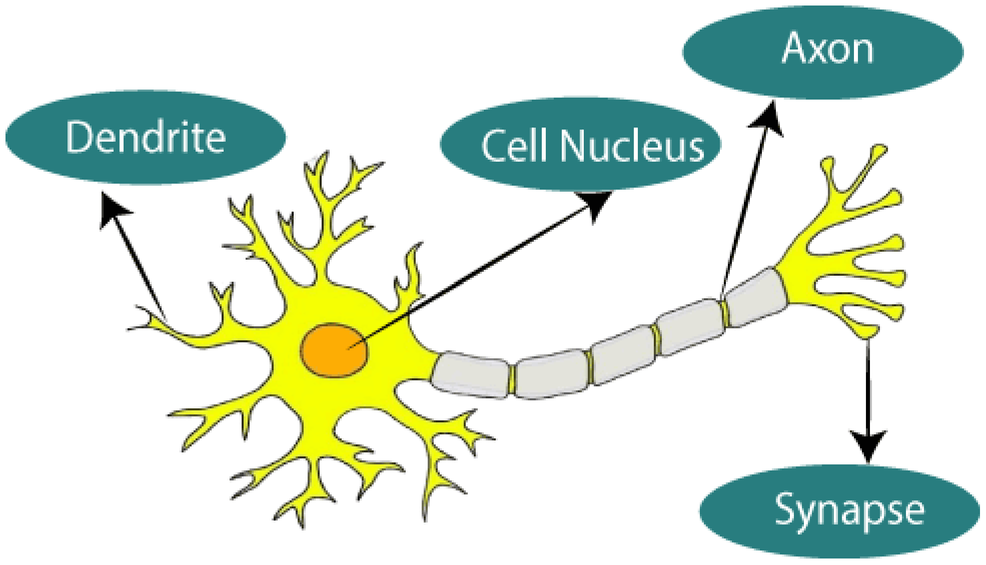
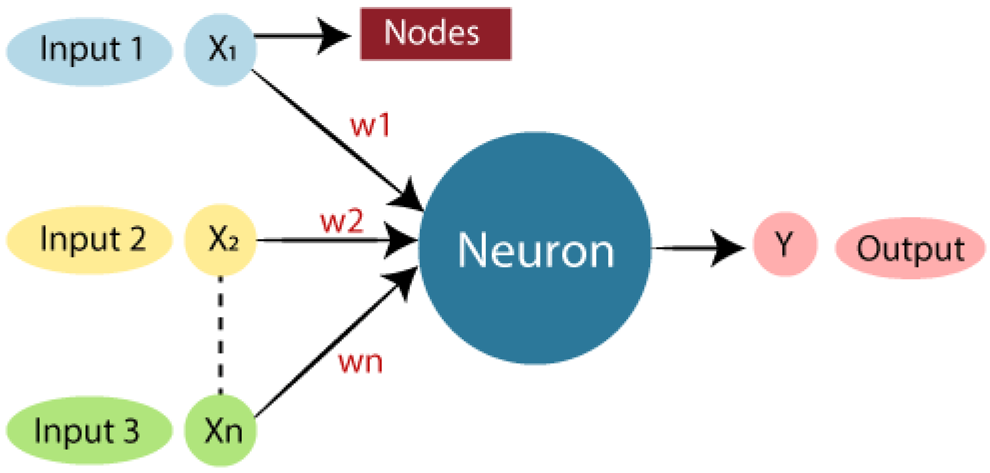
1. Artificial Neural Networks and Its components

Neural Networks is a computational learning system that uses a network of functions to understand and translate a data input of one form into a desired output, usually in another form. The concept of the artificial neural network was inspired by human biology and the way neurons of the human brain function together to understand inputs from human senses.

In simple words, Neural Networks are a set of algorithms that tries to recognize the patterns, relationships, and information from the data through the process which is inspired by and works like the human brain/biology.





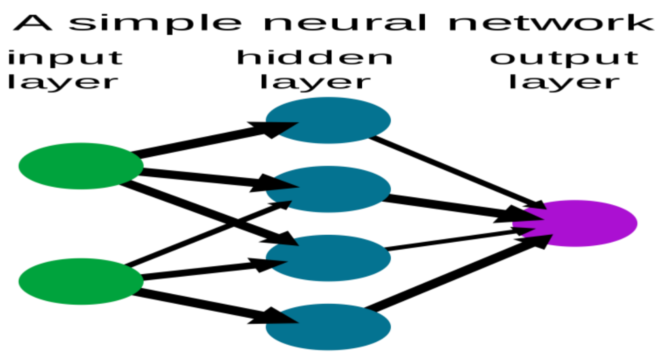
Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

Biological Neural Network	**Artificial Neural Network**
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

A simple neural network consists of three components :

- Input layer
- Hidden layer
- Output layer



Input Layer: Also known as Input nodes are the inputs/information from the outside world is provided to the model to learn and derive conclusions from. Input nodes pass the information to the next layer i.e Hidden layer.

Hidden Layer: Hidden layer is the set of neurons where all the computations are performed on the input data. There can be any number of hidden layers in a neural network. The simplest network consists of a single hidden layer.

Output layer: The output layer is the output/conclusions of the model derived from all the computations performed. There can be single or multiple nodes in the output layer. If we have a binary classification problem the output node is 1 but in the case of multi-class classification, the output nodes can be more than 1.

2. APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN(Artificial Neural Networks) learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones. It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks.

The BACKPROPAGATION algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.***
- ***The training examples may contain errors.*** ANN learning methods are quite robust to noise in the training data.
- ***Long training times are acceptable.*** Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- ***Fast evaluation of the learned target function may be required.*** Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.
- ***The ability of humans to understand the learned target function is not important.*** The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules

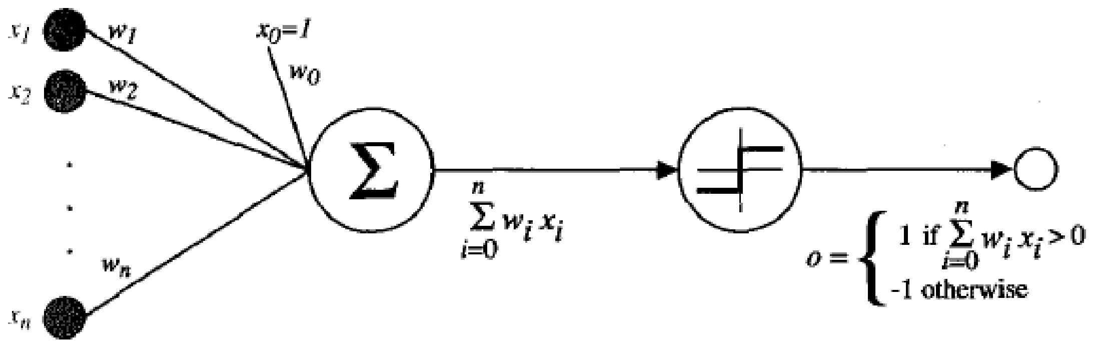
3. PERCEPTRONS

One type of ANN system is based on a unit called a perceptron. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some

threshold and -1 otherwise. More precisely, given inputs ***xl*** through ***x,,*** the output ***o(x1,*** . . . , ***x,)*** computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each ***wi*** is a real-valued constant, or weight, that determines the contribution of input ***xi*** to the perceptron output.


x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

A Perceptron

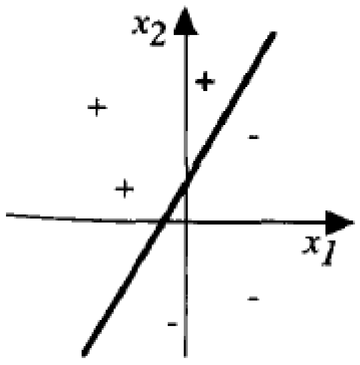
To simplify notation, assume an additional constant input x0=1. We can write the perceptron function as

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

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

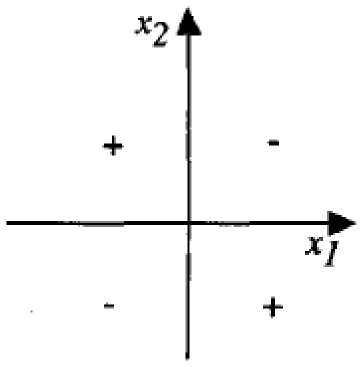
3.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0.$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. The examples that can be separated are called linearly separable.



Linearly Separable

Those examples that cannot be separated by any hyperplane are called linearly non- separable.



Linearly non-separable

Boolean Functions:

A single perceptron can be used to represent many boolean functions 1 (true); 0 (false) . Perceptrons can represent all of the primitive boolean functions AND, OR,NAND (¬ AND), and NOR (¬OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if **xl** # **x2.**

3.2 The Perceptron Training Rule:

Several algorithms are known to solve the learning problem which is to determine a weight vector that causes the perceptron to produce the correct ±1 output for each of the given training examples.

The two of the algorithms are : the perceptron rule and the delta rule.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the ***perceptron training rule,*** which revises the weight ***wi*** associated with input ***xi*** according to the rule:

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

Where

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

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and η is a positive constant called the ***learning rate.*** The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.
Suppose the perceptron outputs a -1, when the target output is +1. To make the perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of
$\vec{w} \cdot \vec{x} = 0.$

3.3. Gradient Descent and Delta Rule

Although, the Perceptron rue finds successful weight vector when the training examples are linearly separable , it can fail to converge if the examples are linearly non-separable.
A Second training rule, called delta rule is designed to overcome this difficulty. If the training examples are linearly non-separable , the delta rule converges toward 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 weight vectors to find the weights that best fit the training examples.

$\vec{w}$ = {w1,w2,…,wn}

The rule is important because the gradient descent provides the basis for the Backpropagation algorithm, which can learn with many interconnected units.
The delta rule is best understood by considering the task of training an unthresholded perceptron i.e., a linear unit for which the output o is given by

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

O= w1x1+w2x2+…+wnxn

In order to derive a weight learning rule for linear units,let us begin by specifying a measure for the training error of a hypothesis(weight vector), relative to the training examples.

Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

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

Where D = set of training examples

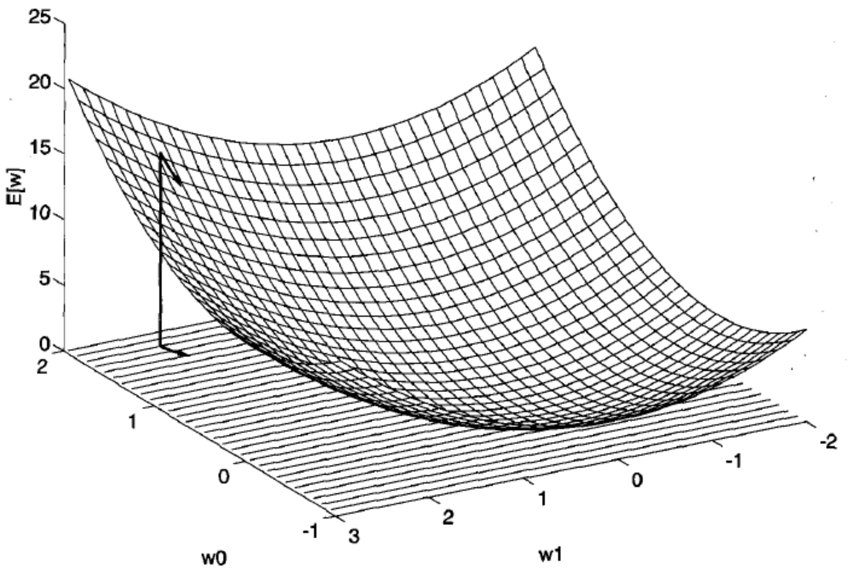
t_d = target output for training example d

o_d = output of the linear unit for training example d

3.4. VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values. Here the axes ***w0*** and ***w1*** represent possible values for the two weights of a simple linear unit. The ***w0, w1*** plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples.

Gradient descent search determines a weight vector that minimizes ***E*** by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface**.** This process continues until the global minimum error is reached.



3.5. DERIVATION OF THE GRADIENT DESCENT RULE

The direction of steepest descent along the error surface can be found by computing the derivative of ***E*** with respect to each component of the vector $\vec{w}$. This vector derivative is called the ***gradient*** of ***E*** with
respect to $\vec{w}$. written $\nabla E(\vec{w})$.

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

$\nabla E(\vec{w})$. is itself a vector, whose components are the partial derivatives of ***E*** with respect to each of the ***wi.***
When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease.

Since the gradient specifies the direction of steepest increase of ***E,*** the training rule for gradient descent is

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

Where

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

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that ***decreases*** E.

This training rule can also be written in its component form

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

Where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

which makes it clear that steepest descent is achieved by altering each component ***wi*** of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E :

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d)(-x_{id})
\end{aligned}$$

where x_{id} denotes the single input component xi for training example d.

Substituting $\frac{\partial E}{\partial w_i}$ in $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$ yields the weight update rule for gradient descent as

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

To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples, then compute Δwi for each weight
- Update each weight ***wi*** by adding Δwi , then repeat this process.
- Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate ***ŋ*** is used.
- If ŋ is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it.
- For this reason, one common modification to the algorithm is to gradually reduce the value of ŋ as the number of gradient descent steps grows.

4. STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

(1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and
(2) the error can be differentiated with respect to these hypothesis parameters.

The key practical difficulties in applying gradient descent are

(1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and
(2) if there are multiple local minima in the error surface, then there is no guarantee that then procedure will find the global minimum.

One common variation on gradient descent to overcome these difficulties is called ***incremental gradient descent,*** or alternatively ***stochastic gradient descent.***

Whereas the gradient descent training rule computes weight updates after summing over ***all*** the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for ***each*** individual example.

The weights are updated by using the following equation

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

where t, ***o,*** and ***xi*** are the target value, unit output, and ith input for the training example.
One way to view this stochastic gradient descent is to consider a distinct error function for each individual training example d **as** follows:

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

where t_d, and o_d are the target value and the unit output value for training example d. Stochastic gradient descent iterates over the training examples ***d*** in D,at each iteration altering the weights according to the gradient with respect to E_d**.** The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function **.**

The key differences between standard gradient descent and stochastic gradient descent are:

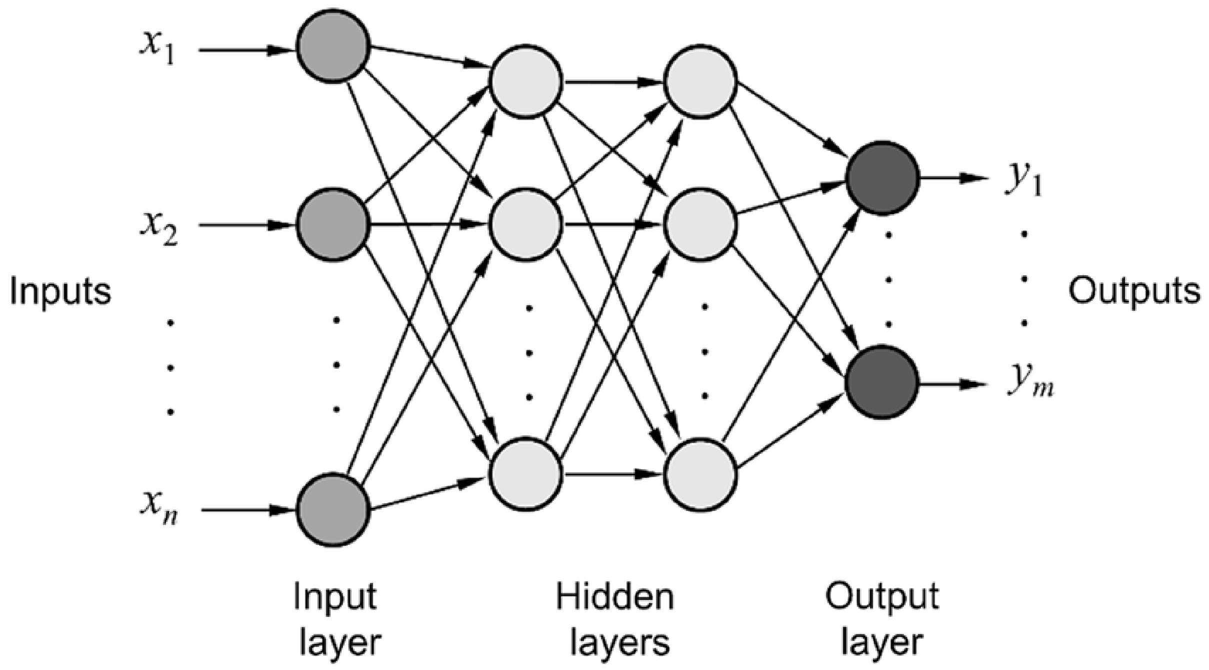
(1) In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.

(2) Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.

(3) In cases where there are multiple local minima with respect to $E(\vec{w})$, stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

5. MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

In a multi layer neural network, there will be one input layer, one output layer and one or more hidden layers.



Representation of a Multi Layer Neural Network

Each and every node in the nth layer will be connected to each and every node in the (n-1)th layer(n>1). So, the input from the input layer is multiplied with the associated weights of every link and will be traversed till the output layer for the final ouput. In case of any error, unlikperceptron, in this case we might need to update several weight vectors in many hidden layers. This is where **Back Propagation** comes into place. **It's nothing but updation of the weight vectors in the hidden layers according to the training error or the loss produced in the ouput layer.**

BACK PROPAGATION ALGORITHM

Backpropagation, short for "backward propagation of errors", is a mechanism used to update the **weights** using gradient descent. It calculates the gradient of the error function with respect to the neural network's weights.

The formula for calculating training error for a neural network can be represented as follows:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \epsilon D} \sum_{k \epsilon outputs} (t_{kd} - o_{kd})^2$$

Error function in multi-layer neural networks

- outputs is the set of output units in network
- d is the data point

- t and o are target values and the output values produced by the network for the kth output unit for data point ‘d’.

Now that we have the error function, input and output units we need to know the rule for updation of weight vector. Before that let’s know about one of the most common activation functions used in multi layer neural networks i.e **sigmoid** function.
A sigmoid function is any function which is continuously differentiable be it e^x or hyberbolic tangent(tanh) which produces the output in the range of 0 to 1 (not including 0 and 1). It can be represented as:

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

Sigmoid Function
where, y is the linear combination of input vector and the weight vector at a given node.

Now, let’s know how the weight vectors are updated in multi layer networks according to Back Propagation Algorithm.

Updation of weights in Back Propagation

The algorithm can be represented in step-wise manner:

- Input the first data point into the network and calculate the output for each output unit and let it be ‘o’ for every unit ‘u’.
- For each output unit ‘k’, training error ‘ δ ‘ can be calculated by the given formula:

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

- For each hidden unit ‘h’, training error ‘ δ ‘ can be calculated by the given formula in which the training error of output units to which the hidden layer is connected is taken into consideration:

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

- Update weight vectors by the given formula:

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

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

- weight vector from jth node to ith node is updated using above formula in which ‘**η’** is the learning rate, ‘δ’ is the training error and ‘x’ is the input vector for the given node.

Termination Criterion for Multi layer networks

The above algorithm is continuously implemented on all data points until we specify a termination criterion, which can be implemented in either of these three ways:

- training the network for a fixed number of epochs (iterations).
- setting the threshold to an error, if the error goes below the given threshold, we can stop training the neural network further.
- Creating a validation sample of data, after every iteration we validate our model with this data and the iteration with the highest accuracy can be considered as the final model.

The first way of termination might not yield us better results , the most recommended way is the third way as we are aware of the accuracy of our model so far.