## 2.1 Perceptron networks

### 2.1.1 Theory

Perceptron networks come under single-layer feed-forward networks and are also called simple perceptrons. Various types of perceptrons were designed by Rosenblatt (1962) and Minsky-Papert (1969, 1988).

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), and response unit (output unit).
2. The sensory units are connected to associator units with fixed weights having values 1, 0 or -l, 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 l, 0 or -1. The binary step with fixed threshold ɵ 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

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

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 ca1culated 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 an error has occurred for a particular training patterns .i.e..,

$$w_i(new) = w_i(old) + \alpha\, tx_i$$

$$b(new) = b(old) + \alpha\, t$$

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-l 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.

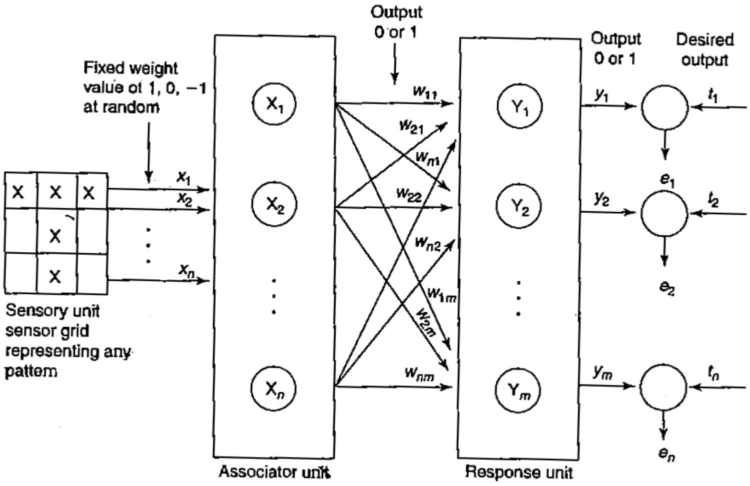


**Figure 2.1: Original perceptron network**

A Perceptron network with its three units is shown in above figure. The 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 hardwired 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.

### 2.1.2 Perceptron Learning Rule

Learning signal is the difference between 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.

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

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

If $y \neq t$, then

$$w(new) = w(old) + \alpha \, tx \; (\alpha - learning \; rate)$$

else,

$$w(new) = w(old)$$

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 convergence to a weight vector that gives the correct response for all training patterns, and this learning takes place within a finite number of steps provided that the solution exists”.

### 2.1.3 Architecture

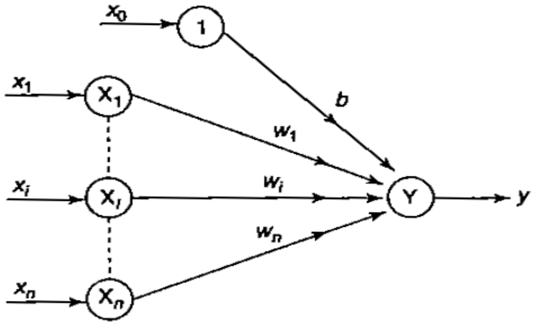


**Figure 2.2: Single classification perceptron network**

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.

### 2.1.4 Flowchart for Training Process

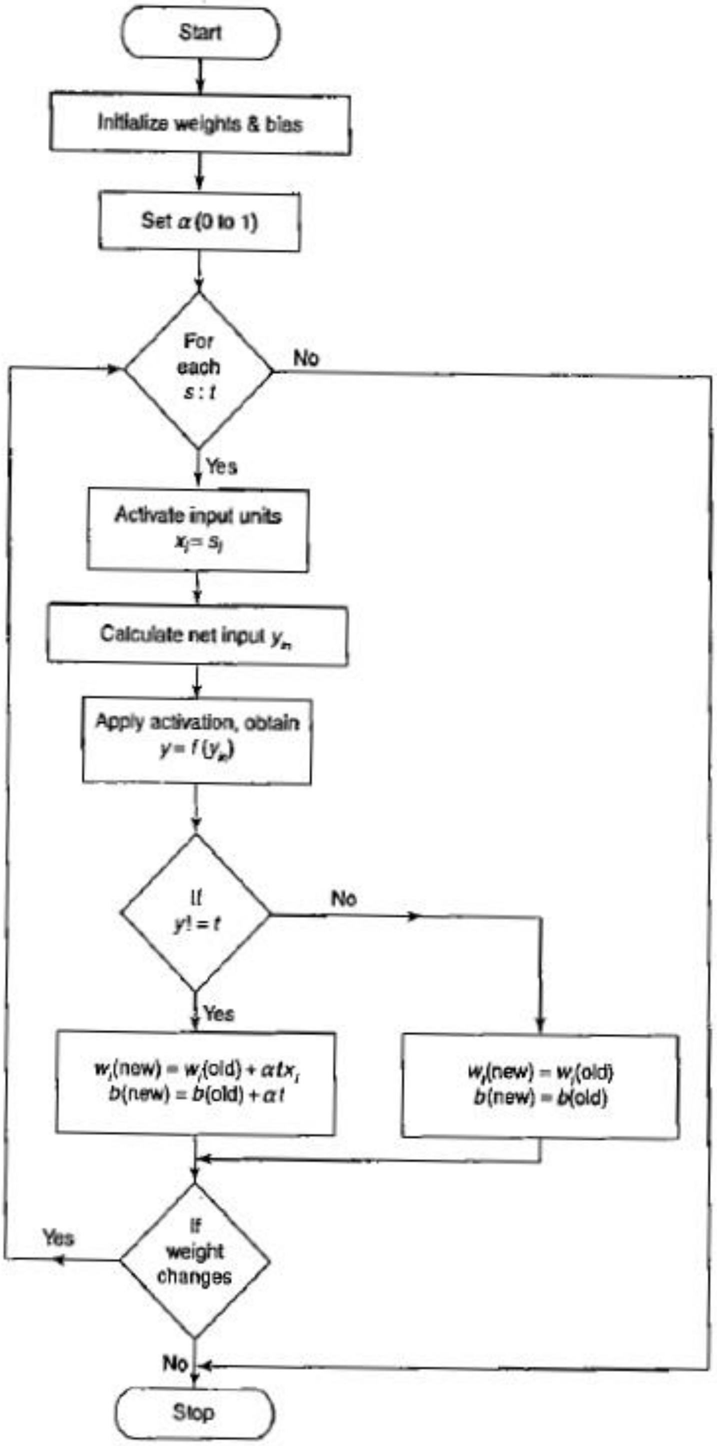

Start
Initialize weights & bias
Set $\alpha$ (0 to 1)
For each $s:t$
No
Yes
Activate input units $x_i = s_i$
Calculate net input $y_{in}$
Apply activation, obtain $y = f(y_{in})$
If $y != t$
No
Yes
$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$
$b(\text{new}) = b(\text{old}) + \alpha t$
$w_i(\text{new}) = w_i(\text{old})$
$b(\text{new}) = b(\text{old})$
If weight changes
Yes
No
Stop

**Figure 2.3: Flowchart for perceptron network with single output**

### 2.1.5 Perceptron Training Algorithm for Single Output Classes

**Step 0**: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate α (0< α ≤ 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:

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

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

If $y \neq t$, then

$$w_i(new) = w_i(old) + \alpha\ t x_i$$

$$b(new) = b(old) + \alpha\ t$$

else,

$$w_i(new) = w_i(old)$$

$$b(new) = b(old)$$

**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.

### 2.1.6 Perceptron Network Testing Algorithm

**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.

**Step 3**: Obtain the response of output unit.

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

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

Thus, the testing algorithm tests the performance of network. In the case of perceptron network, it can be used for linear separability. Here the separating line may be based on the value of threshold that is, the threshold used in the activation function must be a non negative value.

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

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

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

$$w_1x_1 + w_2x_2 + b < -\theta$$

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

## 2.2 Adaptive Linear Neuron

### 2.2.1 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.

### 2.2.2 Delta Rule for Single Output Unit

The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient descendent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of leaning 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 *i* th pattern (i = 1 to n) is

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

Where $\Delta w_i$ 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., $y_{in} = \sum_{i=1}^{n} x_i w_i$ ; t the target output. The delta rule in case of several output units for adjusting the weight from *i*th input unit to the *j*th output unit (for each pattern) is

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

### 2.2.3 Architecture

Adaline is a single unit neuron, which receives input from several units and also from one unit called bias. 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.

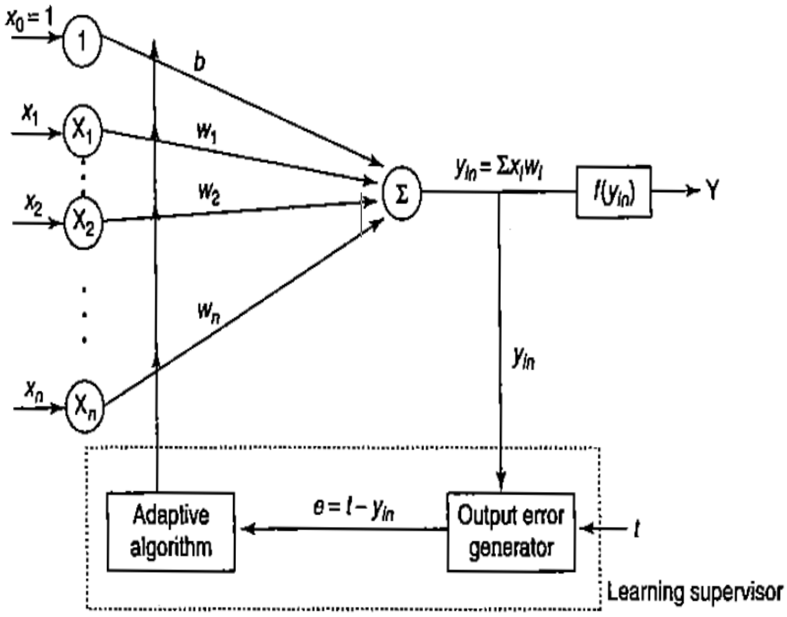


**Figure 2.4: Adaline model**

### 2.2.4 Flowchart for Training Process

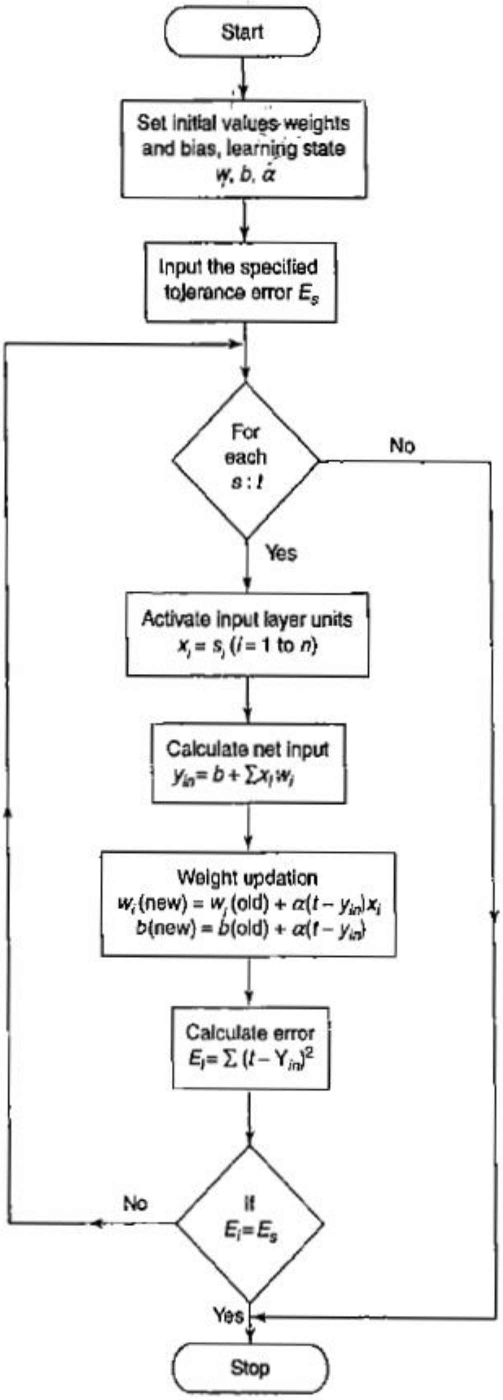

Start
Set initial values weights and bias, learning state $w, b, \alpha$
Input the specified tolerance error $E_s$
For each $s:t$
No
Yes
Activate input layer units $x_i = s_i$ ($i$ = 1 to $n$)
Calculate net input $y_{in} = b + \sum x_i w_i$
Weight updation $w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$ $b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$
Calculate error $E_i = \sum (t - Y_{in})^2$
If $E_i = E_s$
No
Yes
Stop

**Figure 2.5: Flowchart for Adaline training process**

### 2.2.5 Training Algorithm

**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

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

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

**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 rest for stopping condition of a network.

### 2.2.6 Testing Algorithm

**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:

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

## 2.3 Back propagation Network

### 2.3.1 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 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. This is a methods were error is propagated back to the hidden unit. Back propagation network is a training algorithm.

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.

### 2.3.2 Architecture

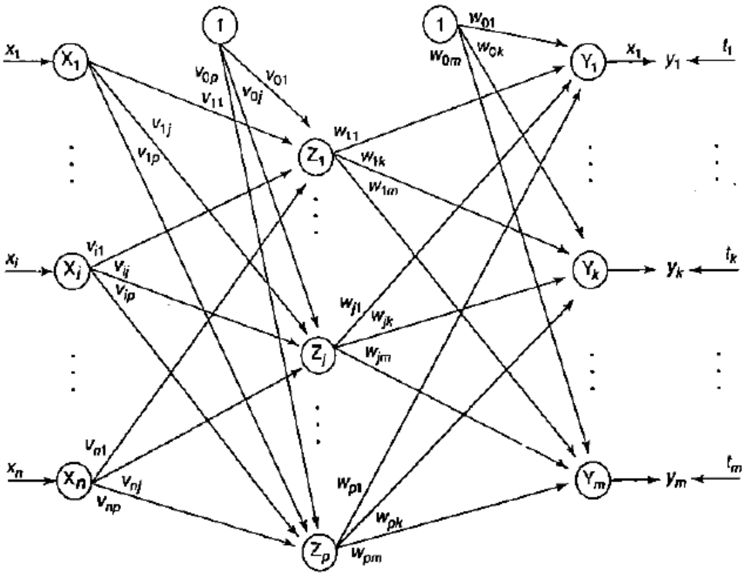


**Figure 2.6: Architecture of a back propagation network**

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. During the back propagation phase of learning, signals are sent in the reverse direction. The inputs 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.

### 2.3.3 Flowchart

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.)
$v_{0i}$ = 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

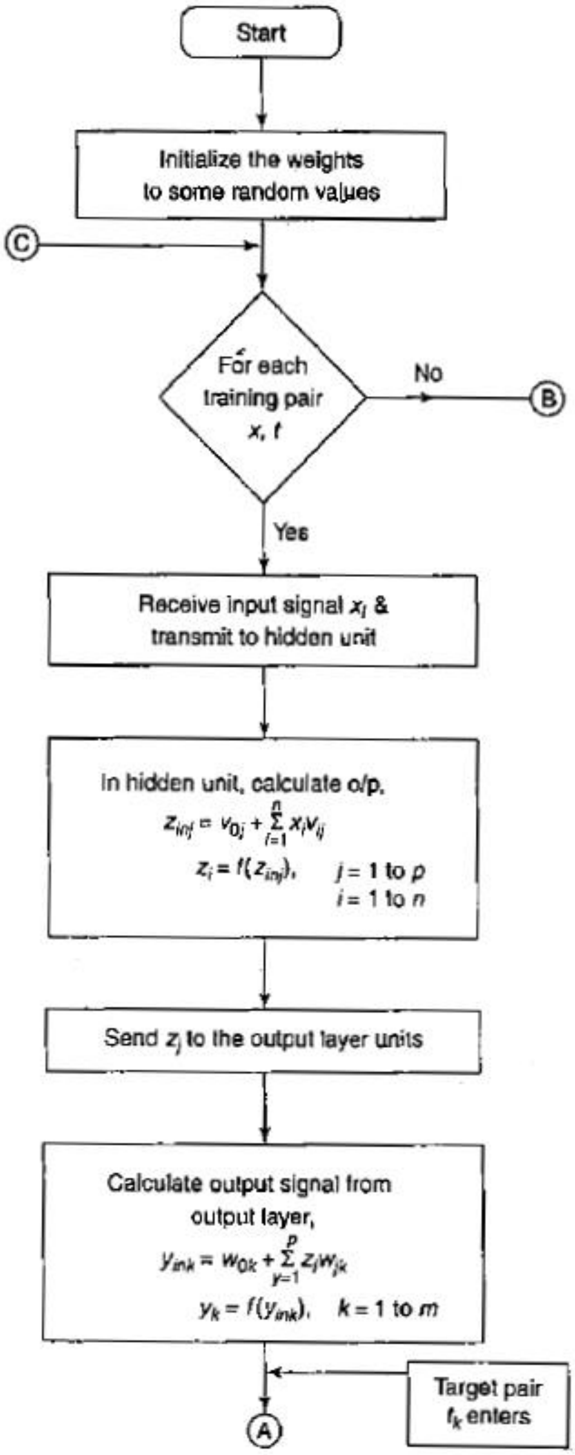
$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and the output is

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

$\delta_k$ = error correction weight adjustment for $w_{jk}$ that is due to an error in unit $y_k$, which is back-propagated to the hidden units that feed into unit $y_k$
$\delta_j$ = error correction weight adjustment for $v_{ij}$ that is due to the back-propagation of error to the hidden unit is $z_{j.}$

The commonly used activation functions are binary, sigmoidal and bipolar sigmoidal activation functions. These functions are used in the BPN because of the following characteristics: (i) Continuity (ii) Differentiability iii) Non decreasing monotonic.


Start
Initialize the weights to some random values
C
For each training pair x, t
No
B
Yes
Receive input signal $x_i$ & transmit to hidden unit
In hidden unit, calculate o/p,
$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$
$z_j = f(z_{inj})$, $j = 1$ to $p$
$i = 1$ to $n$
Send $z_j$ to the output layer units
Calculate output signal from output layer,
$y_{ink} = w_{0k} + \sum_{y=1}^{p} z_j w_{jk}$
$y_k = f(y_{ink})$, $k = 1$ to $m$
Target pair $t_k$ enters
A

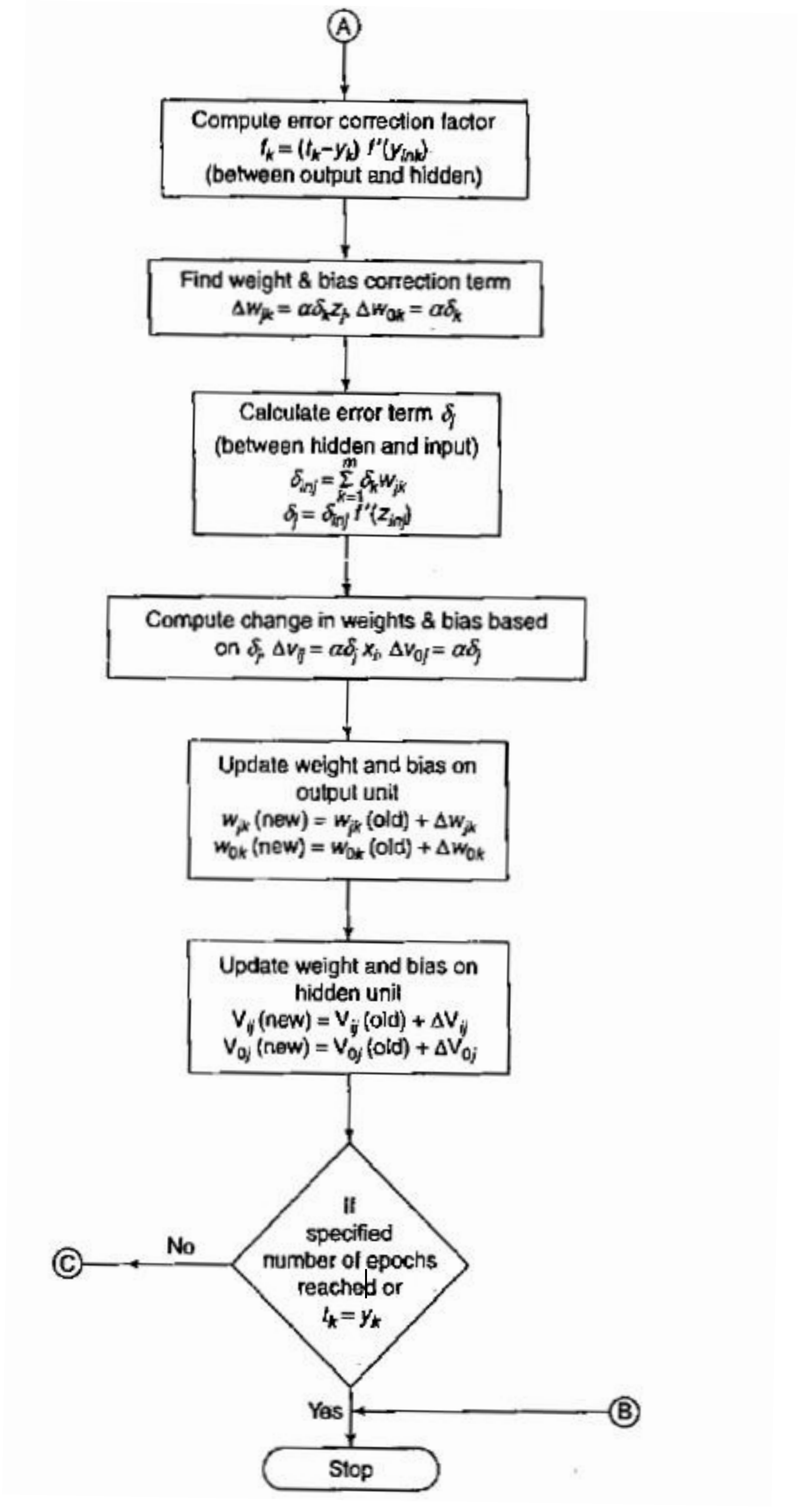

A
Compute error correction factor
$f_k = (t_k - y_k)\, f'(y_{ink})$
(between output and hidden)
Find weight & bias correction term
$\Delta w_{jk} = \alpha\delta_k z_j,\ \Delta w_{0k} = \alpha\delta_k$
Calculate error term $\delta_j$
(between hidden and input)
$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$
$\delta_j = \delta_{inj}\, f'(z_{inj})$
Compute change in weights & bias based on $\delta_j$, $\Delta v_{ij} = \alpha\delta_j x_i$, $\Delta v_{0j} = \alpha\delta_j$
Update weight and bias on output unit
$w_{jk}$ (new) = $w_{jk}$ (old) + $\Delta w_{jk}$
$w_{0k}$ (new) = $w_{0k}$ (old) + $\Delta w_{0k}$
Update weight and bias on hidden unit
$V_{ij}$ (new) = $V_{ij}$ (old) + $\Delta V_{ij}$
$V_{0j}$ (new) = $V_{0j}$ (old) + $\Delta V_{0j}$
If specified number of epochs reached or $t_k = y_k$
No
C
Yes
B
Stop

**Figure 2.7: Flowchart for back - propagation network training**

### 2.3.4 Training 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.

***Feedforward Phase 1***

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

**Step 4**: Each hidden unit $z_j$ *(j* = 1 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* 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 activation function section. 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 $\delta_k$ to the hidden layer backwards.

**Step 7**: Each hidden unit $(z_j$ = 1 to p) sums its delta inputs from the output units:

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

The term $\delta_{inj}$ 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 activation function section depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated $\delta_j$, 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 IIl):***

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

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

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

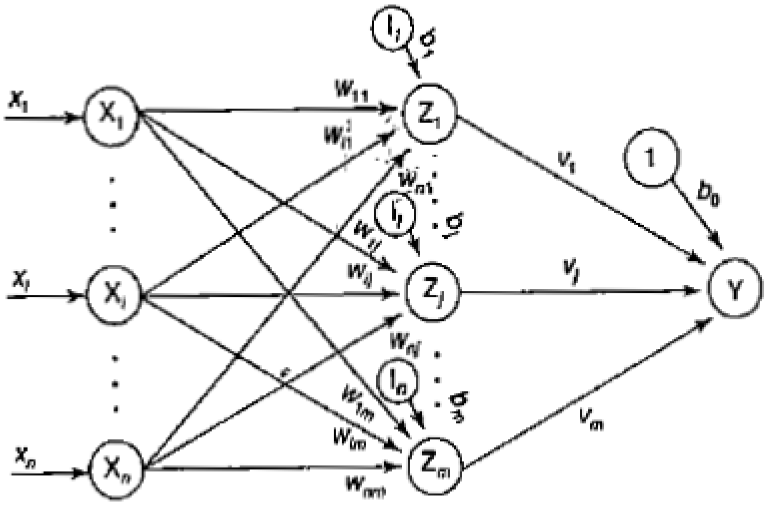
Each hidden unit ($z_{j;}$ j = 1 to p) updates its bias and weights:

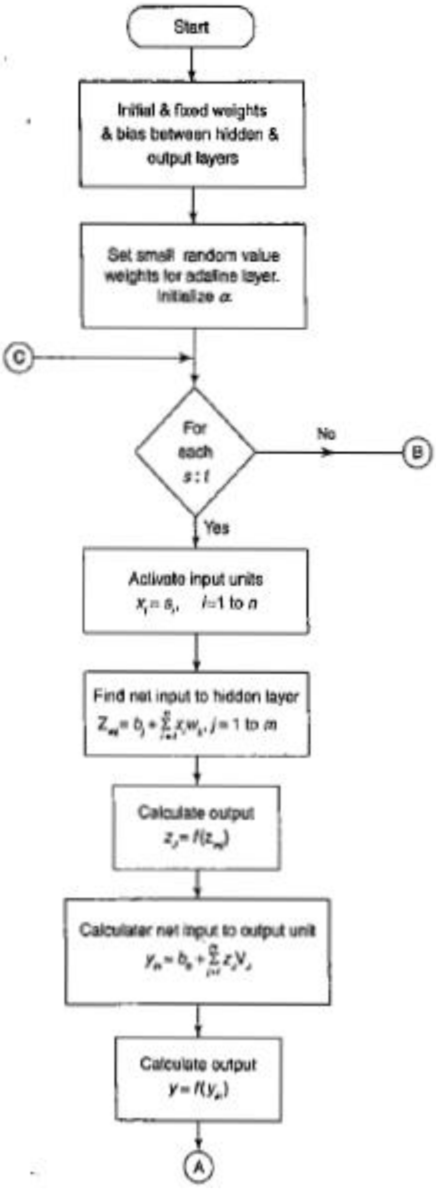
$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$$

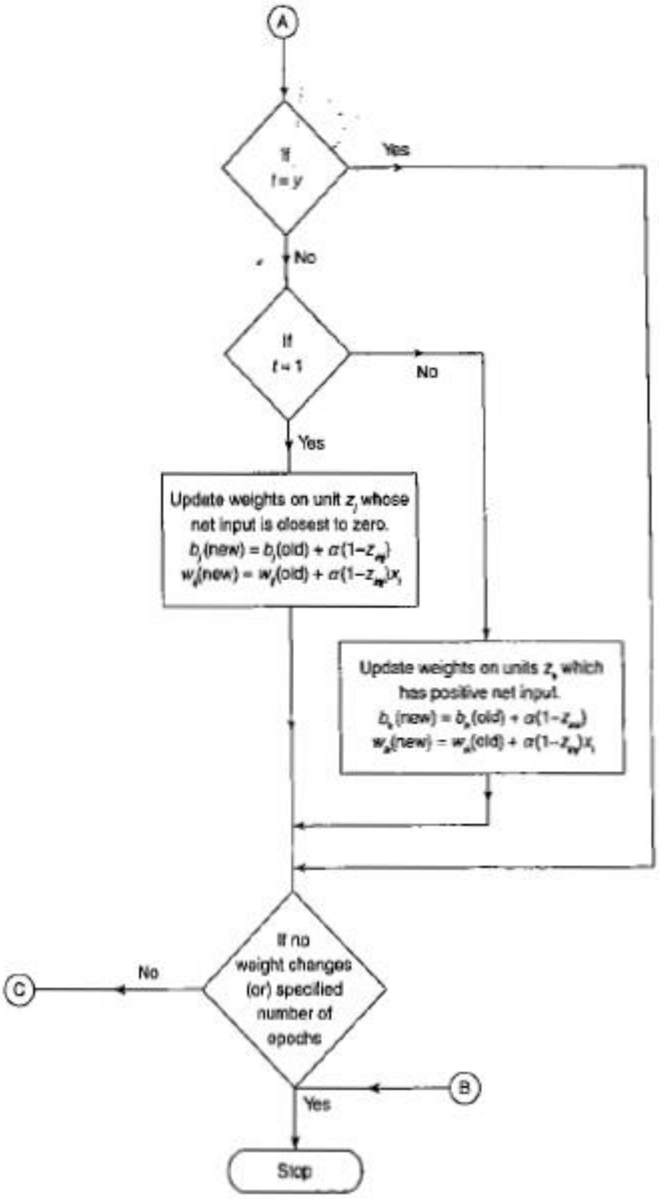
$$v_{0j}(new) = v_{0j}(old) + \Delta v_{0j}$$

**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.

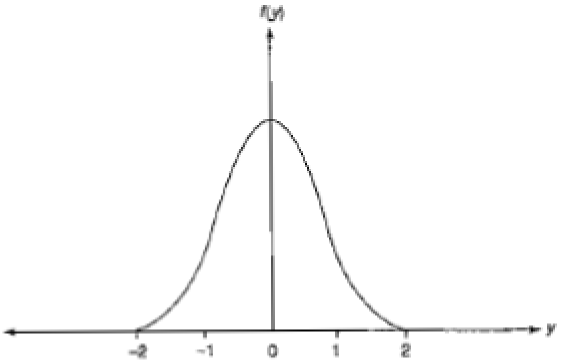
## 2.4 Madaline

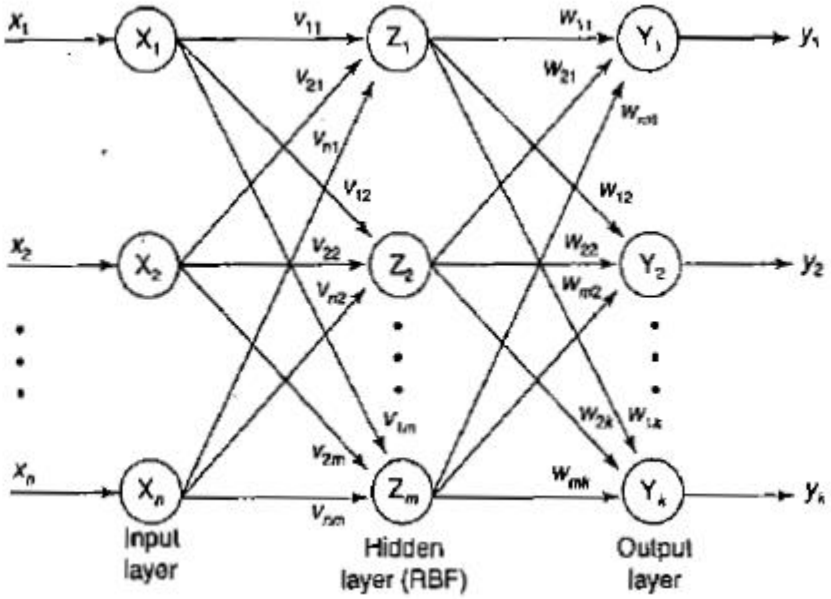

$x_1$
$X_1$
$x_i$
$X_i$
$x_n$
$X_n$
$W_{11}$
$W_{i1}$
$W_{n1}$
$W_{1j}$
$W_{ij}$
$W_{nj}$
$W_{1m}$
$W_{im}$
$W_{nm}$
1
$b_1$
1
$b_j$
1
$b_m$
$Z_1$
$Z_j$
$Z_m$
$v_1$
$v_j$
$v_m$
1
$b_0$
Y


Start
Initial & fixed weights & bias between hidden & output layers
Set small random value weights for adaline layer. Initialize $\alpha$.
C
For each $s:t$
No
B
Yes
Activate input units $x_i = s_i$, $i$=1 to $n$
Find net input to hidden layer $Z_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$, $j$ = 1 to $m$
Calculate output $z_j = f(z_{inj})$
Calculater net input to output unit $y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$
Calculate output $y = f(y_{in})$
A

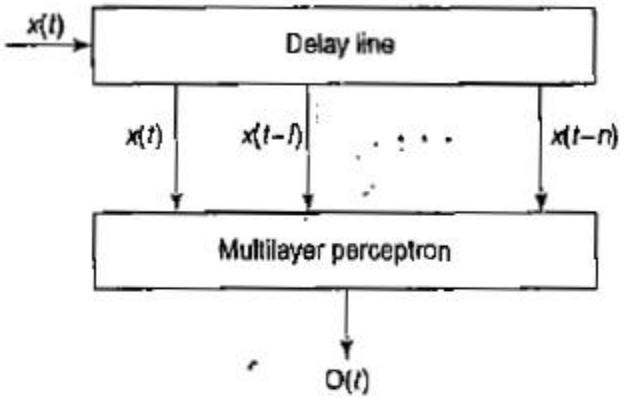

A
If
t = y
Yes
No
If
t = 1
No
Yes
Update weights on unit $z_j$ whose net input is closest to zero.
$b_j$(new) = $b_j$(old) + $\alpha(1-z_{inj})$
$w_{ij}$(new) = $w_{ij}$(old) + $\alpha(1-z_{inj})x_i$
Update weights on units $z_k$ which has positive net input.
$b_k$(new) = $b_k$(old) + $\alpha(1-z_{ink})$
$w_{ik}$(new) = $w_{ik}$(old) + $\alpha(1-z_{ink})x_i$
If no weight changes (or) specified number of epochs
No
C
B
Yes
Stop

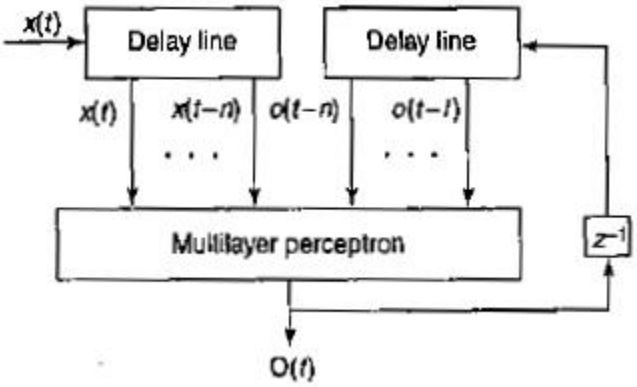
## 2.5 Radial Basis Function Network



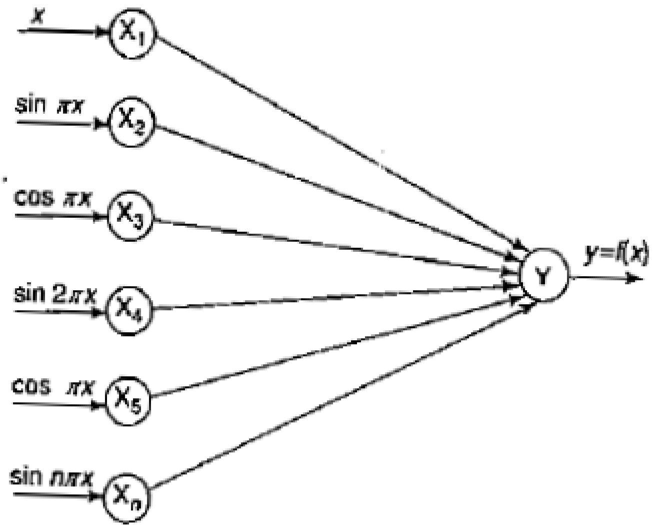


## 2.6 Time Delay Neural Network

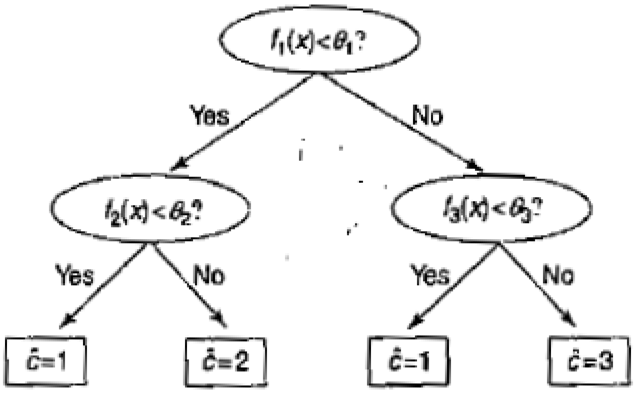




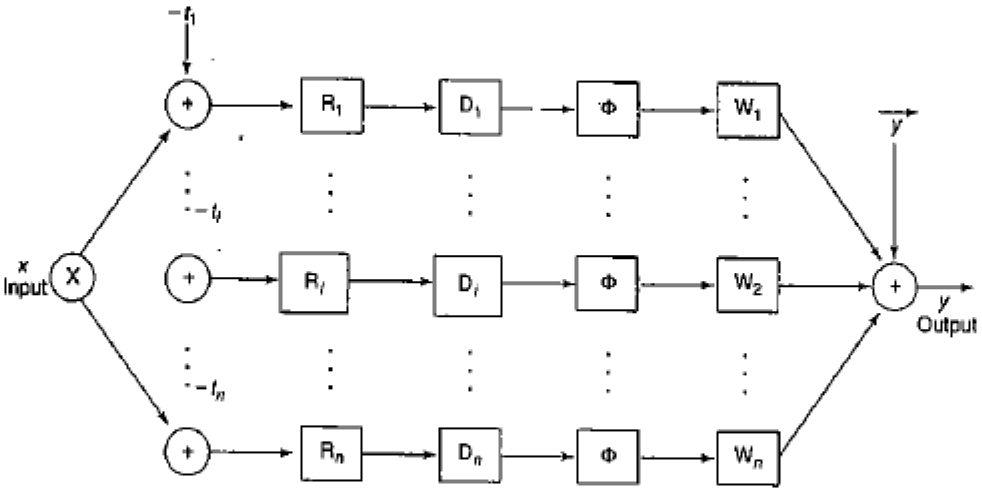
## 2.7 Functional Link Networks



## 2.8 Tree Neural Networks



## 2.9 Wavelet Neural Networks



## 2.10 Advantages of Neural Networks

1. Mimicks human control logic.
2. Uses imprecise language.
3. Inherently robust.
4. Fails safely.
5. Modified and tweaked easily.

## 2.11 Disadvantages of Neural Networks

1. Operator's experience required.
2. System complexity.

## 2.12 Applications of Neural Networks

1. Automobile and other vehicle subsystems, such as automatic transmissions, ABS and cruise control (e.g. Tokyo monorail).
2. Air conditioners.
3. Auto focus on cameras.
4. Digital image processing, such as edge detection.
5. Rice cookers.
6. Dishwashers.
7. Elevators.