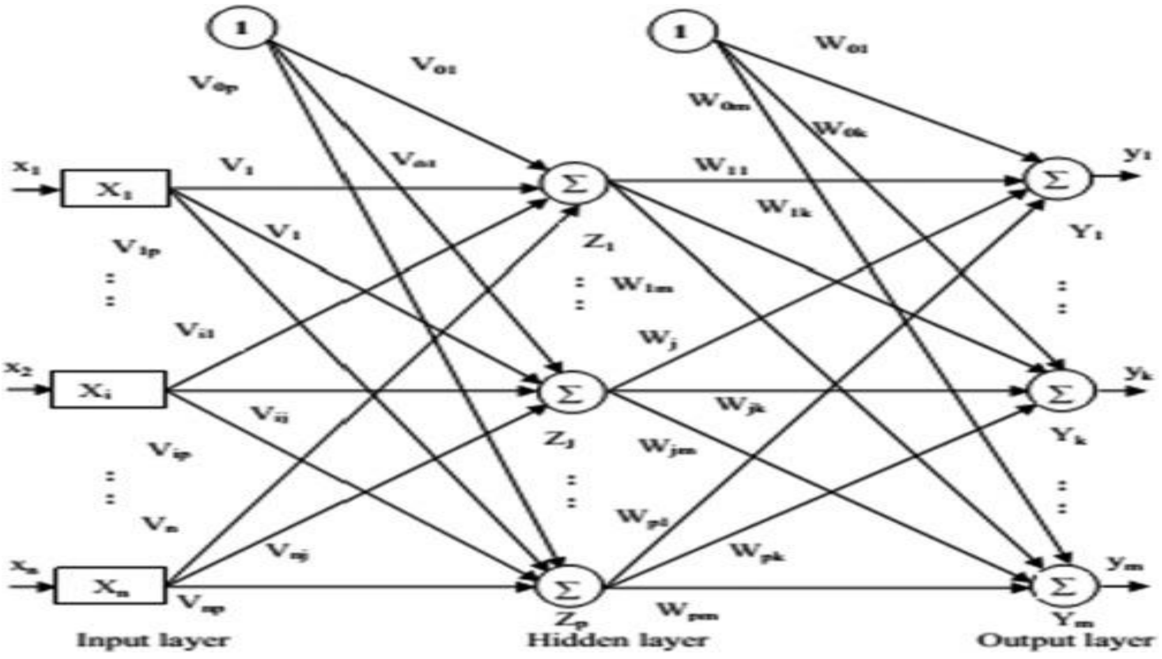
Multilayer perceptron neural network

. Multi-layer perceptron (MLP) is a supplement of feed forward neural network supervisor learning algorithm. It consists of three types of layers (the input layer, output layer and hidden layer).

- The input layer (sensory layer) receives the input signal to be processed.
- An arbitrary number of hidden layers(asocial layers) that are placed in between the input
- Output layer (Response layer)are the true computational engine of the MLP. The major use cases of MLP are pattern classification, recognition, prediction and approximation.



But why do need a hidden layer?

Each layer in a multilayer neural network has its own specific function. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. Actually, the input layer rarely includes computing neurons, and thus does not process input patterns. The output layer accepts output signals, or in other words a stimulus pattern, from the hidden layer and establishes the output pattern of the entire network.

Neurons in the hidden layer detect the features; the weights of the neurons represent the features hidden in the input patterns. These features are then used by the output layer in determining the output pattern.

With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.

Algorithm of Multilayer perceptron Algorithm

Step 1: Initialize the following to start the training (Weights, Bias, Learning rate α), for easy calculation and simplicity,

Step 2: Continue step 3-8 when the stopping condition is not true.

Step 3: Continue step 4-6 for every training vector x.

Step 4: Activate each input unit as follows

$$x_i = s_i(i = 1\ \ to\ n)$$

Step 5: Obtain the net input with the following relation**:**

$$\boldsymbol{y_{in} = \sum_{i}^{n} x_i\ w_{ij} + b}$$

Here ‘b’ is bias and ‘n’ is the total number of input neurons.

Step 6 − Apply the following activation function to obtain the final output for each output unit

j = 1 to m

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

Step **7** : update the weight and bias for x = 1 to n and j=1 to m as following :

− Case 1 : if y_j # t_j

$$w_{ij}(new) = w_{ij}(old) + \alpha t_j x_j$$

$$b_{ij}(new) = b_{ij}(old) + \alpha t_j$$

Case 2 − if $yj_j = tj_j$ then,

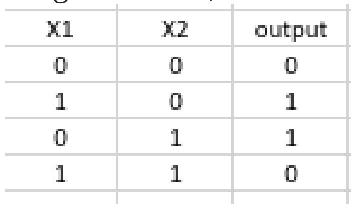
$$w_{ij}(new) = w_{ij}(old)$$
$$b_{ij}(new) = b_{ij}(old)$$

Here ‘**y**’ is the actual output and ‘t’ is the desired/target output.

Step 8: Test for the stopping condition, which will happen when there is no change in weight.

Explanation

The truth table for a two-input XOR-Gate is given below,



X1	X2	output
0	0	0
1	0	1
0	1	1
1	1	0

Fig 1.1 : XOR-Gate Truth Table

We want to get outputs as shown in the above truth table. For this purpose, we have made an MLP (Multilayer Perceptron) architecture shown below.

Here, the circles are neurons(O1, N1, N2, X1, X2), and the orange and blue lines with numbers are the representation of input direction with weights. The numbers on the arrows represent weights.B1 and B2 represent biases.

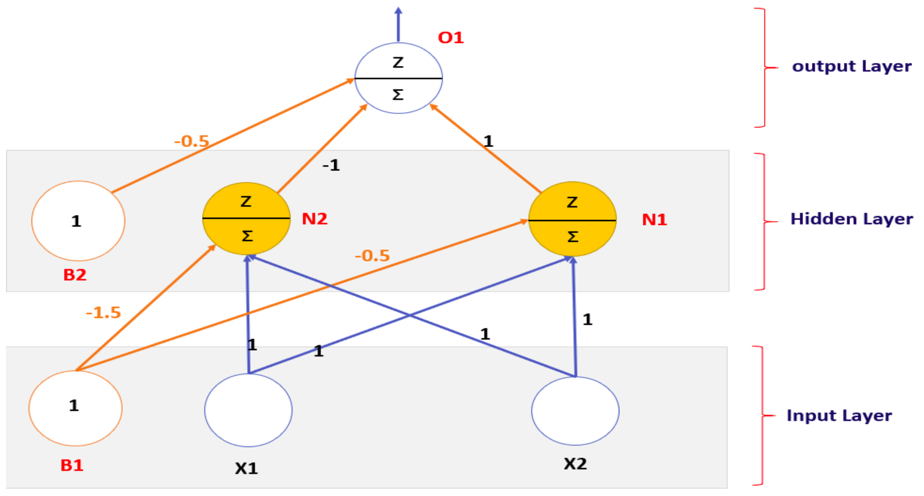


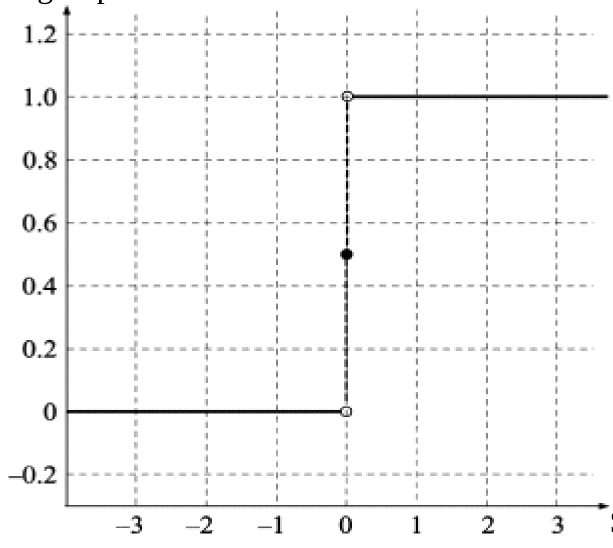
Fig 1.2:XOR-Gate representation using perceptrons.

Step function:

The step function (z) ,triggers only if the weighted sum is 1 or greater than 1. That is to say ,

$$\begin{aligned} if X_1 * W_1 + X_2 * W_2 \geq 1, then, Step - Fuction(X_1 * W_1 + X_2 * W_2) = 1 \\ if X_1 * W_1 + X_2 * W_2 < 1, then, Step - Fuction(X_1 * W_1 + X_2 * W_2) = 0 \end{aligned}$$

Equation 1.1 :Defining step function



Step function in graph

Calculation of XOR gate output

Recall that if we get a value of 1 or greater than 1 for the weighted sum, we will get a value of 1 as an output of the step function otherwise we will get a value of 0 in the output.

Row 1 ,Truth table Fig(1.1),

The XOR gate truth table says, if X1 = 0 and X2 =0 ,the output should be 0 .

For hidden layer neuron N1 (Fig1.2),

$$\begin{aligned} N1 &= X_2 * 1 + X_1 * 1 + B_1 * (-0.5) \\ &= 0 * 1 + 0 * 1 - 0.5 = -0.5 \end{aligned}$$

So ,step_function(-0.5) = 0 ,output of **N1 = 0**

For the hidden layer neuron N2

$$\begin{aligned} N_2 &= X_2 * 1 + X_1 * 1 + B_1 * (-1.5) \\ &= 0 * 1 + 0 * 1 - 1.5 = -1.5 \end{aligned}$$

So , step_function(-1.5) = 0 ,output of **N2 = 0**

For the output neuron O1 ,

$$\begin{aligned} O_1 &= N_2 * (-1) + N_1 * 1 + B_2 * (-0.5) \\ &= (0) * (-1) + (0) * 1 + 1 * (-0.5) \\ &= 0 - 0 - 0.5 = -0.5 \end{aligned}$$

So, step_function(-0.5) = 0 ,output of **O1 = 0**

Matched with the XOR truth table first row.

Row 2,Truth table Fig(1.1),

The XOR gate truth table says, if X1 = 1 and X2 =0 ,the output should be 1 .

For the hidden layer neuron N1

$$N_1 = X_2 * 1 + X_1 * 1 + B_1 * (-0.5) = 0 * 1 + 1 * 1 - 0.5 = 1 - 0.5 = 0.5$$

So, step_function(0.5) = 1 ,output of **N1 = 1**

For the hidden layer neuron N2 ,

$$N_2 = X_2 * 1 + X_1 * 1 + B_1 * (-1.5) = 0 * 1 + 1 * 1 - 1.5 = 1 - 1.5 = -0.5$$

So, step_function(-0.5) = 0 ,output of **N2 = 0**

For the output neuron O1

$$\begin{aligned} O_1 &= N_2 * (-1) + N_1 * 1 + B_2 * (-0.5) \\ &= (0) * (-1) + (1) * 1 + 1 * (-0.5) = 0 - 1 - 0.5 = 0.5 \end{aligned}$$

So, step_function(0.5) = 1 ,output of **O1 = 1**

Matched with the Fig 1.1 ,XOR truth table second row.

Row 3,Truth table Fig,

The XOR gate truth table says, if X1 = 0 and X2 =1 , the output should be 1.

For the hidden layer neuron N1,

$$N_1 = X_2 * 1 + X_1 * 1 + B_1 * (-0.5) = 1 * 1 + 0 * 1 - 0.5 = 1 - 0.5 = 0.5$$

So, step_function(0.5) = 1 ,output of **N1 = 1**

For the hidden layer neuron N2 (fig1.2),

$$N_2 = X_2 * 1 + X_1 * 1 + B_1 * (-1.5) = 1 * 1 + 0 * 1 - 1.5 = 1 - 1.5 = -0.5$$

So, step_function(-0.5) = 0 ,output of **N2 = 0**

For the output neuron O1),

$$\begin{aligned} O_1 &= N_2 * (-1) + N_1 * 1 + B_2 * (-0.5) \\ &= (0) * (-1) + (1) * 1 + 1 * (-0.5) = 0 - 1 - 0.5 = 0.5 \end{aligned}$$

So, step function(0.5) = 1 ,output of **O1 = 1**

Matched with the XOR truth table third row.

Row 4,Truth table Fig(1.1),

The XOR gate truth table says, if X1 = 1 and X2 =1, the output should be 0.

For the hidden layer neuron N1 ,

$$N_1 = X_2 \ast 1 + X_1 \ast 1 + B_1 \ast (-0.5) = 1 \ast 1 + 1 \ast 1 - 0.5 = 1 + 1 - 0.5 = 1.5$$

So, step_function(1.5) = 1 ,output of **N1 = 1**

For the hidden layer neuron N2 ,

$$N_2 = X_2 \ast 1 + X_1 \ast 1 + B_1 \ast (-1.5) = 1 \ast 1 + 1 \ast 1 - 1.5 = 1 + 1 - 1.5 = 0.5$$

So, step_function(0.5) = 1 ,output of **N2 = 1**

For the output neuron O1 (fig1.2),

$$O_1 = N_2 \ast (-1) + N_1 \ast 1 + B_2 \ast (-0.5)$$
$$= (1) \ast (-1) + (1) \ast 1 + 1 \ast (-0.5) = -1 + 1 - 0.5 = -0.5$$

So, step function (-0.5) = 1 ,output of **01 = 0**

Matched with the ,XOR truth table fourth row.

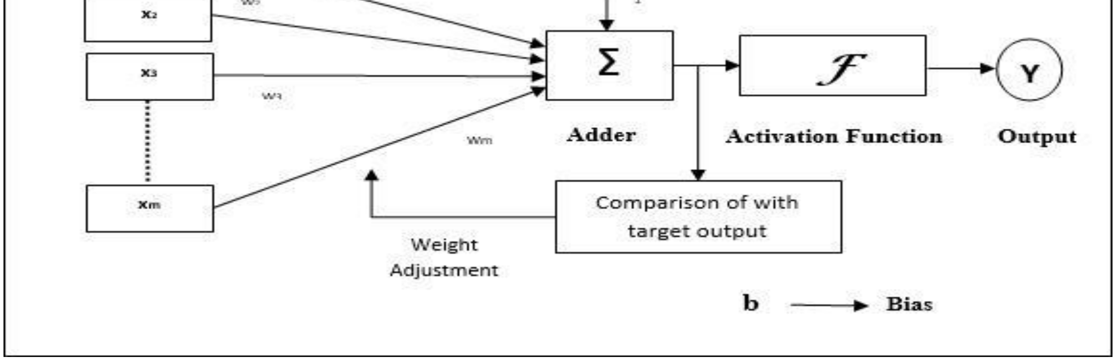
Adaptive Linear Neuron (Adaline)

Adaline which stands for Adaptive Linear Neuron is a network having a single linear unit. The weights are updated by minimizing the cost function via gradient descent.. Some important points about Adaline are as follows:

- It uses bipolar (-1, +1) activation function.
- It uses delta rule for training to least mean square (LMS) between the actual output and the desired/target output.
- The weights from the input units are adjustable.

Architecture

The basic structure of Adaline is similar to perceptron having an extra feedback loop with the help of which the actual output is compared with the desired/target output. After comparison on the basis of training algorithm, the weights and bias will be updated



Training Algorithm

Step 1: Initialize the following to start the training (Weights, Bias and Learning rate)

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2: Continue step 3-8 when the stopping condition is not true.

Step 3: Continue step 4-6 for every bipolar training pair s:t.

Step 4: Activate each input unit as follows −

$$x_i = s_i (i = 1 \ \ to\, n)$$

Step 5: Obtain the net input with the following relation :

$$\boldsymbol{y_{in} = \sum_{i}^{n} x_i \ w_{ij} + b}$$

Here ‘b’ is bias and ‘n’ is the total number of input neurons.

Step 6: Apply the following activation function to obtain the final output :

$$\boldsymbol{f(y_{in}) = \begin{cases} 1 \ \ if \ y_{in} \geq \ \ \theta \\ -1 \ \ \ if \ \ y_{in} < \theta \end{cases}}$$

Step 7 : Adjust the weight and bias as follows :

Case 1 − if y ≠ t then,

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

Case 2 : if y = t then,

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

Here ‘**y**’ is the actual output and ‘**t**’ is the desired/target output (t-yin) is the computed error.

Step 8 − Test for the stopping condition, which will happen when there is no change in weight or the highest weight change occurred during training is smaller than the specified tolerance.

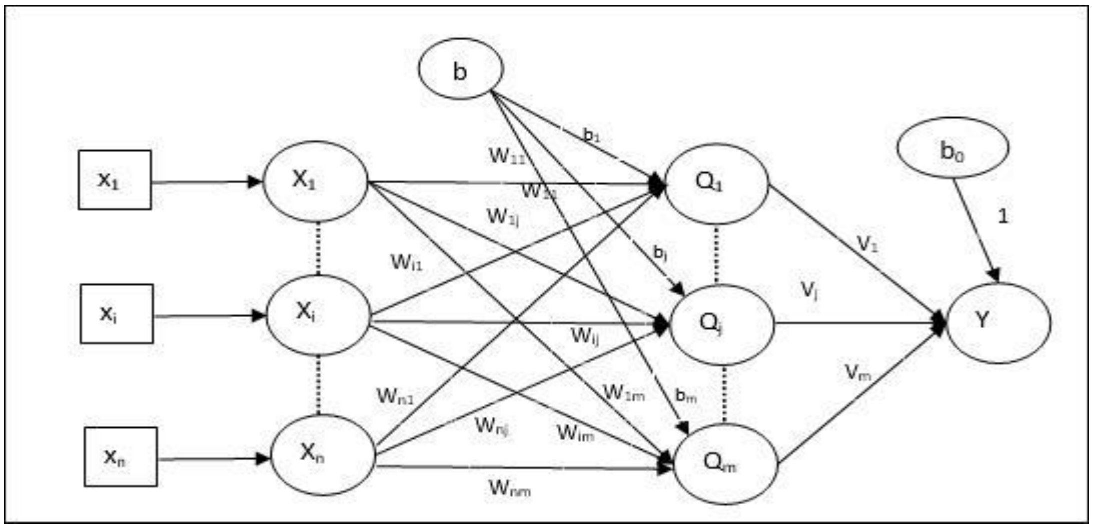
Multiple Adaptive Linear Neuron (Madaline)

Madaline which stands for Multiple Adaptive Linear Neuron is a network which consists of many Adalines in parallel connected with single output unit. Some important points about Madaline are as follows:

- It is just like a multilayer perceptron, where Adaline will act as a hidden unit between the input and the Madaline layer
- The weights between the input and Adaline layers, as the Adaline architecture, are adjustable.
- Only the weights for the hidden Adalines are adjusted; the weights for the output unit are fixed.
- Training can be done with the help of Delta rule.

Architecture

The architecture of Madaline consists of “n” neurons of the input layer, “m” neurons of the Adaline layer, and 1 neuron of the Madaline layer. The Adaline layer can be considered as the hidden layer as it is between the input layer and the output layer, i.e. the Madaline layer



The weights between hidden layer and output layer are fixed positive and equal

$V_1=V_2=\ldots\ldots V_m$

While the weights between input layer and hidden layer are adjusted in training process

The activation function between Adaline and madaline are given by :

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

Training Algorithm

By now that only the weights and bias between the input and the Adaline layer are to be adjusted, and the weights and bias between the Adaline and the Madaline layer are fixed.

Step 1: Initialize the following to start the training (Weights, Bias and Learning rate)

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2: Continue step 3-8 when the stopping condition is not true.

Step 3: Continue step 4-7 for every bipolar training pair s:

Step 4: Activate each input unit as follows:

$$x_i = s_i (i = 1 \;\; to\, n)$$

Step 5: Obtain the net input at each hidden layer, i.e. the Adaline layer with the following relation :

$$Q_{inj} = b + \sum_{i}^{n} x_i \; w_{ij} \qquad j = 1\, to\, m$$

Here ‘b’ is bias and ‘n’ is the total number of input neurons.

Step 6: Apply the following activation function to obtain the final output at the Adaline and the Madaline layer :

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

Output at the hidden (Adaline) unit:

$$Q_j = f\ (Q_{inj})$$

Final output of the network:

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

i.e $y_{inj} = b_0 + \sum_i^m Q_i\ v_i$

Step 7 : Calculate the error and adjust the weights a follows –

Case 1 : if y ≠ t and t = 1 then,

$$w_{ij}(new) = w_{ij}(old) + \alpha(1 - Q_{inj})x_j$$
$$b_j(new) = b_j(old) + \alpha((1 - Q_{inj})$$

In this case, the weights would be updated on Qj_j where the net input is close to 0 because t = 1

. Case 2 − if y ≠ t and t = -1 then :

$$w_{ik}(new) = w_{ik}(old) + \alpha(-1 - Q_{ink})x_j$$
$$b_k(new) = b_k(old) + \alpha((-1 - Q_{inj})$$

In this case, the weights would be updated on Q_k where the net input is positive because t = -1.

Here 'y' is the actual output and 't' is the desired/target output.

Case 3 − if y = t then
There would be no change in weights.

Step 8: Test for the stopping condition, which will happen when there is no change in weight or the highest weight change occurred during training is smaller than the specified tolerance.

Back Propagation Neural Networks

Back Propagation Neural (BPN) is a multilayer neural network consisting of the input layer, at least one hidden layer and output layer. As its name suggests, back propagating will take place in this network. The error which is calculated at the output layer, by comparing the target output and the actual output, will be propagated back towards the input layer. it used to update weights when they are not able to make the correct predications.

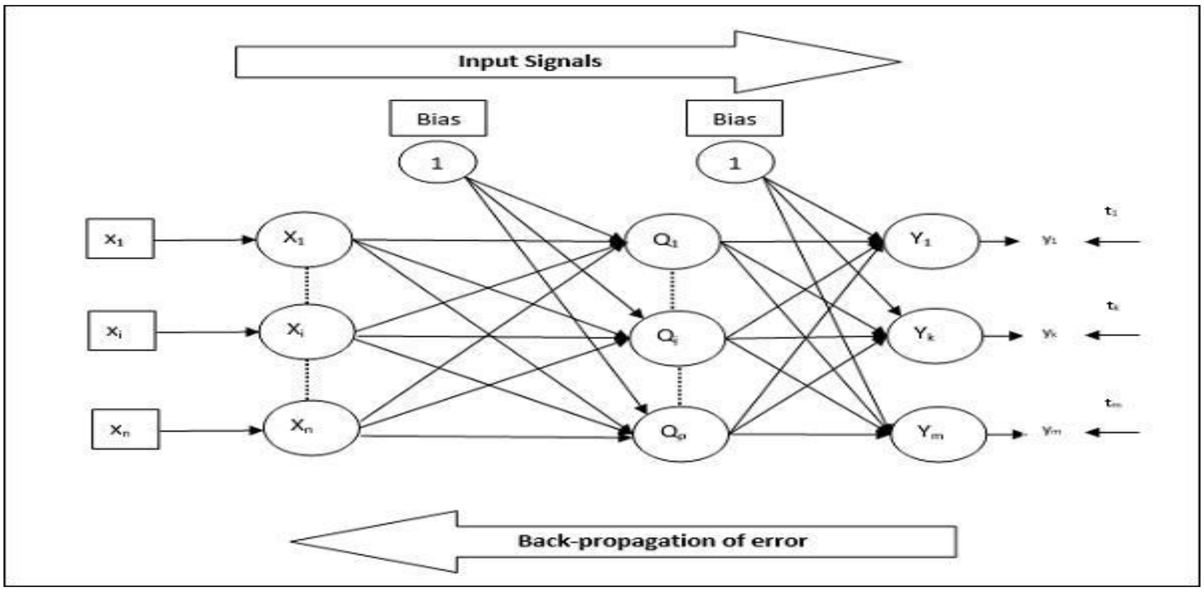
One of the more popular activation functions for backpropagation networks is the sigmoid, a real function sc : IR $\rightarrow$(0, 1) defined by the expression

$$s_c(x) = \frac{1}{1 + e^{-cx}}.$$

The graph shows the shape of the sigmoid for c = 1, c = 2 and c = 3. Higher values of c bring the shape of the sigmoid closer to that of the step function and in the limit c $\rightarrow\infty$ the sigmoid converges to a step function at the origin

Architecture

As shown in the diagram, the architecture of BPN has three interconnected layers having weights on them. The hidden layer as well as the output layer also has bias, whose weight is always 1, on them. As is clear from the diagram, the working of BPN is in two phases. One phase sends the signal from the input layer to the output layer, and the other phase back propagates the error from the output layer to the input layer.



Training Algorithm

For training, BPN will use binary sigmoid activation function. The training of BPN will have the following three phases.

- Phase 1 − Feed Forward Phase.
- Phase 2 − Back Propagation of error.
- Phase 3 − Updating of weights.

All these steps will be concluded in the algorithm as follows:

Step 1 : Initialize the following to start the training (Weights, Bias and Learning rate α)

For easy calculation and simplicity, take some small random values.

Step 2: Continue step 3-11 when the stopping condition is not true.

Step 3 : Continue step 4-10 for every training pair.

Phase 1

Step 4: Each input unit receives input signal x_i and sends it to the hidden unit for all i = 1 to n

Step 5: Calculate the net input at the hidden unit using the following relation **:**

$$Q_{inj} = b_{0j} + \sum_{i}^{n} x_i \; v_{ij} \qquad j = 1 \; to \; p$$

Here b_{0j} is the bias on hidden unit, $\boldsymbol{v_{ij}}$ is the weight $\boldsymbol{o}_j$ unit of the hidden layer coming from i unit of the input layer.

Now calculate the net output by applying the following activation function

$$Q_j = f\;(Q_{inj})$$

Send these output signals of the hidden layer units to the output layer units.

Step 6 − Calculate the net input at the output layer unit using the following relation −

$$y_{ink} = b_{0k} + \sum_{j=1}^{np} Q_j \ w_{ik} \qquad k = 1 \ to \ m$$

Here b_{0k} is the bias on output unit; w_{jk} is the weight on k unit of the output layer coming from j unit of the hidden layer.
Calculate the net output by applying the following activation function:

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

Phase 2
Step 7 − Compute the error correcting term, in correspondence with the target pattern received at each output unit, as follows

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

On this basis, update the weight and bias as follows:

$$\Delta v_{jk} = \alpha \delta_k Q_{ij}$$
$$\Delta b_{0k} = \alpha \delta_k$$

Then send δ_k back to the hidden layer

Step 8: Now each hidden unit will be the sum of its delta inputs from the output units:

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

Error term can be calculated as follows:

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

On this basis, update the weight and bias as follows:

$$\Delta w_{ij} = \alpha \delta_j x_i$$
$$\Delta b_{0j} = \alpha \delta_j$$

Phase 3

Step 9 : Each output unit *($y_k k = 1$ to m)* updates the weight and bias as follows:

$$v_{ik}(new) = v_{jk}(old) + \Delta v_{jk}$$
$$b_{0k}(new) = b_{0k}(old) + \Delta b_{jk}$$

Step 10: Each output unit *($z_j j = 1$ to p)* updates the weight and bias as follows :

$$w_{ij}(new) = w_{ij}(old) + +\Delta w_{ij}$$
$$b_{0j}(new) = b_{0j}(old) + \Delta b_{0j}$$

Step 11: Check for the stopping condition, which may be either the number of epochs reached or the target output matches the actual output.

Generalized Delta Learning Rule
Delta rule works only for the output layer. On the other hand, generalized delta rule, also called as back-propagation rule, is a way of creating the desired values of the hidden layer.

Mathematical Formulation

For the activation function$y_k = f(y_{ink})$ the derivation of net input on Hidden layer as well as on output layer can be given by:

$$y_{ink} = \sum_{k=1}^{m} z_i w_{jk}$$

And

$$y_{inj} = \sum_{k=1}^{m} x_i\, v_{ij}$$

Now the error which has to be minimized is :

$$E = \frac{1}{2} \sum_k [t_k - y_k]^2$$

By using the chain rule, we have:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}(\frac{1}{2}\sum_k [t_k \; - y_k]^2)$$
$$= \frac{\partial}{\partial w_{jk}}(\frac{1}{2}[t_k \; - t(y_{ink})]^2)$$
$$= -[t_k \; - y_k]\frac{\partial}{\partial w_{jk}} f(y_{ink})$$
$$= -[t_k \; - y_k] f(y_{ink})\frac{\partial}{\partial w_{jk}} f(y_{ink})$$
$$= -[t_k \; - y_k] f'(y_{ink}) z_j$$

Now let us say

$$\delta_k = -[t_k \; - y_k] f'(y_{ink}) z_j$$

The weights on connections to the hidden unit zz can be given by :

$$\frac{\partial E}{\partial v_{ij}} = -\sum_k \delta_k \frac{\partial}{\partial v_{ij}}(y_{ink})$$

Putting the value of y ink we will get the following

$$\delta_{inj} = -\sum_{k=1} \delta_k w_{jk} f'(z_{inj})$$

Weight updating can be done as follows , For the output unit :

For the hidden unit

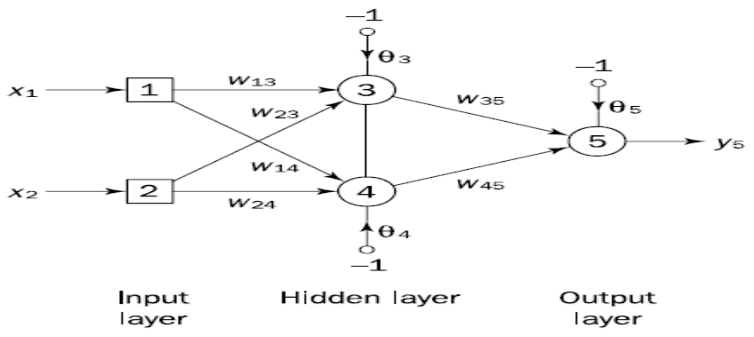
$$\Delta w_{jk} = \alpha \frac{\partial E}{\partial w_{jk}} = \; \alpha \; \delta_k \, z_j$$

\

For the hidden unit

$$\Delta v_{ij} = -\alpha \frac{\partial E}{\partial v_{ij}} = \; \alpha \; \delta_j \, x_i$$

Example: consider the three-layer back-propagation network shown in Figure below. Suppose that the network is required to perform logical operation Exclusive-OR.



w_{13} = 0.5, w_{14} = 0.9, w_{23} = 0.4, w_{24} = 1.0, w_{35} = _1.2, w_{45} = 1.1, Θ_3 = 0.8, Θ_4 = - 0.1 and Θ_5 = 0.3.

Consider a training set where inputs x1 and x2 are equal to 1 and desired output y_{ds} is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as:

$$y_3 = sigmoid(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times0.5+1\times0.4-1\times0.8)}] = 0.5250$$
$$y_4 = sigmoid(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times0.9+1\times1.0+1\times0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = sigmoid(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times1.2+0.8808\times1.1-1\times0.3)}] = 0.5097$$

Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Then we determine the weight corrections assuming that the learning rate parameter α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$
$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$
$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$
$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$
$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$
$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$
$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$
$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$
$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$
$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$
$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$
$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$
$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$
$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$
$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$
$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.