# ARTIFICIAL NEURAL NETWORK

# UNIT-2

# *INTRODUCTION TO PERCEPTRON

**What is Perceptron?**

**Perceptron is one of the simplest Artificial neural network architectures. It was introduced by Frank Rosenblatt in 1957s. It is the simplest type of feedforward neural network, consisting of a single layer of input nodes that are fully connected to a layer of output nodes. It can learn the linearly separable patterns. it uses slightly different types of artificial neurons known as threshold logic units (TLU). it was first introduced by McCulloch and Walter Pitts in the 1940s.**

**Types of Perceptron**

- **Single-Layer Perceptron: This type of perceptron is limited to learning linearly separable patterns. effective for tasks where the data can be divided into distinct categories through a straight line.**
- **Multilayer Perceptron: Multilayer perceptrons possess enhanced processing capabilities as they consist of two or more layers, adept at handling more complex patterns and relationships within the data.**

**Basic Components of Perceptron**

**A perceptron, the basic unit of a neural network, comprises essential components that collaborate in information processing.**

- **Input Features: The perceptron takes multiple input features, each input feature represents a characteristic or attribute of the input data.**
- **Weights: Each input feature is associated with a weight, determining the significance of each input feature in influencing the perceptron's output. During training, these weights are adjusted to learn the optimal values.**
- **Summation Function: The perceptron calculates the weighted sum of its inputs using the summation function. The summation function combines the inputs with their respective weights to produce a weighted sum.**
- **Activation Function: The weighted sum is then passed through an activation function. Perceptron uses Heaviside step function functions. which take the summed values as input and compare with the threshold and provide the output as 0 or 1.**
- **Output: The final output of the perceptron, is determined by the activation function's result. For example, in binary classification problems, the output might represent a predicted class (0 or 1).**

- **Bias: A bias term is often included in the perceptron model. The bias allows the model to make adjustments that are independent of the input. It is an additional parameter that is learned during training.**
- **Learning Algorithm (Weight Update Rule): During training, the perceptron learns by adjusting its weights and bias based on a learning algorithm. A common approach is the perceptron learning algorithm, which updates weights based on the difference between the predicted output and the true output.**

**These components work together to enable a perceptron to learn and make predictions. While a single perceptron can perform binary classification, more complex tasks require the use of multiple perceptrons organized into layers, forming a neural network.**

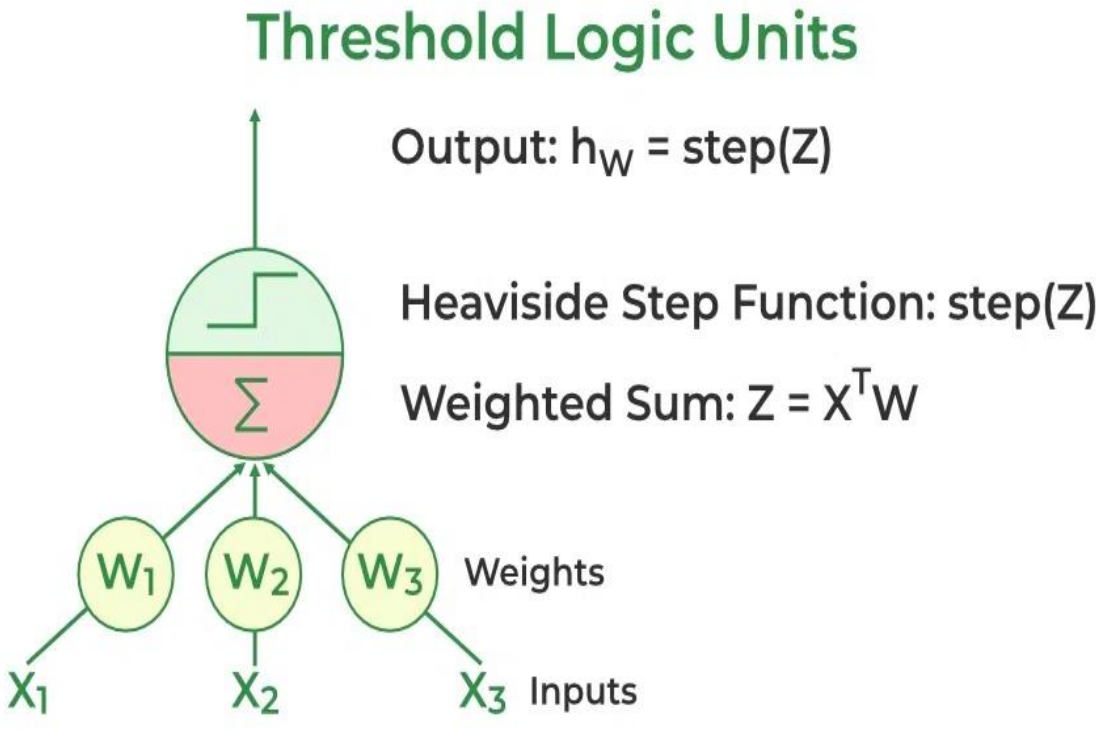
**How does Perceptron work?**

**A weight is assigned to each input node of a perceptron, indicating the significance of that input to the output. The perceptron's output is a weighted sum of the inputs that have been run through an activation function to decide whether or not the perceptron will fire. it computes the weighted sum of its inputs as**

**z = w1x1 + w1x2 + ... + wnxn = XTW**

**The step function compares this weighted sum to the threshold, which outputs 1 if the input is larger than a threshold value and 0 otherwise, is the activation function that perceptrons utilize the most frequently. The most common step function used in perceptron is the Heaviside step function:**

**h(z) = \begin{cases}0 & \text{ if } z< Threshold \\ 1 & \text{ if } z \geq Threshold\end{cases}**



**When all the neurons in a layer are connected to every neuron of the previous layer, it is known as a fully connected layer or dense layer.**

**The output of the fully connected layer can be:**

**$f_{W,b}(X) = h(XW + b)$**

**where X is the input W is the weight for each inputs neurons and b is the bias and h is the step function.**

**During training, The perceptron's weights are adjusted to minimize the difference between the predicted output and the actual output. Usually, supervised learning algorithms like the delta rule or the perceptron learning rule are used for this.**

**Here wi,j is the weight between the ith input and jth output neuron, xi is the ith input value, and yj and is the jth actual and predicted value is the learning rate.**

**Implementations code**

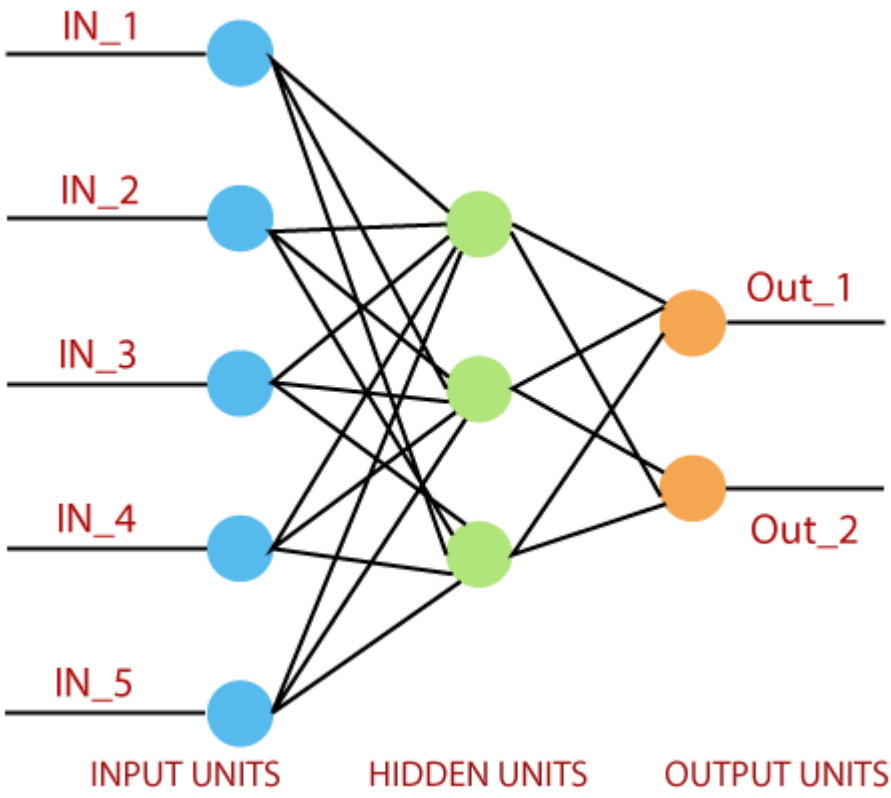
**Build the single Layer Perceptron Model**

- **Initialize the weight and learning rate, Here we are considering the weight values number of input + 1. i.e +1 for bias.**
- **Define the first linear layer**
- **Define the activation function. Here we are using the Heaviside Step function.**
- **Define the Prediction**
- **Define the loss function.**
- **Define training, in which weight and bias are updated accordingly.**
- **define fitting the model.**
- **Python3**

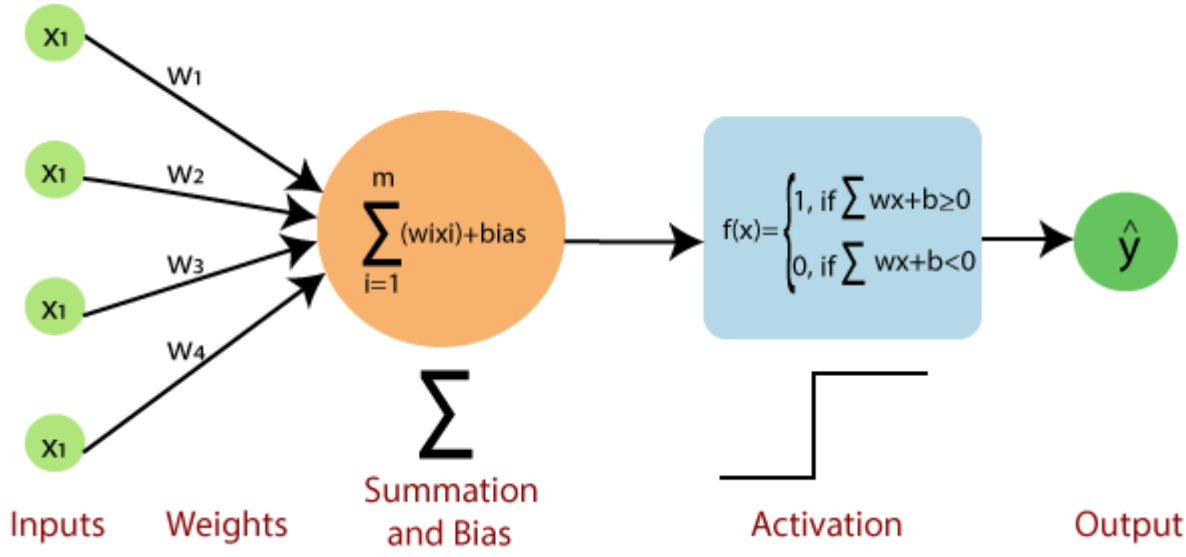
# Single Layer Perceptron

. perceptron is a single processing unit of any neural network. **Frank Rosenblatt** first proposed in **1958** is a simple neuron which is used to classify its input into one or two categories. Perceptron is a linear classifier, and is used in supervised learning. It helps to organize the given input data.

- A perceptron is a neural network unit that does a precise computation to detect features in the input data. Perceptron is mainly used to classify the data into two parts. Therefore, it is also known as **Linear Binary Classifier**.



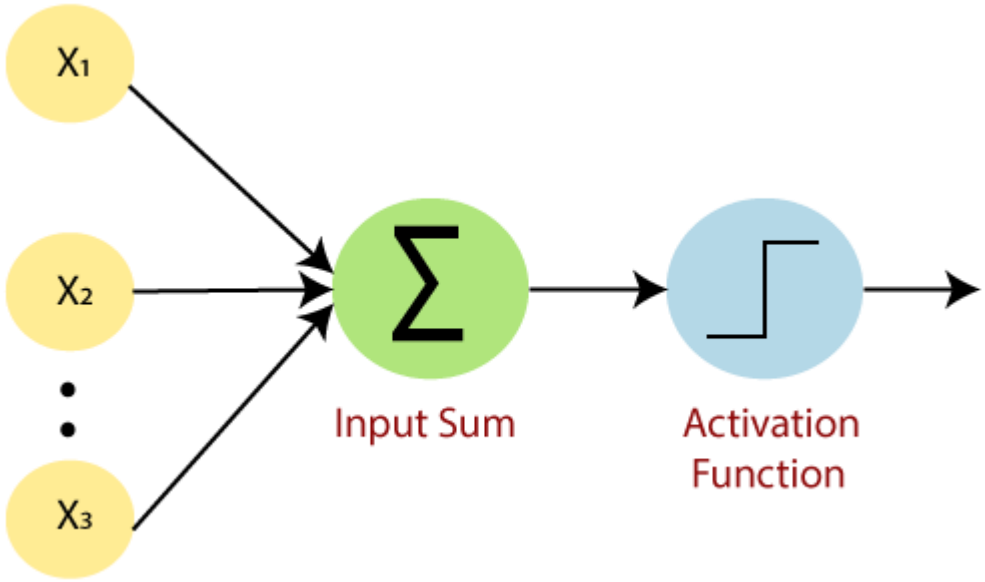

- Perceptron uses the step function that returns +1 if the weighted sum of its input 0 and -1.
- The activation function is used to map the input between the required value like (0, 1) or (-1, 1).




**The perceptron consists of 4 parts.**

- **Input value or One input layer: The input layer of the perceptron is made of artificial input neurons and takes the initial data into the system for further processing.**

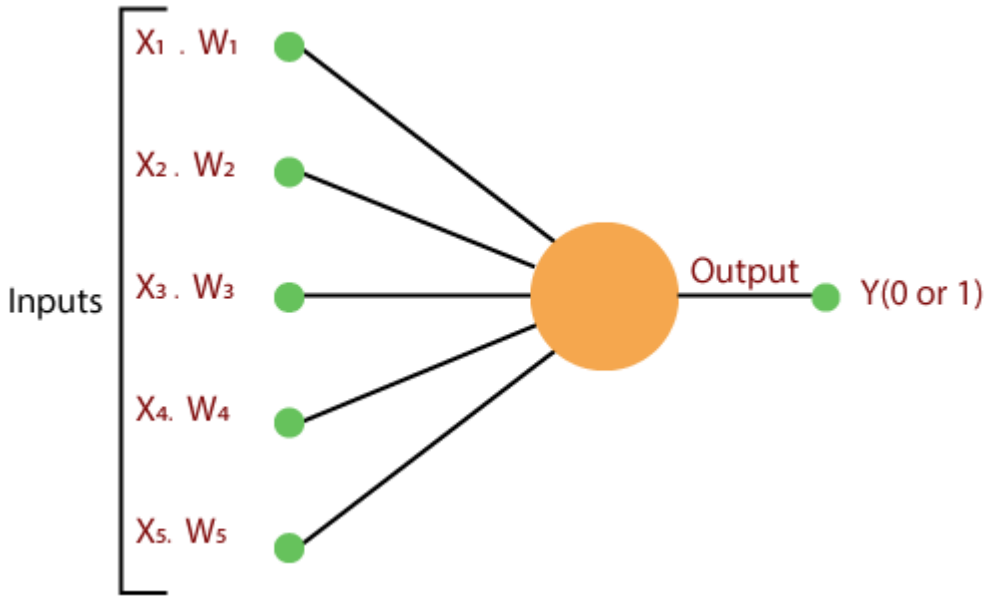
- **Weights and Bias:**
  **Weight: It represents the dimension or strength of the connection between units. If the weight to node 1 to node 2 has a higher quantity, then neuron 1 has a more considerable influence on the neuron.**
  **Bias: It is the same as the intercept added in a linear equation. It is an additional parameter which task is to modify the output along with the weighted sum of the input to the other neuron.**
- **Net sum: It calculates the total sum.**
- **Activation Function: A neuron can be activated or not, is determined by an activation function. The activation function calculates a weighted sum and further adding bias with it to give the result.**



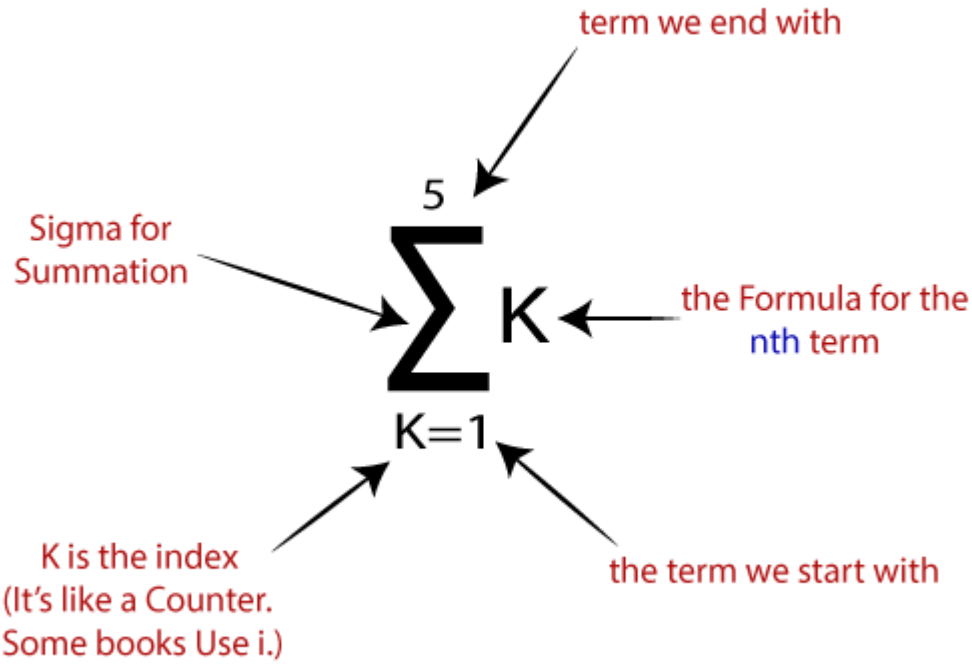

**How does it work?**

**The perceptron works on these simple steps which are given below:**

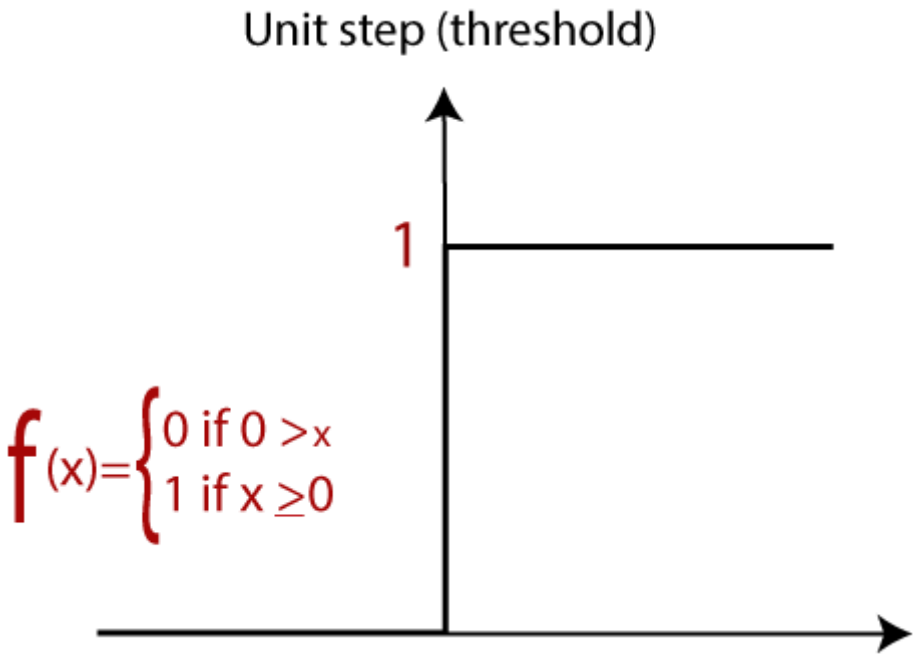
**a. In the first step, all the inputs x are multiplied with their weights w.**




**b. In this step, add all the increased values and call them the Weighted sum.**




**n our last step, apply the weighted sum to a correct Activation Function.**




**There are two types of architecture. These types focus on the functionality of artificial neural networks as follows-**

- **Single Layer Perceptron**
- **Multi-Layer Perceptron**

**Single Layer Perceptron**

**The single-layer perceptron was the first neural network model, proposed in 1958 by Frank Rosenbluth. It is one of the earliest models for learning. Our goal is to find a linear decision function measured by the weight vector w and the bias parameter b.**

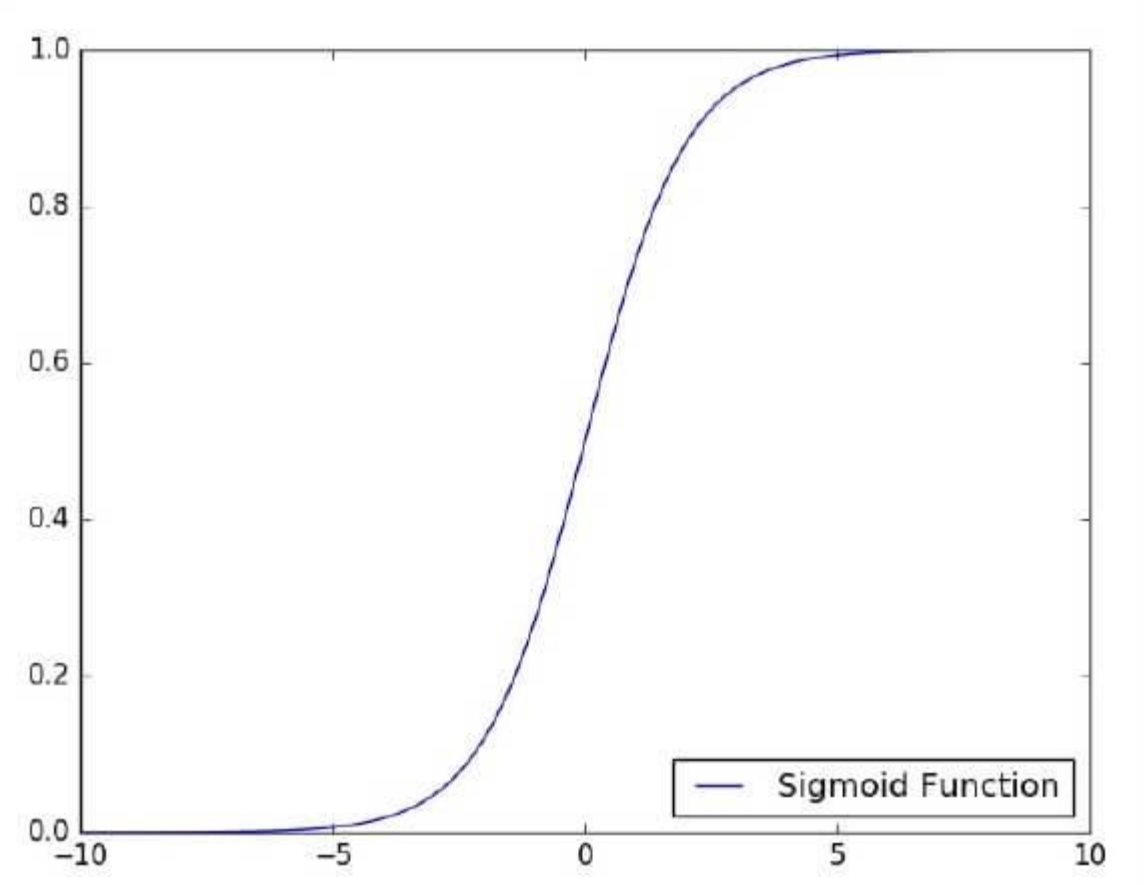
**To understand the perceptron layer, it is necessary to comprehend artificial neural networks (ANNs).**

**The artificial neural network (ANN) is an information processing system, whose mechanism is inspired by the functionality of biological neural circuits. An artificial neural network consists of several processing units that are interconnected.**

**This is the first proposal when the neural model is built. The content of the neuron's local memory contains a vector of weight.**

**The single vector perceptron is calculated by calculating the sum of the input vector multiplied by the corresponding element of the vector, with each increasing the amount of the corresponding component of the vector by weight. The value that is displayed in the output is the input of an activation function.**

**Let us focus on the implementation of a single-layer perceptron for an image classification problem using TensorFlow. The best example of drawing a single-layer perceptron is through the representation of "logistic regression."**




- **The weights are initialized with the random values at the origination of each training.**
- **For each element of the training set, the error is calculated with the difference between the desired output and the actual output. The calculated error is used to adjust the weight.**
- **The process is repeated until the fault made on the entire training set is less than the specified limit until the maximum number of iterations has been reached.**

**Complete code of Single layer perceptron**

```
# Import the MINST dataset
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_ ("/tmp/data/", one_hot=True)

import tensorflow as tf
import matplotlib.pyplot as plt
# Parameters
learning_rate = 0.01
```

```
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder("float", [none, 784]) # MNIST data image of shape 28*28 = 784
y = tf.placeholder("float", [none, 10]) # 0-9 digits recognition => 10 classes
# Create model
# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
# Constructing the model
activation=tf.nn.softmaxx(tf.matmul (x, W)+b) # Softmax
 of function
# Minimizing error using cross entropy
cross_entropy = y*tf.log(activation)
cost = tf.reduce_mean\ (-tf.reduce_sum\ (cross_entropy, reduction_indice = 1))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
#Plot settings
avg_set = []
epoch_set = []
# Initializing the variables where init = tf.initialize_all_variables()
# Launching the graph
with tf.Session() as sess:
   sess.run(init)

# Training of the cycle in  the dataset
   for epoch in range(training_epochs):
      avg_cost = 0.
      total_batch = int(mnist.train.num_example/batch_size)
```

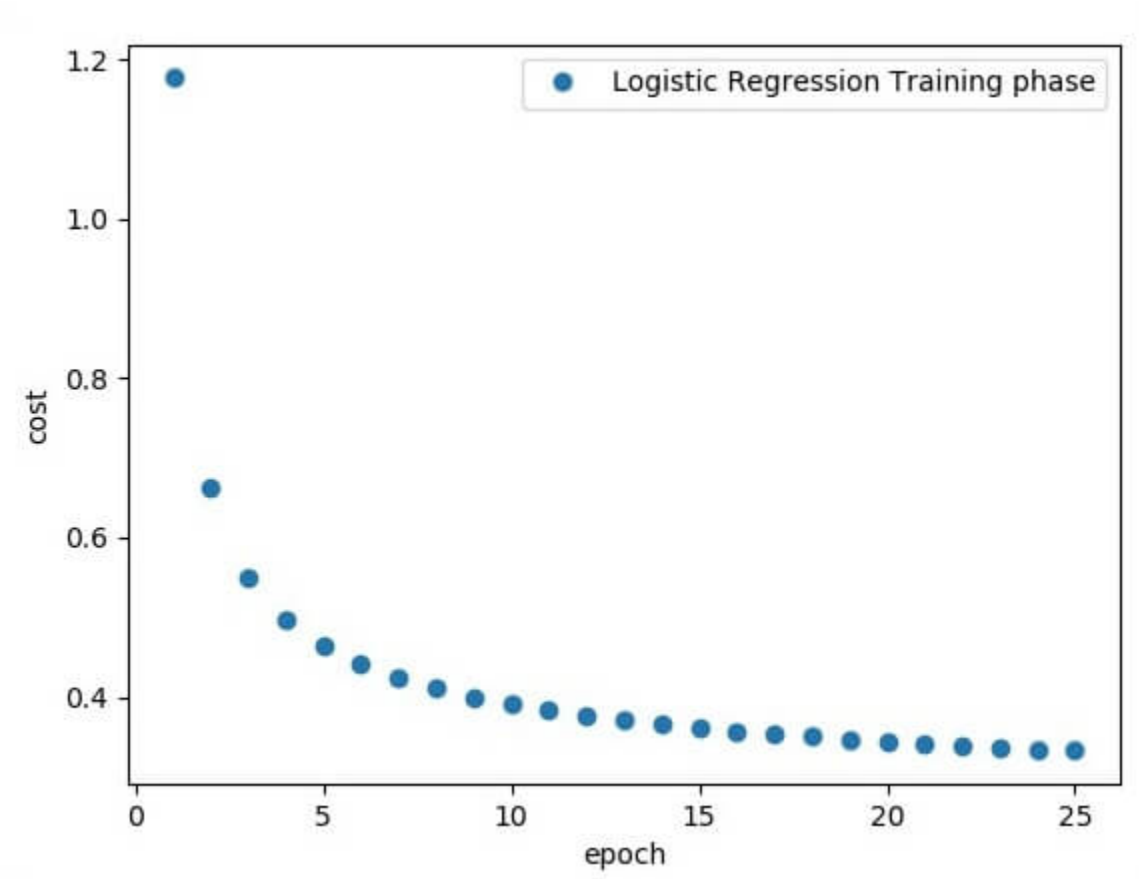
```
39.
40. # Creating loops at all the batches in the code
41.       for i in range(total_batch):
42. batch_xs, batch_ys = mnist.train.next_batch(batch_size)
43.          # Fitting the training by the batch data sess.run(optimizr,  feed_dict = {
44. x: batch_xs, y: batch_ys})
45.  # Compute all the average of loss avg_cost += sess.run(cost, \ feed_dict = {
46. x: batch_xs, \ y: batch_ys}) //total batch
47.       # Display the logs at each epoch steps
48.       if epoch % display_step==0:
49.       print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format (avg_cost))
50.            avg_set.append(avg_cost) epoch_set.append(epoch+1)
51.    print ("Training phase finished")
52.
53.    plt.plot(epoch_set,avg_set, 'o', label = 'Logistics Regression Training')
54.    plt.ylabel('cost')
55.    plt.xlabel('epoch')
56.    plt.legend()
57.    plt.show()
58.
59. # Test the model
60.    correct_prediction = tf.equal (tf.argmax (activation, 1),tf.argmax(y,1))
61.
62. # Calculating the accuracy of dataset
63. accuracy = tf.reduce_mean(tf.cast (correct_prediction, "float")) print
64. ("Model accuracy:", accuracy.eval({x:mnist.test.images, y: mnist.test.labels}))
```

**The output of the Code:**

```
Use `tf.global_variables_initializer` instead.
2018-07-09 11:41:19.926820: I T:\src\github\tensorflow\tensorflow\core\platform\cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was
 not compiled to use: AVX2
Epoch: 0001 cost= 1.176560601
Epoch: 0002 cost= 0.662504855
Epoch: 0003 cost= 0.550647454
Epoch: 0004 cost= 0.496803975
Epoch: 0005 cost= 0.463813413
Epoch: 0006 cost= 0.440974006
Epoch: 0007 cost= 0.423977673
Epoch: 0008 cost= 0.410635707
Epoch: 0009 cost= 0.399920001
Epoch: 0010 cost= 0.390970191
Epoch: 0011 cost= 0.383367628
Epoch: 0012 cost= 0.376823489
Epoch: 0013 cost= 0.371032368
Epoch: 0014 cost= 0.365939091
Epoch: 0015 cost= 0.361380407
Epoch: 0016 cost= 0.357227336
Epoch: 0017 cost= 0.353001805
Epoch: 0018 cost= 0.350137204
Epoch: 0019 cost= 0.347027825
Epoch: 0020 cost= 0.344160418
Epoch: 0021 cost= 0.341485278
Epoch: 0022 cost= 0.339003812
Epoch: 0023 cost= 0.336675840
Epoch: 0024 cost= 0.334456453
Epoch: 0025 cost= 0.332455397
Training phase finished
Model accuracy: 0.9135
```



**The logistic regression is considered as predictive analysis. Logistic regression is mainly used to describe data and use to explain the relationship between the dependent binary variable and one or many nominal or independent variables.**

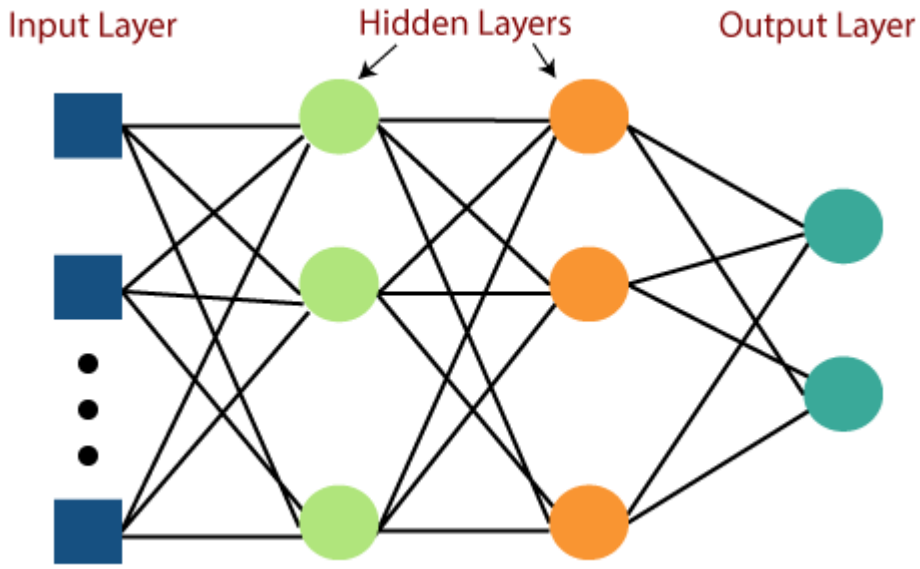



**Note: Weight shows the strength of the particular node.**

## • Multi-layer Perceptron

- **Multi-Layer perceptron defines the most complex architecture of artificial neural networks. It is substantially formed from multiple layers of the perceptron.**

**TensorFlow is a very popular deep learning framework released by, and this notebook will guide to build a neural network with this library. If we want to understand what is a Multi-layer perceptron, we have to develop a multi-layer perceptron from scratch using Numpy.**




**MLP networks are used for supervised learning format. A typical learning algorithm for MLP networks is also called back propagation's algorithm.**

**A multilayer perceptron (MLP) is a feed forward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input nodes connected as a directed graph between the input and output layers. MLP uses backpropagation for training the network. MLP is a deep learning method.**

**implementation with MLP for an image classification problem.**

```
# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot = True)

import tensorflow as tf
import matplotlib.pyplot as plt

# Parameters
learning_rate = 0.001
training_epochs = 20
batch_size = 100
```

```
12. display_step = 1
13.
14. # Network Parameters
15. n_hidden_1 = 256
16.
17. # 1st layer num features
18. n_hidden_2 = 256 # 2nd layer num features
19. n_input = 784 # MNIST data input (img shape: 28*28) n_classes = 10
20. # MNIST total classes (0-9 digits)
21.
22. # tf Graph input
23. x = tf.placeholder("float", [None, n_input])
24. y = tf.placeholder("float", [None, n_classes])
25.
26. # weights layer 1
27. h = tf.Variable(tf.random_normal([n_input, n_hidden_1])) # bias layer 1
28. bias_layer_1 = tf.Variable(tf.random_normal([n_hidden_1]))
29. # layer 1 layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, h), bias_layer_1))
30.
31. # weights layer 2
32. w = tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2]))
33.
34. # bias layer 2
35. bias_layer_2 = tf.Variable(tf.random_normal([n_hidden_2]))
36.
37. # layer 2
38. layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, w), bias_layer_2))
39.
40. # weights output layer
41. output = tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
```

```
42.
43. # biar output layer
44. bias_output = tf.Variable(tf.random_normal([n_classes])) # output layer
45. output_layer = tf.matmul(layer_2, output) + bias_output
46.
47. # cost function
48. cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
49.     logits = output_layer, labels = y))
50.
51. #cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(output_layer, y)
    )
52. # optimizer
53. optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(cos
    t)
54.
55. # optimizer = tf.train.GradientDescentOptimizer(
56.     learning_rate = learning_rate).minimize(cost)
57.
58. # Plot settings
59. avg_set = []
60. epoch_set = []
61.
62. # Initializing the variables
63. init = tf.global_variables_initializer()
64.
65. # Launch the graph
66. with tf.Session() as sess:
67.     sess.run(init)
68.
69.     # Training cycle
70.     for epoch in range(training_epochs):
```

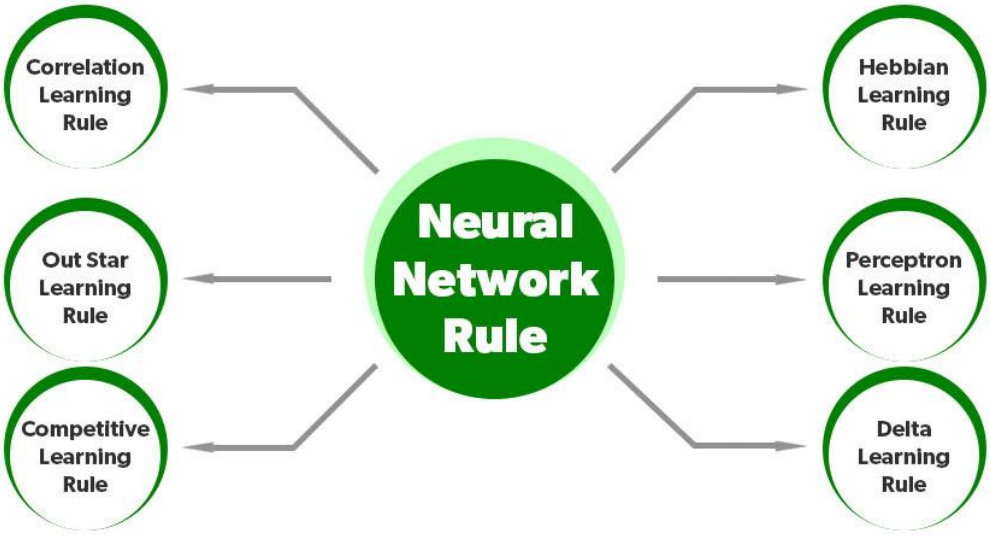
```
71.     avg_cost = 0.
72.     total_batch = int(mnist.train.num_examples / batch_size)
73.
74.     # Loop over all batches
75.     for i in range(total_batch):
76.       batch_xs, batch_ys = mnist.train.next_batch(batch_size)
77.       # Fit training using batch data sess.run(optimizer, feed_dict = {
78.         x: batch_xs, y: batch_ys})
79.       # Compute average loss
80.       avg_cost += sess.run(cost, feed_dict = {x: batch_xs, y: batch_ys}) / total_bat
    ch
81.     # Display logs per epoch step
82.     if epoch % display_step == 0:
83.       print
84.       Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost)
85.     avg_set.append(avg_cost)
86.     epoch_set.append(epoch + 1)
87.   print
88.   "Training phase finished"
89.
90.   plt.plot(epoch_set, avg_set, 'o', label = 'MLP Training phase')
91.   plt.ylabel('cost')
92.   plt.xlabel('epoch')
93.   plt.legend()
94.   plt.show()
95.
96.   # Test model
97.   correct_prediction = tf.equal(tf.argmax(output_layer, 1), tf.argmax(y, 1))
98.
99.   # Calculate accuracy
100.          accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

```
print
"Model Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.la
bels})
```

**The above line of codes generating the following output-**

# Perceptron Learning Rule

**Learning rule enhances the Artificial Neural Network's performance by applying this rule over the network. Thus learning rule updates the weights and bias levels of a network when certain conditions are met in the training process. it is a crucial part of the development of the Neural Network.**




**1. Hebbian Learning Rule**

**Donald Hebb developed it in 1949 as an unsupervised learning algorithm in the neural network. We can use it to improve the weights of nodes of a network. The following phenomenon occurs when**

- **If two neighbor neurons are operating in the same phase at the same period of time, then the weight between these neurons should increase.**
- **For neurons operating in the opposite phase, the weight between them should decrease.**
- **If there is no signal correlation, the weight does not change, the sign of the weight between two nodes depends on the sign of the input between those nodes**
- **When inputs of both the nodes are either positive or negative, it results in a strong positive weight.**
- **If the input of one node is positive and negative for the other, a strong negative weight is present.**

**Mathematical Formulation:**

**δw=αxiy**

**where δw=change in weight,*α* is the learning rate.xi the input vector,y the output.**

**2. Perceptron Learning Rule**

**It was introduced by Rosenblatt. It is an error-correcting rule of a single-layer feedforward network. it is supervised in nature and calculates the error between the desired and actual output and if the output is present then only adjustments of weight are done.**

**Computed as follows:**

***Assume (x1,x2,x3……..xn) –>set of input vectors***

***and (w1,w2,w3……...wn) –>set of weights***

***y=actual output***

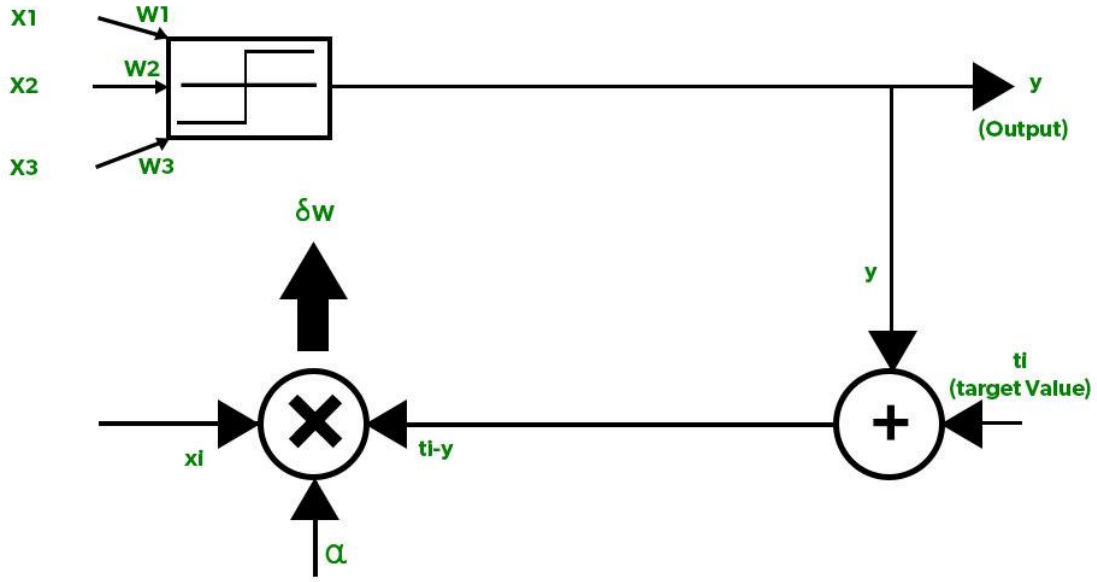
***wo=initial weight***

***wnew=new weight***

***δw=change in weight***

***α=learning rate***

***actual output(y)=wixi***




**3. Delta Learning Rule**

**It was developed by Bernard Widrow and Marcian Hoff and It depends on supervised learning and has a continuous activation function. It is also known as the Least Mean Square method and it minimizes error over all the training patterns.**

**It is based on a gradient descent approach which continues forever. It states that the modification in the weight of a node is equal to the product of the error and the input where the error is the difference between desired and actual output.**

**Computed as follows:**

***Assume (x1,x2,x3……..xn) –>set of input vectors***

***and (w1,w2,w3……...wn) –>set of weights***

***y=actual output***

***wo=initial weight***

***wnew=new weight***

***δw=change in weight***

***Error= ti-y***

***Learning signal(ej)=(ti-y)y'***

***y=f(net input)= ∫wixi***

***δw=αxiej=αxi(ti-y)y'***

***wnew=wo+δw***

***The updating of weights can only be done if there is a difference between the target and actual output(i.e., error) present:***

***case I: when t=y***

***then there is no change in weight***

***case II: else***

***wnew=wo+δw***

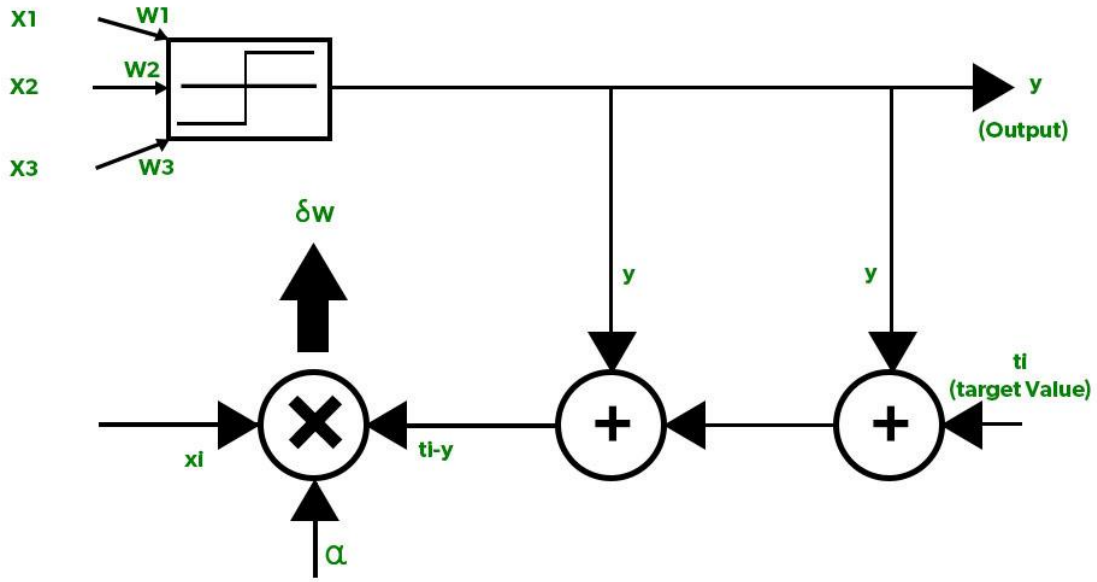
**4. Correlation Learning Rule**

**The correlation learning rule follows the same similar principle as the Hebbian learning rule,i.e., If two neighbor neurons are operating in the same phase at the same period of time, then the weight between these neurons should be more positive. For neurons operating in the opposite phase, the weight between them should be more negative but unlike the Hebbian rule, the correlation rule is supervised in nature here, the targeted response is used for the calculation of the change in weight.**

**In Mathematical form:**

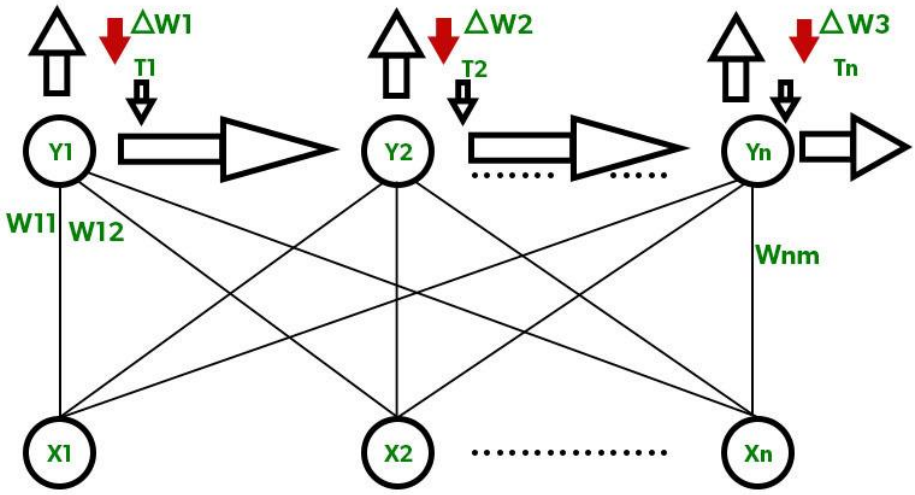
***δw=αxitj***

***where δw=change in weight,α=learning rate,xi=set of the input vector, and tj=target value***




**5. Out Star Learning Rule**

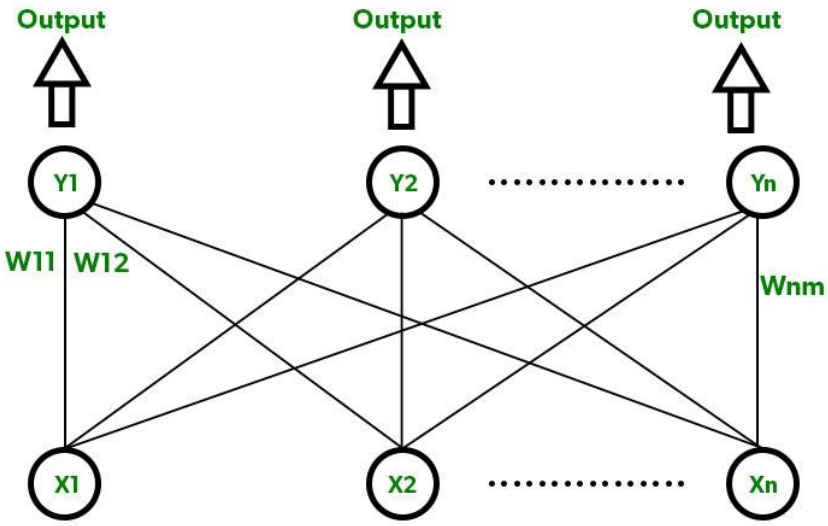
**It was introduced by Grossberg and is a supervised training procedure.**




**Out Star Learning Rule is implemented when nodes in a network are arranged in a layer. Here the weights linked to a particular node should be equal to the targeted outputs for the nodes connected through those same weights. Weight change is thus calculated as=δw=α(t-y)**

**Where α=learning rate, y=actual output, and t=desired output for n layer nodes.**

**6.Competitive Learning Rule**

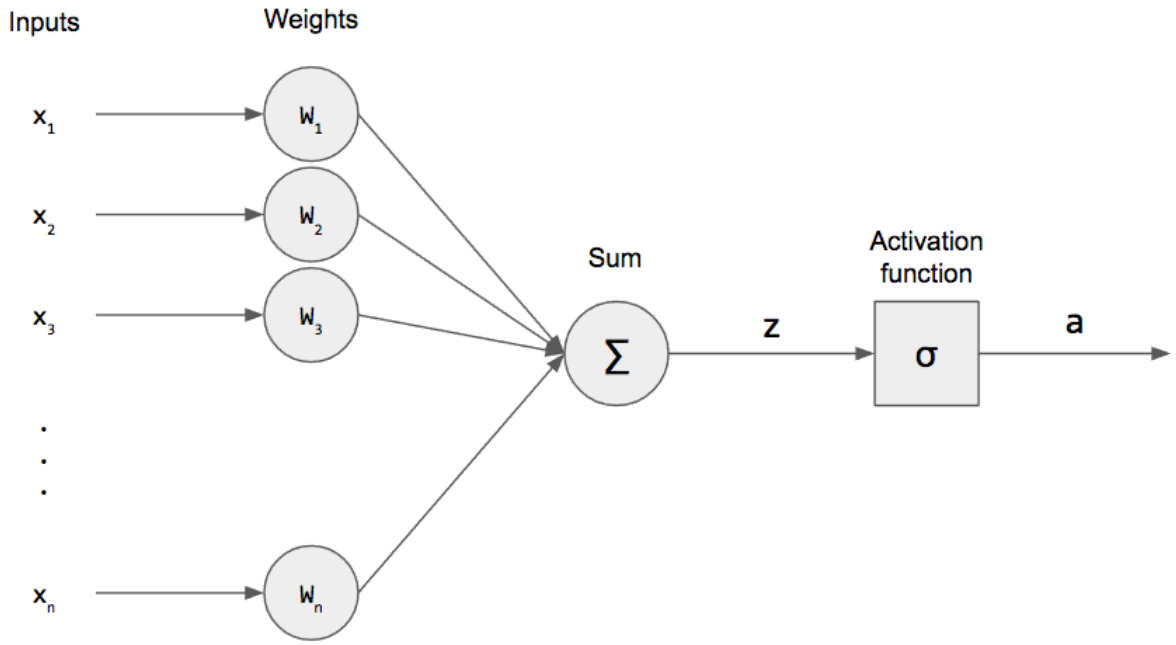



**It is also known as the Winner-takes-All rule and is unsupervised in nature. Here all the output nodes try to compete with each other to represent the input pattern and the winner is declared according to the node having the most outputs and is given the output 1 while the rest are given 0.**

**There are a set of neurons with arbitrarily distributed weights and the activation function is applied to a subset of neurons. Only one neuron is active at a time. Only the winner has updated weights, the rest remain unchanged.**

# Training multi neournos perceptron

**** Neural Representation of AND, OR, NOT, XOR and XNOR Logic Gates (Perceptron Algorithm)**



**First, we need to know that the Perceptron algorithm states that:**

***Prediction (y`) = 1 if Wx+b > 0 and 0 if Wx+b ≤ 0***

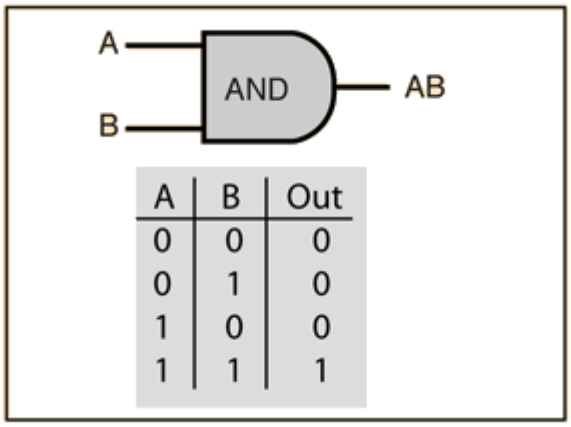
**Also, the steps in this method are very similar to how Neural Networks learn, which is as follows;**

- **Initialize weight values and bias**
- **Forward Propagate**
- **Check the error**
- **Backpropagate and Adjust weights and bias**
- **Repeat for all training examples**

**Now that we know the steps, let's get up and running:**

**AND Gate**

**From our knowledge of logic gates, we know that an AND logic table is given by the diagram below**




| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**First, we need to understand that the output of an AND gate is 1 only if both inputs (in this case, x1 and x2) are 1. So, following the steps listed above;**

**Row 1**

- **From w1*x1+w2*x2+b, initializing w1, w2, as 1 and b as –1, we get;**

***x1(1)+x2(1)–1***

- **Passing the first row of the AND logic table (x1=0, x2=0), we get;**

***0+0–1 = –1***

- **From the Perceptron rule, if Wx+b≤0, then y`=0. Therefore, this row is correct, and no need for Backpropagation.**

**Row 2**

- **Passing (x1=0 and x2=1), we get;**

*0+1–1 = 0*

- **From the Perceptron rule, if Wx+b≤0, then y`=0. This row is correct, as the output is 0 for the AND gate.**
- **From the Perceptron rule, this works (for both row 1, row 2 and 3).**

**Row 4**

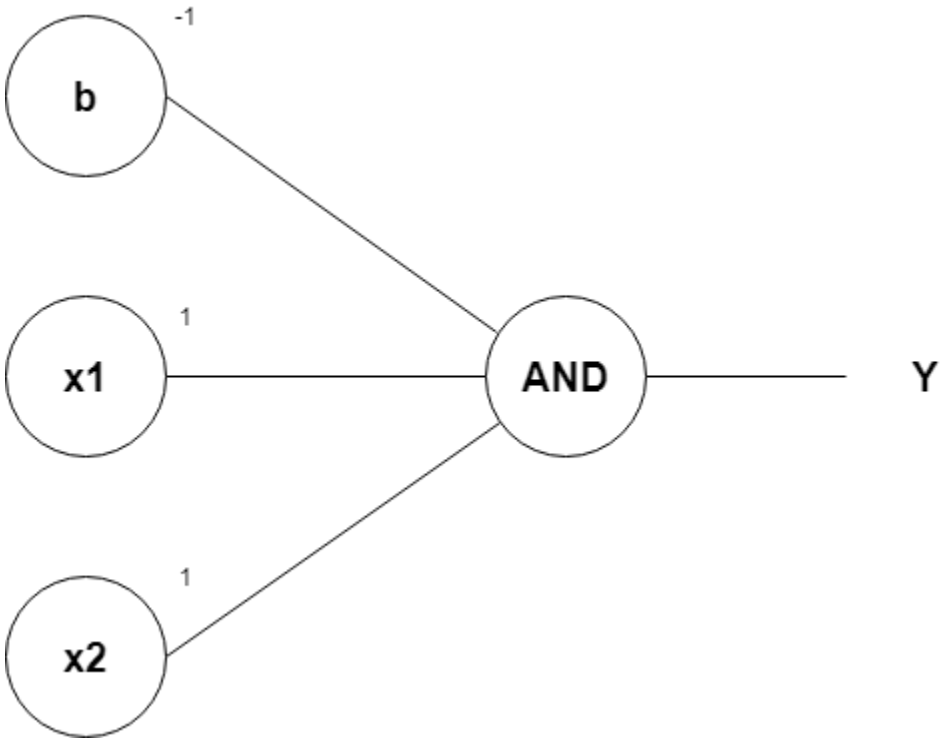
- **Passing (x1=1 and x2=1), we get;**

*1+1–1 = 1*

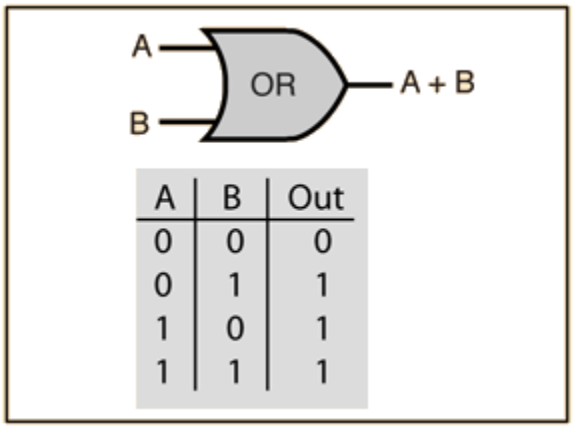
- **Again, from the perceptron rule, this is still valid.**

**Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is;**

*x1+x2–1*




**OR Gate**




| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**From the diagram, the OR gate is 0 only if both inputs are 0.**

**Row 1**

- **From w1x1+w2x2+b, initializing w1, w2, as 1 and b as –1, we get;**

*x1(1)+x2(1)–1*

- **Passing the first row of the OR logic table (x1=0, x2=0), we get;**

*0+0–1 = –1*

- **From the Perceptron rule, if Wx+b≤0, then y`=0. Therefore, this row is correct.**

**Row 2**

- **Passing (x1=0 and x2=1), we get;**

*0+1–1 = 0*

- **From the Perceptron rule, if Wx+b <= 0, then y`=0. Therefore, this row is incorrect.**
- **So we want values that will make inputs x1=0 and x2=1 give y` a value of 1. If we change w2 to 2, we have;**

*0+2–1 = 1*

- **From the Perceptron rule, this is correct for both the row 1 and 2.**

**Row 3**

- **Passing (x1=1 and x2=0), we get;**

*1+0–1 = 0*

- **From the Perceptron rule, if Wx+b <= 0, then y`=0. Therefore, this row is incorrect.**
- **Since it is similar to that of row 2, we can just change w1 to 2, we have;**

*2+0–1 = 1*

- **From the Perceptron rule, this is correct for both the row 1, 2 and 3.**

**Row 4**

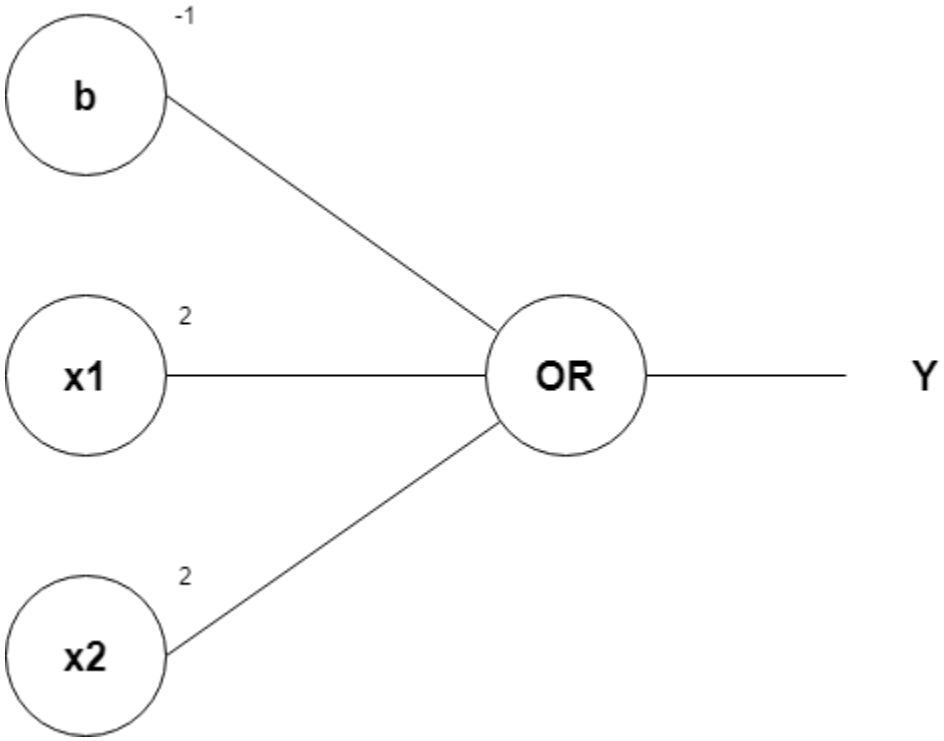
- **Passing (x1=1 and x2=1), we get;**

*2+2–1 = 3*

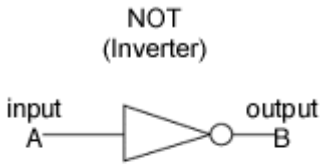
- **Again, from the perceptron rule, this is still valid. Quite Easy!**

**Therefore, we can conclude that the model to achieve an OR gate, using the Perceptron algorithm is;**

*2x1+2x2–1*




**NOT Gate**




| A | B |
|---|---|
| 0 | 1 |
| 1 | 0 |

**rom the diagram, the output of a NOT gate is the inverse of a single input. So, following the steps listed above;**

**Row 1**

- **From w1x1+b, initializing w1 as 1 (since single input), and b as –1, we get;**

***x1(1)–1***

- **Passing the first row of the NOT logic table (x1=0), we get;**

***0–1 = –1***

- **From the Perceptron rule, if Wx+b≤0, then y`=0. This row is incorrect, as the output is 1 for the NOT gate.**
- **So we want values that will make input x1=0 to give y` a value of 1. If we change b to 1, we have;**

***0+1 = 1***

- **From the Perceptron rule, this works.**

**Row 2**

- **Passing (x1=1), we get;**

***1+1 = 2***

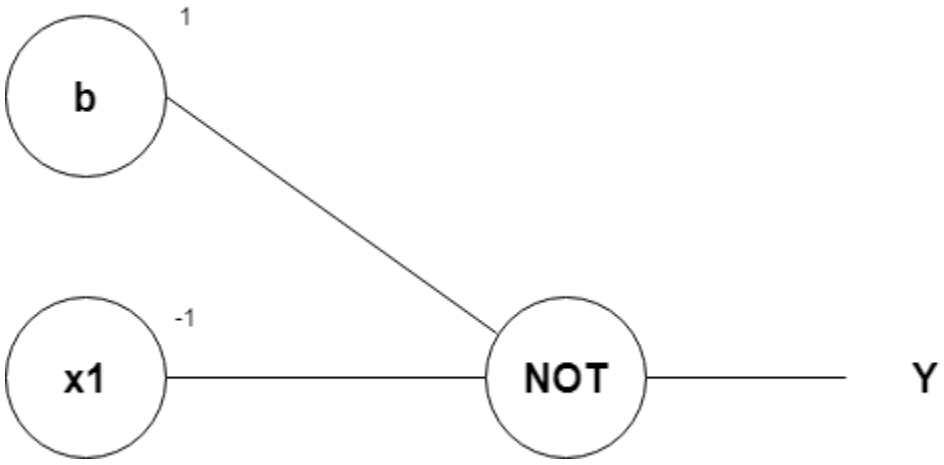
- **From the Perceptron rule, if Wx+b > 0, then y`=1. This row is so incorrect, as the output is 0 for the NOT gate.**
- **So we want values that will make input x1=1 to give y` a value of 0. If we change w1 to –1, we have;**

***–1+1 = 0***

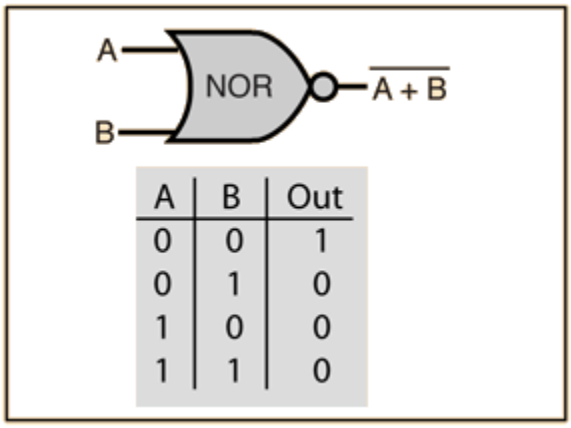
- **From the Perceptron rule, if Wx+b ≤ 0, then y`=0. Therefore, this works (for both row 1 and row 2).**

**Therefore, we can conclude that the model to achieve a NOT gate, using the Perceptron algorithm is;**

***–x1+1***




**NOR Gate**




| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

- **Complexity of perceptron learning**

**The perceptron learning algorithm is a foundational concept in machine learning and neural networks. Its complexity can be analyzed from different perspectives: time complexity, space complexity, and convergence properties.**

**Time Complexity**

**The time complexity of the perceptron learning algorithm depends on a few factors:**

1. **Number of Iterations: In the worst case, the perceptron algorithm might need a large number of iterations to converge, especially if the data is not linearly separable or if the data is difficult to separate. In practice, the number of iterations required for convergence is influenced by the specific dataset and can be variable.**
2. **Number of Features (d): Each iteration involves updating weights based on the feature values of the data point. If there are ddd features, updating the weights takes O(d)O(d)O(d) time per iteration.**
3. **Number of Data Points (n): Each iteration of the algorithm may involve processing all nnn data points, so for each iteration, the time complexity is O(n·d)O(n \cdot d)O(n·d).**

**Given TTT as the total number of iterations needed for convergence, the overall time complexity can be expressed as O(T·n·d)O(T \cdot n \cdot d)O(T·n·d). However, in practice, the number of iterations TTT can vary widely depending on the dataset and its characteristics.**

**Space Complexity**

**The space complexity of the perceptron algorithm is relatively straightforward:**

1. **Weights Vector: The space required to store the weights is O(d)O(d)O(d), where ddd is the number of features.**
2. **Data Storage: The space required to store the dataset itself is O(n·d)O(n \cdot d)O(n·d), where nnn is the number of data points and ddd is the number of features.**

**Overall, the space complexity is O(n·d)O(n \cdot d)O(n·d), assuming the dataset is stored in memory.**

**Convergence Properties**

**The convergence of the perceptron algorithm is guaranteed for linearly separable data. In this case, the number of iterations required to find a solution**

**is bounded, but the bound depends on the margin between the classes and the distribution of the data. Specifically, if the data is linearly separable, the number of updates required is O(R2γ2)O\left(\frac{R^2}{\gamma^2}\right)O(γ2R2), where RRR is the radius of the data (maximum norm of any data point) and γ\gammaγ is the margin (the smallest distance between the decision boundary and any data point).**

**In summary:**

- **Time Complexity: O(T·n·d)O(T \cdot n \cdot d)O(T·n·d)**
- **Space Complexity: O(n·d)O(n \cdot d)O(n·d)**
- **Convergence: Guaranteed for linearly separable data, with the number of iterations depending on the margin and data distribution.**

- # computational limits of perceptron

**1. Linearly Separable Data Requirement**

**The perceptron algorithm can only find a solution for data that is linearly separable. If the data is not linearly separable, the perceptron algorithm will not converge, and the weights will continue to be updated indefinitely without reaching a stable solution. This limits its applicability to problems where linear boundaries can adequately separate the classes.**

**2. Convergence Issues**

**Even though the perceptron algorithm guarantees convergence for linearly separable data, the number of iterations required can be very large, especially if the margin (the distance between the decision boundary and the closest data point) is small. In practice, this can lead to slow convergence.**

**3. No Probabilistic Output**

**The perceptron model provides deterministic binary classification outputs (either 0 or 1) and does not offer probabilistic interpretations or confidence levels. For many applications, especially those involving uncertainty or risk assessment, probabilistic outputs are crucial.**

**4. Limited Capacity**

**The perceptron has a limited capacity as it can only model linear decision boundaries. It is unable to capture more complex relationships or interactions between features. For tasks involving non-linear patterns, the perceptron is insufficient. This limitation led to the development of more advanced models like multi-layer perceptrons (MLPs) or support vector machines (SVMs), which can handle non-linear decision boundaries.**

**5. Feature Scaling Sensitivity**

**The perceptron algorithm can be sensitive to the scale of the input features. Features with larger ranges or different scales can disproportionately influence the weight updates, which might affect the convergence and performance of the algorithm. Proper feature scaling or normalization is often required to mitigate this issue.**

**6. Lack of Regularization**

**The basic perceptron algorithm does not include regularization to prevent overfitting. In scenarios where the data is noisy or where there are many features relative to the number of samples, the perceptron can overfit the training data.**

**Regularized variants, like the Perceptron with weight decay, can help, but they are not part of the basic algorithm.**

**7. No Hidden Layers**

**The basic perceptron is a single-layer model. It does not have the capability to represent more complex functions that require multiple layers or non-linear transformations. Multi-layer perceptrons (MLPs) with hidden layers, also known as feedforward neural networks, are required to capture more complex patterns.**

**8. No Consideration for Data Imbalance**

**The perceptron algorithm does not inherently account for class imbalance in the data. If one class is significantly underrepresented, the perceptron might be biased towards the majority class, leading to poor performance on the minority class.**

**9. Computational Efficiency with Large Datasets**

**For very large datasets, the perceptron algorithm may become computationally expensive. Each iteration requires processing the entire dataset, which can be inefficient for large-scale problems. Stochastic gradient descent (SGD) is often used to improve efficiency, but the basic perceptron may still struggle with large-scale data.**

# linearly separable functions

**1. Single-Layer Perceptrons**

**A single-layer perceptron can only solve linearly separable problems. This means that it can find a linear decision boundary that separates classes if and only if the data is linearly separable. Mathematically, for a binary classification problem, the decision boundary is a hyperplane defined by a linear combination of input features.**

- **Linear Separability: Data is linearly separable if there exists a hyperplane that can completely separate the classes without error. For instance, data points belonging to class 1 are on one side of the hyperplane, and data points of class 2 are on the other side.**
- **Limitations: For problems where the classes are not linearly separable, a single-layer perceptron cannot find a suitable decision boundary. For example, the XOR problem is a classic example of a non-linearly separable problem.**

**2. Multi-Layer Perceptrons (MLPs)**

**Multi-layer perceptrons (MLPs), or feedforward neural networks with one or more hidden layers, can handle both linearly separable and non-linearly separable problems. Here's how they address linear separability:**

- **Hidden Layers and Non-Linearity: The introduction of one or more hidden layers, along with non-linear activation functions (e.g., ReLU, sigmoid, tanh), allows MLPs to model complex, non-linear decision boundaries. Each hidden layer applies non-linear transformations to the input data, enabling the network to learn and approximate non-linear functions.**
- **Universal Approximation Theorem: This theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function to a desired level of accuracy, given appropriate activation functions and sufficient neurons. This means that MLPs**

**can represent complex decision boundaries, including those needed for non-linearly separable problems.**

- **Complex Decision Boundaries: With multiple layers, the network can learn hierarchical features and build complex decision boundaries. Each layer can capture different levels of abstraction, which helps in effectively separating classes even when they are not linearly separable.**

**3. Convolutional Neural Networks (CNNs)**

**For image and spatial data, convolutional neural networks (CNNs) extend the idea of MLPs by introducing convolutional layers that automatically learn spatial hierarchies of features.**

- **Local Features and Pooling: CNNs can learn local features through convolutional layers and then aggregate these features through pooling layers. This process helps in capturing patterns and spatial hierarchies in data, making CNNs effective for tasks like image classification, where the concept of separability is extended to higher-dimensional spaces.**

**4. Recurrent Neural Networks (RNNs)**

**For sequential and time-series data, recurrent neural networks (RNNs) and their advanced variants (like LSTMs and GRUs) handle data that has temporal dependencies.**

- **Sequential Learning: RNNs process data in sequences and learn dependencies over time, allowing them to handle complex patterns that are not linearly separable in the traditional sense.**

- ****Single-Layer Perceptrons: Can only handle linearly separable problems.**
- **Multi-Layer Perceptrons (MLPs): Can solve both linearly and non-linearly separable problems by learning complex, non-linear decision boundaries through hidden layers and non-linear activation functions.**
- **Convolutional Neural Networks (CNNs): Extend the concept of learning to spatial hierarchies in image data, capturing complex patterns.**
- **Recurrent Neural Networks (RNNs): Address sequential data and temporal dependencies, handling complex, non-linear patterns over time.**

**In essence, while single-layer perceptrons are limited to linearly separable problems, more complex neural network architectures like MLPs, CNNs, and RNNs can model a broad range of functions, including those involving non-linear and hierarchical patterns.**

# learning XOR-feed forward network

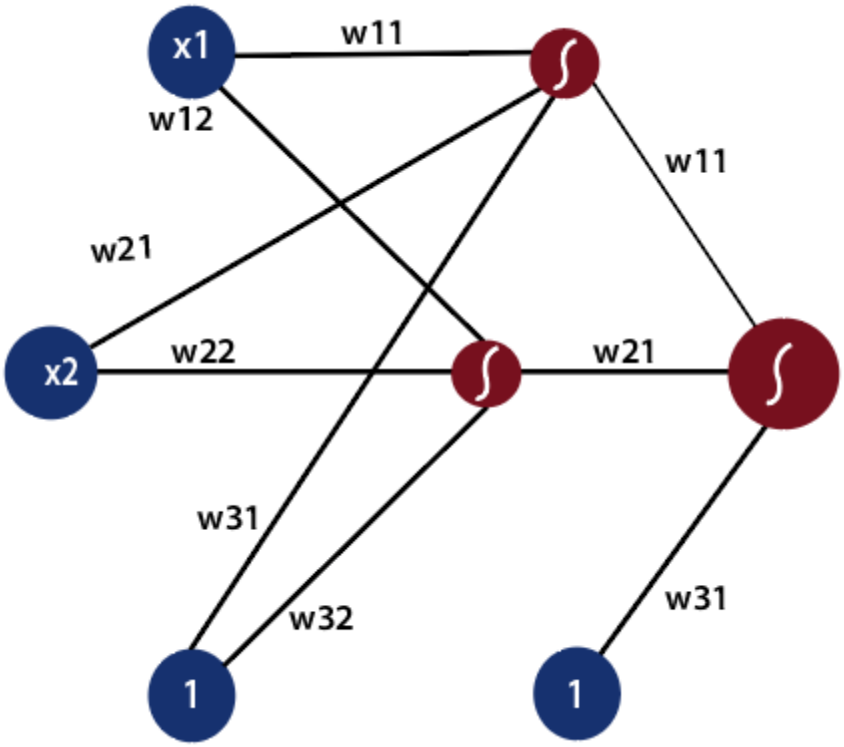
**Feed Forward Process in Deep Neural Network**

**Now, we know how with the combination of lines with different weight and biases can result in non-linear models. How does a neural network know what weight and biased values to have in each layer? It is no different from how we did it for the single based perceptron model.**

**We are still making use of a gradient descent optimization algorithm which acts to minimize the error of our model by iteratively moving in the direction with the steepest descent, the direction which updates the parameters of our model while ensuring the minimal error. It updates the weight of every model in every single**

**layer. We will talk more about optimization algorithms and backpropagation later.**
**It is important to recognize the subsequent training of our neural network. Recognition is done by dividing our data samples through some decision boundary.**
**"The process of receiving an input to produce some kind of output to make some kind of prediction is known as Feed Forward." Feed Forward neural network is the core of many other important neural networks such as convolution neural network.**

**In the feed-forward neural network, there are not any feedback loops or connections in the network. Here is simply an input layer, a hidden layer, and an output layer.**




**There can be multiple hidden layers which depend on what kind of data you are dealing with. The number of hidden layers is known as the depth of the neural network. The deep neural network can learn from more functions. Input layer first provides the neural network with data and the output layer then make predictions on that data which is based on a series of functions. ReLU Function is the most commonly used activation function in the deep neural network.**
**To gain a solid understanding of the feed-forward process, let's see this mathematically.**
**1) The first input is fed to the network, which is represented as matrix x1, x2, and one where one is the bias value.**

$$[x_1 \quad x_2 \quad 1]$$

**2) Each input is multiplied by weight with respect to the first and second model to obtain their probability of being in the positive region in each model.**

**So, we will multiply our inputs by a matrix of weight using matrix multiplication.**

$$[x_1 \quad x_2 \quad 1] = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = [\text{score} \quad \text{score}]$$

**3) After that, we will take the sigmoid of our scores and gives us the probability of the point being in the positive region in both models.**

$$\frac{1}{1+e^{-x}}[\text{score score}] = \text{probability}$$

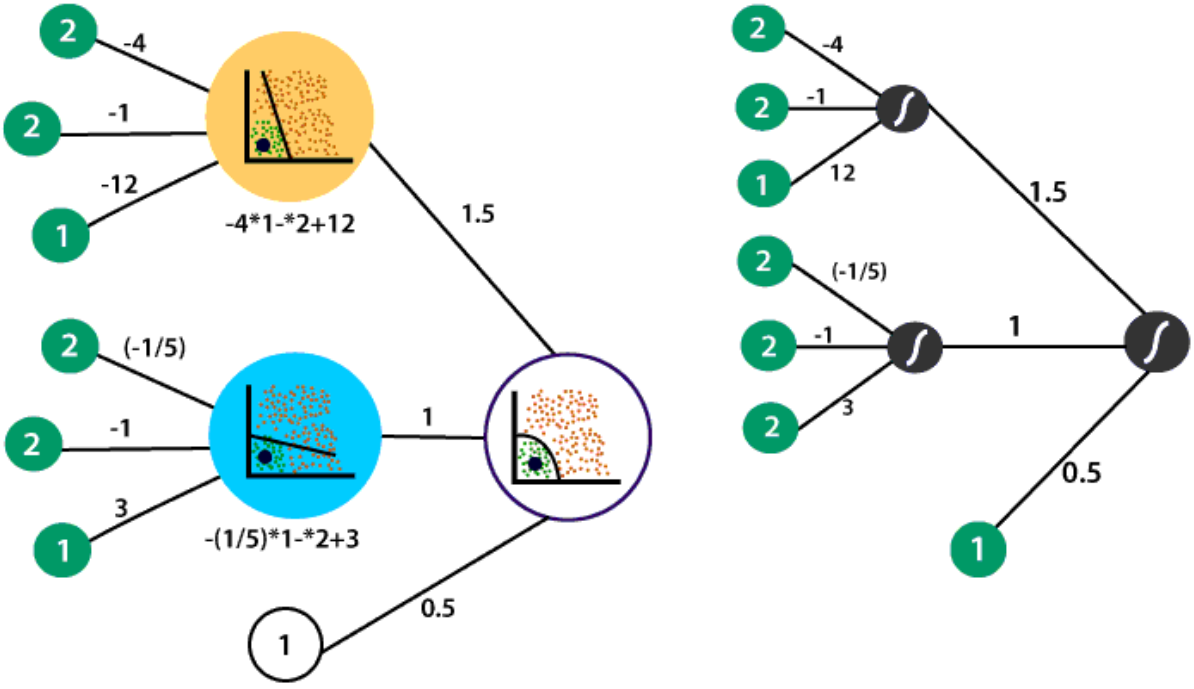
**4) We multiply the probability which we have obtained from the previous step with the second set of weights. We always include a bias of one whenever taking a combination of inputs.**

$$[\text{probability} \quad \text{probability} \quad 1] \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} = [\text{score}]$$

**And as we know to obtain the probability of the point being in the positive region of this model, we take the sigmoid and thus producing our final output in a feed-forward process.**

$$\frac{1}{1+e^{-x}}[\text{score}] = [\text{probability}]$$

**Let takes the neural network which we had previously with the following linear models and the hidden layer which combined to form the non-linear model in the output layer.**




**So, what we will do we use our non-linear model to produce an output that describes the probability of the point being in the positive region. The point was represented by 2 and 2. Along with bias, we will represent the input as**

$$[2 \quad 2 \quad 1]$$

**The first linear model in the hidden layer recall and the equation defined it**

$$-4x_1 - x_2 + 12$$

**Which means in the first layer to obtain the linear combination the inputs are multiplied by -4, -1 and the bias value is multiplied by twelve.**

$$[2 \quad 2 \quad 1] \quad \times \begin{bmatrix} -4 & w_{12} \\ -1 & w_{22} \\ 12 & w_{32} \end{bmatrix}$$

**he weight of the inputs are multiplied by -1/5, 1, and the bias is multiplied by three to obtain the linear combination of that same point in our second model.**

$$[2 \quad 2 \quad 1] \quad \times \begin{bmatrix} -4 & -1/5 \\ -1 & -1 \\ 12 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 2(-4) + 2(-1) + 1(12) \\ 2(-4) + 2(-1) + 1(12) \end{bmatrix}$$

$$[2 \quad 0.6]$$

**Now, to obtain the probability of the point is in the positive region relative to both models we apply sigmoid to both points as**

$$\left[\frac{1}{1+\frac{1}{e^x}} \quad \frac{1}{1+\frac{1}{e^x}}\right] = \left[\frac{1}{1+\frac{1}{e^2}} \quad \frac{1}{1+\frac{1}{e^{0.6}}}\right] = [0.88 \quad 0.64]$$

**The second layer contains the weights which dictated the combination of the linear models in the first layer to obtain the non-linear model in the second layer. The weights are 1.5, 1, and a bias value of 0.5.**

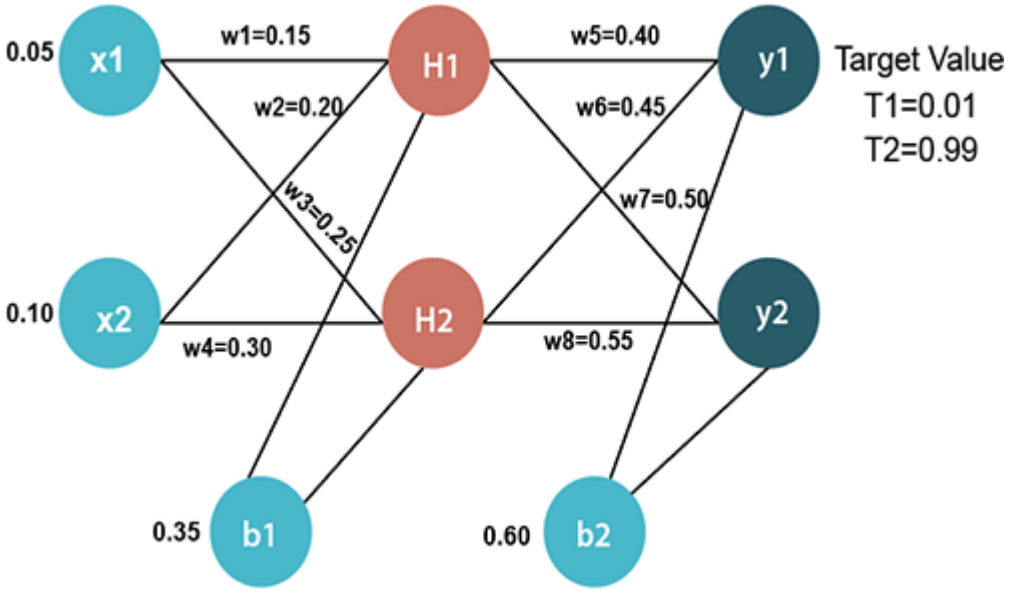
# Backpropagation in neural network

**Backpropagation is one of the important concepts of a neural network. Our task is to classify our data best. For this, we have to update the weights of parameter and bias, but how can we do that in a deep neural network? In the linear regression model, we use gradient descent to optimize the parameter. Similarly here we also use gradient descent algorithm using Backpropagation.**

**For a single training example, Backpropagation algorithm calculates the gradient of the error function. Backpropagation can be written as a function of the neural network. Backpropagation algorithms are a set of methods used to efficiently train artificial neural networks following a gradient descent approach which exploits the chain rule.**

**The main features of Backpropagation are the iterative, recursive and efficient method through which it calculates the updated weight to improve the network until it is not able to perform the task for which it is being trained. Derivatives of**

**the activation function to be known at network design time is required to Backpropagation.**




**Input values**
**X1=0.05**
**X2=0.10**
**Initial weight**
**W1=0.15 w5=0.40**
**W2=0.20 w6=0.45**
**W3=0.25 w7=0.50**
**W4=0.30 w8=0.55**
**Bias Values**
**b1=0.35 b2=0.60**
**Target Values**
**T1=0.01**
**T2=0.99**
**Now, we first calculate the values of H1 and H2 by a forward pass.**
**Forward Pass**
**To find the value of H1 we first multiply the input value from the weights as**
**$H1=x1\times w_1+x2\times w_2+b1$**

$$H1=0.05\times0.15+0.10\times0.20+0.35$$

$$H1=0.3775$$

**To calculate the final result of H1, we performed the sigmoid function as**

$$H1_{final}=\frac{1}{1+\frac{1}{e^{H1}}}$$

$$H1_{final}=\frac{1}{1+\frac{1}{e^{0.3775}}}$$

$$H1_{final}=0.593269992$$

**We will calculate the value of H2 in the same way as H1**

$$H2=x1\times w_3+x2\times w_4+b1$$

$$H2=0.05\times 0.25+0.10\times 0.30+0.35$$

$$H2=0.3925$$

**To calculate the final result of H1, we performed the sigmoid function as**

$$H2_{final}=\frac{1}{1+\frac{1}{e^{H2}}}$$

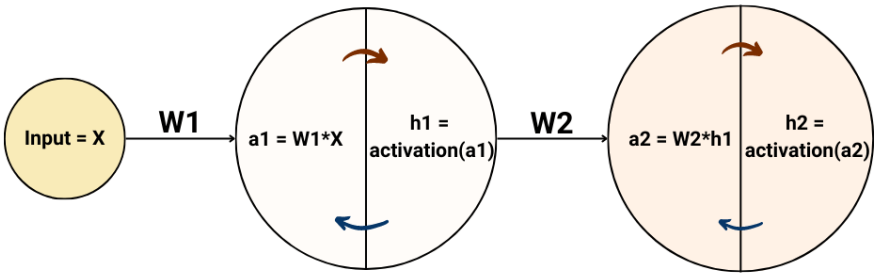
$$H2_{final}=\frac{1}{1+\frac{1}{e^{0.3925}}}$$

$$H2_{final}=0.596884378$$

**Now, we calculate the values of y1 and y2 in the same way as we calculate the H1 and H2.**

# Chain rule calculus

$$\frac{\partial y}{\partial x}=\frac{\partial y}{\partial u}*\frac{\partial u}{\partial x}$$




enjoyalgorithms.com

**The Chain Rule: A Fundamental Concept**

**The chain rule of calculus is a foundational concept that allows us to find the derivative of a composite function. In the context of deep learning, a neural network can be seen as a composition of many functions, with each layer representing a transformation of the input data. The chain rule enables us to calculate the gradient of the network's output with respect to its parameters, which is essential for updating the weights and biases during the training process.**

**The chain rule states that if we have a composite function $f(g(x))$, then the derivative of this function with respect to $x$ is given by:**

$$\frac{d}{dx} f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx}$$

**In the context of deep learning, *f* represents the final output of the neural network, *g* represents the intermediate activations at each layer, and *x* represents the input data.**

**Backpropagation: The Learning Algorithm**

**Backpropagation is the cornerstone of training neural networks. It involves the iterative process of computing gradients and updating the network's parameters to minimize a chosen loss function. At the core of backpropagation is the application of the chain rule to compute these gradients efficiently.**

**Here's a step-by-step overview of the backpropagation process:**

**1.Forward Pass:**

- **The input data is passed through the network layer by layer.**
- **Each layer performs a weighted sum and applies an activation function to produce the output.**
- **These outputs are stored for later use in the backward pass.**

**2.Backward Pass (Backpropagation):**

- **Starting from the output layer, the gradient of the loss function with respect to the output is computed.**
- **The chain rule is applied iteratively to compute the gradient of the loss function with respect to each layer's parameters and inputs.**
- **Gradients are propagated backward through the network.**

**3.Parameter Update:**

- **The computed gradients are used to update the network's parameters (weights and biases) through optimization algorithms like gradient descent.**
- **This process is repeated for a predefined number of iterations or until convergence.**

**The Chain Rule in Action**

**Let's break down how the chain rule is applied during the backpropagation process with a simple example. Consider a single-layer neural network with a linear transformation followed by a sigmoid activation function.**

1. **Forward Pass:**

- **The input data *x* is transformed through a linear transformation: $z=Wx+b$, where *W* is the weight matrix and *b* is the bias.**
- **The output *y* is obtained by applying the sigmoid activation function: $y=\sigma(z)$.**

**2. Backward Pass (Backpropagation):**

**a. Compute the gradient of the loss (*L*) with respect to the output (*y*): *dy/ dL***

**b. Apply the chain rule to find the gradient of the loss with respect to the input *z*:**

**Compute the gradient of the loss with respect to the parameters (*W* and *b*):**

- $\frac{dL}{dW} = \frac{dL}{dz} \cdot \frac{dz}{dW} = x \cdot \frac{dL}{dz}$
- $\frac{dL}{db} = \frac{dL}{dz} \cdot \frac{dz}{db} = \frac{dL}{dz}$

**3. Parameter Update:**

- **Update the parameters *W* and *b* using the computed gradients and an optimization algorithm.**

**This example illustrates how the chain rule is used to efficiently compute the gradients necessary for parameter updates. The process is analogous for deep neural networks, with the chain rule applied layer by layer during the backpropagation process.**

**The Significance of Deep Learning**

**Deep learning models, characterized by their depth (many layers), have become increasingly popular because of their capacity to learn hierarchical representations of data. Each layer in a deep network captures different levels of abstraction, allowing the network to automatically extract features from raw data.**

**The chain rule is crucial in this context because it enables efficient gradient propagation through the network, ensuring that each layer learns to adjust its parameters to minimize the overall loss. Without the chain rule, the training of deep neural networks would be computationally infeasible, as the number of possible parameters and their interactions grows exponentially with network depth.**

**Finally!**

**The chain rule of calculus, a fundamental concept, is the backbone of deep learning and the key to backpropagation, enabling neural networks to learn complex patterns from data efficiently. With the chain rule, gradients can be computed and propagated through deep networks, allowing for automatic differentiation and parameter updates during training.**

**As deep learning continues to advance and tackle more complex problems, the role of the chain rule in enabling the training of deep neural networks becomes increasingly significant. This powerful technique has revolutionized the field of artificial intelligence, enabling machines to perform tasks that were once considered impossible, and it continues to drive innovation and breakthroughs in the world of technology and data science.**

$$\frac{dL}{dz} = \frac{dL}{dy} \cdot \frac{dy}{dz}$$

- The derivative of the sigmoid function is $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$.
- So, $\frac{dy}{dz} = \sigma(z)(1 - \sigma(z))$.