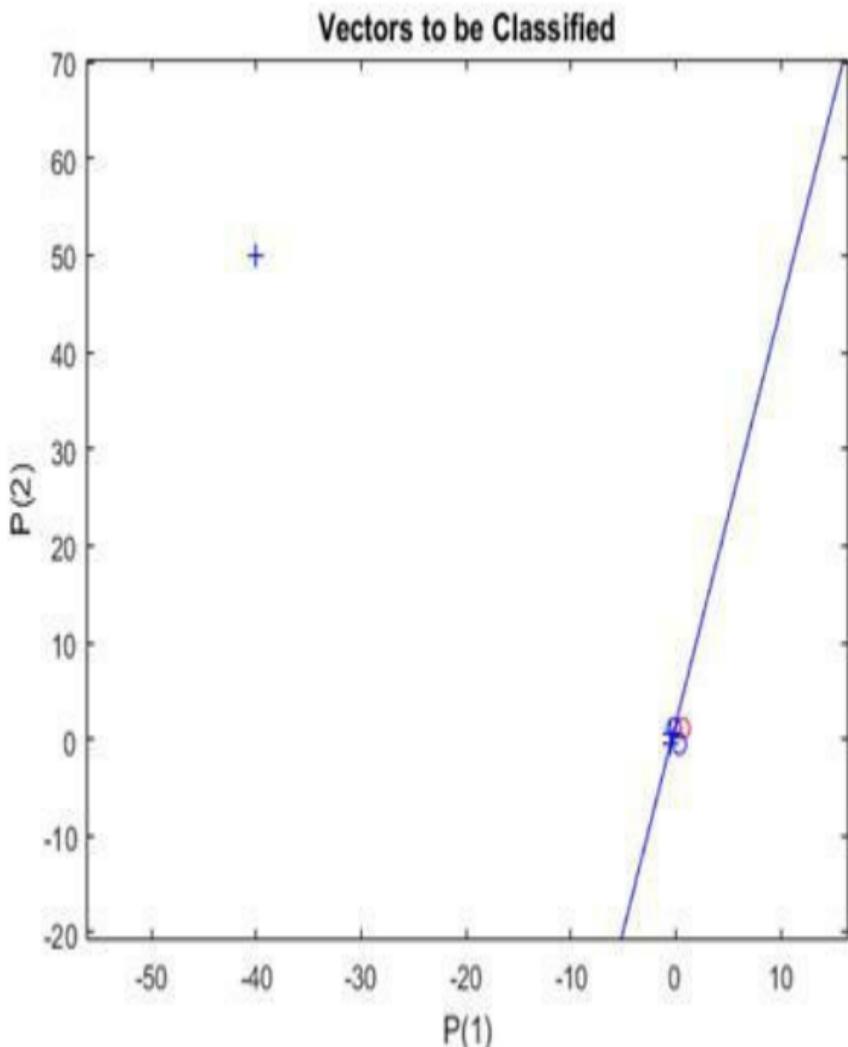
classification line to the plot.

```
hold on;
```

```
plotpv(X,T);
```

```
plotpc(net.IW{1},net
```

```
hold off;
```

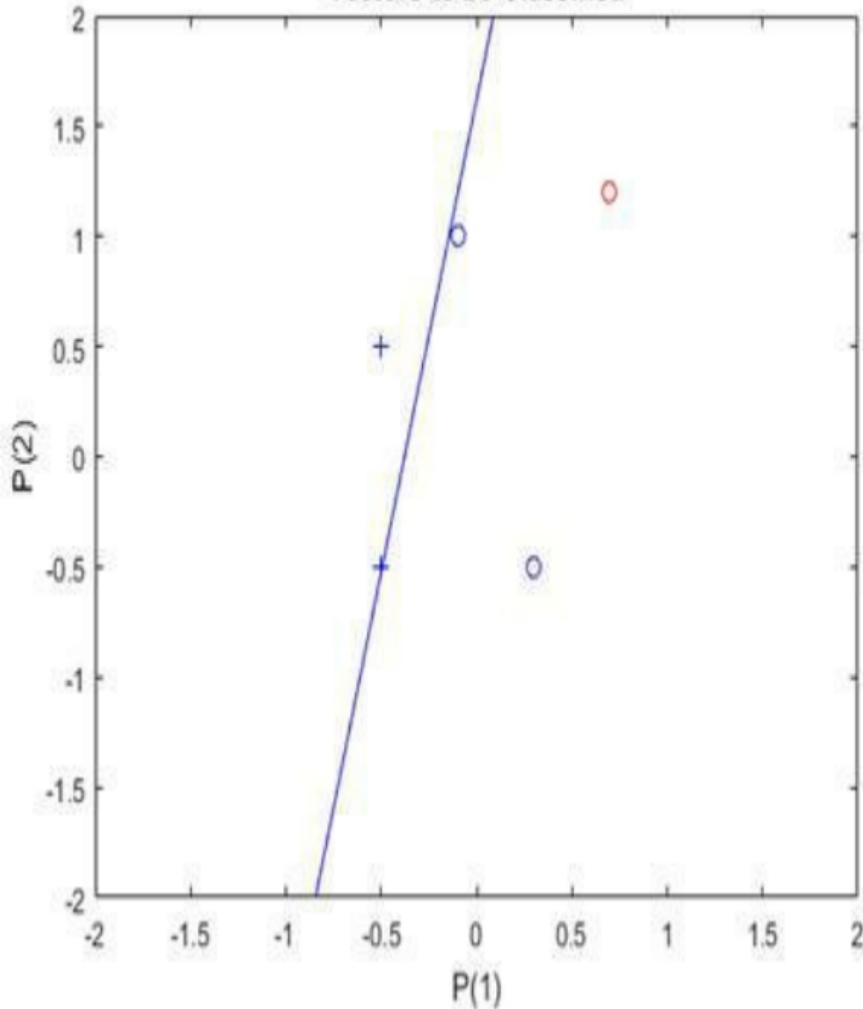


Finally, zoom into the area of interest.
The perceptron correctly classified our

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

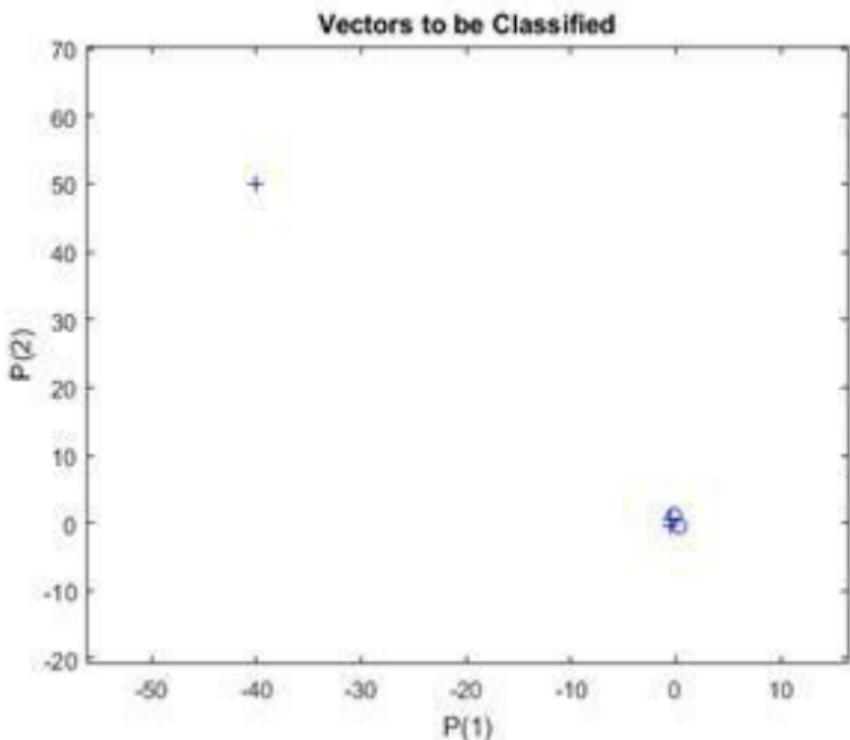


5.8.3 Normalized Perceptron Rule

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);
```

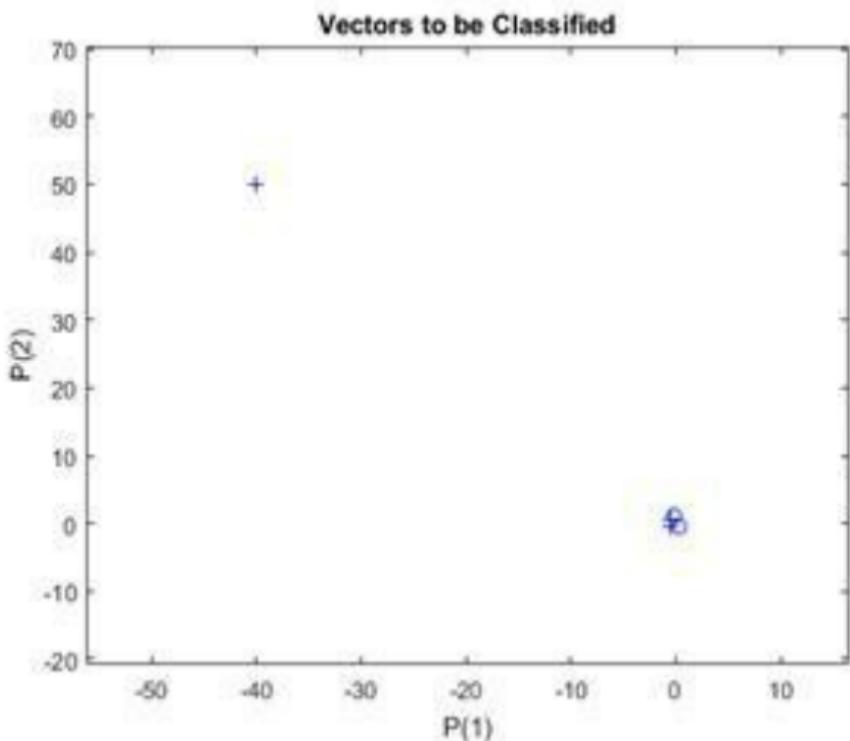
Add the neuron's initial attempt at classification to the plot.

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.

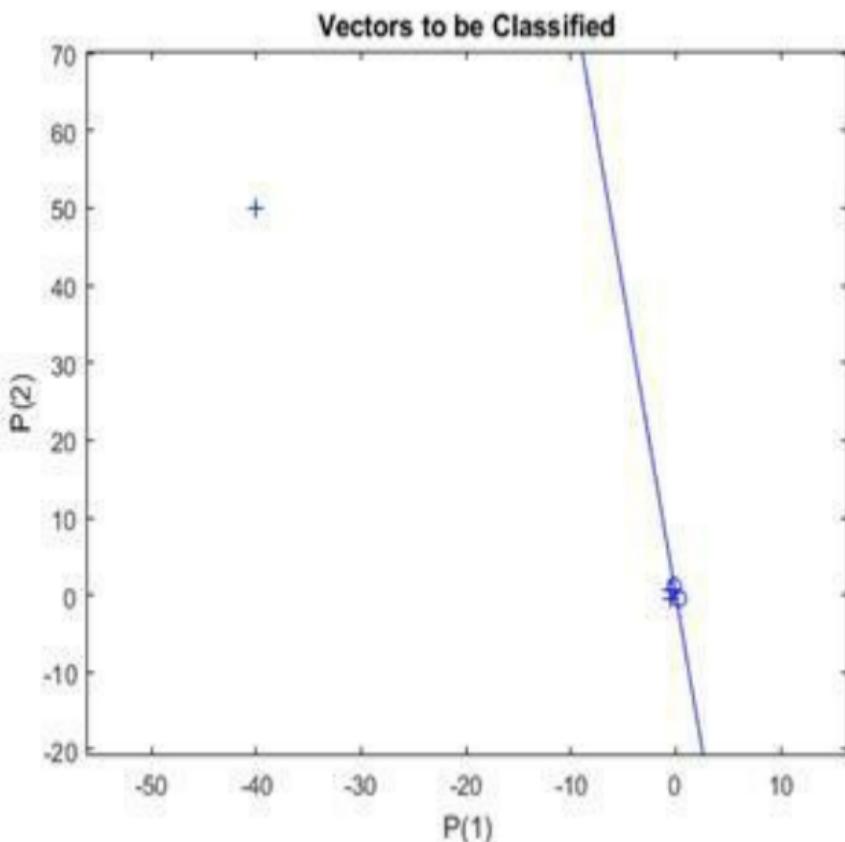
E = 1;

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].

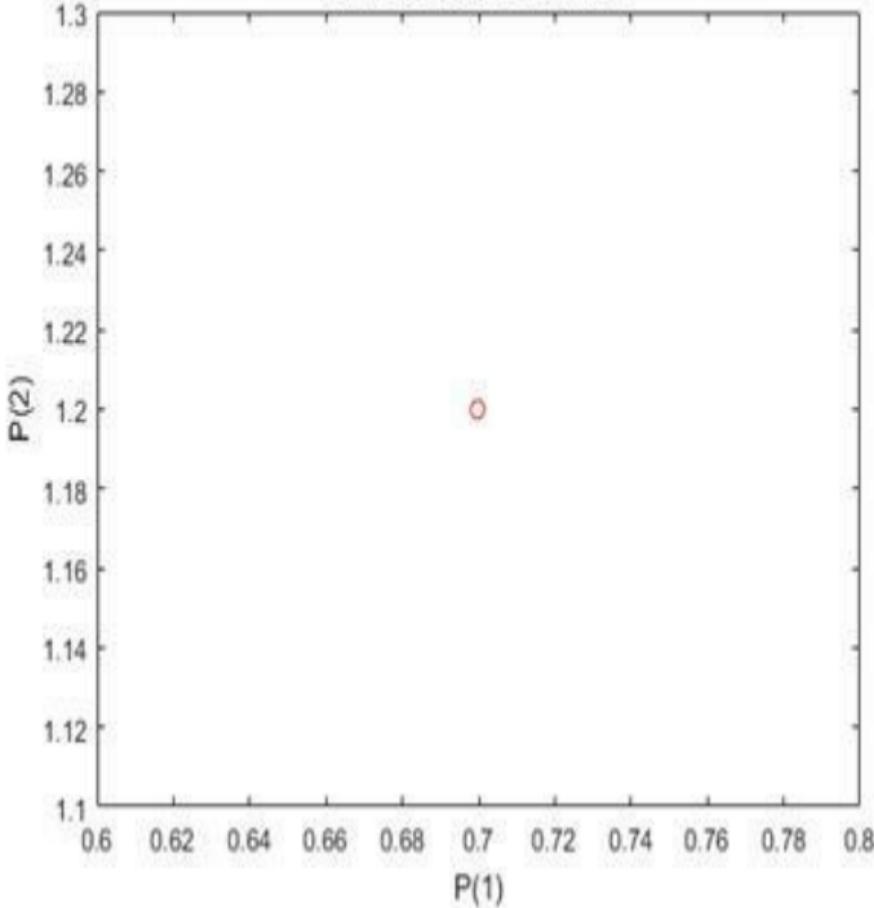
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
x = [0.7; 1.2];
```

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

```
plotpv(x,y);  
  
circle =  
findobj(gca,'type','  
  
circle.Color =  
'red';
```

Vectors to be Classified



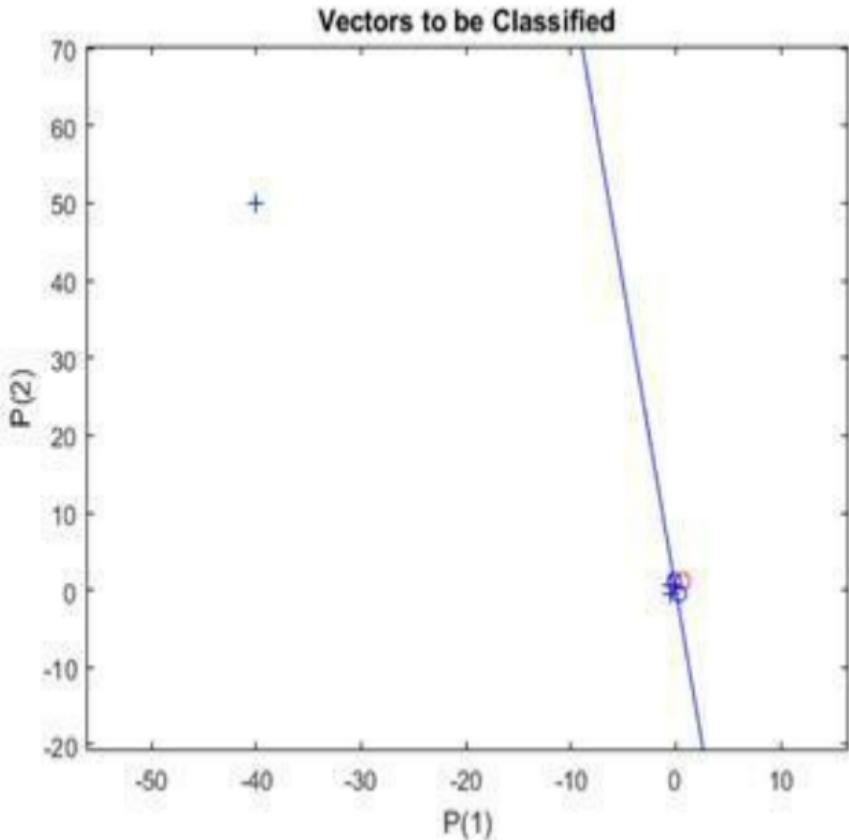
Turn on "hold" so the previous plot is not erased. Add the training set and the classification line to the plot.

```
hold on;
```

```
plotpv(X,T);
```

```
plotpc(net.IW{1},net
```

```
hold off;
```

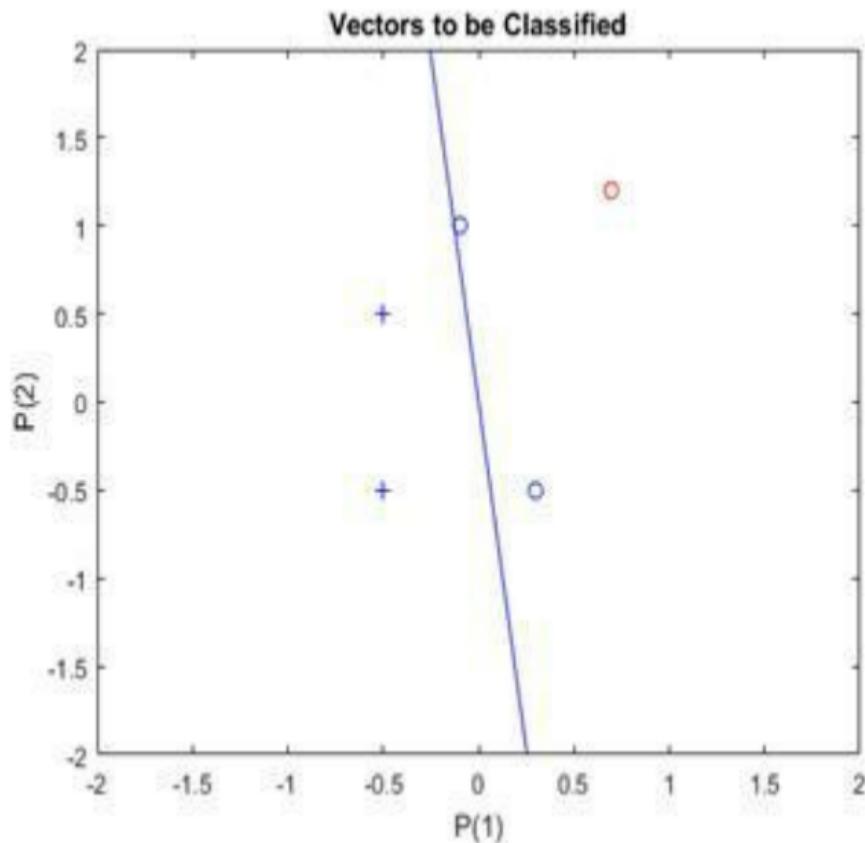


Finally, zoom into the area of interest.

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] );
```



5.8.4 Linearly Non-separable Vectors

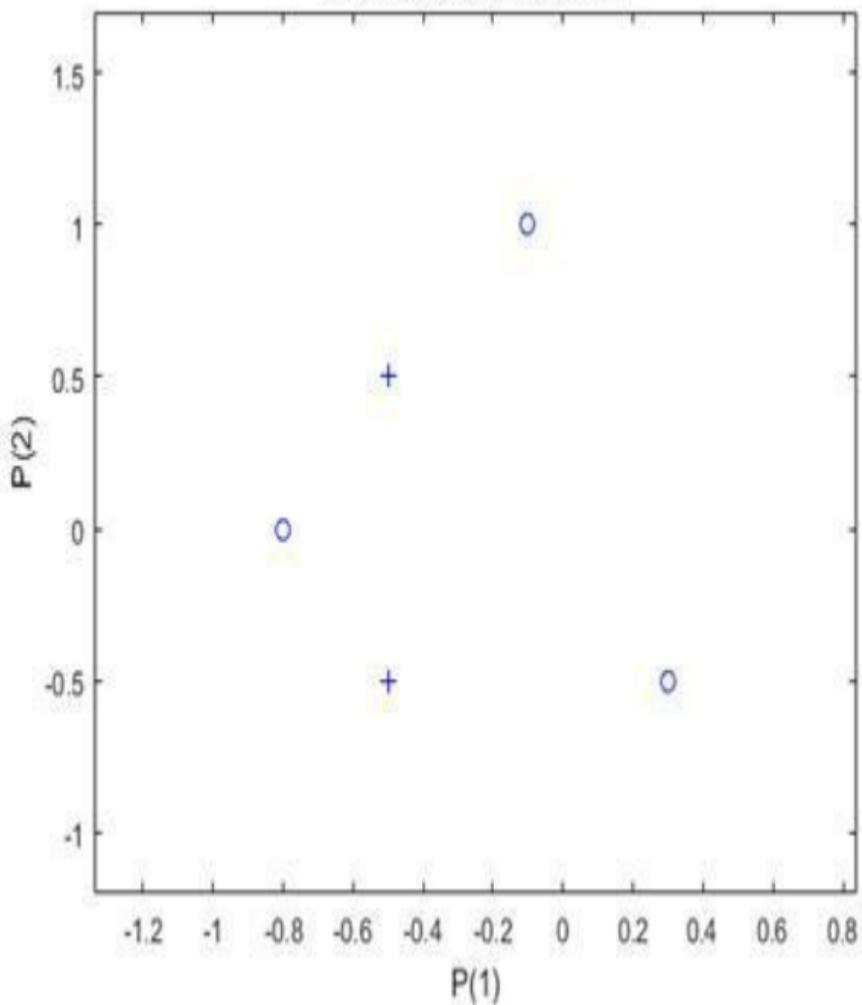
A 2-input hard limit neuron fails to properly classify 5 input vectors because they are linearly non-separable.

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.

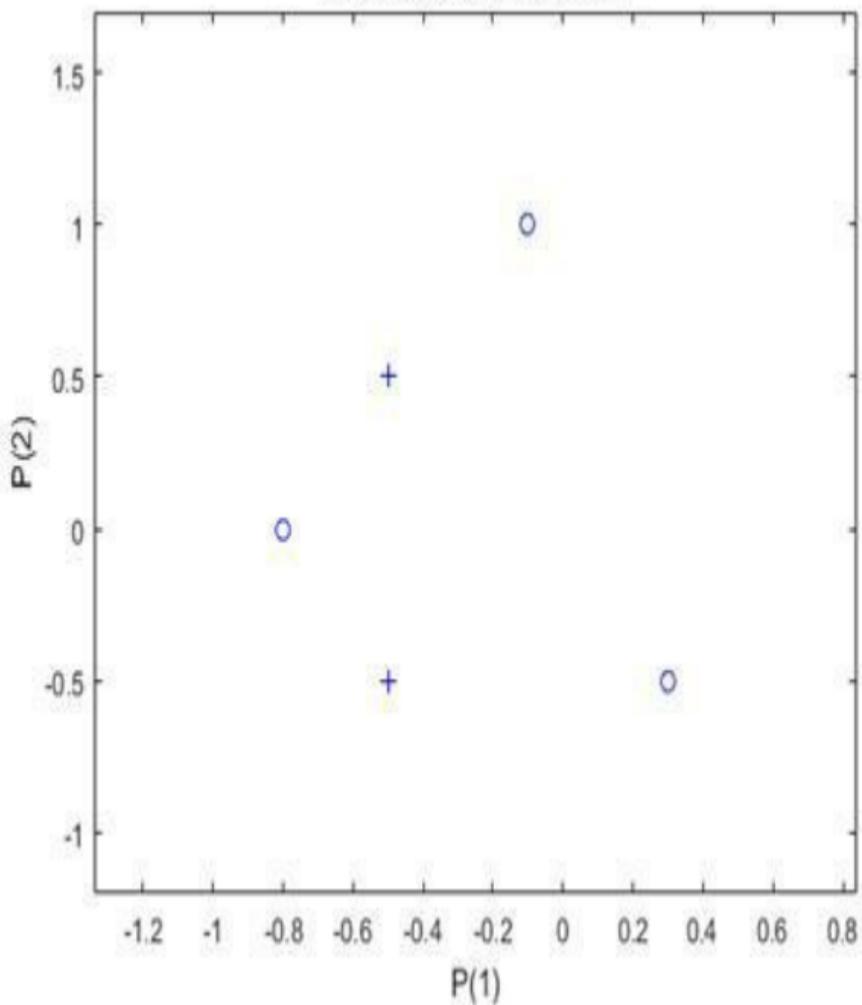
hold on

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

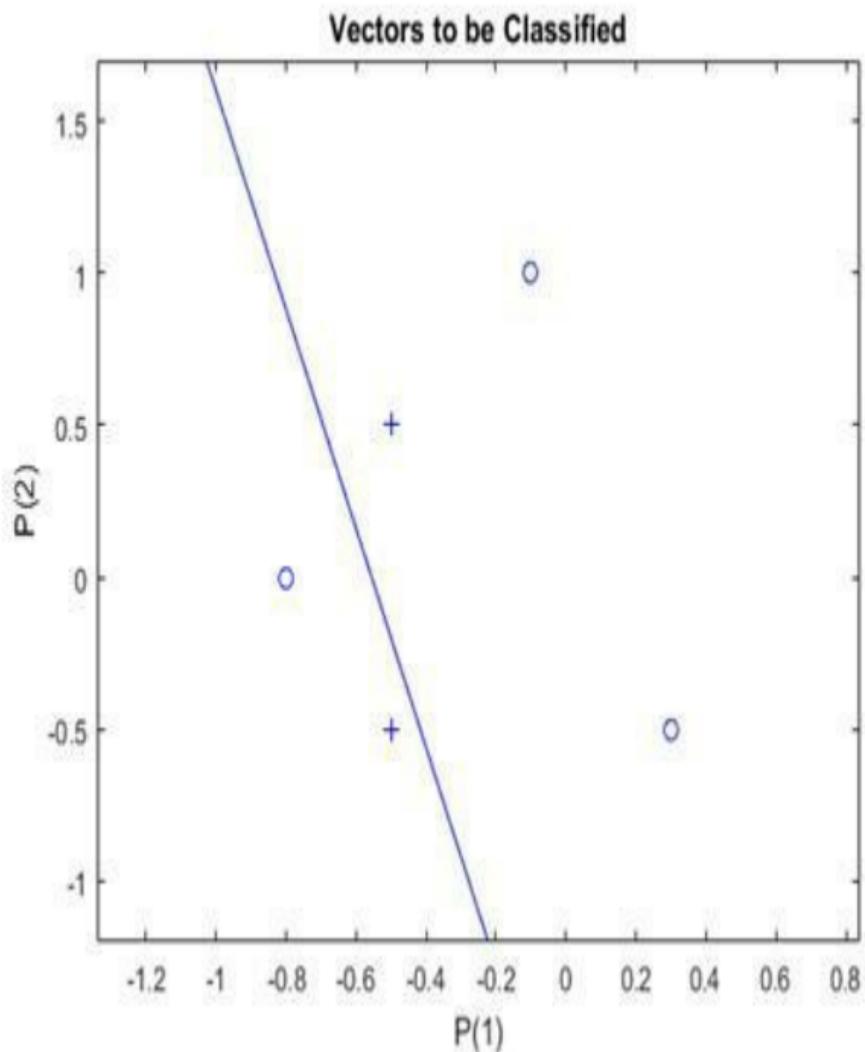
outputs and error. The loop allows the network to repeatedly adapt, plots the classification line, and stops after 25 iterations.

```
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.

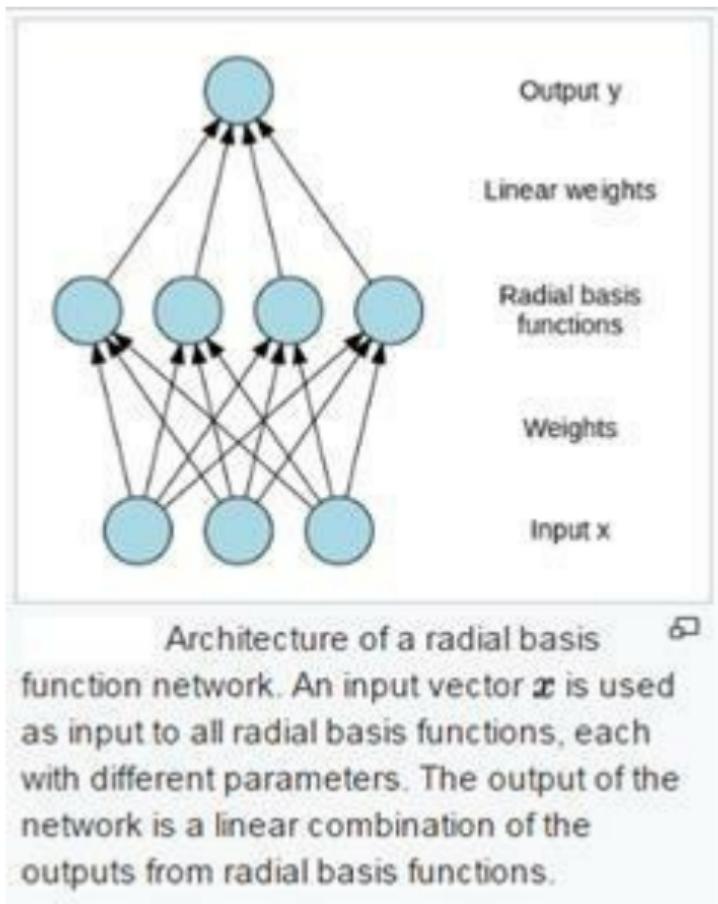
Chapter 6

RADIAL BASIS NEURAL NETWORKS

6.1 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.



RBF networks are typically trained by a two-step algorithm. In the first step, the

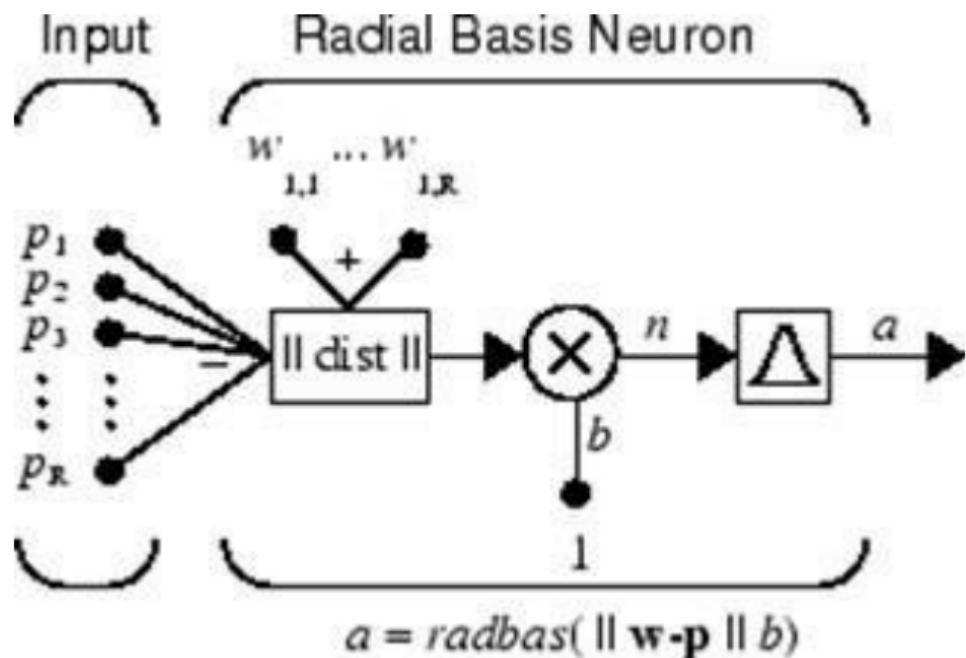
center vectors c_i of the RBF functions in the hidden layer are chosen. This step can be performed in several ways; centers can be randomly sampled from some set of examples, or they can be determined using k-means clustering. Note that this step is unsupervised. A third backpropagation step can be performed 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.

6.2 NEURON MODEL

Here is a radial basis network with R inputs.

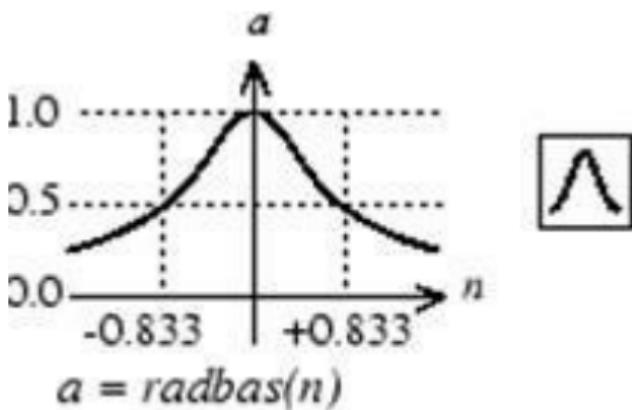


Notice that the expression for the net input of a radbas neuron is different from that of other neurons. Here the net input to the radbas transfer function is the vector distance between its weight vector w and the input vector p , multiplied by the bias b . (The $\parallel \text{dist} \parallel$ box in this figure accepts the input vector p and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is

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

Here is a plot of the [radbas](#) transfer function.



Radial Basis Function

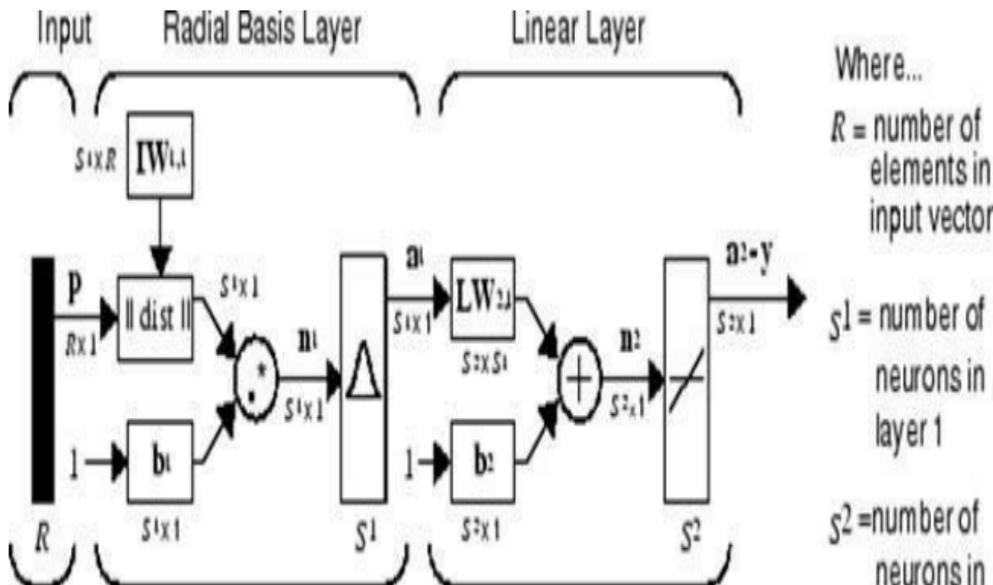
The radial basis function has a maximum of 1 when its input is 0. As the distance between \mathbf{w} and \mathbf{p} decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1

whenever the input \mathbf{p} is identical to its weight vector \mathbf{w} .

The bias b allows the sensitivity of the radbas neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector \mathbf{p} at vector distance of 8.326 ($0.8326/b$) from its weight vector \mathbf{w} .

6.3 NETWORK ARCHITECTURE

Radial basis networks consist of two layers: a hidden radial basis layer of S^1 neurons, and an output linear layer of S^2 neurons.



$$a^1_i = \text{radbas}(\| \mathbf{IW}_{1,1} \cdot p \| b_1)$$

$$a^2 = \text{purelin}(\mathbf{LW}_{2,1} a^1 + b_2)$$

a^1_i is i^{th} element of a^1 where $\mathbf{IW}_{1,1}$ is a vector made of the i^{th} row of $\mathbf{IW}_{1,1}$

Where...
 R = number of elements in input vector
 S^1 = number of neurons in layer 1
 S^2 = number of neurons in layer 2

The $\| \text{dist} \|$ box in this figure accepts the input vector p and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are

the distances between the input vector and vectors $i\mathbf{W}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector \mathbf{b}^1 and the output of $\| \text{dist} \|$ are combined with the MATLAB[®] operation $.*$, which does element-by-element multiplication.

The output of the first layer for a feedforward network net can be obtained with the following code:

```
a{1} =  
radbas(netprod(dist(net.IW{1,1}),p),1)
```

Fortunately, you won't have to write such lines of code. All the details of

designing this network are built into design functions [newrbe](#) and [newrb](#), and you can obtain their outputs with [sim](#).

You can understand how this network behaves by following an input vector \mathbf{p} through the network to the output \mathbf{a}^2 . If you present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector \mathbf{p} have outputs near zero. These small outputs have only a negligible

effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector \mathbf{p} produces a value near 1. If a neuron has an output of 1, its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0s (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with [dist](#). Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with [netprod](#). Each neuron's output is its net input passed through [radbas](#). If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input is spread, its net input is $\text{sqrt}(-\log(.5))$ (or 0.8326), therefore

its output is 0.5.

6.4 EXACT DESIGN (NEWRBE)

You can design radial basis networks with the function [newrbe](#). This function can produce a network with zero error on training vectors. It is called in the following way:

```
net = newrbe(P,T,SPREAD)
```

The function [newrbe](#) takes matrices of input vectors P and target vectors T, and a spread constant SPREAD for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly T when the inputs

are P .

This function [newrbe](#) creates as many [radbas](#) neurons as there are input vectors in P , and sets the first-layer weights to P' . Thus, there is a layer of [radbas](#) neurons in which each neuron acts as a detector for a different input vector. If there are Q input vectors, then there will be Q neurons.

Each bias in the first layer is set to $0.8326/\text{SPREAD}$. This gives radial basis functions that cross 0.5 at weighted inputs of $+/- \text{SPREAD}$. This determines the width of an area in the input space to

which each neuron responds. If SPREAD is 4, then each radbas neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. SPREAD should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights IW^{2,1} (or in code, IW{2,1}) and biases b² (or in code, b{2}) are found by simulating the first-layer outputs a¹ (A{1}), and then solving the following linear expression:

$$[W\{2,1\} \ b\{2\}] * [A\{1\}; \text{ones}(1,Q)] = T$$

You know the inputs to the second layer ($A\{1\}$) and the target (T), and the layer is linear. You can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

$$Wb = T/[A\{1\}; \text{ones}(1,Q)]$$

Here Wb contains both weights and biases, with the biases in the last column. The sum-squared error is always 0, as explained below.

There is a problem with C constraints (input/target pairs) and each neuron has $C + 1$ variables (the C weights from the Crdbas neurons, and a bias). A

linear problem with C constraints and more than C variables has an infinite number of zero error solutions.

Thus, [newrb](#) creates a network with zero error on training vectors. The only condition required is to make sure that SPREAD is large enough that the active input regions of the [radbas](#) neurons overlap enough so that several [radbas](#) neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in

the design. (However, SPREAD should not be so large that each neuron is effectively responding in the same large area of the input space.)

The drawback to [newrbe](#) is that it produces a network with as many hidden neurons as there are input vectors. For this reason, [newrbe](#) does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

6.5 MORE EFFICIENT DESIGN (NEWRB)

The function [newrb](#) iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is

```
net = newrb(P,T,GOAL,SPREAD)
```

The function [newrb](#) takes matrices of input and target vectors P and T, and design parameters GOAL and SPREAD, and

returns the desired network.

The design method of [newrb](#) is similar to that of [newrbe](#). The difference is that [newrb](#) creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most is used to create a [radbas](#) neuron. The error of the new network is checked, and if low enough [newrb](#) is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met or the maximum number of neurons is reached.

As with [newrbe](#), it is important that the spread parameter be large enough that

the radbas neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feedforward network? Radial basis networks, even when designed efficiently with newrbe, tend to have many times more neurons than a comparable feedforward network with tansig or logsig neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the

input space, while radbas neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more radbas neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons' being used, as can be seen in the next example.

6.6 RADIAL BASIS EXAMPLES

6.6.1 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 ...

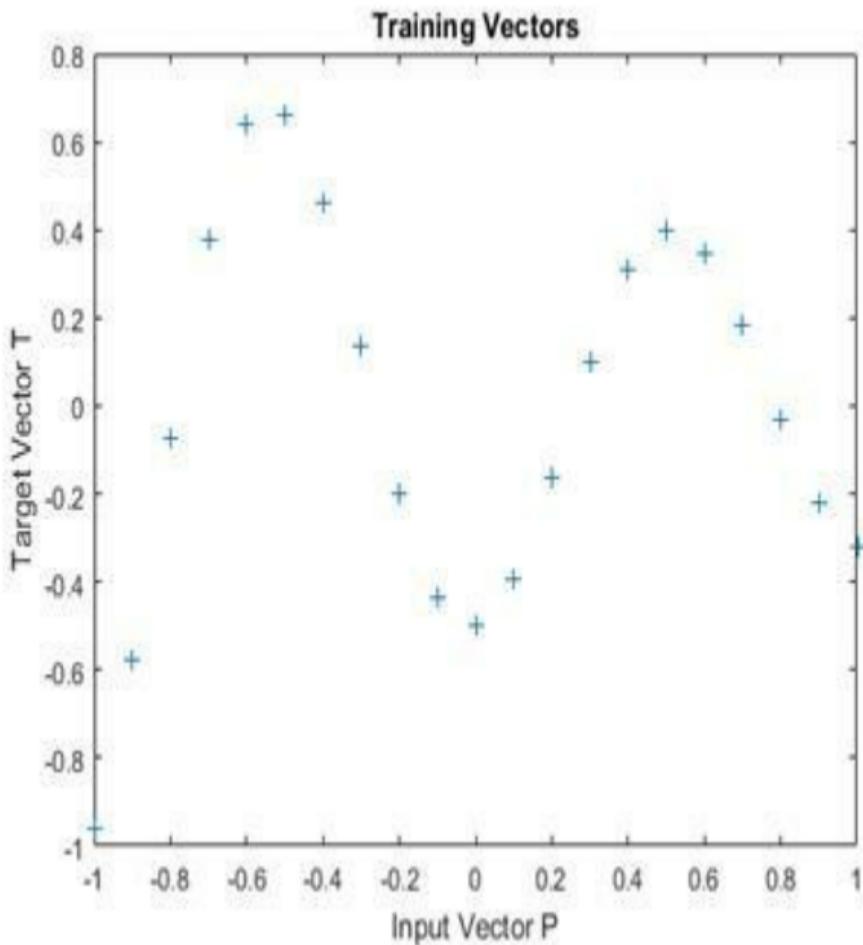
.3072 .3960

.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;
```

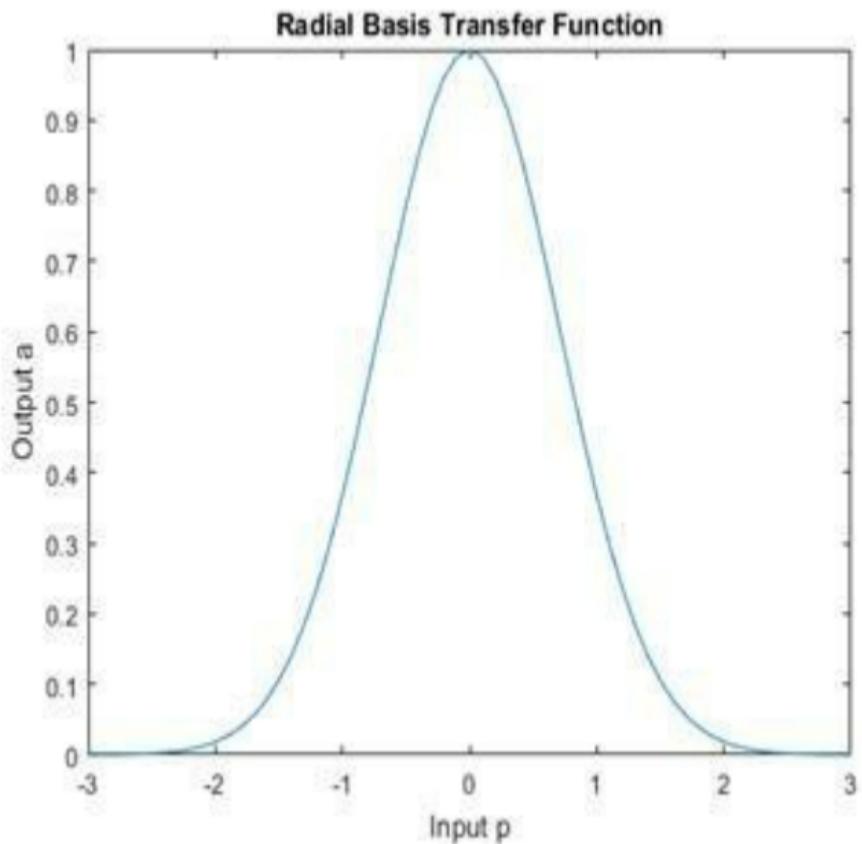
```
a = radbas(x);
```

```
plot(x,a)
```

```
title('Radial Basis Transfer Function');
```

```
xlabel('Input p');
```

```
ylabel('Output a');
```



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);

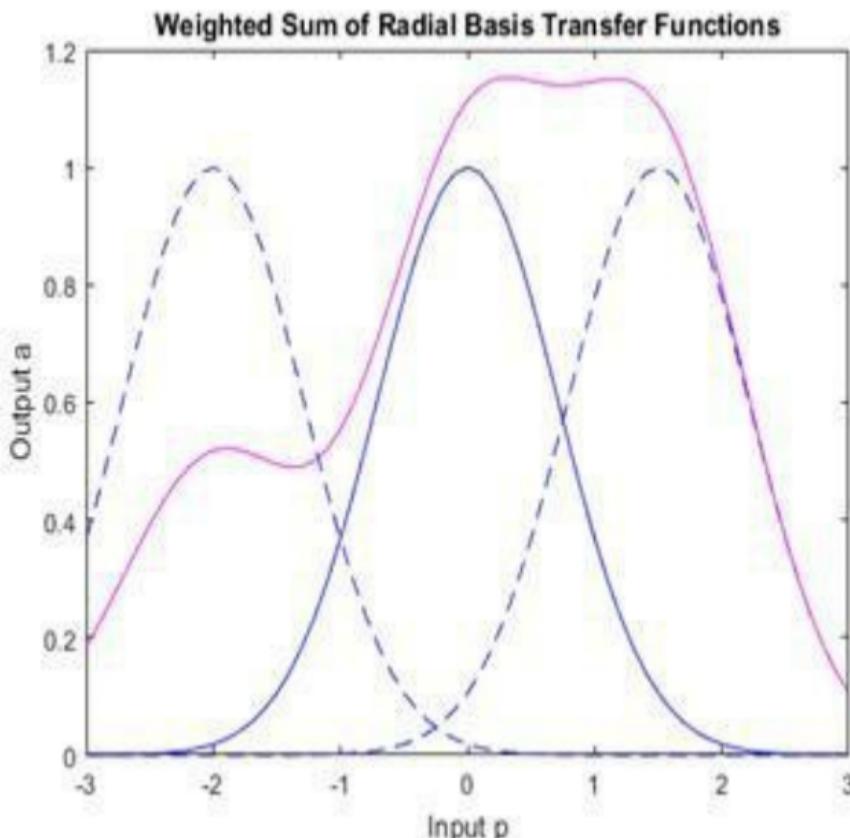
```
a3 = radbas(x+2);
```

```
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;
```

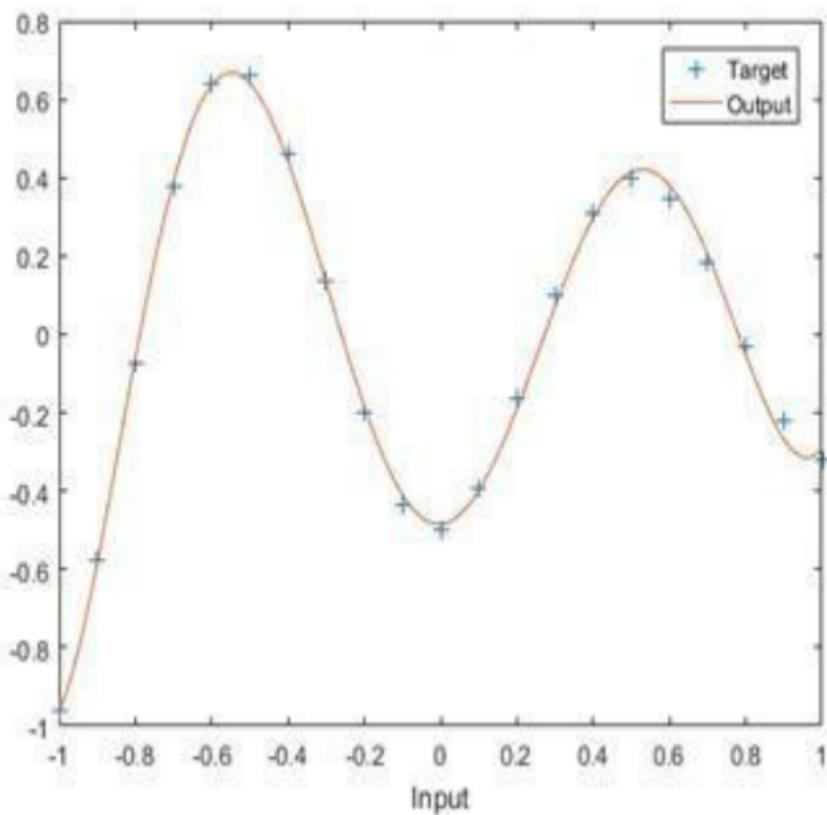
Y = net(X);

hold on;

plot(X, Y);

hold off;

legend({'Target', 'Ou..'});



6.6.2 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 ;

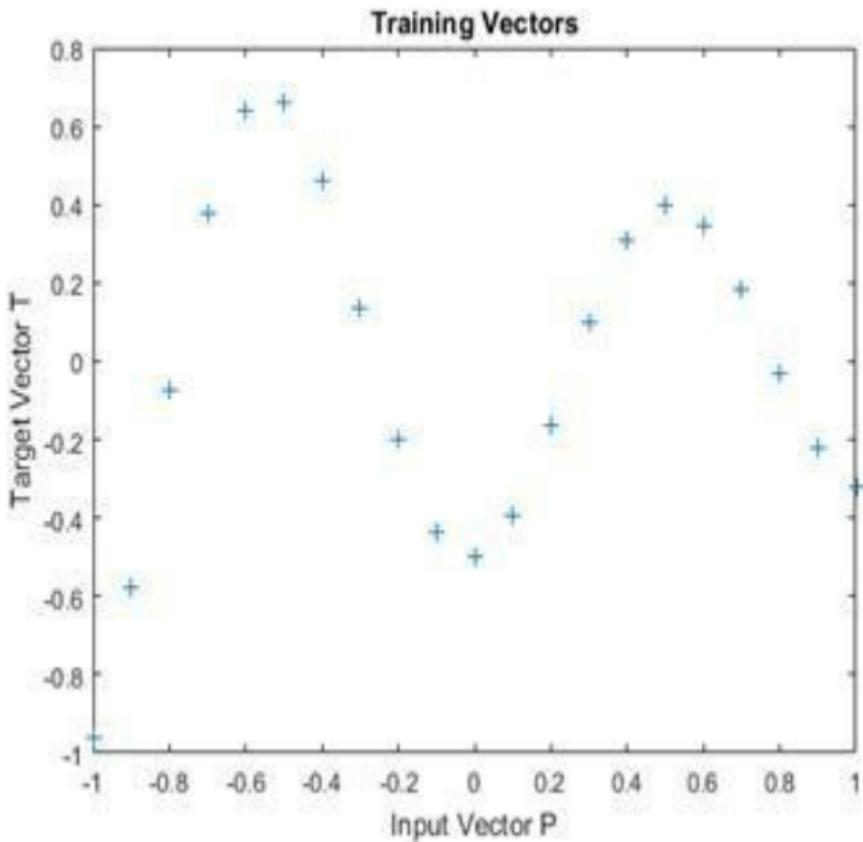
T = [-.9602 -.5770

```
- .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;
```

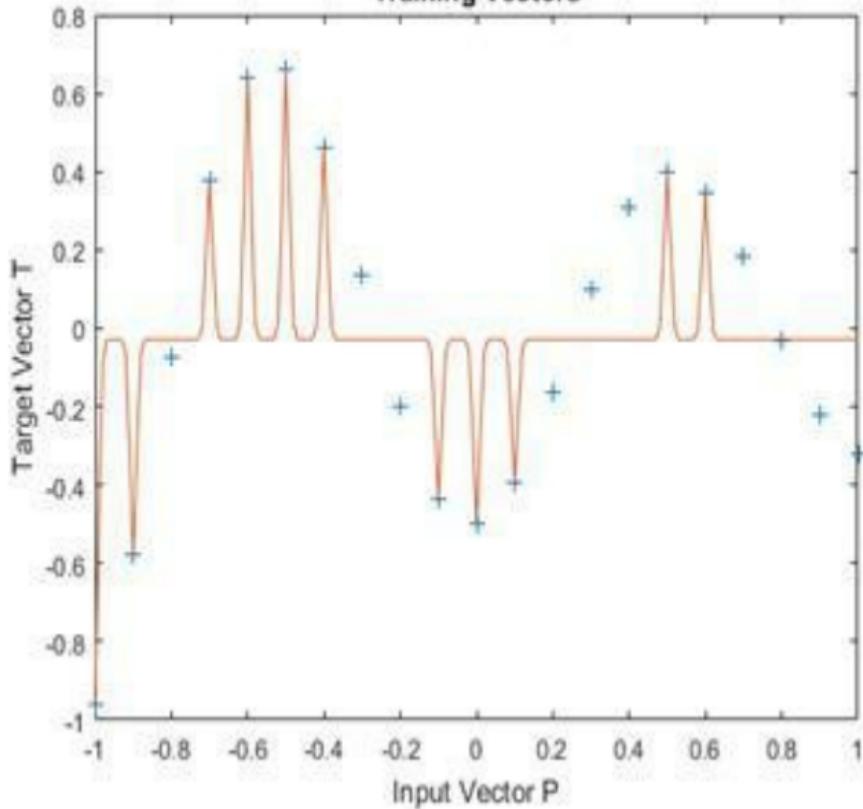
Y = net(X);

hold on;

plot(X, Y);

hold off;

Training Vectors



6.6.3 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')

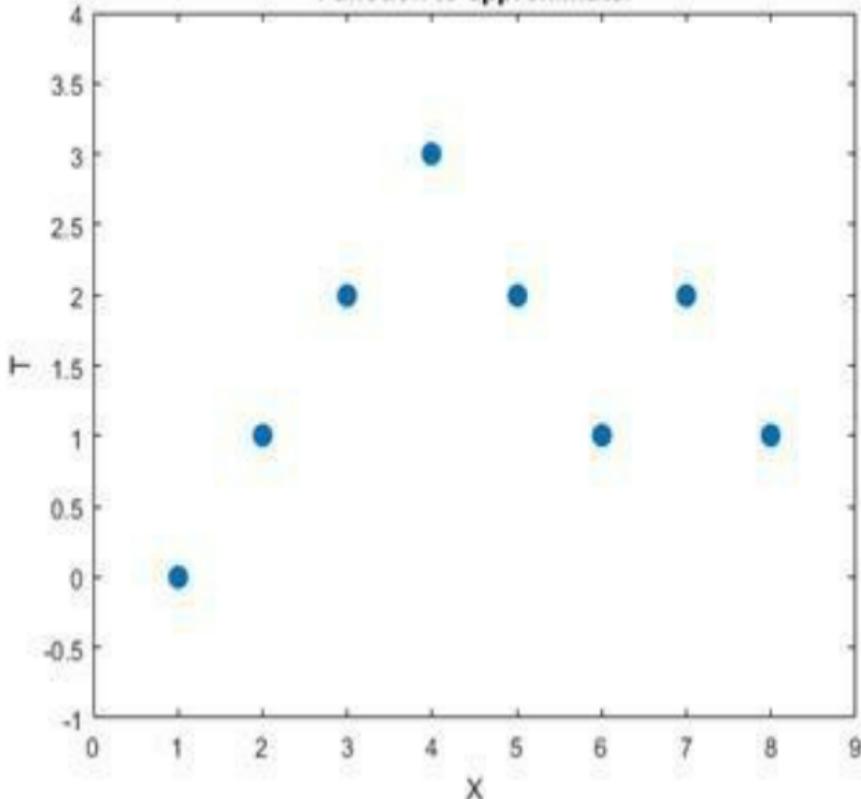
axis([0 9 -1 4])

title('Function to
approximate. ')

xlabel('X')

ylabel('T')
```

Function to approximate.



We use NEWGRNN to create a generalized regression network. We use SPREAD slightly lower than 1, the distance between input values, in order, to get a function that fits individual data

points fairly closely. A smaller spread would fit data better but be less smooth.

```
spread = 0.7;
```

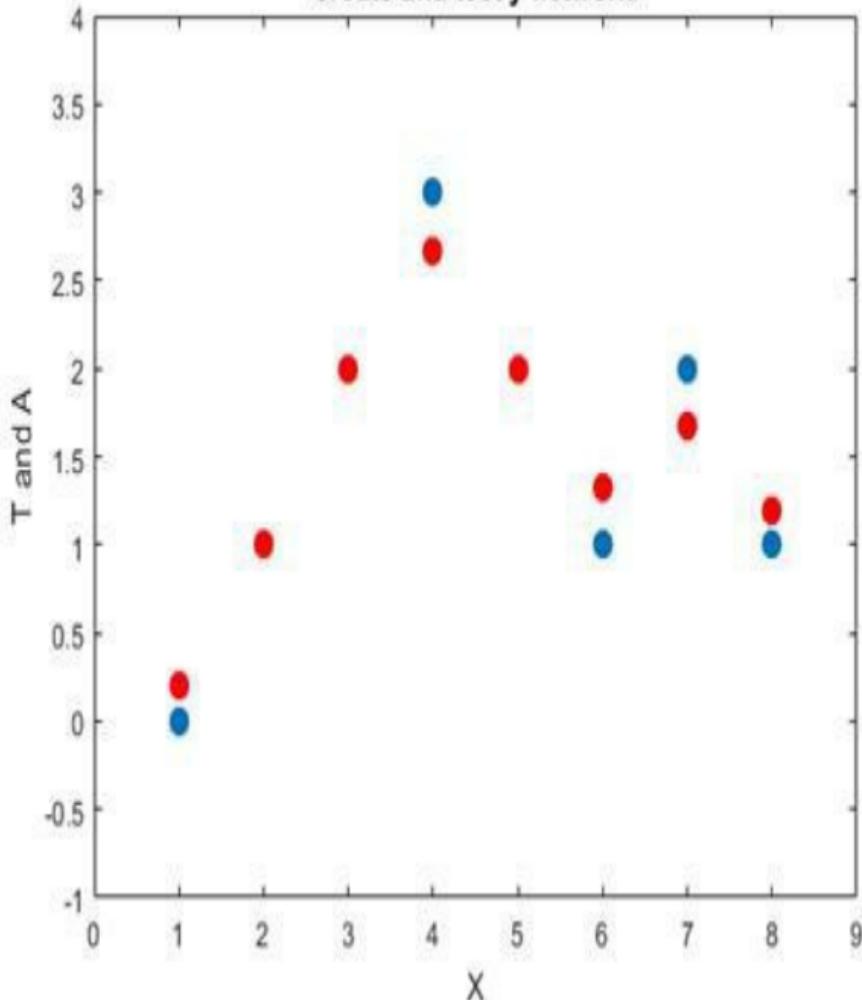
```
net =
newgrnn(X, T, spread);
```

```
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);
```

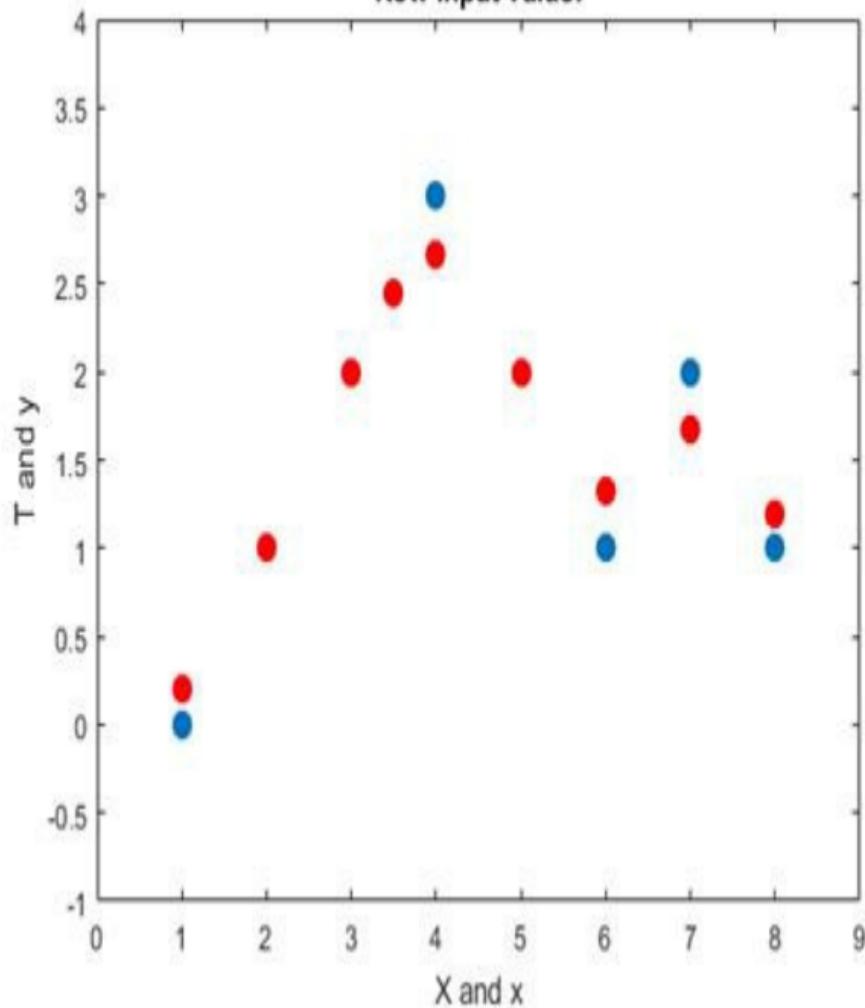
```
plot(x,y,'.', 'marker'  
[1 0 0]);
```

```
title('New input  
value.')
```

```
xlabel('X and x')
```

`ylabel('T and y')`

New input value.



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);
```

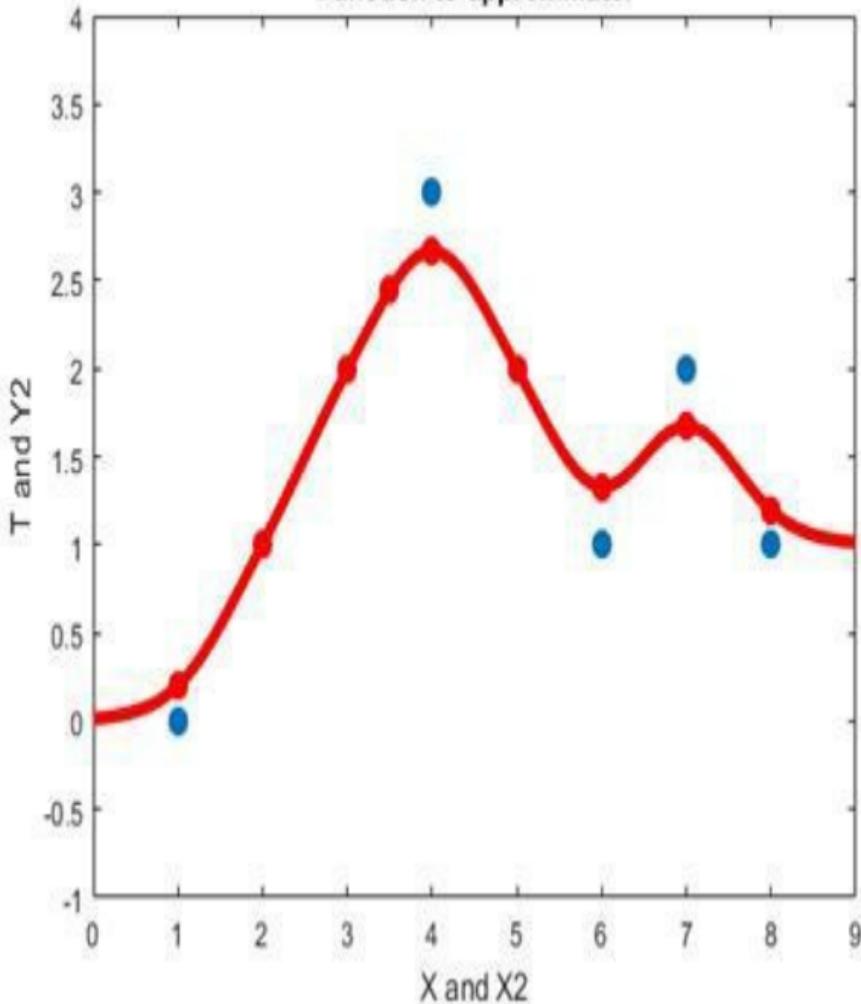
```
plot(X2,Y2,'linewidt:  
[1 0 0])
```

```
title('Function to  
approximate.')
```

```
xlabel('X and X2')
```

```
ylabel('T and Y2')
```

Function to approximate.



6.6.4 PNN Classification

This example uses functions NEWPNN and SIM.

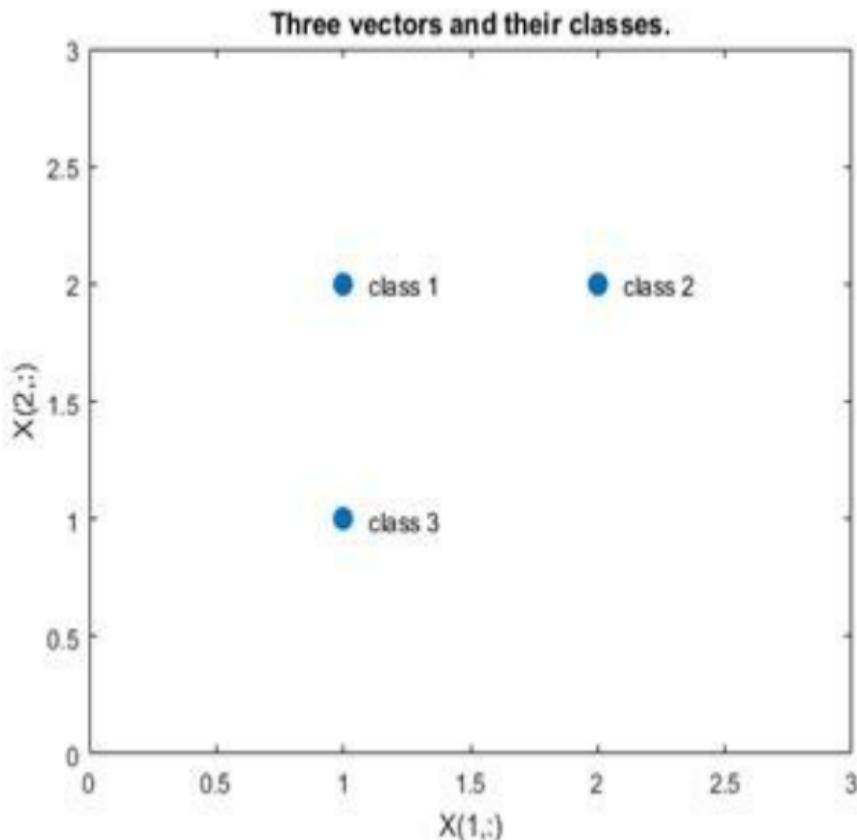
Here are three two-element input vectors X and their associated classes Tc. We would like to create y probabilistic neural network that classifies these vectors properly.

```
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,:')
```



First we convert the target class indices T_c to vectors T . Then we design y probabilistic neural network with

NEWPNN. We use y SPREAD value of 1 because that is y 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])
```

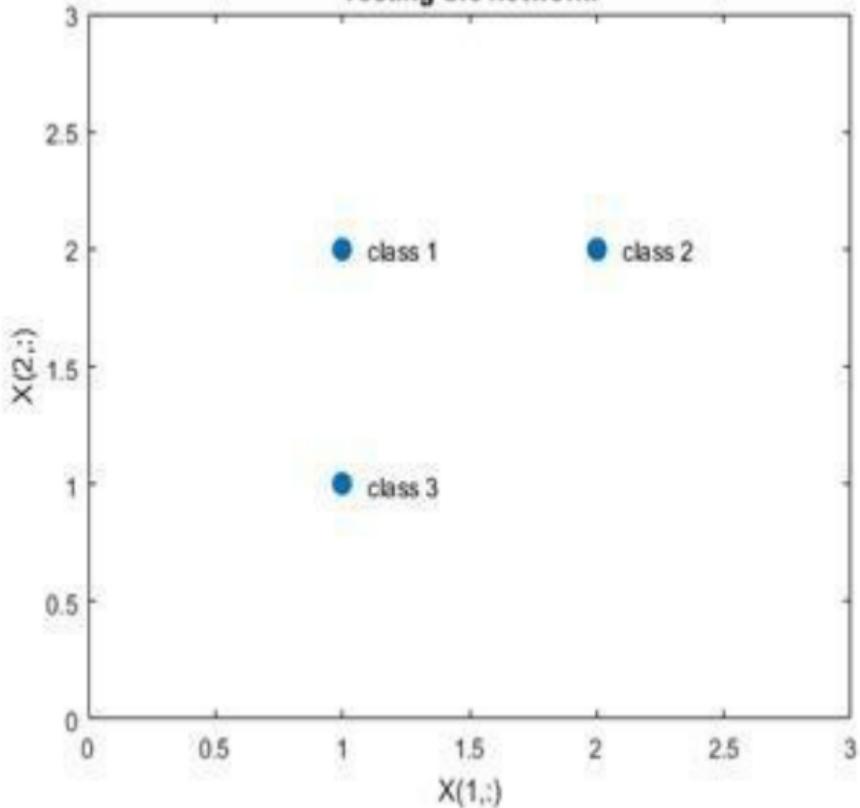
```
for i =
1:3, text(X(1,i)+0.1,
%g', Yc(i))), end
```

```
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),s)%g',ac) )
```

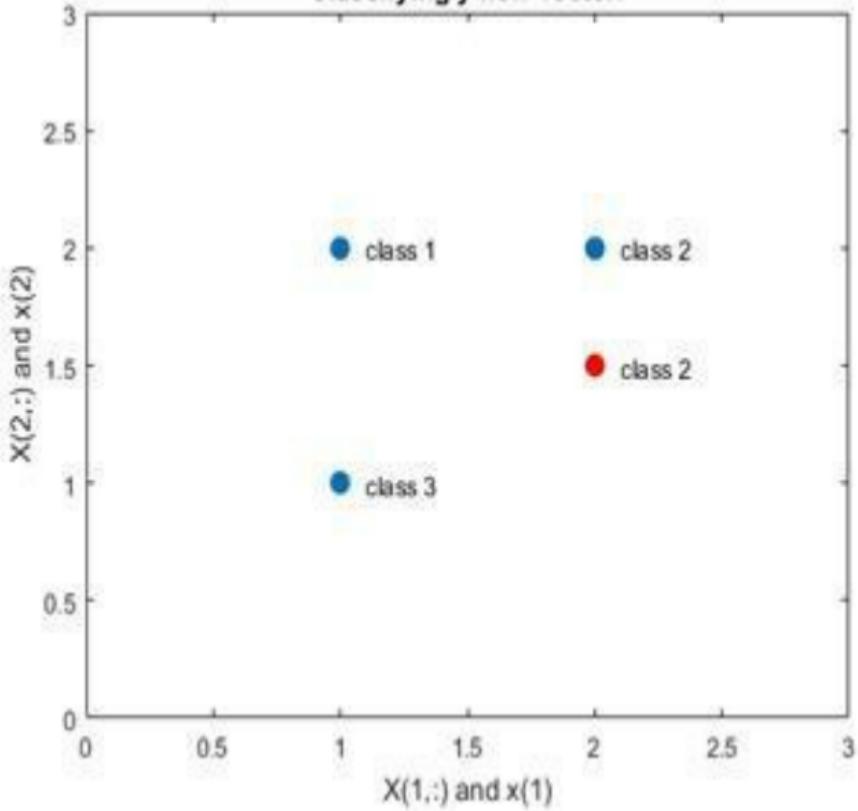
hold off

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];
```

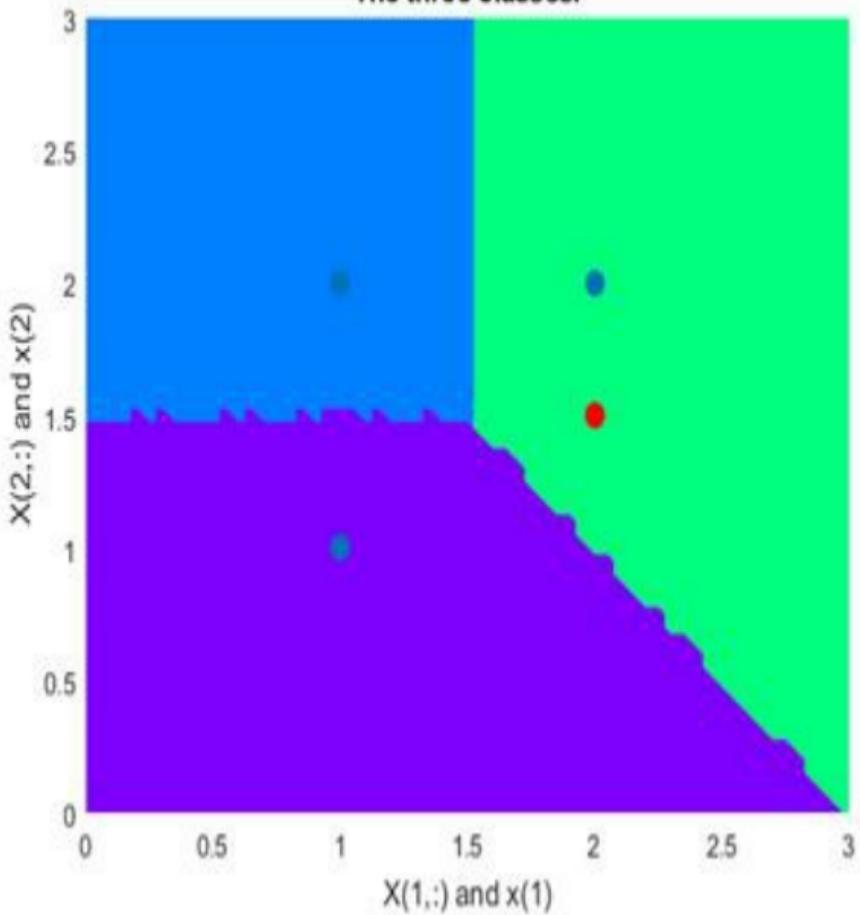
```
m.LineStyle =  
'none';  
  
plot3(X(1,:),X(2,:),  
[1 1  
1]+0.1,'.', 'markersi  
  
plot3(x(1),x(2),1.1,  
[1 0 0])  
  
hold off  
  
view(2)
```

```
title('The three  
classes.')
```

```
xlabel('X(1,:)' and  
x(1)')
```

```
ylabel('X(2,:)' and  
x(2)')
```

The three classes.



Chapter 7

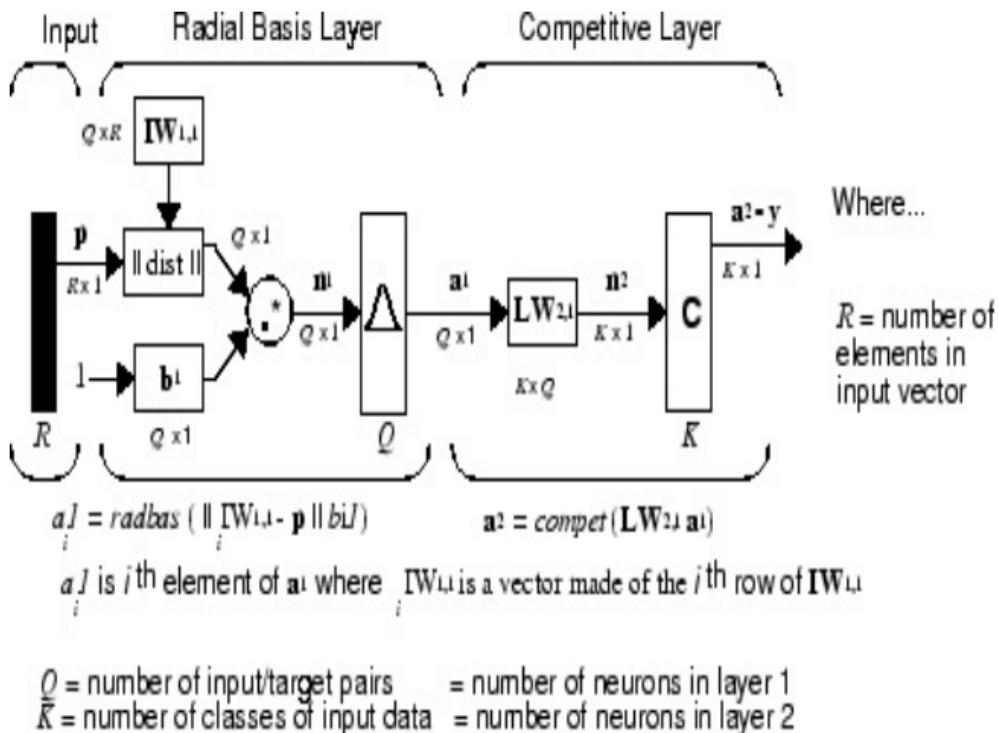
PROBABILISTIC, GENERALIZED REGRESSION AND LVQ NEURAL NETWORKS

7.1 PROBABILISTIC NEURAL NETWORKS

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum

of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

7.1.1 Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these elements is 1 and the rest are 0. Thus, each input vector is associated with one

of K classes.

The first-layer input weights, $\text{IW}^{1,1}$ (`net.IW{1,1}`), are set to the transpose of the matrix formed from the Q training pairs, \mathbf{P}' . When an input is presented, the $\| \cdot \|$ box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent to the radbas transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector \mathbf{a}^1 . If an input is close to several training vectors of a single class, it is represented by several elements

of \mathbf{a}^1 that are close to 1.

The second-layer weights, $\text{LW}^{1,2}$ (`net.LW{2,1}`), are set to the matrix \mathbf{T} of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0s elsewhere. (Use function [ind2vec](#) to create the proper vectors.) The multiplication $\mathbf{T}\mathbf{a}^1$ sums the elements of \mathbf{a}^1 due to each of the K input classes. Finally, the second-layer transfer function, [compet](#), produces a 1 corresponding to the largest element of \mathbf{n}^2 , and 0s elsewhere. Thus, the network classifies the input vector into a specific K class because that class has

the maximum probability of being correct.

7.1.2 Design (newpnn)

You can use the function [newpnn](#) to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

$$P = [0 \ 0; 1 \ 1; 0 \ 3; 1 \\ 4; 3 \ 1; 4 \ 1; 4 \ 3]'$$

which yields

$$P =$$

$$\begin{matrix} 0 & 1 \end{matrix}$$

0 1 3
4 4

0 1
3 4 1
1 3

$T_C = [1 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3]$
which yields

$T_C =$

1 1

2	2	3
3	3	

You need a target matrix with 1s in the right places. You can get it with the function [ind2vec](#). It gives a matrix with 0s except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
```

(1, 1)	1
--------	---

(1, 2) 1

(2, 3) 1

(2, 4) 1

(3, 5) 1

(3, 6) 1

(3, 7) 1

Now you can create a network and simulate it, using the input P to make sure that it does produce the correct

classifications. Use the function [vec2ind](#) to convert the output Y into a row Y_C to make the classifications clear.

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

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

```
Y_C = vec2ind(Y)
```

This produces

```
Y_C =
```

	1	1
2	2	3
3	3	

You might try classifying vectors other than those that were used to design the network. Try to classify the vectors shown below in P2.

P2 = [1 4; 0 1; 5 2] '

P2 =

1	0	5
---	---	---

4 1 2

Can you guess how these vectors will be classified? If you run the simulation and plot the vectors as before, you get

$Y_C =$

2 1 3

These results look good, for these test vectors were quite close to members of classes 2, 1, and 3, respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

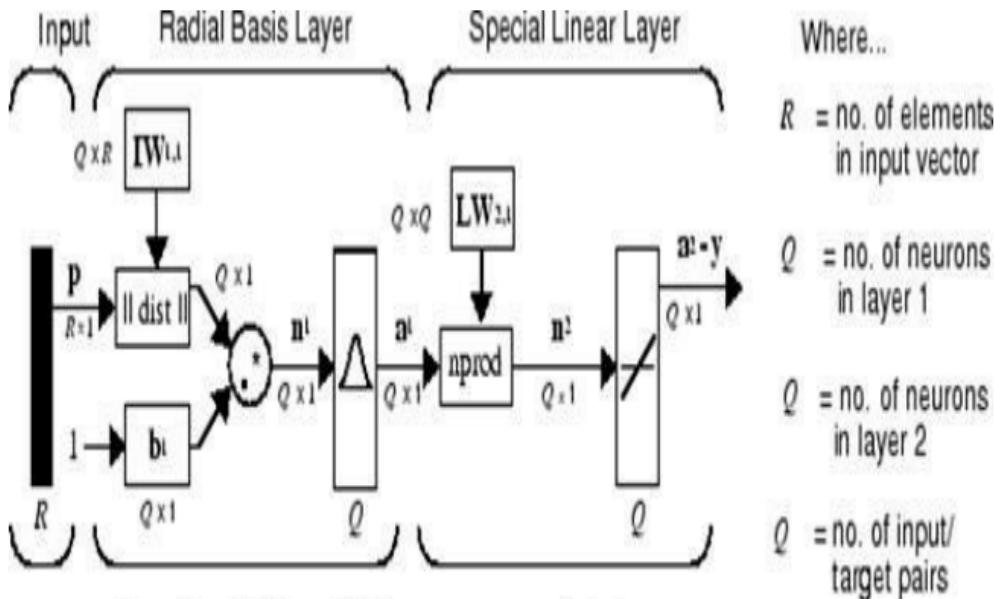
You might want to try demopnn1. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

7.2 GENERALIZED REGRESSION NEURAL NETWORKS

7.2.1 Network Architecture

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



$$a^1 = \text{radbas}(\|IW_{1,1} \cdot p\| b^1)$$

$$a^1 = \text{purelin}(n^2)$$

a^1 is i^{th} element of a^1 where $IW_{1,1}$ is a vector made of the i^{th} row of $IW_{1,1}$

Here the **nprod** box shown above (code function [normprod](#))

produces S^2 elements in vector n^2 . Each element is the dot product of a row of $LW_{2,1}$ and the input vector a^1 , all normalized by the sum of the elements

Where...

R = no. of elements in input vector

Q = no. of neurons in layer 1

Q = no. of neurons in layer 2

Q = no. of input/target pairs

of \mathbf{a}^1 . For instance, suppose that

$$\text{LW}\{2, 1\} = [1 \ -2; 3 \ 4; 5 \\ 6];$$

$$\mathbf{a}\{1\} = [0.7; 0.3];$$

Then

$$\mathbf{a}_{\text{out}} =$$

$$\text{normprod}(\text{LW}\{2, 1\}, \mathbf{a}\{1\})$$

$$\mathbf{a}_{\text{out}} =$$

0.1000

3.3000

5.3000

The first layer is just like that for newrbe networks. It has as many neurons as there are input/ target vectors in \mathbf{P} . Specifically, the first-layer weights are set to \mathbf{P}' . The bias \mathbf{b}^1 is set to a column vector of $0.8326/\text{SPREAD}$. The user chooses SPREAD, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like

the [newrbe](#) radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with [dist](#). Each neuron's net input is the product of its weighted input with its bias, calculated with [netprod](#). Each neuron's output is its net input passed through [radbas](#). If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input will be spread, and its net input will be $\text{sqrt}(-\log(.5))$ (or 0.8326). Therefore its

output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here $LW\{2, 1\}$ is set to T.

Suppose you have an input vector \mathbf{p} close to \mathbf{p}_i , one of the input vectors among the input vector/target pairs used in designing layer 1 weights. This input \mathbf{p} produces a layer 1 \mathbf{a}^i output close to 1. This leads to a layer 2 output close to t_i , one of the targets used to form layer 2 weights.

A larger spread leads to a large area around the input vector where layer 1 neurons will respond with significant outputs. Therefore if spread is small the

radial basis function is very steep, so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network tends to respond with the target vector associated with the nearest design input vector.

As spread becomes larger the radial basis function's slope becomes smoother and several neurons can respond to an input vector. The network then acts as if it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As spread becomes larger more and more neurons contribute to the average, with the result that the network

function becomes smoother.

7.2.2 Design (newgrnn)

You can use the function [newgrnn](#) to create a GRNN. For instance, suppose that three input and three target vectors are defined as

$$P = [4 \ 5 \ 6];$$

$$T = [1.5 \ 3.6 \ 6.7];$$

You can now obtain a GRNN with

`net = newgrnn(P, T);`
and simulate it with

```
P = 4.5;
```

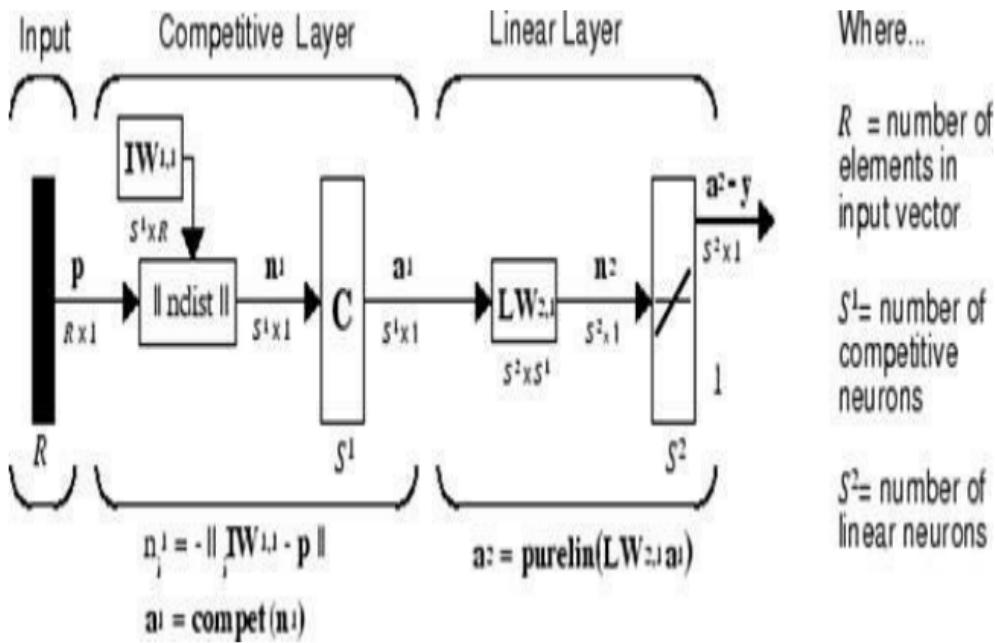
```
v = sim(net, P);
```

You might want to try `demogrnn1`. It shows how to approximate a function with a GRNN.

7.3 LEARNING VECTOR QUANTIZATION (LVQ) NEURAL NETWORKS

7.3.1 Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of Cluster with Self-Organizing Map Neural Network described in this topic. The

linear layer transforms the competitive layer's classes into target classifications defined by the user. The classes learned by the competitive layer are referred to as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to S^1 subclasses. These, in turn, are combined by the linear layer to form S^2 target classes. (S^1 is always larger than S^2 .)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that

belongs to the linear layer target class 2. Then competitive neurons 1, 2, and 3 will have $\mathbf{LW}^{2,1}$ weights of 1.0 to neuron n^2 in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1, 2, or 3) wins the competition and outputs a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a 1 in the i th row of \mathbf{a}^1 (the rest to the elements of \mathbf{a}^1 will be zero) effectively picks the i th column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus,

subclass 1s from layer 1 are put into various classes by the $\mathbf{LW}^{2,1}\mathbf{a}^1$ multiplication in layer 2.

You know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so you can specify the elements of $\mathbf{LW}^{2,1}$ at the start. However, you have to go through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. This training is discussed in [Training](#). First, consider how to create the original network.

7 .3 .2 Creating an LVQ Network

You can create an LVQ network with the function [lvqnet](#),

$$\text{net} = \text{lvqnet}(S1, LR, LF)$$

where

- S1 is the number of first-layer hidden neurons.
- LR is the learning rate (default 0.01).
- LF is the learning function (default is [learnlv1](#)).

Suppose you have 10 input vectors.

Create a network that assigns each of these input vectors to one of four subclasses. Thus, there are four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

$$P = [-3 -2 -2 0 0 0 0 2 2 3; 0 1 -1 2 1 -1 -2 1 -1 0];$$

and

$$Tc = [1 1 1 2 2 2 2 1 1 1];$$

It might help to show the details of what you get from these two lines of code.

$$P, Tc$$

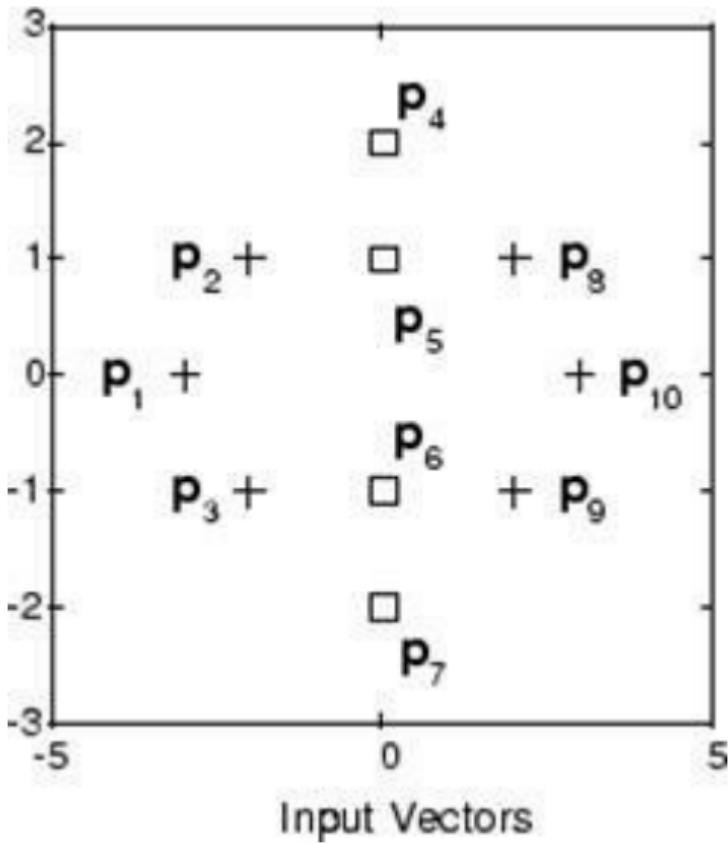
$P =$

$$\begin{matrix} -3 & -2 & -2 & 0 & 0 & 0 & 0 \\ 2 & 2 & 3 & & & & \\ & 0 & 1 & -1 & 2 & 1 & -1 & -2 \\ 1 & -1 & 0 & & & & \end{matrix}$$

$Tc =$

$$\begin{matrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & & & & \end{matrix}$$

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. You want a network that classifies p_1 , p_2 , p_3 , p_8 , p_9 , and p_{10} to produce an output of 1, and that classifies vectors p_4 , p_5 , p_6 , and p_7 to

produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next convert the Tc matrix to target vectors.

```
T = ind2vec(Tc);
```

This gives a sparse matrix T that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
```

1	1	1	0	0	0	0
1	1	1				

0	0	0	1	1	1	1
0	0	0				

This looks right. It says, for instance, that if you have the first column of P as input, you should get the first column of targets as an output; and that output says the input falls in class 1, which is correct. Now you are ready to call lvqnet.

Call lvqnet to create a network with four neurons.

```
net = lvqnet(4);
```

Configure and confirm the initial values of the first-layer weight matrix are initialized by the function midpoint to values in the center of the input data

range.

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

```
net.IW{1}
```

```
ans =
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```

Confirm that the second-layer weights have 60% (6 of the 10 in T_c) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns have a 1 in the second row (corresponding to class 2). With only

four columns, the 60% and 40% actually round to 50% and there are two 1's in each row.

```
net.LW{2,1}
```

```
ans =
```

```
1 1 0 0  
0 0 1 1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element, the input vector is classified as class 1; otherwise it is a class 2.

You might notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1),

while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

You can simulate the network with [sim](#). Use the original P matrix as input just to see what you get.

$Y = \text{net}(P);$

$Y_c = \text{vec2ind}(Y)$

$Y_c =$

1 1 1 1 1 1 1

1 1 1

The network classifies all inputs into class 1. Because this is not what you want, you have to train the network (adjusting the weights of layer 1 only), before you can expect a good result. The next two sections discuss two LVQ learning rules and the training process.

7.3.3 LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

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

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$$

Here there are input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector \mathbf{p} is presented, and the distance from \mathbf{p} to each row of the input weight matrix $\mathbf{IW}^{1,1}$ is computed with the function [negdist](#). The hidden neurons of layer 1 compete. Suppose that the i th element of \mathbf{n}^1 is most positive, and neuron i^* wins the competition. Then the competitive transfer function produces a 1 as the i^* th element of \mathbf{a}^1 . All other elements of \mathbf{a}^1 are 0.

When \mathbf{a}^1 is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in \mathbf{a}^1 selects the class k^* associated with the input. Thus, the network has assigned the input vector \mathbf{p} to class k^* and $\alpha_{k^*}^2$ will be 1. Of course, this assignment can be a good one or a bad one, for t_{k^*} can be 1 or 0, depending on whether the input belonged to class k^* or not.

Adjust the i^* th row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector \mathbf{p} if the assignment is correct, and to move the row away from \mathbf{p} if the assignment is incorrect. If \mathbf{p} is classified correctly,

$$(\alpha_{k^*}^2 = t_{k^*} = 1)$$

compute the new value of the i^* th row of $\mathbf{IW}^{1,1}$ as

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

On the other hand, if \mathbf{p} is classified incorrectly,

$$(\alpha_{k^*}^2 = 1 \neq t_{k^*} = 0)$$

compute the new value of the i^* th row of $\mathbf{IW}^{1,1}$ as

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

You can make these corrections to the i^* th row of $\mathbf{IW}^{1,1}$ automatically, without affecting other rows of $\mathbf{IW}^{1,1}$, by back-propagating the output errors to layer 1.

Such corrections move the hidden neuron toward vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is [learnlv1](#). It can be applied during training.

7.3.4 Training

Next you need to train the network to obtain first-layer weights that lead to the correct classification of input vectors.

You do this with [train](#) as with the following commands. First, set the training epochs to 150. Then, use [train](#):

```
net.trainParam.epochs = 150;
```

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

Now confirm the first-layer weights.

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

```
ans =
```

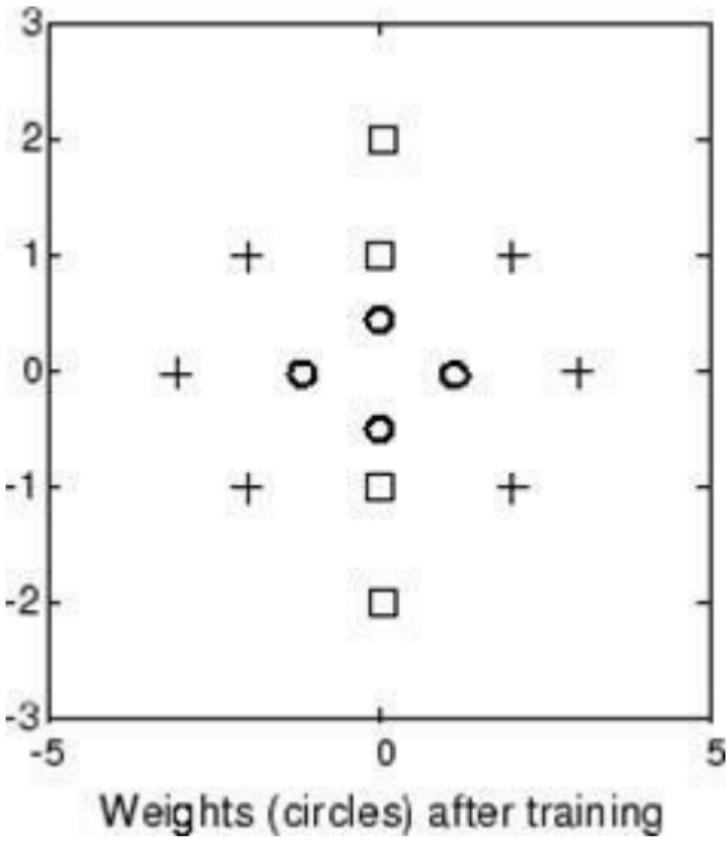
```
0.3283 0.0051
```

-0.1366 0.0001

-0.0263 0.2234

0 -0.0685

The following plot shows that these weights have moved toward their respective classification groups.



Weights (circles) after training

To confirm that these weights do indeed lead to the correct classification, take the matrix P as input and simulate the network. Then see what classifications are produced by the network.

$\mathbf{Y} = \text{net}(\mathbf{P});$

$\mathbf{Yc} = \text{vec2ind}(\mathbf{Y})$

This gives

$\mathbf{Yc} =$

1 1 1 2 2 2
1 1 1

which is expected. As a last check, try an input close to a vector that was used in training.

$\mathbf{pchk1} = [0; 0.5];$

$\mathbf{Y} = \text{net}(\mathbf{pchk1});$

$\mathbf{Yc1} = \text{vec2ind}(\mathbf{Y})$

This gives

$Yc1 =$
2

This looks right, because pchk1 is close to other vectors classified as 2.
Similarly,

```
pchk2 = [1; 0];  
Y = net(pchk2);  
Yc2 = vec2ind(Y)
```

gives

$Yc2 =$
1

This looks right too, because pchk2 is close to other vectors classified as 1.

You might want to try the example program demolvq1. It follows the discussion of training given above.

7.3.5 Supplemental LVQ2.1 Learning Rule ([learnlv2](#))

The following learning rule is one that might be applied *after* first applying LVQ1. It can improve the result of the first learning. This particular version of LVQ2 is embodied in the function [learnlv2](#). Note again that LVQ2.1 is to be used only after LVQ1 has been applied.

Learning here is similar to that in [learnlv2](#) except now two vectors of layer 1 that are closest to the input vector can be updated, provided that one belongs to the correct class and one

belongs to a wrong class, and further provided that the input falls into a "window" near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s$$

where

$$s \equiv \frac{1-w}{1+w}$$

(where d_i and d_j are the Euclidean distances of \mathbf{p} from $i^*\mathbf{IW}^{1,1}$ and $j^*\mathbf{IW}^{1,1}$, respectively). Take a value for w in the

range 0.2 to 0.3. If you pick, for instance, 0.25, then $s = 0.6$. This means that if the minimum of the two distance ratios is greater than 0.6, the two vectors are adjusted. That is, if the input is near the midplane, adjust the two vectors, provided also that the input vector \mathbf{p} and $_{j^*}\mathbf{IW}^{1,1}$ belong to the same class, and \mathbf{p} and $_{i^*}\mathbf{IW}^{1,1}$ do not belong in the same class.

The adjustments made are

$$\begin{aligned}_{i^*}\mathbf{IW}^{1,1}(q) &= {}_{i^*}\mathbf{IW}^{1,1}(q \\ &\quad - 1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q - 1))\end{aligned}$$

and

$${}_{j^*}\mathbf{IW}^{1,1}(q) = {}_{j^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{j^*}\mathbf{IW}^{1,1}(q-1))$$

Thus, given two vectors closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors are adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

7.4 LEARNING VECTOR QUANTIZATION 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.

$$x = [-3 -2 -2 \ 0 \ 0 \ 0 \ 0 +2 +2 +3; \\ 0 +1 -1 +2 +1 -1 -2 +1 -1 \ 0];$$

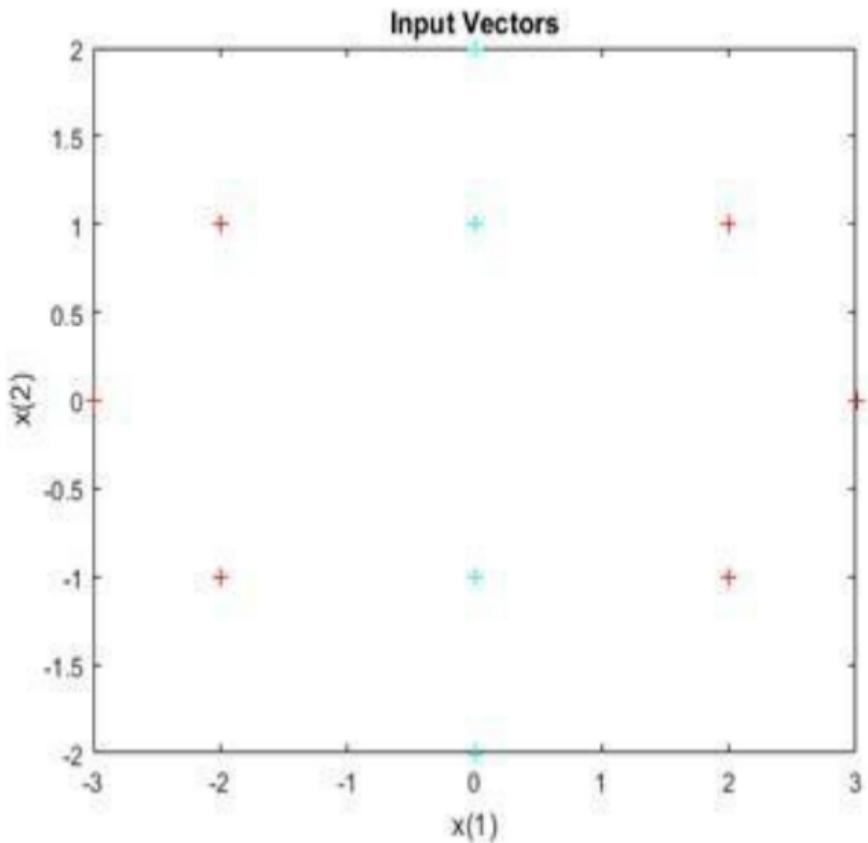
$$c = [1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2 \ 1 \ 1 \ 1];$$

$$t = \text{ind2vec}(c);$$

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
```

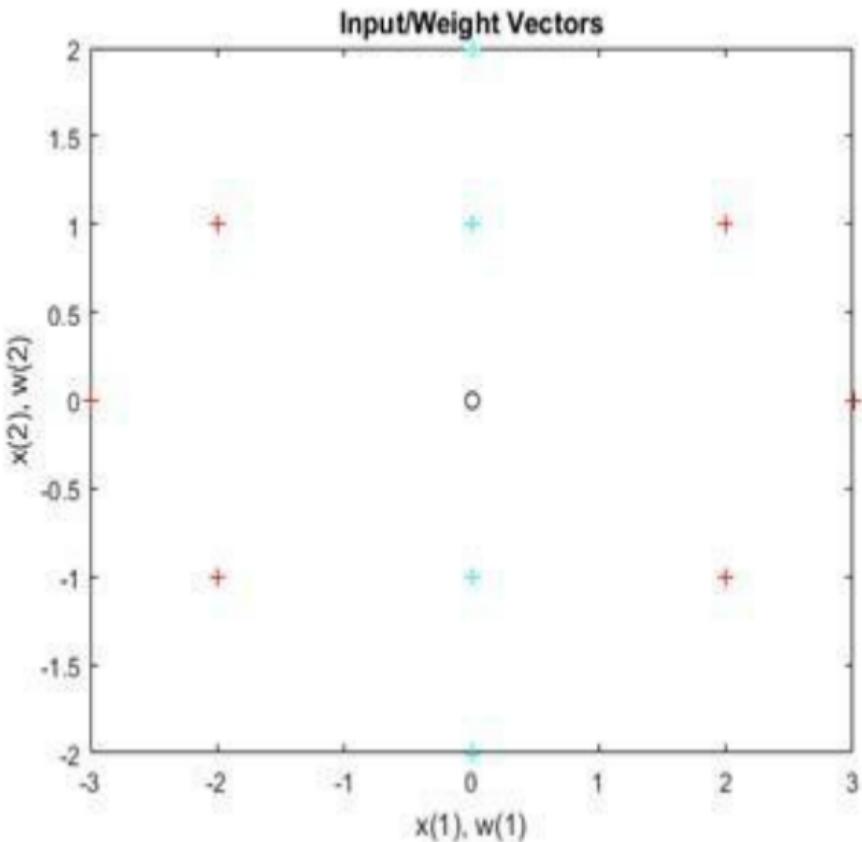
```
w1 = net.IW{1};
```

```
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.

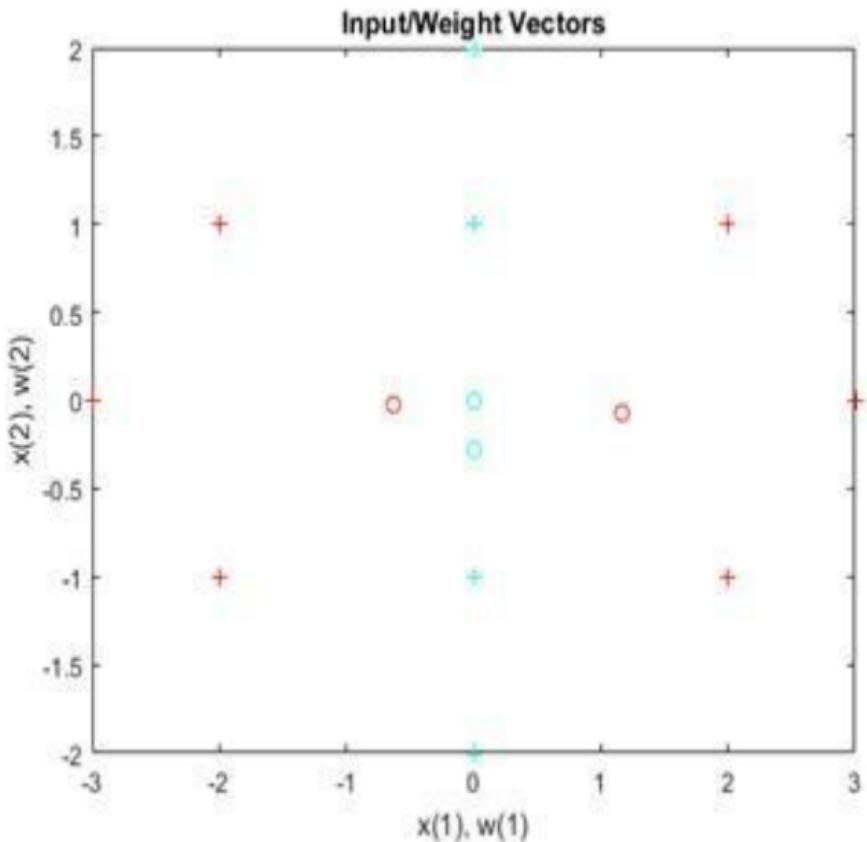
```
net.trainParam.epochs=150;  
net=train(net,x,t);
```

```
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 8

HOPFIELD AND LINEAR NEURAL NETWORKS

8.1 LINEAR NEURAL NETWORKS

The linear networks discussed in this section are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here you design a linear network that, when presented with a set of given input vectors, produces outputs of

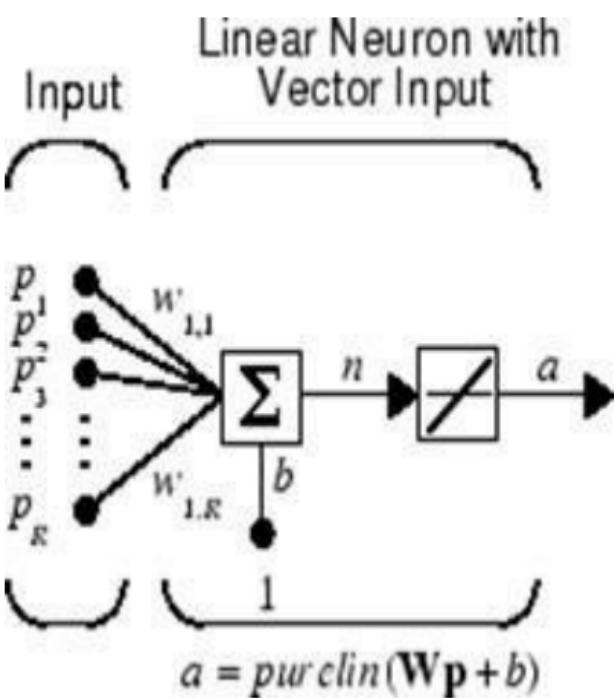
corresponding target vectors. For each input vector, you can calculate the network's output vector. The difference between an output vector and its target vector is the error. You would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, you can calculate a linear network directly, such that its error is a minimum for the given input vectors and target vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, you can always train the

network to have a minimum error by using the least mean squares (Widrow-Hoff) algorithm.

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

8.1.1 Neuron Model

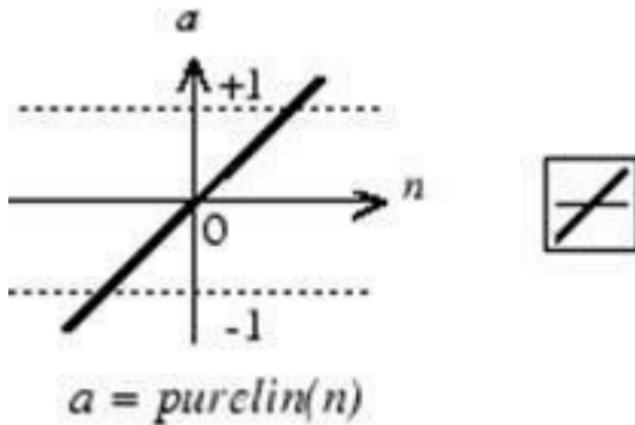
A linear neuron with R inputs is shown below.



Where...

R = number of elements in input vector

This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function [purelin](#).



Linear Transfer Function

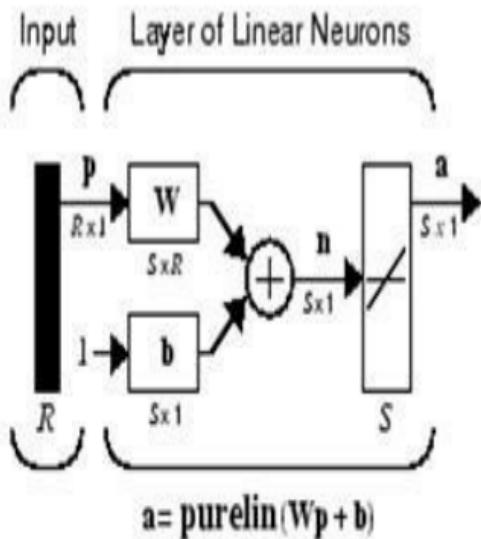
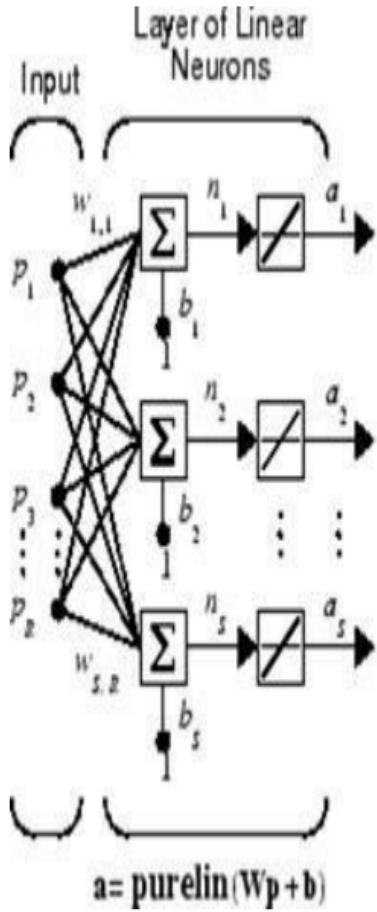
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

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

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

8.1.2 Network Architecture

The linear network shown below has one layer of S neurons connected to R inputs through a matrix of weights \mathbf{W} .



Where...

R = number of elements in input vector

S = number of neurons in layer

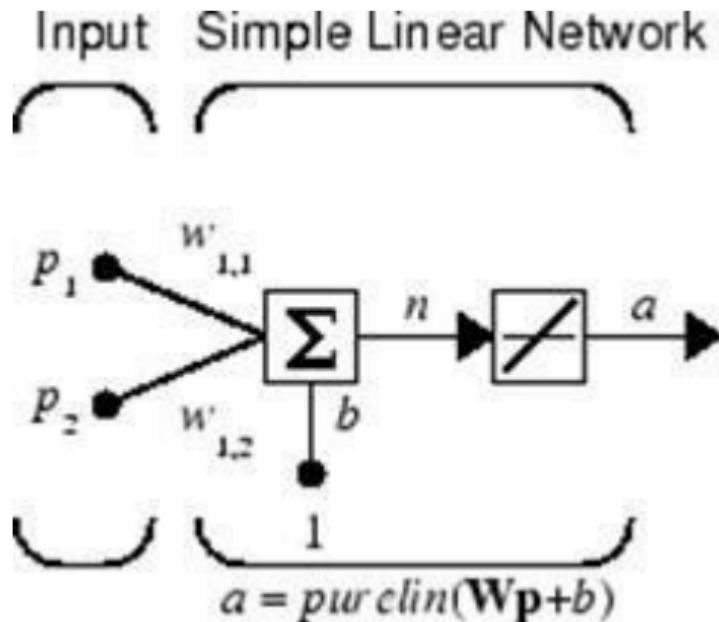
Note that the figure on the right defines an S -length output vector \mathbf{a} .

A single-layer linear network is shown.

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

8.1.3 Create a Linear Neuron (linearlayer)

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



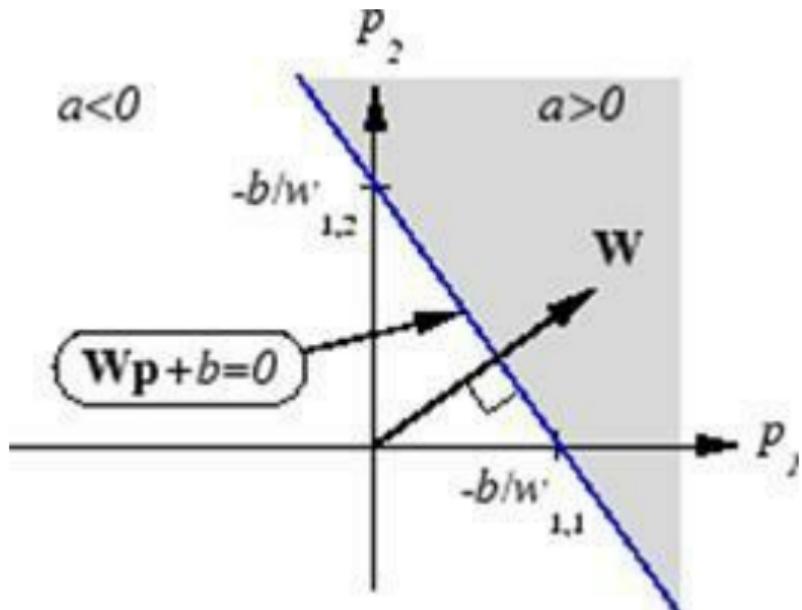
The weight matrix \mathbf{W} in this case has only one row. The network output is

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

or

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

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{W}\mathbf{p} + b = 0$ specifies such a decision boundary, as shown below.



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear

network has the same limitation as the perceptron.

You can create this network using [linearlayer](#), and configure its dimensions with two values so the input has two elements and the output has one.

```
net = linearlayer;
```

```
net = configure(net,[0;0],0);
```

The network weights and biases are set to zero by default. You can see the current values with the commands

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

```
W =
```

```
0 0
```

and

$b = \text{net.b}\{1\}$

$b =$

0

However, you can give the weights any values that you want, such as 2 and 3, respectively, with

$\text{net.IW}\{1,1\} = [2 \ 3];$

$W = \text{net.IW}\{1,1\}$

$W =$

2 3

You can set and check the bias in the same way.

$\text{net.b}\{1\} = [-4];$

`b = net.b{1}`

`b =`

-4

You can simulate the linear network for a particular input vector. Try

`p = [5;6];`

You can find the network output with the function [sim](#).

`a = net(p)`

`a =`

24

To summarize, you can create a linear network with `newlin`, adjust its elements as you want, and simulate it with [sim](#). You

can find more about newlin by typing help newlin.

8 .1 .4 Least Mean Square Error

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

$$\{p_1, t_1\}, \{p_2, t_2\}, \dots \{p_Q, t_Q\}$$

Here p_q is an input to the network, and t_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and

the network output. The goal is to minimize the average of the sum of these errors.

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

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

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the

input vectors determine whether or not a unique solution exists.

8.1.5 Linear System Design (newlind)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. You can obtain specific network values for weights and biases to minimize the mean square error by using the function [newlind](#).

Suppose that the inputs and targets are

$$P = [1 \ 2 \ 3];$$

$$T = [2.0 \ 4.1 \ 5.9];$$

Now you can design a network.

$$\text{net} = \text{newlind}(P, T);$$

You can simulate the network behavior to check that the design was done properly.

$$Y = \text{net}(P)$$

$$Y =$$

$$2.0500 \quad 4.0000 \quad 5.9500$$

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

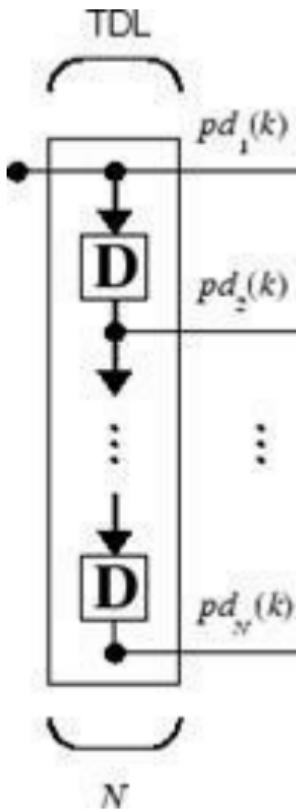
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

You can also use the function [newlind](#) to design linear networks having delays in the input.

8.1.6 Linear Networks with Delays

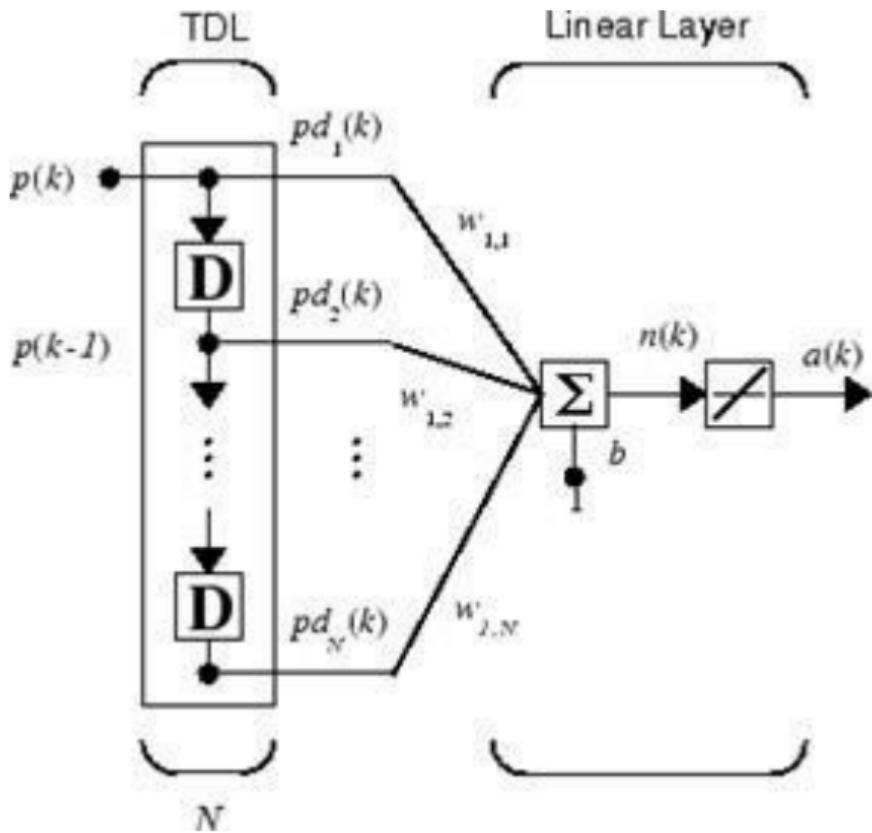
Tapped Delay Line

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



Linear Filter

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



The output of the filter is given by

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

The network shown is referred to in the

digital signal processing field as a finite impulse response (FIR) filter

Suppose that you want a linear layer that outputs the sequence T, given the sequence P and two initial input delay states Pi.

$$P = \{1\ 2\ 1\ 3\ 3\ 2\};$$

$$Pi = \{1\ 3\};$$

$$T = \{5\ 6\ 4\ 20\ 7\ 8\};$$

You can use [newlind](#) to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument, as shown below.

```
net = newlind(P,T,Pi);
```

You can obtain the output of the designed network with

$$Y = \text{net}(P, P_i)$$

to give

$$\begin{aligned} Y = & [2.7297] [10.5405] [5.0090] \\ & [14.9550] [10.7838] [5.9820] \end{aligned}$$

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

8 .1 .7 LMS Algorithm ([learnwh](#))

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

Fortunately, there is a toolbox function, [learnwh](#), that does all the calculation for you. It calculates the change in weights as

$$dw = lr * e * p'$$

and the bias change as

$$db = lr * e$$

The constant 2, shown a few lines above, has been absorbed into the code learning rate lr. The function [maxlinlr](#) calculates this maximum stable learning rate lr as $0.999 * P' * P$.

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

8 .1 .8 Linear Classification (train)

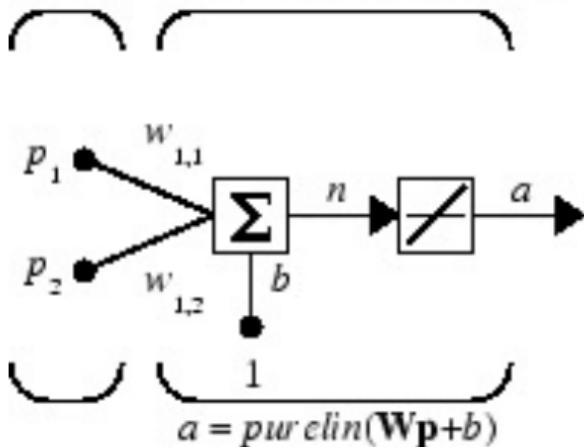
Linear networks can be trained to perform linear classification with the function [train](#). This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to [learnp](#). Then the network is adjusted with the sum of all these corrections. Each pass through the input vectors is called an *epoch*. This contrasts with [adapt](#) which adjusts weights for each input vector as it is presented.

Finally, [train](#) applies the inputs to the

new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped, and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

A simple problem illustrates this procedure. Consider the linear network introduced earlier.

Input Simple Linear Network



Suppose you have the following classification problem.

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

Here there are four input vectors, and you want a network that produces the output corresponding to each input vector when that vector is presented.

Use [train](#) to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network are 0 by default. Set the error goal to 0.1 rather than accept its default of 0.

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

```
T = [0 1 0 1];
```

```
net = linearlayer;
```

```
net.trainParam.goal= 0.1;
```

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

The problem runs for 64 epochs, achieving a mean square error of 0.0999. The new weights and bias are

```
weights = net.iw{1,1}
```

weights =

-0.0615 -0.2194

bias = net.b(1)

bias =

[0.5899]

You can simulate the new network as shown below.

A = net(P)

A =

0.0282 0.9672 0.2741
0.4320

You can also calculate the error.

err = T - sim(net,P)

err =

-0.0282 0.0328 -0.2741

0.5680

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had a smaller error goal been chosen, but in this problem it is not possible to obtain a goal of 0. The network is limited in its capability.

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

You might also try running the example program nnd10lc. It addresses a classic

and historically interesting problem, shows how a network can be trained to classify various patterns, and shows how the trained network responds when noisy patterns are presented.

8 .1 .9 Limitations and Cautions

Linear networks can only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate lr is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Because parabolas have only one

minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have various other limitations. Some of them are discussed below.

Overdetermined Systems

Consider an overdetermined system. Suppose that you have a network to be trained with four one-element input vectors and four targets. A perfect solution to $wp + b = t$ for each of the inputs might not exist, for there are four constraining equations, and only one weight and one bias to adjust. However, the LMS rule still minimizes the error.

You might try `demolin4` to see how this is done.

Underdetermined Systems

Consider a single linear neuron with one input. This time, in `demolin5`, train it on only one one-element input vector and its one-element target vector:

$$P = [1.0];$$

$$T = [0.5];$$

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can

try demolin5 to explore this topic.

Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ($S * R + S =$ number of weights and biases) as constraints ($Q =$ pairs of input/target vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with the example demolin6, the network cannot solve the problem with zero error. You

might want to try demolin6.

Too Large a Learning Rate

You can always train a linear network with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Example demolin7 shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by [maxlinlr](#). The network is trained with two different learning rates to show the results of using too large a learning rate.

8.2 HOPFIELD NEURAL NETWORK

8.2.1 Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points.

The design method presented is not perfect in that the designed network can have spurious undesired equilibrium points in addition to the desired ones.

However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

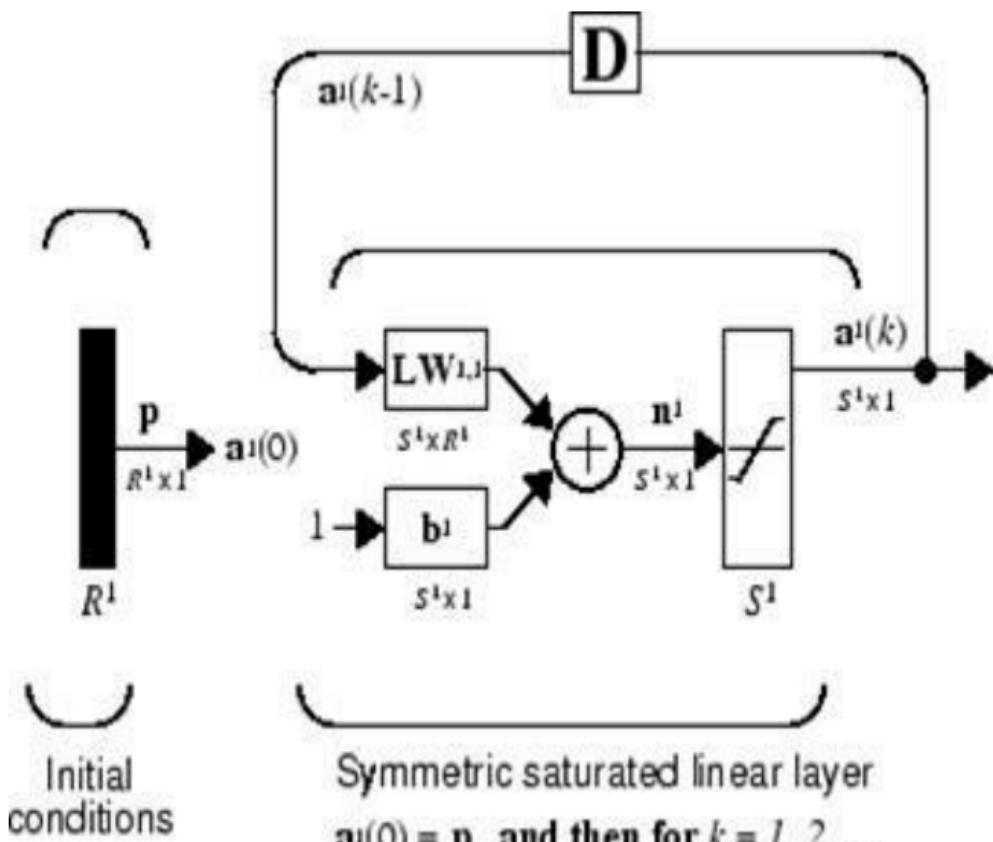
The material in this section is based on the following paper: Jian-Hua Li, Anthony N. Michel, and Wolfgang

Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Trans. on Circuits and Systems*, Vol. 36, No. 11, November 1989, pp. 1405–22.

For further information on Hopfield networks, see Chapter 18, "Hopfield Network," of Hagan, Demuth, and Beale.

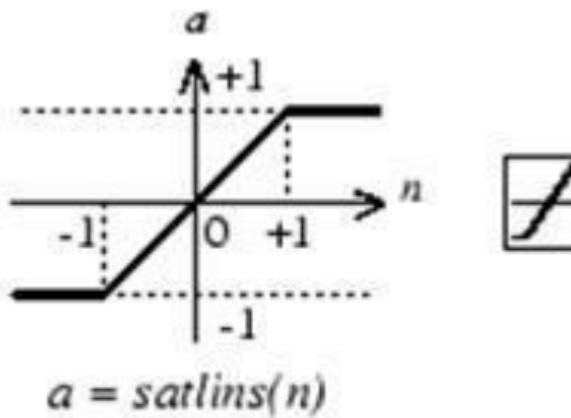
8.2.2 Architecture

The architecture of the Hopfield network follows.



As noted, the *input p* to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function [satlins](#).



Satlins Transfer Function

For inputs less than -1 [satlins](#) produces -1 . For inputs in the range -1 to $+1$ it simply returns the input value. For inputs greater than $+1$ it produces $+1$.

This network can be tested with one or more input vectors that are presented as initial conditions to the network. After the initial conditions are given, the network produces an output that is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

8.2.3 Design (newhop)

Li et al. have studied a system that has the basic structure of the Hopfield network but is, in Li's own words, "easier to analyze, synthesize, and implement than the Hopfield model." The authors are enthusiastic about the reference article, as it has many excellent points and is one of the most readable in the field. However, the design is mathematically complex, and even a short justification of it would burden this guide. Thus the Li design method is presented, with thanks to Li et al., as a recipe that is found in the function newhop.

Given a set of target equilibrium points represented as a matrix T of vectors, `newhop` returns weights and biases for a recursive network. The network is guaranteed to have stable equilibrium points at the target vectors, but it could contain other spurious equilibrium points as well. The number of these undesired points is made as small as possible by the design method.

Once the network has been designed, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium points will find their targets. As suggested by the network figure, an array of input vectors is presented one at a time or in a batch.

The network proceeds to give output vectors that are fed back as inputs. These output vectors can be compared to the target vectors to see how the solution is proceeding.

The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time. First you might check to see that the target equilibrium point vectors are indeed contained in the network. Then you could try other input vectors to determine the domains of attraction of the target equilibrium points and the locations of spurious equilibrium points if they are present.

Consider the following design example.

Suppose that you want to design a network with two stable points in a three-dimensional space.

$$T = [-1 \ -1 \ 1; 1 \ -1 \ 1]'$$

$$T =$$

$$\begin{matrix} -1 & 1 \end{matrix}$$

$$\begin{matrix} -1 & -1 \end{matrix}$$

$$\begin{matrix} 1 & 1 \end{matrix}$$

You can execute the design with

$$\text{net} = \text{newhop}(T);$$

Next, check to make sure that the designed network is at these two points, as follows. Because Hopfield networks have no inputs, the first argument to the

network is an empty cell array whose columns indicate the number of time steps.

$$A_i = \{T\};$$

$$[Y, Pf, Af] = \text{net}(\text{cell}(1, 2), \{\}, A_i);$$

$$Y\{2\}$$

This gives you

$$\begin{matrix} -1 & 1 \end{matrix}$$

$$\begin{matrix} -1 & -1 \end{matrix}$$

$$\begin{matrix} 1 & 1 \end{matrix}$$

Thus, the network has indeed been designed to be stable at its design points. Next you can try another input condition that is not a design point, such as

$$Ai = \{[-0.9; -0.8; 0.7]\};$$

This point is reasonably close to the first design point, so you might anticipate that the network would converge to that first point. To see if this happens, run the following code.

$$[Y, Pf, Af] = \text{net}(\text{cell}(1, 5), \{\}, Ai);$$

$$Y\{end\}$$

This produces

-1

-1

1

Thus, an original condition close to a design point did converge to that point.

This is, of course, the hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include spurious undesired stable points that attract the solution.

Example

Consider a Hopfield network with just two neurons. Each neuron has a bias and weights to accommodate two-element input vectors weighted. The target equilibrium points are defined to be stored in the network as the two columns of the matrix \mathbf{T} .

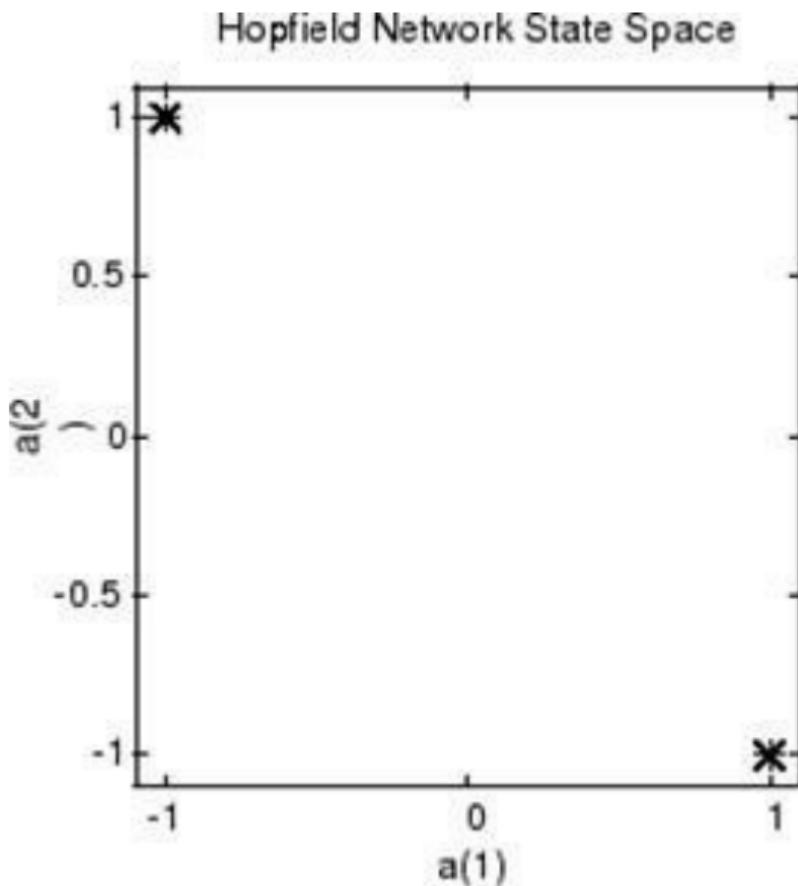
$$\mathbf{T} = [1 \ -1; \ -1 \ 1]'$$

$$\mathbf{T} =$$

$$1 \quad -1$$

-1 1

Here is a plot of the Hopfield state space with the two stable points labeled with * markers.



These target stable points are given to newhop to obtain weights and biases of a Hopfield network.

```
net = newhop(T);
```

The design returns a set of weights and a bias for each neuron. The results are obtained from

```
W = net.LW{1,1}
```

which gives

```
W =
```

```
0.6925 -0.4694
```

```
-0.4694 0.6925
```

and from

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

which gives

$$\mathbf{b} =$$

$$0$$

$$0$$

Next test the design with the target vectors \mathbf{T} to see if they are stored in the network. The targets are used as inputs for the simulation function [sim](#).

$$\mathbf{Ai} = \{\mathbf{T}\};$$

$$[\mathbf{Y}, \mathbf{Pf}, \mathbf{Af}] = \text{net}(\text{cell}(1,2), \{\}, \mathbf{Ai});$$

$$\mathbf{Y} = \mathbf{Y}\{\text{end}\}$$

$$\mathbf{ans} =$$

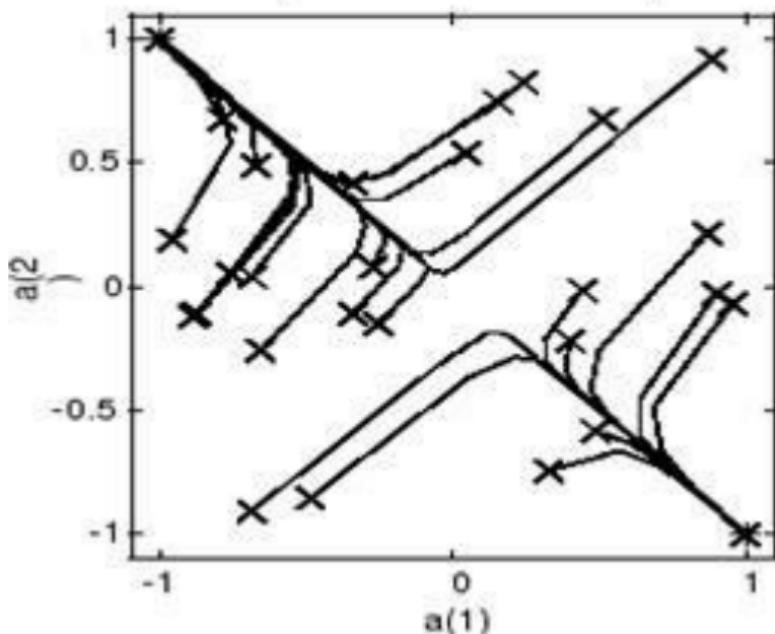
$$1 \quad -1$$

-1 1

As hoped, the new network outputs are the target vectors. The solution stays at its initial conditions after a single update and, therefore, will stay there for any number of updates.

Now you might wonder how the network performs with various random input vectors. Here is a plot showing the paths that the network took through its state space to arrive at a target point.

Hopfield Network State Space



This plot shows the trajectories of the solution for various starting points. You can try the example `demohop1` to see more of this kind of network behavior.

Hopfield networks can be designed for an arbitrary number of dimensions. You can try `demohop3` to see a three-

dimensional design.

Unfortunately, Hopfield networks can have both unstable equilibrium points and spurious stable points. You can try examples `demohop2` and `demohop4` to investigate these issues.

8.2.4 Summary

Hopfield networks can act as error correction or vector categorization networks. Input vectors are used as the initial conditions to the network, which recurrently updates until it reaches a stable output vector.

Hopfield networks are interesting from a theoretical standpoint, but are seldom used in practice. Even the best Hopfield designs may have spurious stable points that lead to incorrect answers. More efficient and reliable error correction techniques, such as backpropagation, are available.

8.3 LINEAR PREDICTION DESIGN EXAMPLE

This example illustrates how to design a linear neuron to predict the next value in a time series given the last five values.

8.3.1 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);
```

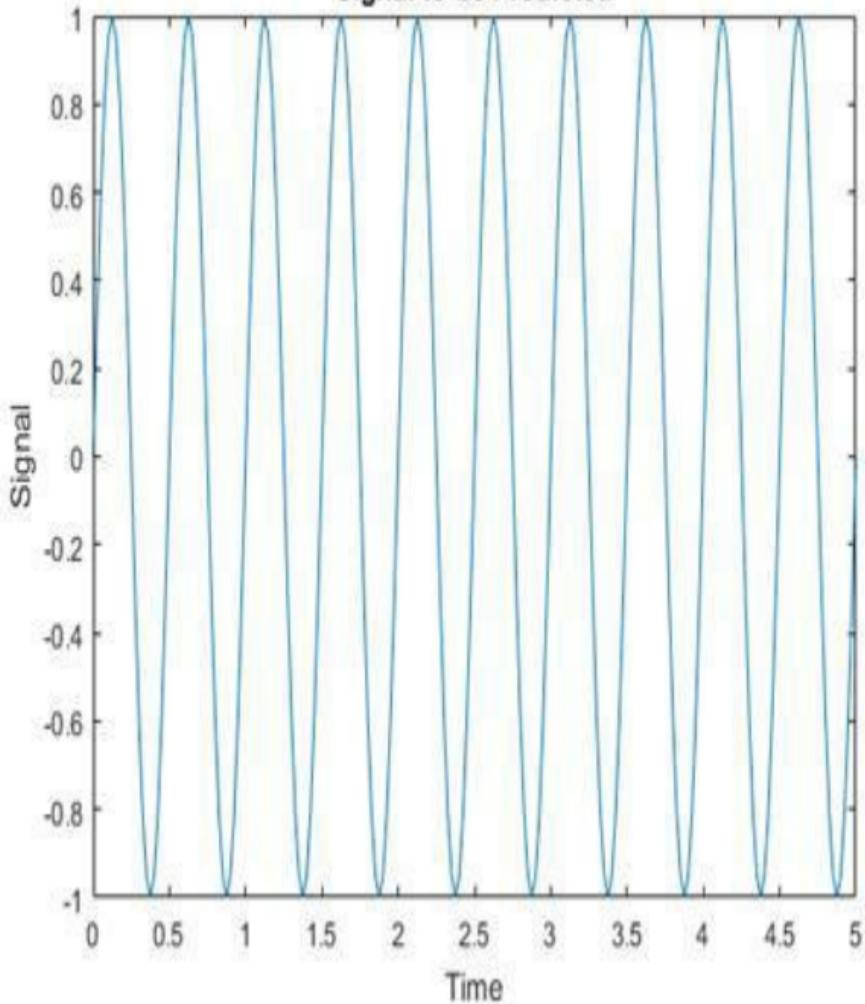
```
plot(time, signal)
```

```
xlabel('Time');
```

```
ylabel('Signal');
```

```
title('Signal to be  
Predicted');
```

Signal to be Predicted



8.3.2 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.

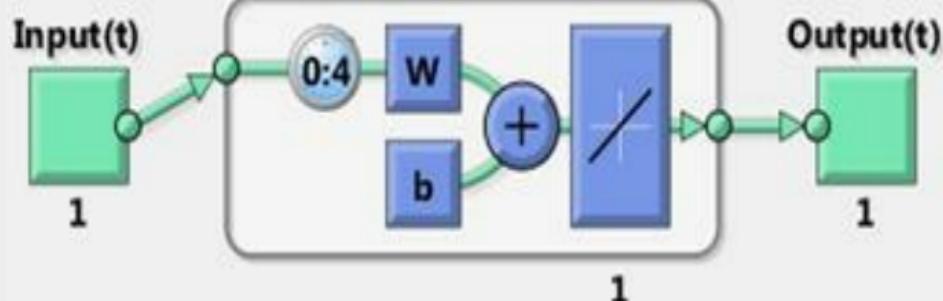
```
T = signal(6:end);
```

8.3.3 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



8.3.4 Testing the Linear Layer

The network can now be called like a function on the inputs and delayed states to get its time response.

$Y = \text{net}(X, X_i);$

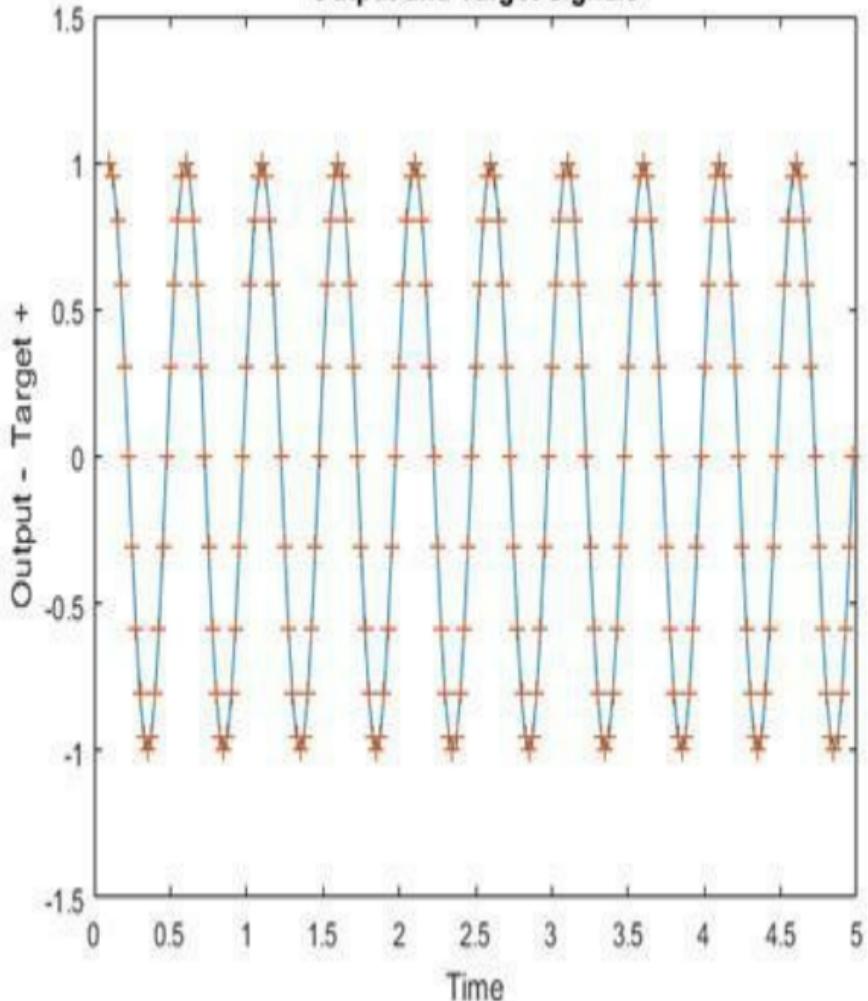
The output signal is plotted with the targets.

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);
```

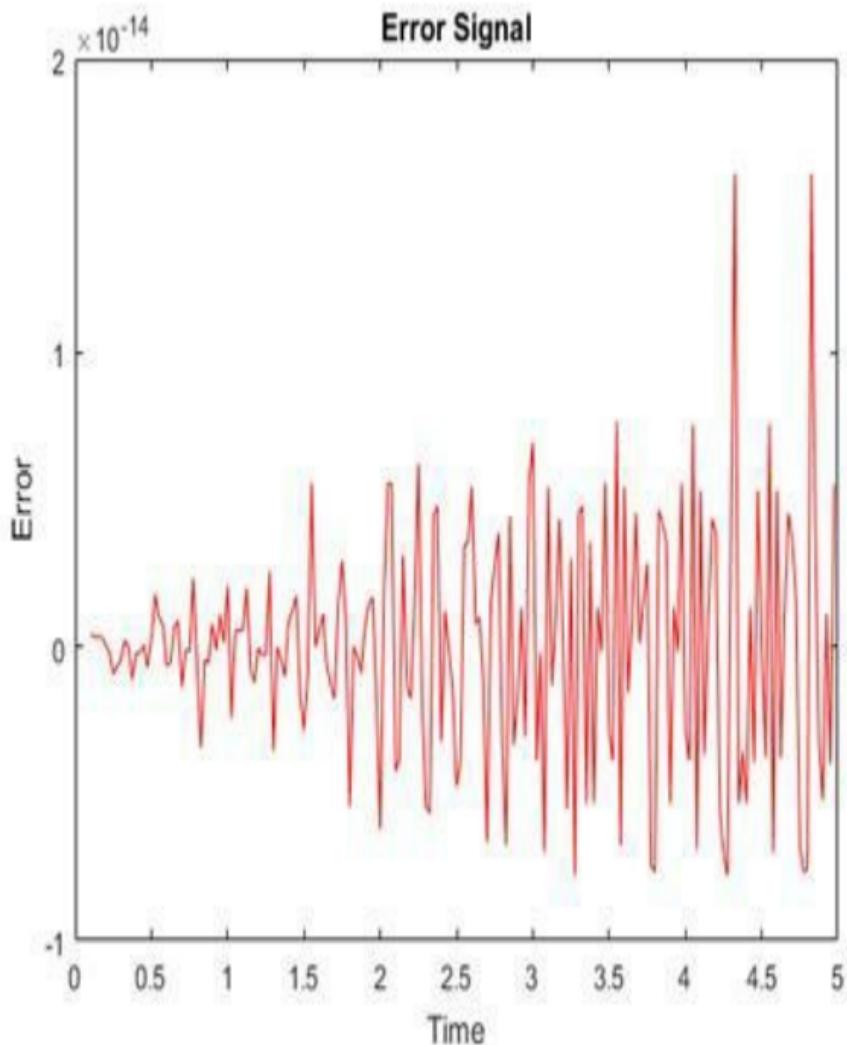
```
plot(timex, E, 'r')
```

```
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.

8.4 ADAPTIVE LINEAR PREDICTION EXAMPLE

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.

8.4.1 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.

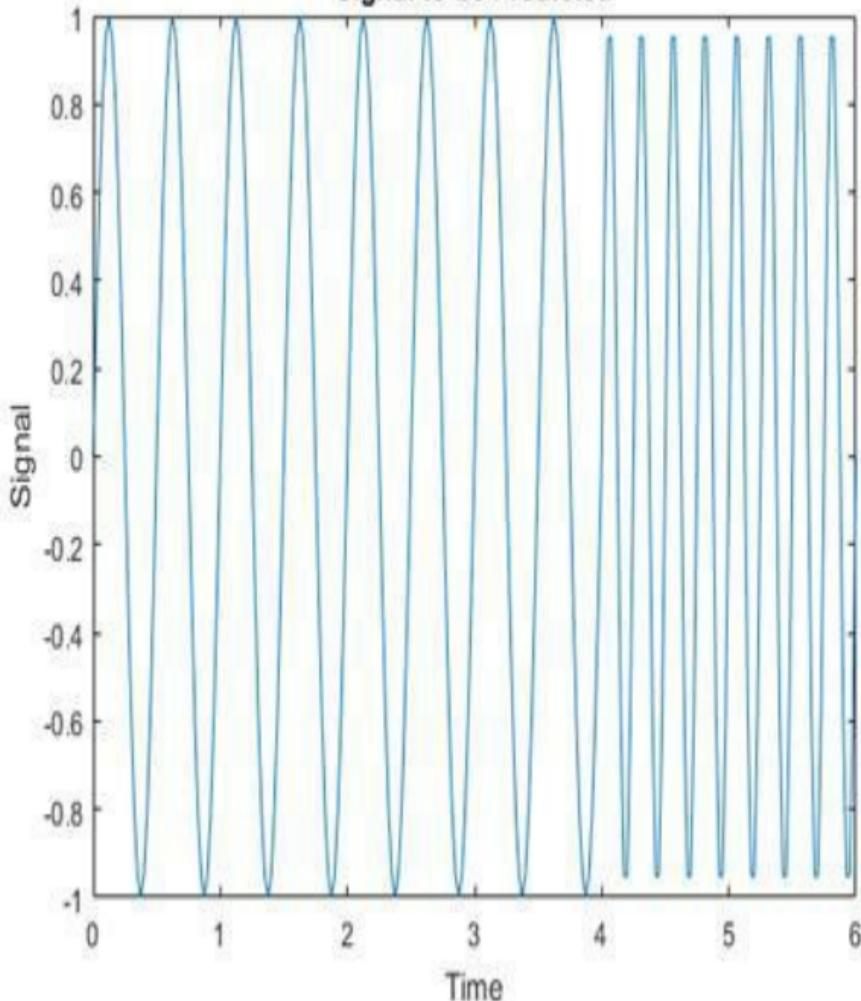
```
signal =  
[sin(time1*4*pi)  
sin(time2*8*pi)];  
  
plot(time, signal)
```

```
xlabel('Time');
```

```
ylabel('Signal');
```

```
title('Signal to be  
Predicted');
```

Signal to be Predicted



8.4.2 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.

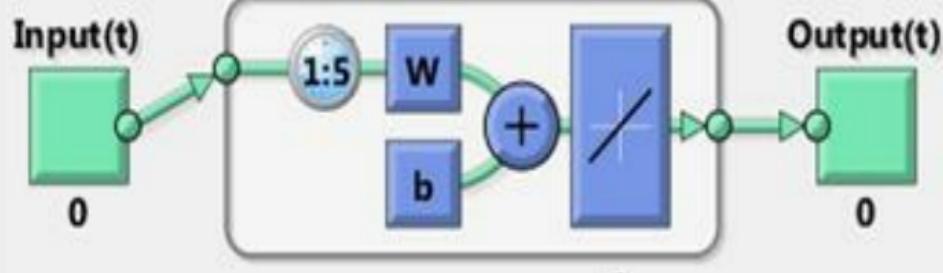
```
T = signal(6:end);
```

8.4.3 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



8.4.4 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(
```

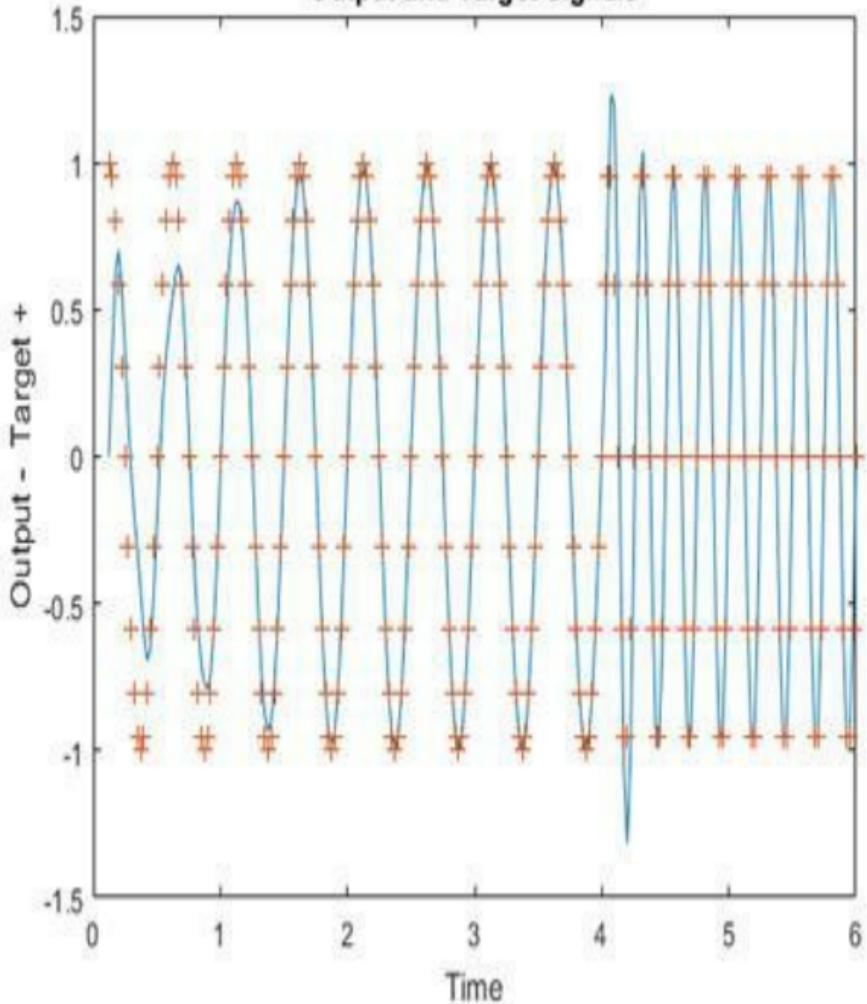
```
xlabel('Time');
```

```
ylabel('Output -  
Target +');
```

```
title('Output and
```

Target Signals');

Output and Target Signals



The error can also be plotted.

```
figure
```

```
E = cell2mat(T) -  
cell2mat(Y);
```

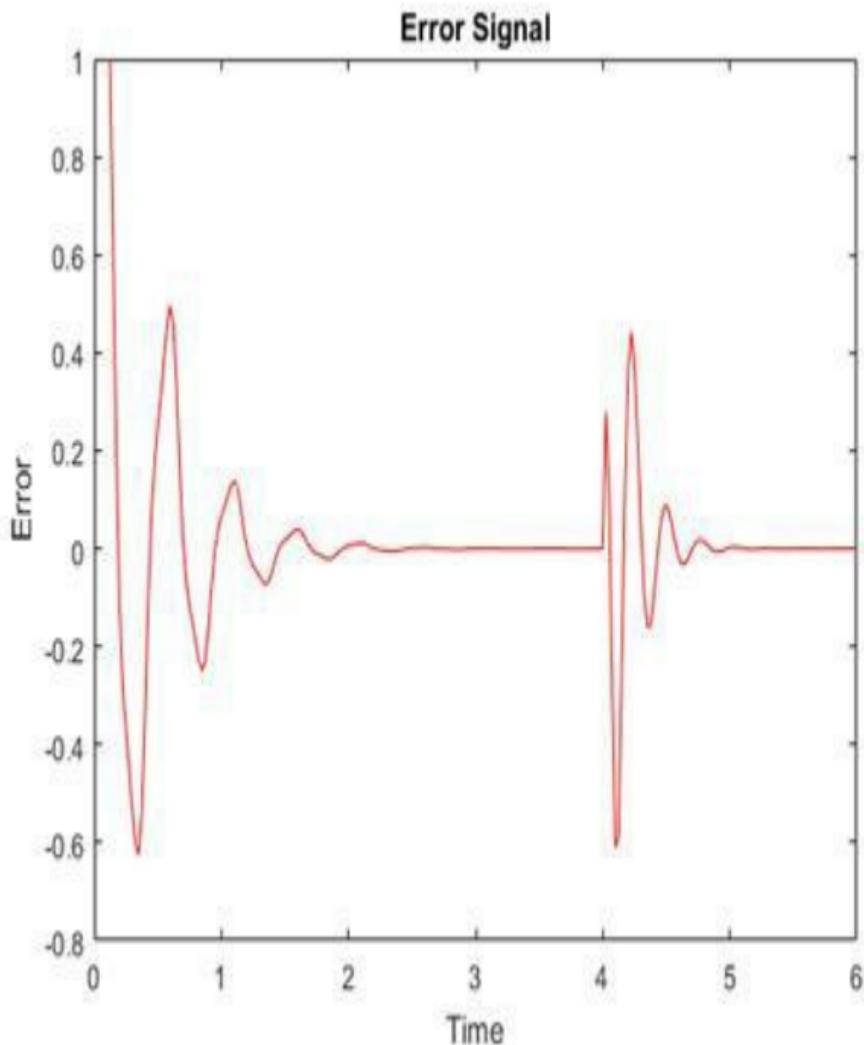
```
plot(timex, E, 'r')
```

```
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.

8.5 HOPFIELD TWO NEURON DESIGN EXAMPLE

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.

$$T = [+1 \ -1 ; \ \dots]$$

-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.

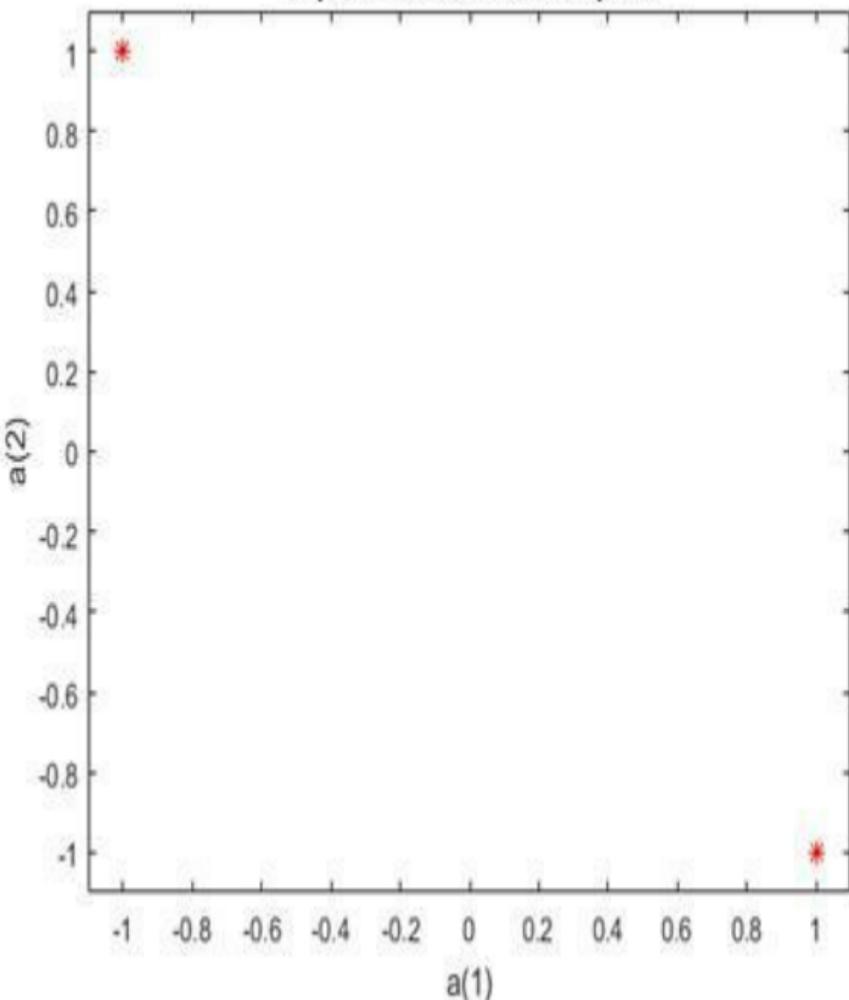
```
plot(T(1, :), T(2, :), '  
axis([-1.1 1.1 -1.1  
1.1])  
title('Hopfield  
Network State
```

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.

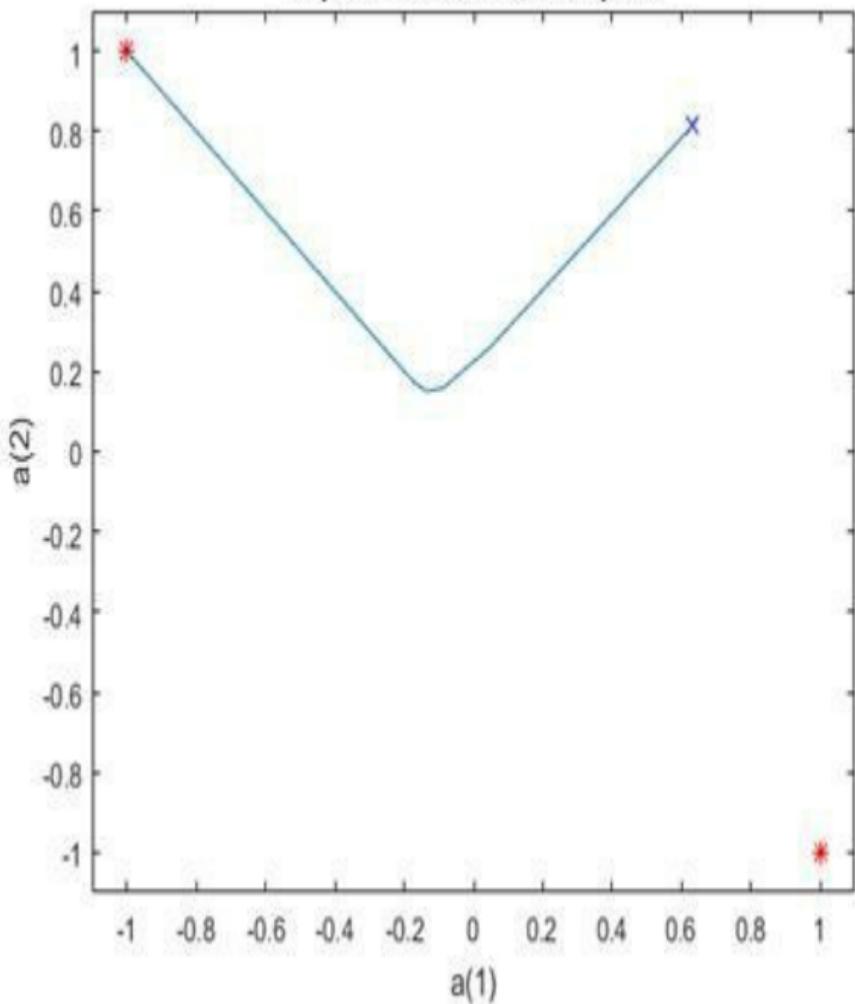
```
record =  
[cell2mat(a)  
cell2mat(y)];
```

```
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)
```

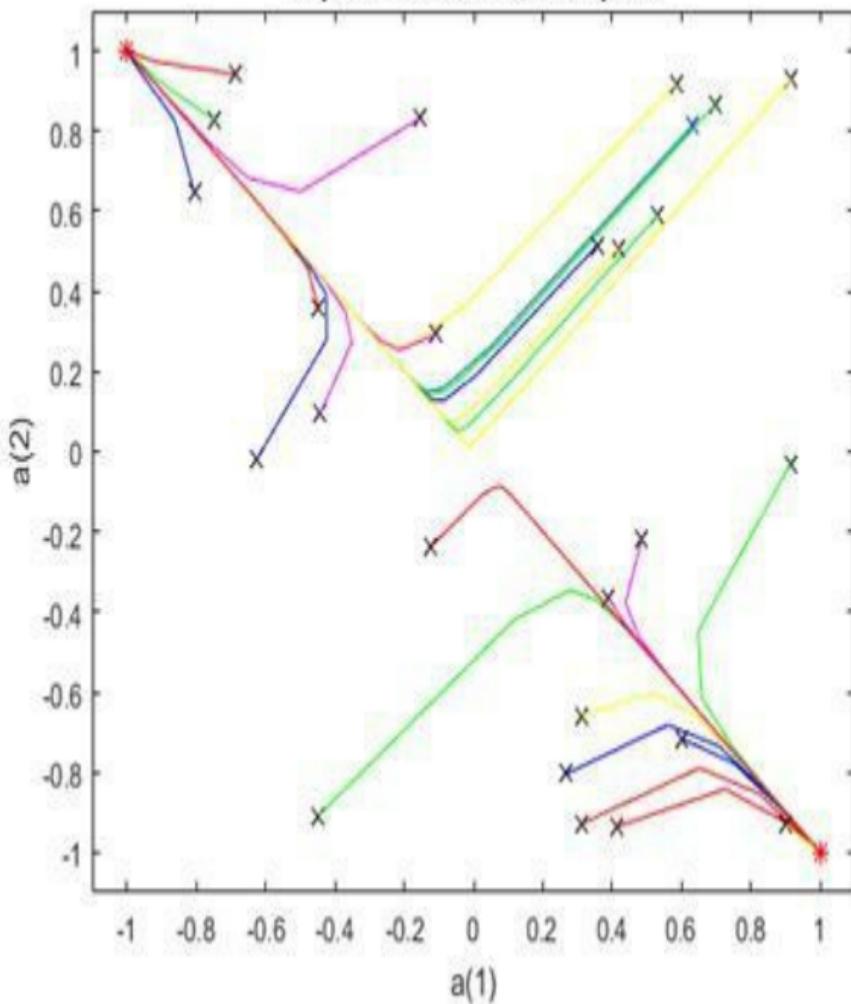
```
cell2mat(y) ] ;
```

```
start=cell2mat(a) ;
```

```
plot(start(1,1),star
```

```
end
```

Hopfield Network State Space



8.6 HOPFIELD UNSTABLE EQUILIBRIA EXAMPLE

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

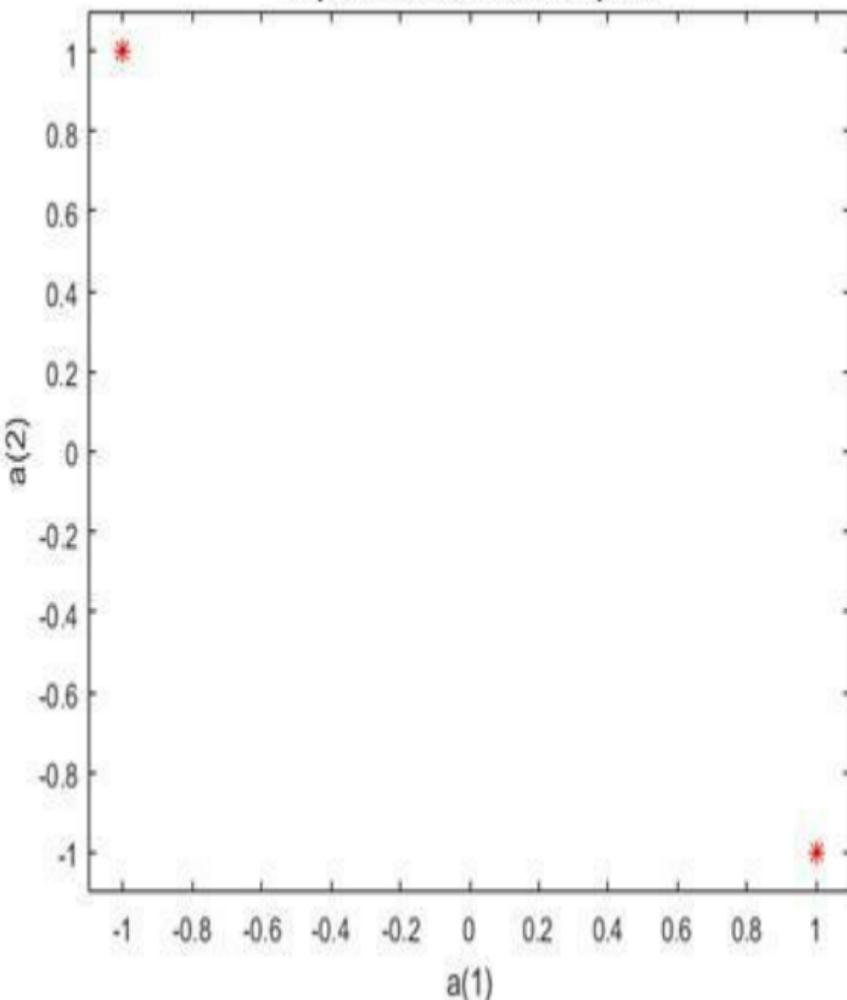
define by the two target (column) vectors in T.

$$T = [+1 \ -1; \dots \\ \ -1 \ +1];$$

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.

```
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.

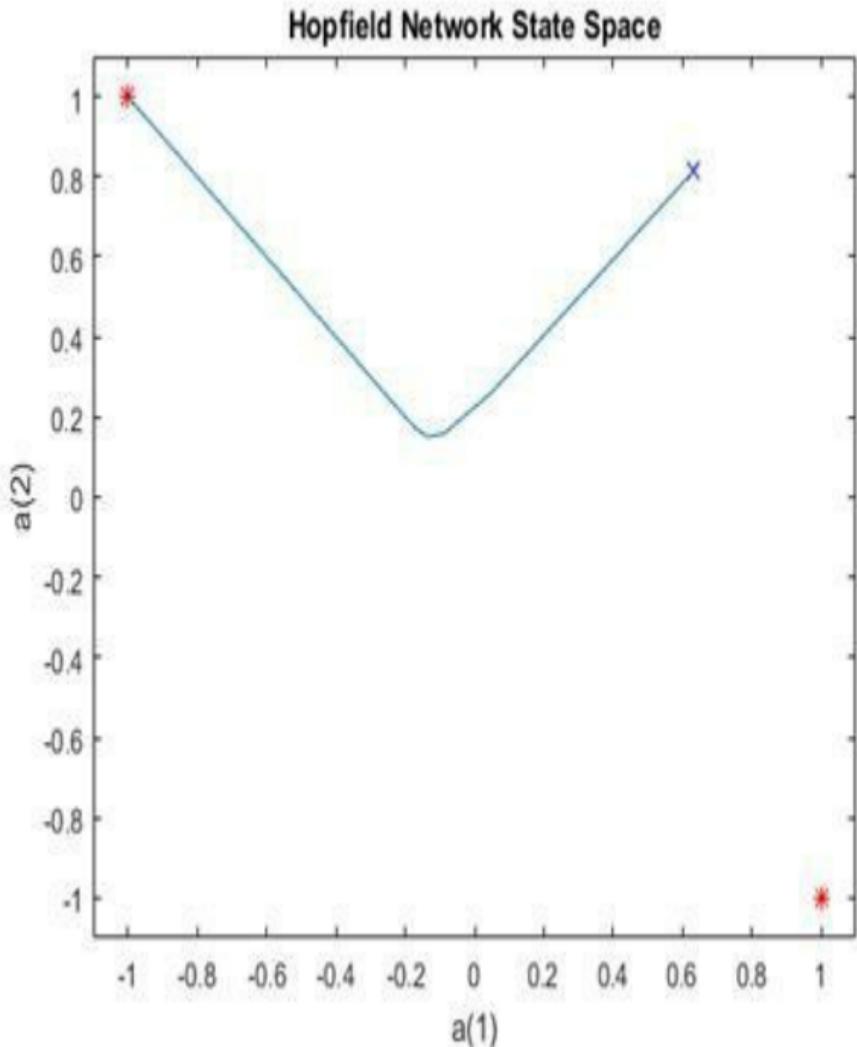
Sure enough, the network ends up in either the upper-left or lower right corners of the plot.

```
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.

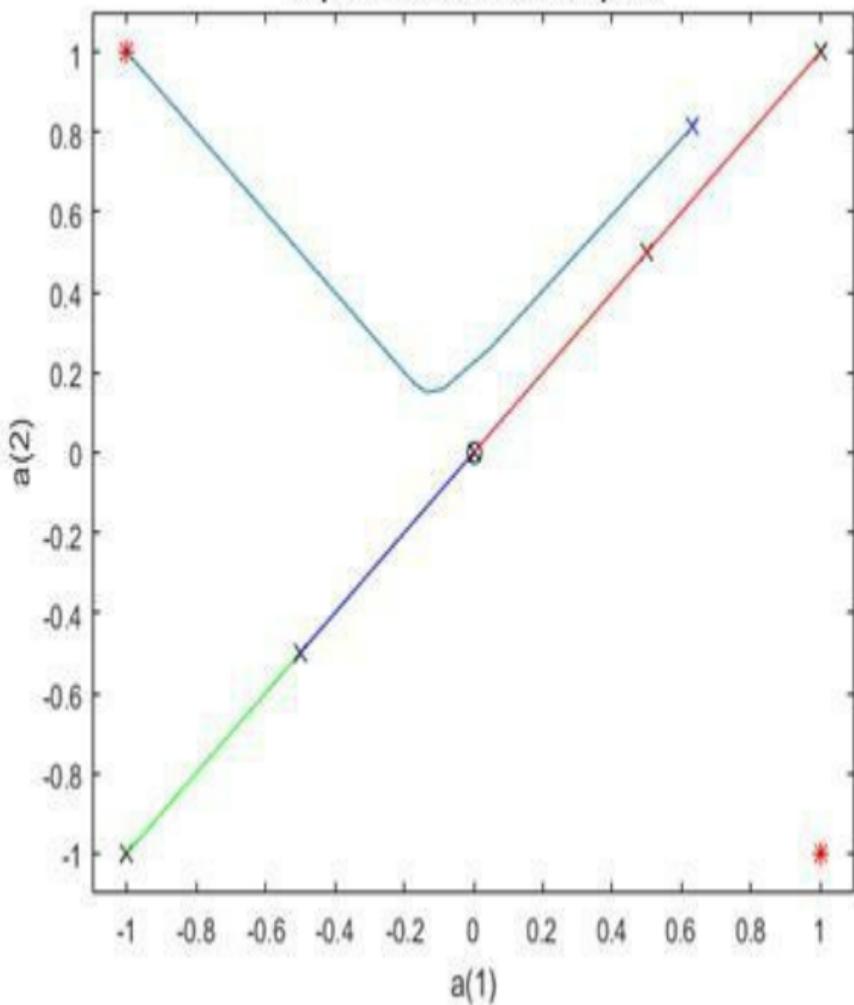
These points are exactly between the two target stable points. The result is that they all move into the center of the state space, where an undesired stable point exists.

```
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



8.7 HOPFIELD THREE NEURON DESIGN EXAMPLE

A Hopfield network is designed with target stable points. The behavior of the Hopfield network for different initial conditions is studied.

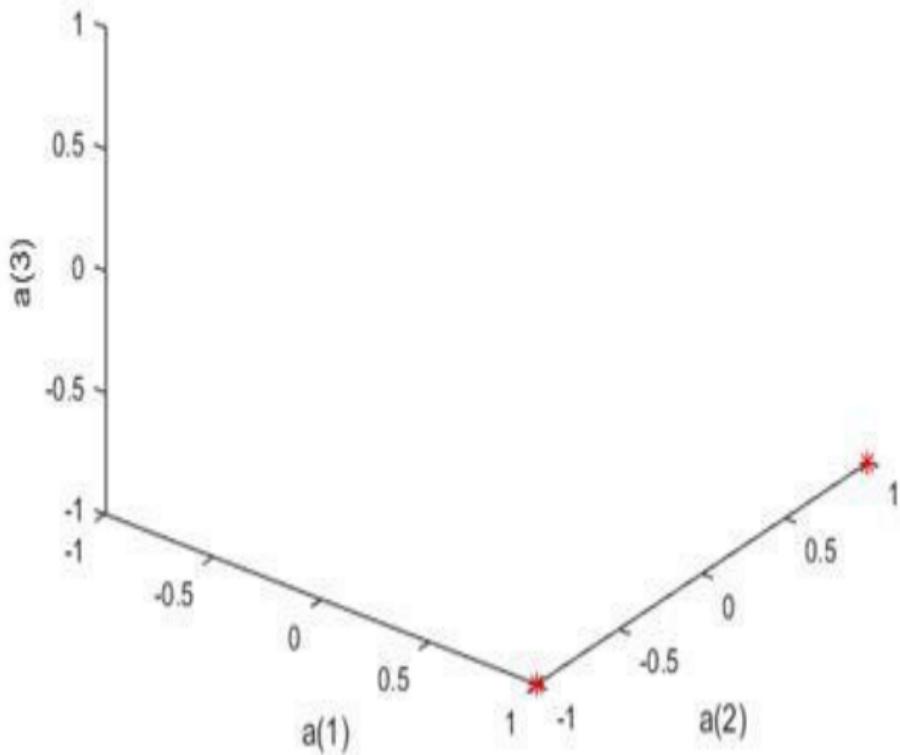
We would like to obtain a Hopfield network that has the two stable points defined by the two target (column) vectors in T.

$$T = [+1 \ 1; \dots \\ -1 \ 1; \dots \\ -1 \ -1];$$

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);
```

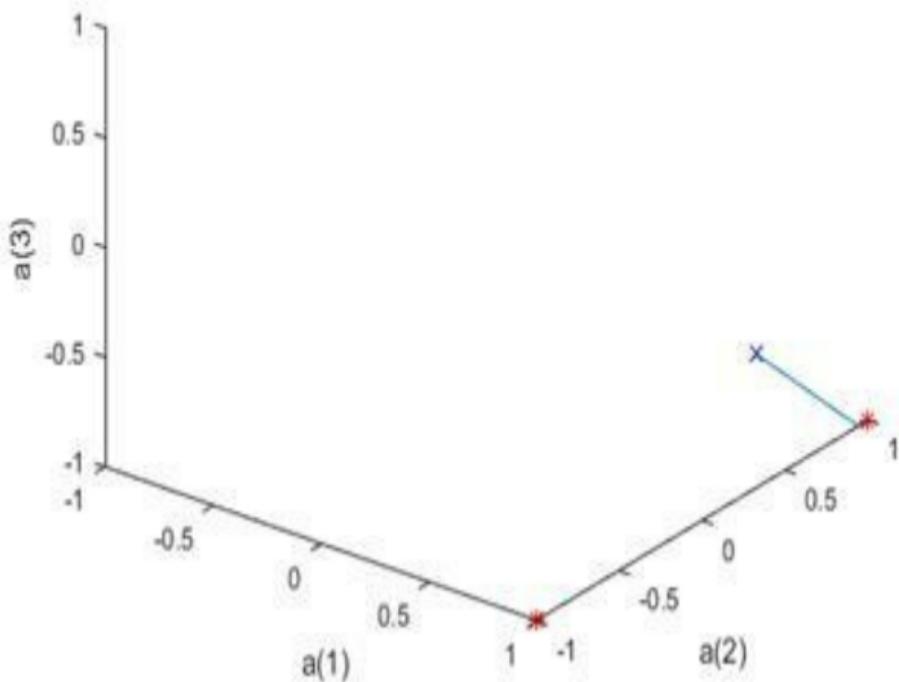
```
hold on
```

```
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);
```

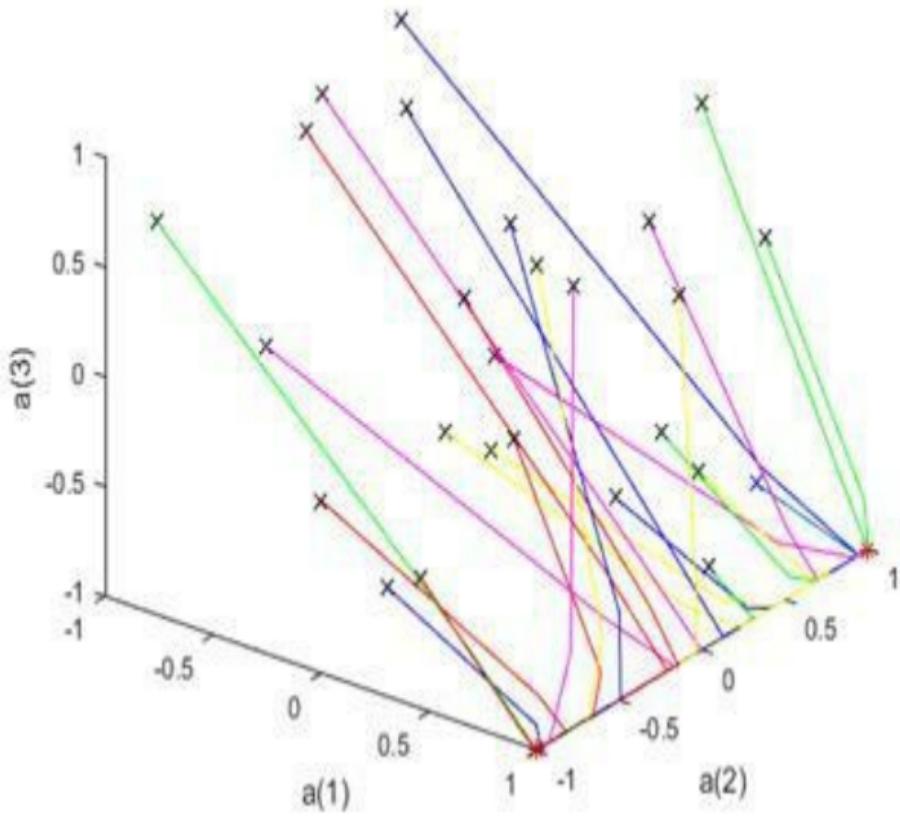
```
plot3(start(1,1),start(2,1),start(3,1),'kx',
```

```
...
```

```
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)];
```

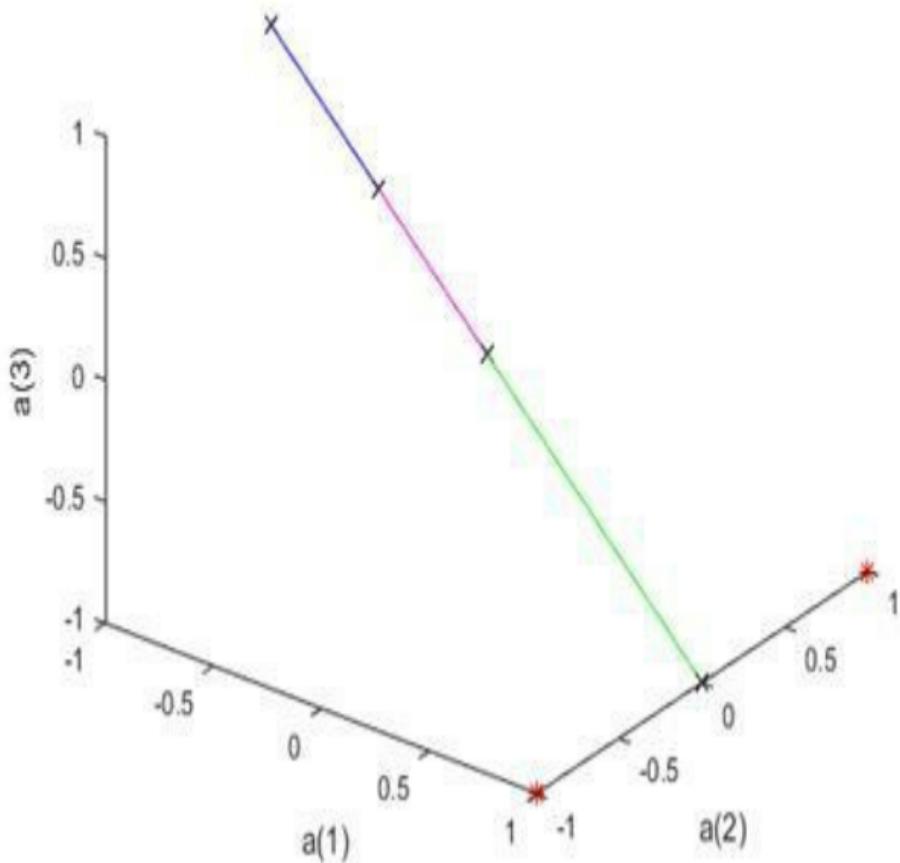
```
start = cell2mat(a);
```

```
plot3(start(1,1),start(2,1),start(3,1),'kx',
```

```
...
```

```
record(1,:),record(2,:),record(3,:),color(  
end
```

Hopfield Network State Space



8.8 HOPFIELD SPURIOUS STABLE POINTS EXAMPLE

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 9

CUSTOM NEURAL NETWORKS

9.1 CREATE NEURAL NETWORK OBJECT

The easiest way to create a neural network is to use one of the network creation functions. To investigate how this is done, you can create a simple, two-layer feedforward network, using the command [feedforwardnet](#):

```
net = feedforwardnet
```

```
net =
```

Neural Network

```
name: 'Feed-Forward Neural  
Network'
```

userdata: (your custom info)

dimensions:

numInputs: 1

numLayers: 2

numOutputs: 1

numInputDelays: 0

numLayerDelays: 0

numFeedbackDelays: 0

numWeightElements: 10

sampleTime: 1

connections:

biasConnect: [1; 1]

inputConnect: [1; 0]

layerConnect: [0 0; 1 0]

outputConnect: [0 1]

subobjects:

inputs: {1x1 cell array of 1
input}

layers: {2x1 cell array of 2
layers}

outputs: {1x2 cell array of 1
output}

biases: {2x1 cell array of 2
biases}

inputWeights: {2x1 cell array of 1
weight}

layerWeights: {2x2 cell array of 1
weight}

functions:

```
adaptFcn: 'adaptwb'  
adaptParam: (none)  
derivFcn: 'defaultderiv'  
divideFcn: 'dividerand'  
divideParam: .trainRatio, .valRatio,  
.testRatio  
divideMode: 'sample'  
initFcn: 'initlay'  
performFcn: 'mse'  
performParam: .regularization,  
.normalization  
plotFcns: {'plotperform',  
plottrainstate, ploterrhist,  
plotregression}
```

plotParams: {1x4 cell array of 4 params}

trainFcn: 'trainlm'

trainParam: .showWindow,
.showCommandLine, .show, .epochs,
.time, .goal, .min_grad,
.max_fail, .mu, .mu_dec,
.mu_inc, .mu_max

weight and bias values:

IW: {2x1 cell} containing 1 input weight matrix

LW: {2x2 cell} containing 1 layer weight matrix

b: {2x1 cell} containing 2 bias vectors

methods:

adapt: Learn while in continuous use

configure: Configure inputs & outputs

gensim: Generate Simulink model

init: Initialize weights & biases

perform: Calculate performance

sim: Evaluate network outputs given inputs

train: Train network with examples

view: View diagram

unconfigure: Unconfigure inputs &

outputs

evaluate: outputs = net(inputs)

This display is an overview of the network object, which is used to store all of the information that defines a neural network. There is a lot of detail here, but there are a few key sections that can help you to see how the network object is organized.

The dimensions section stores the overall structure of the network. Here you can see that there is one input to the network (although the one input can be a vector containing many elements), one network output, and two layers.

The connections section stores the connections between components of the network. For example, there is a bias connected to each layer, the input is connected to layer 1, and the output comes from layer 2. You can also see that layer 1 is connected to layer 2. (The rows of `net.layerConnect` represent the destination layer, and the columns represent the source layer. A one in this matrix indicates a connection, and a zero indicates no connection. For this example, there is a single one in element 2,1 of the matrix.)

The key subobjects of the network object are inputs, layers, outputs, biases, `inputWeights`, and `layerWeights`.

View the layers subobject for the first layer with the command

```
net.layers{1}
```

Neural Network Layer

```
    name: 'Hidden'  
    dimensions: 10  
    distanceFcn: (none)  
    distanceParam: (none)  
    distances: []  
    initFcn: 'initnw'  
    netInputFcn: 'netsum'  
    netInputParam: (none)  
    positions: []  
    range: [10x2 double]  
    size: 10  
    topologyFcn: (none)
```

```
transferFcn: 'tansig'  
transferParam: (none)  
userdata: (your custom info)
```

The number of neurons in a layer is given by its size property. In this case, the layer has 10 neurons, which is the default size for the [feedforwardnet](#) command. The net input function is [netsum](#) (summation) and the transfer function is the [tansig](#). If you wanted to change the transfer function to [logsig](#), for example, you could execute the command:

```
net.layers{1}.transferFcn = 'logsig';
```

To view the layerWeights subobject for the weight between layer 1 and layer 2,

use the command:

```
net.layerWeights{2,1}
```

Neural Network Weight

```
delays: 0
initFcn: (none)
initConfig: .inputSize
learn: true
learnFcn: 'learngdm'
learnParam: .lr, .mc
size: [0 10]
weightFcn: 'dotprod'
weightParam: (none)
userdata: (your custom info)
```

The weight function is dotprod, which represents standard matrix multiplication

(dot product). Note that the size of this layer weight is 0-by-10. The reason that we have zero rows is because the network has not yet been configured for a particular data set. The number of output neurons is equal to the number of rows in your target vector. During the configuration process, you will provide the network with example inputs and targets, and then the number of output neurons can be assigned.

This gives you some idea of how the network object is organized. For many applications, you will not need to be concerned about making changes directly to the network object, since that is taken care of by the network creation

functions. It is usually only when you want to override the system defaults that it is necessary to access the network object directly. Other topics will show how this is done for particular networks and training methods.

To investigate the network object in more detail, you might find that the object listings, such as the one shown above, contain links to help on each subobject. Click the links, and you can selectively investigate those parts of the object that are of interest to you.

9.2 CONFIGURE NEURAL NETWORK INPUTS AND OUTPUTS

After a neural network has been created, it must be configured. The configuration step consists of examining input and target data, setting the network's input and output sizes to match the data, and choosing settings for processing inputs and outputs that will enable best network performance. The configuration step is normally done automatically, when the training function is called. However, it can be done manually, by using the configuration function. For example, to configure the network you created

previously to approximate a sine function, issue the following commands:

```
p = -2:.1:2;
```

```
t = sin(pi*p/2);
```

```
net1 = configure(net,p,t);
```

You have provided the network with an example set of inputs and targets (desired network outputs). With this information, the [configure](#) function can set the network input and output sizes to match the data.

After the configuration, if you look again at the weight between layer 1 and layer 2, you can see that the dimension of the weight is 1 by 20. This is because the

target for this network is a scalar.

net1.layerWeights{2,1}

Neural Network Weight

delays: 0

initFcn: (none)

initConfig: .inputSize

learn: true

learnFcn: 'learngdm'

learnParam: .lr, .mc

size: [1 10]

weightFcn: 'dotprod'

weightParam: (none)

userdata: (your custom info)

In addition to setting the appropriate dimensions for the weights, the configuration step also defines the settings for the processing of inputs and outputs. The input processing can be located in the inputs subobject:

net1.inputs{1}

Neural Network Input

name: 'Input'

feedbackOutput: []

processFcns:

{'removeconstantrows',
mapminmax}

processParams: {1x2 cell array}

of 2 params}

processSettings: {1x2 cell array of
2 settings}

processedRange: [1x2 double]

processedSize: 1

range: [1x2 double]

size: 1

userdata: (your custom info)

Before the input is applied to the network, it will be processed by two functions: [removeconstantrows](#) and [mapn](#). These are discussed fully in [Multilayer Neural Networks and Backpropagation Training](#) so we won't address the particulars here. These processing

functions may have some processing parameters, which are contained in the subobject `net1.inputs{1}.processParam`. These have default values that you can override. The processing functions can also have configuration settings that are dependent on the sample data. These are contained

in `net1.inputs{1}.processSettings` and are set during the configuration process. For example, the [mapminmax](#) processing function normalizes the data so that all inputs fall in the range $[-1, 1]$. Its configuration settings include the minimum and maximum values in the sample data, which it needs to perform

the correct normalization. This will be discussed in much more depth in [Multilayer Neural Networks and Backpropagation Training](#).

As a general rule, we use the term "parameter," as in process parameters, training parameters, etc., to denote constants that have default values that are assigned by the software when the network is created (and which you can override). We use the term "configuration setting," as in process configuration setting, to denote constants that are assigned by the software from an analysis of sample data. These settings do not have default values, and should not generally be overridden.

9.3 CREATE AND TRAIN CUSTOM NEURAL NETWORK ARCHITECTURES

Neural Network Toolbox™ software provides a flexible network object type that allows many kinds of networks to be created and then used with functions such as [init](#), [sim](#), and [train](#).

Type the following to see all the network creation functions in the toolbox.

```
help nnetwork
```

This flexibility is possible because networks have an object-oriented

representation. The representation allows you to define various architectures and assign various algorithms to those architectures.

To create custom networks, start with an empty network (obtained with the [network](#) function) and set its properties as desired.

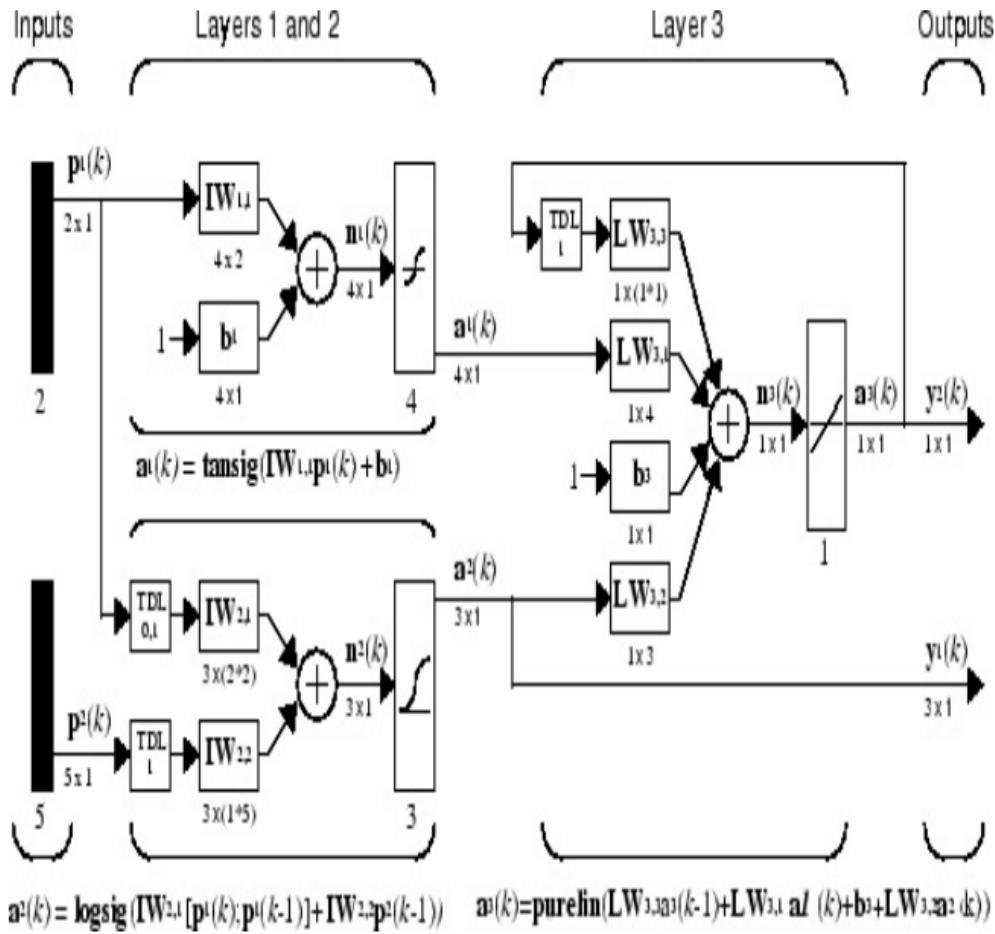
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network.

The following sections show how to create a custom network by using these properties.

9.3.1 Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input

ranges from -2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

Each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method ([initnw](#)). The network is trained with Levenberg-Marquardt backpropagation ([trainlm](#)), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error ([mse](#)).

9.3.2 Network Definition

The first step is to create a new network. Type the following code to create a network and view its many properties:

```
net = network
```

Architecture Properties

The first group of properties displayed is labeled architecture properties. These properties allow you to select the number of inputs and layers and their connections.

Number of Inputs and Layers. The first two properties displayed in the dimensions group

are `numInputs` and `numLayers`. These properties allow you to select how many inputs and layers you want the network to have.

`net =`

dimensions:

`numInputs: 0`

`numLayers: 0`

...

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in the custom network diagram.

```
net.numInputs = 2;
```

```
net.numLayers = 3;
```

net.numInputs is the number of input sources, not the number of elements in an input vector (net.inputs{i}.size).

Bias Connections. Type net and press **Enter** to view its properties again. The network now has two inputs and three layers.

```
net =
```

Neural Network:

dimensions:

numInputs: 2

numLayers: 3

Examine the next four properties in the connections group:

biasConnect: [0; 0; 0]

inputConnect: [0 0; 0 0; 0 0]

layerConnect: [0 0 0; 0 0 0; 0 0
0]

outputConnect: [0 0 0]

These matrices of 1s and 0s represent the presence and absence of bias, input weight, layer weight, and output connections. They are currently all zeros, indicating that the network does not have any such connections.

The bias connection matrix is a 3-by-1 vector. To create a bias connection to

the i th layer you can set `net.biasConnect(i)` to 1. Specify that the first and third layers are to have bias connections, as the diagram indicates, by typing the following code:

```
net.biasConnect(1) = 1;
```

```
net.biasConnect(3) = 1;
```

You could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

Input and Layer Weight Connections. The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three

destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the *i*th layer from the *j*th input.

To connect the first input to the first and second layers, and the second input to the second layer (as indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;
```

```
net.inputConnect(2,1) = 1;
```

```
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i,j)` represent

the presence of a layer-weight connection going to the i th layer from the j th layer. Connect layers 1, 2, and 3 to layer 3 as follows:

```
net.layerConnect = [0 0 0; 0 0 0; 1 1  
1];
```

Output Connections. The output connections are a 1-by-3 matrix, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to the network output, type

```
net.outputConnect = [0 1 1];
```

Number of Outputs

Type net and press **Enter** to view the updated properties. The final three architecture properties are read-only values, which means their values are determined by the choices made for other properties. The first read-only property in the dimension group is the number of outputs:

numOutputs: 2

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

Subobject Properties

The next group of properties in the output display is subobjects:

subobjects:

inputs: {2x1 cell array of 2 inputs}

layers: {3x1 cell array of 3 layers}

outputs: {1x3 cell array of 2 outputs}

biases: {3x1 cell array of 2 biases}

inputWeights: {3x2 cell array of 3 weights}

layerWeights: {3x3 cell array of

3 weights}

Inputs

When you set the number of inputs (net.numInputs) to 2, the inputs property becomes a cell array of two input structures. Each i th input structure (net.inputs{i}) contains additional properties associated with the i th input.

To see how the input structures are arranged, type

```
net.inputs
```

```
ans =
```

```
[1x1 nnetInput]
```

[1x1 nnetInput]

To see the properties associated with the first input, type

net.inputs{1}

The properties appear as follows:

ans =

name: 'Input'

feedbackOutput: []

processFcns: {}

processParams: {1x0 cell array
of 0 params}

processSettings: {0x0 cell array of
0 settings}

processedRange: []

processedSize: 0

range: []

size: 0

userdata: (your custom info)

If you set the exampleInput property, the range, size, processedSize, and processedRange properties will automatically be updated to match the properties of the value of exampleInput.

Set the exampleInput property as follows:

```
net.inputs{1}.exampleInput = [0 10  
5; 0 3 10];
```

If you examine the structure of the first input again, you see that it now has new values.

The property processFcns can be set to one or more processing functions. Type help nnprocess to see a list of these functions.

Set the second input vector ranges to be from -2 to 2 for five elements as follows:

```
net.inputs{1}.processFcns =  
'removeconstantrows','mapminmax'
```

View the new input properties. You will see that processParams, processSettings, processedRange and prc all been updated to reflect that inputs

will be processed using [removeconstantrows](#) and [mapminmax](#) before being given to the network when the network is simulated or trained. The

property processParams contains the default parameters for each processing function. You can alter these values, if you like. See the reference page for each processing function to learn more about their parameters.

You can set the size of an input directly when no processing functions are used:

```
net.inputs{2}.size = 5;
```

Layers. When you set the number of layers (net.numLayers) to 3,

the layers property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}
```

```
ans =
```

```
Neural Network Layer
```

```
name: 'Layer'
```

```
dimensions: 0
```

```
distanceFcn: (none)
```

```
distanceParam: (none)
```

```
distances: []
```

```
initFcn: 'initwb'
```

netInputFcn: 'netsum'

netInputParam: (none)

positions: []

range: []

size: 0

topologyFcn: (none)

transferFcn: 'purelin'

transferParam: (none)

userdata: (your custom info)

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to tansig, and its initialization function to the Nguyen-Widrow function, as required for the

custom network diagram.

```
net.layers{1}.size = 4;
```

```
net.layers{1}.transferFcn = 'tansig';
```

```
net.layers{1}.initFcn = 'initnw';
```

The second layer is to have three neurons, the logsig transfer function, and be initialized with initnw. Set the second layer's properties to the desired values as follows:

```
net.layers{2}.size = 3;
```

```
net.layers{2}.transferFcn = 'logsig';
```

```
net.layers{2}.initFcn = 'initnw';
```

The third layer's size and transfer function properties don't need to be

changed, because the defaults match those shown in the network diagram. You need to set only its initialization function, as follows:

```
net.layers{3}.initFcn = 'initnw';
```

Outputs. Use this line of code to see how the outputs property is arranged:

```
net.outputs
```

```
ans =
```

```
[]    [1x1 nnetOutput]    [1x1  
nnetOutput]
```

Note that outputs contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when net.outputConnect is

set to [0 1 1].

View the second layer's output structure with the following expression:

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

```
ans =
```

Neural Network Output

```
name: 'Output'
```

```
feedbackInput: []
```

```
feedbackDelay: 0
```

```
feedbackMode: 'none'
```

```
processFcns: {}
```

```
processParams: {1x0 cell array}
```

of 0 params}

processSettings: {0x0 cell array of
0 settings}

processedRange: [3x2 double]

processedSize: 3

range: [3x2 double]

size: 3

userdata: (your custom info)

The size is automatically set to 3 when the second layer's size (net.layers{2}.size) is set to that value. Look at the third layer's output structure if you want to verify that it also has the correct size.

Outputs have processing properties that are automatically applied to target values before they are used by the network during training. The same processing settings are applied in reverse on layer output values before they are returned as network output values during network simulation or training.

Similar to input-processing properties, setting the `exampleOutput` property automatically causes `size`, `range`, `processedSize`, and `processedRange` to be updated. Setting `processFcns` to a cell array list of processing function names causes `processParams`, `processSettings`, |

be updated. You can then alter the processParam values, if you want to.

Biases, Input Weights, and Layer Weights. Enter the following commands to see how bias and weight structures are arranged:

`net.biases`

`net.inputWeights`

`net.layerWeights`

Here are the results of typing `net.biases`:

`ans =`

`[1x1 nnetBias]`

`[]`

`[1x1 nnetBias]`

Each contains a structure where the corresponding connections (net.biasConnect, net.inputConnect, and net.layerConnect) contain a 1.

Look at their structures with these lines of code:

net.biases{1}

net.biases{3}

net.inputWeights{1,1}

net.inputWeights{2,1}

net.inputWeights{2,2}

net.layerWeights{3,1}

net.layerWeights{3,2}

net.layerWeights{3,3}

For example, typing `net.biases{1}` results in the following output:

`initFcn: (none)`

`learn: true`

`learnFcn: (none)`

`learnParam: (none)`

`size: 4`

`userdata: (your custom info)`

Specify the weights' tap delay lines in accordance with the network diagram by setting each weight's delays property:

```
net.inputWeights{2,1}.delays = [0  
1];
```

```
net.inputWeights{2,2}.delays = 1;  
net.layerWeights{3,3}.delays = 1;
```

Network Functions

Type net and press **Return** again to see the next set of properties.

functions:

adaptFcn: (none)

adaptParam: (none)

derivFcn: 'defaultderiv'

divideFcn: (none)

divideParam: (none)

divideMode: 'sample'

initFcn: 'initlay'

performFcn: 'mse'

performParam: .regularization,
.normalization

plotFcns: {}

plotParams: {1x0 cell array of 0
params}

trainFcn: (none)

trainParam: (none)

Each of these properties defines a function for a basic network operation.

Set the initialization function to initlay so the network initializes itself

according to the layer initialization functions already set to [initnw](#), the Nguyen-Widrow initialization function.

```
net.initFcn = 'initlay';
```

This meets the initialization requirement of the network.

Set the performance function to [mse](#) (mean squared error) and the training function to [trainlm](#) (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = 'mse';
```

```
net.trainFcn = 'trainlm';
```

Set the divide function to [dividerand](#) (divide training data

randomly).

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

During supervised training, the input and target data are randomly divided into training, test, and validation data sets. The network is trained on the training data until its performance begins to decrease on the validation data, which signals that generalization has peaked. The test data provides a completely independent test of network generalization.

Set the plot functions to [plotperform](#) (plot training, validation and test performance) and [plottrainstate](#) (plot the state of the

training algorithm with respect to epochs).

```
net.plotFcns =  
{'plotperform','plottrainstate'};
```

Weight and Bias Values

Before initializing and training the network, type net and press **Return**, then look at the weight and bias group of network properties.

weight and bias values:

IW: {3x2 cell} containing 3 input weight matrices

LW: {3x3 cell} containing 3

layer weight matrices

b: {3x1 cell} containing 2 bias vectors

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (net.inputConnect, net.layerConnect, net.biasConnect) contain 1s and the subobject properties (net.inputWeights, net.layerWeights, net.biases) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

net.IW{1,1},
net.IW{2,2}

net.IW{2,1},

net.LW{3,1},
net.LW{3,3}

net.LW{3,2},

net.b{1}, net.b{3}

Each input weight $\text{net.IW}\{i,j\}$, layer weight $\text{net.LW}\{i,j\}$, and bias vector $\text{net.b}\{i\}$ has as many rows as the size of the i th layer ($\text{net.layers}\{i\}.\text{size}$).

Each input weight $\text{net.IW}\{i,j\}$ has as many columns as the size of the j th input ($\text{net.inputs}\{j\}.\text{size}$) multiplied by the number of its delay values ($\text{length}(\text{net.inputWeights}\{i,j\}.\text{delays})$).

Likewise, each layer weight has as many columns as the size of the j th layer ($\text{net.layers}\{j\}.\text{size}$) multiplied by the number of its delay values

(length(net.layerWeights {i,j }.delays)).

9.3.3 Network Behavior

Initialization

Initialize your network with the following line of code:

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

Check the network's biases and weights again to see how they have changed:

```
net.IW{1,1},  
net.IW{2,2}
```

```
net.IW{2,1},
```

```
net.LW{3,1},  
net.LW{3,3}
```

```
net.LW{3,2},
```

```
net.b{1}, net.b{3}
```

For example,

net.IW{1,1}

ans =

-0.3040 0.4703

-0.5423 -0.1395

0.5567 0.0604

0.2667 0.4924

Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

X = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1]}

$[-1; -1; 1; 0; 1]\};$

You want the network to respond with the following target sequences for the second layer, which has three neurons, and the third layer with one neuron:

$T = \{[1; 1; 1] [0; 0; 0]; 1 -1\};$

Before training, you can simulate the network to see whether the initial network's response Y is close to the target T .

$Y = \text{sim}(\text{net}, X)$

$Y =$

$[3 \times 1 \text{ double}] \quad [3 \times 1 \text{ double}]$

$[\quad 1.7148] \quad [\quad 2.2726]$

The cell array Y is the output sequence of the network, which is also the output sequence of the second and third layers. The values you got for the second row can differ from those shown because of different initial weights and biases. However, they will almost certainly not be equal to targets T, which is also true of the values shown.

The next task is optional. On some occasions you may wish to alter the training parameters before training. The following line of code displays the default Levenberg-Marquardt training parameters (defined when you set net.trainFcn to [trainlm](#)).

`net.trainParam`

The following properties should be displayed.

ans =

 Show Training Window
Feedback showWindow: true

 Show Command Line Feedback
showCommandLine: false

 Command Line
Frequency show: 25

 Maximum Epochs
epochs: 1000

 Maximum Training
Time time: Inf

 Performance Goal
goal: 0

Minimum Gradient
min_grad: 1e-07

Maximum Validation Checks
max_fail: 6

Mu mu:
0.001

Mu Decrease Ratio
mu_dec: 0.1

Mu Increase Ratio
mu_inc: 10

Maximum mu
mu_max: 10000000000

You will not often need to modify these values. See the documentation for the training function for information about

what each of these means. They have been initialized with default values that work well for a large range of problems, so there is no need to change them here.

Next, train the network with the following call:

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

Training launches the neural network training window. To open the performance and training state plots, click the plot buttons.

After training, you can simulate the network to see if it has learned to respond correctly:

```
Y = sim(net,X)
```

```
[3x1 double] [3x1 double]  
[ 1.0000] [-1.0000]
```

The second network output (i.e., the second row of the cell array Y), which is also the third layer's output, matches the target sequence T.

Chapter 10

BIBLIOGRAPHY

10.1 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.

