Supervised Learning Network

3

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

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

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.

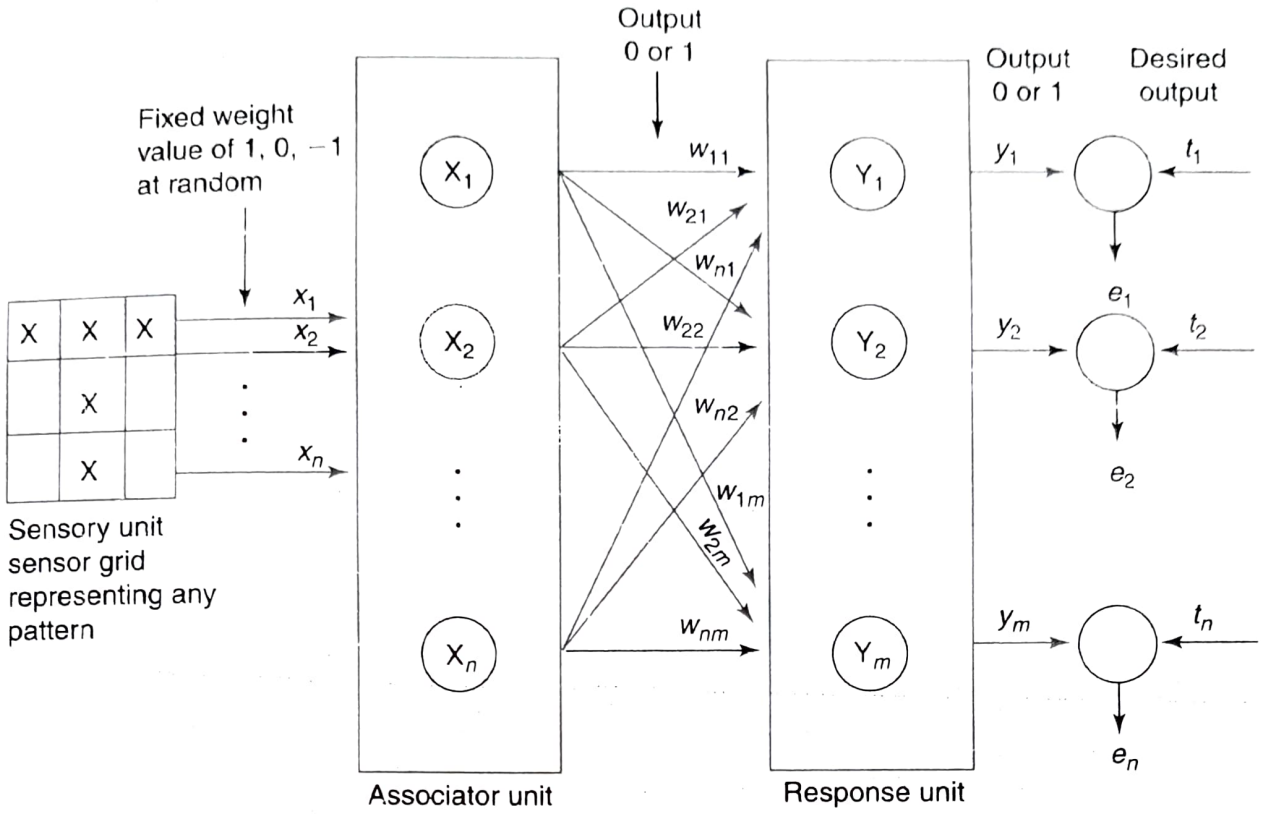

Figure 3-1 Original perceptron network.

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.

$$\text{If } y \neq t, \text{ then}$$

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

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

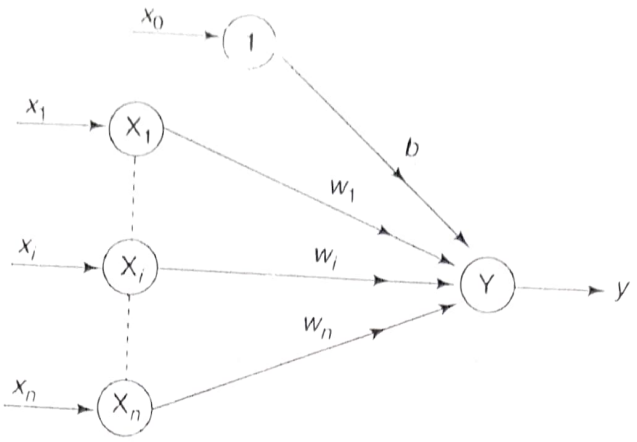

Figure 3-2 Single classification perceptron network.

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

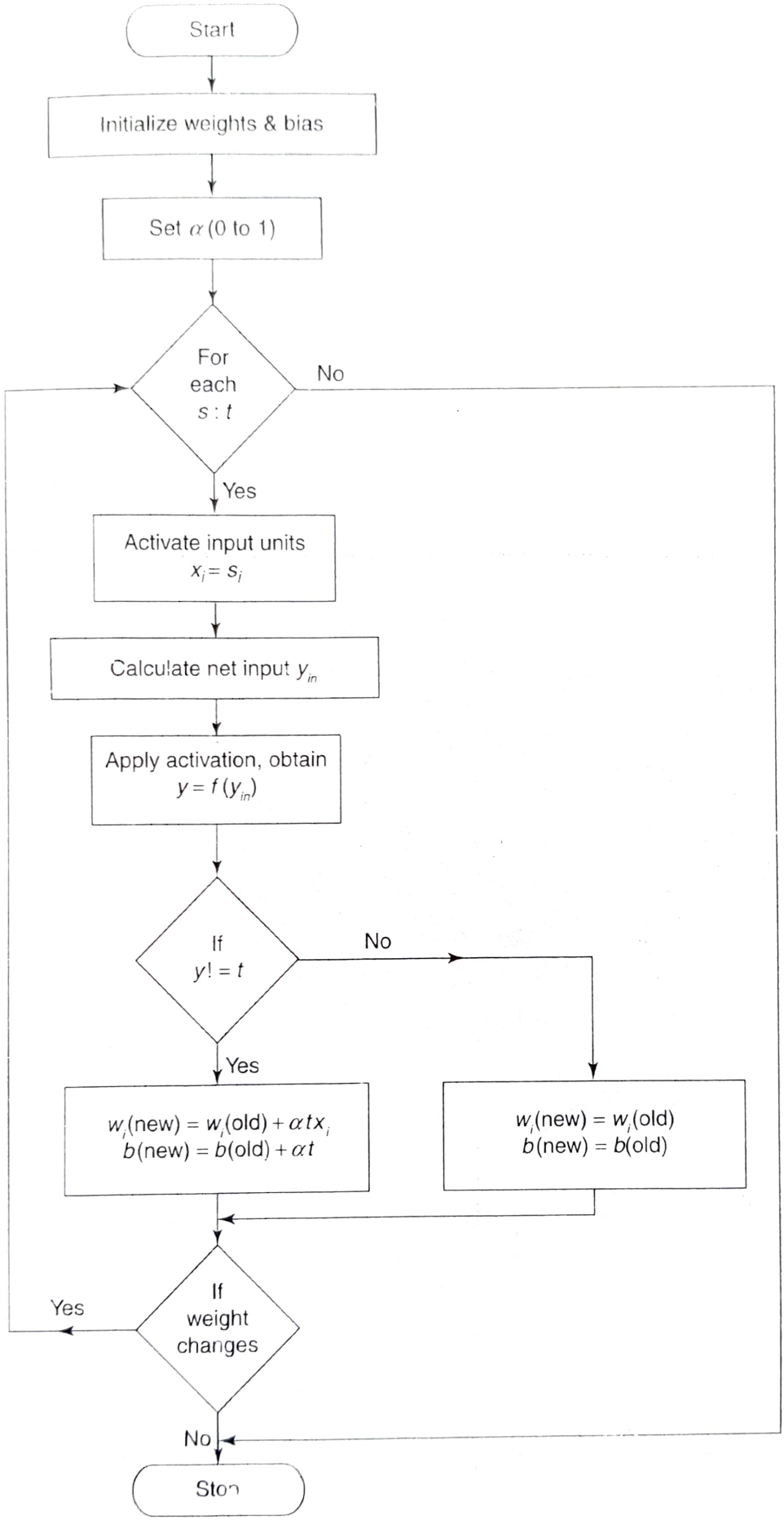

Figure 3-3 Flowchart for perceptron network with single output.

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:

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

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.

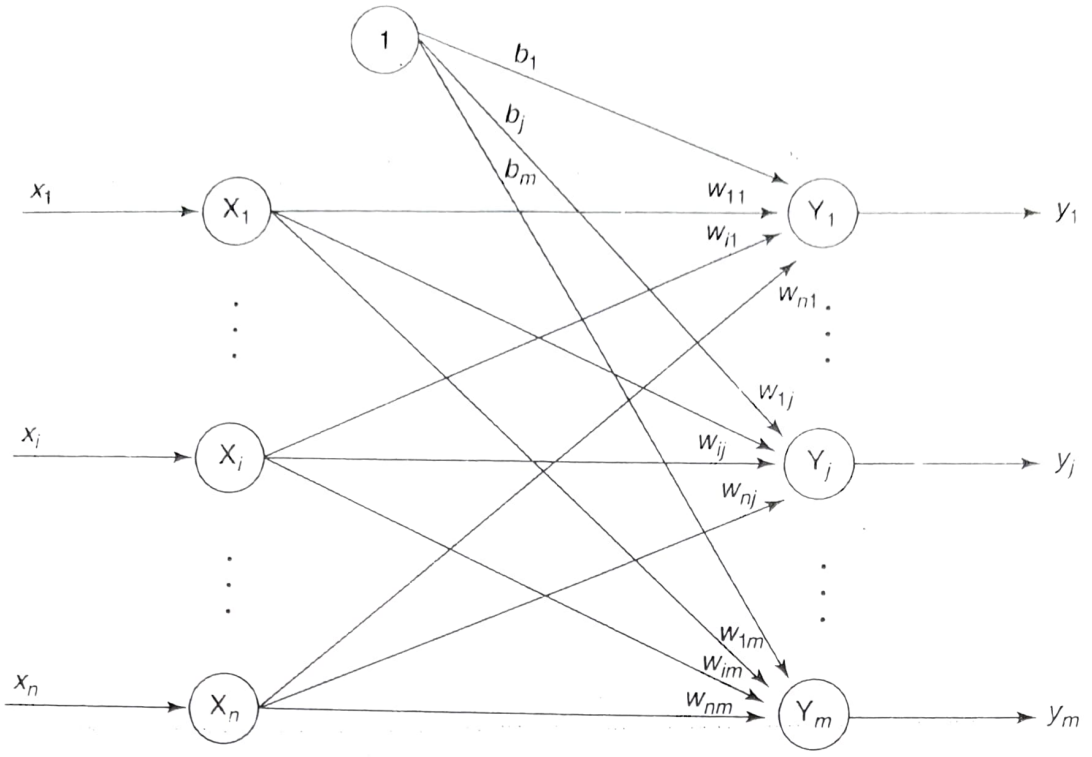

Figure 3-4 Network architecture for perceptron network for several output classes.

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.

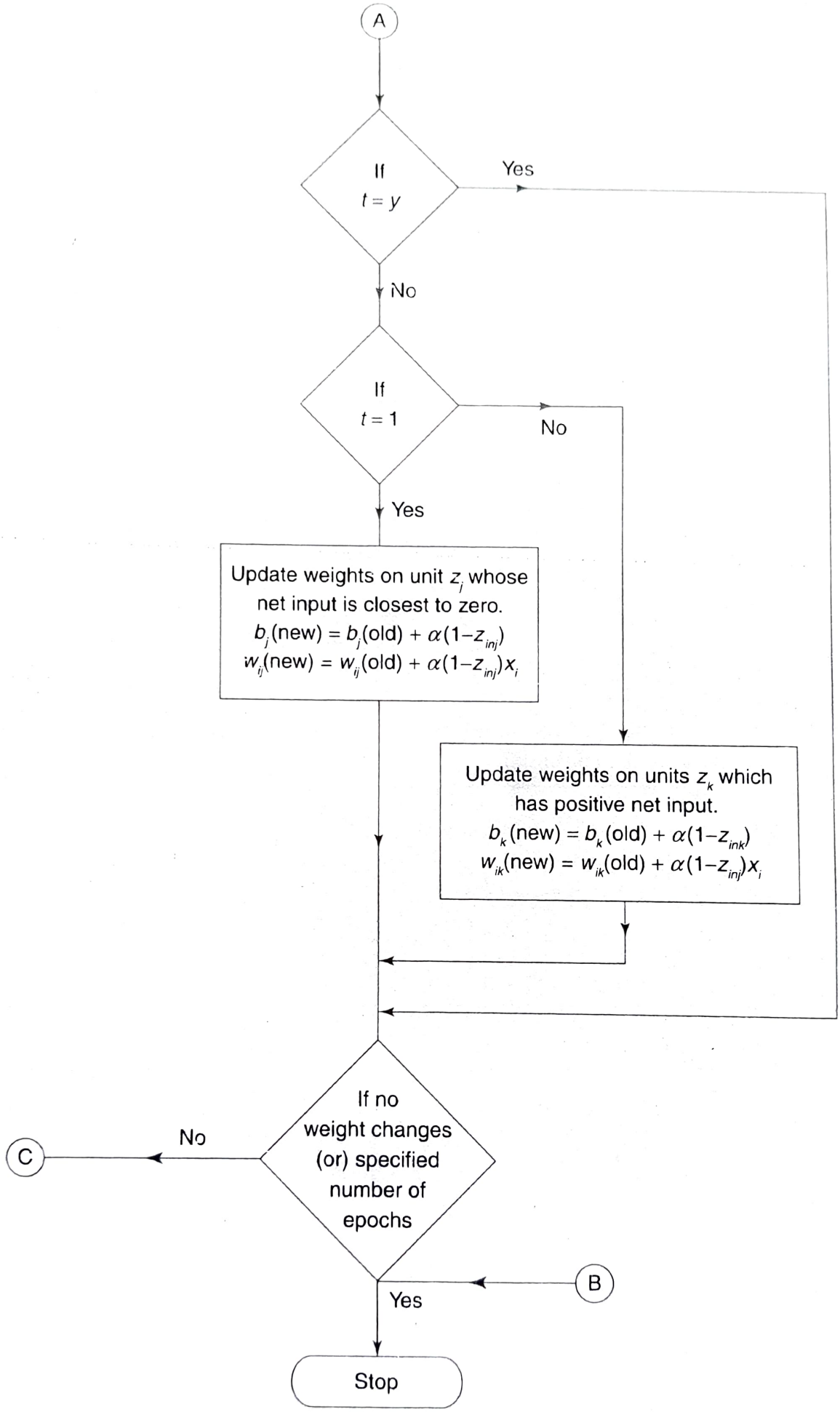

Figure 3-8 (*Continued*).

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.

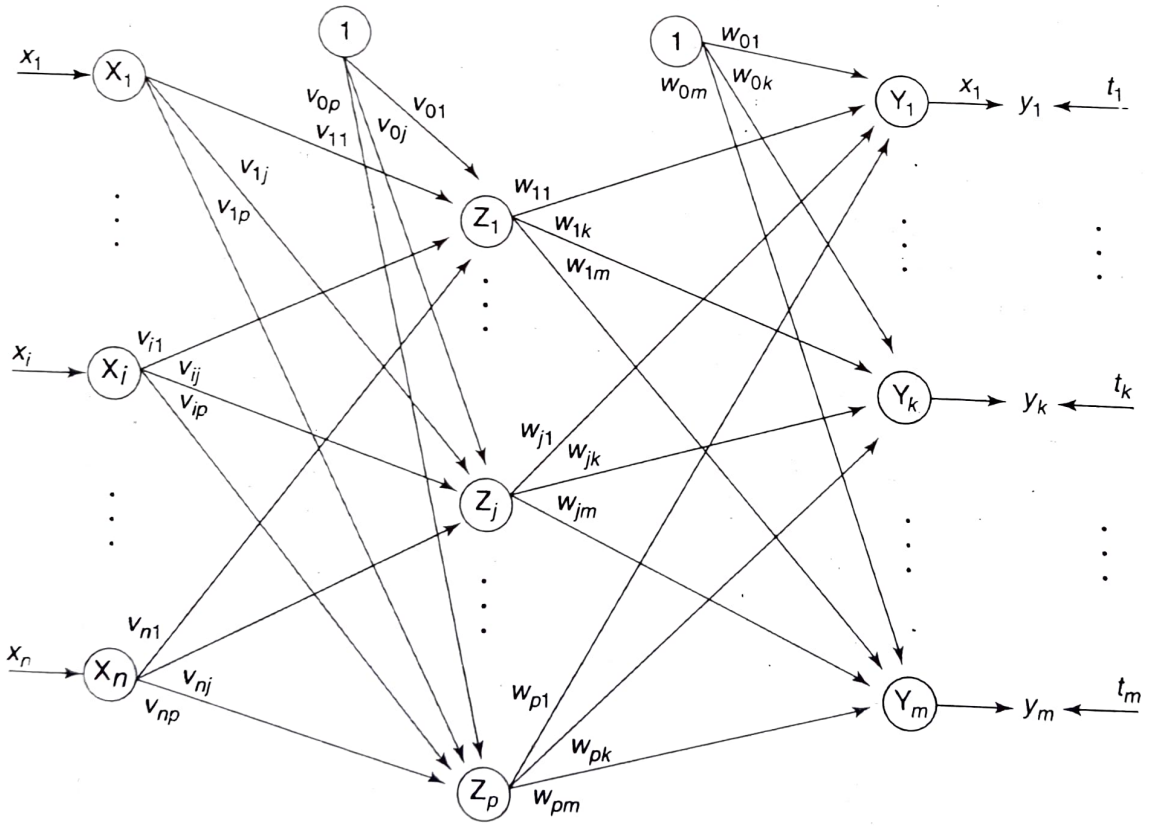

Figure 3-9 Architecture of a back-propagation network.

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.

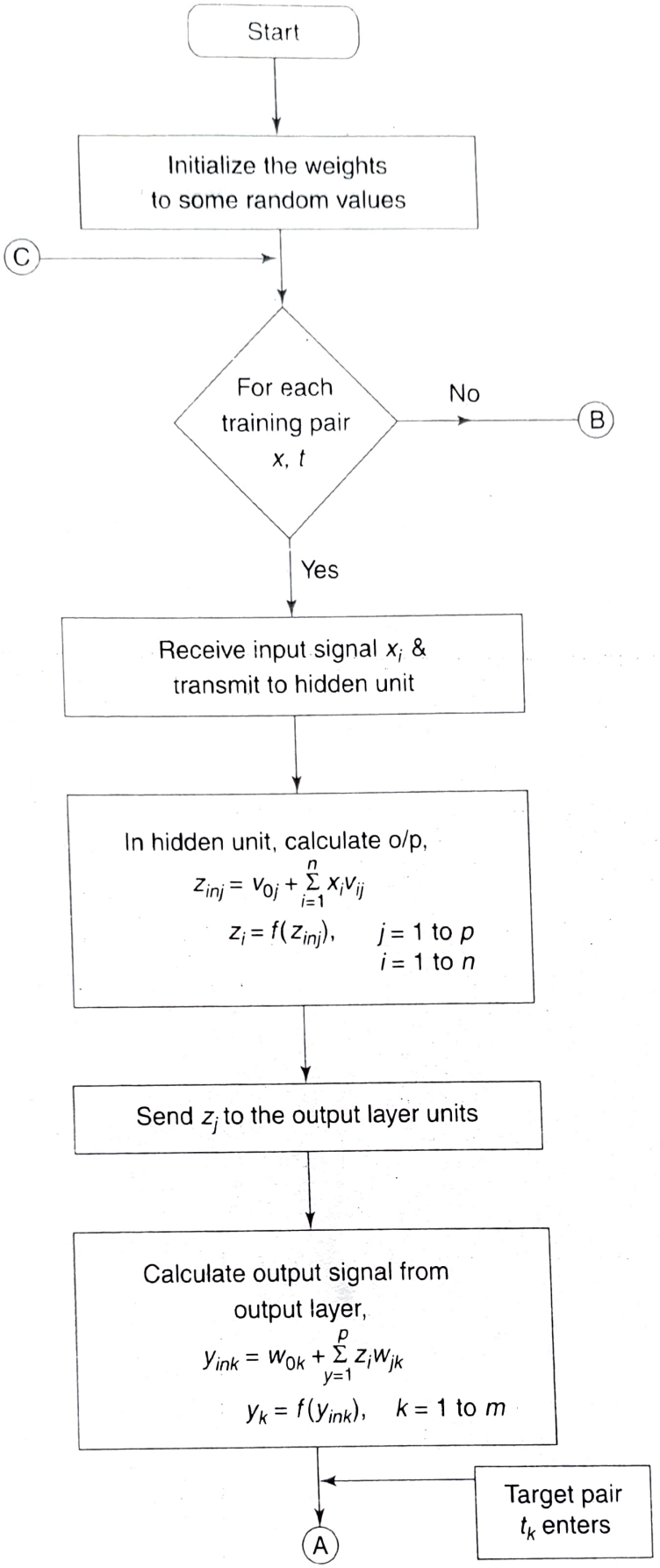

Figure 3-10 Flowchart for back-propagation network training.

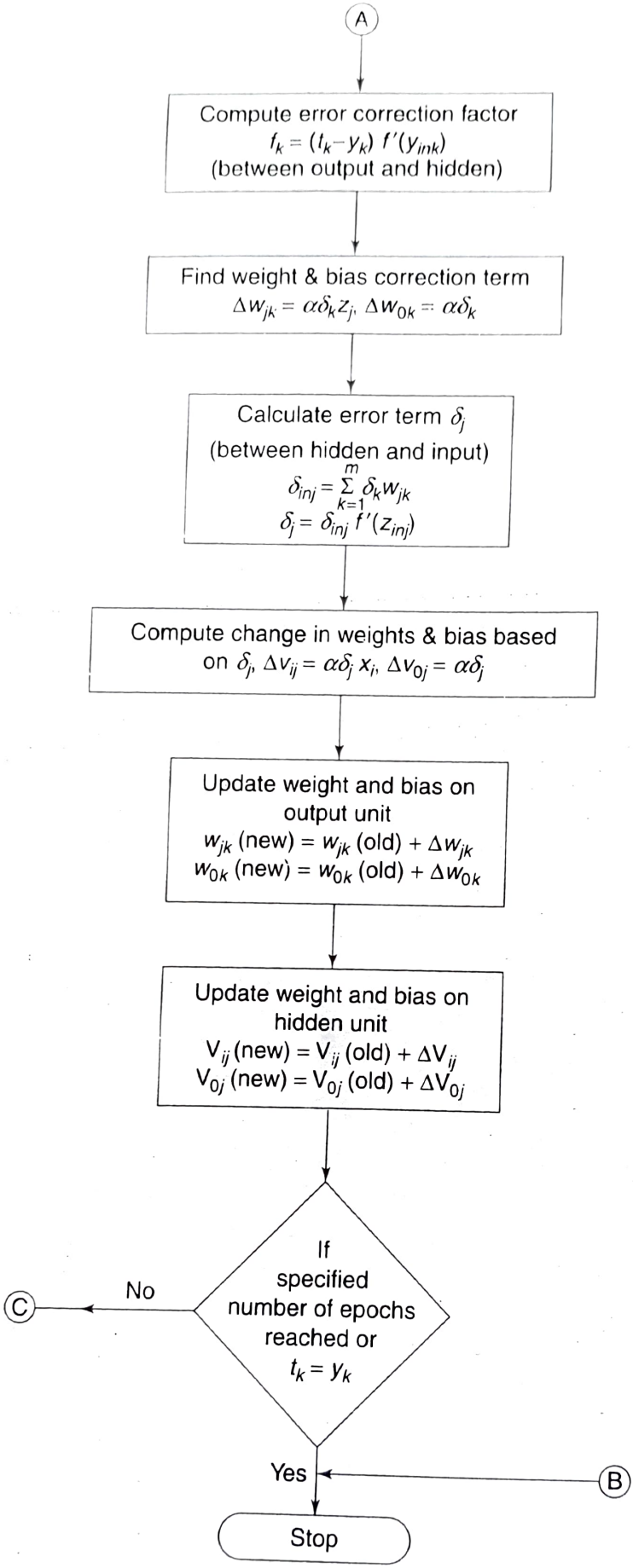

Figure 3-10 (*Continued*).

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

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

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

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

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,

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

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

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 $|y| \rightarrow 0$. The Gaussian function is generally defined as

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

The derivative of this function is given by

$$f'(y) = -2y\,e^{-y^2} = -2yf(y)$$

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.

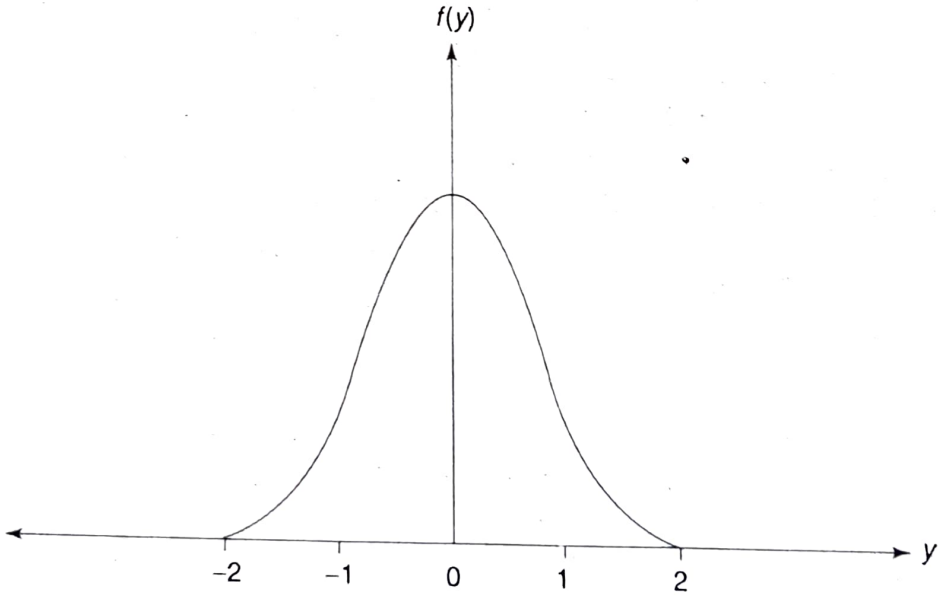

Figure 3-11 Gaussian kernel function.

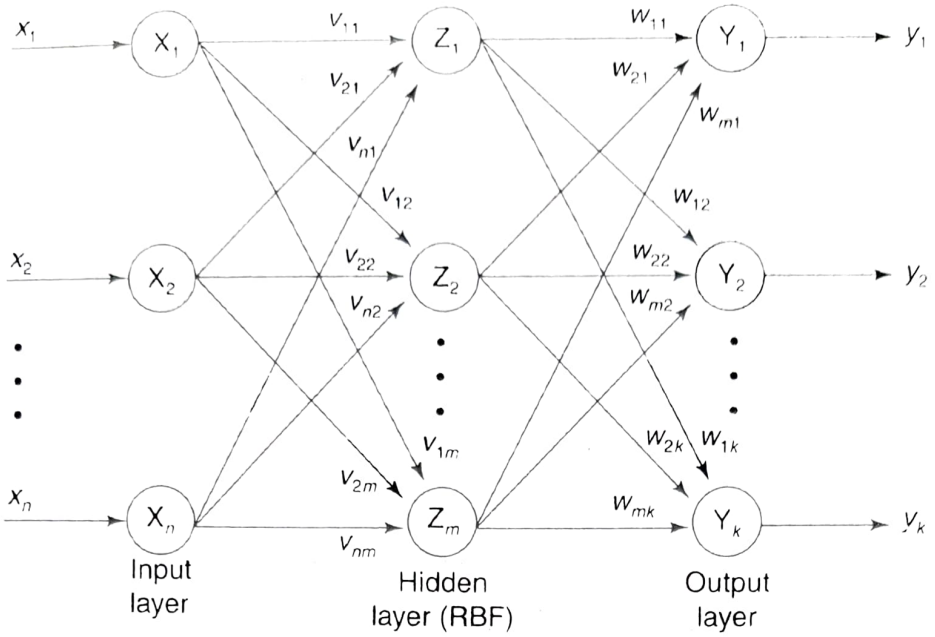

Figure 3-12 Architecture of RBF.

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.

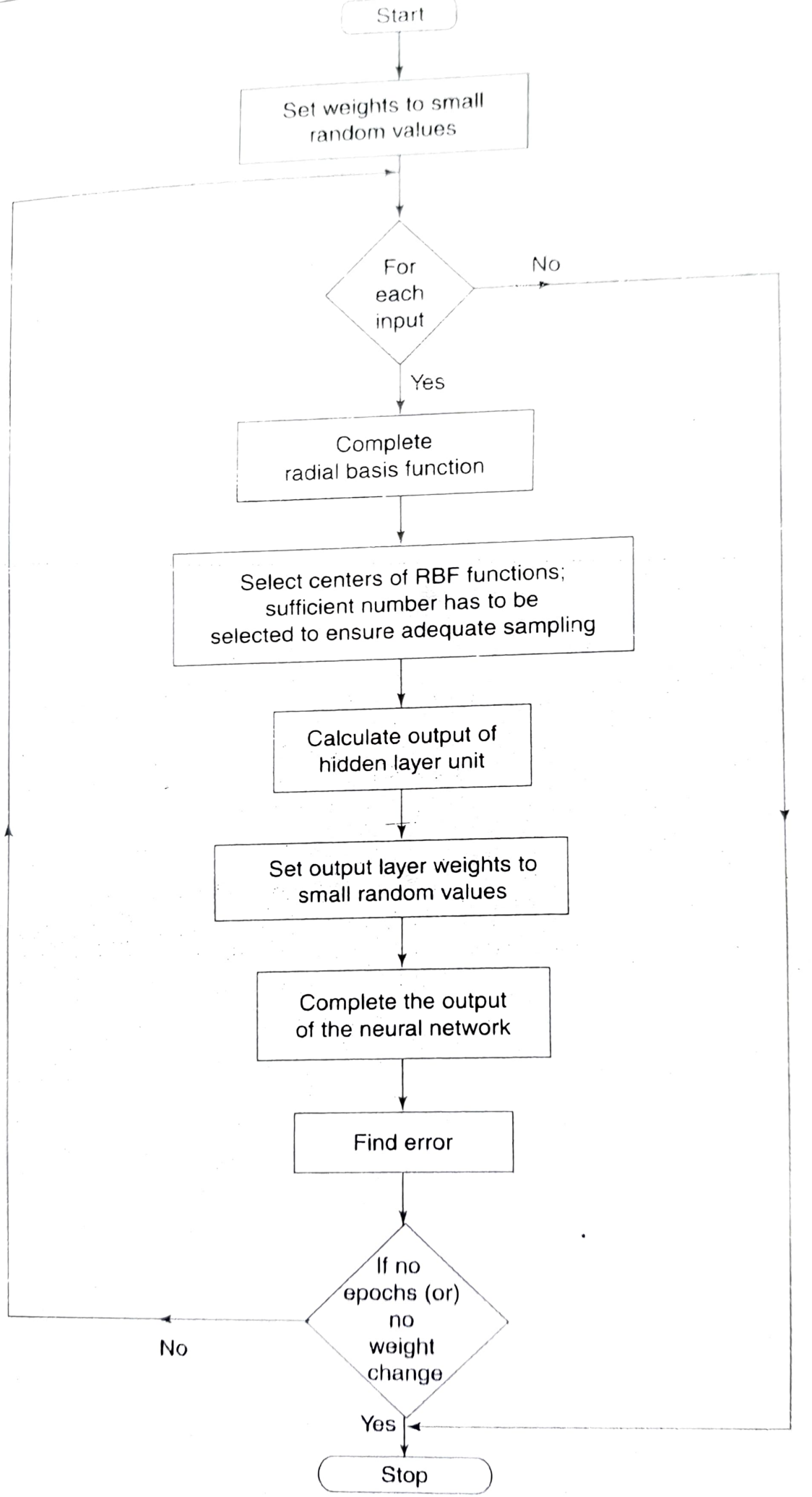

Figure 3-13 Flowchart for