network with bipolar sigmoidal units ($\lambda = 1$) to achieve the following two-to-one mappings:

- $y = 6\sin(\pi x_1) + \cos(\pi x_2)$
- $y = \sin(\pi x_1)\cos(0.2\pi x_2)$

Set up two sets of data, each consisting of 10 input-output pairs, one for training and other for testing. The input–output data are obtained by varying input variables (x_1, x_2) within $[-1, +1]$ randomly. Also the output data are normalized within $[-1, 1]$. Apply training to find proper weights in the network.

Omsairam

3 Supervised Learning Network

Learning Objectives

- The basic networks in supervised learning.
- How the perceptron learning rule is better than the Hebb rule.
- Original perceptron layer description.
- Delta rule with single output unit.
- Architecture, flowchart, training algorithm and testing algorithm for perceptron, Adaline, Madaline, back-propagation and radial basis function network.
- The various learning factors used in BPN.
- An overview of Time Delay, Function Link, Wavelet and Tree Neural Networks.
- Difference between back-propagation and RBF networks.

3.1 Introduction

The chapter covers major topics involving supervised learning networks and their associated single-layer and multilayer feed-forward networks. The following topics have been discussed in detail – the perceptron learning rule for simple perceptrons, the delta rule (Widrow-Hoff rule) for Adaline and single-layer feed-forward networks with continuous activation functions, and the back-propagation algorithm for multilayer feed-forward networks with continuous activation functions. In short, all the feed-forward networks have been explored.

3.2 Perceptron Networks

3.2.1 Theory

Perceptron networks come under single-layer feed-forward networks and are also called *simple perceptrons.* As described in Table 2-2 (Evolution of Neural Networks) in Chapter 2, various types of perceptrons were designed by Rosenblatt (1962) and Minsky-Papert (1969, 1988). However, a simple perceptron network was discovered by Block in 1962.

The key points to be noted in a perceptron network are:

1. The perceptron network consists of three units, namely, sensory unit (input unit), associator unit (hidden unit), response unit (output unit).

2. The sensory units are connected to associator units with fixed weights having values 1, 0 or −1, which are assigned at random.
3. The binary activation function is used in sensory unit and associator unit.
4. The response unit has an activation of 1, 0 or −1. The binary step with fixed threshold θ is used as activation for associator. The output signals that are sent from the associator unit to the response unit are only binary.
5. The output of the perceptron network is given by

$$y = f(y_{in})$$

where $f(y_{in})$ is activation function and is defined as

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

6. The perceptron learning rule is used in the weight updation between the associator unit and the response unit. For each training input, the net will calculate the response and it will determine whether or not an error has occurred.
7. The error calculation is based on the comparison of the values of targets with those of the calculated outputs.
8. The weights on the connections from the units that send the nonzero signal will get adjusted suitably.
9. The weights will be adjusted on the basis of the learning rule if an error has occurred for a particular training pattern, i.e.,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

If no error occurs, there is no weight updation and hence the training process may be stopped. In the above equations, the target value "t" is +1 or −1 and α is the learning rate. In general, these learning rules begin with an initial guess at the weight values and then successive adjustments are made on the basis of the evaluation of an objective function. Eventually, the learning rules reach a near-optimal or optimal solution in a finite number of steps.

A perceptron network with its three units is shown in Figure 3-1. As shown in Figure 3-1, a sensory unit can be a two-dimensional matrix of 400 photodetectors upon which a lighted picture with geometric black and white pattern impinges. These detectors provide a binary (0) electrical signal if the input signal is found to exceed a certain value of threshold. Also, these detectors are connected randomly with the associator unit. The associator unit is found to consist of a set of subcircuits called *feature predicates*. The feature predicates are hard-wired to detect the specific feature of a pattern and are equivalent to the *feature detectors*. For a particular feature, each predicate is examined with a few or all of the responses of the sensory unit. It can be found that the results from the predicate units are also binary (0 or 1). The last unit, i.e. response unit, contains the pattern-recognizers or perceptrons. The weights present in the input layers are all fixed, while the weights on the response unit are trainable.

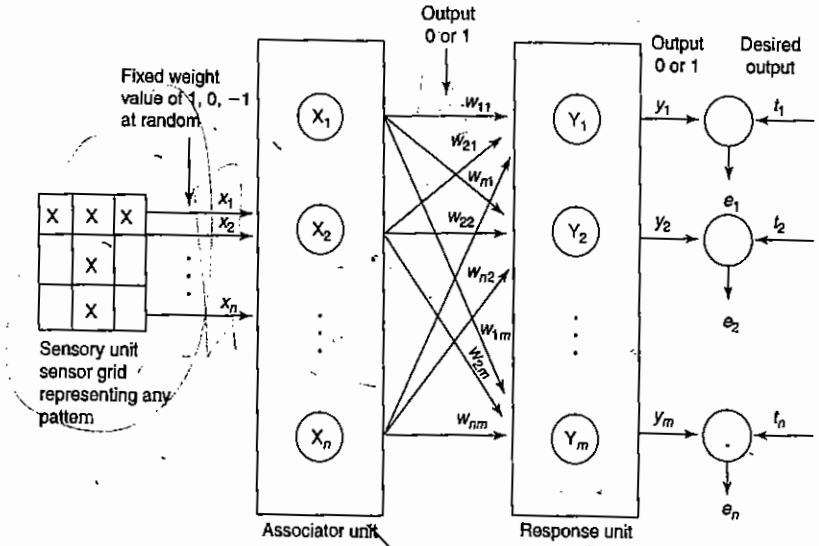

Figure 3-1 Original perceptron network.

3.2.2 Perceptron Learning Rule

In case of the perceptron learning rule, the learning signal is the difference between the desired and actual response of a neuron. The perceptron learning rule is explained as follows:

Consider a finite "n" number of input training vectors, with their associated target (desired) values $x(n)$ and $t(n)$, where "n" ranges from 1 to N. The target is either +1 or −1. The output "y" is obtained on the basis of the net input calculated and activation function being applied over the net input.

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

The weight updation in case of perceptron learning is as shown.

If $y \neq t$, then

$$w(\text{new}) = w(\text{old}) + \alpha t x \quad (\alpha - \text{learning rate})$$

else, we have

$$w(\text{new}) = w(\text{old})$$

The weights can be initialized at any values in this method. The perceptron rule convergence theorem states that "If there is a weight vector W such that $f(x(n)W) = t(n)$, for all n, then for any starting vector w_1, the perceptron learning rule will converge to a weight vector that gives the correct response for all

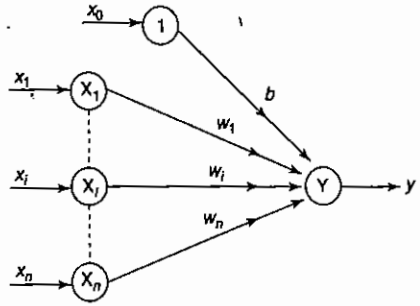

Figure 3-2 Single classification perceptron network.

training patterns, and this learning takes place within a finite number of steps provided that the solution exists."

3.2.3 Architecture

In the original perceptron network, the output obtained from the associator unit is a binary vector, and hence that output can be taken as input signal to the response unit, and classification can be performed. Here only the weights between the associator unit and the output unit can be adjusted, and the weights between the sensory and associator units are fixed. As a result, the discussion of the network is limited to a single portion. Thus, the associator unit behaves like the input unit. A simple perceptron network architecture is shown in Figure 3-2.

In Figure 3-2, there are n input neurons, 1 output neuron and a bias. The input-layer and output-layer neurons are connected through a directed communication link, which is associated with weights. The goal of the perceptron net is to classify the input pattern as a member or not a member to a particular class.

3.2.4 Flowchart for Training Process

The flowchart for the perceptron network training is shown in Figure 3-3. The network has to be suitably trained to obtain the response. The flowchart depicted here presents the flow of the training process.

As depicted in the flowchart, first the basic initialization required for the training process is performed. The entire loop of the training process continues until the training input pair is presented to the network. The training (weight updation) is done on the basis of the comparison between the calculated and desired output. The loop is terminated if there is no change in weight.

3.2.5 Perceptron Training Algorithm for Single Output Classes

The perceptron algorithm can be used for either binary or bipolar input vectors, having bipolar targets, threshold being fixed and variable bias. The algorithm discussed in this section is not particularly sensitive to the initial values of the weights or the value of the learning rate. In the algorithm discussed below, initially the inputs are assigned. Then the net input is calculated. The output of the network is obtained by applying the activation function over the calculated net input. On performing comparison over the calculated and

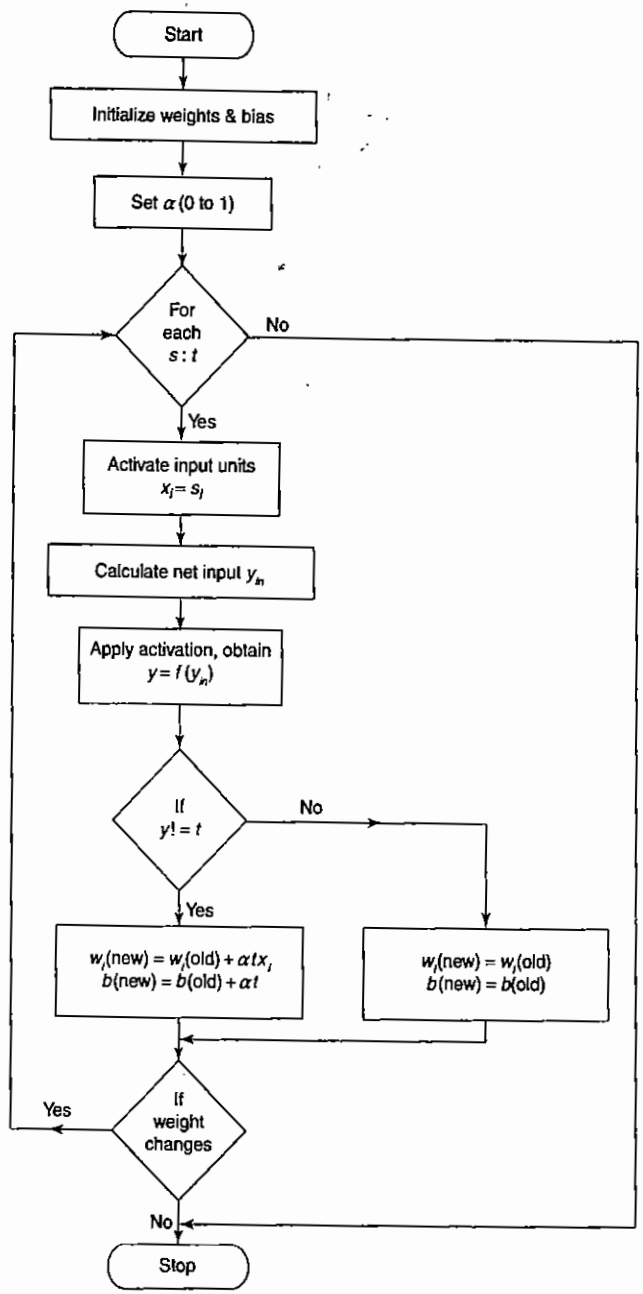

Figure 3-3 Flowchart for perceptron network with single output.

the desired output, the weight updation process is carried out. The entire network is trained based on the mentioned stopping criterion. The algorithm of a perceptron network is as follows:

Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha(0 < \alpha \leq 1)$. For simplicity α is set to 1.

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by *s:t*.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "n" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else, we have

$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

The algorithm discussed above is not sensitive to the initial values of the weights or the value of the learning rate.

3.2.6 Perceptron Training Algorithm for Multiple Output Classes

For multiple output classes, the perceptron training algorithm is as follows:

Step 0: Initialize the weights, biases and learning rate suitably.

Step 1: Check for stopping condition; if it is false, perform Steps 2–6.

Step 2: Perform Steps 3–5 for each bipolar or binary training vector pair *s:t*.

Step 3: Set activation (identity) of each input unit $i = 1$ to n:

$$x_i = s_i$$

Step 4: Calculate output response of each output unit $j = 1$ to m: First, the net input is calculated as

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

Then activations are applied over the net input to calculate the output response:

$$y_j = f(y_{inj}) = \begin{cases} 1 & \text{if } y_{inj} > \theta \\ 0 & \text{if } -\theta \leq y_{inj} \leq \theta \\ -1 & \text{if } y_{inj} < -\theta \end{cases}$$

Step 5: Make adjustment in weights and bias for $j = 1$ to m and $i = 1$ to n.

If $t_j \neq y_j$, then

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$$

else, we have

$$w_{ij}(\text{new}) = w_{ij}(\text{old})$$
$$b_j(\text{new}) = b_j(\text{old})$$

Step 6: Test for the stopping condition, i.e., if there is no change in weights then stop the training process, else start again from Step 2.

It can be noticed that after training, the net classifies each of the training vectors. The above algorithm is suited for the architecture shown in Figure 3-4.

3.2.7 Perceptron Network Testing Algorithm

It is best to test the network performance once the training process is complete. For efficient performance of the network, it should be trained with more data. The testing algorithm (application procedure) is as follows:

Step 0: The initial weights to be used here are taken from the training algorithms (the final weights obtained during training).

Step 1: For each input vector X to be classified, perform Steps 2–3.

Step 2: Set activations of the input unit.

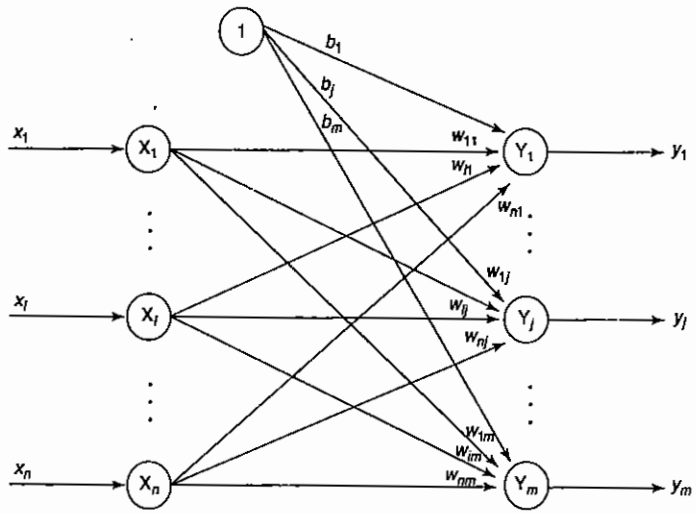

Figure 3-4 Network architecture for perceptron network for several output classes.

Step 3: Obtain the response of output unit.

$$y_{in} = \sum_{i=1}^{n} x_i w_i$$

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Thus, the testing algorithm tests the performance of network.

Note: In the case of perceptron network, it can be used for linear separability concept. Here the separating line may be based on the value of threshold, i.e., the threshold used in activation function must be a non-negative value.

The condition for separating the response from region of positive to region of zero is

$$w_1 x_1 + w_2 x_2 + b > \theta$$

The condition for separating the response from region of zero to region of negative is

$$w_1 x_1 + w_2 x_2 + b < -\theta$$

The conditions above are stated for a single-layer perceptron network with two input neurons and one output neuron and one bias.

3.3 Adaptive Linear Neuron (Adaline)

3.3.1 Theory

The units with linear activation function are called linear units. A network with a single linear unit is called an *Adaline* (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable. The bias in Adaline acts like an adjustable weight, whose connection is from a unit with activations being always 1. Adaline is a net which has only one output unit. The Adaline network may be trained using delta rule. The delta rule may also be called as *least mean square* (LMS) rule or Widrow-Hoff rule. This learning rule is found to minimize the mean-squared error between the activation and the target value.

3.3.2 Delta Rule for Single Output Unit

The Widrow-Hoff rule is very similar to perceptron learning rule. However, their origins are different. The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient-descent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of learning steps, but the gradient-descent approach continues forever, converging only asymptotically to the solution. The delta rule updates the weights between the connections so as to minimize the difference between the net input to the output unit and the target value. The major aim is to minimize the error over all training patterns. This is done by reducing the error for each pattern, one at a time.

The delta rule for adjusting the weight of ith pattern ($i = 1$ to n) is

$$\Delta w_i = \alpha(t - y_{in})x_i$$

where Δw_i is the weight change; α the learning rate; x the vector of activation of input unit; y_{in} the net input to output unit, i.e., $Y = \sum_{i=1}^{n} x_i w_i$; t the target output. The delta rule in case of several output units for adjusting the weight from ith input unit to the jth output unit (for each pattern) is

$$\Delta w_{ij} = \alpha(t_j - y_{inj})x_i$$

3.3.3 Architecture

As already stated, Adaline is a single-unit neuron, which receives input from several units and also from one unit called bias. An Adaline model is shown in Figure 3-5. The basic Adaline model consists of trainable weights. Inputs are either of the two values (+1 or −1) and the weights have signs (positive or negative). Initially, random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or −1. The Adaline model compares the actual output with the target output and on the basis of the training algorithm, the weights are adjusted.

3.3.4 Flowchart for Training Process

The flowchart for the training process is shown in Figure 3-6. This gives a pictorial representation of the network training. The conditions necessary for weight adjustments have to be checked carefully. The weights and other required parameters are initialized. Then the net input is calculated, output is obtained and compared with the desired output for calculation of error. On the basis of the error factor, weights are adjusted.

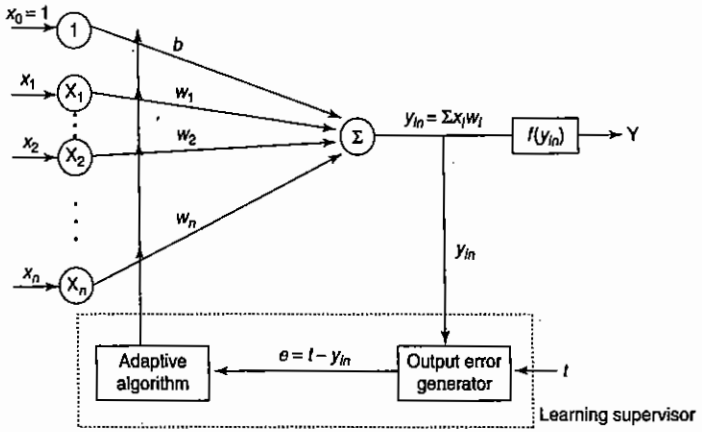

Figure 3-5 Adaline model.

3.3.5 Training Algorithm

The Adaline network training algorithm is as follows:

Step 0: Weights and bias are set to some random values but not zero. Set the learning rate parameter α.

Step 1: Perform Steps 2–6 when stopping condition is false.

Step 2: Perform Steps 3–5 for each bipolar training pair *s:t*.

Step 3: Set activations for input units $i = 1$ to n.

$$x_i = s_i$$

Step 4: Calculate the net input to the output unit.

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

Step 5: Update the weights and bias for $i = 1$ to n:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha (t - y_{in}) x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha (t - y_{in})$$

Step 6: If the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for stopping condition of a network.

The range of learning rate can be between 0.1 and 1.0.

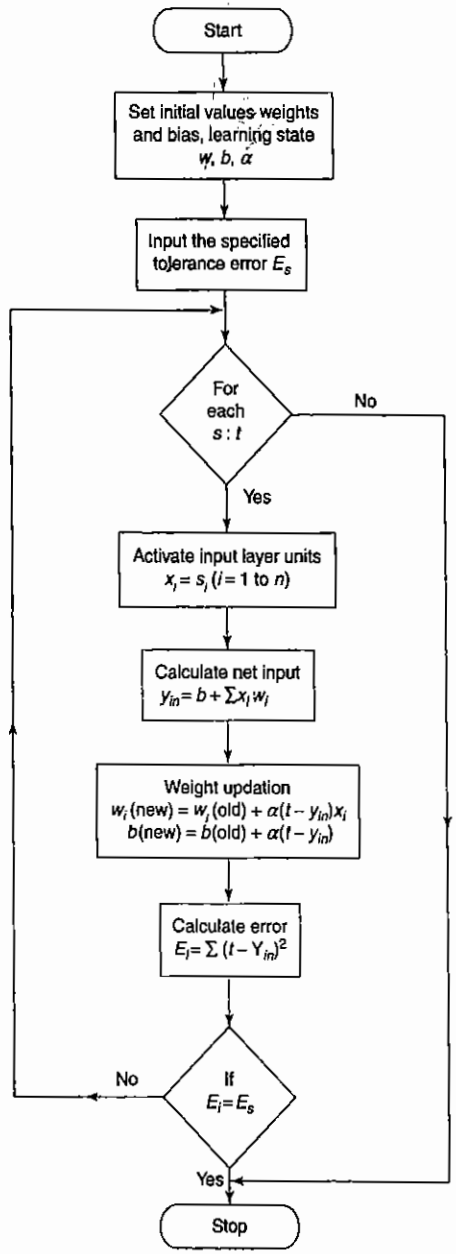

Figure 3-6 Flowchart for Adaline training process.

3.3.6 Testing Algorithm

It is essential to perform the testing of a network that has been trained. When training is completed, the Adaline can be used to classify input patterns. A step function is used to test the performance of the network. The testing procedure for the Adaline network is as follows:

Step 0: Initialize the weights. (The weights are obtained from the training algorithm.)

Step 1: Perform Steps 2–4 for each bipolar input vector x.

Step 2: Set the activations of the input units to x.

Step 3: Calculate the net input to the output unit:

$$y_{in} = b + \sum x_i w_i$$

Step 4: Apply the activation function over the net input calculated:

$$y = \begin{cases} 1 & \text{if } y_{in} \geq 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

3.4 Multiple Adaptive Linear Neurons

3.4.1 Theory

The multiple adaptive linear neurons (Madaline) model consists of many Adalines in parallel with a single output unit whose value is based on certain selection rules. It may use majority vote rule. On using this rule, the output would have as answer either true or false. On the other hand, if AND rule is used, the output is true if and only if both the inputs are true, and so on. The weights that are connected from the Adaline layer to the Madaline layer are fixed, positive and possess equal values. The weights between the input layer and the Adaline layer are adjusted during the training process. The Adaline and Madaline layer neurons have a bias of excitation "1" connected to them. The training process for a Madaline system is similar to that of an Adaline.

3.4.2 Architecture

A simple Madaline architecture is shown in Figure 3-7, which consists of "n" units of input layer, "m" units of Adaline layer and "1" unit of the Madaline layer. Each neuron in the Adaline and Madaline layers has a bias of excitation 1. The Adaline layer is present between the input layer and the Madaline (output) layer; hence, the Adaline layer can be considered a hidden layer. The use of the hidden layer gives the net computational capability which is not found in single-layer nets, but this complicates the training process to some extent.

The Adaline and Madaline models can be applied effectively in communication systems of adaptive equalizers and adaptive noise cancellation and other cancellation circuits.

3.4.3 Flowchart of Training Process

The flowchart of the training process of the Madaline network is shown in Figure 3-8. In case of training, the weights between the input layer and the hidden layer are adjusted, and the weights between the hidden layer

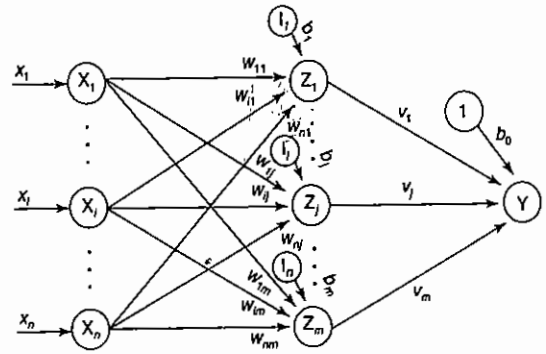

Figure 3-7 Architecture of Madaline layer.

and the output layer are fixed. The time taken for the training process in the Madaline network is very high compared to that of the Adaline network.

3.4.4 Training Algorithm

In this training algorithm, only the weights between the hidden layer and the input layer are adjusted, and the weights for the output units are fixed. The weights $v_1, v_2, \ldots, v_m$ and the bias b_0 that enter into output unit Y are determined so that the response of unit Y is 1. Thus, the weights entering Y unit may be taken as

$$v_1 = v_2 = \cdots = v_m = \tfrac{1}{2}$$

and the bias can be taken as

$$b_0 = \tfrac{1}{2}$$

The activation for the Adaline (hidden) and Madaline (output) units is given by

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Step 0: Initialize the weights. The weights entering the output unit are set as above. Set initial small random values for Adaline weights. Also set initial learning rate α.

Step 1: When stopping condition is false, perform Steps 2–3.

Step 2: For each bipolar training pair $s:t$, perform Steps 3–7.

Step 3: Activate input layer units. For $i = 1$ to n,

$$x_i = s_i$$

Step 4: Calculate net input to each hidden Adaline unit:

$$z_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

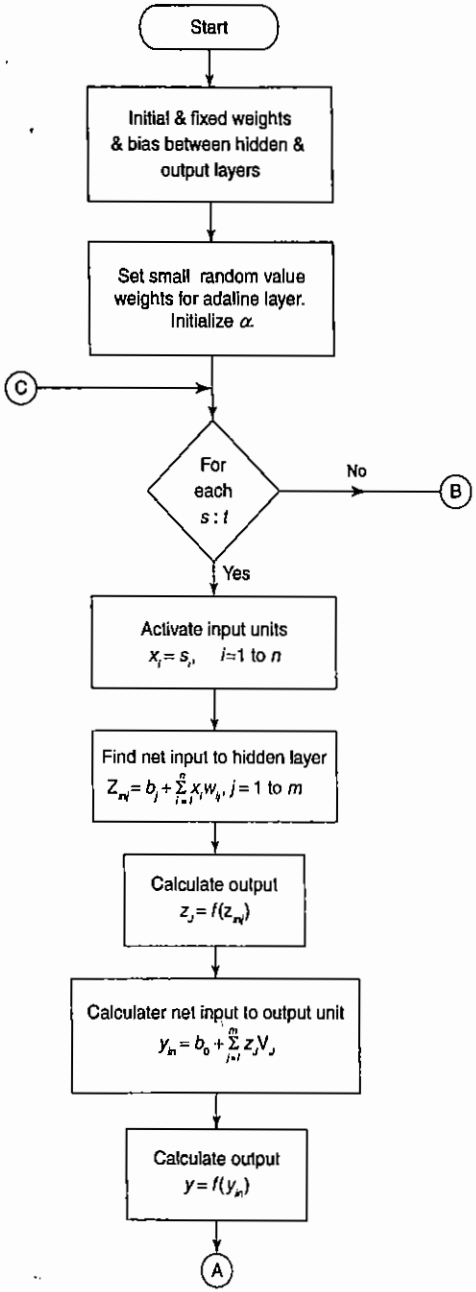

Figure 3-8 Flowchart for training of Madaline.

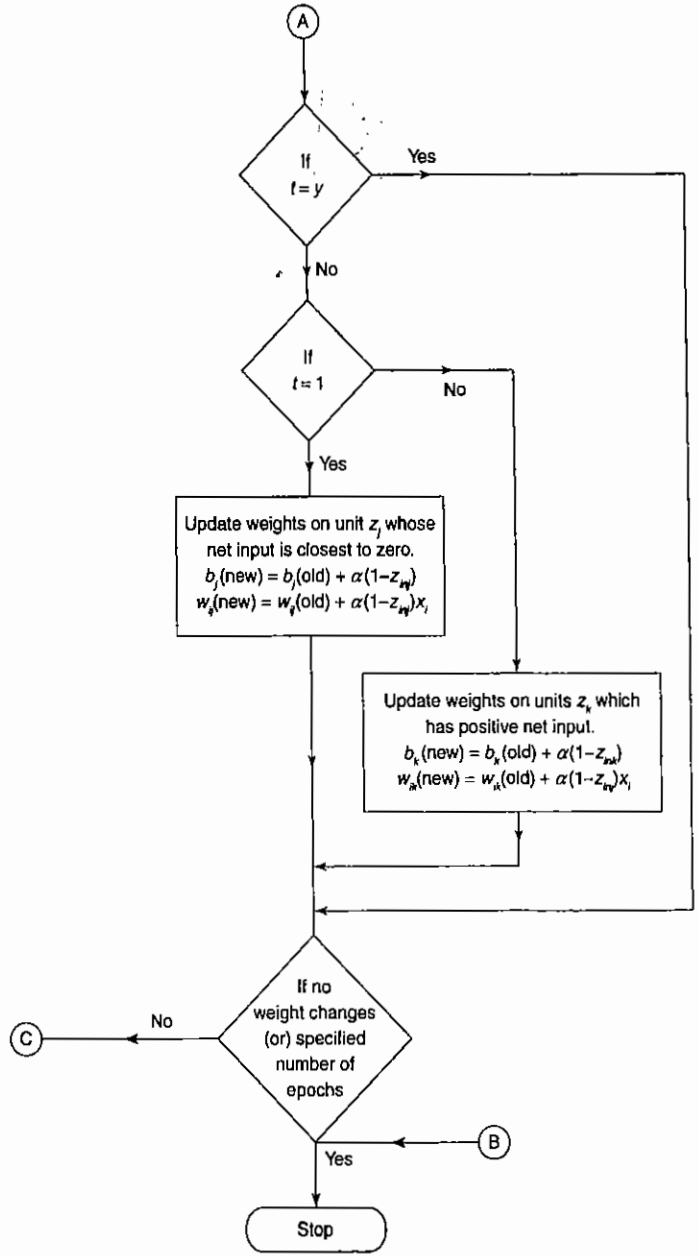

Figure 3-8 *(Continued).*

Step 5: Calculate output of each hidden unit:

$$z_j = f(z_{inj})$$

Step 6: Find the output of the net:

$$y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$$

$$y = f(y_{in})$$

Step 7: Calculate the error and update the weights.

1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha(1 - z_{inj})$$
$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(1 - z_{inj})x_i$$

3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z_{ink})x_i$$
$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z_{ink})$$

Step 8: Test for the stopping condition. (If there is no weight change or weight reaches a satisfactory level, or if a specified maximum number of iterations of weight updation have been performed then stop, or else continue).

Madalines can be formed with the weights on the output unit set to perform some logic functions. If there are only two hidden units present, or if there are more than two hidden units, then the "majority vote rule" function may be used.

3.5 Back-Propagation Network

3.5.1 Theory

The back-propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986). This network has re-awakened the scientific and engineering community to the modeling and processing of numerous quantitative phenomena using neural networks. This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called *back-propagation networks* (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units. This is a method where the error is propagated back to the hidden unit. The aim of the neural network is to train the net to achieve a balance between the net's ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization).

The back-propagation algorithm is different from other networks in respect to the process by which the weights are calculated during the learning period of the network. The general difficulty with the multilayer perceptrons is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error. When the hidden layers are increased the network training becomes more complex. To update weights, the error must be calculated. The error, which is the difference between the actual (calculated) and the desired (target) output, is easily measured at the output layer. It should be noted that at the hidden layers, there is no direct information of the error. Therefore, other techniques should be used to calculate an error at the hidden layer, which will cause minimization of the output error, and this is the ultimate goal.

The training of the BPN is done in three stages – the feed-forward of the input training pattern, the calculation and back-propagation of the error, and updation of weights. The testing of the BPN involves the computation of feed-forward phase only. There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient. Even though the training is very slow, once the network is trained it can produce its outputs very rapidly.

3.5.2 Architecture

A back-propagation neural network is a multilayer, feed-forward neural network consisting of an input layer, a hidden layer and an output layer. The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights. Figure 3-9 shows the architecture of a BPN, depicting only the direction of information flow for the feed-forward phase. During the back-propagation phase of learning, signals are sent in the reverse direction.

The inputs are sent to the BPN and the output obtained from the net could be either binary (0, 1) or bipolar (−1, +1). The activation function could be any function which increases monotonically and is also differentiable.

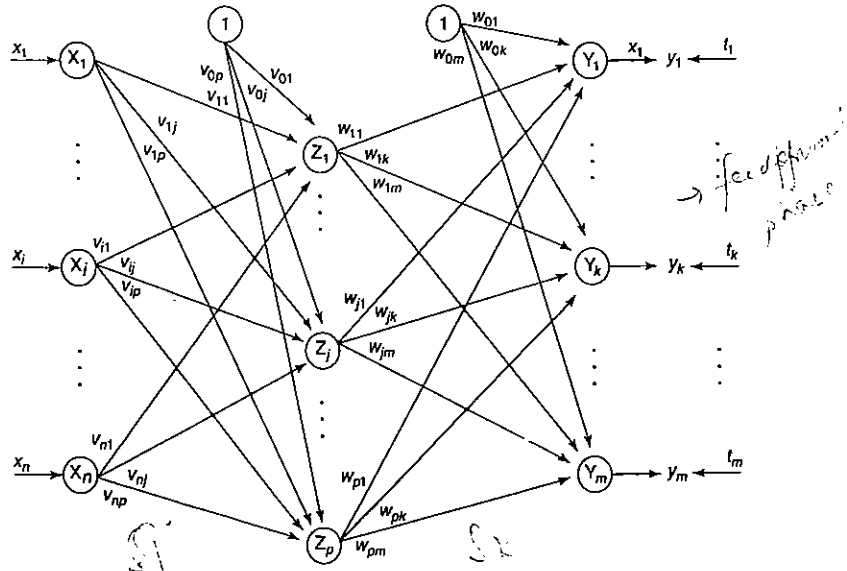

Figure 3-9 Architecture of a back-propagation network.

3.5.3 Flowchart for Training Process

The flowchart for the training process using a BPN is shown in Figure 3-10. The terminologies used in the flowchart and in the training algorithm are as follows:

x = input training vector $(x_1, \ldots, x_i, \ldots, x_n)$
t = target output vector $(t_1, \ldots, t_k, \ldots, t_m)$
α = learning rate parameter
x_i = input unit i. (Since the input layer uses identity activation function, the input and output signals here are same.)
v_{0j} = bias on jth hidden unit
w_{0k} = bias on kth output unit
z_j = hidden unit j. The net input to z_j is

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is

$$z_j = f(z_{inj})$$

y_k = output unit k. The net input to y_k is

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and the output is

$$y_k = f(y_{ink})$$

δ_k = error correction weight adjustment for w_{jk} that is due to an error at output unit y_k, which is back-propagated to the hidden units that feed into unit y_k
δ_j = error correction weight adjustment for v_{ij} that is due to the back-propagation of error to the hidden unit z_j.

Also, it should be noted that the commonly used activation functions are binary sigmoidal and bipolar sigmoidal activation functions (discussed in Section 2.3.3). These functions are used in the BPN because of the following characteristics: (i) continuity; (ii) differentiability; (iii) nondecreasing monotony.

The range of binary sigmoid is from 0 to 1, and for bipolar sigmoid it is from -1 to $+1$.

3.5.4 Training Algorithm

The error back-propagation learning algorithm can be outlined in the following algorithm:

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2–9 when stopping condition is false.

Step 2: Perform Steps 3–8 for each training pair.

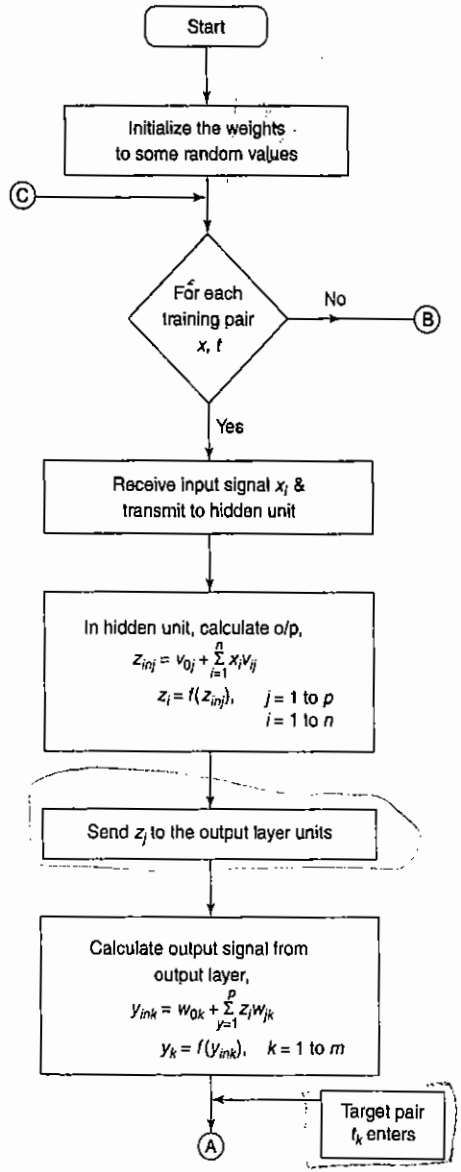

Figure 3-10 Flowchart for back-propagation network training.

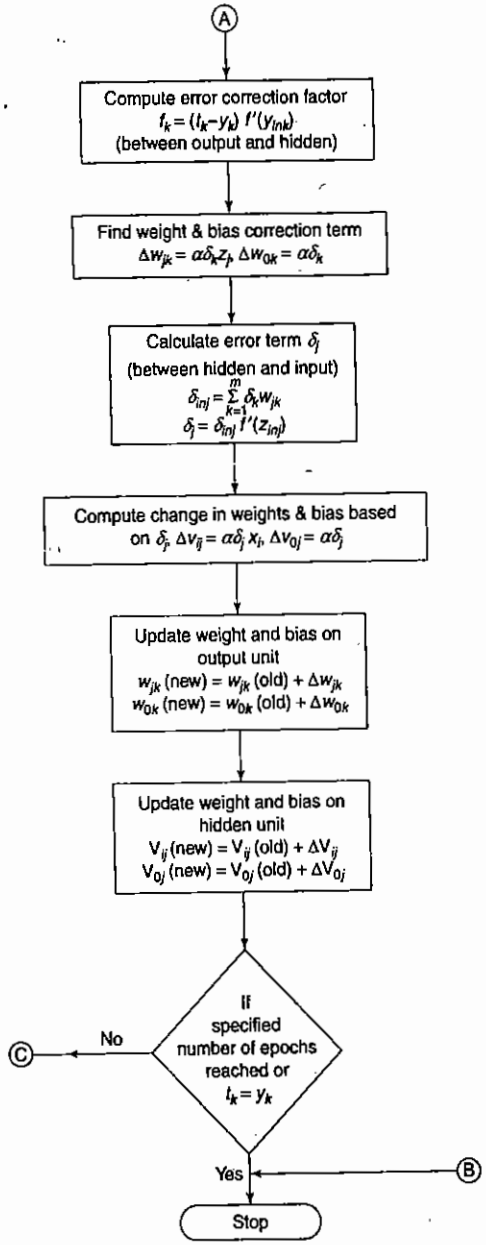

Figure 3-10 (*Continued*).

Feed-forward phase (Phase I)

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit $z_j (j = 1 \text{ to } p)$ sums its weighted input signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units.

Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

Back-propagation of error (Phase II):

Step 6: Each output unit $y_k (k = 1 \text{ to } m)$ receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit ($z_j, j = 1 \text{ to } p$) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$

Weight and bias updation (Phase III):

Step 8: Each output unit (y_k, $k = 1$ to m) updates the bias and weights:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

Each hidden unit (z_j, $j = 1$ to p) updates its bias and weights:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$$
$$v_{0j}(\text{new}) = v_{0j}(\text{old}) + \Delta v_{0j}$$

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

The above algorithm uses the incremental approach for updation of weights, i.e., the weights are being changed immediately after a training pattern is presented. There is another way of training called *batch-mode training*, where the weights are changed only after all the training patterns are presented. The effectiveness of two approaches depends on the problem, but batch-mode training requires additional local storage for each connection to maintain the immediate weight changes. When a BPN is used as a classifier, it is equivalent to the optimal Bayesian discriminant function for asymptotically large sets of statistically independent training patterns.

The problem in this case is whether the back-propagation learning algorithm can always converge and find proper weights for network even after enough learning. It will converge since it implements a gradient-descent on the error surface in the weight space, and this will roll down the error surface to the nearest minimum error and will stop. This becomes true only when the relation existing between the input and the output training patterns is deterministic and the error surface is deterministic. This is not the case in real world because the produced square-error surfaces are always at random. This is the stochastic nature of the back-propagation algorithm, which is purely based on the stochastic gradient-descent method. The BPN is a special case of stochastic approximation.

If the BPN algorithm converges at all, then it may get stuck with local minima and may be unable to find satisfactory solutions. The randomness of the algorithm helps it to get out of local minima. The error functions may have large number of global minima because of permutations of weights that keep the network input–output function unchanged. This causes the error surfaces to have numerous troughs.

3.5.5 Learning Factors of Back-Propagation Network

The training of a BPN is based on the choice of various parameters. Also, the convergence of the BPN is based on some important learning factors such as the initial weights, the learning rate, the updation rule, the size and nature of the training set, and the architecture (number of layers and number of neurons per layer).

3.5.5.1 Initial Weights

The ultimate solution may be affected by the initial weights of a multilayer feed-forward network. They are initialized at small random values. The choice of the initial weight determines how fast the network converges. The initial weights cannot be very high because the sigmoidal activation functions used here may get saturated

from the beginning itself and the system may be stuck at a local minima or at a very flat plateau at the starting point itself. One method of choosing the weight w_{ij} is choosing it in the range

$$w_{ij} = \left[\frac{-3}{\sqrt{o_i}}, \frac{3}{\sqrt{o_i}}\right]$$

where o_i is the number of processing elements j that feed-forward to processing element i. The initialization can also be done by a method called Nyugen–Widrow initialization. This type of initialization leads to faster convergence of network. The concept here is based on the geometric analysis of the response of hidden neurons to a single input. The method is used for improving the learning ability of the hidden units. The random initialization of weights connecting input neurons to the hidden neurons is obtained by the equation

$$v_{ij}(\text{new}) = \gamma \frac{v_{ij}(\text{old})}{\left\| v_j(\text{old}) \right\|}$$

where $\overline{v_j}$ is the average weight calculated for all values of i, and the scale factor $\gamma = 0.7(P)^{1/n}$ ("n" is the number of input neurons and "P" is the number of hidden neurons).

3.5.5.2 Learning Rate α

The learning rate (α) affects the convergence of the BPN. A larger value of α may speed up the convergence but might result in overshooting, while a smaller value of α has vice-versa effect. The range of α from 10^{-3} to 10 has been used successfully for several back-propagation algorithmic experiments. Thus, a large learning rate leads to rapid learning but there is oscillation of weights, while the lower learning rate leads to slower learning.

3.5.5.3 Momentum Factor

The gradient descent is very slow if the learning rate α is small and oscillates widely if α is too large. One very efficient and commonly used method that allows a larger learning rate without oscillations is by adding a momentum factor to the normal gradient descent method.

The momentum factor is denoted by $\eta \in [0, 1]$ and the value of 0.9 is often used for the momentum factor. Also, this approach is more useful when some training data are very different from the majority of data. A momentum factor can be used with either pattern by pattern updating or batch-mode updating. In case of batch mode, it has the effect of complete averaging over the patterns. Even though the averaging is only partial in the pattern-by-pattern mode, it leaves some useful information for weight updation.

The weight updation formulas used here are

$$w_{jk}(t+1) = w_{jk}(t) + \underbrace{\alpha \delta_k z_j + \eta \left[w_{jk}(t) - w_{jk}(t-1) \right]}_{\Delta w_{jk}(t+1)}$$

and

$$v_{ij}(t+1) = v_{ij}(t) + \underbrace{\alpha \delta_j x_i + \eta \left[v_{ij}(t) - v_{ij}(t-1) \right]}_{\Delta v_{ij}(t+1)}$$

The momentum factor also helps in faster convergence.

3.5.5.4 Generalization

The best network for generalization is BPN. A network is said to be generalized when it sensibly interpolates with input networks that are new to the network. When there are many trainable parameters for the given amount of training data, the network learns well but does not generalize well. This is usually called *overfitting* or *overtraining*. One solution to this problem is to monitor the error on the test set and terminate the training when the error increases. With small number of trainable parameters, the network fails to learn the training data and performs very poorly on the test data. For improving the ability of the network to generalize from a training data set to a test data set, it is desirable to make small changes in the input space of a pattern, without changing the output components. This is achieved by introducing variations in the input space of training patterns as part of the training set. However, computationally, this method is very expensive. Also, a net with large number of nodes is capable of memorizing the training set at the cost of generalization. As a result, smaller nets are preferred than larger ones.

3.5.5.5 Number of Training Data

The training data should be sufficient and proper. There exists a rule of thumb, which states that the training data should cover the entire expected input space, and while training, training-vector pairs should be selected randomly from the set. Assume that the input space as being linearly separable into "L" disjoint regions with their boundaries being part of hyper planes. Let "T" be the lower bound on the number of training patterns. Then, choosing T such that $T/L \gg 1$ will allow the network to discriminate pattern classes using fine piecewise hyperplane partitioning. Also in some cases, scaling or normalization has to be done to help learning.

3.5.5.6 Number of Hidden Layer Nodes

If there exists more than one hidden layer in a BPN, then the calculations performed for a single layer are repeated for all the layers and are summed up at the end. In case of all multilayer feed-forward networks, the size of a hidden layer is very important. The number of hidden units required for an application needs to be determined separately. The size of a hidden layer is usually determined experimentally. For a network of a reasonable size, the size of hidden nodes has to be only a relatively small fraction of the input layer. For example, if the network does not converge to a solution, it may need more hidden nodes. On the other hand, if the network converges, the user may try a very few hidden nodes and then settle finally on a size based on overall system performance.

3.5.6 Testing Algorithm of Back-Propagation Network

The testing procedure of the BPN is as follows:

Step 0: Initialize the weights. The weights are taken from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector.

Step 2: Set the activation of input unit for x_i ($i = 1$ to n).

Step 3: Calculate the net input to hidden unit x and its output. For $j = 1$ to p,

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

$$z_i = f(z_{inj})$$

Step 4: Now compute the output of the output layer unit. For $k = 1$ to m,

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

$$y_k = f(y_{ink})$$

Use sigmoidal activation functions for calculating the output.

3.6 Radial Basis Function Network

3.6.1 Theory

The radial basis function (RBF) is a classification and functional approximation neural network developed by M.J.D. Powell. The network uses the most common nonlinearities such as sigmoidal and Gaussian kernel functions. The Gaussian functions are also used in regularization networks. The response of such a function is positive for all values of y; the response decreases to 0 as $|y| \to 0$. The Gaussian function is generally defined as

$$f(y) = e^{-y^2}$$

The derivative of this function is given by

$$f'(y) = -2ye^{-y^2} = -2yf(y)$$

The graphical representation of this Gaussian function is shown in Figure 3-11 below.

When the Gaussian potential functions are being used, each node is found to produce an identical output for inputs existing within the fixed radial distance from the center of the kernel, they are found to be radically symmetric, and hence the name radial basis function network. The entire network forms a linear combination of the nonlinear basis function.

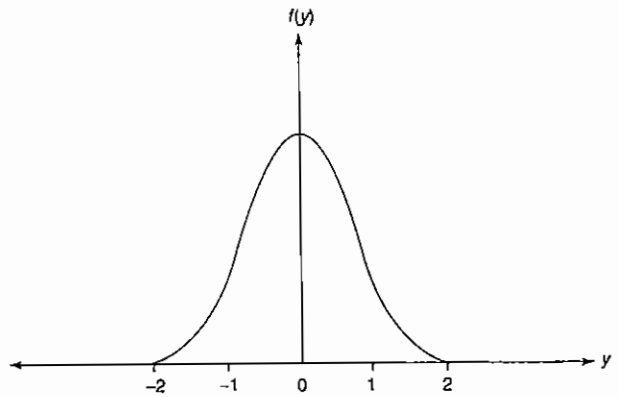

Figure 3-11 Gaussian kernel function.

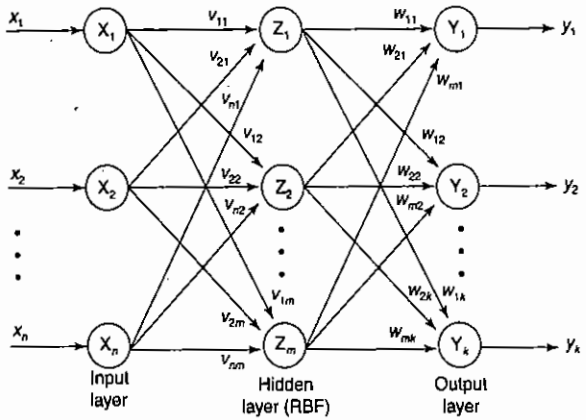

Figure 3-12 Architecture of RBF.

3.6.2 Architecture

The architecture for the radial basis function network (RBFN) is shown in Figure 3-12. The architecture consists of two layers whose output nodes form a linear combination of the kernel (or basis) functions computed by means of the RBF nodes or hidden layer nodes. The basis function (nonlinearity) in the hidden layer produces a significant nonzero response to the input stimulus it has received only when the input of it falls within a small localized region of the input space. This network can also be called as *localized receptive field network*.

3.6.3 Flowchart for Training Process

The flowchart for the training process of the RBF is shown in Figure 3-13 below. In this case, the center of the RBF functions has to be chosen and hence, based on all parameters, the output of network is calculated.

3.6.4 Training Algorithm

The training algorithm describes in detail all the calculations involved in the training process depicted in the flowchart. The training is started in the hidden layer with an unsupervised learning algorithm. The training is continued in the output layer with a supervised learning algorithm. Simultaneously, we can apply supervised learning algorithm to the hidden and output layers for fine-tuning of the network. The training algorithm is given as follows.

Step 0: Set the weights to small random values.

Step 1: Perform Steps 2–8 when the stopping condition is false.

Step 2: Perform Steps 3–7 for each input.

Step 3: Each input unit (x_i for all $i = 1$ to n) receives input signals and transmits to the next hidden layer unit.

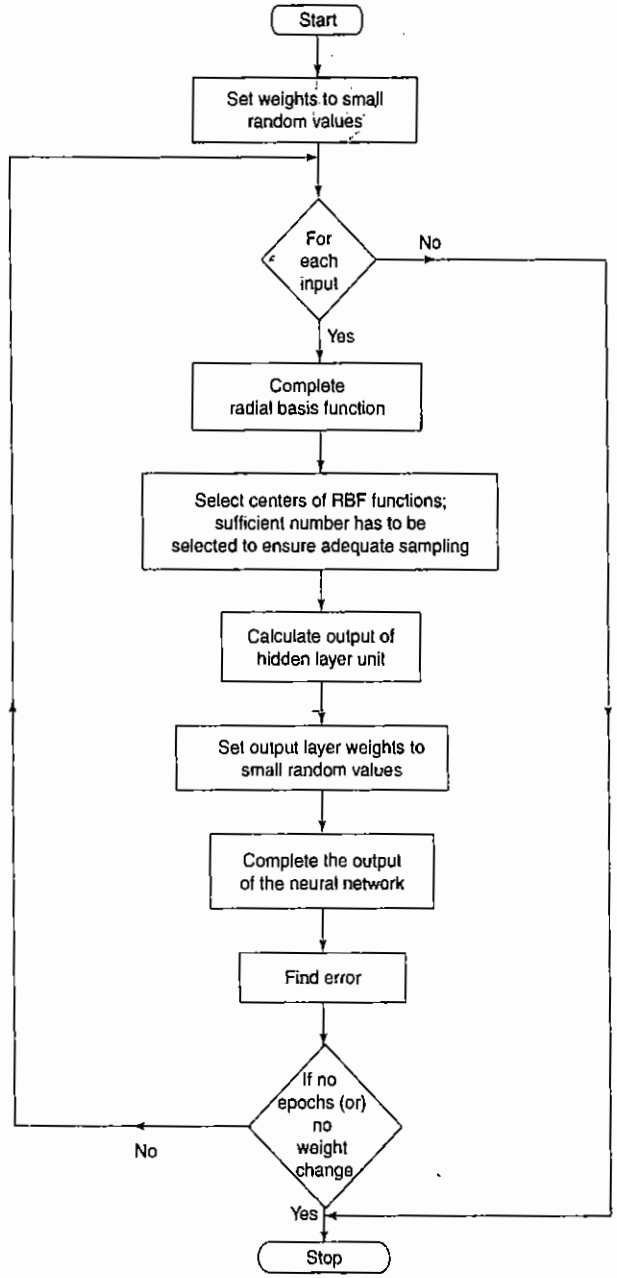

Figure 3-13 Flowchart for the training process of RBF.

Step 4: Calculate the radial basis function.

Step 5: Select the centers for the radial basis function. The centers are selected from the set of input vectors. It should be noted that a sufficient number of centers have to be selected to ensure adequate sampling of the input vector space.

Step 6: Calculate the output from the hidden layer unit:

$$v_i(x_i) = \frac{\exp\left[-\sum_{j=1}^{r}(x_{ji} - \hat{x}_{ji})^2\right]}{\sigma_i^2}$$

where $\hat{x}_{ji}$ is the center of the RBF unit for input variables; σ_i the width of ith RBF unit; x_{ji} the jth variable of input pattern.

Step 7: Calculate the output of the neural network:

$$y_{net} = \sum_{i=1}^{k} w_{im} v_i(x_i) + w_0$$

where k is the number of hidden layer nodes (RBF function); y_{net} the output value of mth node in output layer for the nth incoming pattern; w_{im} the weight between ith RBF unit and mth output node; w_0 the biasing term at nth output node.

Step 8: Calculate the error and test for the stopping condition. The stopping condition may be number of epochs or to a certain extent weight change.

Thus, a network can be trained using RBFN.

3.7 Time Delay Neural Network

The neural network has to respond to a sequence of patterns. Here the network is required to produce a particular output sequence in response to a particular sequence of inputs. A shift register can be considered as a tapped delay line. Consider a case of a multilayer perceptron where the tapped outputs of the delay line are applied to its inputs. This type of network constitutes a *time delay neural network* (TDNN). The output consists of a finite temporal dependence on its inputs, given as

$$U(t) = F[x(t), x(t-1), \ldots, x(t-n)]$$

where F is any nonlinearity function. The multilayer perceptron with delay line is shown in Figure 3-14.

When the function $U(t)$ is a weighted sum, then the TDNN is equivalent to a finite impulse response filter (FIR). In TDNN, when the output is being fed back through a unit delay into the input layer, then the net computed here is equivalent to an infinite impulse response (IIR) filter. Figure 3-15 shows TDNN with output feedback.

Thus, a neuron with a tapped delay line is called a TDNN unit, and a network which consists of TDNN units is called a TDNN. A specific application of TDNNs is speech recognition. The TDNN can be trained using the back-propagation-learning rule with a momentum factor.

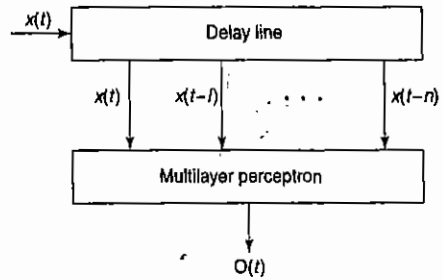

Figure 3-14 Time delay neural network (FIR filter).

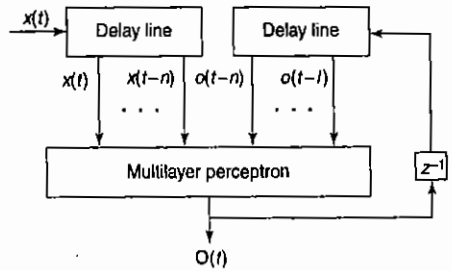

Figure 3-15 TDNN with output feedback (IIR filter).

3.8 Functional Link Networks

These networks are specifically designed for handling linearly non-separable problems using appropriate input representation. Thus, suitable enhanced representation of the input data has to be found out. This can be achieved by increasing the dimensions of the input space. The input data which is expanded is used for training instead of the actual input data. In this case, higher order input terms are chosen so that they are linearly independent of the original pattern components. Thus, the input representation has been enhanced and linear separability can be achieved in the extended space. One of the functional link model networks is shown in Figure 3-16. This model is helpful for learning continuous functions. For this model, the higher-order input terms are obtained using the orthogonal basis functions such as $\sin \pi x$, $\cos \pi x$, $\sin 2\pi x$, $\cos 2\pi x$, etc.

The most common example of linear nonseparability is XOR problem. The functional link networks help in solving this problem. The inputs now are

x_1	x_2	x_1x_2	t
−1	−1	1	1
−1	1	−1	−1
1	−1	−1	−1
1	1	1	1

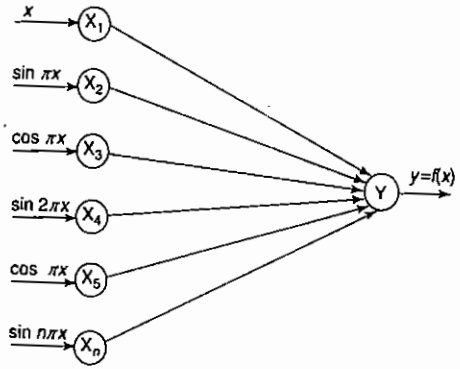

Figure 3-16 Functional line network model.

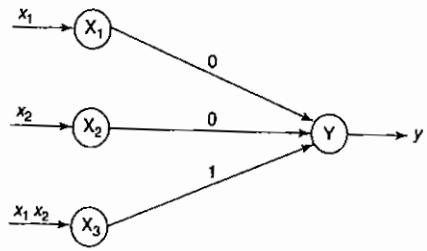

Figure 3-17 The XOR problem.

Thus, it can be easily seen that the functional link network in Figure 3-17 is used for solving this problem. The functional link network consists of only one layer, therefore, it can be trained using delta learning rule instead of the generalized delta learning rule used in BPN. As a result, the learning speed of the functional link network is faster than that of the BPN.

3.9 Tree Neural Networks

The tree neural networks (TNNs) are used for the pattern recognition problem. The main concept of this network is to use a small multilayer neural network at each decision-making node of a binary classification tree for extracting the non-linear features. TNNs completely extract the power of tree classifiers for using appropriate local features at the different levels and nodes of the tree. A binary classification tree is shown in Figure 3-18.

The decision nodes are present as circular nodes and the terminal nodes are present as square nodes. The terminal node has class label denoted by $\hat{c}$ associated with it. The rule base is formed in the decision node (splitting rule in the form of $f(x) < \theta$). The rule determines whether the pattern moves to the right or to the left. Here, $f(x)$ indicates the associated feature of pattern and "θ" is the threshold. The pattern will be given the class label of the terminal node on which it has landed. The classification here is based on the fact that the appropriate features can be selected at different nodes and levels in the tree. The output feature $y = f(x)$

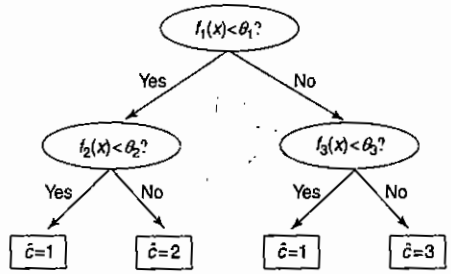

Figure 3-18 Binary classification tree.

obtained by a multilayer network at a particular decision node is used in the following way:

x directed to left child node t_L, if $y < 0$

x directed to right child node t_R, if $y \geq 0$

The algorithm for a TNN consists of two phases:

1. *Tree growing phase*: In this phase, a large tree is grown by recursively finding the rules for splitting until all the terminal nodes have pure or nearly pure class membership, else it cannot split further.
2. *Tree pruning phase*: Here a smaller tree is being selected from the pruned subtree to avoid the overfilling of data.

The training of TNN involves two nested optimization problems. In the inner optimization problem, the BPN algorithm can be used to train the network for a given pair of classes. On the other hand, in outer optimization problem, a heuristic search method is used to find a good pair of classes. The TNN when tested on a character recognition problem decreases the error rate and size of the tree relative to that of the standard classification tree design methods. The TNN can be implemented for waveform recognition problem. It obtains comparable error rates and the training here is faster than the large BPN for the same application. Also, TNN provides a structured approach to neural network classifier design problems.

3.10 Wavelet Neural Networks

The wavelet neural network (WNN) is based on the wavelet transform theory. This network helps in approximating arbitrary nonlinear functions. The powerful tool for function approximation is wavelet decomposition.

Let $f(x)$ be a piecewise continuous function. This function can be decomposed into a family of functions, which is obtained by dilating and translating a single wavelet function $\phi : R^n \rightarrow R$ as

$$f(x) = \sum_{i=1}^{n} w_i \det\left[D_i^{1/2}\right] \phi\left[D_i(x - t_i)\right]$$

where D_i is the diag(d_i), $d_i \in R_n^+$ are dilation vectors; D_i and t_i are the translational vectors; det [ ] is the determinant operator. The wavelet function ϕ selected should satisfy some properties. For selecting $\phi : R^n \rightarrow R$, the condition may be

$$\phi(x) = \phi_1(x_1) \cdots \phi_1(x_n) \quad \text{for } x = (x_1, x_2, \ldots, x_n)$$

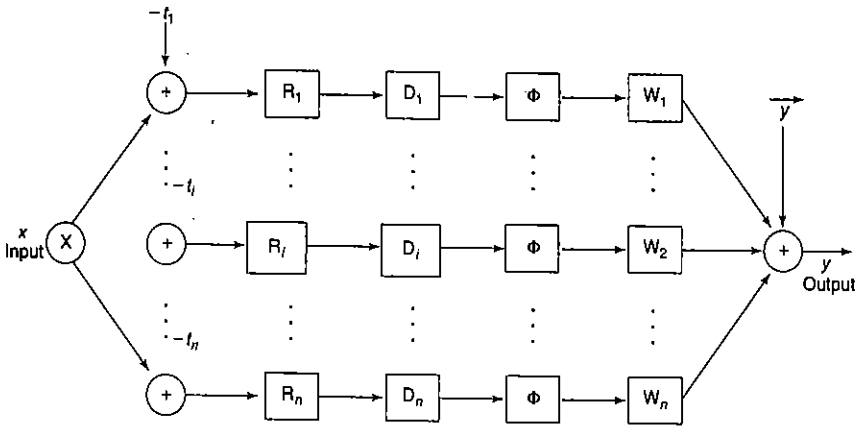

Figure 3-19 Wavelet neural network.

where

$$\phi_t(x) = -x\exp\left(\frac{-x^2}{2}\right)$$

is called scalar wavelet. The network structure can be formed based on the wavelet decomposition as

$$y(x) = \sum_{i=1}^{n} w_i \phi\,[D_i(x - t_i)] + \bar{y}$$

where $\bar{y}$ helps to deal with nonzero mean functions on finite domains. For proper dilation, a rotation can be made for better network operation:

$$y(x) = \sum_{i=1}^{n} w_i \phi\,[D_i R_i(x - t_i)] + \bar{y}$$

where R_i are the rotation matrices. The network which performs according to the above equation is called wavelet neural network. This is a combination of translation, rotation and dilation; and if a wavelet is lying on the same line, then it is called *wavelon* in comparison to the neurons in neural networks. The wavelet neural network is shown in Figure 3-19.

3.11 Summary

In this chapter we have discussed the supervised learning networks. In most of the classification and recognition problems, the widely used networks are the supervised learning networks. The architecture, the learning rule, flowchart for training process and training algorithm are discussed in detail for perceptron network, Adaline, Madaline, back-propagation network and radial basis function network. The perceptron network can be trained for single output classes as well as multioutput classes. Also, many Adaline networks combine together

to form a Madaline network. These networks are trained using delta learning rule. Back-propagation network is the most commonly used network in the real time applications. The error is back-propagated here and is fine tuned for achieving better performance. The basic difference between the back-propagation network and radial basis function network is the activation function used. The radial basis function network mostly uses Gaussian activation function. Apart from these networks, some special supervised learning networks such as time delay neural networks, functional link networks, tree neural networks and wavelet neural networks have also been discussed.

3.12 Solved Problems

1. Implement AND function using perceptron networks for bipolar inputs and targets.

Solution: Table 1 shows the truth table for AND function with bipolar inputs and targets:

Table 1

x_1	x_2	t
1	1	1
1	−1	−1
−1	1	−1
−1	−1	−1

The perceptron network, which uses perceptron learning rule, is used to train the AND function. The network architecture is as shown in Figure 1. The input patterns are presented to the network one by one. When all the four input patterns are presented, then one epoch is said to be completed. The initial weights and threshold are set to zero, i.e., $w_1 = w_2 = b = 0$ and $\theta = 0$. The learning rate α is set equal to 1.

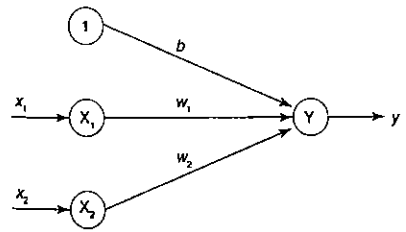

Figure 1 Perceptron network for AND function.

For the first input pattern, $x_1 = 1, x_2 = 1$ and $t = 1$, with weights and bias, $w_1 = 0$, $w_2 = 0$ and $b = 0$:

- Calculate the net input

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0 + 1 \times 0 + 1 \times 0 = 0$$

- The output y is computed by applying activations over the net input calculated:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

Here we have taken $\theta = 0$. Hence, when, $y_{in} = 0$, $y = 0$.

- Check whether $t = y$. Here, $t = 1$ and $y = 0$, so $t \neq y$, hence weight updation takes place:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times 1 \times 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times 1 \times 1 = 1$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times 1 = 1$$

Here, the change in weights are

$$\Delta w_1 = \alpha t x_1;$$
$$\Delta w_2 = \alpha t x_2;$$
$$\Delta b = \alpha t$$

The weights $w_1 = 1$, $w_2 = 1$, $b = 1$ are the final weights after first input pattern is presented. The same process is repeated for all the input patterns. The process can be stopped when all the targets become equal to the calculated output or when a separating line is obtained using the final weights for separating the positive responses from negative responses. Table 2 shows the training of perceptron network until its

Table 2

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	−1	1	−1	1	1	−1	1	−1	0	2	0
−1	1	1	−1	2	1	+1	−1	−1	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1
EPOCH-2											
1	1	1	1	1	1	0	0	0	1	1	−1
1	−1	1	−1	−1	−1	0	0	0	1	1	−1
−1	1	1	−1	−1	−1	0	0	0	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1

target and calculated output converge for all the patterns.

The final weights and bias after second epoch are

$$w_1 = 1, w_2 = 1, \ b = -1$$

Since the threshold for the problem is zero, the equation of the separating line is

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

Here

$$w_1x_1 + w_2x_2 + b > \theta$$
$$w_1x_1 + w_2x_2 + b > 0$$

Thus, using the final weights we obtain

$$x_2 = -\frac{1}{1}x_1 - \frac{(-1)}{1}$$
$$x_2 = -x_1 + 1$$

It can be easily found that the above straight line separates the positive response and negative response region, as shown in Figure 2.

The same methodology can be applied for implementing other logic functions such as OR, AND-NOT, NAND, etc. If there exists a threshold value $\theta \neq 0$, then two separating lines have to be obtained, i.e., one to separate positive response from zero and the other for separating zero from the negative response.

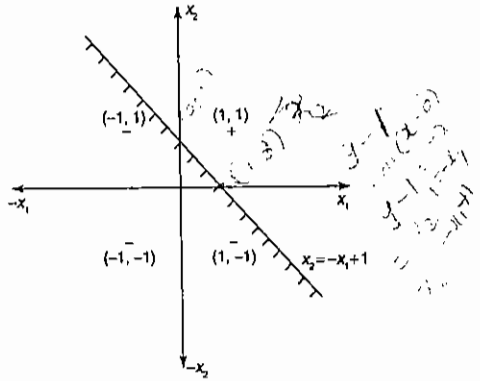

Figure 2 Decision boundary for AND function in perceptron training ($\theta = 0$).

2. Implement OR function with binary inputs and bipolar targets using perceptron training algorithm upto 3 epochs.

Solution: The truth table for OR function with binary inputs and bipolar targets is shown in Table 3.

Table 3

x_1	x_2	t
1	1	1
1	0	1
0	1	1
0	0	−1

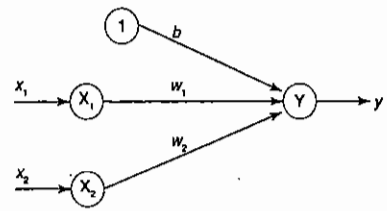

Figure 3 Perceptron network for OR function.

The perceptron network, which uses perceptron learning rule, is used to train the OR function. The network architecture is shown in Figure 3. The initial values of the weights and bias are taken as zero, i.e.,

$$w_1 = w_2 = b = 0$$

Also the learning rate is 1 and threshold is 0.2. So, the activation function becomes

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \end{cases}$$

The network is trained as per the perceptron training algorithm and the steps are as in problem 1 (given for first pattern). Table 4 gives the network training for 3 epochs.

The final weights at the end of third epoch are

$$w_1 = 2, w_2 = 1, b = -1$$

Further epochs have to be done for the convergence of the network.

3. Find the weights using perceptron network for ANDNOT function when all the inputs are presented only one time. Use bipolar inputs and targets.

Solution: The truth table for ANDNOT function is shown in Table 5.

Table 5

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

The network architecture of ANDNOT function is shown as in Figure 4. Let the initial weights be zero and $\alpha = 1, \theta = 0$. For the first input sample, we compute the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$
$$= 0 + 1 \times 0 + 1 \times 0 = 0$$

Table 4

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	0	1	1	2	1	0	0	0	1	1	1
0	1	1	1	2	1	0	0	0	1	1	0
0	0	1	−1	1	1	0	0	−1	1	1	0
EPOCH-2											
1	1	1	1	2	1	0	0	0	1	1	0
1	0	1	1	1	1	0	0	0	1	1	0
0	1	1	1	1	1	0	0	0	1	1	0
0	0	1	−1	0	0	0	0	0	1	1	−1
EPOCH-3											
1	1	1	1	1	1	0	0	0	1	1	−1
1	0	1	1	0	0	1	0	1	2	1	0
0	1	1	1	1	1	0	0	0	2	1	0
0	0	1	−1	0	0	0	0	−1	2	1	−1

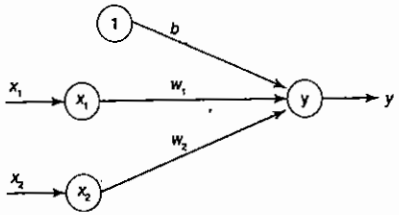

Figure 4 Network for ANDNOT function.

Applying the activation function over the net input, we obtain

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } -0 \le y_{in} \le 0 \\ -1 & \text{if } y_{in} < -0 \end{cases}$$

Hence, the output $y = f(y_{in}) = 0$. Since $t \neq y$, the new weights are computed as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times -1 \times 1 = -1$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times -1 \times 1 = -1$$

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times -1 = -1$$

The weights after presenting the first sample are

$$w = [-1 \ -1 \ -1]$$

For the second input sample, we calculate the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= -1 + 1 \times -1 + (-1 \times -1)$$

$$= -1 - 1 + 1 = -1$$

The output $y = f(y_{in})$ is obtained by applying activation function, hence $y = -1$.

Since $t \neq y$, the new weights are calculated as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = -1 + 1 \times 1 \times 1 = 0$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = -1 + 1 \times 1 \times -1 = -2$$

$$b(\text{new}) = b(\text{old}) + \alpha t = -1 + 1 \times 1 = 0$$

The weights after presenting the second sample are

$$w = [0 \ -2 \ 0]$$

For the third input sample, $x_1 = -1$, $x_2 = 1$, $t = -1$, the net input is calculated as,

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= 0 + -1 \times 0 + 1 \times -2 = 0 + 0 - 2 = -2$$

The output is obtained as $y = f(y_{in}) = -1$. Since $t = y$, no weight changes. Thus, even after presenting the third input sample, the weights are

$$w = [0 \ -2 \ 0]$$

For the fourth input sample, $x_1 = -1, x_2 = -1$, $t = -1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= 0 + -1 \times 0 + (-1 \times -2)$$

$$= 0 + 0 + 2 = 2$$

The output is obtained as $y = f(y_{in}) = 1$. Since $t \neq y$, the new weights on updating are given as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times -1 \times -1 = 1$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = -2 + 1 \times -1 \times -1 = -1$$

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times -1 = -1$$

The weights after presenting fourth input sample are

$$w = [1 \ -1 \ -1]$$

One epoch of training for ANDNOT function using perceptron network is tabulated in Table 6.

Table 6

Input			Target	Net input	Calculated output	Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	w_1 (0	w_2 0	b 0)
1	1	1	−1	0	0	−1	−1	−1
1	−1	1	1	−1	−1	0	−2	0
−1	1	1	−1	−2	−1	0	−2	0
−1	−1	1	−1	2	1	1	−1	−1

4. Find the weights required to perform the following classification using perceptron network. The vectors (1, 1, 1, 1) and (−1, 1 −1, −1) are belonging to the class (so have target value 1), vectors (1, 1, 1, −1) and (1, −1, −1, 1) are not belonging to the class (so have target value −1). Assume learning rate as 1 and initial weights as 0.

Solution: The truth table for the given vectors is given in Table 7.

Let $w_1 = w_2 = w_3 = w_4 = b = 0$ and the learning rate $\alpha = 1$. Since the threshold $\theta = 0.2$, so the activation function is

$$y = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \\ -1 & \text{if } y_{in} < -0.2 \end{cases}$$

The net input is given by

$$y_{in} = b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$$

The training is performed and the weights are tabulated in Table 8.

Table 7

Input					
x_1	x_2	x_3	x_4	b	Target (t)
1	1	1	1	1	1
−1	1	−1	−1	1	1
1	1	1	−1	1	−1
1	−1	−1	1	1	−1

Thus, in the third epoch, all the calculated outputs become equal to targets and the network has converged. The network convergence can also be checked by forming separating line equations for separating positive response regions from zero and zero from negative response region.

The network architecture is shown in Figure 5.

5. Classify the two-dimensional input pattern shown in Figure 6 using perceptron network. The symbol "*" indicates the data representation to be +1 and "•" indicates data to be −1. The patterns are I-F. For pattern I, the target is +1, and for F, the target is −1.

Table 8

Inputs					Target	Net input	Output	Weight changes					Weights				
$(x_1$	x_2	x_3	x_4	$b)$	(t)	(y_{in})	(y)	$(\Delta w_1$	Δw_2	Δw_3	Δw_4	$\Delta b)$	$(w_1$ (0	w_2 0	w_3 0	w_4 0	$b)$ 0)
EPOCH-1																	
(1	1	1	1	1)	1	0	0	1	1	1	1	1	1	1	1	1	1
(−1	1	−1	−1	1)	1	−1	−1	−1	1	−1	−1	1	0	2	0	0	2
(1	1	1	−1	1)	−1	4	1	−1	−1	−1	1	−1	−1	1	−1	1	1
(1	−1	−1	1	1)	−1	1	1	−1	1	1	−1	−1	−2	2	0	0	0
EPOCH-2																	
(1	1	1	1	1)	1	0	0	1	1	1	1	1	−1	3	1	1	1
(−1	1	−1	−1	1)	1	3	1	0	0	0	0	0	−1	3	1	1	1
(1	1	1	−1	1)	−1	4	1	−1	−1	−1	1	−1	−2	2	0	2	0
(1	−1	−1	1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0
EPOCH-3																	
(1	1	1	1	1)	1	2	1	0	0	0	0	0	−2	2	0	2	0
(−1	1	−1	−1	1)	1	2	1	0	0	0	0	0	−2	2	0	2	0
(1	1	1	−1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0
(1	−1	−1	1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0

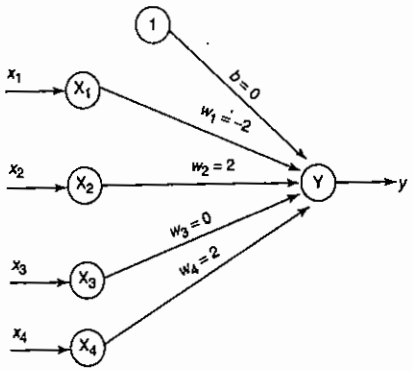

Figure 5 Network architecture.

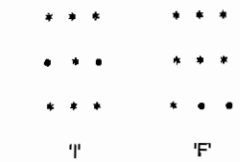

Figure 6 I-F data representation.

Solution: The training patterns for this problem are tabulated in Table 9.

Table 9

	Input									
Pattern	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	1	Target (t)
I	1	1	1	−1	1	−1	1	1	1	1	1
F	1	1	1	1	1	1	−1	−1	1	1	−1

The initial weights are all assumed to be zero, i.e., $\theta = 0$ and $\alpha = 1$. The activation function is given by

$$y = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } -0 \le y_{in} \le 0 \\ -1 & \text{if } y_{in} < -0 \end{cases}$$

For the first input sample, $x_1 = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1]$, $t = 1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{9} x_i w_i$$

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + x_5 w_5 + x_6 w_6 + x_7 w_7 + x_8 w_8 + x_9 w_9$$
$$= 0 + 1 \times 0 + 1 \times 0 + 1 \times 0 + (-1) \times 0 + 1 \times 0 + (-1) \times 0 + 1 \times 0 + 1 \times 0$$
$$y_{in} = 0$$

Therefore, by applying the activation function the output is given by $y = f(y_{in}) = 0$. Now since $t \neq y$, the new weights are computed as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times 1 \times 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times 1 \times 1 = 1$$
$$w_3(\text{new}) = w_3(\text{old}) + \alpha t x_3 = 0 + 1 \times 1 \times 1 = 1$$
$$w_4(\text{new}) = w_4(\text{old}) + \alpha t x_4 = 0 + 1 \times 1 \times -1 = -1$$
$$w_5(\text{new}) = w_5(\text{old}) + \alpha t x_5 = 0 + 1 \times 1 \times 1 = 1$$
$$w_6(\text{new}) = w_6(\text{old}) + \alpha t x_6 = 0 + 1 \times 1 \times -1 = -1$$
$$w_7(\text{new}) = w_7(\text{old}) + \alpha t x_7 = 0 + 1 \times 1 \times 1 = 1$$
$$w_8(\text{new}) = w_8(\text{old}) + \alpha t x_8 = 0 + 1 \times 1 \times 1 = 1$$
$$w_9(\text{new}) = w_9(\text{old}) + \alpha t x_9 = 0 + 1 \times 1 \times 1 = 1$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times 1 = 1$$

The weights after presenting first input sample are

$$w = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1]$$

For the second input sample, $x_2 = [1\ 1\ 1\ 1\ 1\ 1\ 1\ -1\ -1\ 1]$, $t = -1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{9} x_i w_i$$

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + x_5 w_5 + x_6 w_6 + x_7 w_7 + x_8 w_8 + x_9 w_9$$
$$= 1 + 1 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times -1 + 1 \times 1 + 1 \times -1 + 1 \times 1 + (-1) \times 1 + (-1) \times 1$$
$$y_{in} = 2$$

Therefore the output is given by $y = f(y_{in}) = 1$. Since $t \neq y$, the new weights are

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 1 + 1 \times -1 \times 1 = 0$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 1 + 1 \times -1 \times 1 = 0$$
$$w_3(\text{new}) = w_3(\text{old}) + \alpha t x_3 = 1 + 1 \times -1 \times 1 = 0$$
$$w_4(\text{new}) = w_4(\text{old}) + \alpha t x_4 = -1 + 1 \times -1 \times 1 = -2$$
$$w_5(\text{new}) = w_5(\text{old}) + \alpha t x_5 = 1 + 1 \times -1 \times 1 = 0$$
$$w_6(\text{new}) = w_6(\text{old}) + \alpha t x_6 = -1 + 1 \times -1 \times 1 = -2$$
$$w_7(\text{new}) = w_7(\text{old}) + \alpha t x_7 = 1 + 1 \times -1 \times 1 = 0$$
$$w_8(\text{new}) = w_8(\text{old}) + \alpha t x_8 = 1 + 1 \times -1 \times -1 = 2$$
$$w_9(\text{new}) = w_9(\text{old}) + \alpha t x_9 = 1 + 1 \times -1 \times -1 = 2$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 1 + 1 \times -1 = 0$$

The weights after presenting the second input sample are

$$w = [0\ 0\ 0\ -2\ 0\ -2\ 0\ 2\ 2\ 0]$$

The network architecture is as shown in Figure 7. The network can be further trained for its convergence.

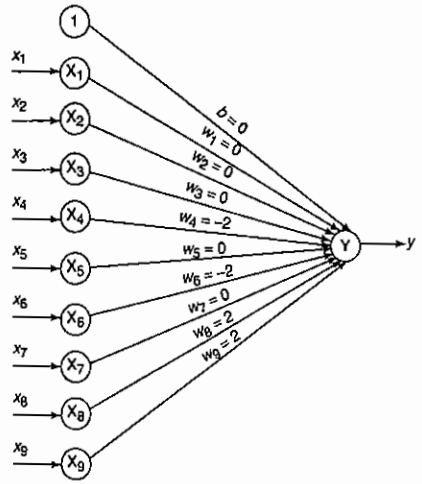

Figure 7 Network architecture.

6. Implement OR function with bipolar inputs and targets using Adaline network.

Solution: The truth table for OR function with bipolar inputs and targets is shown in Table 10.

Table 10

x_1	x_2	1	t
1	1	1	1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

Initially all the weights and links are assumed to be small random values, say 0.1, and the learning rate is also set to 0.1. Also here the least mean square error may be set. The weights are calculated until the least mean square error is obtained.

The initial weights are taken to be $w_1 = w_2 = b = 0.1$ and the learning rate $\alpha = 0.1$. For the first input sample, $x_1 = 1$, $x_2 = 1$, $t = 1$, we calculate the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + \sum_{i=1}^{2} x_i w_i$$
$$= b + x_1 w_1 + x_2 w_2$$
$$= 0.1 + 1 \times 0.1 + 1 \times 0.1 = 0.3$$

Now compute $(t - y_{in}) = (1 - 0.3) = 0.7$. Updating the weights we obrain,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$$

where $\alpha(t - y_{in})x_i$ is called as weight change Δw_i. The new weights are obtained as

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.1 + 0.1 \times 0.7 \times 1$$
$$= 0.1 + 0.07 = 0.17$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1$$
$$+ 0.1 \times 0.7 \times 1 = 0.17$$
$$b(\text{new}) = b(\text{old}) + \Delta b = 0.1 + 0.1 \times 0.7 = 0.17$$

where

$$\Delta w_1 = \alpha(t - y_{in})x_1$$
$$\Delta w_2 = \alpha(t - y_{in})x_2$$
$$\Delta b = \alpha(t - y_{in})$$

Now we calculate the error:

$$E = (t - y_{in})^2 = (0.7)^2 = 0.49$$

The final weights after presenting first input sample are

$$w = [0.17\ \ 0.17\ \ 0.17]$$

and error $E = 0.49$.

These calculations are performed for all the input samples and the error is calculated. One epoch is completed when all the input patterns are presented. Summing up all the errors obtained for each input sample during one epoch will give the total mean square error of that epoch. The network training is continued until this error is minimized to a very small value.

Adopting the method above, the network training is done for OR function using Adaline network and is tabulated below in Table 11 for $\alpha = 0.1$.

The total mean square error after each epoch is given as in Table 12.

Thus from Table 12, it can be noticed that as training goes on, the error value gets minimized. Hence, further training can be continued for further minimization of error. The network architecture of Adaline network for OR function is shown in Figure 8.

Table 12

Epoch	Total mean square error
Epoch 1	3.02
Epoch 2	1.938
Epoch 3	1.5506
Epoch 4	1.417
Epoch 5	1.377

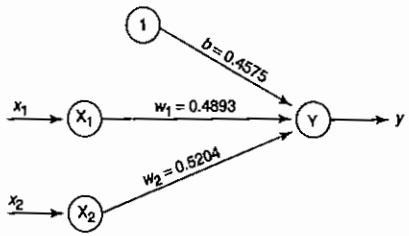

Figure 8 Network architecture of Adaline.

Table 11

Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t-y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.1	w_2 0.1	b 0.1)	$(t-y_{in})^2$
EPOCH-1												
1	1	1	1	0.3	0.7	0.07	0.07	0.07	0.17	0.17	0.17	0.49
1	−1	1	1	0.17	0.83	0.083	−0.083	0.083	0.253	0.087	0.253	0.69
−1	1	1	1	0.087	0.913	−0.0913	0.0913	0.0913	0.1617	0.1783	0.3443	0.83
−1	−1	1	−1	0.0043	−1.0043	0.1004	0.1004	−0.1004	0.2621	0.2787	0.2439	1.01
EPOCH-2												
1	1	1	1	0.7847	0.2153	0.0215	0.0215	0.0215	0.2837	0.3003	0.2654	0.046
1	−1	1	1	0.2488	0.7512	0.7512	−0.0751	0.0751	0.3588	0.2251	0.3405	0.564
−1	1	1	1	0.2069	0.7931	−0.7931	0.0793	0.0793	0.2795	0.3044	0.4198	0.629
−1	−1	1	−1	−0.1641	−0.8359	0.0836	0.0836	−0.0836	0.3631	0.388	0.336	0.699
EPOCH-3												
1	1	1	1	1.0873	−0.0873	−0.087	−0.087	−0.087	0.3543	0.3793	0.3275	0.0076
1	−1	1	1	0.3025	+0.6975	0.0697	−0.0697	0.0697	0.4241	0.3096	0.3973	0.487
−1	1	1	1	0.2827	0.7173	−0.0717	0.0717	0.0717	0.3523	0.3813	0.469	0.515
−1	−1	1	−1	−0.2647	−0.7353	0.0735	0.0735	−0.0735	0.4259	0.4548	0.3954	0.541
EPOCH-4												
1	1	1	1	1.2761	−0.2761	−0.0276	−0.0276	−0.0276	0.3983	0.4272	0.3678	0.076
1	−1	1	1	0.3389	0.6611	0.0661	−0.0661	0.0661	0.4644	0.3611	0.4339	0.437
−1	1	1	1	0.3307	0.6693	−0.0669	0.0669	0.0699	0.3974	0.428	0.5009	0.448
−1	−1	1	−1	−0.3246	−0.6754	0.0675	0.0675	−0.0675	0.465	0.4956	0.4333	0.456
EPOCH-5												
1	1	1	1	1.3939	−0.3939	−0.0394	−0.0394	−0.0394	0.4256	0.4562	0.393	0.155
1	−1	1	1	0.3634	0.6366	0.0637	−0.0637	0.0637	0.4893	0.3925	0.457	0.405
−1	1	1	1	0.3609	0.6391	−0.0639	0.0639	0.0639	0.4253	0.4654	0.5215	0.408
−1	−1	1	−1	−0.3603	−0.6397	0.064	0.064	−0.064	0.4893	0.5204	0.4575	0.409

7. Use Adaline network to train ANDNOT function with bipolar inputs and targets. Perform 2 epochs of training.

Solution: The truth table for ANDNOT function with bipolar inputs and targets is shown in Table 13.

Table 13

x_1	x_2	1	t
1	1	1	−1
1	−1	1	1
−1	1	1	−1
−1	−1	1	−1

Initially the weights and bias have assumed a random value say 0.2. The learning rate is also set to 0.2. The weights are calculated until the least mean square error is obtained. The initial weights are $w_1 = w_2 = b = 0.2$, and $\alpha = 0.2$. For the first input sample $x_1 = 1$, $x_2 = 1$, $t = -1$, we calculate the net input as

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0.2 + 1 \times 0.2 + 1 \times 0.2 = 0.6$$

Now compute $(t - y_{in}) = (-1 - 0.6) = -1.6$. Updating the weights we obtain

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$$

The new weights are obtained as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha(t - y_{in})x_1$$
$$= 0.2 + 0.2 \times (-1.6) \times 1 = -0.12$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha(t - y_{in})x_2$$
$$= 0.2 + 0.2 \times (-1.6) \times 1 = -0.12$$

$$b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$$
$$= 0.2 + 0.2 \times (-1.6) = -0.12$$

Now we compute the error,

$$E = (t - y_{in})^2 = (-1.6)^2 = 2.56$$

The final weights after presenting first input sample are $w = [-0.12 - 0.12 - 0.12]$ and error $E = 2.56$.

The operational steps are carried for 2 epochs of training and network performance is noted. It is tabulated as shown in Table 14.

The total mean square error at the end of two epochs is summation of the errors of all input samples as shown in Table 15.

Table 15

Epoch	Total mean square error
Epoch 1	5.71
Epoch 2	2.43

Hence from Table 15, it is clearly understood that the mean square error decreases as training progresses. Also, it can be noted that at the end of the sixth epoch, the error becomes approximately equal to 1. The network architecture for ANDNOT function using Adaline network is shown in Figure 9.

Table 14

Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t-y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.2	w_2 0.2	b 0.2)	$(t-y_{in})^2$
EPOCH-1												
1	1	1	−1	0.6	−1.6	−0.32	−0.32	−0.32	−0.12	−0.12	−0.12	2.56
1	−1	1	1	−0.12	1.12	0.22	−0.22	0.22	0.10	−0.34	0.10	1.25
−1	1	1	−1	−0.34	−0.66	0.13	−0.13	−0.13	0.24	−0.48	−0.03	0.43
−1	−1	1	−1	0.21	−1.2	0.24	0.24	−0.24	0.48	−0.23	−0.27	1.47
EPOCH-2												
1	1		−1	−0.02	−0.98	−0.195	−0.195	−0.195	0.28	−0.43	−0.46	0.95
1	−1	1	1	0.25	0.76	0.15	−0.15	0.15	0.43	−0.58	−0.31	0.57
−1	1	1	−1	−1.33	0.33	−0.065	0.065	0.065	0.37	−0.51	−0.25	0.106
−1	−1	1	−1	−0.11	−0.90	0.18	0.18	−0.18	0.55	−0.38	0.43	0.8

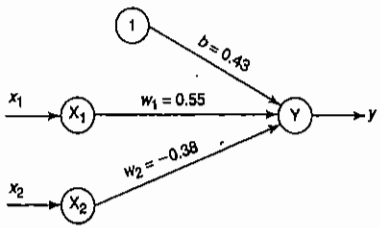

Figure 9 Network architecture for ANDNOT function using Adaline network.

8 Using Madaline network, implement XOR function with bipolar inputs and targets. Assume the required parameters for training of the network.

Solution: The training pattern for XOR function is given in Table 16.

Table 16

x_1	x_2	1	t
1	1	1	−1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

The Madaline Rule I (MRI) algorithm in which the weights between the hidden layer and output layer remain fixed is used for training the network. Initializing the weights to small random values, the network architecture is as shown in Figure 10, with initial weights. From Figure 10, the initial weights and bias are $[w_{11}\ w_{21}\ b_1] = [0.05\ 0.2\ 0.3]$, $[w_{12}\ w_{22}\ b_2] = [0.1\ 0.2\ 0.15]$ and $[v_1\ v_2\ b_3] = [0.5\ 0.5\ 0.5]$. For first

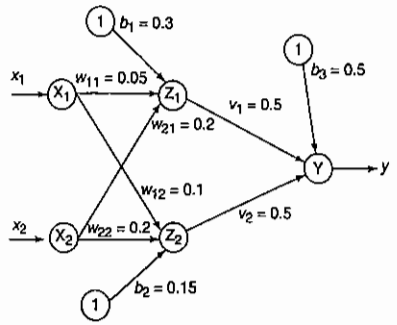

Figure 10 Network architecture of Madaline for XOR functions (initial weights given).

input sample, $x_1 = 1$, $x_2 = 1$, target $t = -1$, and learning rate α equal to 0.5:

- Calculate net input to the hidden units:

$$z_{in1} = b_1 + x_1 w_{11} + x_2 w_{21}$$
$$= 0.3 + 1 \times 0.05 + 1 \times 0.2 = 0.55$$
$$z_{in2} = b_2 + x_1 w_{12} + x_2 w_{22}$$
$$= 0.15 + 1 \times 0.1 + 1 \times 0.2 = 0.45$$

- Calculate the output z_1, z_2 by applying the activations over the net input computed. The activation function is given by

$$f(z_{in}) = \begin{cases} 1 & \text{if } z_{in} \geq 0 \\ -1 & \text{if } z_{in} < 0 \end{cases}$$

Hence,

$$z_1 = f(z_{in1}) = f(0.55) = 1$$
$$z_2 = f(z_{in2}) = f(0.45) = 1$$

- After computing the output of the hidden units, then find the net input entering into the output unit:

$$y_{in} = b_3 + z_1 v_1 + z_2 v_2$$
$$= 0.5 + 1 \times 0.5 + 1 \times 0.5 = 1.5$$

- Apply the activation function over the net input y_{in} to calculate the output y:

$$y = f(y_{in}) = f(1.5) = 1$$

- Since $t \neq y$, weight updation has to be performed. Also since $t = -1$, the weights are updated on z_1 and z_2 that have positive net input. Since here both net inputs z_{in1} and z_{in2} are positive, updating the weights and bias on both hidden units, we obtain

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(t - z_{inj})x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha(t - z_{inj})$$

This implies:

$$w_{11}(\text{new}) = w_{11}(\text{old}) + \alpha(t - z_{in1})x_1$$
$$= 0.05 + 0.5(-1 - 0.55) \times 1 = -0.725$$
$$w_{12}(\text{new}) = w_{12}(\text{old}) + \alpha(t - z_{in2})x_1$$
$$= 0.1 + 0.5(-1 - 0.45) \times 1 = -0.625$$
$$b_1(\text{new}) = b_1(\text{old}) + \alpha(t - z_{in1})$$
$$= 0.3 + 0.5(-1 - 0.55) = -0.475$$

$$w_{21}(\text{new}) = w_{21}(\text{old}) + \alpha(t - z_{in1})x_2$$
$$= 0.2 + 0.5(-1 - 0.55) \times 1 = -0.575$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + \alpha(t - z_{in2})x_2$$
$$= 0.2 + 0.5(-1 - 0.45) \times 1 = -0.525$$
$$b_2(\text{new}) = b_2(\text{old}) + \alpha(t - z_{in2})$$
$$= 0.15 + 0.5(-1 - 0.45) = -0.575$$

All the weights and bias between the input layer and hidden layer are adjusted. This completes the training for the first epoch. The same process is repeated until the weight converges. It is found that the weight converges at the end of 3 epochs. Table 17 shows the training performance of Madaline network for XOR function.

The network architecture for Madaline network with final weights for XOR function is shown in Figure 11.

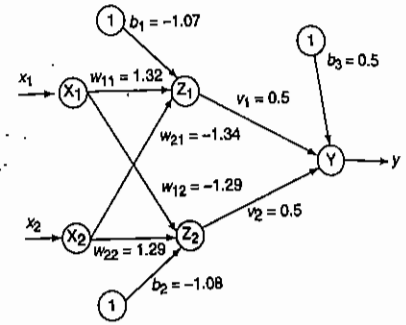

Figure 11 Madaline network for XOR function (final weights given).

9. Using back-propagation network, find the new weights for the net shown in Figure 12. It is presented with the input pattern [0, 1] and the target output is 1. Use a learning rate $\alpha = 0.25$ and binary sigmoidal activation function.

Solution: The new weights are calculated based on the training algorithm in Section 3.5.4. The initial weights are $[v_{11}\ v_{21}\ v_{01}] = [0.6\ -0.1\ 0.3]$,

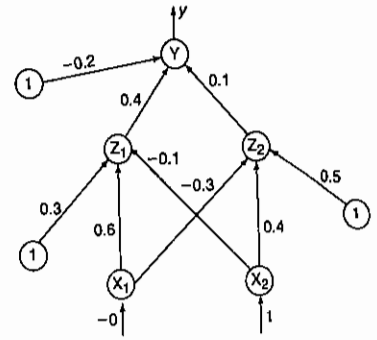

Figure 12 Network.

Table 17

Inputs			Target											
x_1	x_2	1	(t)	z_{in1}	z_{in2}	z_1	z_2	y_{in}	y	w_{11}	w_{21}	b_1	w_{12}	w_{22}	b_2
EPOCH-1															
1	1	1	−1	0.55	0.45	1	1	1.5	1	−0.725	−0.58	−0.475	−0.625	−0.525	−0.575
1	−1	1	1	−0.625	−0.675	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−0.625	−0.525	−0.575
−1	1	1	1	−1.1375	−0.475	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−1.3625	0.2125	0.1625
−1	−1	1	−1	1.6375	1.3125	1	1	1.5	1	1.4065	−0.069	−0.98	−0.207	1.369	−0.994
EPOCH-2															
1	1	1	−1	0.3565	0.168	1	1	1.5	1	0.7285	−0.75	−1.66	−0.791	−0.207	−1.58
1	−1	1	1	−0.1845	−3.154	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−0.791	0.785	−1.58
−1	1	1	1	−3.728	−0.002	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	0.785	−1.08
−1	−1	1	−1	−1.0495	−1.071	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	1.29	−1.08
EPOCH-3															
1	1	1	−1	−1.0865	−1.083	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
1	−1	1	1	1.5915	−3.655	1	−1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	1	1	1	−3.728	1.501	−1	1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	−1	1	−1	−1.0495	−1.701	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08

$[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$. Activation function used is binary sigmoidal activation function and is given by

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

Given the output sample $[x_1, x_2] = [0, 1]$ and target $t = 1$,

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + 0 \times 0.6 + 1 \times -0.1 = 0.2$$

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + 0 \times -0.3 + 1 \times 0.4 = 0.9$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1}{1+e^{-z_{in1}}} = \frac{1}{1+e^{-0.2}} = 0.5498$$

$$z_2 = f(z_{in2}) = \frac{1}{1+e^{-z_{in2}}} = \frac{1}{1+e^{-0.9}} = 0.7109$$

- Calculate the net input entering the output layer. For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + 0.5498 \times 0.4 + 0.7109 \times 0.1$$
$$= 0.09101$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1}{1+e^{-y_{in}}} = \frac{1}{1+e^{-0.09101}} = 0.5227$$

- Compute the error portion δ_k:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = f(y_{in})[1 - f(y_{in})] = 0.5227[1 - 0.5227]$$
$$f'(y_{in}) = 0.2495$$

This implies

$$\delta_1 = (1 - 0.5227)\ (0.2495) = 0.1191$$

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.1191 \times 0.5498$$
$$= 0.0164$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.1191 \times 0.7109$$
$$= 0.02117$$

$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.1191 = 0.02978$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj})$$

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$
$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.1191 \times 0.4 = 0.04764$$
$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.1191 \times 0.1 = 0.01191$$

Error, $\delta_1 = \delta_{in1} f'(z_{in1})$

$$f'(z_{in1}) = f(z_{in1})[1 - f(z_{in1})]$$
$$= 0.5498[1 - 0.5498] = 0.2475$$
$$\delta_1 = \delta_{in1} f'(z_{in1})$$
$$= 0.04764 \times 0.2475 = 0.0118$$

Error, $\delta_2 = \delta_{in2} f'(z_{in2})$

$$f'(z_{in2}) = f(z_{in2})[1 - f(z_{in2})]$$
$$= 0.7109[1 - 0.7109] = 0.2055$$
$$\delta_2 = \delta_{in2} f'(z_{in2})$$
$$= 0.01191 \times 0.2055 = 0.00245$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.0118 \times 0 = 0$$
$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.0118 \times 1 = 0.00295$$
$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.0118 = 0.00295$$
$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.00245 \times 0 = 0$$
$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.00245 \times 1 = 0.0006125$$
$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.00245 = 0.0006125$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 + 0 = 0.6$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 + 0 = -0.3$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$= -0.1 + 0.00295 = -0.09705$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$
$$= 0.4 + 0.0006125 = 0.4006125$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 + 0.0164$$
$$= 0.4164$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.02117$$
$$= 0.12117$$
$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.00295$$
$$= 0.30295$$
$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$= 0.5 + 0.0006125 = 0.5006125$$
$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.02978$$
$$= -0.17022$$

Thus, the final weights have been computed for the network shown in Figure 12.

10. Find the new weights, using back-propagation network for the network shown in Figure 13. The network is presented with the input pattern $[-1, 1]$ and the target output is $+1$. Use a learning rate of $\alpha = 0.25$ and bipolar sigmoidal activation function.

Solution: The initial weights are $[v_{11} v_{21} v_{01}] = [0.6\ -0.1\ 0.3]$, $[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$.

Activation function used is binary sigmoidal activation function and is given by

$$f(x) = \frac{2}{1+e^{-x}} - 1 = \frac{1-e^{-x}}{1+e^{-x}}$$

Given the input sample $[x_1, x_2] = [-1, 1]$ and target $t = 1$:

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + (-1) \times 0.6 + 1 \times -0.1 = -0.4$$

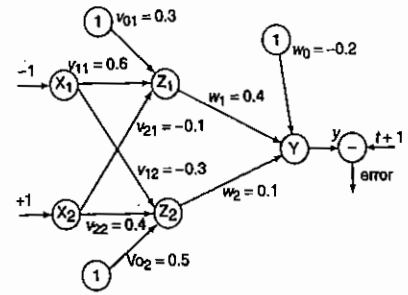

Figure 13 Network.

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + (-1) \times -0.3 + 1 \times 0.4 = 1.2$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1-e^{-z_{in1}}}{1+e^{-z_{in1}}} = \frac{1-e^{0.4}}{1+e^{0.4}} = -0.1974$$

$$z_2 = f(z_{in2}) = \frac{1-e^{-z_{in1}}}{1+e^{-z_{in2}}} = \frac{1-e^{-1.2}}{1+e^{-1.2}} = 0.537$$

- Calculate the net input entering the output layer. For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + (-0.1974) \times 0.4 + 0.537 \times 0.1$$
$$= -0.22526$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1-e^{-y_{in}}}{1+e^{-y_{in}}} = \frac{1-e^{0.22526}}{1+e^{0.22526}} = -0.1122$$

- Compute the error portion δ_k:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = 0.5[1 + f(y_{in})][1 - f(y_{in})]$$
$$= 0.5[1 - 0.1122][1 + 0.1122] = 0.4937$$

This implies

$$\delta_1 = (1 + 0.1122)\,(0.4937) = 0.5491$$

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.5491 \times -0.1974 = -0.0271$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.5491 \times 0.537 = 0.0737$$

$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.5491 = 0.1373$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj})$$

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.5491 \times 0.4 = 0.21964$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.5491 \times 0.1 = 0.05491$$

$$\text{Error, } \delta_1 = \delta_{in1} f'(z_{in1}) = 0.21964 \times 0.5 \times (1 + 0.1974)(1 - 0.1974) = 0.1056$$

$$\text{Error, } \delta_2 = \delta_{in2} f'(z_{in2}) = 0.05491 \times 0.5 \times (1 - 0.537)(1 + 0.537) = 0.0195$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.1056 \times -1 = -0.0264$$

$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.1056 \times 1 = 0.0264$$

$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.1056 = 0.0264$$

$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.0195 \times -1 = -0.0049$$

$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.0195 \times 1 = 0.0049$$

$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.0195 = 0.0049$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 - 0.0264 = 0.5736$$

$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 - 0.0049 = -0.3049$$

$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21} = -0.1 + 0.0264 = -0.0736$$

$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22} = 0.4 + 0.0049 = 0.4049$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 - 0.0271 = 0.3729$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.0737 = 0.1737$$

$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.0264 = 0.3264$$

$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02} = 0.5 + 0.0049 = 0.5049$$

$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.1373 = -0.0627$$

Thus, the final weight has been computed for the network shown in Figure 13.

3.13 Review Questions

1. What is supervised learning and how is it different from unsupervised learning?
2. How does learning take place in supervised learning?
3. From a mathematical point of view, what is the process of learning in supervised learning?
4. What is the building block of the perceptron?
5. Does perceptron require supervised learning? If no, what does it require?
6. List the limitations of perceptron.
7. State the activation function used in perceptron network.
8. What is the importance of threshold in perceptron network?
9. Mention the applications of perceptron network.
10. What are feature detectors?
11. With a neat flowchart, explain the training process of perceptron network.
12. What is the significance of error signal in perceptron network?

13. State the testing algorithm used in perceptron algorithm.
14. How is the linear separability concept implemented using perceptron network training?
15. Define perceptron learning rule.
16. Define delta rule.
17. State the error function for delta rule.
18. What is the drawback of using optimization algorithm?
19. What is Adaline?
20. Draw the model of an Adaline network.
21. Explain the training algorithm used in Adaline network.
22. How is a Madaline network formed?
23. Is it true that Madaline network consists of many perceptrons?
24. State the characteristics of weighted interconnections between Adaline and Madaline.
25. How is training adopted in Madaline network using majority vote rule?
26. State few applications of Adaline and Madaline.
27. What is meant by epoch in training process?
28. What is meant by gradient descent method?
29. State the importance of back-propagation algorithm.
30. What is called as memorization and generalization?
31. List the stages involved in training of back-propagation network.
32. Draw the architecture of back-propagation algorithm.
33. State the significance of error portions δ_k and δ_j in BPN algorithm.
34. What are the activations used in back-propagation network algorithm?
35. What is meant by local minima and global minima?
36. Derive the generalized delta learning rule.
37. Derive the derivations of the binary and bipolar sigmoidal activation function.
38. What are the factors that improve the convergence of learning in BPN network?
39. What is meant by incremental learning?
40. Why is gradient descent method adopted to minimize error?
41. What are the methods of initialization of weights?
42. What is the necessity of momentum factor in weight updation process?
43. Define "over fitting" or "over training."
44. State the techniques for proper choice of learning rate.
45. What are the limitations of using momentum factor?
46. How many hidden layers can there be in a neural network?
47. What is the activation function used in radial basis function network?
48. Explain the training algorithm of radial basis function network.
49. By what means can an IIR and an FIR filter be formed in neural network?
50. What is the importance of functional link network?
51. Write a short note on binary classification tree neural network.
52. Explain in detail about wavelet neural network.

3.14 Exercise Problems

1. Implement NOR function using perceptron network for bipolar inputs and targets.
2. Find the weights required to perform the following classifications using perceptron network. The vectors (1, 1, −1, −1) and (1, −1, 1, −1) are belonging to the class (so have target value 1), vector (−1, −1, −1, 1) and (−1, −1, 1 1) are not belonging to the class (so have target value −1). Assume learning rate 1 and initial weights as 0.