UNIT - ~~IV~~ V

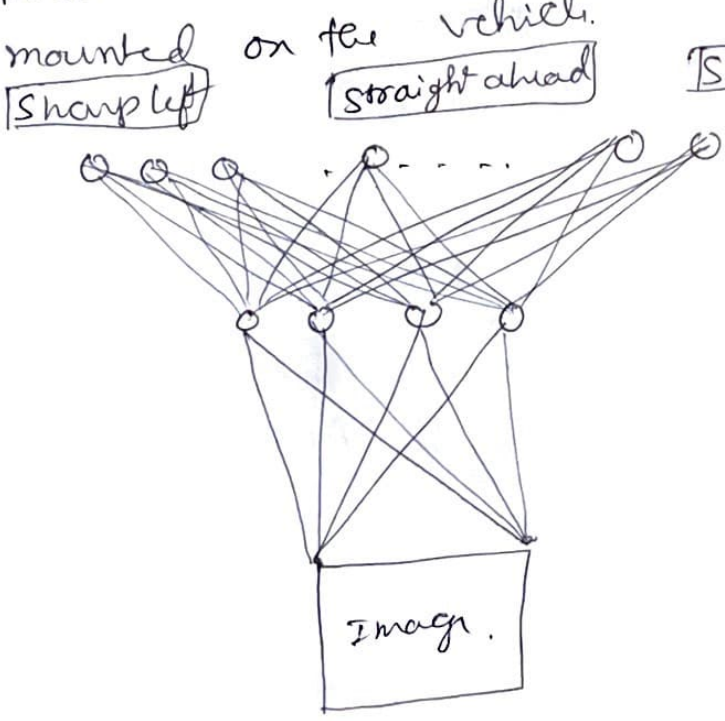
Artificial Neural Networks

4.1. Biological Motivation

ANN mimics the human brain,
Human brain contain 10^{11} neurons.
Each neuron is connected by synaptic weight

4.2. Neural network representation

A prototype example of ANN learning is provided by Pomerleau's System ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.
The I/P to the ANN is a 30 x 32 grid of pixel intensities obtained from a forward point camera mounted on the vehicle.



30 O/P units

4 hidden layer.

30 x 32 Sensor Input Retina.

ANN to learn steer an autonomous vehicle

4.3 Appropriate problems for neural n/w learning

ANN is well suited to problem in which the TD- corresponds to noisy, complex sensor data such as i/p from cameras and microphones.

It is appropriate for the problems with the following characteristics.

(a) Instances are represented by many attribute-value pairs. - a vector of predefined features
Ex:- In ALVINN - the pixel values.

(b) The target fun o/p may be discrete-valued, real-valued or a vector of several real or discrete valued attribute.
Ex:- In ALVINN - o/p is vector of 30 attribute values.

(c) The training example may contain error -

(d) Long training time is acceptable.

(e) Fast evaluation of the learned target fun may be required.

(f) The ability of human to understand the learned target fun is not important.
ii:- weights learned by the ANN are difficult for human to understand.

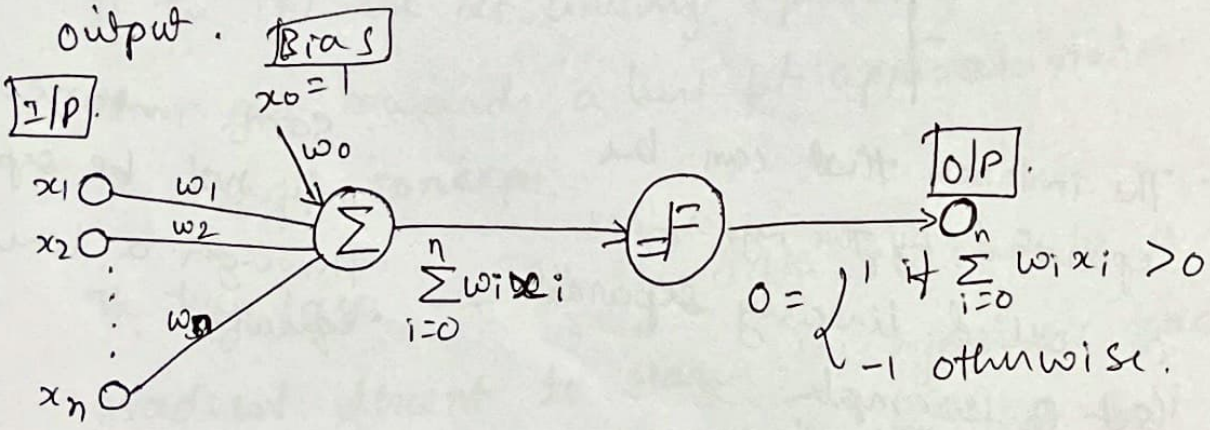
Perceptron

One type of ANN System is based on unit called perceptron.

A perceptron takes a vector of real-valued i/p. calculates a linear combination of these i/p then o/p a 1 if the result is > some threshold value and returns -1 otherwise.

$$o(x_1, x_2 \ldots x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 \ldots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

where w_i is a real-valued constant or weight that determines the contribution of x_i to the perceptron output.



It can be represented with vector as

$$o(\vec{x}) = \text{Sgn}(\vec{w} \cdot \vec{x})$$

where

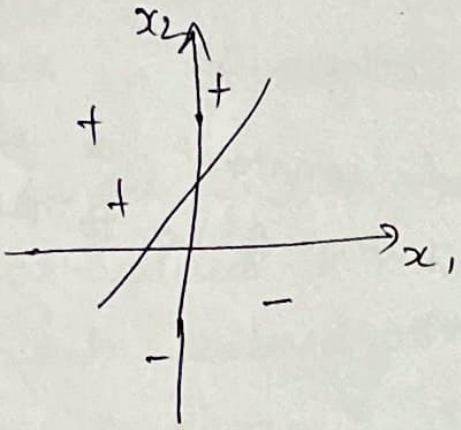
$$\text{Sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$$

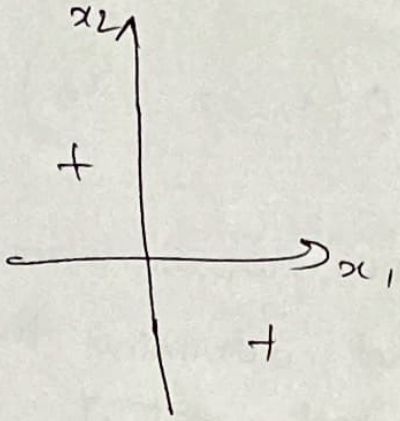
Learning a perceptron involves choosing values for the wts $w_0, \ldots w_n$ $[\because H = \vec{w} \mid \vec{w} \in R^{n+1}]$

4.4.1. Representational power of perceptron

A perceptron can be viewed as representing a hyper plane decision surface in the n-dimentional space of instances (points)

The perceptron O/P 1 for instances lying on side of the hyperplane and O/P -1 for instances lying on the other side





The instances that can be separated by the hyperplane are called linearly separable set of examples

Some instances may not be separable through a linear equation.

4.4.2. The perceptron Training Rule

There are several alg to solve this learning problem. Let us consider two: ① perceptron rule
② delta rule (variant of LMS)

one way to learn an acceptable wt vector is to begin with random wts. then iteratively apply the perceptron to each TE, modifying the perceptron wt whenever it misclassifies an instance.

wts are modified at each step according to the perceptron training rule. which revises the wt w_i associated with i/p x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where $\Delta w_i = \eta (t - o) x_i$

4.4.3. Gradient Descent and the Delta Rule.

perceptron rule may get failed in case of linearly non-separable instances. The delta rule is used to overcome this difficulty.

If the TE are not linearly separable, the delta rule converges towards a best-fit approximation to the target concept.

The key idea behind the delta rule is to use gradient descent to search the hypothesis space of possible wt vectors to find the wts that best fit the TE.

Gradient descent provides the basis for the Backpropagation algorithm, which can learn networks with many interconnected units

The delta training rule is best understood by considering the task of training an unthresholded perceptron.

it a linear unit for which the o/p O is given

$$O(\vec{x}) = \vec{w} \cdot \vec{x}$$

Thus a linear unit corresponds to the first stage of a perceptron, without the threshold.

To derive a wt learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (wt vector) relative TE.

one common way of error measuring is.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2.$$

where D is the set of TE, t_d is the target o/p for TE d, and o_d is the o/p of the linear unit for TE d.

Derivation of the gradient descent rule

How to compute the direction of steepest descent along the error surface.

This can be found by computing the derivative of E wrt each component of the vector $\vec{w}$. This derivative is called the gradient of E wrt $\vec{w}$ written $\Delta E(\vec{w})$

$$\Delta E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots \frac{\partial E}{\partial w_n} \right]$$

$\Delta E(\vec{w})$ is a vector whose components are the partial derivatives of E wrt each of w_i.

since the gradient specifies the direction of steepest ↑ of E, the training rule of gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}.$$

where $\Delta\vec{w} = -\eta \nabla E(\vec{w})$

[$\Delta w_i = \eta (t-o) x_i$ in case of perceptron rule]

where η is a +ve constant called the learning rate. which determine the step size in GDS (gradient descent search).

-ve sign implies that we want to move the wt vector in the direction that decreases E.

This rule can be written in component form as

$$w_i \leftarrow w_i + \Delta w_i$$

where $$\boxed{\Delta w_i = -\eta \frac{\partial E}{\partial w_i}} \quad \text{——} (1)$$

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from $E(\vec{w})$.

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$\boxed{\because E = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2}$$

$$= \frac{1}{2} \sum \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$\left[\because \partial x^y = y x^{y-1} \partial x \right]$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d})$$

$$\boxed{\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})} \quad \text{— (2)}$$

where x_i denote the single i/p component x_i for TE d.

Now substitute eq(2) in (1).

$$\boxed{\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}}$$

Gradient Descent algorithm for training a linear unit

Gradient-Descent (TE, η)

Each TE is a pair of the form $(\vec{x}, t)$, where $\vec{x}$ is the vector of i/p values, and t is the target o/p value, η is the learning rate (e.g. 0.05).

- Initialize each w_i to some small random value
- Untill the termination condition reached, do.
  1* Initialize each Δw_i to zero.
  2* For each $\langle \vec{x}, t \rangle$ in TE, do
     - I/P the instance $\vec{x}$ to the unit and compute the o/p o.
     - For each linear unit wt w_i, do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t-o)x_i \quad -①$$

  3* For each linear unit wt w_i, do

$$w_i \leftarrow w_i + \Delta w_i \quad -②$$

Stochastic approximation to gradient descent

The key practical difficulties in applying gradient descent are

① converging to a local minimum can sometimes be quite slow (ie it can require many thousands of gradient descent steps).

② If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

Soln: In above algo delete Step 3 (ie eq②)
change eq① to

$$w_i \leftarrow w_i + \eta(t-o)x_i$$

4.5. Multilayer networks and the Backpropag algorithm

A single perceptron can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the Backpropagation alg. are capable of expressing a rich variety of non-linear decision surfaces

4.5.1. A differentiable Threshold Unit

Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result.

In case of sigmoid unit, however, the threshold output is a continuous fun of its input. More precisely, the sigmoid unit computes its o/p o as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where $\sigma(y) = \frac{1}{1-e^{-y}}$.

4.5.2. The Backpropagation Algorithm.

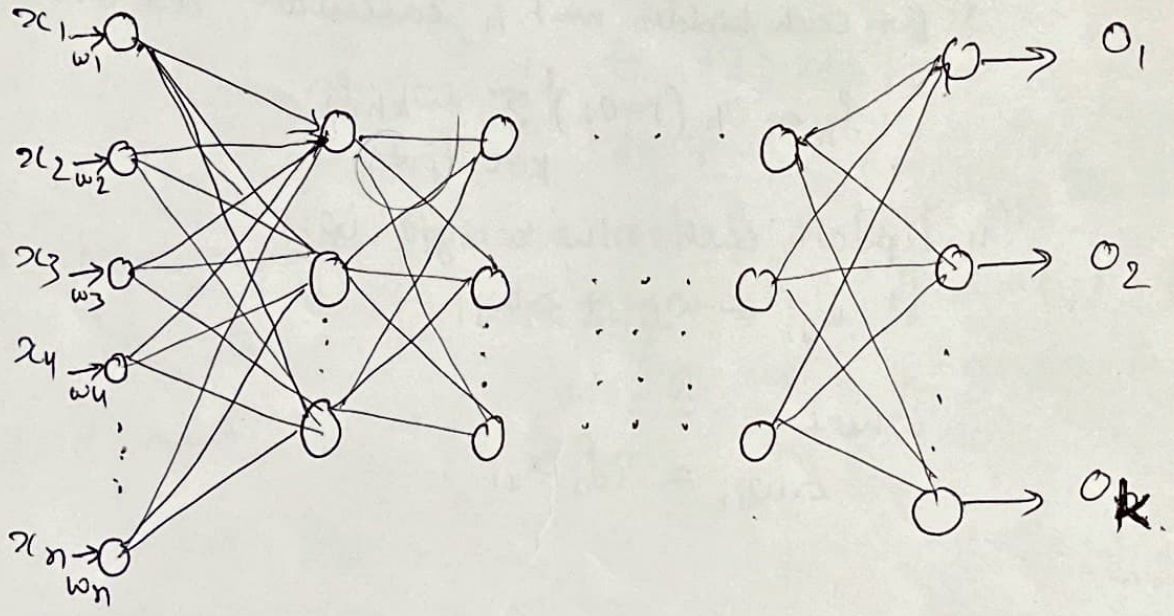
It learns the wts for a multilayer n/w, given a n/w with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the n/w o/p value and the target value for these o/ps.

As we are considering n/w with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the n/w output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in o/ps} (t_{kd} - o_{kd})^2.$$

where outputs is the set of o/p units in the n/w and t_{kd} & o_{kd} are the target and o/p values associated with the k^{th} o/p unit and training example d.

The learning problem faced by BPN is to search a large hyp space defined by all possible wt values for all the units in the n/w



BACKPROPOGATION(TE, η, n_{in}, n_{out}, n_{hidden})

Each TE is a pair of the form $(\vec{x}, \vec{t})$, where $\vec{x}$ is the ve
of n/w i/p value, and $\vec{t}$ is the vector of target n/w o/p valu
η is the learning rate
n_{in} is the no of n/w i/ps.
n_{hidden} is the no of units in the hidden layer.
n_{out} is the no of o/p units.

- create a feed-forward n/w with n_{in} inputs, n_{hidden} hidden units and n_{out} output units.
- Initialize all n/w wts to small random no
- Untill the termination condition is reached, DO
  - *For each $(\vec{x}, \vec{t})$ in TE, DO.

    Propogate the i/p forward through n/w.

    1. I/P $\vec{x}$ and compute O_u of every unit u in n/w

    propogate the error backward through the n/w.

    2. For each n/w o/p unit k, calculate its err term δ_k.

$$\delta_k \leftarrow O_k(1-O_k)(t_k - O_k)$$

3. For each hidden unit h, calculate its err term δ_h.

$$\delta_h \leftarrow O_h(1-O_h)\sum_{k \in outputs} w_{kh}\delta_k$$

4. Update each n/w weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Derivation of BPA

* To derive the eq for updating weights in BPA we use stochastic gradient descent rule. (SGD)
* SGD involves iterating through the TE one at a time, for each TE d descending the gradient of the error Ed wrt this single d.
* In other words, for each TE d every wt w_{ji} is updated by adding to it Δw_{ji}.
* ie $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$

where $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$ — (1)

* where Ed is the error on TE d, that is half the squared difference b/t the target o/p and the actual o/p over all o/p units in the n/w.

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

* Here outputs is the set of output units in the n/w, t_k is the target value of unit k for TE d, and o_k is the o/p of unit k given TE d.

Notations used

x_{ji} = the ith I/P to unit j

w_{ji} = the wt associated with the i^{th} I/P to unit j

$net_j = \sum_i w_{ji} x_{ji}$ (the wtd sum of i/p for unit j)

o_j = the o/p computed by unit j

t_j = the target o/p for unit j.

σ = the sigmoid function

outputs = the set of units in the final layer of nlw.

Downstream(j) = the set of units whose immediate inputs include the o/p of unit j.

*To begin, notice that wt w_{ji} can influence the rest of the nlw only through net_j.

∴ we can use the chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \cdot x_{ji} \quad -(2)$$

Now substitute (2) in (1).

$$\boxed{\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} \cdot x_{ji}} \quad -(3)$$

$$net_j = \sum_i w_{ji} x_{ji}$$

$$\frac{\partial net_j}{\partial w_{ji}} = x_{ji}$$

when Σ move from one side to the other it becomes differentiation.

* To derive the convinient expression for $\frac{\partial E_d}{\partial net_j}$ we consider two cases in turn.

→ case1, where unit j is an o/p unit for the n/w

→ case2, where unit j is an internal unit of the n/w.

case 1: Training Rule for output unit wts.

* Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the n/w only through o_j.

∴ we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \quad — (4)$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \cdot \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \cdot \frac{1}{2} (t_j - o_j)^2$$

$$\frac{\partial E_d}{\partial o_j} = \frac{1}{2} 2 (t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j}$$

$$\boxed{\frac{\partial E_d}{\partial o_j} = -(t_j - o_j)} \quad — (a)$$

$$\frac{\partial O_j}{\partial (net_j)} = \frac{\partial \sigma(net_j)}{\partial (net_j)}$$

$$\frac{\partial O_j}{\partial (net_j)} = \sigma(net_j)\,(1 - \sigma(net_j))$$

$$\boxed{\frac{\partial O_j}{\partial net_j} = O_j(1 - O_j)} \quad -(6)$$

properties

$$\frac{\partial \sigma(x)}{\partial (x)} = \text{case 2}$$

net_j can

$$\sigma(x)\,(1 - \sigma(x))$$

$$\sigma(net_j) = O_j$$

Substitute (a) and (6) in (4).

$$\frac{\partial E_d}{\partial net_j} = -(t_j - O_j)\, O_j(1 - O_j) \quad -(5)$$

Now Substitute (5) in (3).

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} \cdot x_{ji}$$

$$= (-\eta)\left(-(t_j - O_j)\, O_j(1 - O_j)\right) \cdot x_{ji}$$

$$= \eta \boxed{(t_j - O_j)\, O_j\,(1 - O_j)} \cdot x_{ji}$$

$\boxed{\quad} \rightarrow \delta_j$

$$\therefore \Delta w_{ji} = \eta \delta_j x_{ji}$$

And $w_{ji} = w_{ji} + \Delta w_{ji}$

Case 2: Training rule for hidden unit wts
net_j can influence n/w o/p through its DS units.

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in DS(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$\frac{\partial E_d}{\partial net_j} = -\delta_j \therefore \frac{\partial E_d}{\partial net_k} = -\delta_k$

$$= \sum_{k \in DS(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

$\frac{\partial net_k}{\partial o_j} = \frac{\partial x_{kj} w_{kj}}{\partial o_j} = \frac{\partial o_j w_{kj}}{\partial o_j} = w_{kj}$

$$= \sum_{k \in DS(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j}$$

$\frac{\partial o_j}{\partial net_j} = \frac{\partial(\sigma(net_j))}{\partial net_j}$

$$= \sum_{k \in DS(j)} -\delta_k w_{kj} o_j (1 - o_j) \quad \text{— (6)}$$

Now substitute (6) in (3)

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = \eta \left(\sum_{k \in DS(j)} \delta_k w_{kj} o_j (1 - o_j) \right) x_{ji}$$

$$= \eta o_j (1 - o_j) \sum_{k \in DS(j)} \delta_k w_{kj} x_{ji}$$

$$= \eta \delta_j x_{ji} \quad \left[\delta_j = o_j (1 - o_j) \sum_{k \in DS(j)} \delta_k w_{kj} \right]$$

4.5.2.1. Adding Momentum

BPA is a widely used algorithm and hence many variations have been developed. The most alteration is done wrt weight updation.

ie $\Delta w_{ji} = \eta \delta_j x_{ji}$ —①

This update rule can be performed as follows.

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \quad -②$$

The eqn ② represents the wt updation on n^{th} iteration, depends partially on the update that occured during $(n-1)^{th}$ iteration.

α is called momentum lies b/t 0 & 1 [ie $0 < \alpha < 1$]

The second term of RHS of eqn ② is called the momentum term. If we consider the error space as a momentum -um ball, then the effect of α is to add momentum that tends to keep the ball rolling in the same direction from one iteration to next iteration. This momentum term helps in fast convergence of the algorithm.

Neural network to explain Backpropogation algorithm