The back-propagation learning algorithm is one of the most important developments in neural networks . This network has the scientific and engineering community to the modeling and processing of numerous quantitative phenomena using neural nerworks.

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 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 perceptron is calculating the weights of hidden layers in an efficient way that would result in very small or zero output error. When the hidden layers are increased the network training becomes more complexto update weights ,the error must be calculated.A back propogation 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 units whose activation is always 1.the bias terms also acts as weights.

The below figure shows the Architecture of BPN ,depicting only direction of information flow for feed-forward phase.During back propogation phase of learning,signals are sent in reverse direction.

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

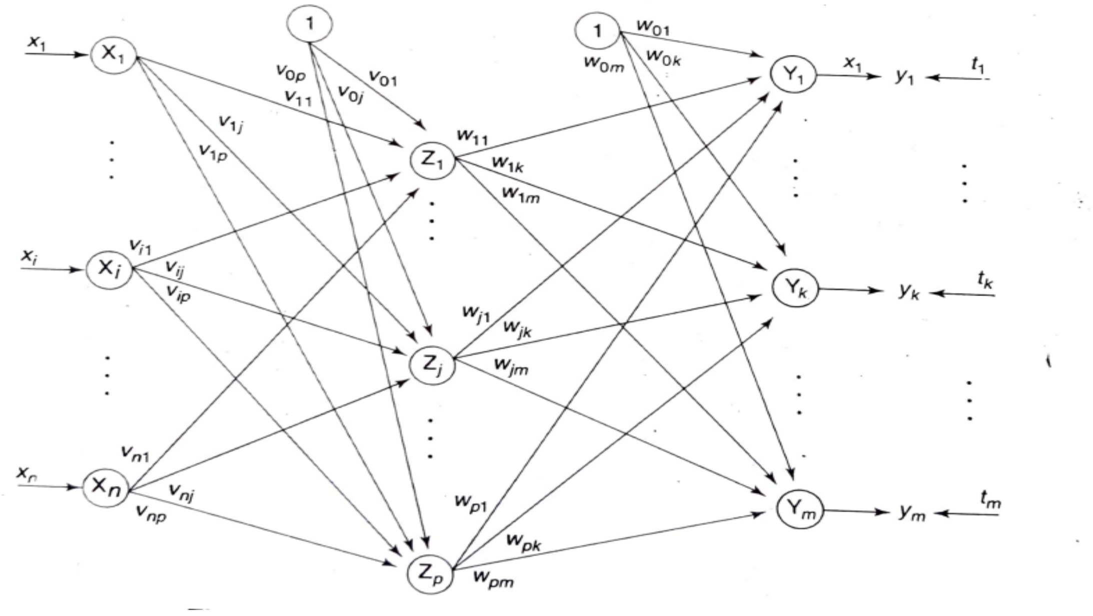
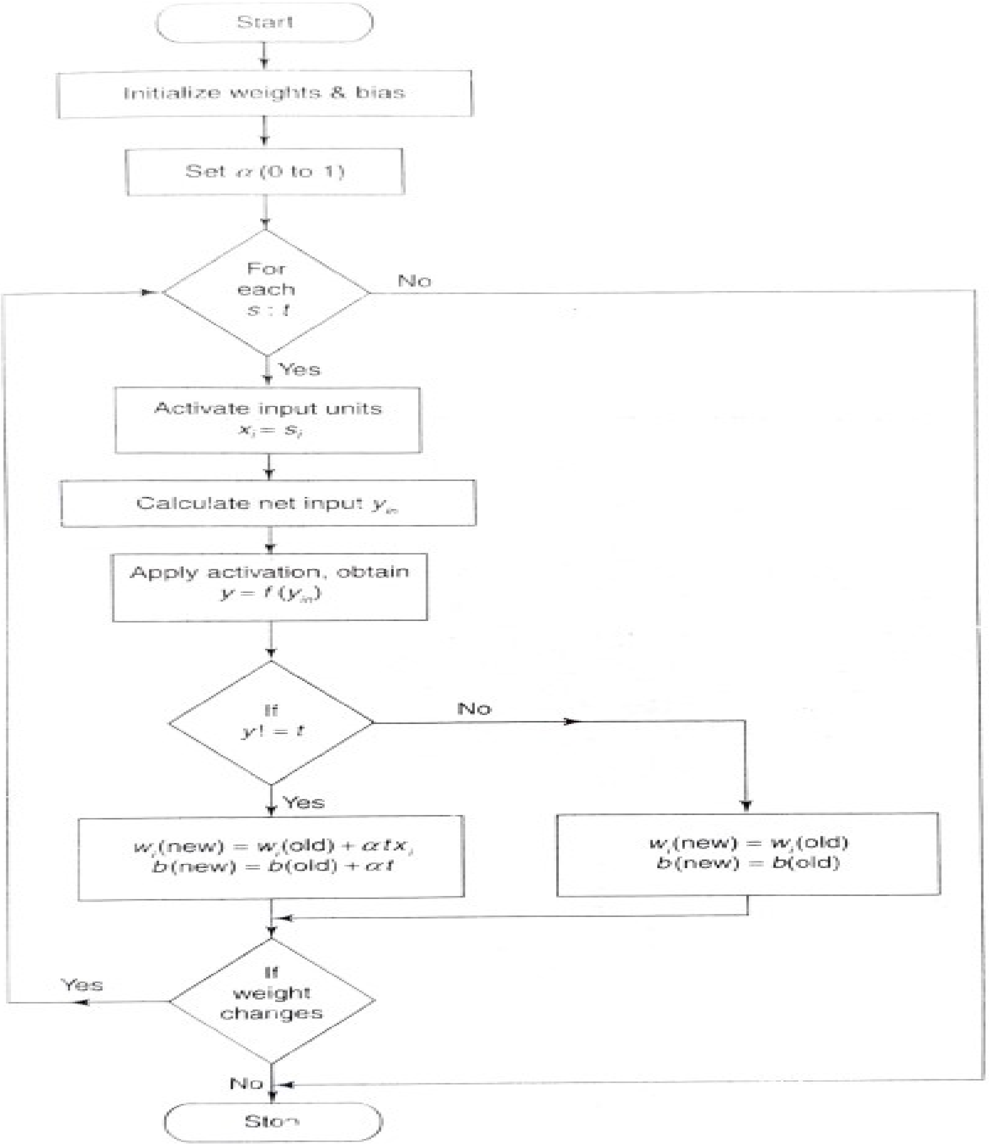


Fig.Architecture of back propogation network

Flowchart:


Start
Initialize weights & bias
Set α (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

Flowchart for perceptron network with single unit

The perceptron algorithm can be used for either binary or bipolar input vectors. Having bipolar targets, threshold being fixed and variable bias.

In this algorithm ,initailly the inputs are assigned. Then input is calculated. The output network is obtained by applying activation function over calculated net input.

Algorithm:

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

: Perform Steps 2-6 until the final stopping condition is false.

: Perforn Steps 3-5 for each training pair indicated by s:t.

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

$x_i = s_i$

Calculate the output of the network. To do so, first obrain 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:

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

: 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

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

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

x= input tranng vector (x_1,... x_j, ..., X_n)

t= target output vector (t_1,... , t_k, ..., 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 zj is

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

and the output is

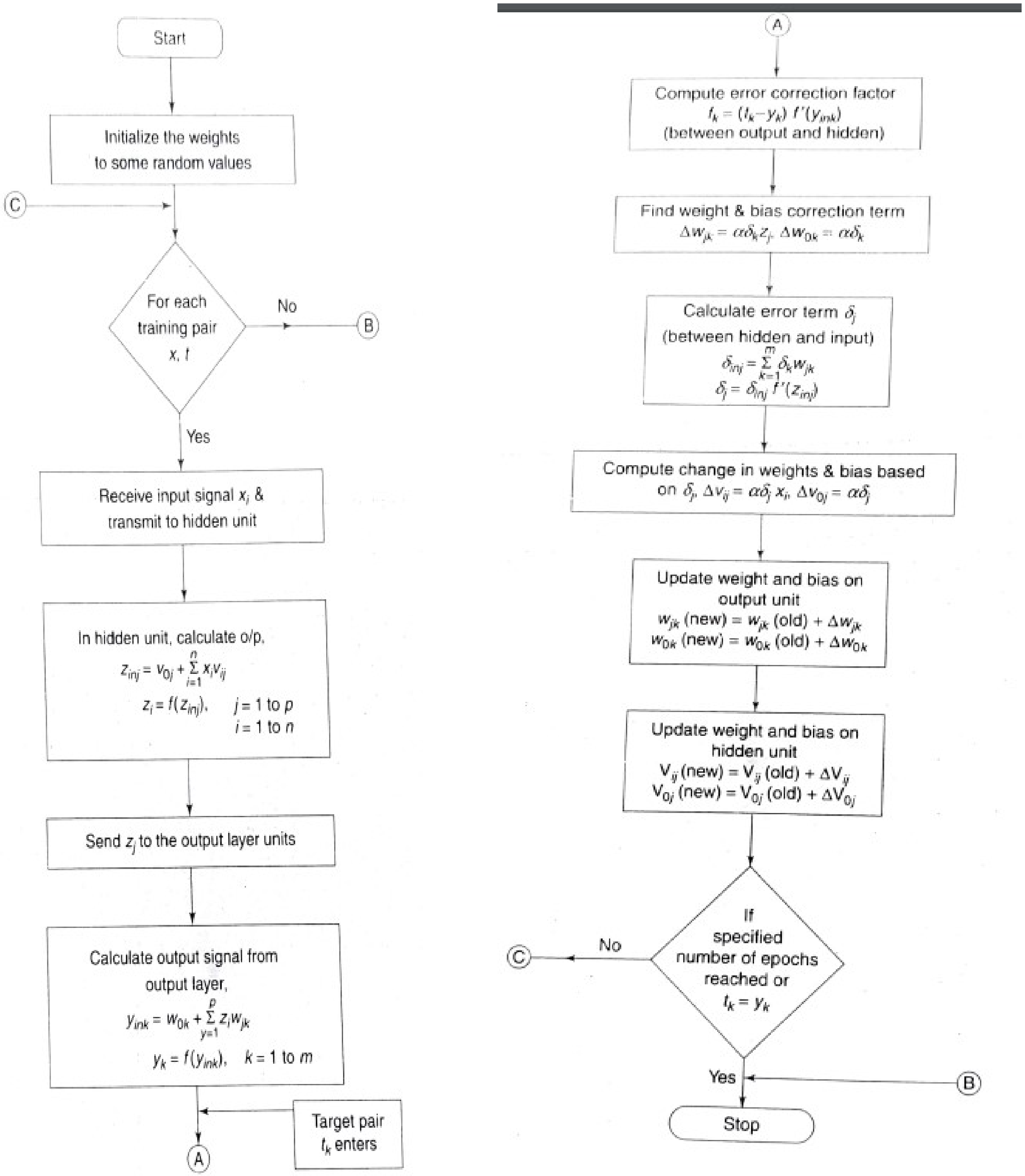
$$z_j = f(z_{inj})$$

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

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


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,
$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$
$z_i = f(z_{inj})$, $j = 1$ to p
$i = 1$ to n
Send z_j to the output layer units
Calculate output signal from output layer,
$y_{ink} = w_{0k} + \sum_{y=1}^{p} z_i w_{jk}$
$y_k = f(y_{ink})$, $k = 1$ 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 δ_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 δ_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) + Δw_{jk}
w_{0k} (new) = w_{0k} (old) + Δw_{0k}
Update weight and bias on hidden unit
V_{ij} (new) = V_{ij} (old) + ΔV_{ij}
V_{0j} (new) = V_{0j} (old) + ΔV_{0j}
If specified number of epochs reached or $t_k = y_k$
No
C
Yes
B
Stop

Flowchart for backpropogation training

Training algorithm for backpropogation:

This algorithmuses the incrementalapproach for updation weights.i.e weights are being changed immediately after training pattern is presented.There is another way of training called batch-mode training,where weights are changed only after all the training patterns are presented.

So, the error back propogation learning algorithm can be outlined in following algorithm:

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

: Perform Steps 2-9 when stopping condition is false.

Perform Steps 3-8 for each training pair.

Each input unit recieves input signals xi and sends it to hidden unit(i= 1 to n).

Each hidden unit zj(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 zinj (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.

For cach output unit ye (k = l 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

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

Each output unit yk(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(amk) 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.

Each hidden unit (zj, j= l top) 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 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated 8;, update the change In weights and bias:

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

: 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:

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

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

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.

L

L α

L

L

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 architecture

:

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 activarion functions used here may get saturated from the beginning itself and the system may be stuck at a local minima at 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 0_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.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 neuron is obtained by equation

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

Where vj is the average weight calculated for ail values of i, and the scale factor

y= $0.7(p)^{1/n}$

L α

The iearning 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.

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 η Ɛ[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

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

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

The momentum factor also helps in faster convergence.

The best network for generalization is BPN, A network is said to be generalized when it sensibly interpolates 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 thc 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 achicved by introducing variations in the input space of a training patterns as part of the training set.However, computationally, this method is very expensive. Also, net with large number of nodes is capable of memorizing the training set at the cost of generaiization.

L

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 with randomly from the set.

L

If there exists more than one hidden layer in a BPN, then the calculations performed for a single layer are the repeated for all the layers and are summed up at the end.In case of all multilayer feed-forward networks the size of 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 reasonable size, the size of hidden nodes has to be only a relatively small fraction of the input layer.

Radial basis function (RBF) is a classification and functional approximation neural network developed by M.J.D.Poweli.

The network uses the most common nonlinearities such as sigmoidal and Guassian kernel function.The Guassian function are also used in regularization networks .The response of such a function is positive for all values of y, the response decrese to 0 as

$|y| \longrightarrow 0.$

The Guassian Function is generally defined as

$$f(y)=e^{-y2}$$

The derivation of this function is given by

$$f'(y)=-2_{ye}{}^{-y2}=-2yf(y)$$

The graphical representation of Guassian function is shown below

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

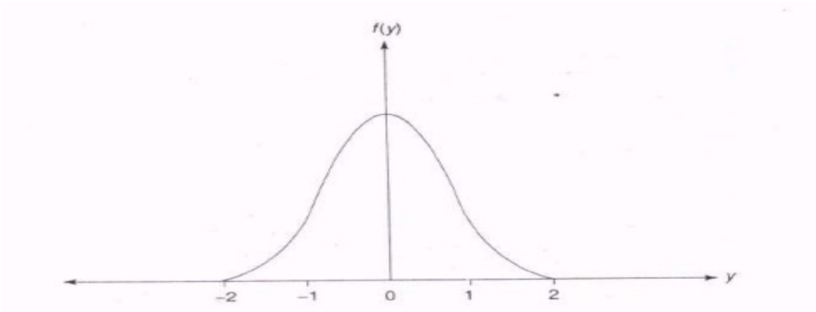
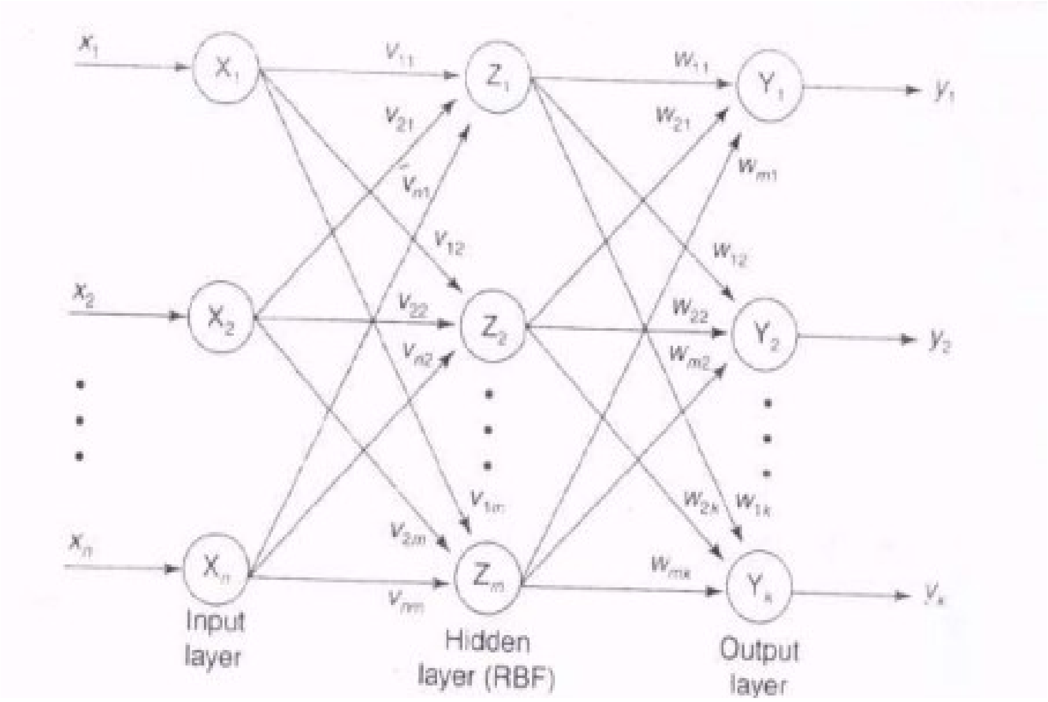


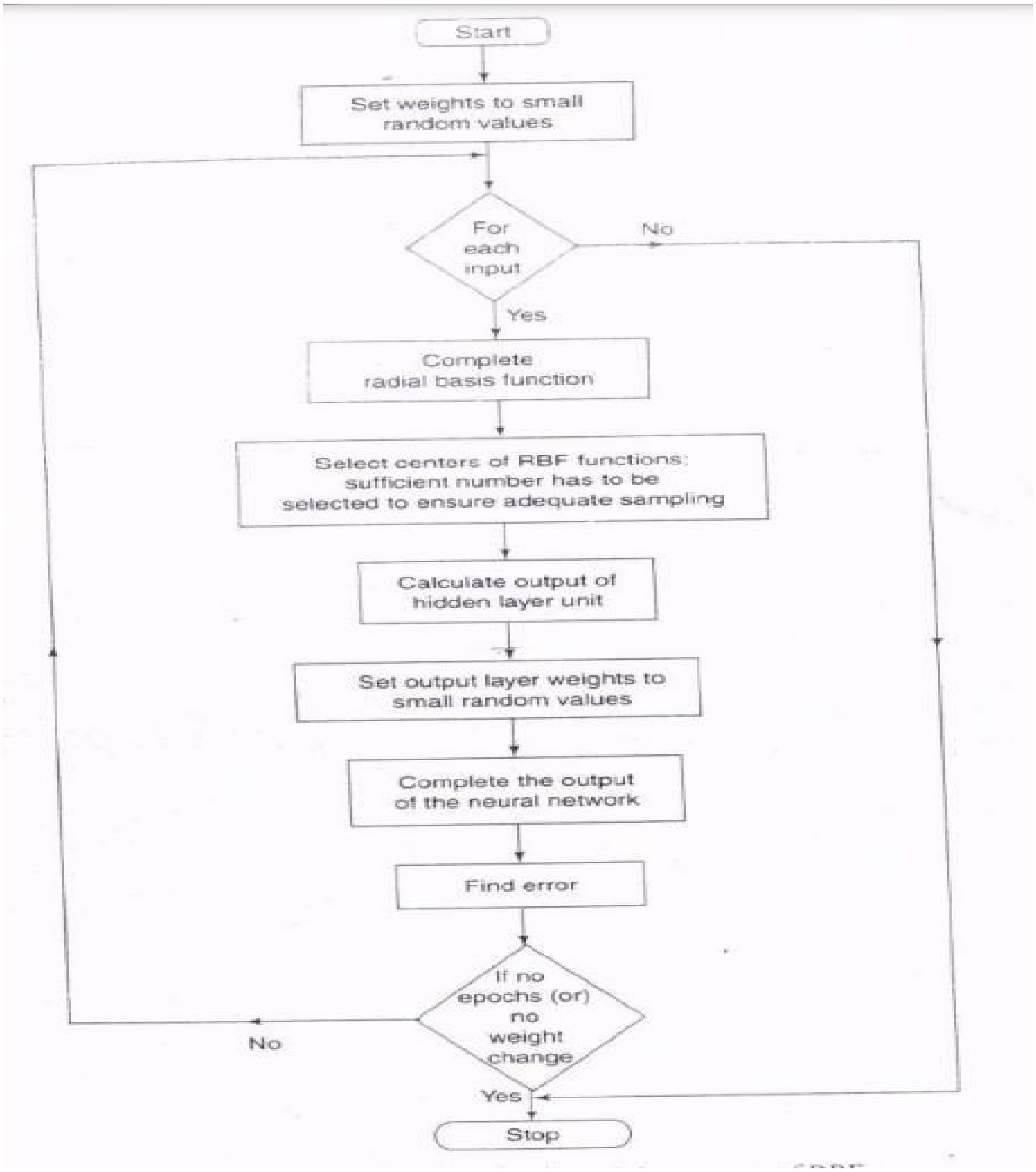
Fig Guassian Kernel function



Architecture of Radial basis function network(RBFN)

The Architecture consists of two layers whose output nodes form a linear combination of the kernel functions computed by means of RBF nodes or hidden layer nodes. The basis function in hidden layer produces significsnt nonzero response to the input stimulus it has received only when input of it falls within small localized region of input space .those network also called as localized recieptive field network.

The training algorithm describes in detail all calculations involved in training process depicted in flowchart.


Start
Set weights to small random values
For each input
No
Yes
Complete radial basis function
Select centers of RBF functions; sufficient number has to be selected to ensure adequate sampling
Calculate output of hidden layer unit
Set output layer weights to small random values
Complete the output of the neural network
Find error
If no epochs (or) no weight change
No
Yes
Stop

The training is continued in the output layer with a supervised learning algorithm.simultaneously , supervised learning algorithm can apply to the hidden and output layers for fine running of network.

So the traing algorithm given as follows -

: Set the weights to small random values.

: Perform steps 2-8 when the stooping condition is false.

:Perform step 3-7 for each input.

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

: calculate the radial basis function

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

: Calculate the output from hidden layer unit.

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

Where $\mathbf{x}_{ji}$ is the centre of RBF unit for input variables, σ_i the width of *i*th RBF unit,x_{ji} the *j*th variable of input pattern.

: Calculate the output of the neural network:

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

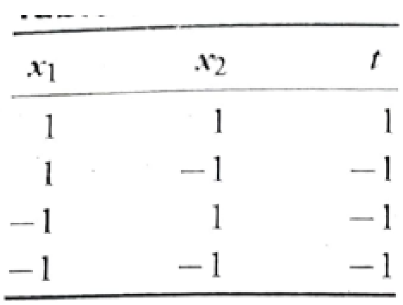
Where k is number of hidden layer nodes(RBF function),y_{net} the output value of *m*th node in output layer for *n*th incoming pattern,wim the weight between ith RBF unit and *m*th output node ,*w*0 the biasing term at nth output node.

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

L

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

L



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. 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., w1 = b= 0 and θ = 0. The learning rate α is set equal to 1.

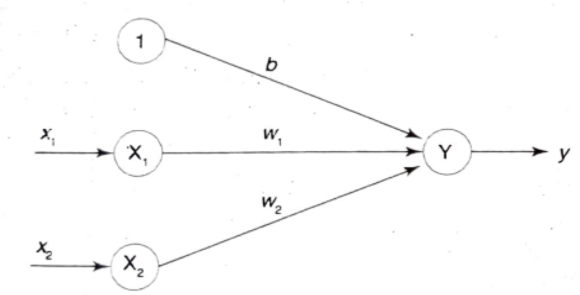


Fig: Perceptron network for AND function.

For the first input pattern, x1 = 1, x2= 1 and t=1 with weights and bias, w1= 0, w2= 0 and b=0

- Calculate the net input

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

- 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 θ =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 w1 = 1, w2 = 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 shows the training of perceptron network until its target and calculated output converge for all the patterns.

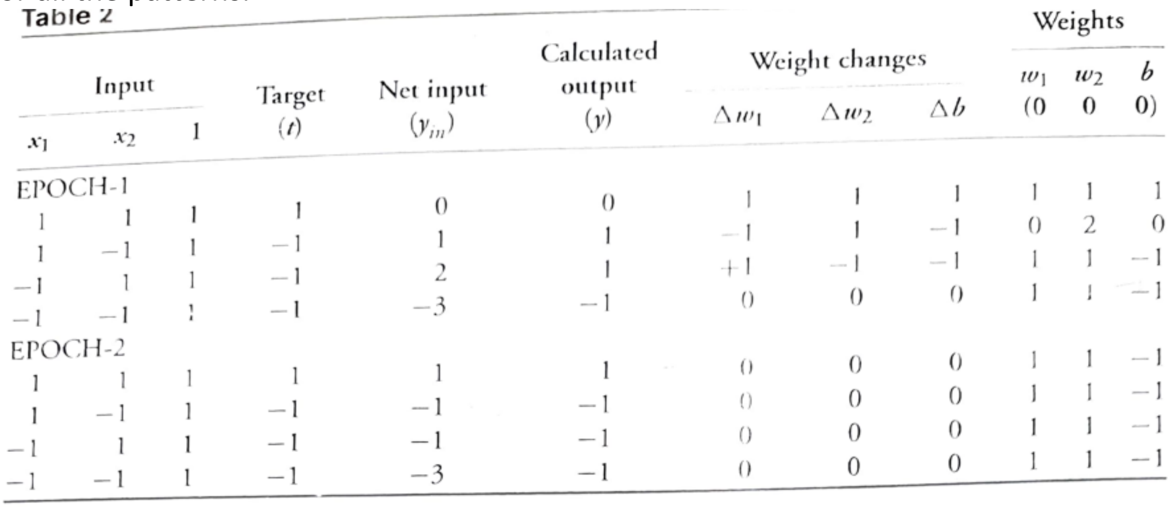


Table 2

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

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. The same methodology can be applied for implementing other logic functions such as OR, AND NOT, NAND, etc. If there exists a threshold value θ≠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.

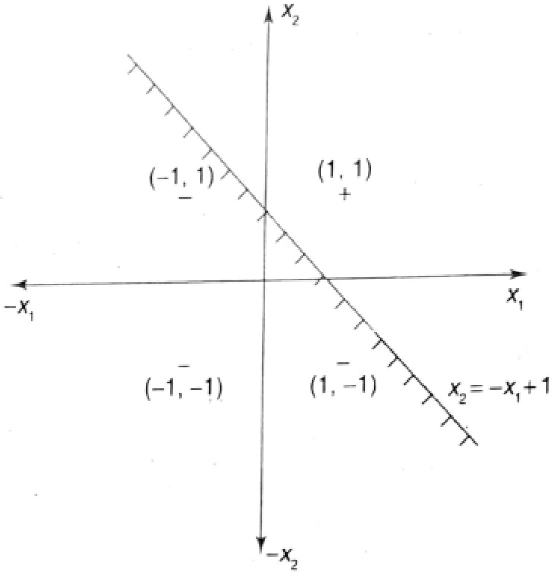
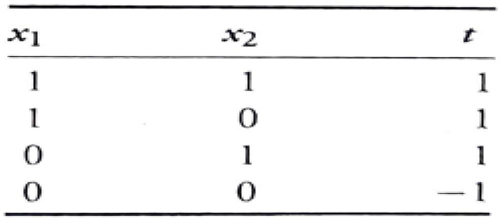


Fig: Decision boundary for AND function in perceptron training(θ=0)

L **L**

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



x_1	x_2	t
1	1	1
1	0	1
0	1	1
0	0	−1

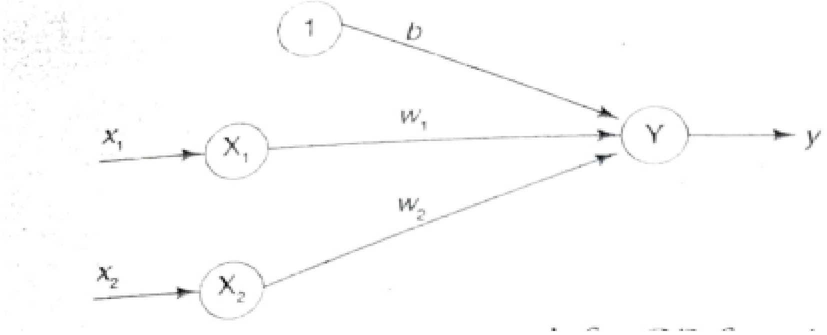


Fig : Perceptron network for OR Function

The perceptron network, which uses perceptron learning rule, is used to train the OR function. The network architecture is shown in Figure.

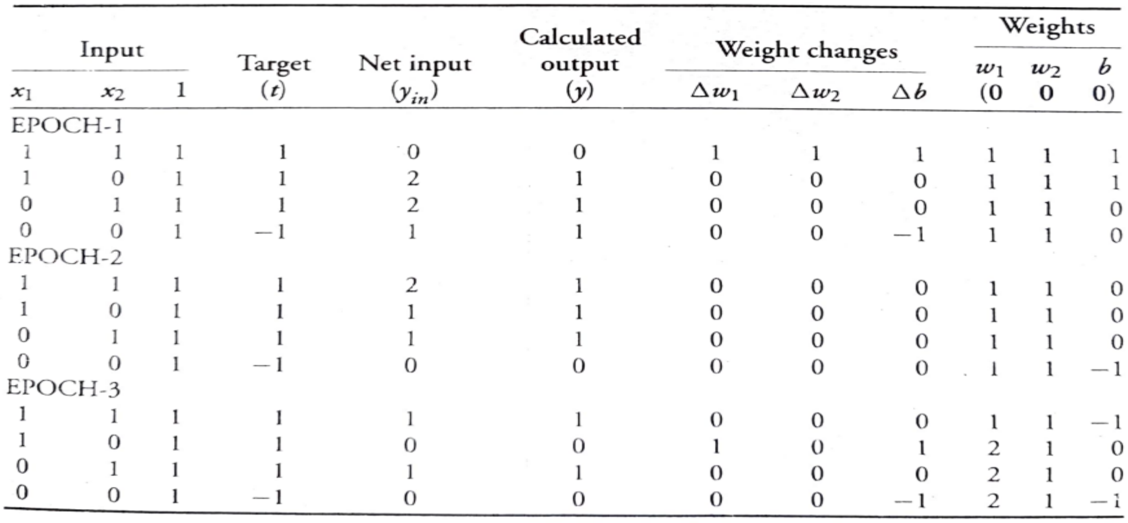
The initial values of the weights and bias are taken as zero,i.e.,

$$w_1 = w_2 = b = 0$$

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.



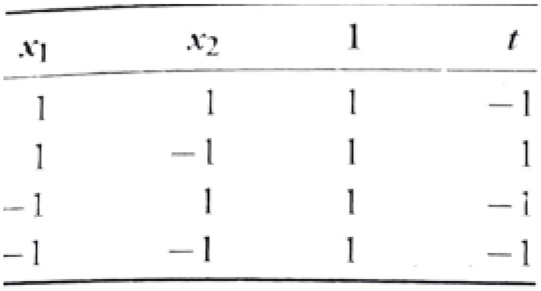
Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	0	1	1	2	1	0	0	0	1	1	1
0	1	1	1	2	1	0	0	0	1	1	0
0	0	1	−1	1	1	0	0	−1	1	1	0
EPOCH-2											
1	1	1	1	2	1	0	0	0	1	1	0
1	0	1	1	1	1	0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

The final weights at the end of third epoch are Further epochs have to be done for the convergence of the network.

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

L

The truth table for ANDNOT function with bipolar inputs and targets is shown in Table



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 w1= w2 = b= 0.2, and α=0.2. For the first input sample x1= 1, x2=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- yin) = (-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

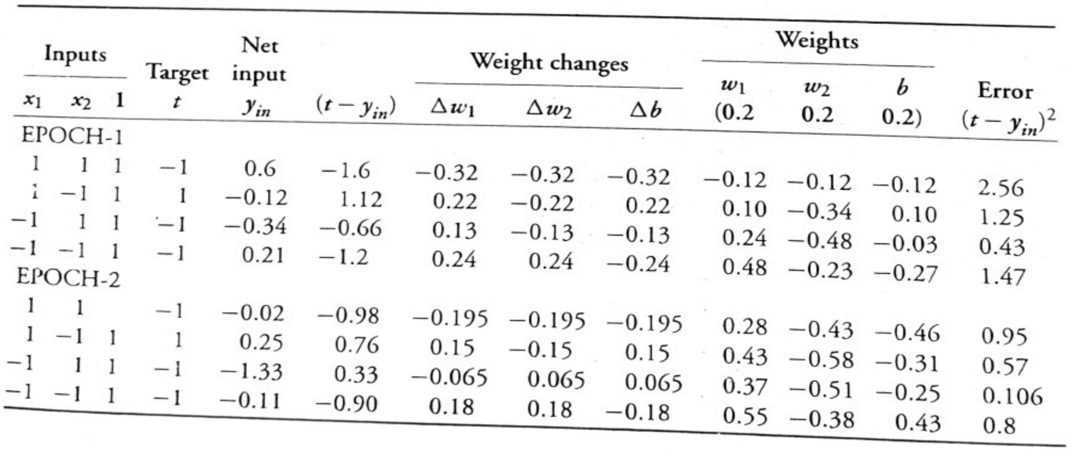
$$w_1(\text{new}) = w_1(\text{old}) + \alpha(t - y_{in})x_1$$
$$= 0.2 + 0.2 \times (-1.6) \times 1 = -0.12$$

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

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

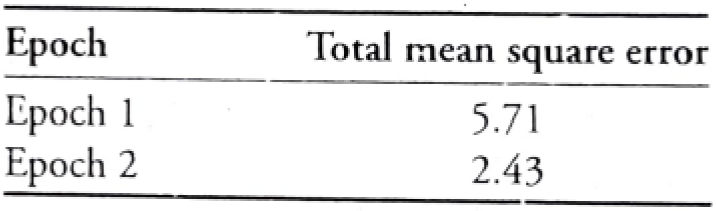
Now we compute the error,

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



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

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



Epoch	Total mean square error
Epoch 1	5.71
Epoch 2	2.43

Hence from above Table, it is clearly understood that he 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.

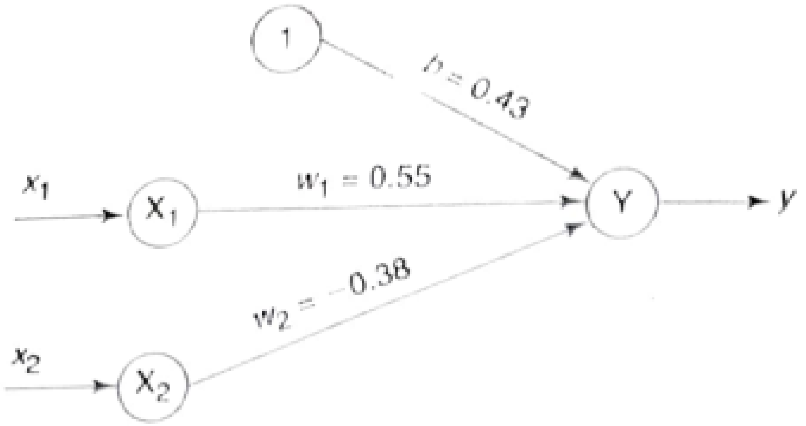


Fig:Nework architecture for ANDNOT funcion using Adaline nctwork.

L

α L

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 [w1 w2 w0] = [0.4 0.1 -0.2], and the learning rate is α= 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 [x1,x2] = [-1, 1] and target t=1:

- Calculate the net input: For z_1 layer

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

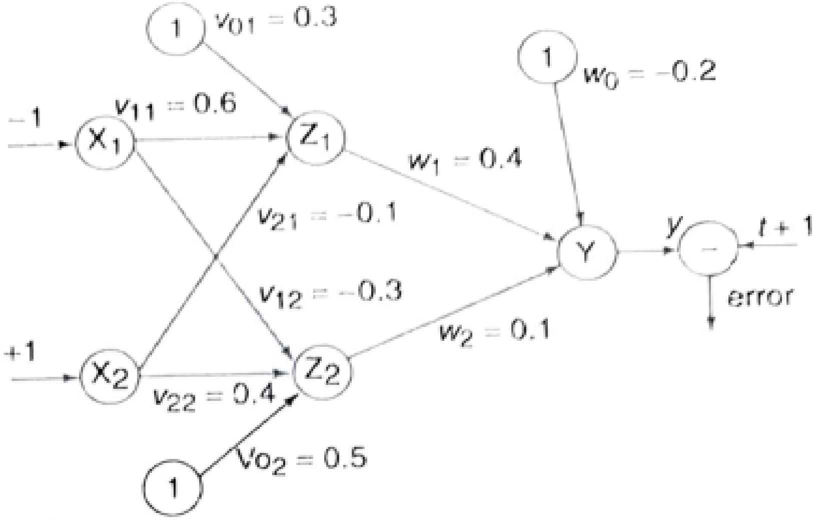


Fig: Network

For z_2 layer

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

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

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

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

$$\begin{aligned} f'(y_{in}) &= 0.5[1 + f(y_{in})]\,[1 - f(y_{in})] \\ &= 0.5[1 - 0.1122][1 + 0.1122] = 0.4937 \end{aligned}$$

This implies

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

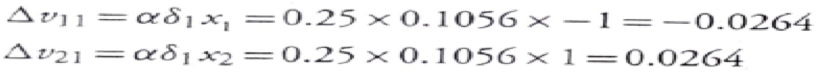
Find the changes in weights between hidden and output layer:

$$\begin{aligned} \Delta w_1 &= \alpha \delta_1 z_1 = 0.25 \times 0.5491 \times -0.1974 \\ &= -0.0271 \\ \Delta w_2 &= \alpha \delta_1 z_2 = 0.25 \times 0.5491 \times 0.537 = 0.0737 \\ \Delta w_0 &= \alpha \delta_1 = 0.25 \times 0.5491 = 0.1373 \end{aligned}$$

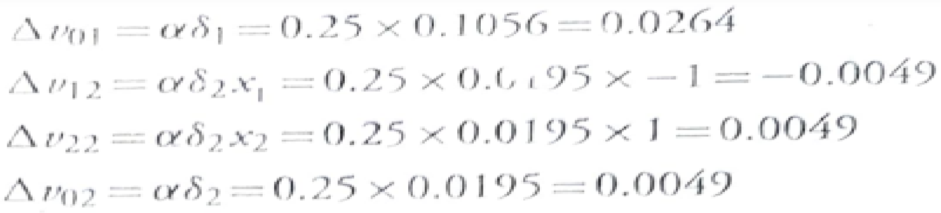
- Compute the error portion δ_j, between input and hidden layer (j = I to 2):

$$\begin{aligned} \delta_j &= \delta_{inj} f'(z_{inj}) \\ \delta_{inj} &= \sum_{k=1}^{m} \delta_k w_{jk} \\ \delta_{inj} &= \delta_1 w_{j1} \qquad [\because \text{only one output neuron}] \\ \Rightarrow \delta_{in1} &= \delta_1 w_{11} = 0.5491 \times 0.4 = 0.21964 \\ \Rightarrow \delta_{in2} &= \delta_1 w_{21} = 0.5491 \times 0.1 = 0.05491 \\ \text{Error, } \delta_1 &= \delta_{in1} f'(z_{in1}) = 0.21964 \times 0.5 \\ &\quad \times (1 + 0.1974)(1 - 0.1974) = 0.1056 \\ \text{Error, } \delta_2 &= \delta_{in2} f'(z_{in2}) = 0.05491 \times 0.5 \\ &\quad \times (1 - 0.537)(1 + 0.537) = 0.0195 \end{aligned}$$

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

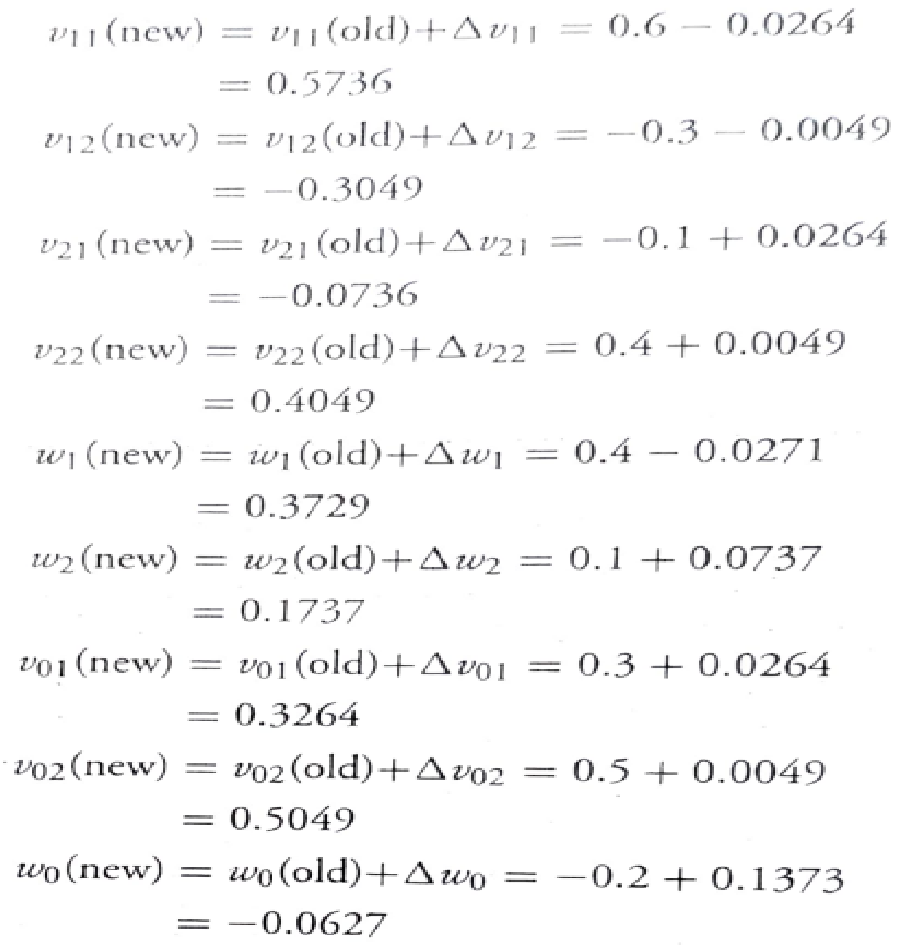


$$\Delta v_{11} = \alpha\delta_1 x_1 = 0.25 \times 0.1056 \times -1 = -0.0264$$
$$\Delta v_{21} = \alpha\delta_1 x_2 = 0.25 \times 0.1056 \times 1 = 0.0264$$



$$\Delta v_{01} = \alpha\delta_1 = 0.25 \times 0.1056 = 0.0264$$
$$\Delta v_{12} = \alpha\delta_2 x_1 = 0.25 \times 0.0195 \times -1 = -0.0049$$
$$\Delta v_{22} = \alpha\delta_2 x_2 = 0.25 \times 0.0195 \times 1 = 0.0049$$
$$\Delta v_{02} = \alpha\delta_2 = 0.25 \times 0.0195 = 0.0049$$

- Compute the final weights of the network:



$$\begin{aligned}
v_{11}(\text{new}) &= v_{11}(\text{old}) + \Delta v_{11} = 0.6 - 0.0264 \\
&= 0.5736 \\
v_{12}(\text{new}) &= v_{12}(\text{old}) + \Delta v_{12} = -0.3 - 0.0049 \\
&= -0.3049 \\
v_{21}(\text{new}) &= v_{21}(\text{old}) + \Delta v_{21} = -0.1 + 0.0264 \\
&= -0.0736 \\
v_{22}(\text{new}) &= v_{22}(\text{old}) + \Delta v_{22} = 0.4 + 0.0049 \\
&= 0.4049 \\
w_1(\text{new}) &= w_1(\text{old}) + \Delta w_1 = 0.4 - 0.0271 \\
&= 0.3729 \\
w_2(\text{new}) &= w_2(\text{old}) + \Delta w_2 = 0.1 + 0.0737 \\
&= 0.1737 \\
v_{01}(\text{new}) &= v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.0264 \\
&= 0.3264 \\
v_{02}(\text{new}) &= v_{02}(\text{old}) + \Delta v_{02} = 0.5 + 0.0049 \\
&= 0.5049 \\
w_0(\text{new}) &= w_0(\text{old}) + \Delta w_0 = -0.2 + 0.1373 \\
&= -0.0627
\end{aligned}$$

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