ARTIFICIAL NEURAL NETWORKS

OVERVIEW

- Introduction to Neural Networks
- Evolution of Neural Network
- Fundamental Concepts.
- Basic Model of ANN
- Important terminologies of ANN
- McCulloch-Pitts Neuron
- Hebb Network
- Perceptron Networks
- Back-propagation Network

FUNDAMENTAL CONCEPTS- TRADITIONAL COMPUTER VS ANN

- Neural networks are those information processing systems, which are constructed and implemented to model the human brain.
- **The main objective** of the neural network research is to develop a computational device for modeling the brain to **perform various computational tasks at a faster rate than the traditional systems .**

- Artificial neural networks performs **various tasks** such as **pattern matching and classification, Optimization function, approximation, vector quantization, data clustering.**

- These tasks are very **difficult for traditional computers**, which are faster in algorithmic computational tasks and precise arithmetic operations.

FUNDAMENTAL CONCEPTS-TRADITIONAL COMPUTER VS ANN

Computers vs. Neural Networks

"Standard" Computers	Neural Networks
• one CPU	highly parallel processing
•fast processing units	slow processing units
•reliable units	unreliable units
•static infrastructure	dynamic infrastructure

FUNDAMENTAL CONCEPTS

- ARTIFICIAL NEURAL NETWORK-DEFINITION

- An artificial neural network (ANN) may be defined as an information processing model that is inspired by the way biological nervous systems, such as the brain, process information. This model tries to replicate only the most basic functions of the brain.
- The key element of ANN is the novel structure of its information processing system.
- An ANN is composed of a large number of highly interconnected processing elements(neurons) working in union to solve specific problems.

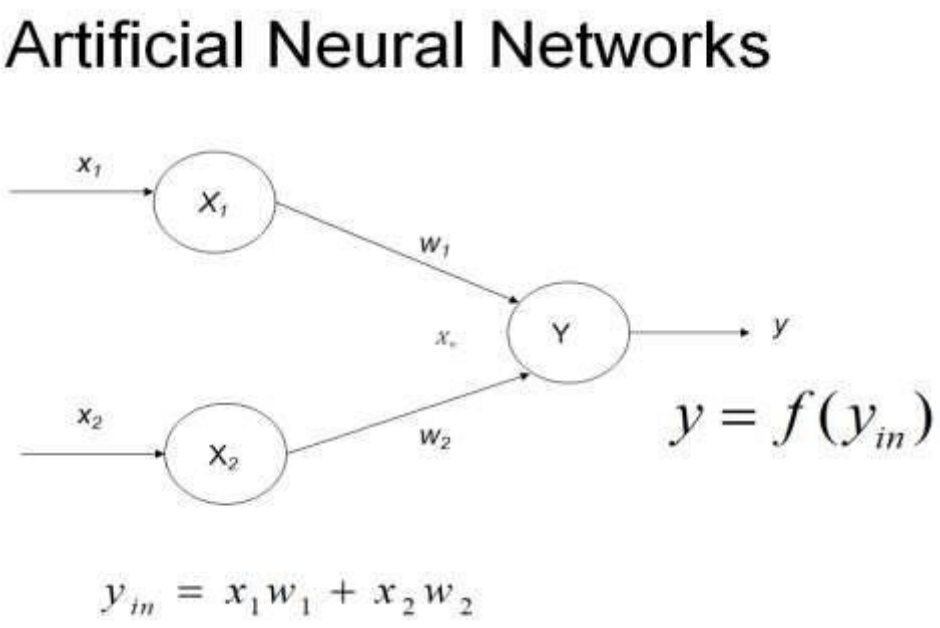
FUNDAMENTAL CONCEPTS

-ARCHITECTURE OF A SIMPLE ARTIFICIAL NEURAL NETWORK

- ANN possess a large number of processing elements called nodes/neurons which operate in parallel.
- Neurons are connected with others by connection link.
- Each link is associated with weights which contain information about the input signal.
- Each neuron has an internal state (activation or activity level) of its own which is a function of the inputs that neuron receives.
- The activation signal of a neuron is transmitted to other neurons. Remember, a neuron can send only one signal at a time, which can be transmitted to several other neurons.

FUNDAMENTAL CONCEPTS

-ARCHITECTURE OF A SIMPLE ARTIFICIAL NEURAL NETWORK



- To depict the basic operation of a neural net, consider a set of neurons, say $\mathbf{X_1}$ and $\mathbf{X_2}$, transmitting signals to another neuron, Y.
- Here $\mathbf{X_1}$ and $\mathbf{X_2}$ are input neurons, which transmit signals, and
  Y is the output neuron, which receives signals.
- Input neurons $\mathbf{X_1}$ and $\mathbf{X_2}$ *are connected to the output neuron Y, over a weighted* interconnection links ($\mathbf{w_1}$, **and** $\mathbf{w_2}$) as shown in Figure.
- For the above simple neuron net architecture, the net input has to be calculated in the following way:

$$y_{in} = x_1 w_1 + x_2 w_2$$

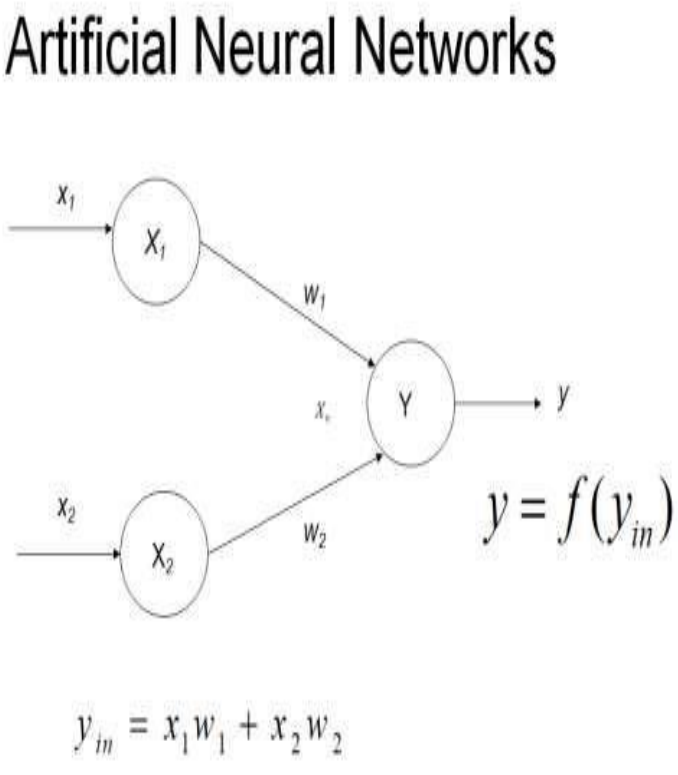
- where $\mathbf{x_1}$ and $\mathbf{x_2}$ are the activations of the input neurons $\mathbf{X_1}$, and $\mathbf{X_2}$, i.e., the output of input signals. The output ***y*** *of the output neuron* ***Y*** *can be obtained by applying activations over the net input, i.e., the function* of the net input:

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

Output= Function (net input calculated)

FUNDAMENTAL CONCEPTS

-ARCHITECTURE OF A SIMPLE ARTIFICIAL NEURAL NETWORK



- The function to be applied over the net input is called **activation function**.

$$y_{in} = x_1 w_1 + x_2 w_2$$

- *The* calculation of the net input is similar to the calculation of output of a pure linear straight line equation (y = *mx).*

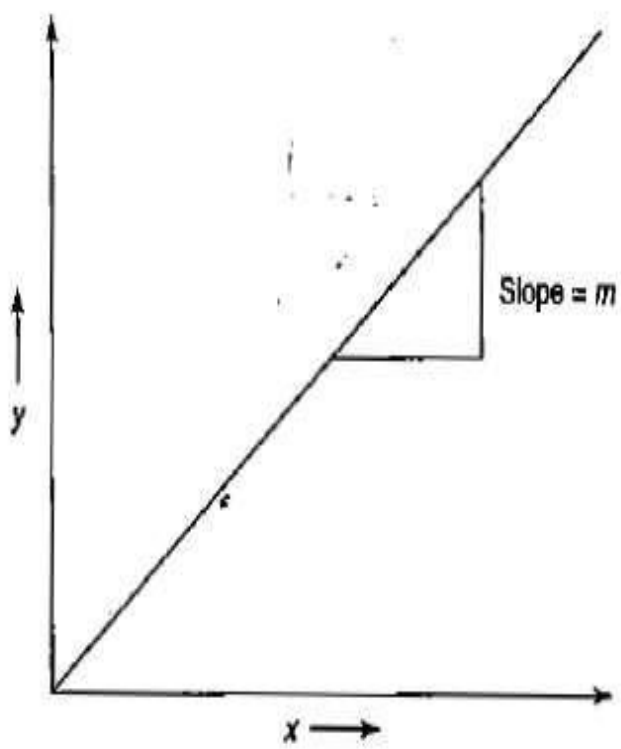


Figure Graph for $y = mx$.

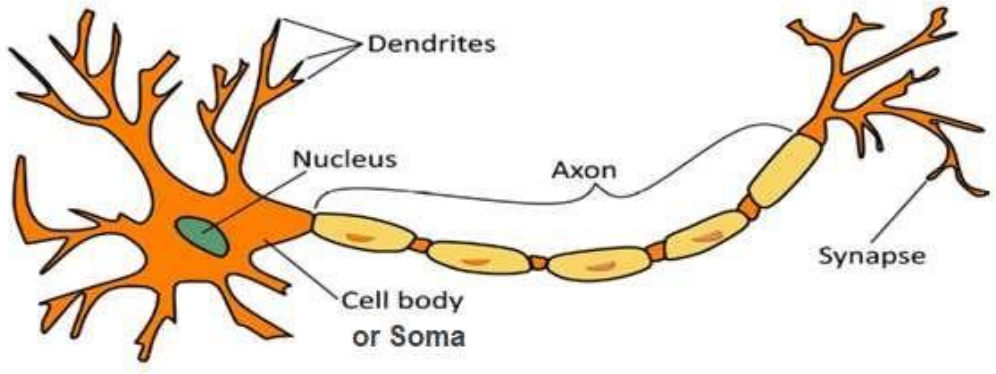
- Here, to obtain the output *y, the slope m is directly multiplied with the input signal.*

This is a linear equation.

- Thus, when slope and input are linearly varied, the output *is also linearly varied, as shown in* graph.
- This shows that the weight involved in the ANN is equivalent to the slope of the linear straight line.

BIOLOGICAL NEURON NETWORK

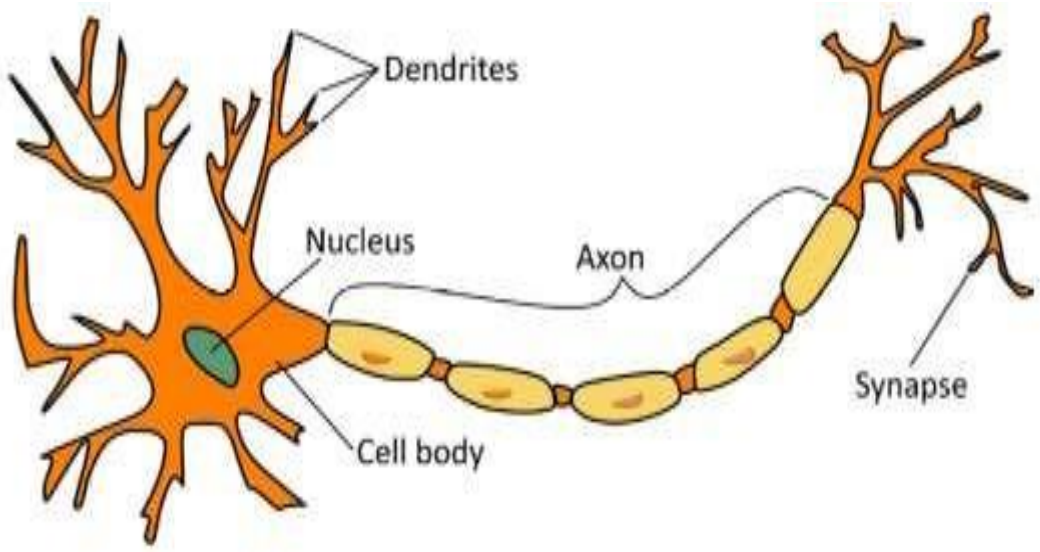
- A nerve cell (neuron) is a special biological cell that processes information.
- According to an estimation, there are huge number of neurons, approximately 10^{11} numerous interconnections, approximately 10^{15}.



The biological neuron depicted in Figure above consists of 4 main parts:

1. **Dendrites –** They are tree-like branches, responsible for receiving the information from other neurons it is connected to.
2. **Soma –** It is the cell body of the neuron and is responsible for processing of information.
3. **Axon –** It is just like a cable through which neurons send the info.
4. **Synapses–** It is the connection between the axon (bulb-like organ) and other neuron dendrites.

A BIOLOGICAL NEURON AND THE ARTIFICIAL NEURON



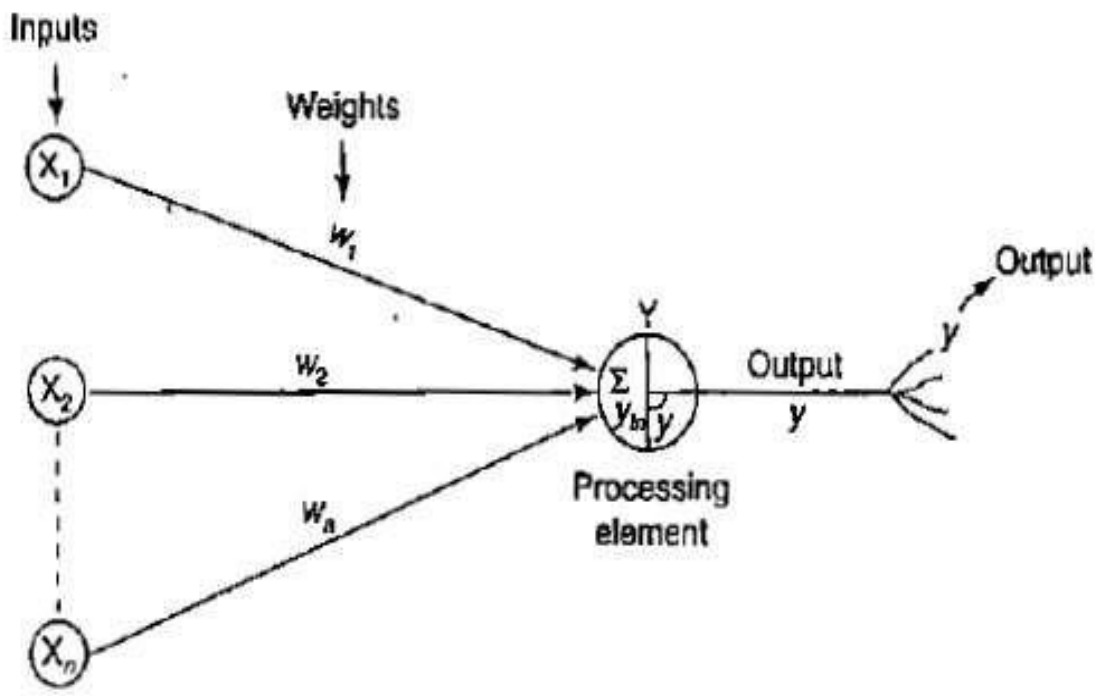
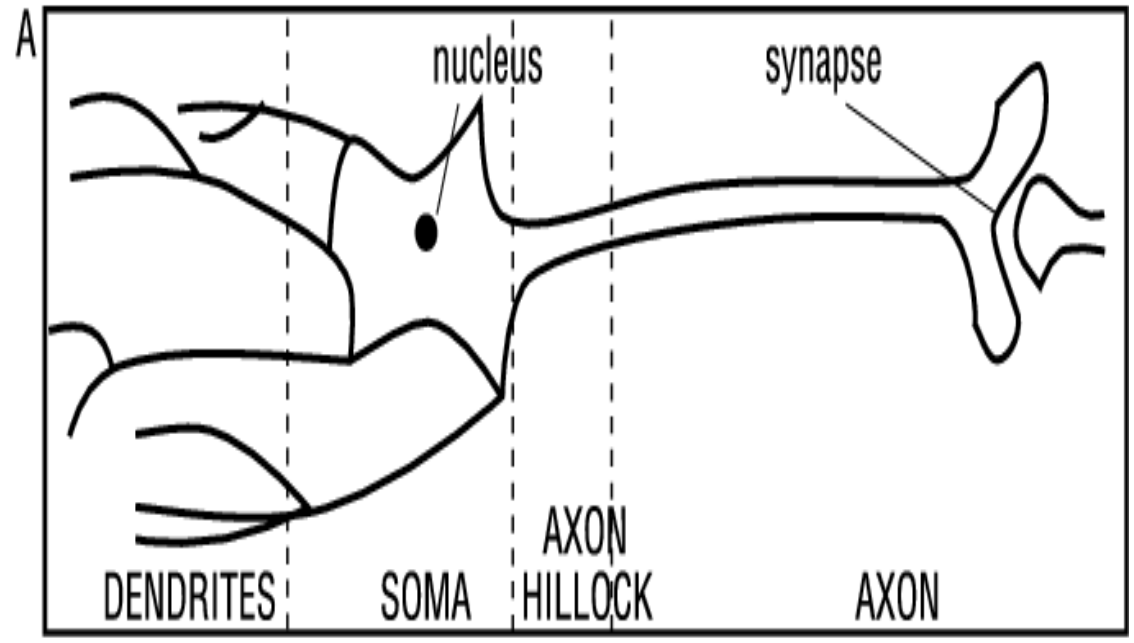
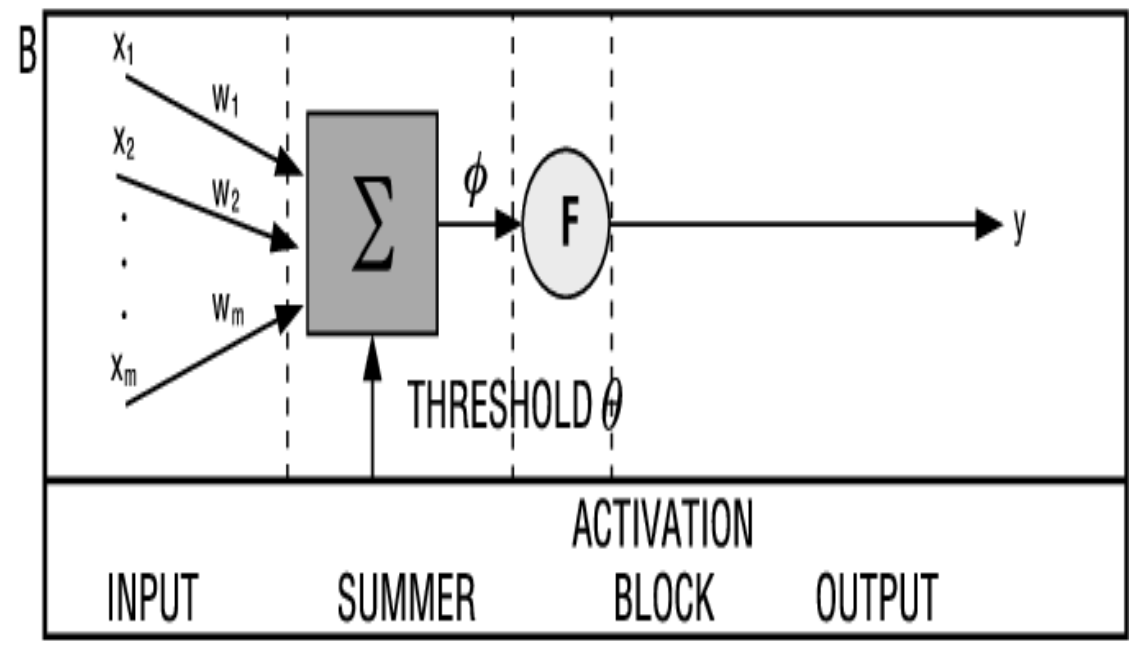


Figure Mathematical model of artificial neuron.





Main Parts of an Artificial Neuron

- Input
- Weights or interconnections
- Activation Function/Processing element
- Output

Terminology relationships between biological and artificial neurons

Biological Neuron	Artificial Neuron
Cell	Neuron/Node
Dendrites	Input
Synapse	Weights
Soma	Net input(Processing element)
Axon	Output

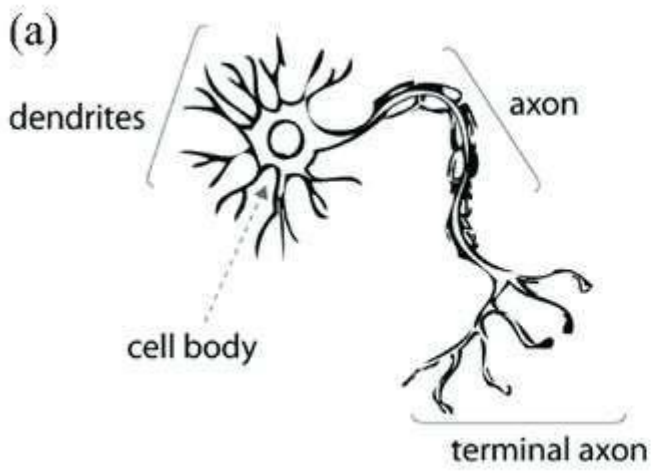
A BIOLOGICAL NEURON AND THE ARTIFICIAL NEURON

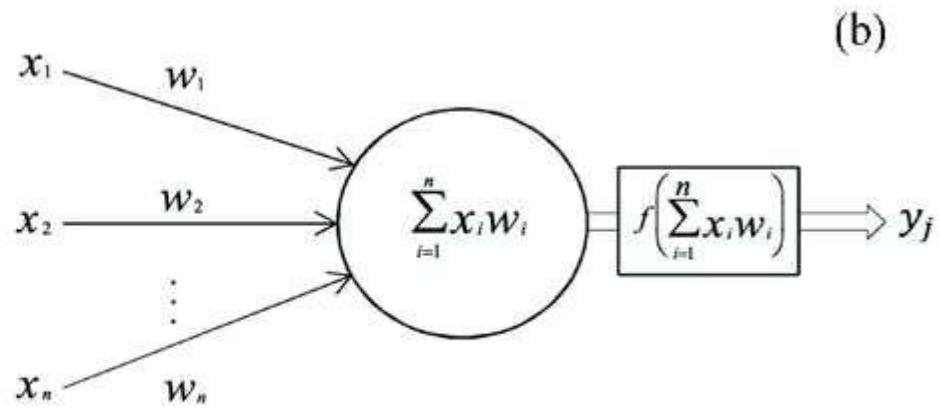
$$y_{in} = x_1.w_1 + x_2.w_2 + x_3.w_3 \ldots x_m.w_m$$

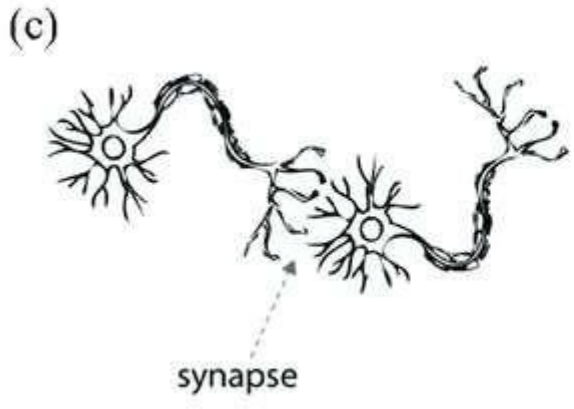
i.e., Net input $y_{in} = \sum_i^m x_i.w_i$

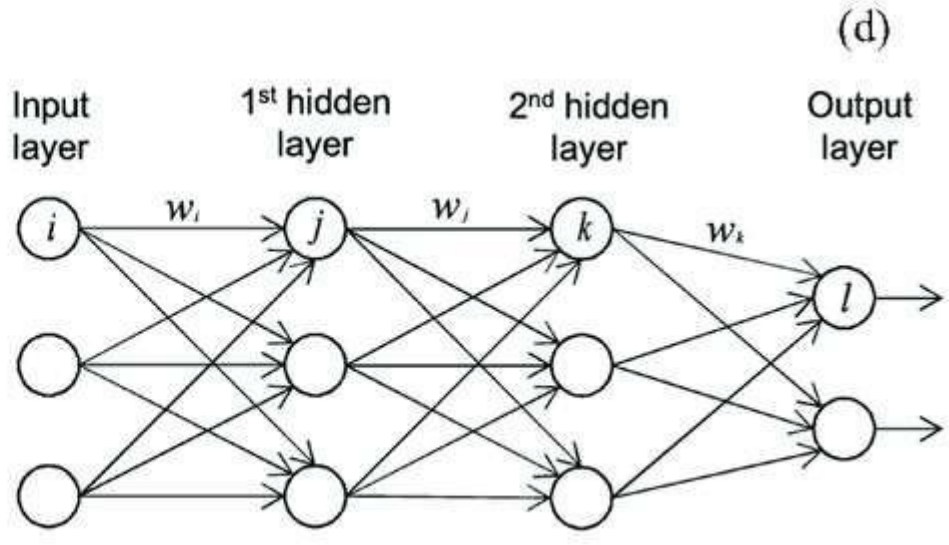
- *where i represents the ith processing element.*
- *The activation function is applied over it to calculate the output.*
- *The weight represents the strength of synapse connecting the input and the output neurons.*
- *A positive* weight corresponds to an **excitatory synapse**, and a negative weight corresponds to an **inhibitory synapse.**

A BIOLOGICAL NEURON AND THE ARTIFICIAL NEURON









BRAINS VS. COMPUTERS

- COMPARISON BETWEEN BIOLOGICAL NEURON AND THE ARTIFICIAL NEURON

Characteristics	Artificial Neural Network	Biological(Real) Neural Network
Speed	Faster in processing information. Response time is in nanoseconds.	Slower in processing information. The response time is in milliseconds.
Processing	Serial processing.	Massively parallel processing.
Size & Complexity	Less size & complexity. It does not perform complex pattern recognition tasks.	Highly complex and dense network of interconnected neurons containing neurons of the order of 10^{11} with 10^{15} of interconnections.
Storage	i) Stored in continuous memory location. ii) Overloading may destroy older locations. iii) Can be easily retrieved	i) Information is stored in interconnections or in synapse strength. ii) New information is stored without destroying old one. iii) Sometimes fails to recollect information
Fault tolerance	Fault intolerant. Information once corrupted cannot be retrieved in case of failure of the system.	Information storage is adaptable means new information is added by adjusting the interconnection strengths without destroying old information
Control Mechanism	There is a control unit for controlling computing activities	No specific control mechanism external to the computing task.

CHARACTERISTICS OF ANN

1. It is a neurally implemented mathematical model.
2. There exist a large number of highly interconnected processing elements called *neurons in an ANN.*
3. The interconnections with their weighted linkages hold the informative knowledge.
4. The input signals arrive at the processing elements through connections and connecting weights.
5. The processing elements of the ANN have the ability to learn, recall and generalize from the given data by suitable assignment or adjustment of weights.
6. The computational power can be demonstrated only by the collective behavior of neurons, and it should be noted that no single neuron carries specific information.

1. For the network shown in Figure 1, calculate the net input to the output neuron.

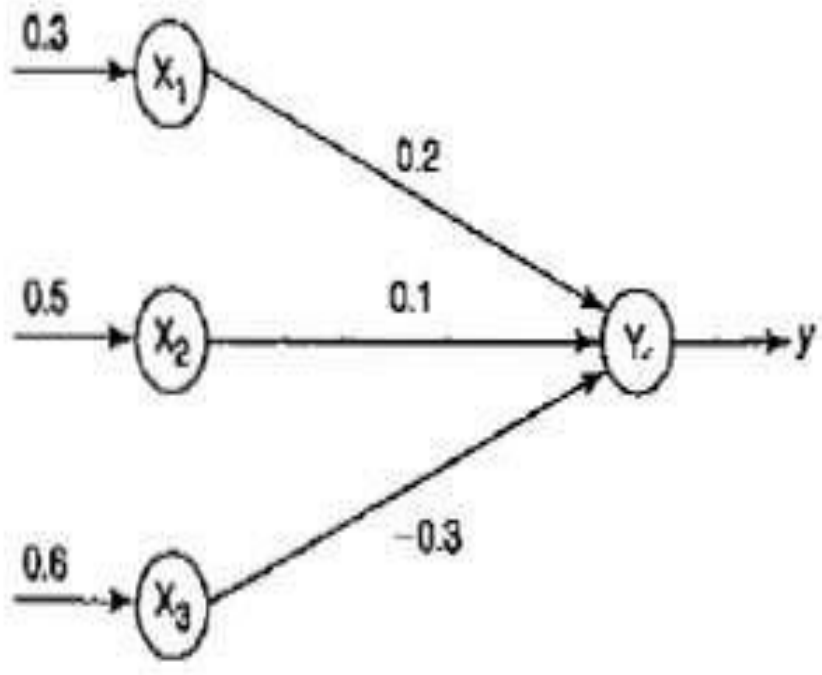


Figure 1 Neural net.

Solution: $y_{in} = x_1.w_1 + x_2.w_2 + x_3.w_3 \ldots x_m.w_m$

i.e., Net input $y_{in} = \sum_i^m x_i.w_i$

- The given neural net consists of **three input neurons** and **one output neuron.**
- The inputs and weights are

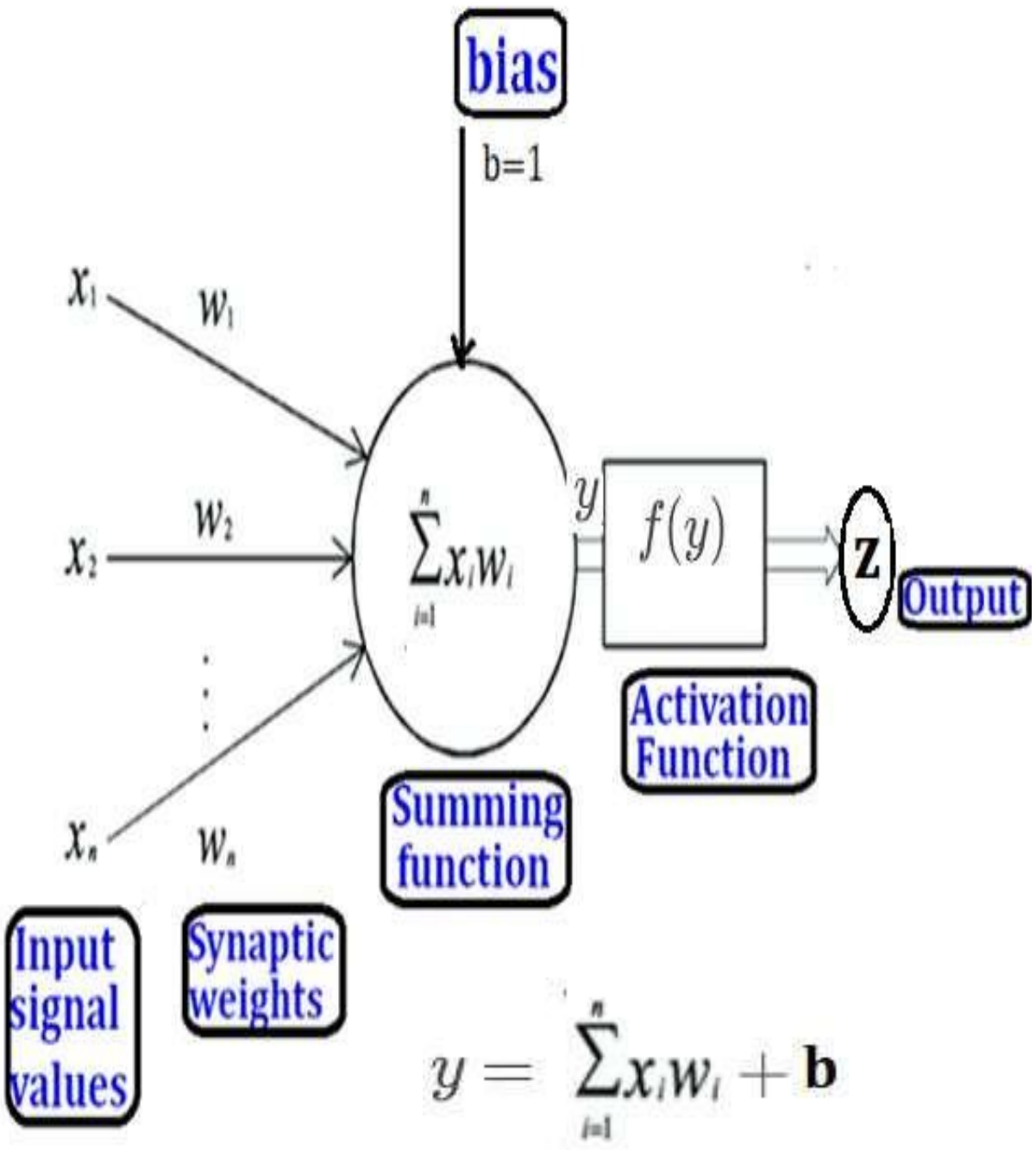
$[x_1,x_2,x_3] = [0.3, 0.5, 0.6]$

$[w_1,w_2,w_3] = [0.2,0.1,-0.3]$

- The net input can be calculated as

$$y_{in} = x_1w_1 + x_2w_2 + x_3w_3$$
$$= 0.3 \times 0.2 + 0.5 \times 0.1 + 0.6 \times (-0.3)$$
$$= 0.06 + 0.05 - 0.18 = -0.07$$

ARTIFICIAL NEURON AND ITS MODEL



$$y = \sum_{i=1}^{n} x_i w_i + \mathbf{b}$$

Artificial Neuron:

- An **Artificial Neuron** is a mathematical function based on the model of biological neurons

Each artificial neuron has the following main functions:

1. Receives inputs from a no. of input signals.
2. Weight each input separately and sums them up.
3. Pass this sum through an activation function to produce output.

THE ARTIFICIAL NEURAL NETWORK

- **Artificial Neurons** (processing node) composed of:
  - (many) **input** neuron(s) connection(s) (dendrites)
  - a **computation unit** (nucleus) composed of:
    - a **Summation function** (ax+b)
    - an **Activation function** (equivalent to the **synapse**)
  - an **output** (axon)

2. Calculate the net input for the network shown in Figure 2 with bias included in the network.

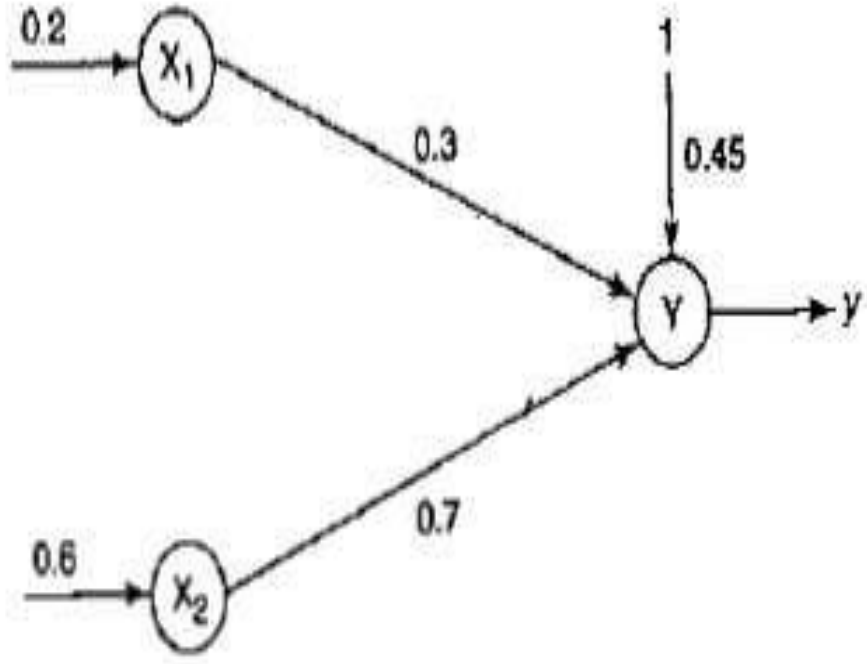


Figure 2 Simple neural net.

Solution: $[x_1, x_2, b] = [0.2, 0.6, 0.45]$

$[w_1, w_2] = [0.3, 0.7]$

The net input can be calculated as,

$$y_{in} = b + x_1 w_1 + x_2 w_2$$

$$y_{in} = 0.45 + 0.2 \times 0.3 + 0.6 \times 0.7$$

$$y_{in} = 0.45 + 0.06 + 0.42 = 0.93$$

OVERVIEW

- **Basic Model of ANN**
  - -Connections
  - -Learning
    - Supervised Learning
    - Unsupervised Learning
    - Reinforcement Learning
  - -Activation Functions

BASIC MODEL OF ANN

- The models of ANN are specified by the three basic entities namely:
  1. The model's synaptic interconnection.***(Network Topology)***
  2. The training rules or learning rules adopted for updating and adjusting the connection weights. (***Adjustments of Weights or Learning)***
  3. Their ***Activation Functions***.

Basic Model of ANN- 1. Connections

1. Connections:-

- An ANN consists of a set of highly interconnected processing elements such that each processing element's output is found to be connected through weights to the other processing elements or to itself;
- The point where the connection originates and terminate should be noted, and the function of each processing element in an ANN should be specified.
- The arrangement of neurons to form layers and connection pattern formed within and between layers is called the **network architecture**.
- There are five basic types of neuron connection architectures:-

a) Single layer feed forward network.
b) Multilayer feed forward network

} **Feed Forward Network**

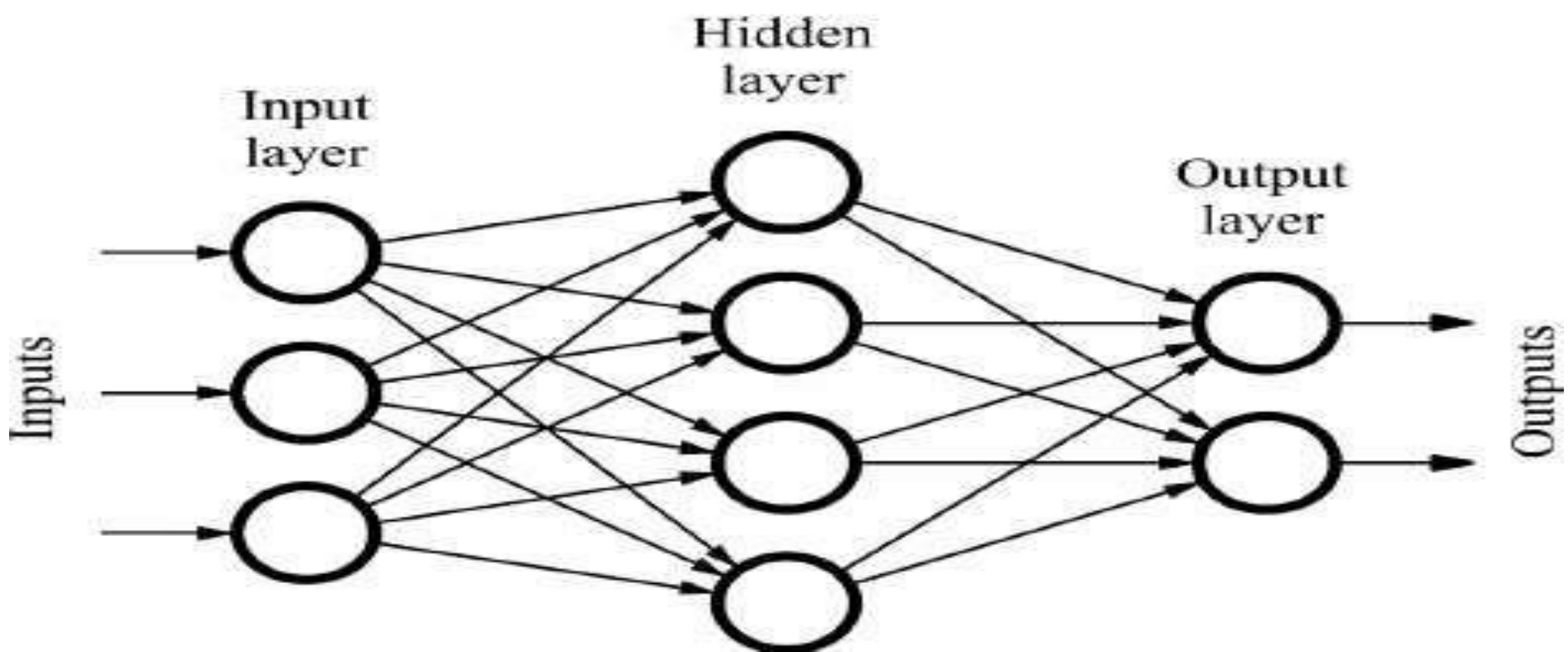
c) Single node with its own feedback
d) Single layer recurrent network
e) Multilayer recurrent network

} Recurrent Network

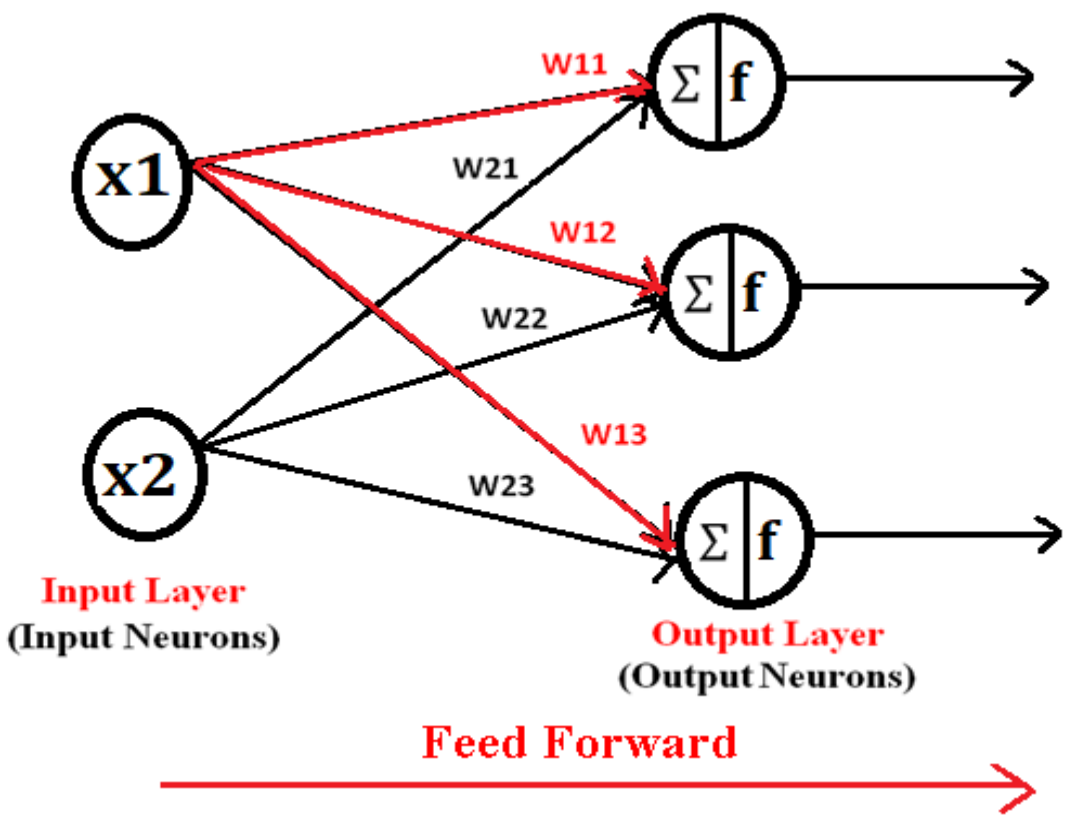
Basic Model of ANN- 1. Connections

Feed Forward Network

- Information is only processed in one direction. While the data may pass through multiple hidden nodes, it always moves in one direction and never backwards.
- It enters into the ANN through the input layer and exits through the output layer while hidden layers may or may not exist.
- A Feed Forward Neural Network is an artificial neural network in which the connections between nodes does not form a cycle.
- The opposite of a feed forward neural network is a recurrent neural network, in which certain pathways are cycled.



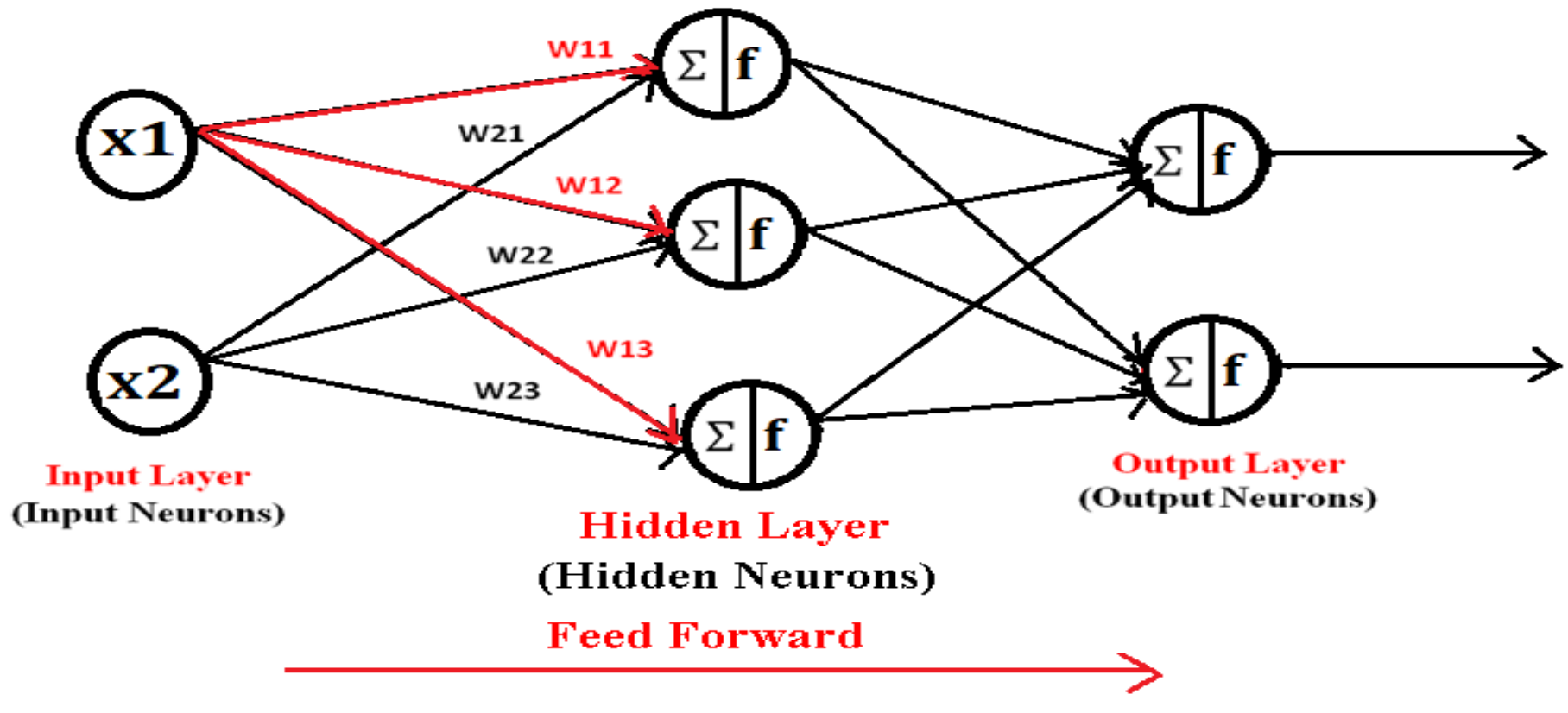
Basic Model of ANN- 1. Connections



1. **Single layer feed forward network**

- A layer is formed by taking a processing element and combining it with other processing elements.
- When a layer of the processing nodes is formed, the inputs can be connected to these nodes with various weights, resulting in a series of outputs, one per node.

Basic Model of ANN- 1. Connections



2.Multilayer feed forward network

1. A multilayer feed forward network is formed by the interconnection of several layers.
2. The input layer is that which receives the input and this layer has no function except buffering the input signal.
3. The output layer generates the output of the network.
4. Any layer that is formed between the input layer and the output layer is called the hidden layer.

Basic Model of ANN- 1. Connections

Recurrent Neural Network

The Recurrent Neural Network saves the output of a layer and feeds this output back to the input to better predict the outcome of the layer.

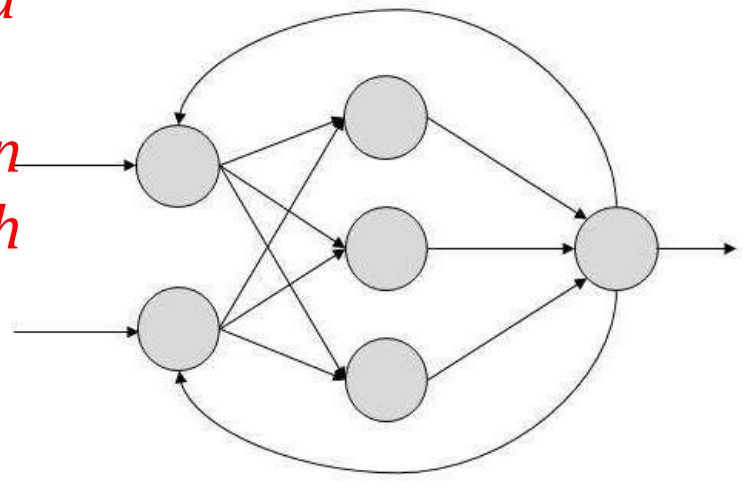
- The first layer in the RNN is quite similar to the feed-forward neural network and the recurrent neural network starts once the output of the first layer is computed.

- After this layer, each unit will remember some information from the previous step so that it can act as a memory cell in performing computations.

- Recurrent networks are the feedback networks with a closed loop.

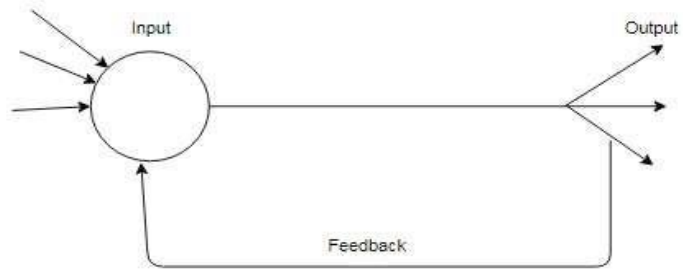
Application:

- *Audio files – This is considered a natural sequence. Audio files clips can be broken down in the audio spectrogram and fed that into RNN's.*
- *Text file – Text is another form of a sequence, text data can be broken into characters or words (remember search engines guessing your next word or character)*

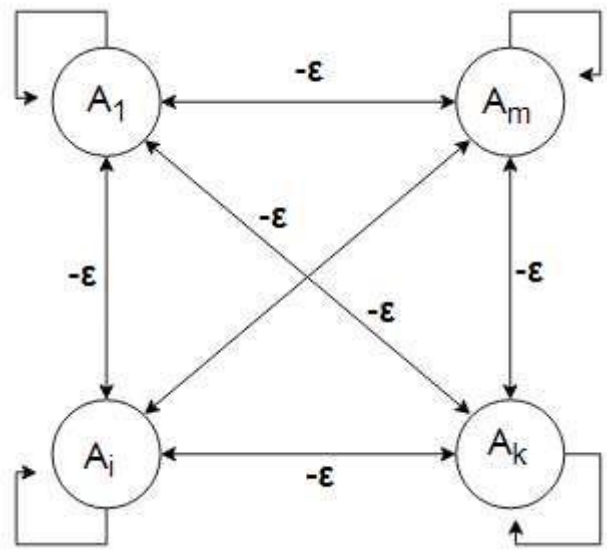


Basic Model of ANN- 1. Connections

3. Single node with its own feedback



- The networks where the output layer output is sent back as an input to the input layer or the other hidden layers is called Feedback Networks.
- In single-node feedback systems, there is a single input layer where the output is redirected back as feedback.



Competitive Net

- The competitive interconnections have fixed weight-**ε**.
- This net is called **MAXNET.**
- **MAXNET**
  - A recurrent network involving both excitatory and inhibitory connections
  - Positive self-feedbacks and negative cross-feedbacks
  - After a number of recurrences, the only non- zero node will be the one with the largest initializing entry from i/p vector .
  - **Thus only one neuron will fire.(*Winner-take-all-policy*)**

Basic Model of ANN- 2. Learning

- The main property of an ANN is its capability to learn.
- An ANN's learning rule is a method, mathematical logic or algorithm which improves the network's performance and/or training time.
- This rule is applied repeatedly over the network(**iterative process**)
- Learning or training is a process by means of which a neural network adapts itself to a stimulus by making proper parameter adjustments resulting in the production of desired response.
- Broadly, there are two kinds of learning in ANNs:

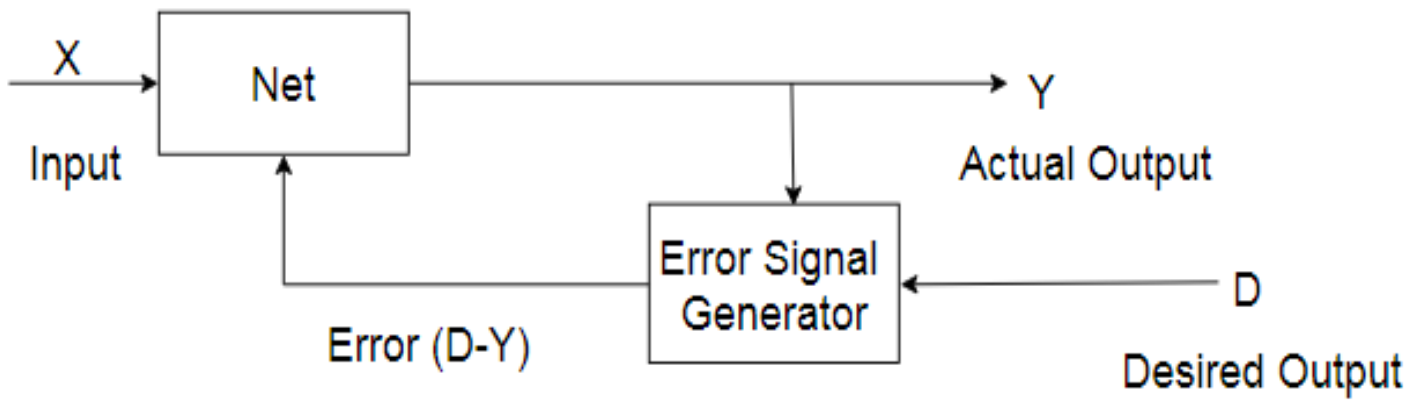
  1.Parameter learning: *It updates the connecting weights in a neural net.*

  2.Structure learning: *It focuses on the change in network structure (which includes the number of processing* elements as well as their connection types).

- Supervised learning;
- Unsupervised learning;
- Reinforcement learning.

Basic Model of ANN- 2. Types of learning

(a) Supervised Learning:-



Working

- During training, the input vector is presented to the network, which results in an output vector. This output vector is the actual output vector.

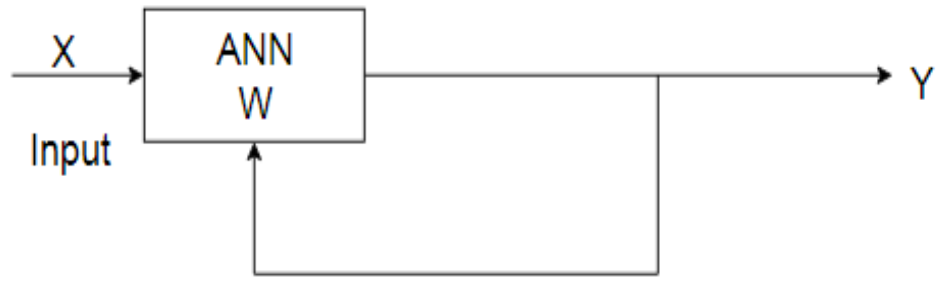
Then the actual output vector is compared with the desired (target)output vector. If there exists a difference between the two output vectors then an error signal is generated by the network. This error signal is used fro adjustment of weights until the actual output matches the desired(target) output.

- In this type of training, supervisor or teacher is required for error minimization. Hence, the network trained by this method is called supervised training methodology.
- In **Supervised Learning,** it is assumed that the correct "target" output values are known for each input pattern.

Basic Model of ANN-

2. Types of learning

(b) Unsupervised Learning:-



Working

➢From Figure it is clear that there is no feedback from the environment to inform what the outputs should be or whether the outputs are correct.

➢In this case, the network must itself discover patterns, regularities, features or categories from the input data and relations for the input data over the output.

➢ While discovering all these features, the network undergoes change in *Its parameters.*

➢This process is called *self-organizing* in which exact clusters will be formed by discovering similarities and dissimilarities among the objects.

Unsupervised Learning

➢The learning is performed without the help of any teacher.

➢It is required when there is no example data set with known answers.

➢For example, searching for a hidden pattern. In this case, clustering i.e. dividing a set of elements into groups according to some unknown pattern is carried out based on the existing data sets present.

➢In ANNs following the unsupervised learning, the input vector s of similar type are grouped without the use of training data to specify how a member of each group looks or to which group it belongs to.

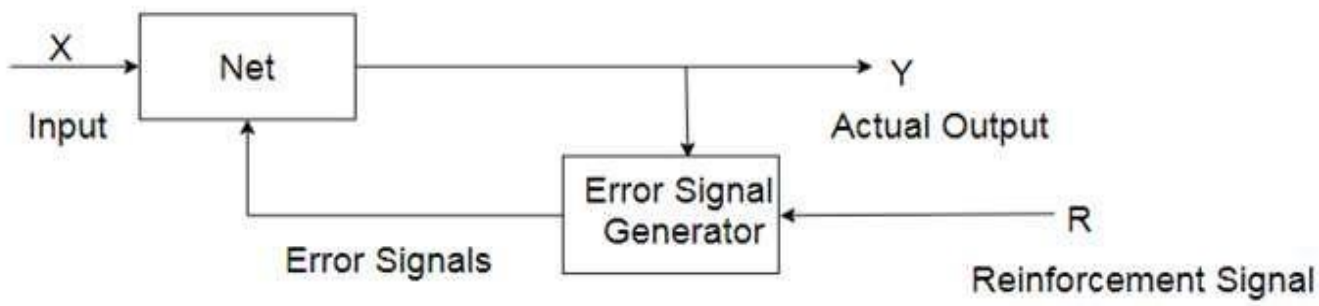
➢In the training process,,

- ➢the network receives the input patterns and organizes these patterns to form clusters.
- ➢When a new input pattern is applied, the neural network gives an output response indicating the class to which the input pattern belongs.
- ➢If for an input, a pattern class cannot be found the a new class is generated.

➢The block diagram of unsupervised learning is as shown in Figure.

Basic Model of ANN- 2. Types of learning

(c) Reinforcement Learning:-



➢Working

➢The reinforcement learning is a form of supervised learning because the network receives some feedback from its environment.

➢However, the feedback obtained here is only evaluative and not instructive.

➢The external reinforcement signals are processed
in the critic signal generator, and the obtained critic signals are sent to the ANN for adjustment of weights properly so as to get better critic feedback in future.

➢The reinforcement learning is also called learning with a critic as opposed to learning with a teacher, which indicates supervised learning.

Reinforcement Learning

➢This strategy built on observation.

➢The ANN makes a decision by observing its environment.

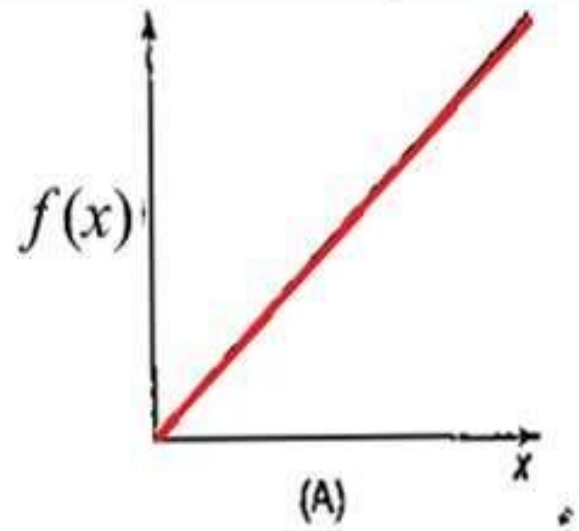
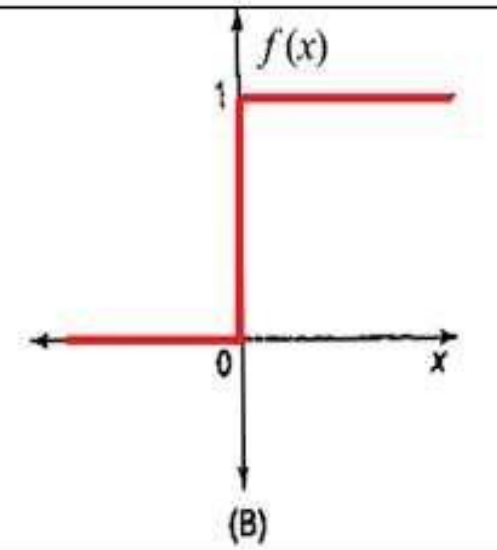
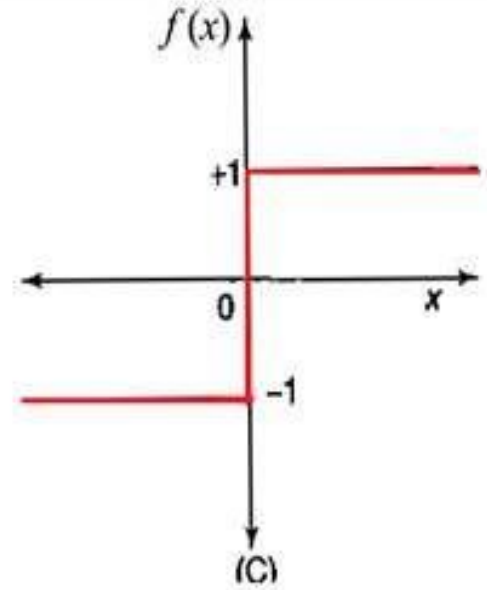
➢If the observation is negative, the network adjusts its weights to be able to make a different required decision the next time.

➢For example, the network might be told that its actual output *is only "50% correct" or so. Thus, here only* critic information is available, not the exact information.

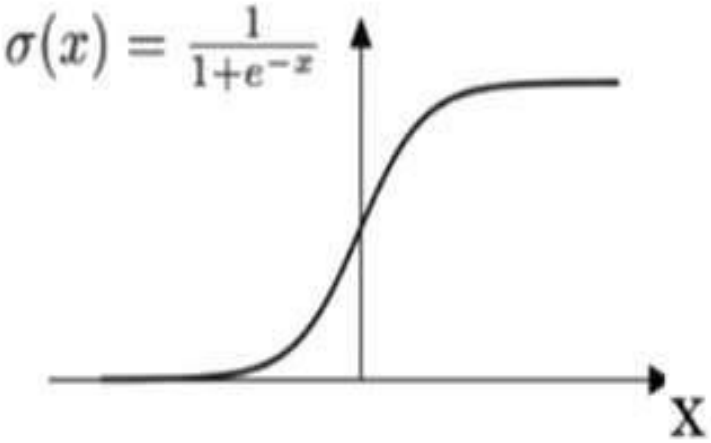
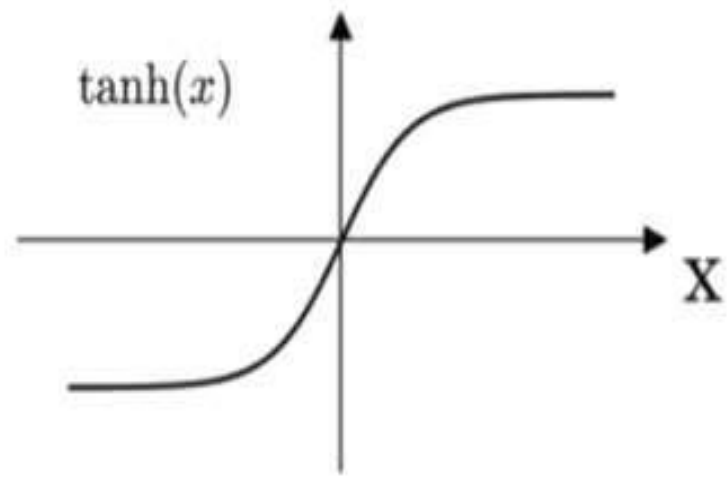
➢The learning based on this critic information is called ***reinforcement learning*** *and the feedback sent is called* ***reinforcement signal.***

➢The block diagram of reinforcement learning is shown in Figure.

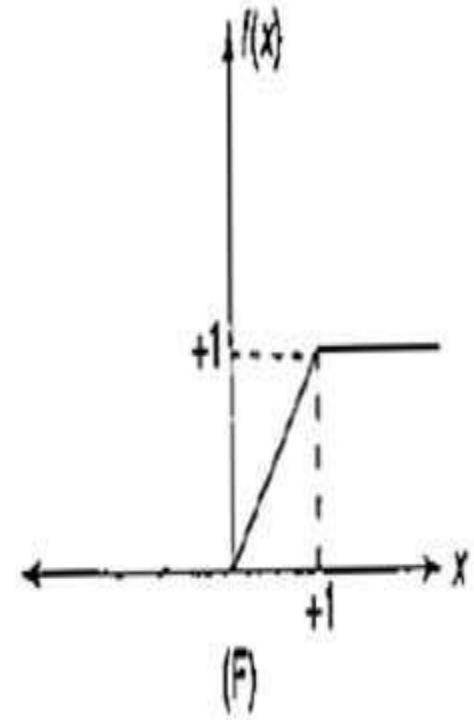
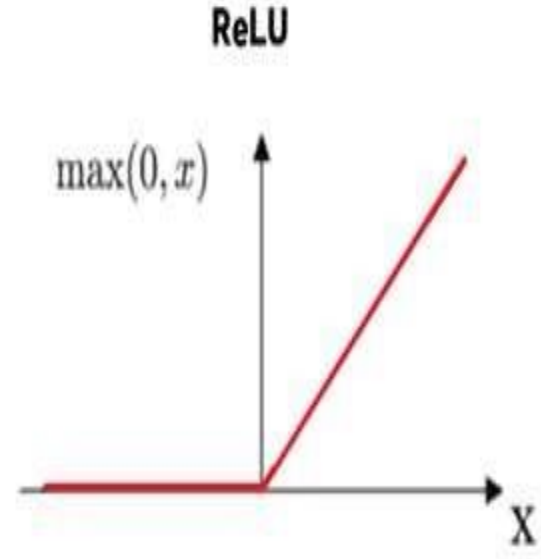
Activation Function

Activation Function	Explanation	Equation	1-D Graph
1. Identity function Or **Linear Function**	▪ The output here remains the same as the input. ▪ The input layer uses the Identity activation function.	$f(x) = x \forall x$	
2. Binary step function	▪ This function is most widely used in single layer nets to convert the net input to an output that is binary (1 or 0).	$f(x) = \begin{cases} 1, & if \quad x \geq \theta \\ 0, & if \quad x < \theta \end{cases}$ Where θ represents the threshold value.	
3. Bipolar step function **Sig / Signum function**	▪ Used in single layer nets to convert the net input to an output that is bipolar (+1 or -1).	$f(x) = \begin{cases} 1, & if\ x > \theta \\ 0, & if\ x = \theta \\ -1, & if\ x < \theta \end{cases}$ Where θ represents the threshold value.	

Activation Function

Activation Function	Explanation	Equation	1-D Graph
4. Sigmoidal functions: ➢ It is a function which is plotted as **'S'** shaped graph. ➢ The sigmoid functions are widely used in back propagation nets because of the relationship between the value of the functions at a point and the value of the derivative at that point which reduces the computational burden during training. ➢ **Sigmoidal functions are of two types: -** **(a) Binary sigmoid function** **(b) Bipolar sigmoid function**			
4. **(a) Binary sigmoid function**	▪ Also called as the **logistics sigmoid function** or **unipolar sigmoid function** or **sigmod function**	$y = f(y_{in})$ $f(x) = \frac{1}{1+e^{-\lambda x}} = \frac{1}{1+e^{-y_{in}}}$ where λ is the steepness parameter. The derivative of this function is, $f'(x) = \lambda f(x)[1 - f(x)]$ Here, the range of the sigmoid function is from 0 to 1.	**Sigmoid** $\sigma(x) = \frac{1}{1+e^{-x}}$ 
4. **(b) Bipolar sigmoid function**	▪ If the network uses the binary data, then it is better to convert it to bipolar form and use the bipolar sigmoidal activation function ▪ Also called **Tanh Function** or **Tangent Hyperbolic function.**	$y = f(y_{in})$ $f(x) = \frac{2}{1+e^{-\lambda x}} - 1 = \frac{1-e^{-\lambda x}}{1+e^{-\lambda x}}$ $f(y_{in}) = \frac{1-e^{-y_{in}}}{1+e^{-y_{in}}}$ where λ is equal to steepness parameter The bipolar sigmoid function is closely related to, hyperbolic tangent function, which is written as, $h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$ The derivative of this function is, $h'(x) = [1 + h(x)][1 - h(x)]$	**Tanh** 

Activation Function

Activation Function	Explanation	Equation	1-D Graph
5. Ramp Function	▪The ramp function is a truncated version of the linear function.	$f(x) = \begin{cases} 1, & if \quad x \geq 1 \\ x, & if \quad 0 \leq x \leq 1 \\ 0, & if \quad x < 0 \end{cases}$	 (F)
6. RELU(*Rectified linear unit*)	▪ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. ▪At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	

3. Obtain the output of the neuron Y for the network shown in Figure 3 using activation functions as: (i) binary sigmoidal and (ii) bipolar sigmoidal.

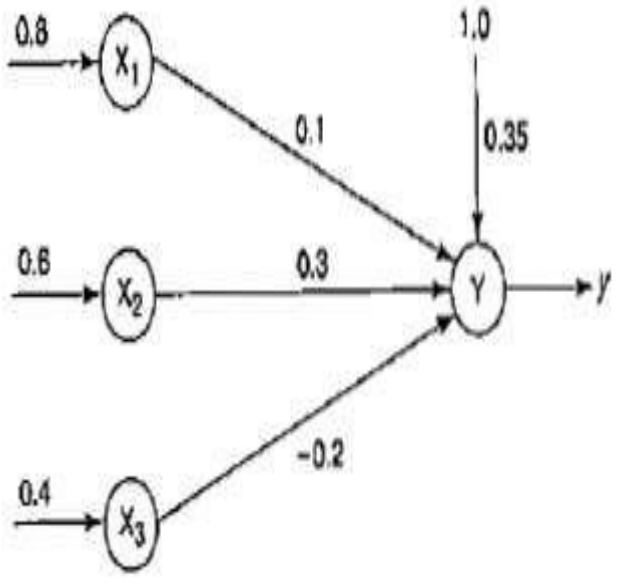


Figure 3 Neural net.

Solution: The given network has three input neurons with bias and one output neuron. These form a single layer network,

The inputs are given as,

$$[x_1, x_2, x_3] = [0.8, 0.6, 0.4]$$

The weights are,

$$[w_1, w_2, w_3] = [0.1, 0.3, -0.2]$$

The net input can be calculated as,

$$y_{in} = b + \sum_{i=1}^{n} (x_i w_i)$$

$$y_{in} = 0.35 + 0.8 \times 0.1 + 0.6 \times 0.3 + 0.4 \times (-0.2)$$

$$y_{in} = 0.35 + 0.08 + 0.18 - 0.08 = 0.53$$

(i) For Binary Sigmoidal Function,

$$y = f(y_{in}) = \frac{1}{1 + e^{-y_{in}}} = \frac{1}{1 + e^{-0.53}} = 0.62$$

(ii) For Bipolar Sigmoidal activation function,

$$y = f(y_{in}) = \frac{2}{1 + e^{-y_{in}}} - 1 = \frac{1 - e^{-y_{in}}}{1 + e^{-y_{in}}}$$

$$y = \frac{1 - e^{-0.53}}{1 + e^{-0.53}} = 0.259$$

OVERVIEW

- Important terminologies of ANN
  - Weights
  - Bias
  - Threshold
  - Learning Rate
  - Momentum Factor
  - Vigilance Parameter
  - Notations

IMPORTANT TERMINOLOGIES OF ANN

Weight

- The weight contain information about the input signal.
- It is used by the net to solve the problem.
- It is represented in terms of matrix & called as *connection matrix.*
- If weight matrix W contains all the elements of an ANN, then the set of all W matrices will determine the set of all possible information processing configuration.
- The ANN can be realized by finding an appropriate matrix W.

$$W = \begin{bmatrix} w_1^T \\ w_2^T \\ \cdot \\ \cdot \\ \cdot \\ w_n^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix}$$

IMPORTANT TERMINOLOGIES OF ANN

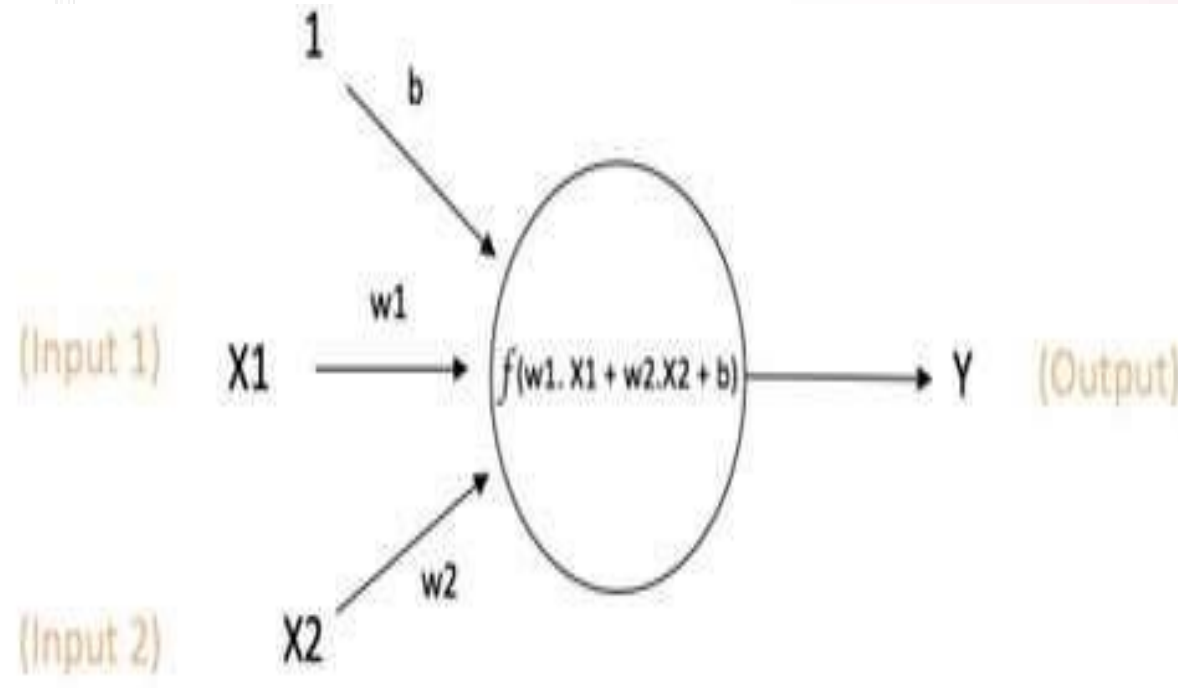
Bias

Bias has an impact in calculating net input.

Bias is included by adding x0 to the input vector x.

The bias is of two types

- **Positive bias**
- Increase the net input
- **Negative bias**
- Decrease the net input



Output of neuron = Y= $f(w1.X1 + w2.X2 + b)$

IMPORTANT TERMINOLOGIES OF ANN

Bias

The bias is considered like another weight, that is $w_{0j} = b_j$ Consider a simple network shown in Figure with bias. From Figure, the net input to the output neuron Y_j is calculated as

$$y_{inj} = \sum_{i=0}^{n} x_i w_{ij} = x_0 w_{0j} + x_1 w_{1j} + x_2 w_{2j} + \cdots + x_n w_{nj}$$

$$= w_{0j} + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

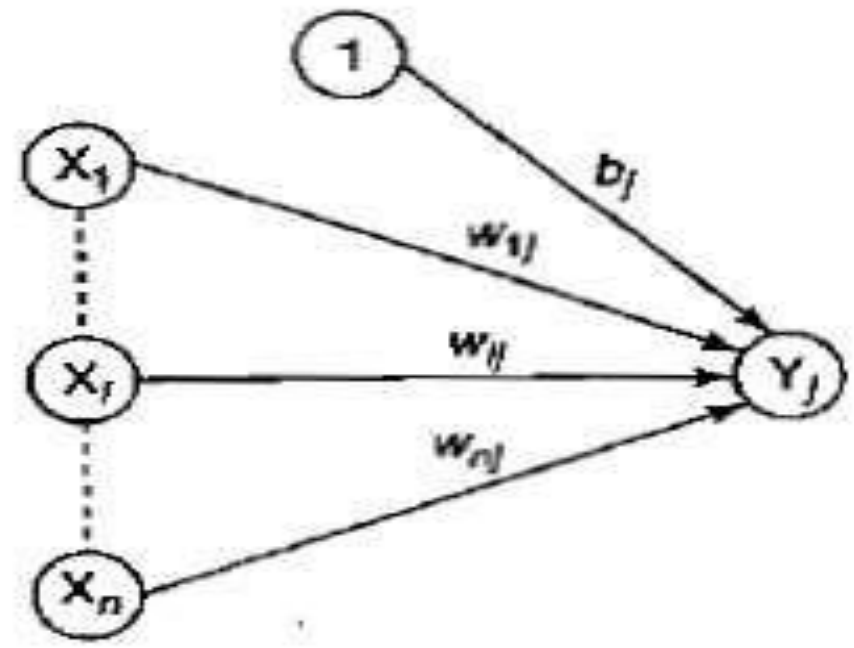


Figure Simple net with bias.

IMPORTANT TERMINOLOGIES OF ANN

Bias

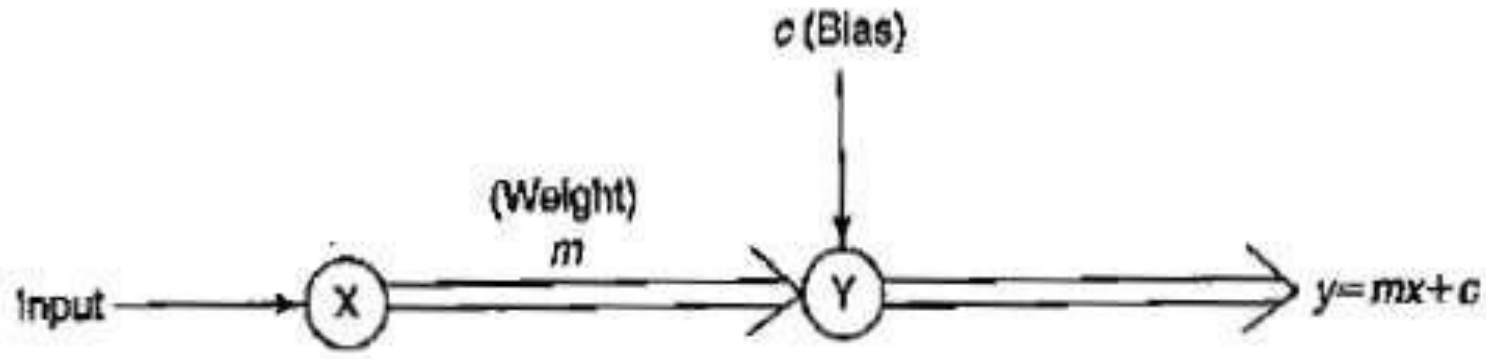


Figure 2-17 Block diagram for straight line.

The activation function discussed in Section 2.3.3 is applied over this net input to calculate the output. The bias can also be explained as follows: Consider an equation of straight line,

$$y = mx + c$$

where x is the input, m is the weight, c is the bias and y is the output. The equation of the straight line can also be represented as a block diagram shown in Figure 2-17. Thus, bias plays a major role in determining the output of the network.

IMPORTANT TERMINOLOGIES OF ANN

Threshold

It is a set value based upon which the final output is calculated.

Calculated net input and threshold is compared to get the network output.

The activation function of threshold is defined as

$$f(net) = \begin{cases} 1 & if\ net \geq \theta \\ -1 & if\ net < \theta \end{cases}$$

where θ is the fixed threshold value

Learning rate

- Denoted by α
- Control the amount of weight adjustment at each step of training.
- The learning rate range from 0 to 1.
- Determine the rate of learning at each step

IMPORTANT TERMINOLOGIES OF ANN

Momentum Factor

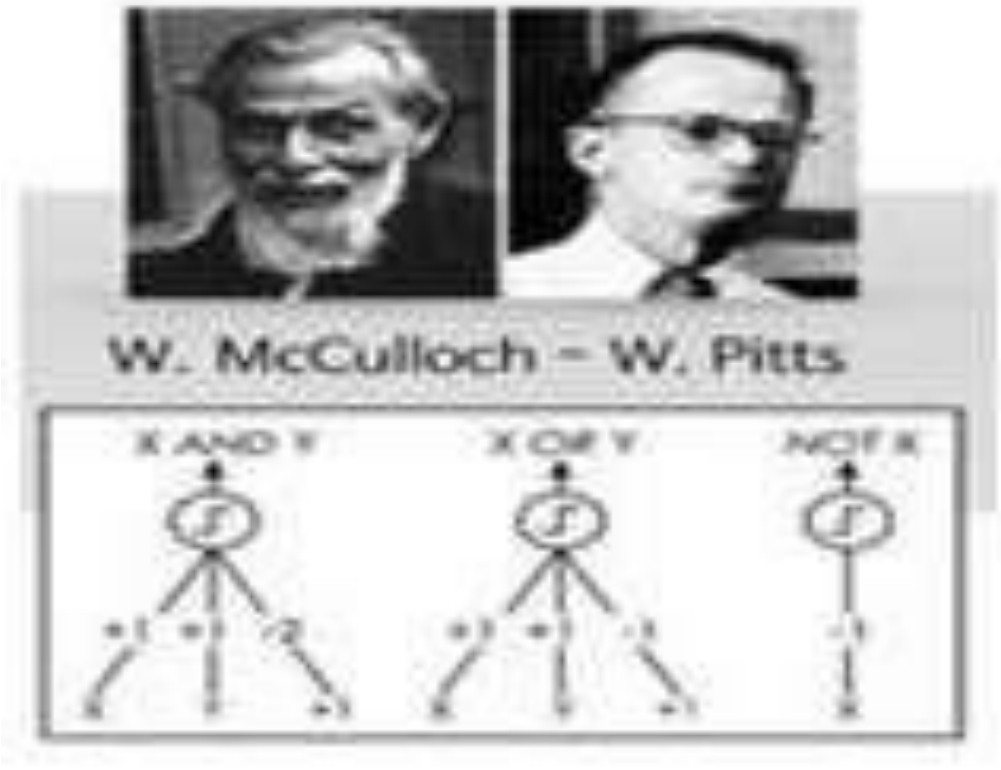
- Convergence is made faster if a momentum factor is added to the weight updation process.
- Done in back propagation network.

Vigilance parameter

- Denoted by ρ.
- Used in Adaptive Resonance Theory (ART) network.
- Used to control the degree of similarity.
- Ranges from 0.7 to 1 to perform useful work in controlling the number of clusters.

MCCULLOCH-PITTS NEURON (MP NEURON)-Theory

- The **McCulloch-Pitts neuron** was the earliest neural network discovered in **1943.**



- It is usually called as **M-P neuron.**

MCCULLOCH-PITTS NEURON (MP NEURON)-Theory

- The M-P neurons are connected by directed weighted parts.
- The activation function of an M-P neuron is **binary(output)**, that is, any step the neuron ***may fire*** or ***may not fire***.
- The weights associated with the communication links maybe:
  - **excitatory w (w>0, weight is positive) or**
  - **inhibitory p (p<0, weight is negative).**
- All excitedly connected weights entering into a particular neuron will have the **same weights**.
- The threshold plays a major role in M-P neuron.
- There is a fixed threshold for each neuron, and if the net input to the neuron is greater than the threshold then the neuron fires.
- Also, any non zero inhibitory input would prevent the neuron from firing.
- The M-P neurons are most widely used in the case of logic functions.

MCCULLOCH-PITTS NEURON (MP NEURON)-Architecture

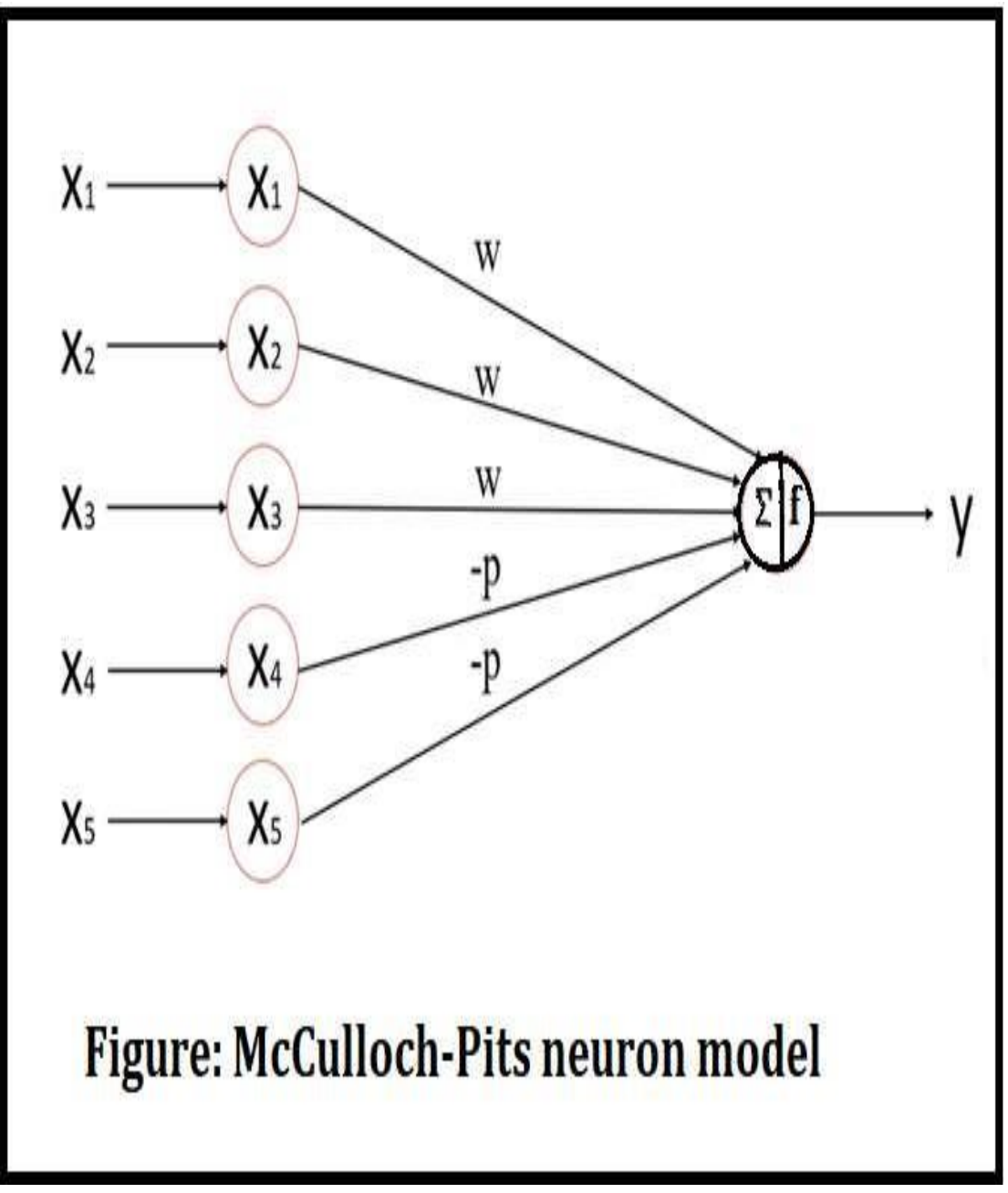
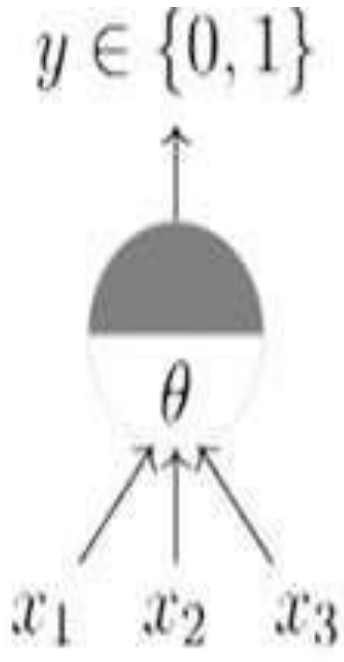


Figure: McCulloch-Pits neuron model

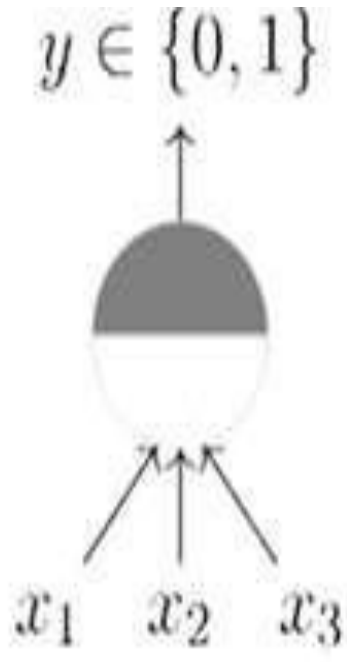
- A simple M-P neuron is shown in Figure.
- As already discussed, the M-P neuron has both excitatory and inhibitory connections.
- It is excitatory with weight ($w > 0$) or inhibitory with weight $-p(p < 0)$.
- In Figure,
  - inputs from x_1 to x_3 possess excitatory weighted connections and
  - inputs x_4 and x_5 possess inhibitory weighted interconnections.

MCCULLOCH-PITTS NEURON (MP NEURON)-LOGICAL FUNCTIONS



A McCulloch Pitts unit

AND function



OR function

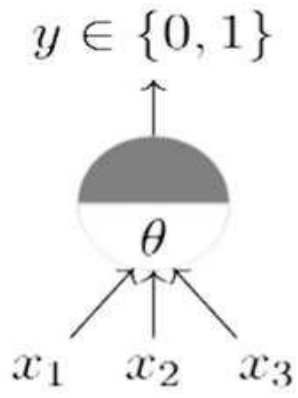
x_1 AND $!x_2$*

NOR function

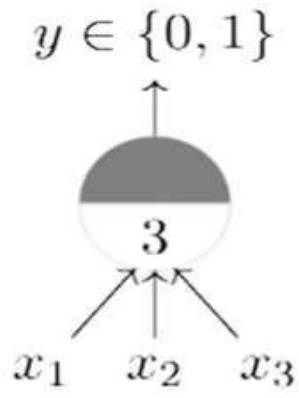
NOT function

*circle at the end indicates inhibitory input: if any inhibitory input is 1 the output will be 0

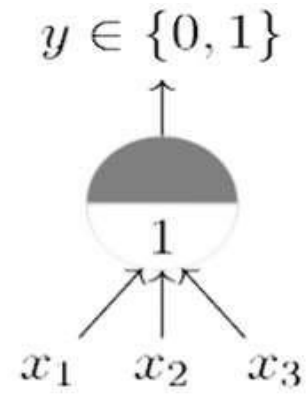
MCCULLOCH-PITTS NEURON (MP NEURON)-LOGICAL FUNCTIONS



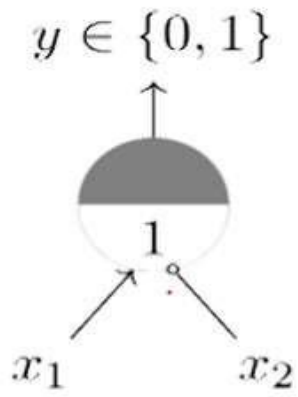
A McCulloch Pitts unit



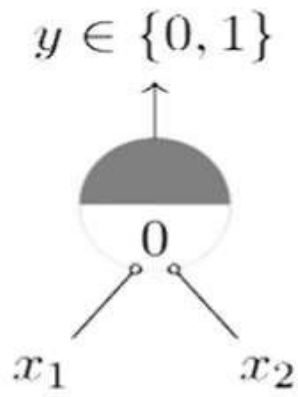
AND function



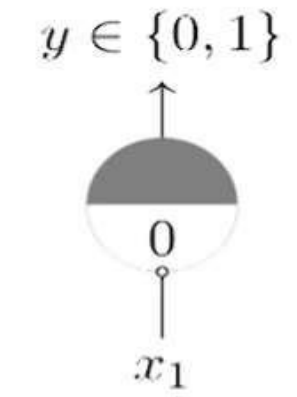
OR function



x_1 AND $!x_2$*



NOR function



NOT function

*circle at the end indicates inhibitory input: if any inhibitory input is 1 the output will be 0

DESIGN MP NEURON MODEL

Q1. Implement AND function using McCulloch-Pitts Neuron (take binary data).

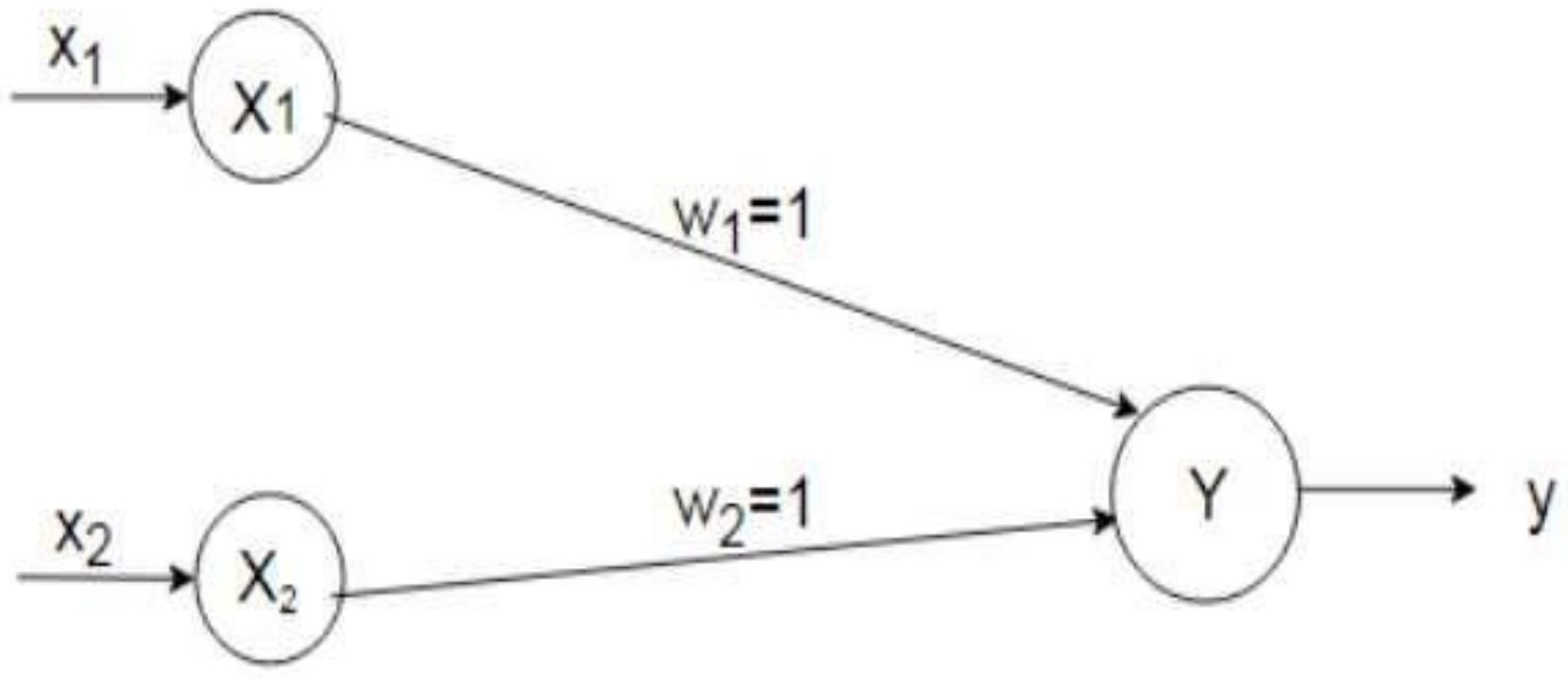
Solution:

Truth table for AND:

X_1	X_2	Y
1	1	1
1	0	0
0	1	0
0	0	0

In McCulloch-Pitts Neuron only analysis is performed. hence, assume weights be $w_1 = w_2 = 1$.

The network architecture is



X_1	X_2	Y
1	1	1
1	0	0
0	1	0
0	0	0

DESIGN MP NEURON MODEL-AND

Let w1=1, w2=1

With these assumed weights the net input is calculated for four inputs,

(i) $(1,1)$ ⇨ $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 1 \times 1 = 2$

(ii) $(1,0)$ ⇨ $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 0 \times 1 = 1$

(iii) $(0,1)$ ⇨ $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 0 + 1 \times 1 = 1$

(iv) $(0,0)$ ⇨ $y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 0 \times 1 = 0$

- For, AND function the output is high if both the inputs are high. For this function, the net input is calculated as 2.
- Hence, based on this input the threshold value is set, i.e., if the output value is greater than or equal to 2 then the neuron fires, else it does not fire.

DESIGN MP NEURON MODEL-AND

- So, the threshold value is set to 2 ($\theta = 2$). This can be obtained by,

$$\theta \geq nw - p$$

- Here, $n = 2, w = 1$ (excitory) and $p = 0$ (inhibitory)

$$\therefore \theta \geq 2 \times 1 - 0$$

$$\Rightarrow \theta \geq 2$$

- The output of neuron Y can be written as,

$$y = f(y_{in}) = \begin{cases} 1, & if \quad y_{in} \geq 2 \\ 0, & if \quad y_{in} < 2 \end{cases}$$

DESIGN MP NEURON MODEL-AND NOT

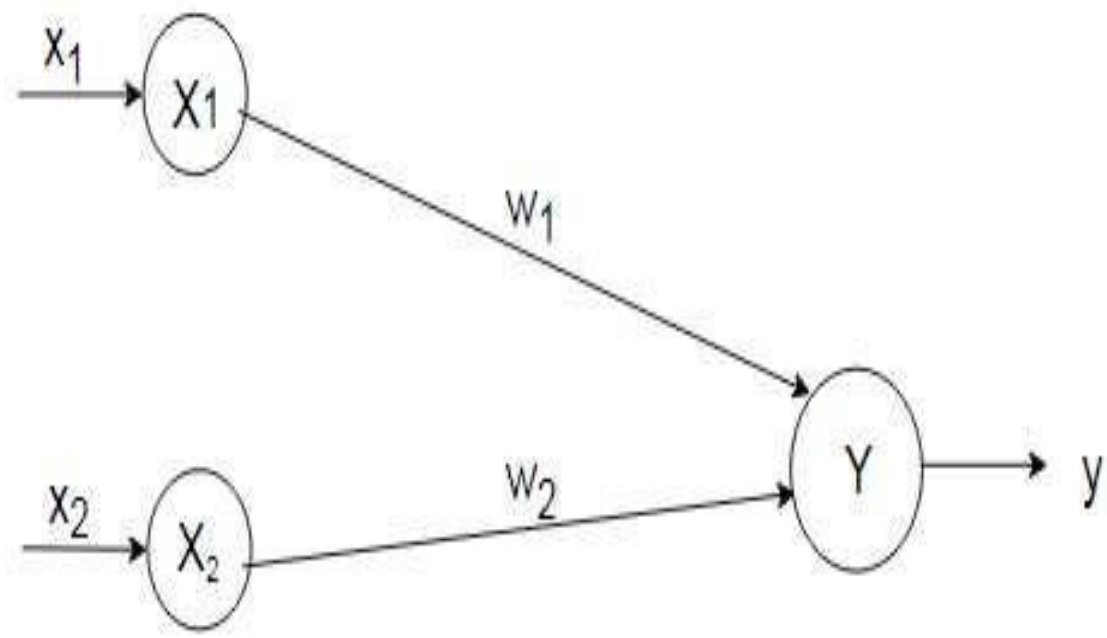
Q2. Use McCulloch-Pitts Neuron to implement AND NOT function (take binary data).

Solution:

Truth table for **AND NOT**:

- The given function gives an output only when X1=1 and X2=0.
- The weights have to be decided only after analysis.

X1	X2	Y
1	1	0
1	0	1
0	1	0
0	0	0



DESIGN MP NEURON MODEL-AND NOT

Case 1: Assume both the weights as excitatory, i.e., **w1=w2=1**, **The net input,**

X1	X2	Y
1	1	0
1	0	1
0	1	0
0	0	0

(i) $(1,1) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 1 \times 1 = 2$

(ii) $(1,0) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 0 \times 1 = 1$

(iii) $(0,1) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 0 + 1 \times 1 = 1$

(iv) $(0,0) - y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 0 \times 1 = 0$

- From the calculated net inputs, it is not possible to fire the neuron for input (1, 0) only. Hence, these weights are not suitable.

DESIGN MP NEURON MODEL-AND NOT

Case 1: Assume one weight as excitatory and another one as inhibitory, i.e., **w1=1, w2=-1,**

The net input,

X1	X2	Y
1	1	0
1	0	1
0	1	0
0	0	0

(i) $(1,1) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 1 \times (-1) = 0$

(ii) $(1,0) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 0 \times (-1) = 1$

(iii) $(0,1) - y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 0 + 1 \times (-1) = -1$

(iv) $(0,0) - y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 0 \times (-1) = 0$

- From the net inputs now it is possible to conclude that the neuron will only fire with input (1,0) by fixing the threshold $\theta \geq 1$.

DESIGN MP NEURON MODEL-AND NOT

- Thus, **w1=1,w2=−1 ;θ≥1**
- The value of **θ** is calculated as,(n=2,w=1,p=1)

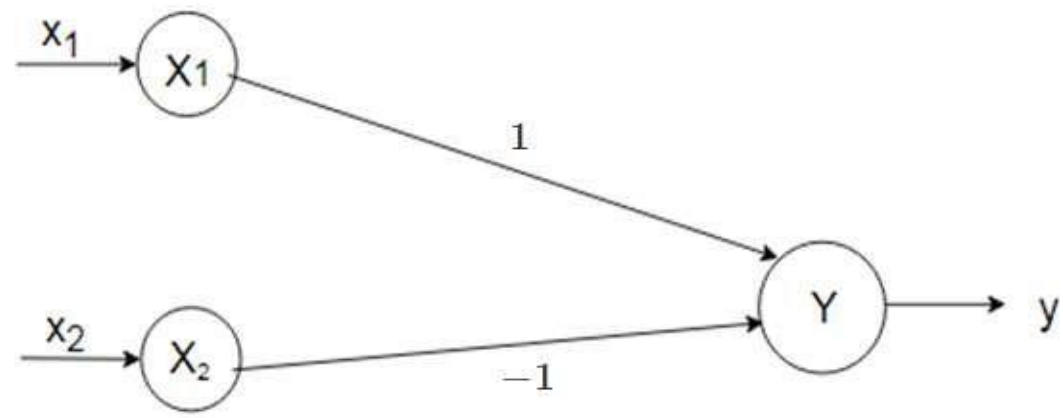
  θ≥nw−p

  θ≥2×1−1

  θ≥1

$$y = f(_{in}) = \begin{cases} 1, & if \quad y_{in} \geq 1 \\ 0, & if \quad y_{in} < 1 \end{cases}$$

- The output of the neuron Y can be written as,



- Finally, MP Neuron network for **AND NOT→**

DESIGN MP NEURON MODEL-XOR

Q3. Implement XOR function using M-P neuron. (consider Binary Data)

Solution:

Truth table for **XOR** :

X1	X2	Y
1	1	0
1	0	1
0	1	1
0	0	0

- In this case, the output is **"ON"** only for odd number of 1's. For the rest it is "OFF".
- XOR function cannot be represented by simple and single logic function, it is represented as

$$y = x_1\overline{x_2} + \overline{x_1}x_2$$

$$y = z_1 + z_2$$

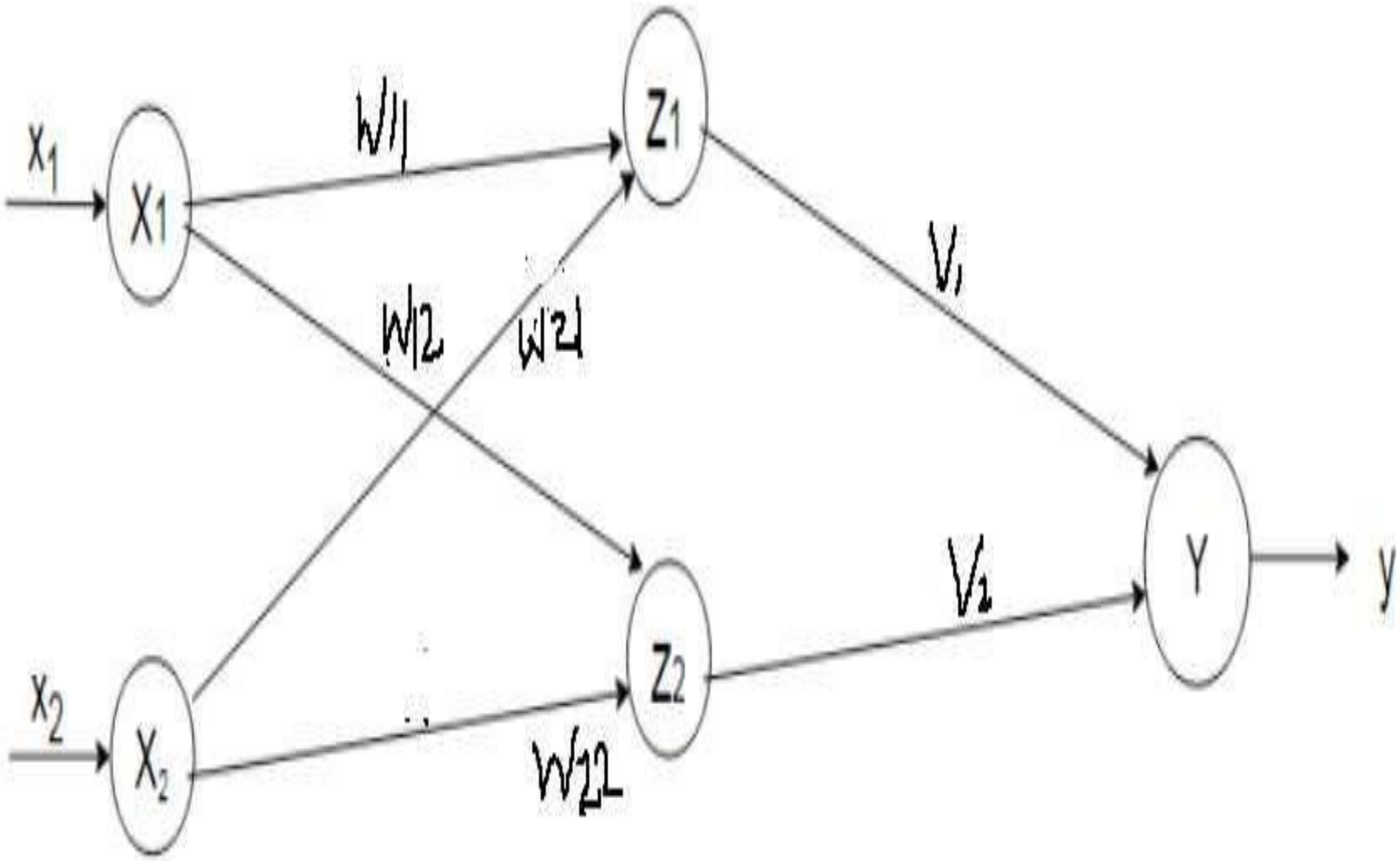
where $z_1 = x_1 . \overline{x_2}$ is the first function

and $z_2 = \overline{x_1} . x_2$ is the second function.

$\Rightarrow y = z_1 + z_2$ is the third function

DESIGN MP NEURON MODEL-XOR

- A single layer net is not sufficientto representthe function. An intermediate layer is necessary. Thus, Neural Network for XOR function



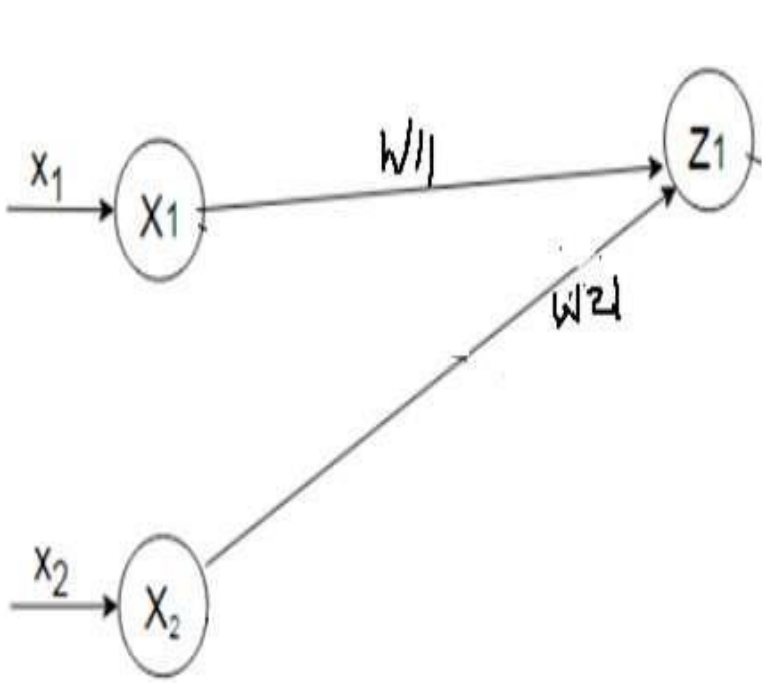
DESIGN MP NEURON MODEL-XOR

- **First Function:**

$$z_1 = x_1 . \overline{x_2}$$

Truth table for $z = x_1 . \overline{x_2}$

x_1	x_2	y
1	1	0
0	1	0
1	0	1
0	0	0



Case 1: Assume both the weights are excitatory,

i.e., $w_{11} = 1, w_{21} = 1$

$$y_{in} = x_1 w_{11} + x_2 w_{21}$$

$$(1,1) - y_{in} = 1 \times 1 + 1 \times 1 = 2$$

$$(1,0) - y_{in} = 1 \times 1 + 0 \times 1 = 1$$

$$(0,1) - y_{in} = 0 \times 1 + 1 \times 1 = 1$$

$$(0,0) - y_{in} = 0 \times 1 + 0 \times 1 = 0$$

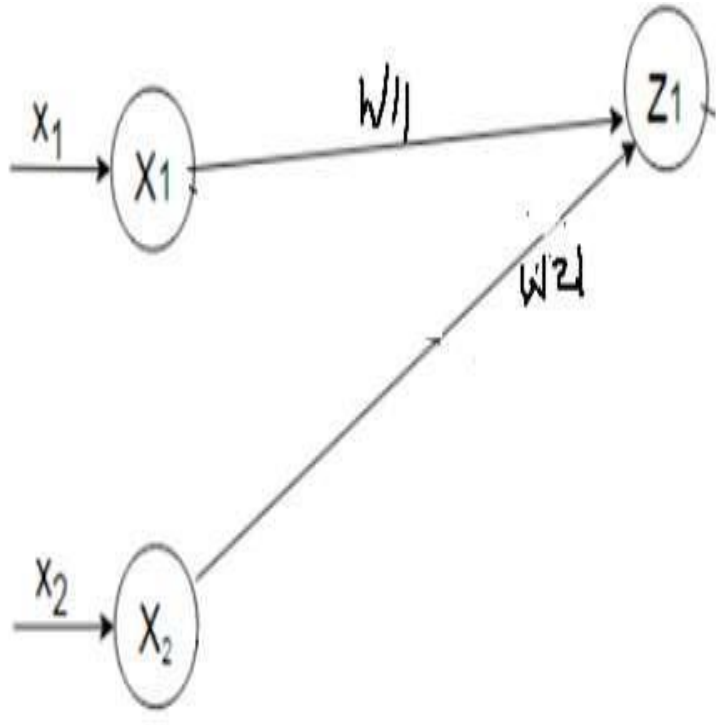
Hence, it is not possible to obtain activation function using this weights.

DESIGN MP NEURON MODEL-XOR

- **First Function:** $z_1 = x_1 . \overline{x_2}$

Truth table for $z_1 = x_1 . \overline{x_2}$

x_1	x_2	z_1
1	1	0
0	1	0
1	0	1
0	0	0



Case 2: Consider one weight excitatory and another weight inhibitor

i.e., $w_{11} = 1, w_{21} = -1$

$$y_{in} = x_1 w_{11} + x_2 w_{21}$$

$$(1,1) - y_{in} = 1 \times 1 + 1 \times (-1) = 0$$

$$(1,0) - y_{in} = 1 \times 1 + 0 \times (-1) = 1$$

$$(0,1) - y_{in} = 0 \times 1 + 1 \times (-1) = -1$$

$$(0,0) - y_{in} = 0 \times 1 + 0 \times (-1) = 0$$

For this weight, it is possible to get the desired output. Hence,

$w_{11} = 1, w_{21} = -1$

$\therefore \theta \geq nw - p$

$\Rightarrow \theta \geq 2 \times 1 - 1$

$\theta \geq 1$

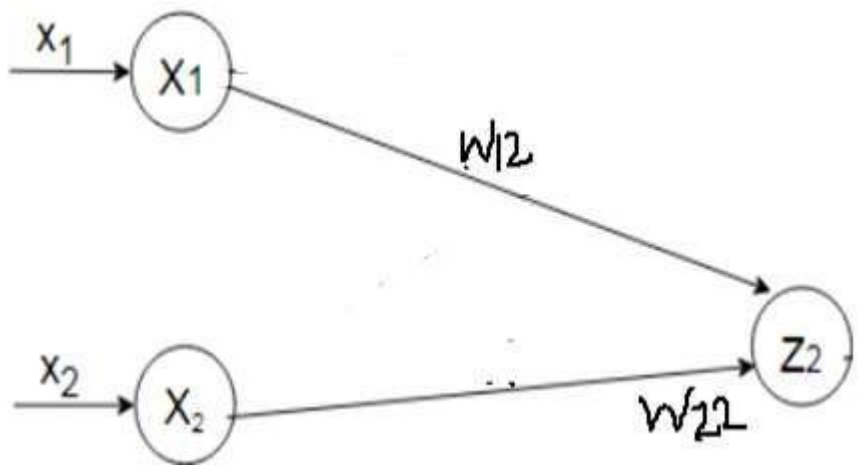
This is for z_1 neuron.

DESIGN MP NEURON MODEL-XOR

- **Second Function:** $z_2 = \overline{x_1}.\, x_2$

The truth table is as shown below,

x_1	x_2	z_2
1	1	0
0	1	1
1	0	0
0	0	0



Case 1: Assume both the weights are excitatory,

i.e., $w_{12} = 1, w_{22} = 1$

$$y_{in} = x_1 w_{11} + x_2 w_{21}$$

$$(1,1) - y_{in} = 1 \times 1 + 1 \times 1 = 2$$

$$(1,0) - y_{in} = 1 \times 1 + 0 \times 1 = 1$$

$$(0,1) - y_{in} = 0 \times 1 + 1 \times 1 = 1$$

$$(0,0) - y_{in} = 0 \times 1 + 0 \times 1 = 0$$

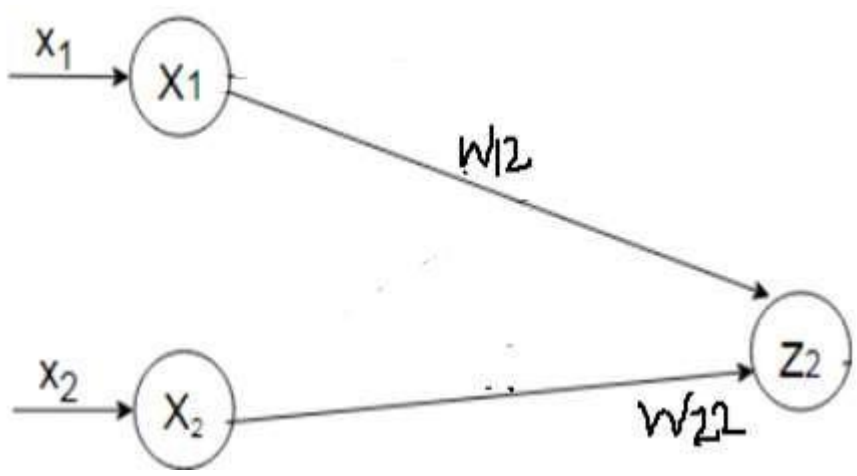
Hence, it is not possible to obtain activation function using this weights.

DESIGN MP NEURON MODEL-XOR

- **Second Function:** $z_2 = \overline{x_1}.\, x_2$

The truth table is as shown below,

x_1	x_2	z_2
1	1	0
0	1	1
1	0	0
0	0	0



Case 2: Consider one weight excitatory and another weight inhibitory

i.e., $w_{12} = -1, w_{22} = 1$

$$y_{in} = x_1 w_{11} + x_2 w_{21}$$

$$(1,1) - y_{in} = 1 \times (-1) + 1 \times 1 = 0$$

$$(1,0) - y_{in} = 1 \times (-1) + 0 \times 1 = -1$$

$$(0,1) - y_{in} = 0 \times (-1) + 1 \times 1 = 1$$

$$(0,0) - y_{in} = 0 \times (-1) + 0 \times 1 = 0$$

For this weight, it is possible to get the desired output. Hence,

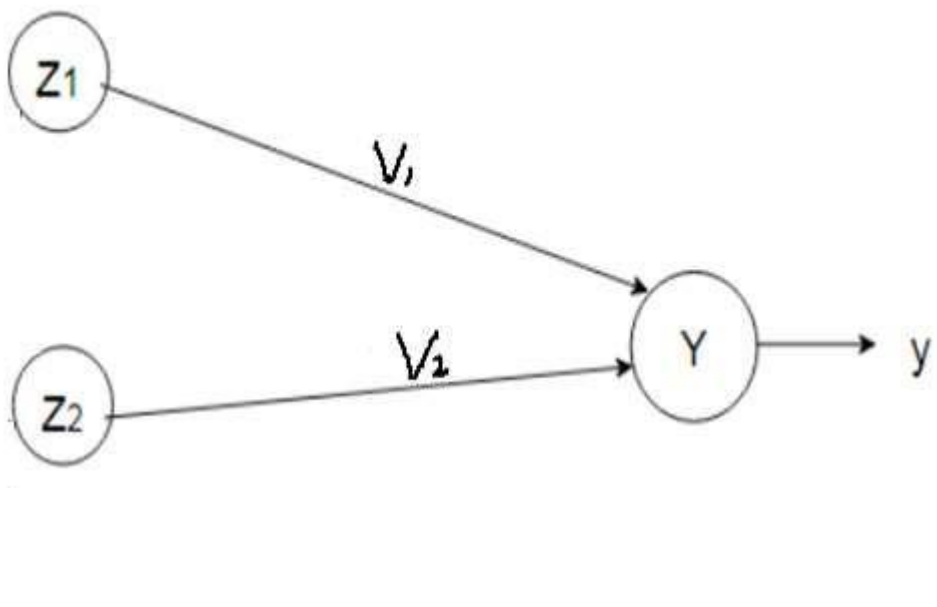
$$w_{21} = -1, w_{22} = 1$$
$$\therefore \theta \geq nw - p$$
$$\Rightarrow \theta > 2 \times 1 - 1$$
$$\theta \geq 1$$

DESIGN MP NEURON MODEL-XOR

- **Third Function:** $y = z_1$ or z_2

The truth table is

x_1	x_2	y	z_1	z_2
0	0	0	0	0
0	1	1	0	1
1	0	1	1	0
1	1	0	0	0



Here the net input is calculated as,

$$y_{in} = z_1 v_1 + z_2 v_2$$

Case 1: Assume both the weights excitatory

i.e., $v_1 = v_2 = 1$

The net input,

$$(0, 0) - y_{in} = 0 \times 1 + 0 \times 1 = 0$$

$$(0, 1) - y_{in} = 0 \times 1 + 1 \times 1 = 1$$

$$(1, 0) - y_{in} = 1 \times 1 + 0 \times 1 = 1$$

$$(1, 1) - y_{in} = 0 \times 1 + 0 \times 1 = 0$$

Setting a threshold of $\theta \geq 1$, $v_1 = v_2 = 1$, which implies that the net is recognized. Therefore, the analysis is made for XOR function using M–P neurons.

DESIGN MP NEURON MODEL-XOR

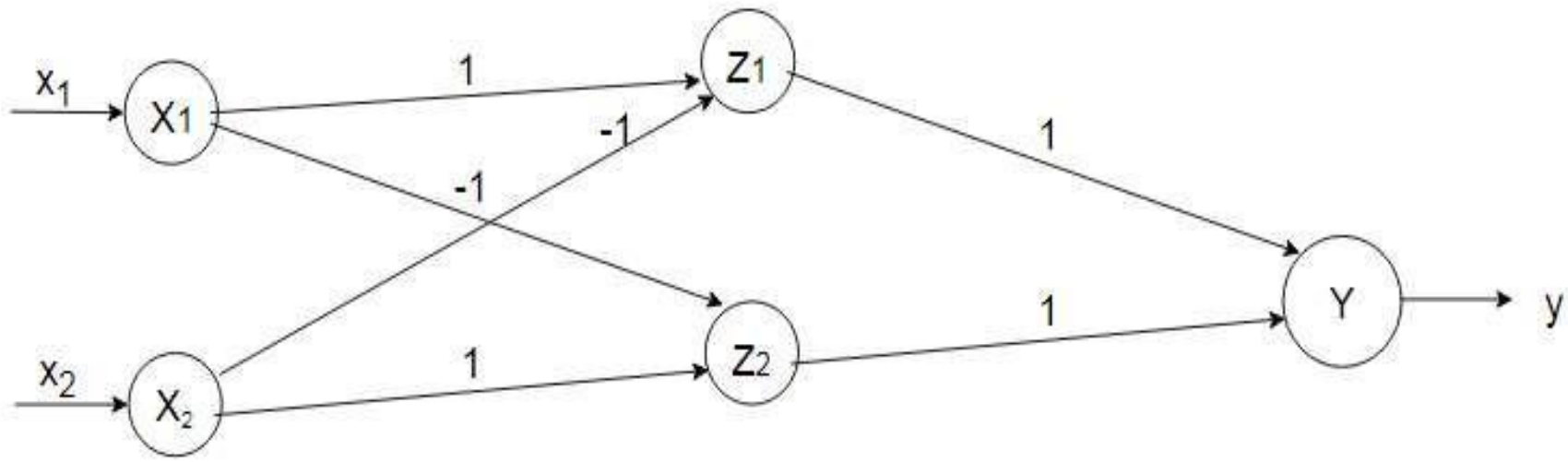
Thus, the weights are obtained as following for the XOR function,

$w_{11} = w_{22} = 1$ (excitatory)

$w_{12} = w_{21} = -1$ (inhibitory)

$v_1 = v_2 = 1$

Finally, MP Neuron network for **XOR**



LIMITATION OF M-P MODEL

- Can only deal with binary inputs and binary output.
- Can't work with real life values like years, price, age etc.
- Considers that all weights are equal
- Using those approach we can concentrate neuron models for only those problems that are linearly separable.
- We are just calculating the threshold values ourselves.

No (Machine) Learning
in
McCulloch-Pitts Neuron Model

- To Overcome the limitations of the M-P neuron,
  - Frank Rosenblatt, an American psychologist, proposed the classical perception model, the mighty *artificial neuron*, in 1958. It is more generalized computational model than the McCulloch-Pitts neuron where weights and thresholds can be learnt over time.

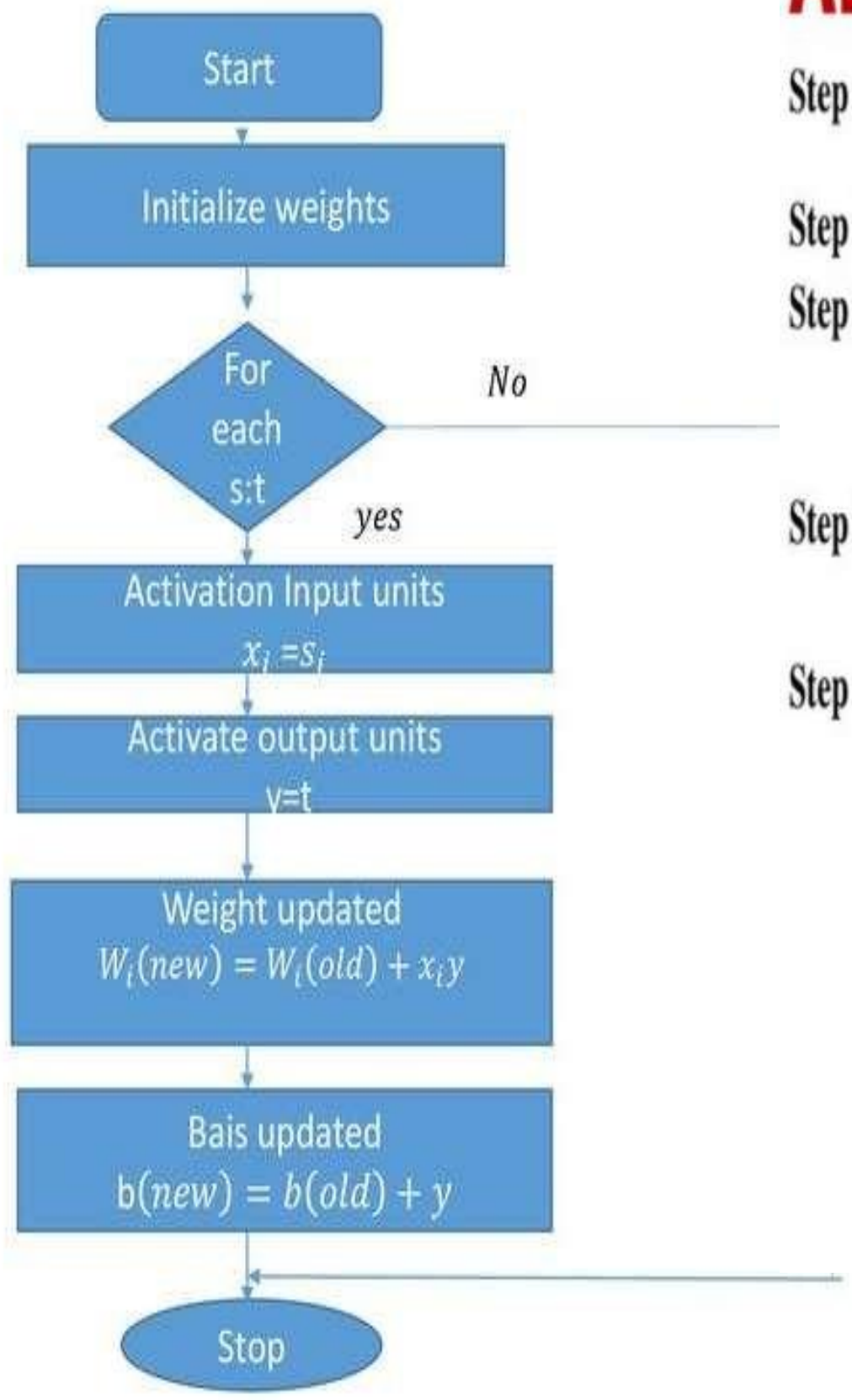


Figure - Flowchart of Hebb training algorithm.

ALGORITHM:-

Step 0:-First Initialize the weights. Basically in this network they may be set to zero i.e., W_i=0 for i=1 to n where "n" may be the total number of input neurons

Step 1:- Steps 2-4 have to be performed for each input training vector and target vector s:t

Step 2:- Input units activation are set. Generally the activation function of input layer its identity function

$$x_i = s_i \text{ for i=1 to n}$$

Step 3:- Output units activations are set

$$y=t$$

Step 4:-Weight adjustments and bias adjustments are performed

$$W_i(new) = W_i(old) + x_i y$$

$$b(new) = b(old) + y$$

The above steps complete the algorithmic process. In step 4,the weight updation formula can also be given in the vector form as

$$\text{delta } w = x_i y$$

As a result,

$$W\ (new) = W\ (old) + \text{delta } w$$

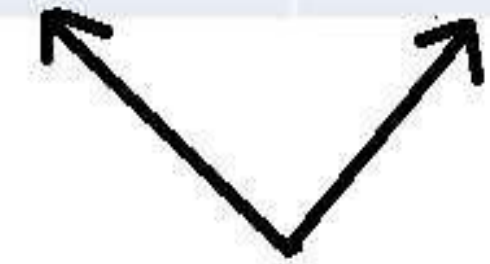
Hebb rule is used for pattern association, pattern categorization, pattern classification and over a range of other areas

HEBB NETWORK-AND

Q. Design A Hebb net to implement logical AND function (use bipolar inputs and targets).

Solution: Truth table for AND wit bipolar inputs and targets

X1	X2	b	Y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1



Bipolar inputs

HEBB NETWORK-AND

Initialize weights and bias i.e. w1=w2=b=0

Update weights

1. For input pair X1,X2,b = (1,1,1)

wi(new) = wi(old) + (xi * y)

w1(new) = w1(old) + (x1*y)

= 0 + 1*1 =1

w2(new) = w2(old) + (x2*y)

= 0 + 1*1 =1

b(new) = b(old) + y

= 0+1 = 1

X1	X2	b	Y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

Bipolar inputs

HEBB NETWORK-AND

2. For input pair (1,-1,-1)

w1(new) = w1(old) + (x1*y)
= 1 + 1 * (-1) = 0
w2(new) = w2(old) + (x2*y)
= 1 + (-1)*(-1) = 2
b(new) = b(old) + y
= 1 + (-1) =0

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	-1			
-1	1	1	-1			
-1	-1	1	-1			

HEBB NETWORK-AND

3. For input pair (-1,1,1)

w1(new) = w1(old) + (x1*y)

= 0 + (-1) * (-1) = 1

w2(new) = w2(old) + (x2*y)

= 2 + (1)*(-1) = 1

b(new) = b(old) + y

= 0 + (-1) = -1

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	-1	0	2	0
-1	1	1	-1			
-1	-1	1	-1			

HEBB NETWORK-AND

4. For input pair (-1,-1,-1)

w1(new) = w1(old) + (x1*y)
= 1 + (-1) * (-1) = 2
w2(new) = w2(old) + (x2*y)
= 1 + (-1)*(-1) = 2
b(new) = b(old) + y
= (-1) + (-1) =-2

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	-1	0	2	0
-1	1	1	-1	1	1	1
-1	-1	1	-1			

HEBB NETWORK-AND

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	-1	0	2	0
-1	1	1	-1	1	1	-1
-1	-1	1	-1	2	2	-2

Final values w1 = 2 w2 = 2 b=-2

HEBB NETWORK-AND

Net Value

Net value = w1x1 + w2x2 + b

1. For input (1,1)

Net = 2*1 + 2*1 - 2 = 2

2. For input (1,-1)

Net = 2*1 + 2*(-1) - 2 = -2

3. For input (-1,1)

Net = 2*(-1) + 2*1 - 2 = -2

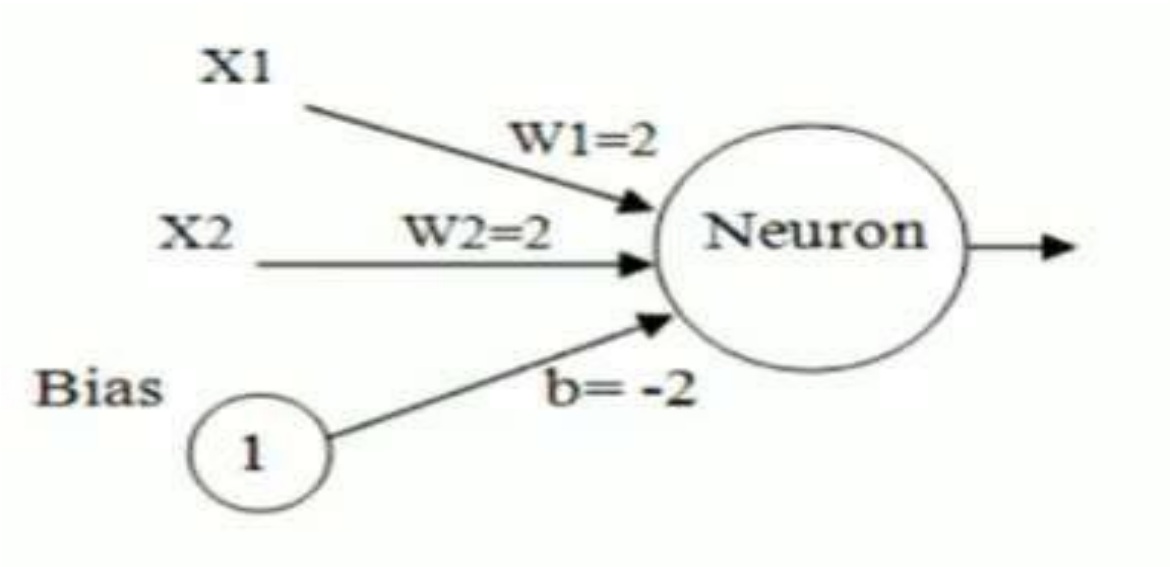
4. For input (-1,-1)

Net = 2*(-1) + 2*(-1) - 2 = -6

Final values w1 = 2 w2 = 2 b=-2

X1	X2	b	Y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

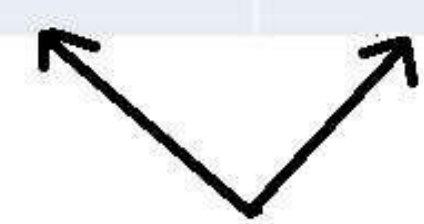
$$f(net) = \begin{cases} 1, & Net >= 2 \\ -1, & Net < 2 \end{cases}$$



HEBB NETWORK-OR

Design a Hebb net to implement OR function (consider bipolar inputs and targets).

X1	X2	b	Y
1	1	1	1
1	-1	1	1
-1	1	1	1
-1	-1	1	-1



Bipolar inputs

HEBB NETWORK-OR

Initialize weights and bias i.e. w1=w2=b=0

Update weights

1. For input pair X1,X2,b = (1,1,1)

wi(new) = wi(old) + (xi * y)
w1(new) = w1(old) + (x1*y)
= 0 + 1*1 =1
w2(new) = w2(old) + (x2*y)
= 0 + 1*1 =1
b(new) = b(old) + y
= 0+1 = 1

X1	X2	b	Y
1	1	1	1
1	-1	1	1
-1	1	1	1
-1	-1	1	-1

Bipolar inputs

HEBB NETWORK-OR

2. For input pair (1,-1, 1)

w1(new)=w1(old)+(x1*y)
=1 + 1 * 1 =2
w2(new)=w2(old)+(x2*y)
=1 + (-1) *(1) =0
b(new)=b(old) + y
=1 + 1 =2

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	1			
-1	1	1	1			
-1	-1	1	-1			

HEBB NETWORK-OR

3. For input pair (-1, 1,1)

w1(new)=w1(old)+(x1*y)
=2 + (-1) * 1 =1
w2(new)=w2(old)+(x2*y)
=0 + (1) *(1) = 1
b(new)=b(old) + y
=2 + 1 =3

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	1	2	0	2
-1	1	1	1			
-1	-1	1	-1			

HEBB NETWORK-AND

4. For input pair (-1,-1, -1)

w1(new)=w1(old)+(x1*y)
=1 + (-1) * (-1) =2
w2(new)=w2(old)+(x2*y)
=1 + (-1) *(-1) = 2
b(new)=b(old) + y
=3 + (-1) =2

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	1	2	0	2
-1	1	1	1	1	1	3
-1	-1	1	-1			

HEBB NETWORK-OR

X1	X2	b	Y	w1	w2	b
1	1	1	1	1	1	1
1	-1	1	1	2	0	2
-1	1	1	1	1	1	3
-1	-1	1	-1	2	2	2

Final values w1 = 2 w2 = 2 b= 2

HEBB NETWORK-OR

Net Value

Net value = w1x1 + w2x2 + b

1. For input (1,1)

Net = 2*1 + 2*1 +2 =6

2. For input (1,-1)

Net = 2*1 + 2*(-1) +2 = 2

3. For input (-1,1)

Net = 2*(-1) + 2*1 +2 = 2

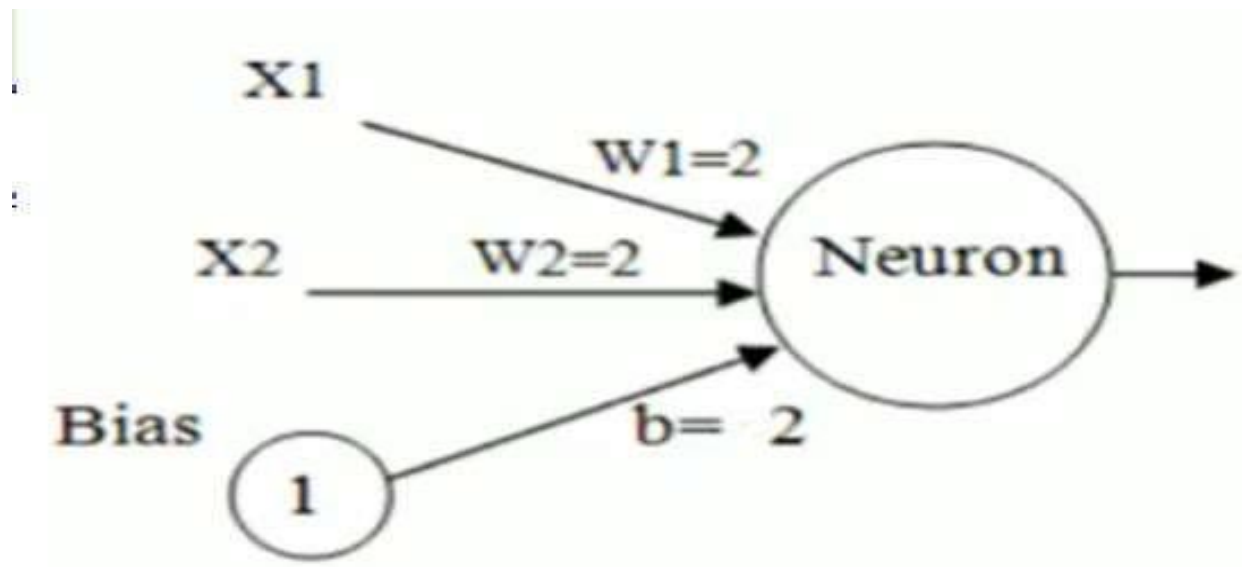
4. For input (-1,-1)

Net = 2*(-1) + 2*(-1) +2 = -2

$$f(net) = \begin{cases} 1, & Net >= 2 \\ -1, & Net < 2 \end{cases}$$

Final values w1 = 2 w2 = 2 b= 2

X1	X2	b	Y
1	1	1	1
1	-1	1	1
-1	1	1	1
-1	-1	1	-1



HEBB NETWORK-Classification Of Patterns

Q. Using the Hebb rule, find the weights required to perform the following classifications of the given input patterns shown in Figure. The pattern is shown as 3 x 3 matrix form in the squares. The "+" symbols represent the value "1" and empty squares indicate "-1." Consider "I" belongs to the members of class (so has target value 1) and "O" does not belong to the members of class (so has target value -1).

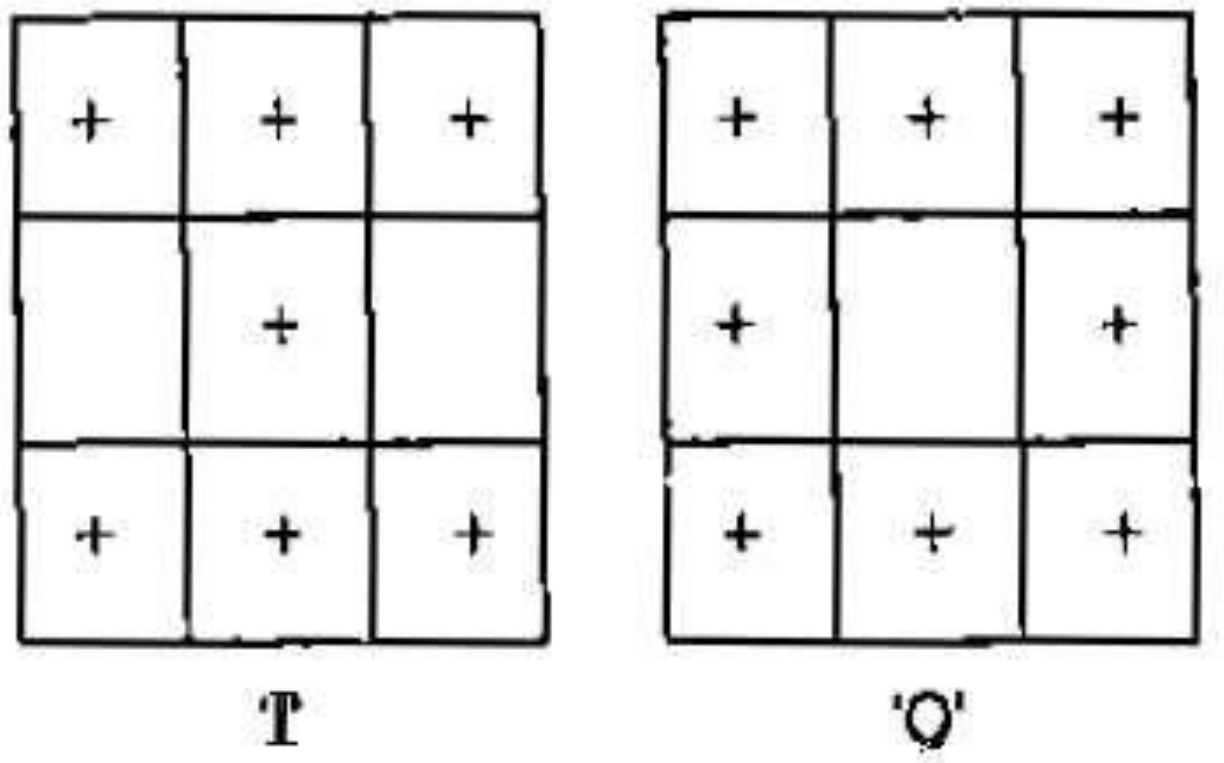
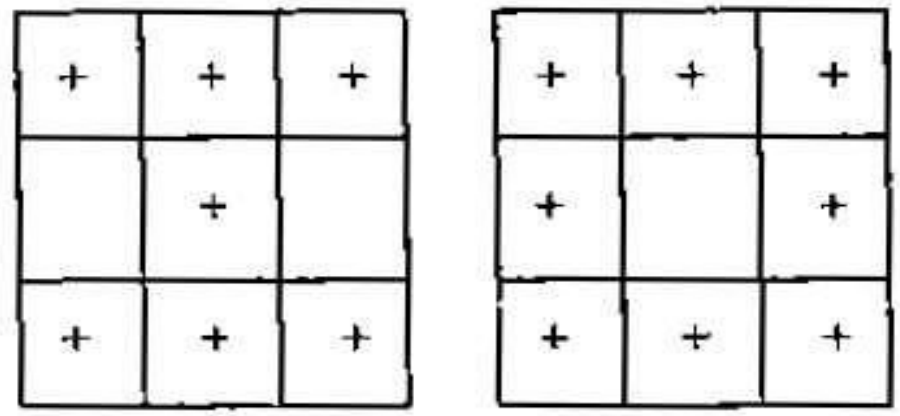


Figure Data for input patterns.

HEBB NETWORK-Classification Of Patterns



T 'O'

Solution: The training input patterns for the given net (Figure) are indicated in Table

Table

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
I	1	1	1	−1	1	−1	1	1	1	1	1
O	1	1	1	1	−1	1	1	1	1	1	−1

HEBB NETWORK-Classification Of Patterns

Table

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
I	1	1	1	−1	1	−1	1	1	1	1	1
O	1	1	1	1	−1	1	1	1	1	1	−1

➢Here a single-layer network with nine input neurons, one bias and one output neuron is formed.

➢ Set the **initial weights and bias to zero**, i.e.,

w1=w2=w3=w4=w5=w6=w7=w8=w9=b=0

HEBB NETWORK-Classification Of Patterns

Table

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
I	1	1	1	−1	1	−1	1	1	1	1	1
O	1	1	1	1	−1	1	1	1	1	1	−1

Case 1: Presenting first input pattern (I), we calculate change in weights:

$$\Delta w_i = x_i y, \quad i = 1 \text{ to } 9$$

$\Delta w_1 = x_1 y = 1 \times 1 = 1$

$\Delta w_2 = x_2 y = 1 \times 1 = 1$

$\Delta w_3 = x_3 y = 1 \times 1 = 1$

$\Delta w_4 = x_4 y = -1 \times 1 = -1$

$\Delta w_5 = x_5 y = 1 \times 1 = 1$

$\Delta w_6 = x_6 y = -1 \times 1 = -1$

$\Delta w_7 = x_7 y = 1 \times 1 = 1$

$\Delta w_8 = x_8 y = 1 \times 1 = 1$

$\Delta w_9 = x_9 y = 1 \times 1 = 1$

$\Delta b = y = 1$

Step 4:-Weight adjustments and bias adjustments are performed

$$W_i(new) = W_i(old) + x_i y$$

$$\text{b}(new) = b(old) + y$$

The above steps complete the algorithmic process. In step 4,the weight updation formula can also be given in the vector form as

$$\text{delta w} = x_i y$$

As a result,

$$W \ \ (new) = W \ \ (old) + delta\, w$$

HEBB NETWORK-Classification Of Patterns

We now calculate the new weights using the formula

$$w_i(\text{new}) = w_i(\text{old}) + \Delta w_i$$

Setting the old weights as the initial weights here, we obtain

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0 + 1 = 1$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0 + 1 = 1$$

$$w_3(\text{new}) = w_3(\text{old}) + \Delta w_3 = 0 + 1 = 1$$

Similarly, calculating for other weights we get

$$w_4(\text{new}) = -1, \quad w_5(\text{new}) = 1, \quad w_6(\text{new}) = -1,$$
$$w_7(\text{new}) = 1, \quad w_8(\text{new}) = 1, \quad w_9(\text{new}) = 1,$$
$$b(\text{new}) = 1$$

The weights after presenting first input pattern are

$$W_{(\text{new})} = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1]$$

HEBB NETWORK-Classification Of Patterns

Table

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
I	1	1	1	−1	1	−1	1	1	1	1	1
O	1	1	1	1	−1	1	1	1	1	1	−1

Case 2: Now we present the second input pattern (O). The initial weights used here are the final weights obtained after presenting the first input pattern. Here, the weights are calculated as shown below (y = -1 with the initial weights being [111-11-11111]).

$$w_i(\text{new}) = w_i(\text{old}) + \Delta w_i \qquad [\Delta w_i = x_i y]$$

$$w_1(\text{new}) = w_1(\text{old}) + x_1 y = 1 + 1 \times -1 = 0$$

$$w_2(\text{new}) = w_2(\text{old}) + x_2 y = 1 + 1 \times -1 = 0$$

$$w_3(\text{new}) = w_3(\text{old}) + x_3 y = 1 + 1 \times -1 = 0$$

$$w_4(\text{new}) = w_4(\text{old}) + x_4 y = -1 + 1 \times -1 = -2$$

$$w_5(\text{new}) = w_5(\text{old}) + x_5 y = 1 + -1 \times -1 = 2$$

$$w_6(\text{new}) = w_6(\text{old}) + x_6 y = -1 + 1 \times -1 = -2$$

$$w_7(\text{new}) = w_7(\text{old}) + x_7 y = 1 + 1 \times -1 = 0$$

$$w_8(\text{new}) = w_8(\text{old}) + x_8 y = 1 + 1 \times -1 = 0$$

$$w_9(\text{new}) = w_9(\text{old}) + x_9 y = 1 + 1 \times -1 = 0$$

$$b(\text{new}) = b(\text{old}) + y = 1 + 1 \times -1 = 0$$

HEBB NETWORK-Classification Of Patterns

The final weights after presenting the second input pattern are given as $W_{(new)} = [0\ 0\ 0\ -2\ 2\ -2\ 0\ 0\ 0\ 0]$

The weights obtained are indicated in the Hebb net shown in Figure

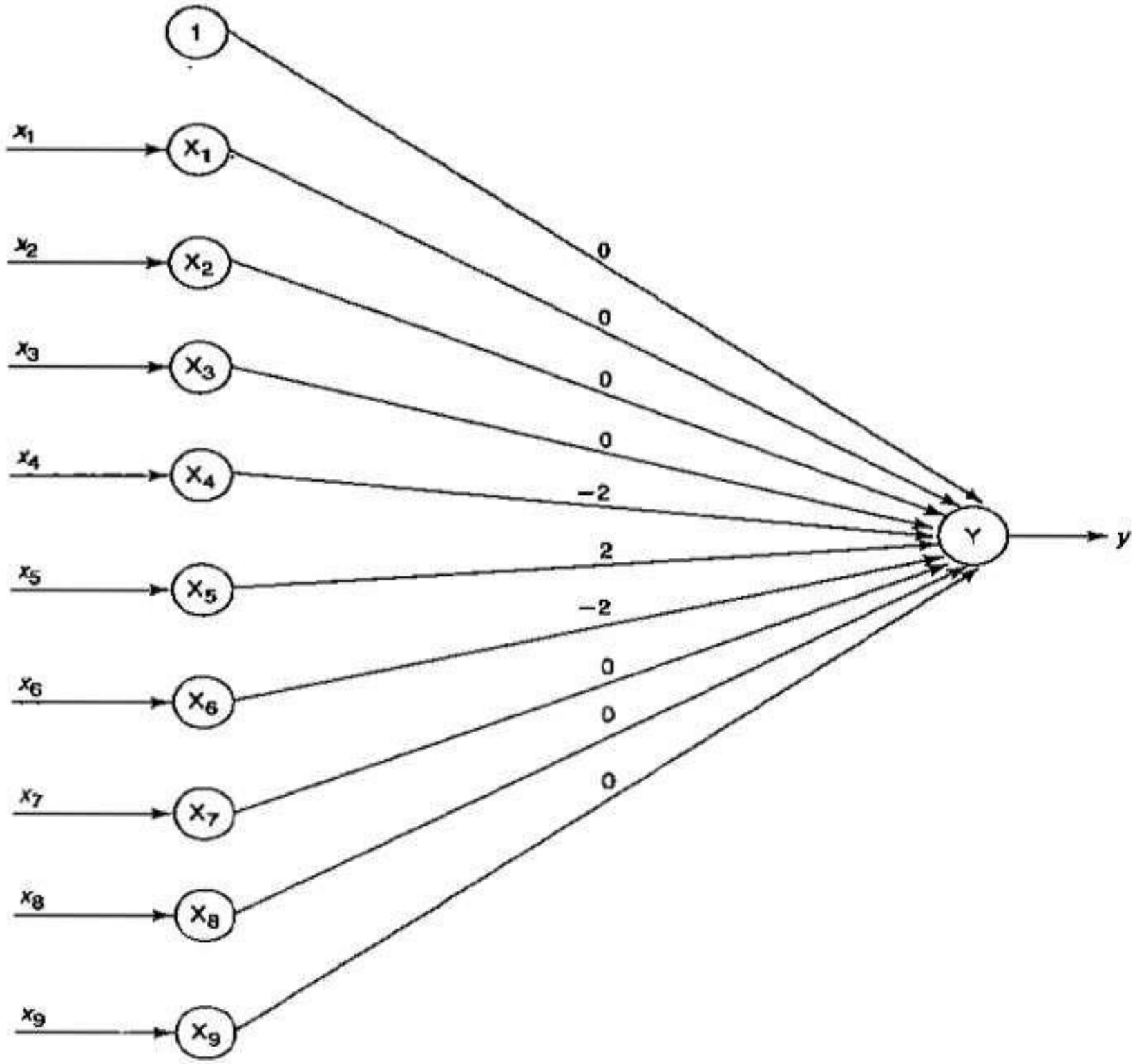


Figure Hebb net for the data matrix

HEBB NETWORK-Classification Of Patterns

Q. Find the weights required to perform the following classifications of given input patterns using the Hebb rule. The inputs are "1" where"+" symbol is present and" -1 "where"," is present. "L" pattern belongs to the class (target value+ 1) and "U" pattern does not belong to the class (target value -1).

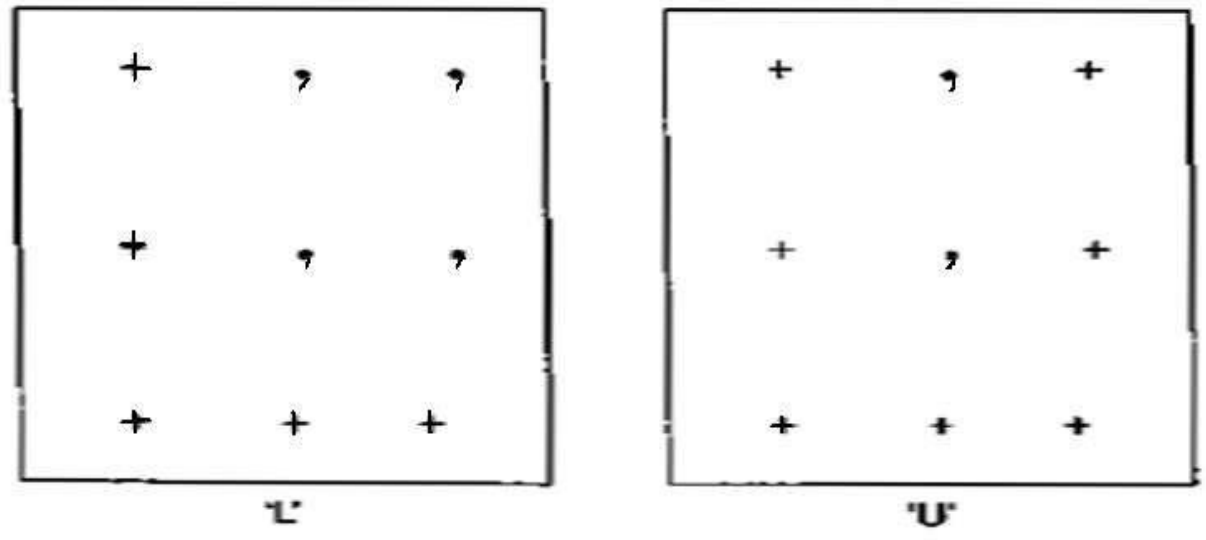


Figure Input data for given patterns.

HEBB NETWORK-Classification Of Patterns

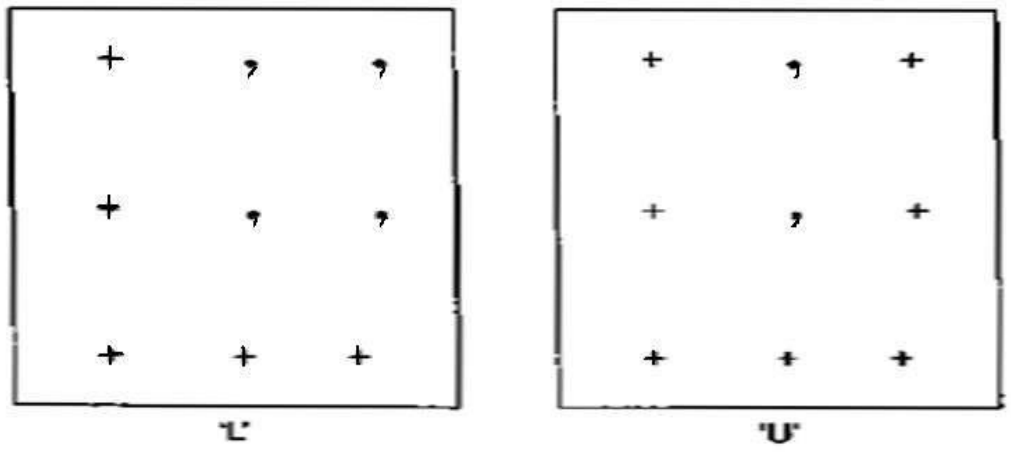


Figure Input data for given patterns.

Solution: The training input patterns for Figure are given in Table .

Table

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
L	1	−1	−1	1	−1	−1	1	1	1	1	1
U	1	−1	1	1	−1	1	1	1	1	1	−1

A single-layer network with nine input neurons, one bias and one output neuron is formed. Set the initial weights and bias to zero, i.e.,

$$w_1 = w_2 = w_3 = w_4 = w_5 = w_6 = w_7 = w_8 = w_9 = b = 0$$

The weights are calculated using

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$

HEBB NETWORK-Classification Of Patterns

The calculated weights are given in Table

Table

Inputs										Target	Weights									
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	b
											(0	0	0	0	0	0	0	0	0	0)
1	−1	−1	1	−1	−1	1	1	1	1	1	1	−1	−1	1	−1	−1	1	1	1	1
1	−1	1	1	−1	1	1	1	1	1	−1	0	0	−2	0	0	−2	0	0	0	0

The final weights after presenting the two *input* patterns are

$$W_{(new)} = [0\ 0\ -2\ 0\ 0\ -2\ 0\ 0\ 0\ 0]$$

HEBB NETWORK-Classification Of Patterns

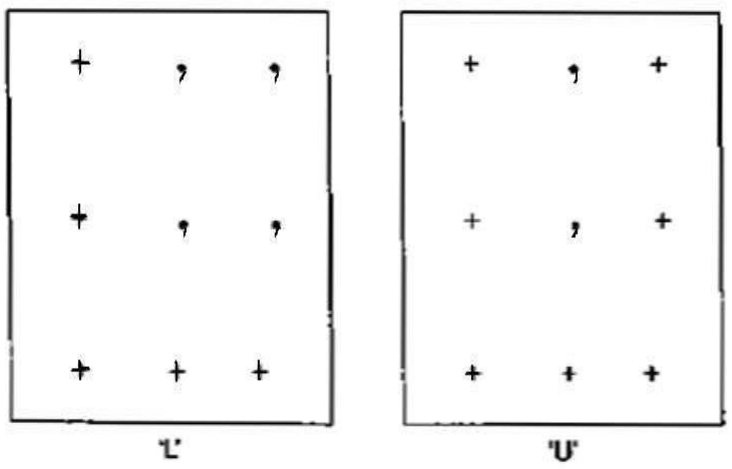
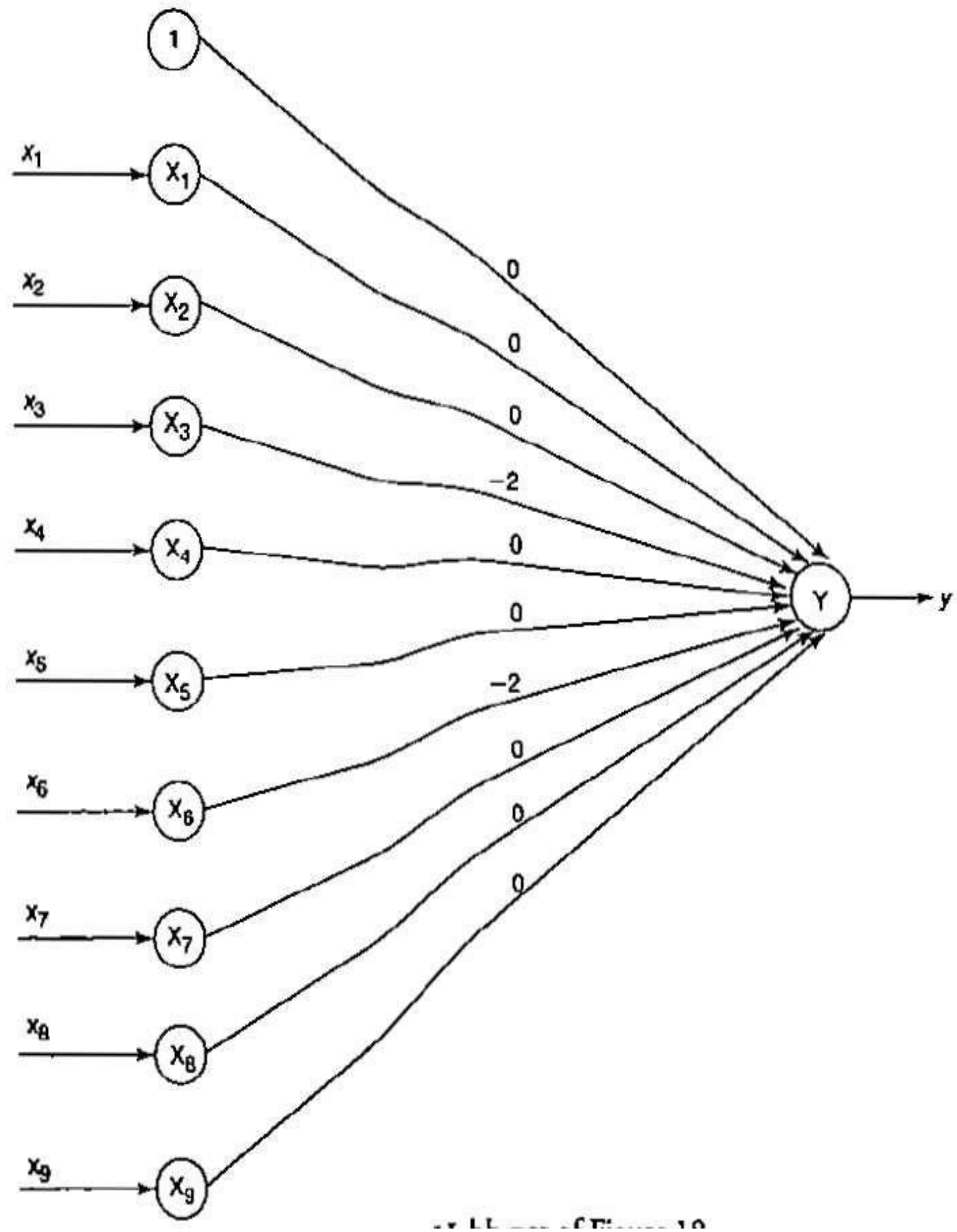


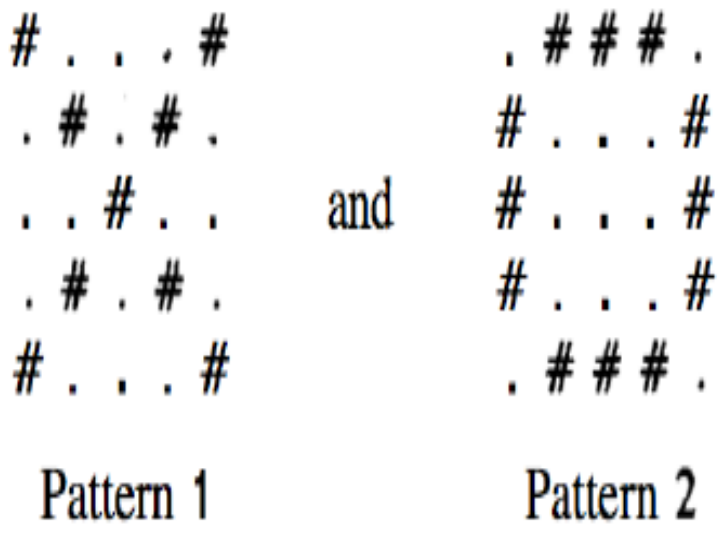
Figure Input data for given patterns.

The obtained weights are indicated in the Hebb net shown in Figure



HEBB NET TO CLASSIFY 2-D INPUT PATTERNS

Train a Hebb net to distinguish between the pattern X and the pattern O. The patterns are given below.



Pattern 1 Pattern 2

We can think of it as a classification problem with one output class. Let this class be X. The pattern O is an example of a pattern not in class X.

Convert the patterns into vectors by writing the rows together, and writing # as 1 and . as -1.

```python
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
y = y/100

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)
```

```python
#Variable initialization
epoch=5000                  #Setting training iterations
lr=0.1                      #Setting learning rate
inputlayer_neurons = 2      #number of features in data set
hiddenlayer_neurons = 3     #number of hidden layers neurons
output_neurons = 1          #number of neurons at output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

```
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):

#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)

#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)

#how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad

# dotproduct of nextlayererror and currentlayerop
    wout += hlayer_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hiddenlayer) *lr


print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

```
nput:
[0.66666667 1.        ]
[0.33333333 0.55555556]
[1.         0.66666667]]
ctual Output:
[0.92]
[0.86]
[0.89]]
```

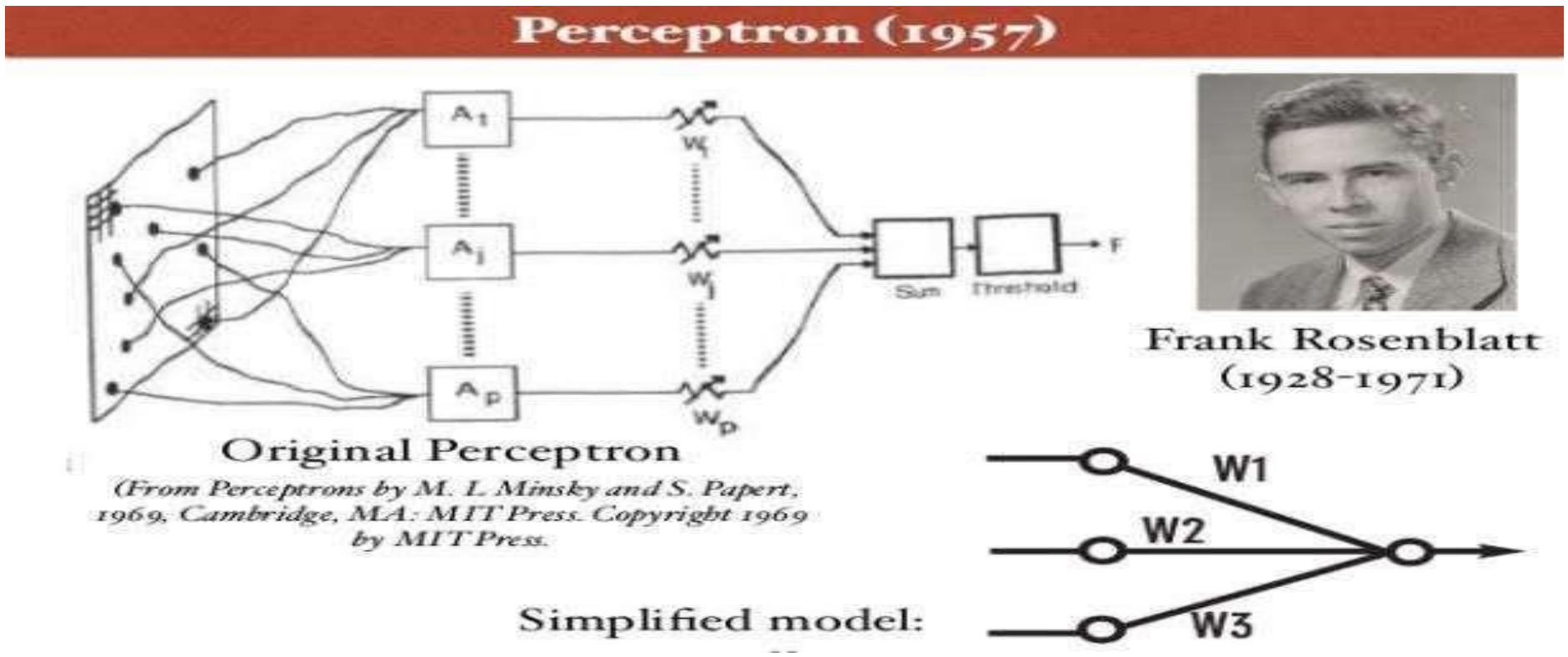
Input:
[[0.6666667 1.]
[0.3333333 0.5555556]
[1. 0.6666667]]
Actual Output:
[[0.92] [0.86] [0.89]]
Predicted Output:
[[0.89571283] [0.88239245] [0.89153673]]

OVERVIEW

- **Perceptron Networks**
  - Theory
  - Perceptron Learning Rule
  - Architecture
  - Flowchart for Training Process
  - Perceptron Training Algorithm for Single Output Classes
  - Perceptron Training Algorithm for Multiple Output Classes
  - Perceptron Network Testing Algorithm

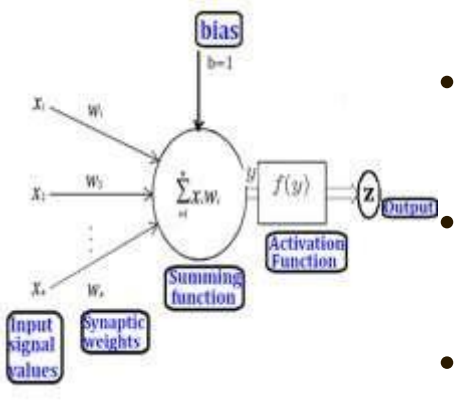
PERCEPTRON NETWORKS

- Frank Rosenblatt [1962] and Minsky and Papert [1988], developed large classes of artificial neural networks called **Perceptron.**



•Perceptron networks come under **single-layer feed-forward networks** and are also called ***simple Perceptrons***.

PERCEPTRON NETWORKS



- A Perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data.
- Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron.
- A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.
- The Perceptron learning rule uses an iterative weight adjustment that is more powerful than the Hebb rule.
- There are two types of Perceptrons: Single layer and Multilayer.
  - Single layer Perceptrons can learn only linearly separable patterns.
  - Multilayer Perceptrons or feed forward neural networks with two or more layers have the greater processing power.
- The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary.
- This enables you to distinguish between the two linearly separable classes +1 and -1.

PERCEPTRON NETWORKS-KEY POINTS

1. The Perceptron network consists of three units, namely, **sensory unit (input unit), associator unit (hidden unit), response unit (output unit).**
2. The sensory units are connected to associator units with fixed weights having values 1, 0 or -l, which are assigned at random.
3. The binary activation function is used in sensory unit and associator unit.
4. The response unit has an activation of l, 0 or -1. The binary step with fixed threshold **θ** *is used as* activation for associator. The output signals that are sent from the associator unit to the response unit are only binary.
5. The output of the Perceptron network is given by, $y = f(y_{in})$. Where $f(y_{in})$ is activation function and is defined as

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

PERCEPTRON NETWORKS-KEY POINTS

6.The Perceptron learning rule is used in the weight updation between the associator unit and the response unit. For each training input, the net will calculate the response and it will determine whether or not an error has occurred.

7.The error calculation is based on the comparison of the values of targets with those of the calculated outputs.

8. The weights on the connections from the units that send the nonzero signal will get adjusted suitably.

9.The weights will be adjusted on the basis of the learning rule. if an error has occurred for a particular training pattern, .i.e..,

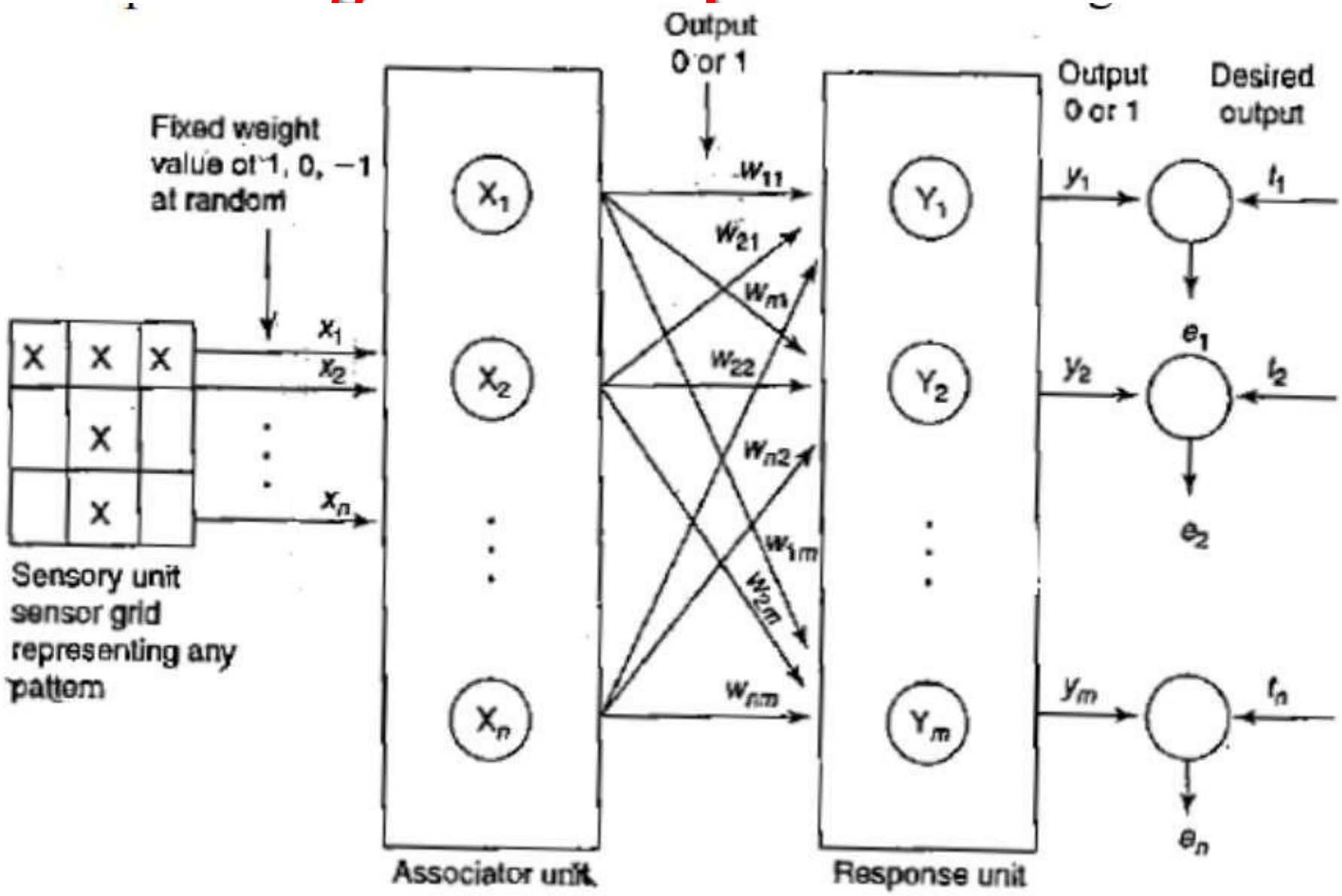
$$w_i(\mathbf{new}) = w_i(\mathbf{old}) + \alpha * x_i * t$$
$$b(\mathbf{new}) = b(\mathbf{old}) + \alpha * t$$

PERCEPTRON NETWORKS-KEY POINTS

- If no error occurs, there is no weight updation and hence the training process may be stopped. In the above equations, the target value *"t" is +1 or -1and* α *is the learning rate.*
- *In general, these learning rules begin with* an initial guess at the weight values and then successive adjustments are made on the basis of the evaluation of an objective function.
- Eventually, the learning rules reach a near-optimal or optimal solution in a finite number of steps.

PERCEPTRON NETWORKS-

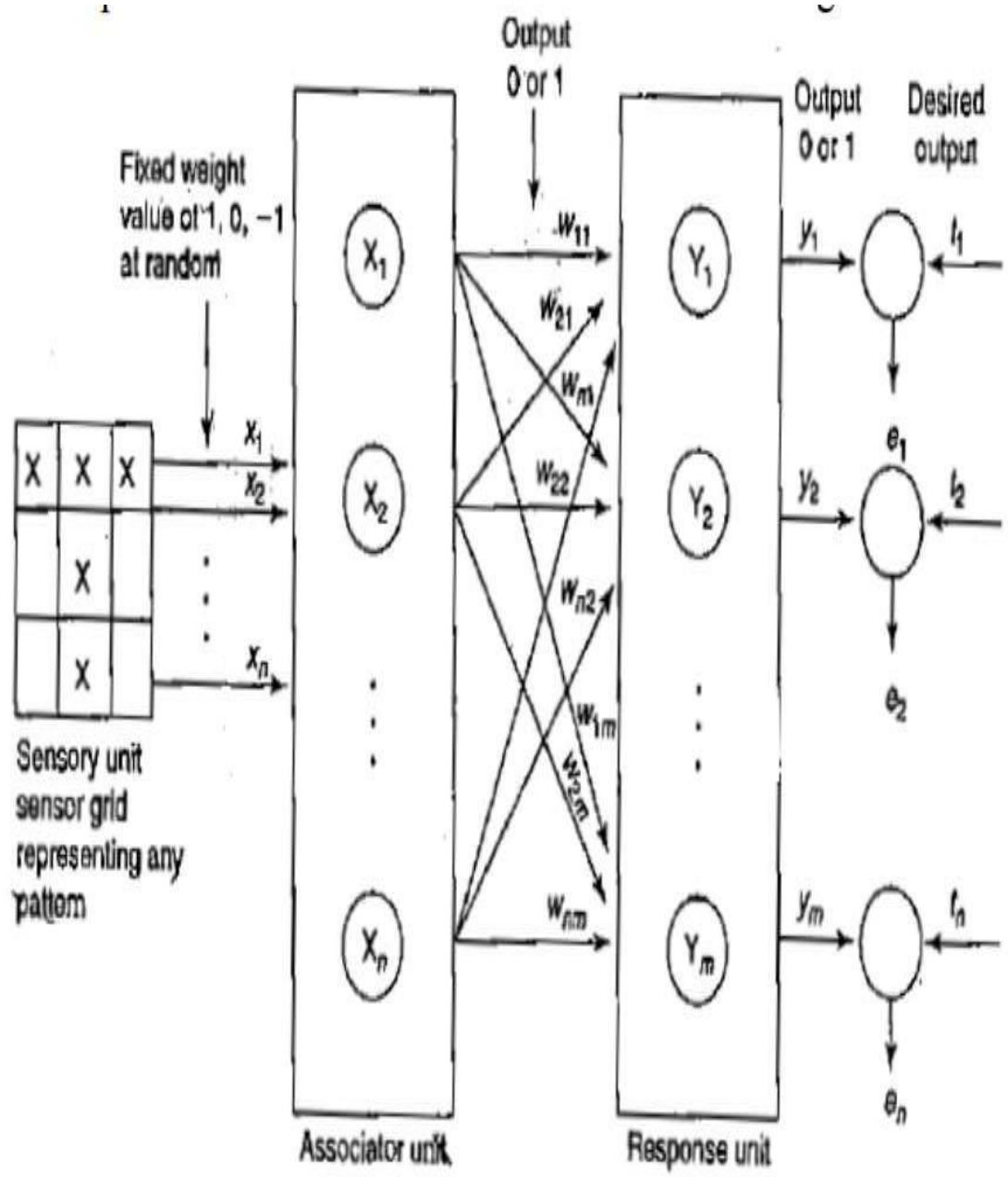
Original Perception Network



PERCEPTRON NETWORKS

- **Sensory unit**

- It is a input layer.
- A sensory unit can be a two-dimensional matrix of photo detectors upon which a lighted picture with geometric black and white pattern impinges.
- These detectors provide a binary(0) electrical signal if the input signal is found to exceed a certain value of threshold.
- Also, these detectors are connected randomly with the associator unit.
- The sensory units are connected to associator units with fixed weights (1, 0 or –l) which are assigned at random.



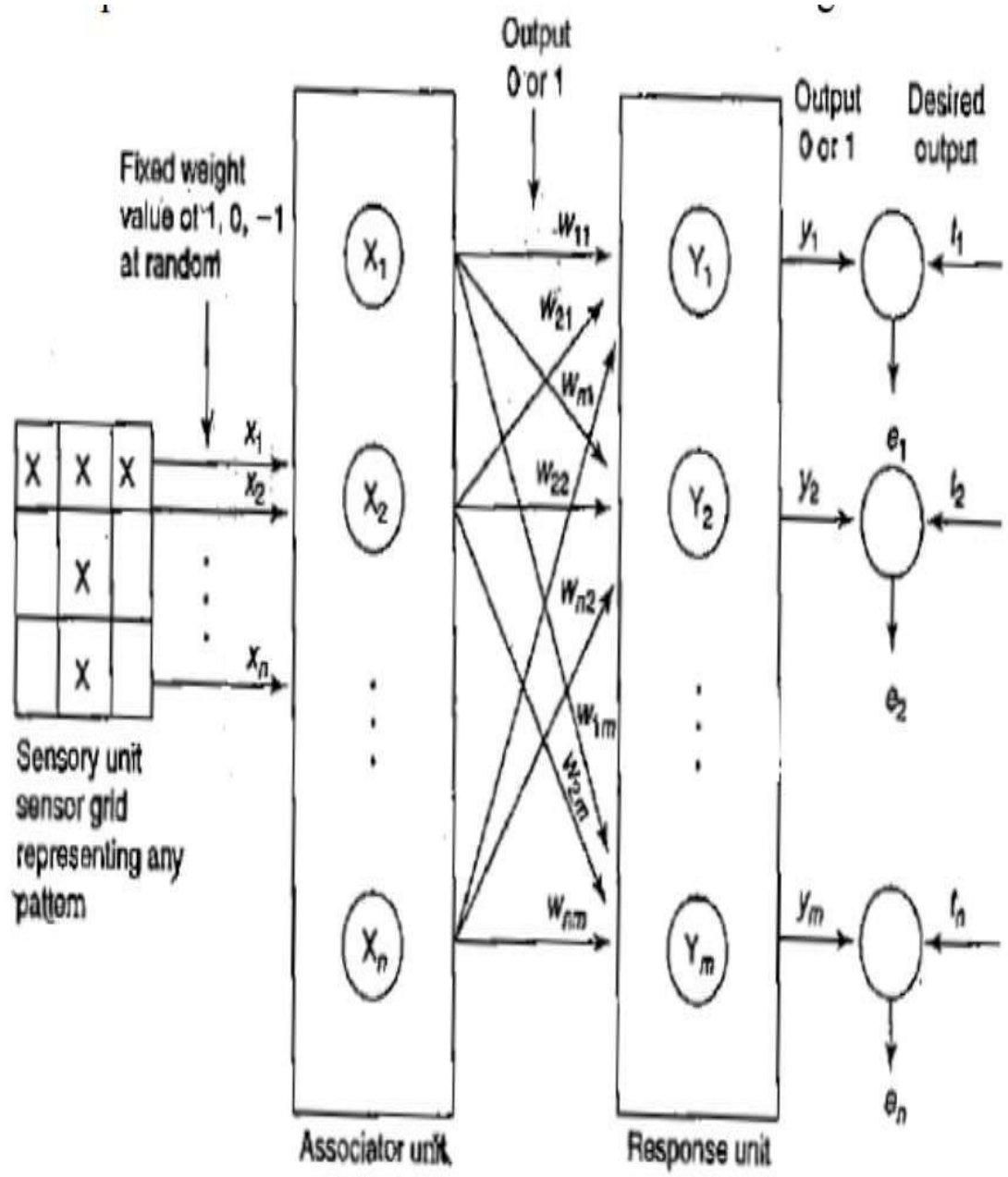
PERCEPTRON NETWORKS

- **Associator unit**

-The associator unit is found to consist of a set of sub circuits called feature predicates.

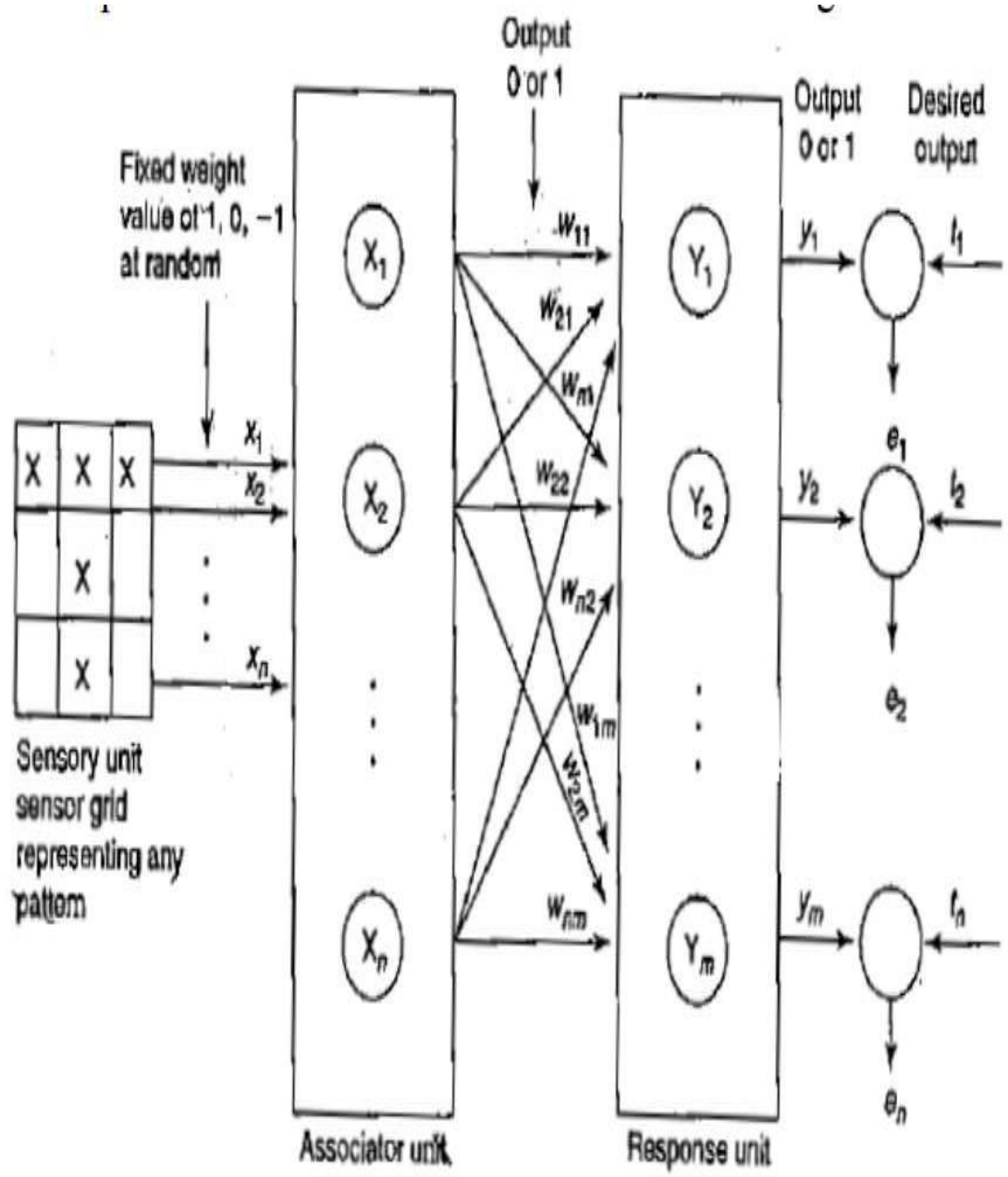
-The feature predicates are hard-wired to detect the specific feature of a pattern and are equivalent to the feature detectors.

-For a particular feature, each predicate is examined with a few or all of the responses of the sensory unit. It can be found that the results from the predicate units are also binary (0 or 1).



PERCEPTRON NETWORKS

- **Response unit**
  - It is the output layer
  - The last unit, i.e. response unit, contains the pattern recognizers or Perceptrons. The weights present in the input layers are all fixed, while the weights on the response unit are trainable.
  - It is used to update the synaptic weights.
  - The output is binary signals.



PERCEPTRON LEARNING RULE

➢In case of the Perceptron learning rule, the learning signal is the difference between the desired and actual response of a neuron.

➢**The Perceptron learning rule is explained as follows:**

- ❑Consider a finite "n" number of input training vectors, with their associated target values **x(n)**and **t(n)**, where "n" ranges from 1 to N.
- ❑The target is either +1 or -1. The output ''y" is obtained on the basis of the net input calculated and activation function being applied over the net input.

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

- ❑The weight updation in case of Perceptron learning is as shown

If $y \neq t$, then

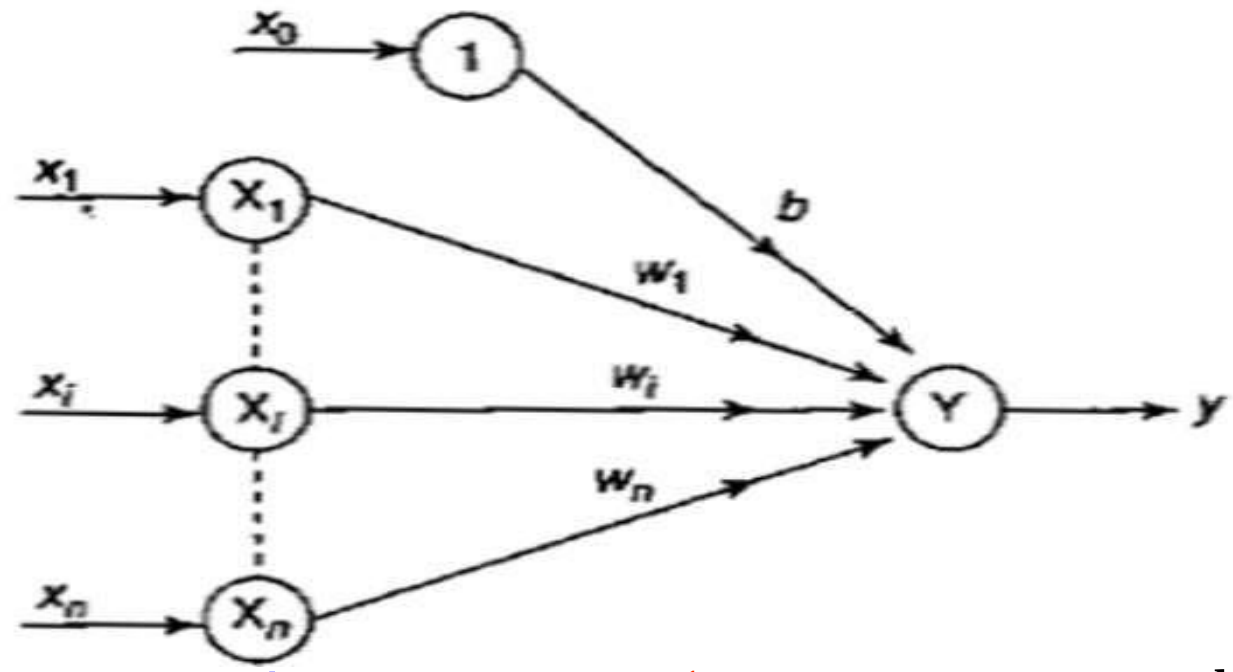
$$w\,(\mathbf{new}) = w\,(\mathbf{old}) + \alpha * x * t$$

else, we have

$$w\,(\mathbf{new}) = w\,(\mathbf{old})$$

PERCEPTRON -ARCHITECTURE

➢A simple Perceptron network architecture is shown in Figure below:

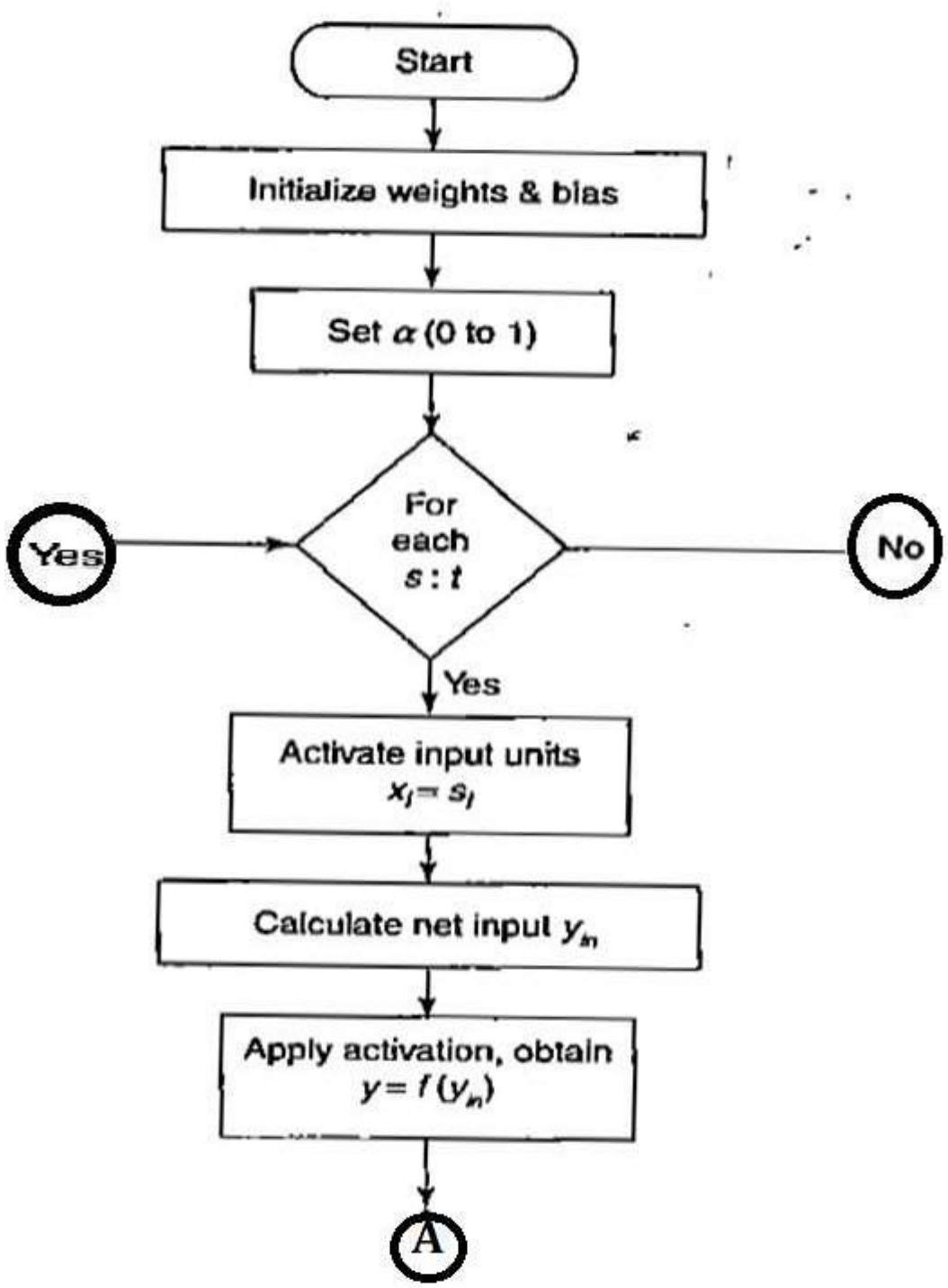


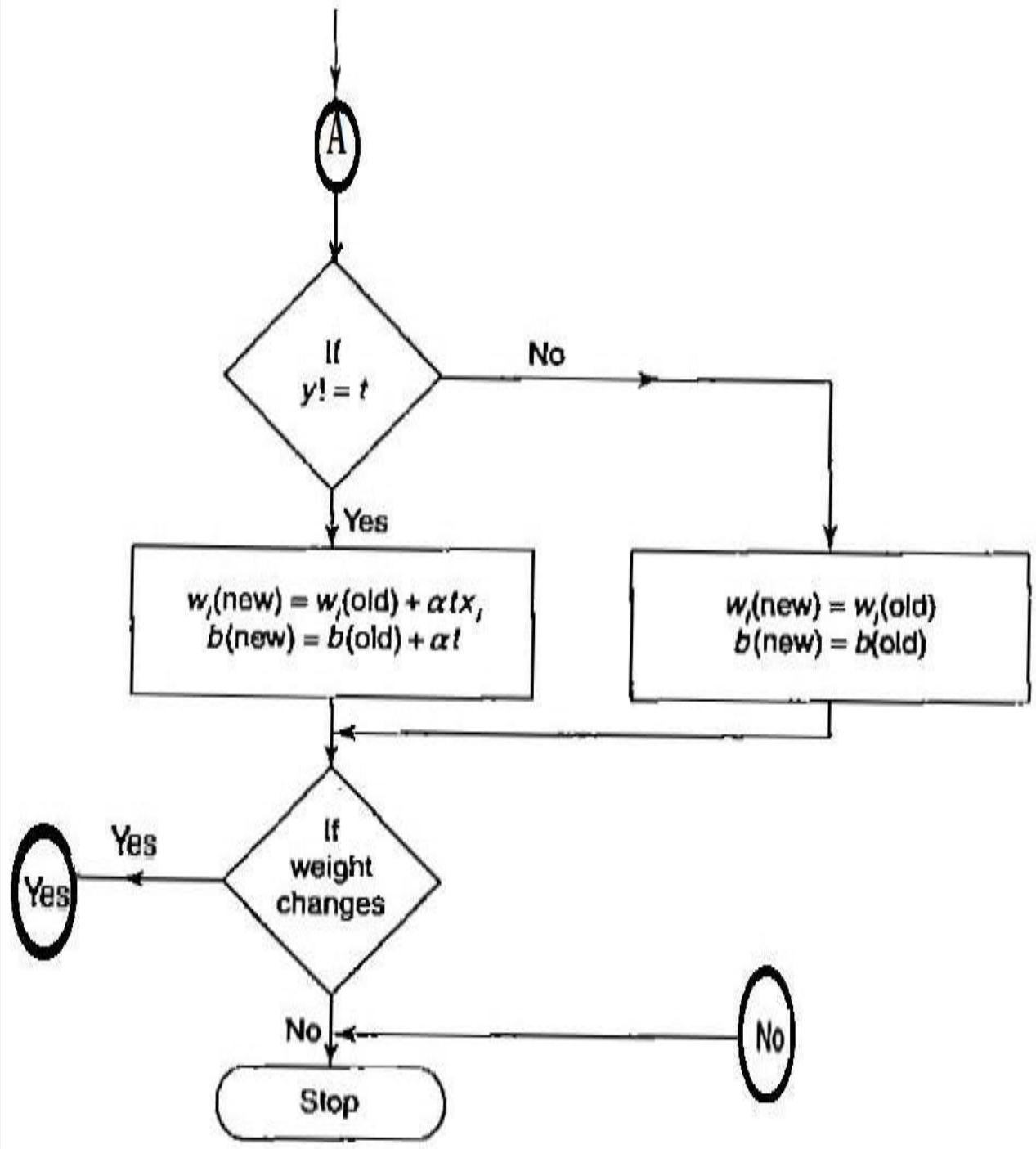
➢In Figure, there are n input neurons, 1 output neuron and a bias.

➢The input-layer and output layer neurons are connected through a directed communication link, which is associated with weights.

➢The goal of the Perceptron net is to classify the input pattern as a member or not a member to a particular class.

PERCEPTRON - FLOWCHART





PERCEPTRON

Training Algorithm For Single Output Classes

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

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by *s:t*.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "*n*" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

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

PERCEPTRON

Training Algorithm For Single Output Classes

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$

$$b(\text{new}) = b(\text{old}) + \alpha t$$

else, we have

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

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

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

PERCEPTRON

Training Algorithm For Multiple Output Classes

Step 0: Initialize the weights, biases and learning rate suitably.

Step 1: Check for stopping condition; if it is false, perform Steps 2–6.

Step 2: Perform Steps 3–5 for each bipolar or binary training vector pair $s:t$.

Step 3: Set activation (identity) of each input unit $i = 1$ to n:

$$x_i := s_i$$

Step 4: Calculate output response of each output unit $j = 1$ to m: First, the net input is calculated as

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

Then activations are applied over the net input to calculate the output response:

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

PERCEPTRON

Training Algorithm For Multiple Output Classes

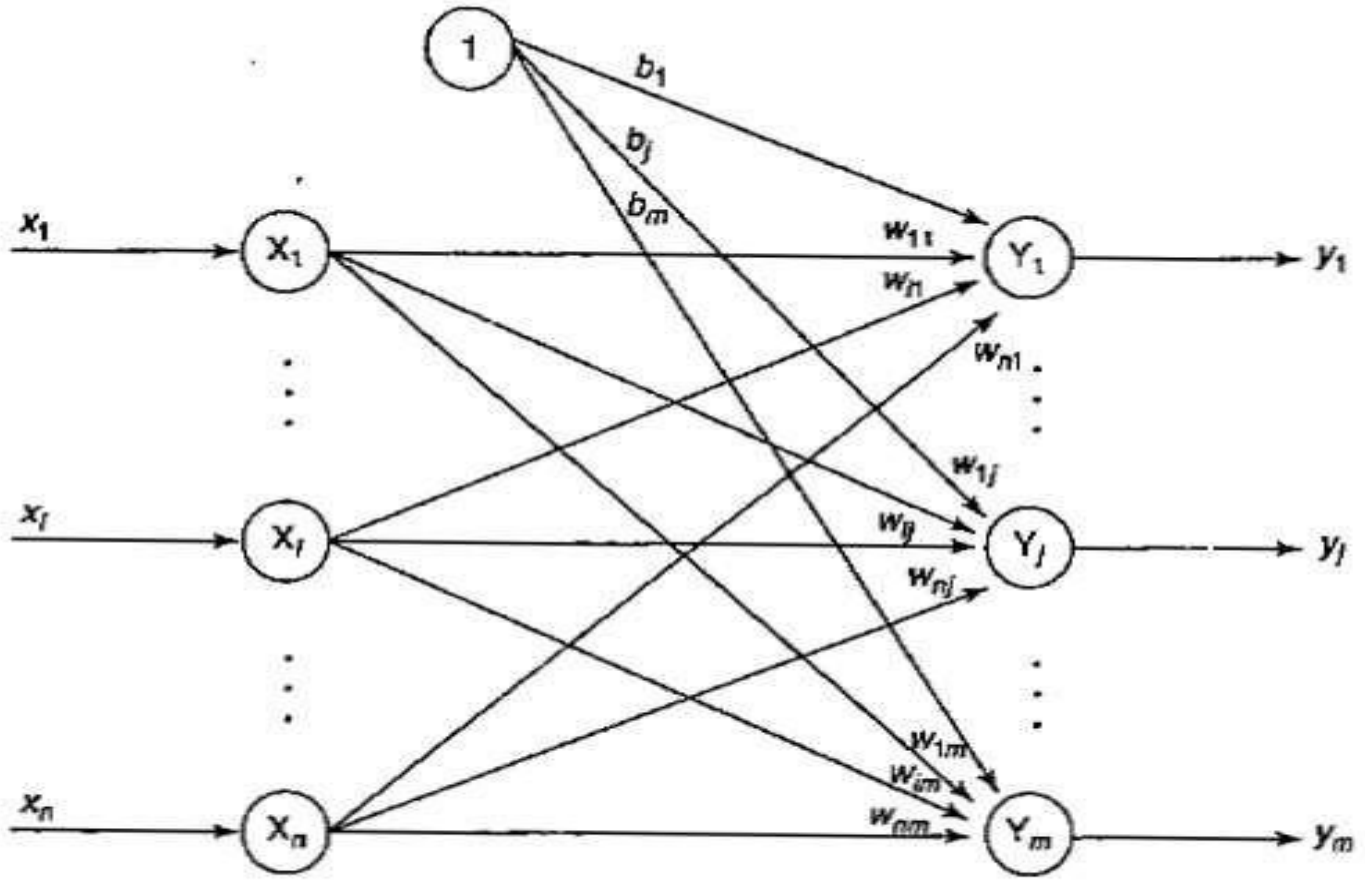


Figure 3-4 Network architecture for perceptron network for several output classes.

PERCEPTRON

Training Algorithm For Multiple Output Classes

Step 5: Make adjustment in weights and bias for $j = 1$ to m and $i = 1$ to n.

If $t_j \neq y_j$, then

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$$

else, we have

$$w_{ij}(\text{new}) = w_{ij}(\text{old})$$
$$b_j(\text{new}) = b_j(\text{old})$$

Step 6: Test for the stopping condition, i.e., if there is no change in weights then stop the training process, else start again from Step 2.

PERCEPTRON NETWORK

TESTING ALGORITHM

The testing algorithm is as follows:

Step 0: The initial weights to be used here are taken from the training algorithms.

Step 1: For each input vector X to be classified, perform Steps 2-3.

Step 2: Set activations of the input unit.

Step 3: Obtain the response of output unit.

$$y_{in} = \sum_{i=1}^{n} (x_i w_i)$$

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

PERCEPTRON NETWORKS-AND

Q. Implement AND function using Perceptron networks for bipolar inputs and targets.

X1	X2	b	Y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

Bipolar inputs

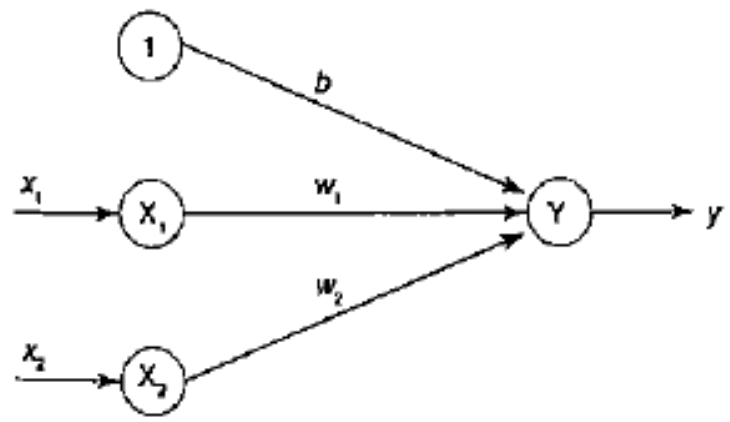


Figure 1 Perceptron network for AND function.

- The Perceptron network, which uses Perceptron learning rule, is used to train the AND function.
- The network architecture is as shown in Figure l.
- The input patterns are presented to the network one by one.
- When all the four input patterns are presented, then one **epoch** is said to be completed.
- The initial weights and threshold are set to zero, i.e., **w1 = w2 = b = 0 and θ = 0.**
- *The learning rate* **α** *is set equal to* **1.(α=1)**

PERCEPTRON NETWORKS-AND

Solution :

- Initialize the weigths and bias= 0 and learning rate to 1.

  w1=w2=b=0 & α=1

X1	X2	b	Y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

PERCEPTRON NETWORKS-AND

For intput pattern 1 1 1 i.e. x1=1, x2=1, t=1

Calculate the Net input

yin= b + w1x1 + w2x2

= 0 + 0*1 + 0*1 = 0

y = f(yin) = 0

Check y = t, No, so weight change is required.

w1(new) = w1(old) + αtx1

= 0 + 1*1*1 = 1

w2(new) = w2(old) + αtx2

= 0 + 1*1*1 = 1

b(new) = b(old) + αt

= 0 + 1*1 = 1

$$f(yin) = \begin{cases} 1 & , y(in)>0 \\ 0 & , y(in)=0 \\ -1 & , y(in)<0 \end{cases}$$

so, w1=w2=b=1

PERCEPTRON NETWORKS-AND

For intput pattern 1 -1 -1 i.e. x1=1, x2=-1, t=-1

Calculate the Net input

yin= b + w1x1 + w2x2

= 1 + 1*1 + 1*-1 = 1

y = f(yin) = 1

Check y = t, No, so weight change is required.

w1(new) = w1(old) + αtx1

= 1 + 1*-1*1 = 0

w2(new) = w2(old) + αtx2

= 1 + 1*-1*-1 = 2

b(new) = b(old) + αt

= 1 + 1*-1 = 0

$$f(yin) = \begin{cases} 1, & y(in)>0 \\ 0, & y(in)=0 \\ -1, & y(in)<0 \end{cases}$$

so, w1=0, w2=2, b=0

PERCEPTRON NETWORKS-AND

For intput pattern -1 1 -1 i.e. x1=-1, x2=1, t=-1

Calculate the Net input

yin= b + w1x1 + w2x2

= 1 + 0*-1 + 2*1 = 2

y = f(yin) = 1

Check y = t, No, so weight change is required.

w1(new) = w1(old) + αtx1

= 0 + 1*-1*-1 = 1

w2(new) = w2(old) + αtx2

= 2 + 1*-1*1 = 1

b(new) = b(old) + αt

= 0 + 1*-1 = -1

$$f(yin) = \begin{cases} 1, & y(in)>0 \\ 0, & y(in)=0 \\ -1, & y(in)<0 \end{cases}$$

PERCEPTRON NETWORKS-AND

For intput pattern -1 -1 -1 i.e. x1=-1, x2=-1, t=-1

Calculate the Net input

yin= b + w1x1 + w2x2

= -1 + 1*-1 + 1*-1 = -3

y = f(yin) = -1

Check y = t, YES, so weight change is not required.

Final weights : w1=1, w2=1, b=-1

PERCEPTRON NETWORKS-AND

$$f(yin) = \begin{cases} 1 & y(in)>0 \\ 0 & y(in)=0 \\ -1 & y(in)<0 \end{cases}$$

Perceptron - AND					
Input (1,X1,X2)	**Net**	**Out f(Yin)**	**Target**	**Update Weight**	**Weights (b,W1,W2)**
Epoch- 1					**(0 , 0 , 0)**
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(1*0)+(1*0)=0	0	1	(o!=t),True, hence change weights b(new)=b(old)+α*t=0+(1*1)=1 w1(new)=w1(old)+α*t*x1=0+(1*1*1)=1 w2(new)=w2(old)+α*t*x2=0+(1*1*1)=1	(1 , 1 , 1)
(1 , 1 , -1)	Yin=b+x1w1+x2w2 Yin=1+(1*1)+(-1*1)=1	1	-1	(o!=t), True, hence change weights b(new)=b(old)+α*t=1+(1*-1)=2 w1(new)=w1(old)+α*t*x1=1+(1*-1*1)=0 w2(new)=w2(old)+α*t*x2=1+(1*-1*-1)=0	(0 , 0 , 2)
(1 , -1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(-1*0)+(1*2)=2	1	-1	(o!=t), True, hence change weights b(new)=b(old)+α*t=0+(1*-1)=-1 w1(new)=w1(old)+α*t*x1=0+(1*-1*-1)=1 w2(new)=w2(old)+α*t*x2=2+(1*-1*1)=1	(-1 , 1 , 1)
(1 , -1 , -1)	Yin=b+x1w1+x2w2 Yin=-1+(-1*1)+(-1*1)=-3	-1	-1	(o!=t), False, hence no change in weights	(-1 , 1 , 1)
Epoch- 2					**(-1 , 1 , 1)**
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=-1+(1*1)+(1*1)=1	1	1	(o!=t), False, hence no change in weights	(-1 , 1 , 1)
(1 , 1 , -1)	Yin=b+x1w1+x2w2 Yin=-1+(1*1)+(-1*1)=-1	-1	-1	(o!=t), False, hence no change in weights	(-1 , 1, 1)
(1 , -1 , 1)	Yin=b+x1w1+x2w2 Yin=-1+(-1*1)+(1*1)=-1	-1	-1	(o!=t), False, hence no change in weights	(-1 , 1 , 1)
(1 , -1 , -1)	Yin=b+x1w1+x2w2 Yin=-1+(-1*1)+(-1*1)=-1	-1	-1	(o!=t), False, hence no change in weights	(-1 , 1 , 1)

PERCEPTRON NETWORKS-OR

Q. Implement OR function with binary inputs and ·bipolar target using Perceptron training algorithm upto 3 epochs.

Solution: The truth table for OR function with binary inputs and bipolar targets is shown in Table 3.

Table 3

x_1	x_2	t
1	1	1
1	0	1
0	1	1
0	0	−1

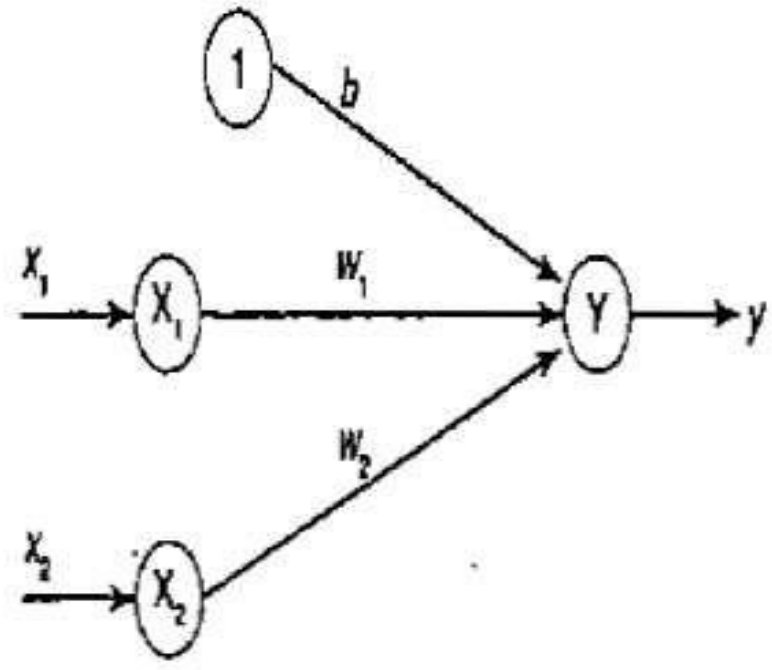


Figure 3 Perceptron network for OR function.

PERCEPTRON NETWORKS-OR

Solution :

- Initialize the weigths and bias= 0 and learning rate to 1.

  w1=w2=b=0 & α=1

Applying the activation function over the net input,

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

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

Perceptron - OR

Input (1,X1,X2) (Binary)	Net	Out	Target Bipolar	Update Weight α=1	Weights (b,W1,W2)
Epoch- 1					**(0 , 0 , 0)**
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(1*0)+(1*0)=0	0	1	(o!=t),True, hence change weights b(new)=b(old)+α*t=0+(1*1)=1 w1(new)=w1(old)+α*t*x1=0+(1*1*1)=1 w2(new)=w2(old)+α*t*x2=0+(1*1*1)=1	(1 , 1 , 1)
(1 , 1 , 0)	Yin=b+x1w1+x2w2 Yin=1+(1*1)+(0*1)=2	1	1	(o!=t), False, hence no change in weights	(1 , 1 , 1)
(1 , 0 , 1)	Yin=b+x1w1+x2w2 Yin=1+(0*1)+(1*1)=2	1	1	(o!=t), False, hence no change in weights	(1 , 1 , 1)
(1 , 0 , 0)	Yin=b+x1w1+x2w2 Yin=1+(0*1)+(0*1)=1	1	-1	(o!=t),True, hence change weights b(new)=b(old)+α*t=1+(1*-1)=0 w1(new)=w1(old)+α*t*x1=1+(1*-1*0)=1 w2(new)=w2(old)+α*t*x2=1+(1*-1*0)=1	(0 , 1 , 1)

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

Perceptron - OR					
Input (1,X1,X2) (Binary)	Net	Out	Target Bipolar	Update Weight α=1	Weights (b,W1,W2)
Epoch- 2					(0 , 1 , 1)
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(1*1)+(1*1)=2	1	1	(o!=t), False, hence no change in weights	(0 , 1 , 1)
(1 , 1 , 0)	Yin=b+x1w1+x2w2 Yin=0+(1*1)+(0*1)=1	1	1	(o!=t), False, hence no change in weights	(0 , 1 , 1)
(1 , 0 , 1)	Yin=b+x1w1+x2w2 Yin=0+(0*1)+(1*1)=1	1	1	(o!=t), False, hence no change in weights	(0 , 1 , 1)
(1 , 0 , 0)	Yin=b+x1w1+x2w2 Yin=0+(0*1)+(0*1)=0	0	-1	(o!=t),True, hence change weights b(new)=b(old)+α*t=0+(1*-1)=-1 w1(new)=w1(old)+α*t*x1=1+(1*-1*0)=1 w2(new)=w2(old)+α*t*x2=1+(1*-1*0)=1	(-1 , 1 , 1)

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

Perceptron - OR

Input (1,X1,X2) (Binary)	Net	Out	Target Bipolar	Update Weight α=1	Weights (b,W1,W2)
Epoch- 3					(-1 , 1 , 1)
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=-1+(1*1)+(1*1)=1	1	1	(o!=t), False, hence no change in weights	(-1 , 1 , 1)
(1 , 1 , 0)	Yin=b+x1w1+x2w2 Yin=-1+(1*1)+(0*1)=0	0	1	(o!=t),True, hence change weights b(new)=b(old)+α*t=-1+(1*1)=0 w1(new)=w1(old)+α*t*x1=1+(1*1*1)=2 w2(new)=w2(old)+α*t*x2=1+(1*1*0)=1	(0 , 2, 1)
(1 , 0 , 1)	Yin=b+x1w1+x2w2 Yin=0+(0*2)+(1*1)=1	1	1	(o!=t), False, hence no change in weights	(0 , 2, 1)
(1 , 0 , 0)	Yin=b+x1w1+x2w2 Yin=0+(0*2)+(0*1)=0	0	-1	(o!=t),True, hence change weights b(new)=b(old)+α*t=0+(1*-1)=-1 w1(new)=w1(old)+α*t*x1=2+(1*-1*0)=2 w2(new)=w2(old)+α*t*x2=1+(1*-1*0)=1	(-1 , 2 , 1)

- The final weights at the end of third epoch are w1=2,w2=1,b=-1
- Further epochs have to be done for the convergence of the network.

PERCEPTRON NETWORKS- AND NOT

Q. Find the weights using Perceptron network for AND NOT function when all the inputs are presented only one time. Use bipolar inputs and targets.

Solution: The truth table for ANDNOT function is shown in Table 5.

Table 5

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

The network architecture of ANDNOT function is shown as in Figure 4.

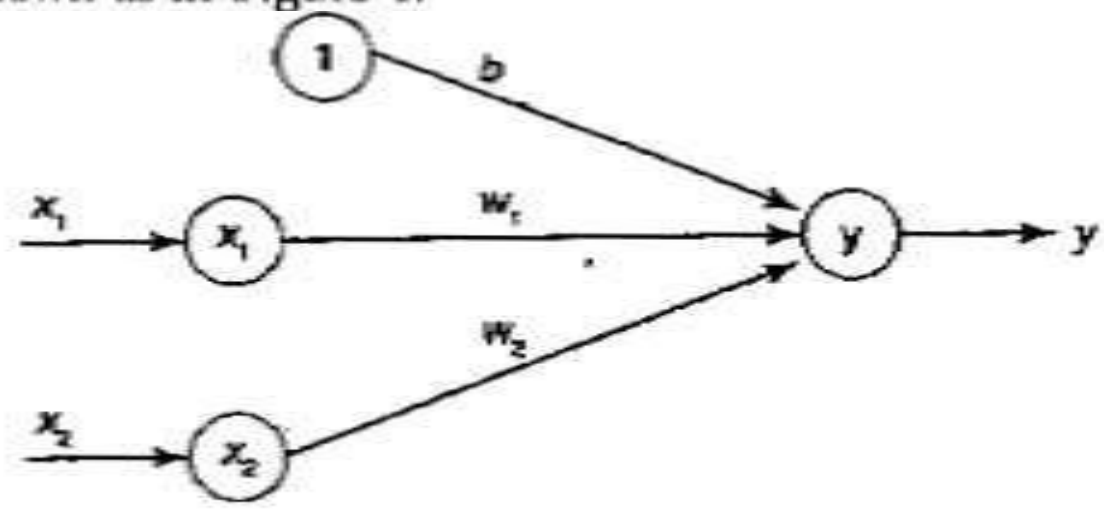


Figure 4 Network for ANDNOT function.

- The initial weights and threshold are set to zero, i.e.,

  w1 = w2 = b = 0 and Θ= 0.

- *The learning rate* **α** *is set equal to* **1.(α=1)**

PERCEPTRON NETWORKS- AND NOT

- The initial weights and threshold are set to zero, i.e.,

  w1 = w2 = b = 0 and θ= 0.

- *The learning rate* **α** *is set equal to* **1.(α=1)**

Solution: The truth table for ANDNOT function is shown in Table 5.

Table 5

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

Applying the activation function over the net input,

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

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

Perceptron – AND NOT

Input (1,X1,X2)	Net	Out	Target	Update Weight α=1	Weights (b,W1,W2)
Epoch- 1					(0 , 0 , 0)
(1 , 1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(1*0)+(1*0)=0	0	-1	(o!=t),True, hence change weights b(new)=b(old)+α*t=0+(1*-1)=-1 w1(new)=w1(old)+α*t*x1=0+(1*-1*1)=-1 w2(new)=w2(old)+α*t*x2=0+(1*-1*1)=-1	(-1 , -1 , -1)
(1 , 1 , -1)	Yin=b+x1w1+x2w2 Yin=-1+(1*-1)+(-1*-1) =-1	-1	1	(o!=t), True, hence change weights b(new)=b(old)+α*t=-1+(1*1)=0 w1(new)=w1(old)+α*t*x1=-1+(1*1*1)=0 w2(new)=w2(old)+α*t*x2=-1+(1*1*-1)=-2	(0 , 0 , -2)
(1 , -1 , 1)	Yin=b+x1w1+x2w2 Yin=0+(-1*0)+(1*-2)=-2	-1	-1	(o!=t), False, hence no change in weights	(0 , 0 , -2)
(1 ,-1 ,-1)	Yin=b+x1w1+x2w2 Yin=0+(-1*0)+(-1*-2)=2	1	-1	(o!=t), True, hence change weights b(new)=b(old)+α*t=0+(1*-1)=-1 w1(new)=w1(old)+α*t*x1=0+(1*-1*-1)=1 w2(new)=w2(old)+α*t*x2=-2+(1*-1*-1)=-1	(-1 , 1 , -1)

- One epoch of training for AND NOT function using Perceptron network is tabulated.
- The final weights at the end of FIRST epoch are w1=1,w2=-1,b=-1

PERCEPTRON NETWORKS-CLASSIFICATION

Q. Find the weights required to perform the following classification using Perceptron network. The *vectors (1,1, 1, 1) and (-1, 1 -1, -1) are belonging* to the class (so have target value 1), vectors (1, 1, 1, -1) and (1, -1, -1, 1) are not belonging to the class (so have target value -1). Assume learning rate as 1 and initial weights as 0.

Solution: The truth table for the given vectors is given in Table 7

Table 7

Input					
x_1	x_2	x_3	x_4	b	Target (t)
1	1	1	1	1	1
−1	1	−1	−1	1	1
1	1	1	−1	1	−1
1	−1	−1	1	1	−1

PERCEPTRON NETWORKS-CLASSIFICATION

Table 7

Input					
x_1	x_2	x_3	x_4	b	Target (t)
1	1	1	1	1	1
−1	1	−1	−1	1	1
1	1	1	−1	1	−1
1	−1	−1	1	1	−1

Let $w_1 = w_2 = w_3 = w_4 = b = 0$ and the learning rate $\alpha = 1$. Since the threshold $\theta = 0.2$, so the activation function is

$$y = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \\ -1 & \text{if } y_{in} < -0.2 \end{cases}$$

The net input is given by

$$y_{in} = b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$$

Weight changes is done using the equation:

$$\mathbf{b(new)=b(old)+\alpha*t}$$

$$\mathbf{w_i(new)=w_i(old)+\alpha*t*xi}$$

The training is performed and the weights are tabulated in Table 8.

Perceptron for given vector

x1	x2	x3	x4	b	t	yin	y	Change/ No Change	w1	w2	w3	w4	b
Epoch-1									0	0	0	0	0
1	1	1	1	1	1	0	0	change	1	1	1	1	1
-1	1	-1	-1	1	1	-1	-1	change	0	2	0	0	2
1	1	1	-1	1	-1	4	1	change	-1	1	-1	1	1
1	-1	-1	1	1	-1	1	1	change	-2	2	0	0	0
Epoch-2									-2	2	0	0	0
1	1	1	1	1	1	0	0	change	-1	3	1	1	1
-1	1	-1	-1	1	1	3	1	No change	-1	3	1	1	1
1	1	1	-1	1	-1	3	1	change	-2	2	0	2	0
1	-1	-1	1	1	-1	-2	-1	No change	-2	2	0	2	0
Epoch-3									-2	2	0	2	0
1	1	1	1	1	1	2	1	No change	-2	2	0	2	0
-1	1	-1	-1	1	1	2	1	No change	-2	2	0	2	0
1	1	1	-1	1	-1	-2	-1	No change	-2	2	0	2	0
1	-1	-1	1	1	-1	-2	-1	No change	-2	2	0	2	0

PERCEPTRON NETWORKS-CLASSIFICATION

- Thus, in the third epoch, all the calculated outputs become equal to targets and the network has converged.
- The network architecture is shown in Figure 5.

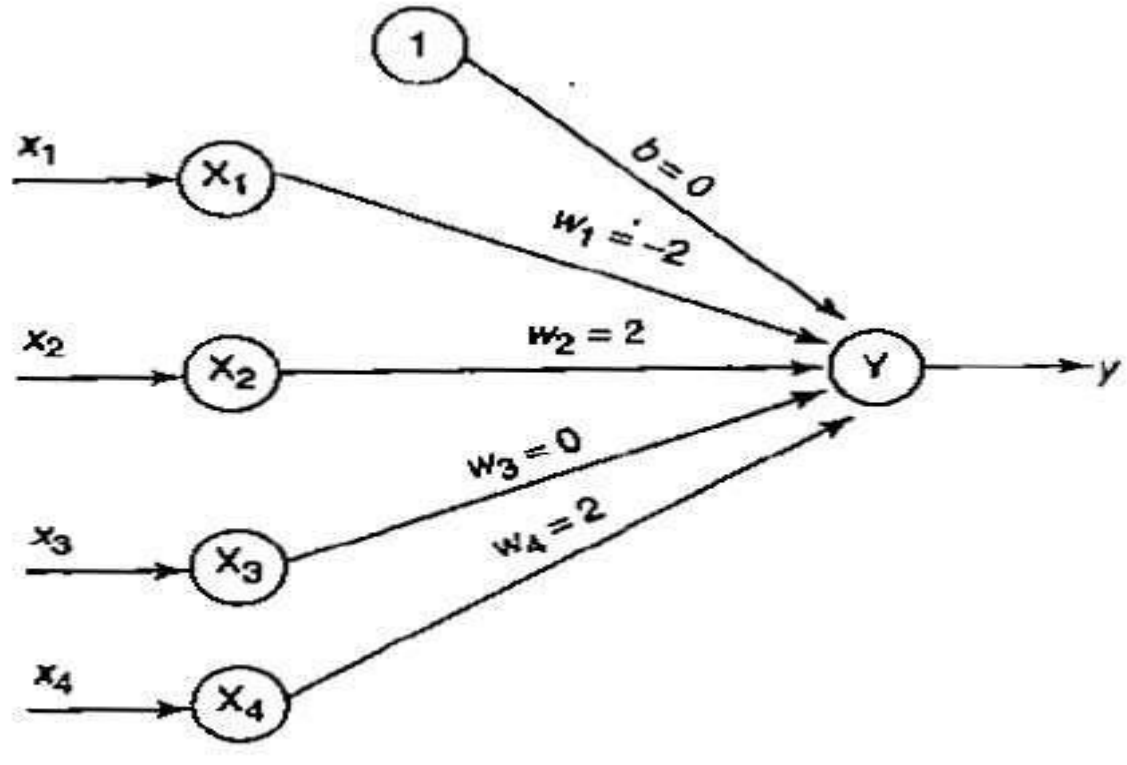


Figure 5 Network architecture.

PERCEPTRON NETWORKS-CLASSIFICATION

Q. Classify the two-dimensional input pattern shown *in Figure 6 using Perceptron network. The symbol* "*" indicates the data representation to be + 1 and "•" indicates data to be -1. The patterns are I-F. For pattern I, the target is + 1, and for F, the target is -1.

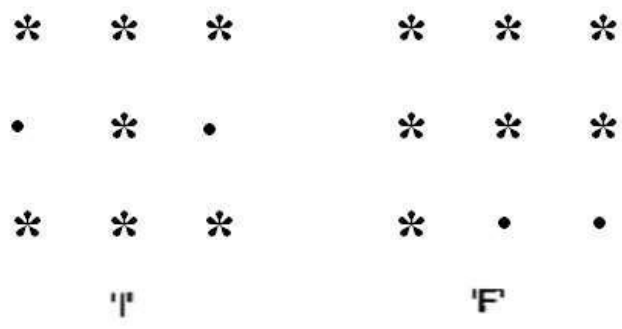


Figure 6 I-F data representation.

Solution: The training patterns for this problem are tabulared in Table 9.

Table 9

	Input										
Pattern	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	1	Target (t)
I	1	1	1	−1	1	−1	1	1	1	1	1
F	1	1	1	1	1	1	1	−1	−1	1	−1

PERCEPTRON NETWORKS-CLASSIFICATION

Let $w_1 = w_2 = w_3 = w_4 = b = 0$ and the learning rate $\alpha = 1$. Since the threshold $\theta = 0$, so the activation function is

	Input										
Pattern	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	1	Target (t)
I	1	1	1	−1	1	−1	1	1	1	1	1
F	1	1	1	1	1	1	1	−1	−1	1	−1

$$y = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

$$y_{in} = b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$$

Weight changes is done using the equation:

b(new)=b(old)+α*t

w_i(new)=w_i(old)+α*t*xi

PERCEPTRON NETWORKS-CLASSIFICATION

The training is performed and the weights are tabulated in Table

x1	x2	x3	x4	x5	x6	x7	x8	x9	b	t	yin	y	Change/ No Change	w1	w2	w3	w4	w5	w6	w7	w8	w9	b
														0	0	0	0	0	0	0	0	0	0
1	1	1	-1	1	-1	1	1	1	1	1	0	0	Change	1	1	1	-1	1	-1	1	1	1	1
1	1	1	1	1	1	1	-1	-1	1	-1	2	1	Change	0	0	0	-2	0	-2	0	2	2	0

The final weights are : w = [0 0 0 - 2 0 -2 0 2 2 0]

The network can be further trained for its convergence.

PERCEPTRON NETWORKS-CLASSIFICATION

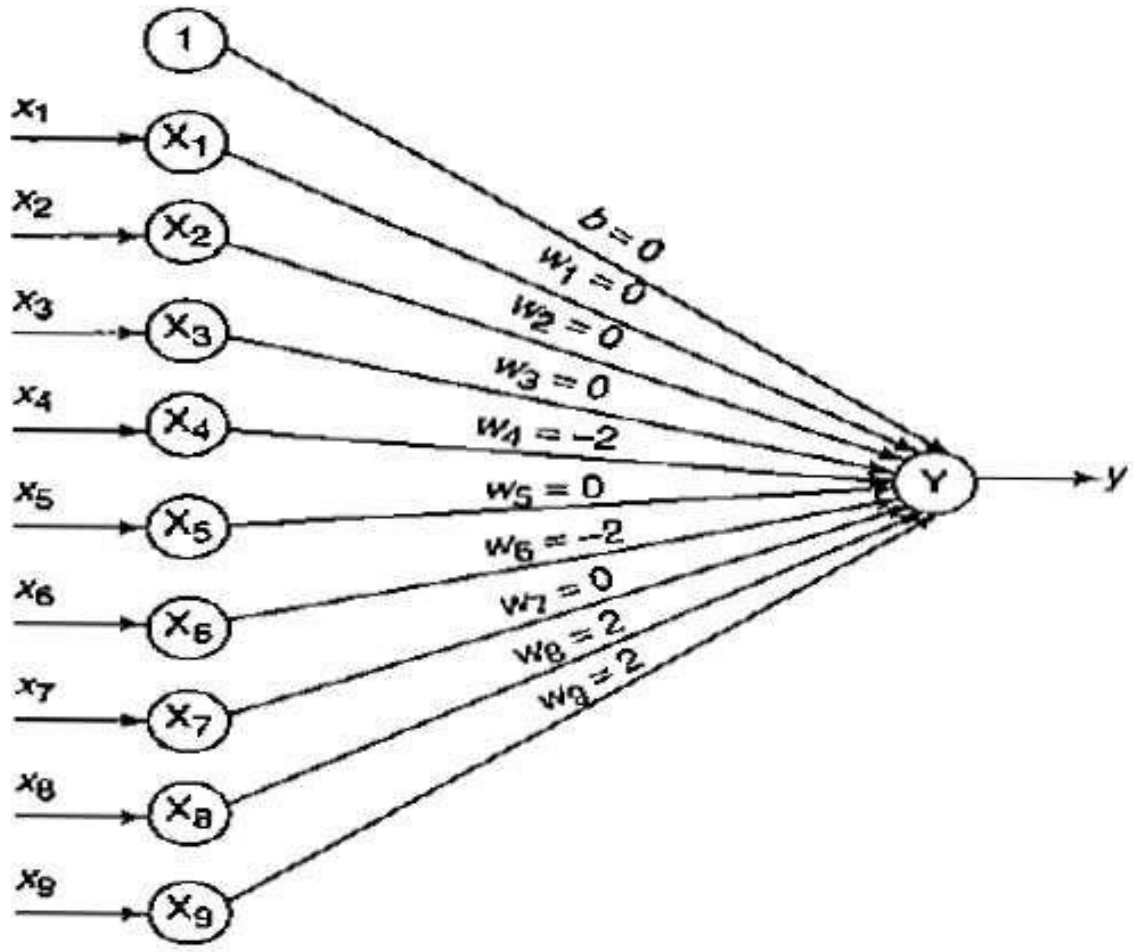


Figure 7 Network architecture.

PERCEPTRON NETWORKS-CLASSIFICATION

Q. Design a Perceptron classifier for identifying the class of given fruit based on its weight and lengths. Also identify the class of fruit with **weight=140gms and length=19.7 cms**

	Weights(gms)	Length(cms)
Fruit 1 (class 1→1)	**121**	**16.8**
	114	**15.2**
Fruit 2 (class 2→-1)	**210**	**9.4**
	195	**8.1**

PERCEPTRON NETWORKS-CLASSIFICATION

	Weights(gms) X1	Length(cms) x2	target
Fruit 1 (class 1)	121	16.8	1
	114	15.2	1
Fruit 2 (class 2)	210	9.4	-1
	195	8.1	-1

Let $w_1 = w_2 = b = 0$ and the learning rate $\alpha = 1$.
Since the threshold $\theta = 0$, so the activation function is

$$y = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

$$y_{in} = b + x_1 w_1 + x_2 w_2$$

Weight changes is done using the equation:

b(new)=b(old)+α*t

w_i(new)=w_i(old)+α*t*xi

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-1							**0**	**0**	**0**
121	16.8	1	1	0	1	No Change	**0**	**0**	**0**
114	15.2	1	1	0	1	No Change	**0**	**0**	**0**
210	9.4	1	-1	0	1	Change	-210	-9.4	-1
195	8.1	1	-1	-4103.14	-1	No Change	-210	-9.4	-1

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-2							-210	-9.4	-1
121	16.8	1	1	-25568.92	-1	Change	-89	7.4	0
114	15.2	1	1	-10033.52	-1	Change	25	22.6	1
210	9.4	1	-1	5463.44	1	Change	-185	13.2	0
195	8.1	1	-1	-35968.08	-1	No Change	-185	13.2	0

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-3							-185	13.2	0
121	16.8	1	1	-22163.24	-1	Change	-64	30	1
114	15.2	1	1	-6839	-1	Change	50	45.2	2
210	9.4	1	-1	10926.88	1	Change	-160	35.8	1
195	8.1	1	-1	-30909.02	-1	No Change	-160	35.8	1

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-4							-160	35.8	1
121	16.8	1	1	-18757.56	-1	Change	-39	52.6	2
114	15.2	1	1	-3644.48	-1	Change	75	67.8	3
210	9.4	1	-1	16390.32	1	Change	-135	58.4	2
195	8.1	1	-1	-25849.96	-1	No Change	-135	58.4	2

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-5							-135	58.4	2
121	16.8	1	1	-15351.88	-1	Change	-14	75.2	3
114	15.2	1	1	-449.96	-1	Change	100	90.4	4
210	9.4	1	-1	21853.76	1	Change	-110	81	3
195	8.1	1	-1	-20790.9	-1	No Change	-110	81	3

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-6							-110	81	3
121	16.8	1	1	-11946.2	-1	Change	11	97.8	4
114	15.2	1	1	2744.56	1	No Change	11	97.8	4
210	9.4	1	-1	3233.32	1	Change	-199	88.4	3
195	8.1	1	-1	-38085.96	-1	No Change	-199	88.4	3

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-7							-199	88.4	3
121	16.8	1	1	-22590.88	-1	Change	-78	105	4
114	15.2	1	1	-7288.96	-1	Change	36	120	5
210	9.4	1	-1	8696.76	1	Change	-174	111	4
195	8.1	1	-1	-33026.9	-1	No Change	-174	111	4

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-8							-174	111	4
121	16.8	1	1	-19185.2	-1	Change	-53	128	5
114	15.2	1	1	-4094.44	-1	Change	61	143	6
210	9.4	1	-1	14160.2	1	Change	-149	134	5
195	8.1	1	-1	-27967.84	-1	No Change	-149	134	5

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-9							-149	134	5
121	16.8	1	1	-15779.52	-1	Change	-28	150	6
114	15.2	1	1	-899.92	-1	Change	86	166	7
210	9.4	1	-1	19623.64	1	Change	-124	156	6
195	8.1	1	-1	-22908.78	-1	No Change	-124	156	6

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-10							-149	134	5
121	16.8	1	1	-15772.8	-1	Change	-89	142	5.5
114	15.2	1	1	-7919.02	-1	Change	-32	150	6
210	9.4	1	-1	-5199	-1	No Change	-32	150	6
195	8.1	1	-1	-4921.5	-1	No Change	-32	150	6

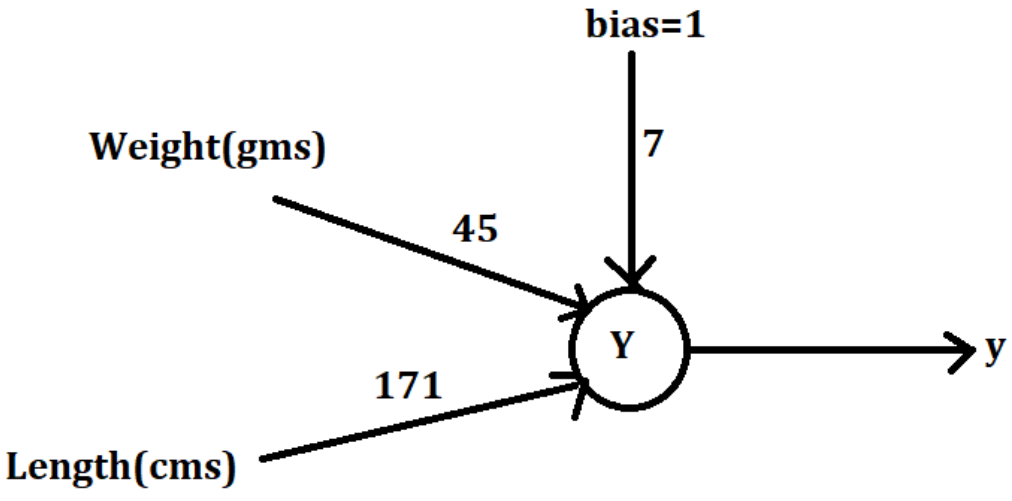
x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-11							-32	150	6
121	16.8	1	1	-1285.5	-1	Change	29	158	6.5
114	15.2	1	1	5720.18	1	No Change	29	158	6.5
210	9.4	1	-1	7585.46	1	Change	-76	154	6
195	8.1	1	-1	-13569.03	-1	No Change	-76	154	6

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-12							-76	154	6
121	16.8	1	1	-6607.84	-1	Change	-16	162	6.5
114	15.2	1	1	703.42	1	No Change	-16	162	6.5
210	9.4	1	-1	-1724.76	-1	No Change	-16	162	6.5
195	8.1	1	-1	-1702.99	-1	No Change	-16	162	6.5

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-13							-16	162	6.5
121	16.8	1	1	854.28	-1	Change	45	171	7
114	15.2	1	1	7728.6	1	No Change	45	171	7
210	9.4	1	-1	11059.7	-1	No Change	45	171	7
195	8.1	1	-1	10163.05	-1	No Change	45	171	7

x1	x2	b	t	yin	y	Change/ No Change	w1	w2	b
Epoch-14							45	171	7
121	16.8	1	1	8316.4	1	No Change	45	171	7
114	15.2	1	1	7728.6	1	No Change	45	171	7
210	9.4	1	-1	11059.7	-1	No Change	45	171	7
195	8.1	1	-1	10163.05	-1	No Change	45	171	7

- **The convergence is after 14 Epochs and the final weights are : w1=45,w2=171,b=7**
- **The network architecture is shown in Figure**



- Now, identify fruit with weight=140gms and length=19.7 cms
- **Yin=140*45+19.7*171+7=9665.85**
- **f(Yin)=Yin>0=1.**
- **Therefore, it belongs to class 1**

OVERVIEW

- **Back-propagation Network**
  - Theory
  - Architecture
  - Flowchart for Training Process
  - Training Algorithm
  - Learning Factors of Back-Propagation Network
    - Initial Weights
    - Learning Rate α
    - Momentum Factor
    - Generalization
    - Number of Training Data
    - Number of Hidden Layer Nodes
  - Testing Algorithm of Back-Propagation Network

BACK-PROPAGATION NETWORK

- A single-layer neural network has many restrictions. This network can accomplish very limited classes of tasks.

- Minsky and Papert (1969) showed that a two layer feed-forward network can overcome many restrictions, but they did not present a solution to the problem as "how to adjust the weights from input to hidden layer" ?

- An answer to this question was presented by Rumelhart, Hinton and Williams in 1986. The central idea behind this solution is that the errors for the units of the hidden layer are determined by back-propagating the errors of the units of the output layer.

- This method is often called the Back-propagation learning rule.

- Back-propagation can also be considered as a generalization of the delta rule for non-linear activation functions and multi-layer networks.

- Back-propagation is a systematic method of training multi-layer artificial neural networks.

BACK-PROPAGATION NETWORK

Real world is faced with a situations where data is incomplete or noisy. To make reasonable predictions about what is missing from the information available is a difficult task when there is no a good theory available that may to help reconstruct the missing data. It is in such situations the Back-propagation (Back-Prop) networks may provide some answers.

- A BackProp network consists of at least three layers of units :
  - an **input layer**,
  - at least one intermediate **hidden layer**, and
  - an **output layer**.
- Typically, units are connected in a **feed-forward fashion** with input units fully connected to units in the hidden layer and hidden units fully connected to units in the output layer.
- When a BackProp network is cycled, an input pattern is propagated forward to the output units through the intervening input-to-hidden and hidden-to-output weights.
- The output of a BackProp network is interpreted as a classification decision.

BACK-PROPAGATION NETWORK

➢With BackProp networks, learning occurs during a training phase. The steps followed during learning are :

- ❑each input pattern in a training set is applied to the input units and then propagated forward.
- ❑the pattern of activation arriving at the output layer is compared with the correct (associated) output pattern to calculate an error signal.
- ❑the error signal for each such target output pattern is then back-propagated from the outputs to the inputs in order to appropriately adjust the weights in each layer of the network.
- ❑after a BackProp network has learned the correct classification for a set of inputs, it can be tested on a second set of inputs to see how well it classifies untrained patterns.

BACK-PROPAGATION NETWORK-THEORY

- The back propagation learning algorithm is one of the most important developments in neural networks.
- This learning algorithm is applied to a multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are al*so called back -propagation networks* ***(BPNs).***
- For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple Perceptron networks with differentiable units.
- This is a method where the error is propagated back to the hidden unit.
- The aim of the neural network is to train the net to achieve a balance between the net's ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization).

BACK-PROPAGATION NETWORK-THEORY

- The back propagation algorithm is different from other networks in respect to the process by which weights are calculated during the learning period of the network.
- The general difficulty with the multilayer Perceptrons is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error.
- When the hidden layers are increased, the network training becomes more complex.
- To update weights, the error must be calculated. The error, Which is the difference between the actual (calculated) and the desired (target) output, is easily measured at the Output layer.
- It should be noted that at the hidden layers, there is no direct information of the error. Therefore, other techniques should be used to calculate an error at the hidden layer, which will cause minimization of the output error, and this is the ultimate goal.

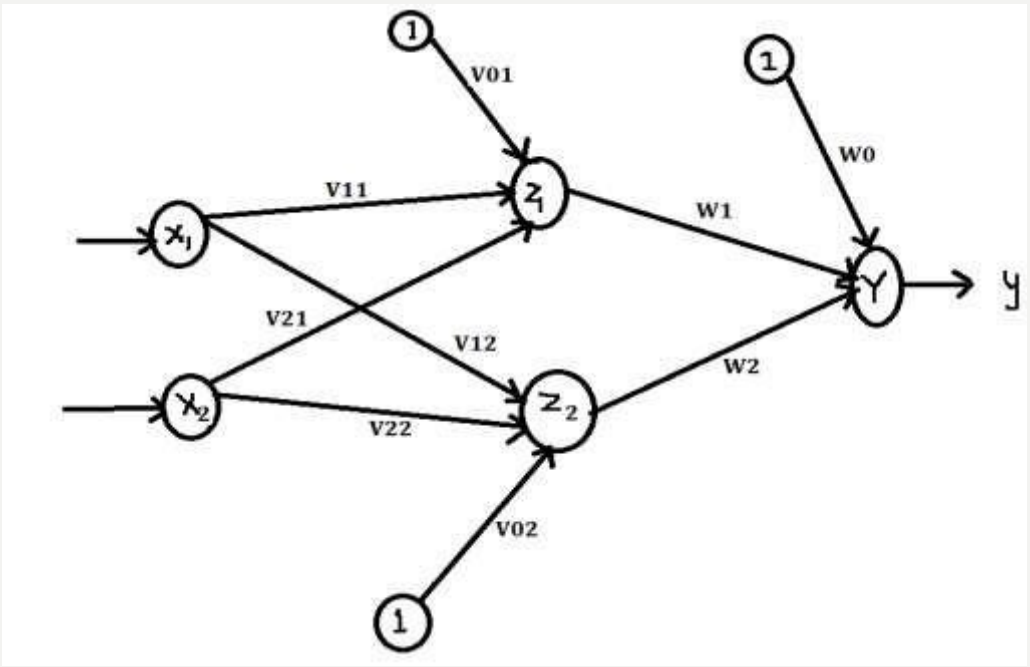
BACK-PROPAGATION NETWORK-THEORY

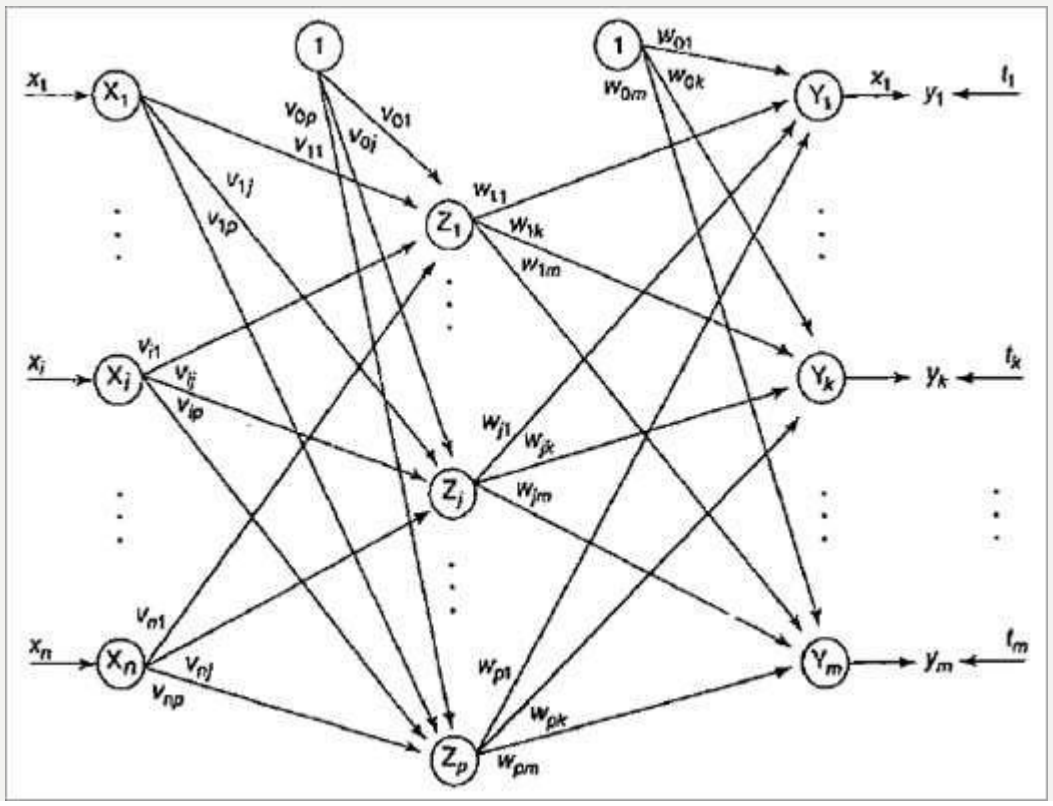
- The training of the BPN is done in three stages:
  - ***the feed-forward of the input training pattern,***
  - ***The calculation and back-propagation of the error, and***
  - ***updation of weights.***

- The testing of the BPN involves the computation of feed-forward phase only. There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient. Even though the training is very slow, once the network is trained it can produce its outputs very rapidly.

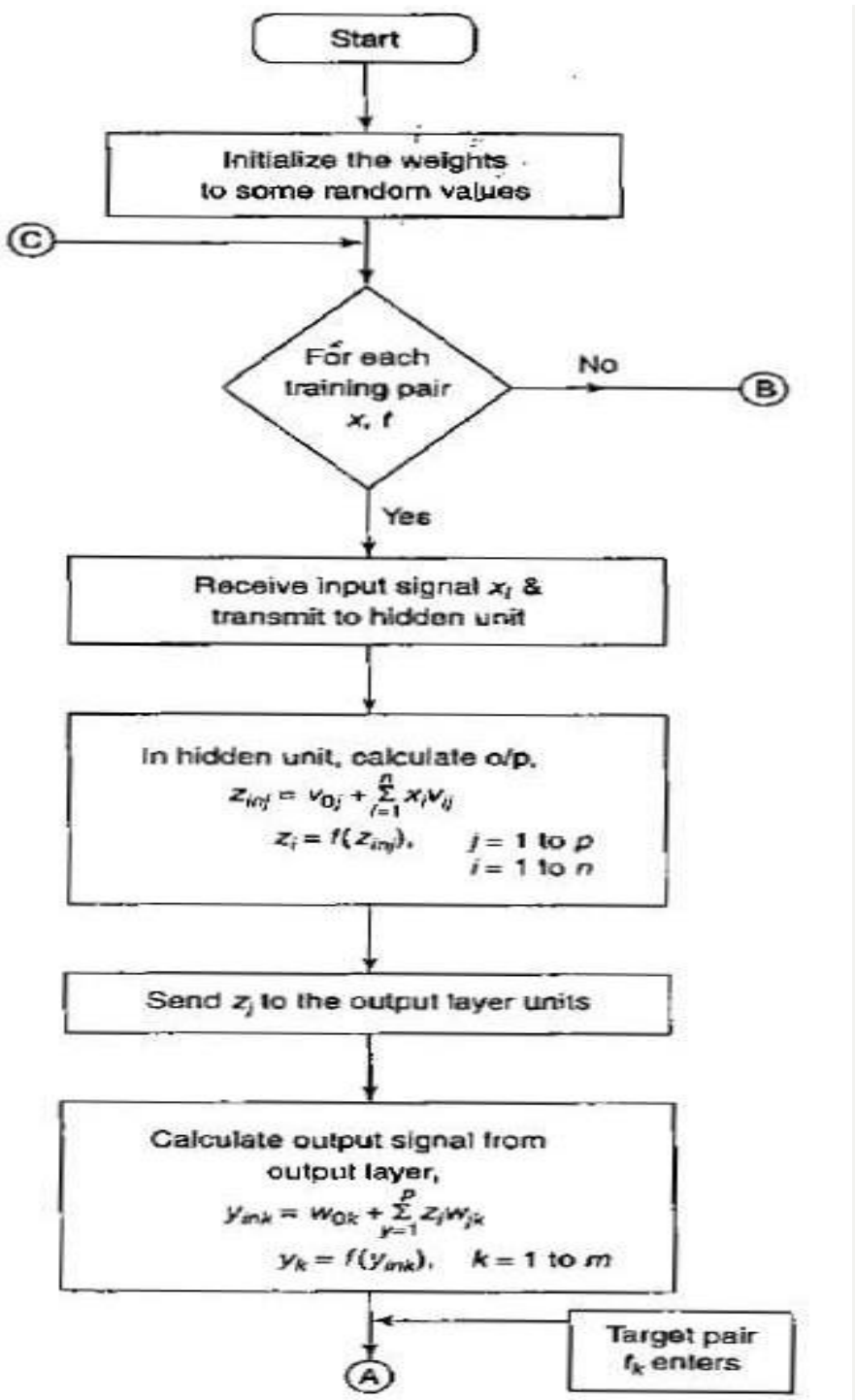
BACK-PROPAGATION NETWORK-ARCHITECTURE

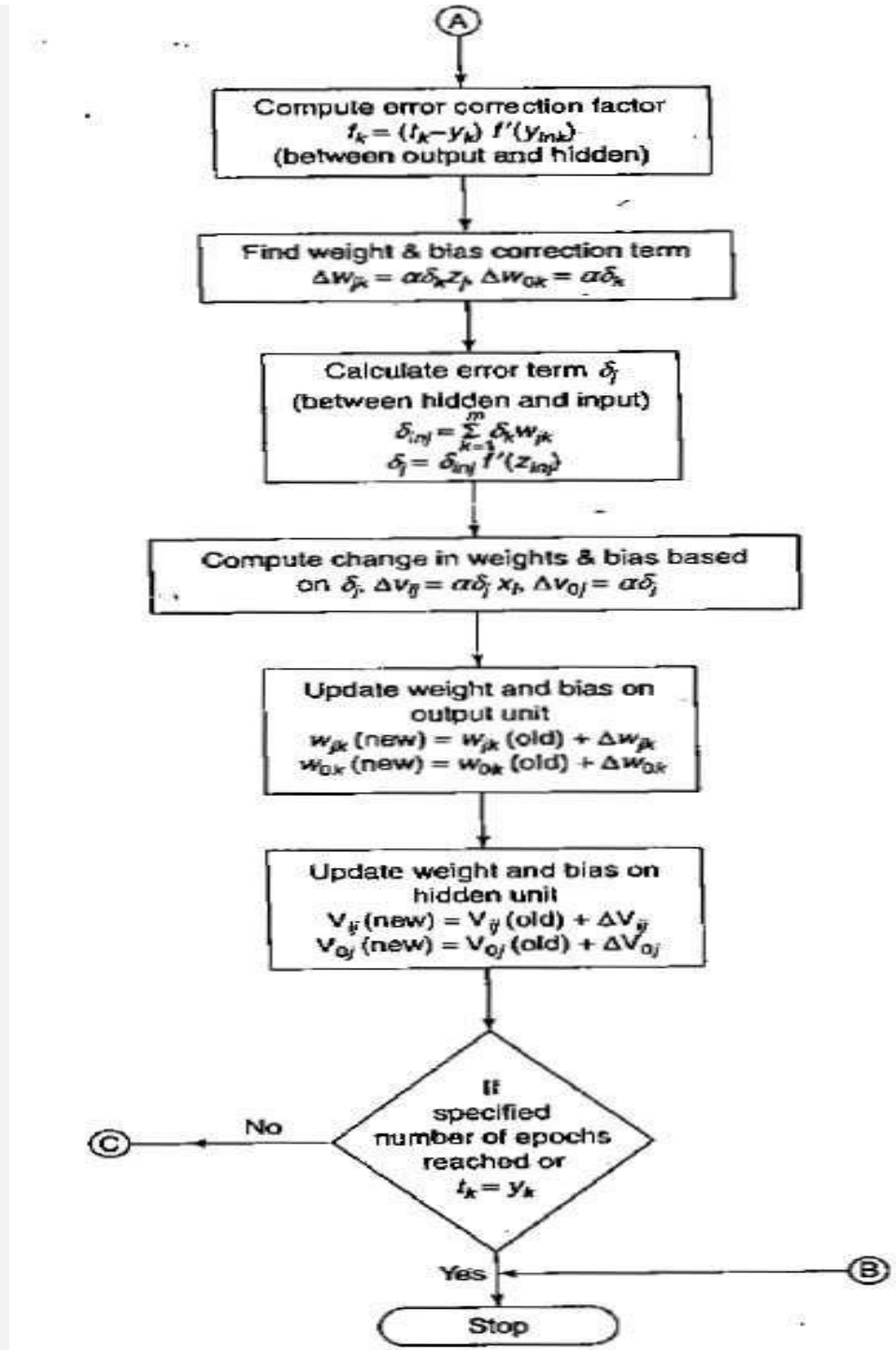
- A back-propagation neural network *is a* ***multilayer, feed-forward neural network*** *consisting of an input layer,* a hidden layer and an output layer.
- The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights. Figure below shows the architecture of a **BPN**, depicting only the direction of information flow for the feed-forward phase.
- During the back-propogation phase of learning, *signals are sent in the reverse direction.*
- The inputs sent to the BPN and the output obtained from the net could be either binary (0, 1) or bipolar (-1, + 1). The activation function could be any function which increases monotonically and is also differentiable.





BACK-PROPAGATION NETWORK-FLOWCHART FOR TRAINING PROCESS





The error back-propagation learning algorithm can be outlined in the following algorithm:

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2–9 when stopping condition is false.

Step 2: Perform Steps 3–8 for each training pair.

Feed-forward phase (Phase I)

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit $z_j (j = 1$ to $p)$ sums its weighted input signals to calculate net input:

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

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

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

and send the output signal from the hidden unit to the input of output layer units.

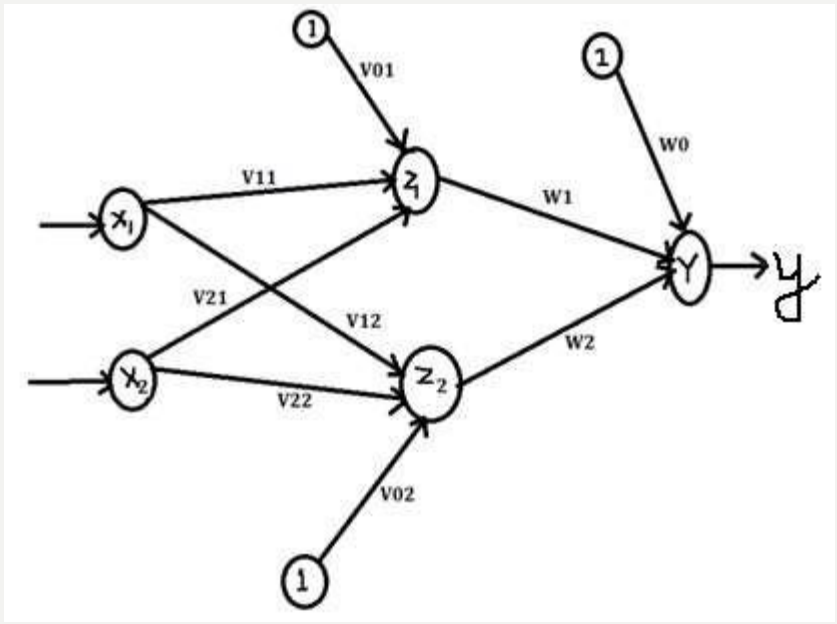
Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

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

Feed-forward phase (Phase I)



From the above diagram we can see that

Input vector to Z_1: $[v_{11}, v_{21}, v_{01}]$

Input vector to Z_2: $[v_{12}, v_{22}, v_{02}]$

Input vector to Y: $[w_1, w_2, w_0]$

The activation function is given by **$f(x)= 1/ (1 +e^{-x})$**

Input = [x1,x2] and target **t=y**

1: Calculate the net input weight for Z_1 and Z_2

$Z_{in1} = v_{01} + x_1 * v_{11} + x_2 * v_{21}$

$Z_{in2} = v_{02} + x_1 * v_{21} + x_2 * v_{22}$

2: Apply the Activation Function

$z_1 = f(Z_{in1}) = 1/1+e^{-Zin1}$

$z_2 = f(Z_{in2}) = 1/1+e^{-Zin2}$

3: Calculate the Net input of Output layer

$y_{in} = w_0 + z_i * w_1 + z_j * w_2$

4: Calculate the Net output using activation

$y = f(y_{in}) = 1/1+e^{-yin}$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Back-propagation of error (Phase II):

Step 6: Each output unit $y_k (k = 1 \text{ to } m)$ receives a target pattern corresponding to the input training pattern and computes the error correction term:

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

δ_k = (target − output) * derivative of $f(y_{in})$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

$$\Delta w_1 = \alpha(t - y_{in}) x_1$$

$$\Delta w_2 = \alpha(t - y_{in}) x_2$$

$$\Delta b = \alpha(t - y_{in})$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit $(z_j, j = 1 \text{ to } p)$ sums its delta inputs from the output units:

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

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

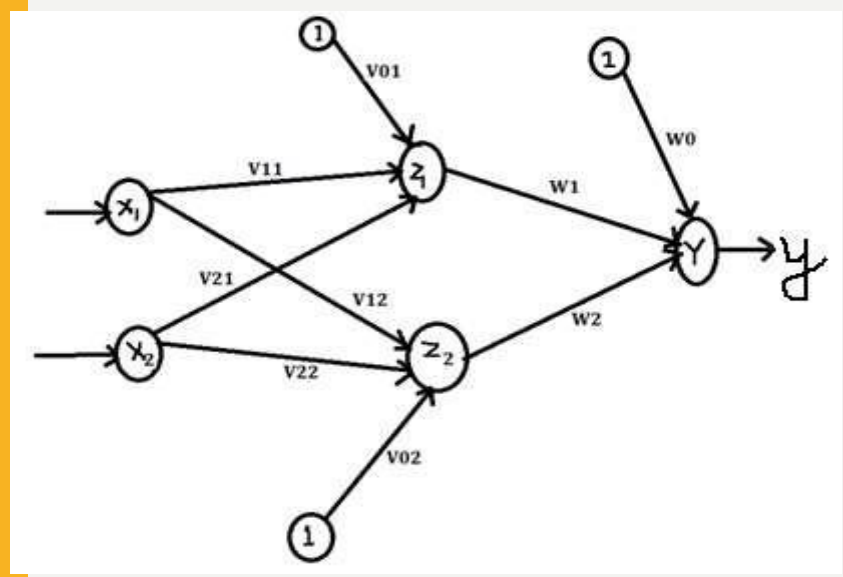
$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

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

BACK-PROPAGATION NETWORK-
TRAINING ALGORITHM

Back-propagation of error (Phase II):



5: Calculation of Error between hidden layer and output layer

$$\delta_k = (t_k - y_k) * \text{derivative } f(y_{in})$$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

$$\delta_1 = (t - y) * \text{derivative } f(y_{in})$$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

6: Find the changes in weights between hidden and output layer:

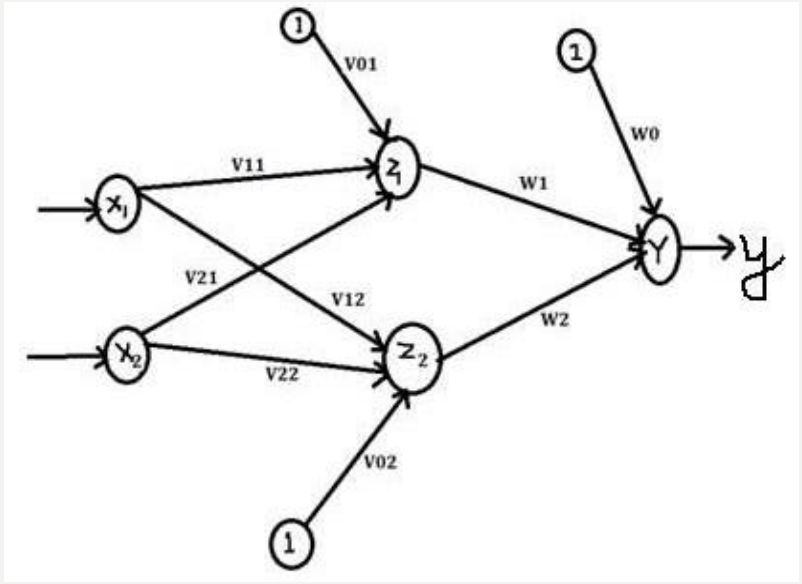
$$\Delta w_1 = \alpha \delta_1 z_1$$

$$\Delta w_2 = \alpha \delta_1 z_2$$

$$\Delta w_0 = \alpha \delta_1$$

BACK-PROPAGATION NETWORK-
TRAINING ALGORITHM

Back-propagation of error (Phase II):



Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj}) \qquad \delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11}$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21}$$

$$\text{Error, } \delta_1 = \delta_{in1} f'(z_{in1})$$

$$f'(z_{in1}) = f(z_{in1})[1 - f(z_{in1})]$$

$$\delta_1 = \delta_{in1} f'(z_{in1})$$

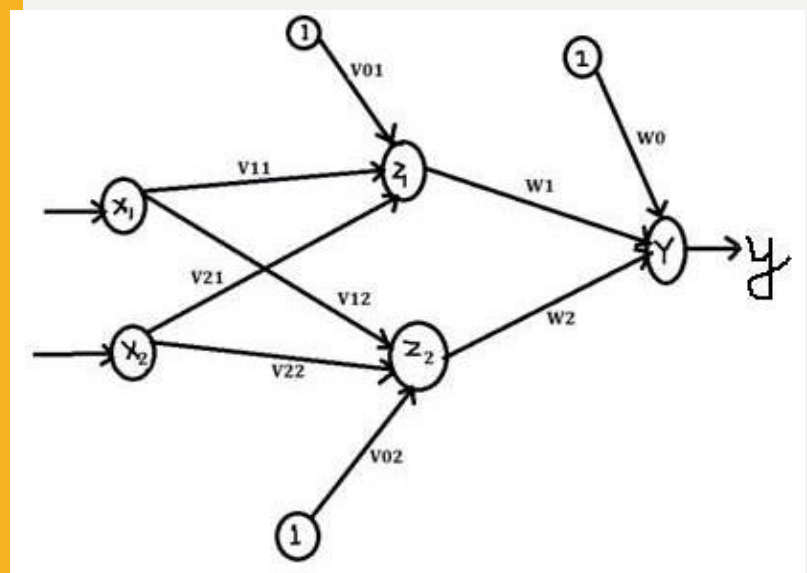
$$\text{Error, } \delta_2 = \delta_{in2} f'(z_{in2})$$

$$f'(z_{in2}) = f(z_{in2})[1 - f(z_{in2})]$$

$$\delta_2 = \delta_{in2} f'(z_{in2})$$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Back-propagation of error (Phase II):



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

$$\Delta v_{11} = \alpha \delta_1 x_1$$

$$\Delta v_{21} = \alpha \delta_1 x_2$$

$$\Delta v_{01} = \alpha \delta_1$$

$$\Delta v_{12} = \alpha \delta_2 x_1$$

$$\Delta' v_{22} = \alpha \delta_2 x_2$$

$$\Delta v_{02} = \alpha \delta_2$$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Weight and bias updation (Phase III):

Step 8: Each output unit (y_k, $k = 1$ to m) updates the bias and weights:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

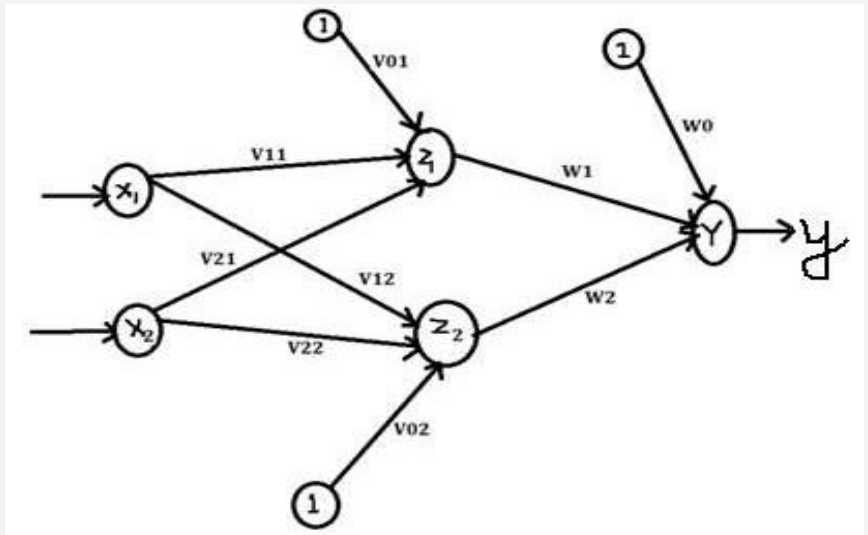
Each hidden unit (z_j, $j = 1$ to p) updates its bias and weights:

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

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Weight and bias updation (Phase III):



$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01}$$
$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11}$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12}$$

$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$

$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2$$

Learning Factors of Back-Propagation Network

➢The training of a BPN is based on the choice of various parameters. Also, the convergence of the BPN is based on some important learning factors, they are:

1. **Initial Weights:**
   - The initial random weights chosen are of very small value as the larger inputs in binary sigmoidal functions may lead to saturation at the very beginning, thereby leading the function been stuck at local minima.
   - Some ways of initialization of weights can be using Nguyen-Widow's initialization.
   - It analyzes the response of hidden neurons to a single input, by improving the learning ability of hidden units.
   - This leads to faster convergence of BPN.

1. **Learning rate:**
   - The learning rate (***α***) affects the convergence of the BPN.
   - A large value of learning rate, helps in faster convergence but might lead to overshooting.
   - The range of from 10^{-3} to 10 is used for various BPN experiments.
   - Thus, a large learning rate leads to rapid learning but there is oscillation of weights, while the lower learning rare leads to slower learning.

Learning Factors of Back-Propagation Network

3. **Number of Training Data:**
   - The training data should be sufficient and proper.
   - There exists a rule of thumb, which states that the training data should cover the entire expected input space, and while training, training-vector pairs should be selected randomly from the set.

4. **Number of Hidden Layer Nodes:**
   - If there exists more than one hidden layer in a BPN, the calculation performed for a single layer are repeated for all the layers and are summed up at the end.
   - In case of all multilayer feed-forward networks, the size of a hidden layer is very important. The number of hidden units required for an application needs to be determined separately.
   - The number of hidden layer nodes is chosen for optimum performance of the network.
   - For networks that do not converge to a solution, more hidden nodes can be chosen while for networks with fast convergence few hidden layer nodes are selected.
5. **Generalization:**
   - The best network for generalization is BPN.
   - A network is said to be generalized when it sensibly interpolates with input networks that are new to the network.
   - When there are many trainable parameters for the given amount of training data, the network learns well but does not generalize well. This is usually called overfitting or overtraining.
   - One solution to this problem is to monitor the error on the test set and terminate the training when the error increases.

6. Momentum Factor:

The gradient descent is very slow if the learning rate α is small and oscillates widely if α is too large. One very efficient and commonly used method that allows a larger learning rate without oscillations is by adding a momentum factor to the normal gradient-descent method.

The momentum factor is denoted by $\eta \in [0, 1]$ and the value of 0.9 is often used for the momentum factor. Also, this approach is more useful when some training data are very different from the majority of data. A momentum factor can be used with either pattern by pattern updating or batch-mode updating. In case of batch mode, it has the effect of complete averaging over the patterns. Even though the averaging is only partial in the pattern-by-pattern mode, it leaves some useful information for weight updation.

The weight updation formulas used here are

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

and

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

The momentum factor also helps in faster convergence.

BACK-PROPAGATION NETWORK-
Testing Algorithm

The testing procedure of the BPN is as follows:

Step 0: Initialize the weights. The weights are taken from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector.

Step 2: Set the activation of input unit for x_i ($i = 1$ to n).

Step 3: Calculate the net input to hidden unit x and its output. For $j = 1$ to p,

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

Step 4: Now compute the output of the output layer unit. For $k = 1$ to m,

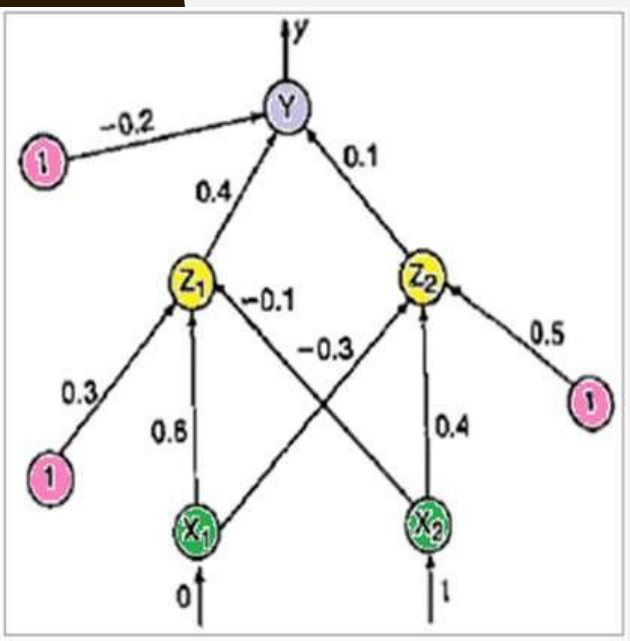
$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

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

Use sigmoidal activation functions for calculating the output.

BACK-PROPAGATION NETWORK- EXAMPLE

Q. Using back-propagation network, find the new weights for the net shown in Figure. It is presented with the input pattern [0, 1] and the target output is 1. Use a learning rate *a*= 0.25 *and* binary sigmoidal activation function.



Given:

->Input vector = [0,1]

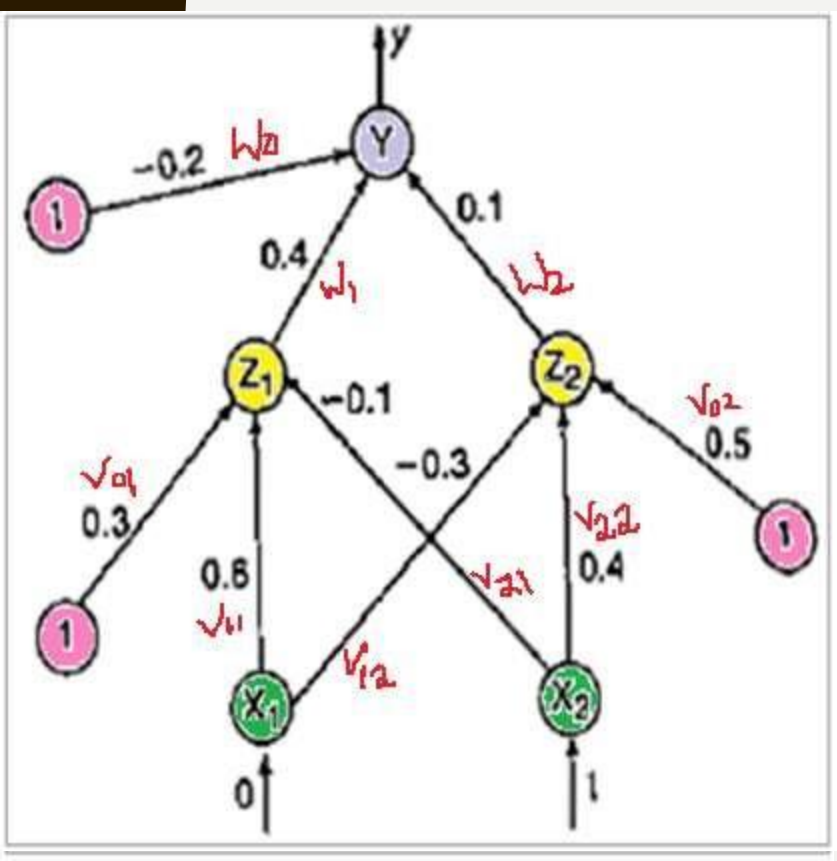
->Target output = 1

->Learning Rate = 0.25

->Activation function**= binary sigmoidal activation function**

$$=f(x)= 1/ (1 +e^{-x})$$

BACK-PROPAGATION NETWORK- EXAMPLE



Given:

->Input vector = [0,1] ->Target output = 1 ->Learning Rate = 0.25

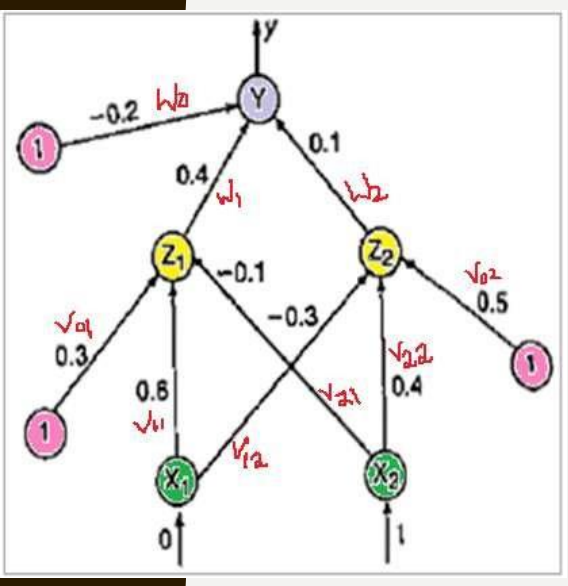
->Activation function= binary sigmoidal activation function

Solution:

From the diagram we can see that

- **Input vector to Z_1**: $[v_{11}, v_{21}, v_{01}]$ = [0.6, -0.1, 0.3]
- **Input vector to Z_2:** $[v_{12}, v_{22}, v_{02}]$ = [-0.3, 0.3, 0.5]
- **Input vector to Y:** $[w_1, w_2, w_0]$ = [0.4, 0.1, -0.2]
- The activation function is given by **$f(x)= 1/ (1 +e^{-x})$**
- **Input x= [0,1]** and target **t=1**

BACK-PROPAGATION NETWORK- EXAMPLE



Input vector to Z_1: $[v_{11}, v_{21}, v_{01}]$ = [0.6, -0.1, 0.3]
Input vector to Z_2: $[v_{12}, v_{22}, v_{02}]$ = [-0.3, 0.3, 0.5]
Input vector to Y: $[w_1, w_2, w_0]$ = [0.4, 0.1, -0.2]
The activation function is given by **$f(x)= 1/ (1 +e^{-x})$**
Input x= [0,1] and target **t=1**

Step 1: Calculate the net input weight for Z_1 and Z_2

$Z_{in1} = v_{01} + x_1 * v_{11} + x_2 * v_{21}$
$= 0.3 + 0* 0.6 +1 *0.1$
$= 0.2$

$Z_{in2} = v_{02} + x_1 * v_{21} + x_2 * v_{22}$
$= 0.5+ 0*0.6 +1*(0.4)$
$= 0.9$

Step 2: Apply the Activation Function

$z_1 = f(Z_{in1})$
$= 1/1+e^{-Zin1}$
$= 1/1+e^{-0.2}$
$= 0.5498$

$z_2 = f(Z_{in2})$
$= 1/1+e^{-Zin2}$
$= 1/1+e^{-0.9}$
$= 0.7109$

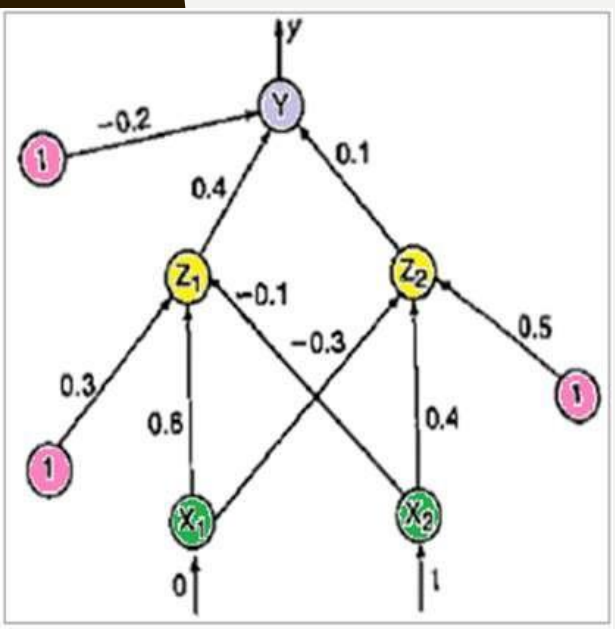
Step 3: Calculate the Net input of Output layer

$y_{in} = w_0 + z_i * w_1 + z_j * w_2$
$= -0.2+ 0.5498 * 0.4 + 0.7109 *0.1$
$= 0.09101$

Step 4: Calculate the Net output using activation

$y = f(y_{in})$
$= 1/1+e^{-yin}$
$= 1/1+e^{-0.09101}$
$= 0.5227$

BACK-PROPAGATION NETWORK- EXAMPLE



Step 5: Calculation of Error between hidden layer and output layer

$\delta_k = (t_k - y_k) *$ derivative $f(y_{in})$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in})) = 0.5227(1-0.5227)$

$= 0.2495$

⇨ $\delta_k = (t_k - y_k) *$ derivative $f(y_{in})$

⇨ $(1 - 0.5227) * (0.2495)$

⇨ 0.1191

This implies

$$\delta_1 = (1 - 0.5227)\ (0.2495) = 0.1191$$

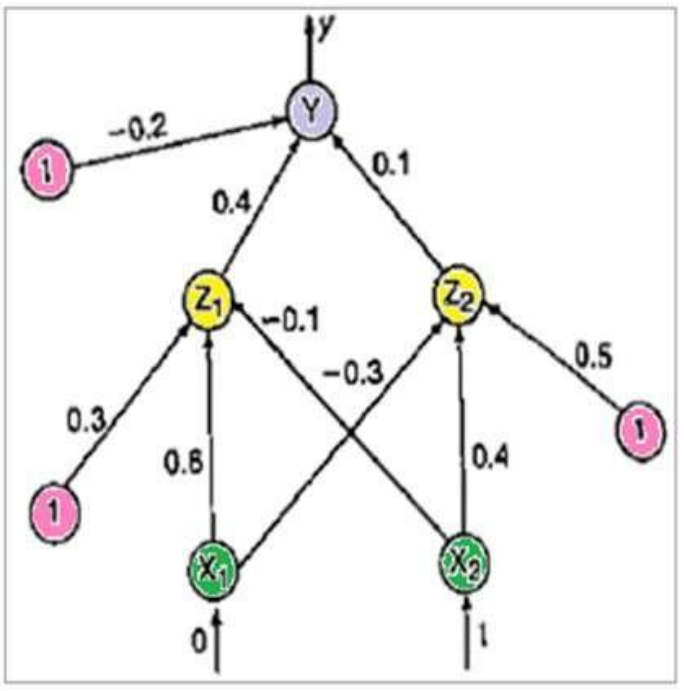
Step 6: Weight Updation

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.1191 \times 0.5498$$
$$= 0.0164$$
$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.1191 \times 0.7109$$
$$= 0.02117$$
$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.1191 = 0.02978$$

BACK-PROPAGATION NETWORK- EXAMPLE

Step 7: Calculation of Error between hidden layer and output layer



- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

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

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

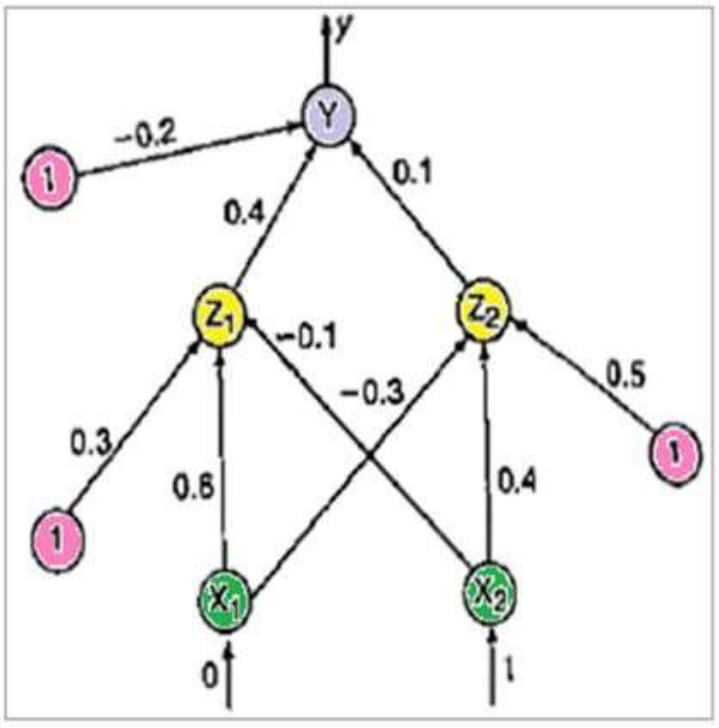
$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.1191 \times 0.4 = 0.04764$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.1191 \times 0.1 = 0.01191$$

BACK-PROPAGATION NETWORK- EXAMPLE

Step 7: Calculation of Error between hidden layer and output layer



$$\text{Error, } \delta_1 = \delta_{in1} f'(z_{in1})$$

$$f'(z_{in1}) = f(z_{in1})\,[1 - f(z_{in1})]$$

$$= 0.5498[1 - 0.5498] = 0.2475$$

$$\delta_1 = \delta_{in1} f'(z_{in1})$$

$$= 0.04764 \times 0.2475 = 0.0118$$

$$\text{Error, } \delta_2 = \delta_{in2} f'(z_{in2})$$

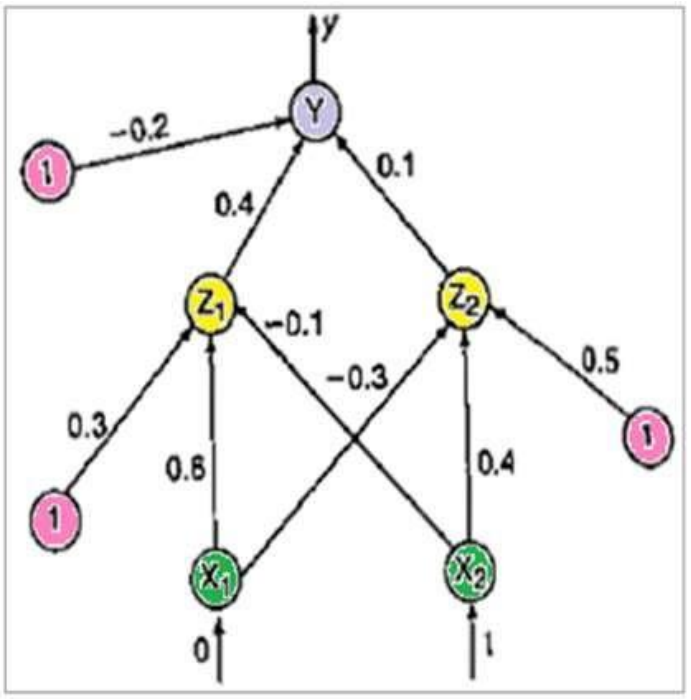
$$f'(z_{in2}) = f(z_{in2})\,[1 - f(z_{in2})]$$

$$= 0.7109[1 - 0.7109] = 0.2055$$

$$\delta_2 = \delta_{in2} f'(z_{in2})$$

$$= 0.01191 \times 0.2055 = 0.00245$$

BACK-PROPAGATION NETWORK- EXAMPLE



Step 8: Weight Updation

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

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.0118 \times 0 = 0$$

$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.0118 \times 1 = 0.00295$$

$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.0118 = 0.00295$$

$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.00245 \times 0 = 0$$

$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.00245 \times 1 = 0.0006125$$

$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.00245 = 0.0006125$$

BACK-PROPAGATION NETWORK- EXAMPLE

Step 8: Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 + 0 = 0.6$$

$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 + 0 = -0.3$$

$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$= -0.1 + 0.00295 = -0.09705$$

$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$
$$= 0.4 + 0.0006125 = 0.4006125$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 + 0.0164$$
$$= 0.4164$$

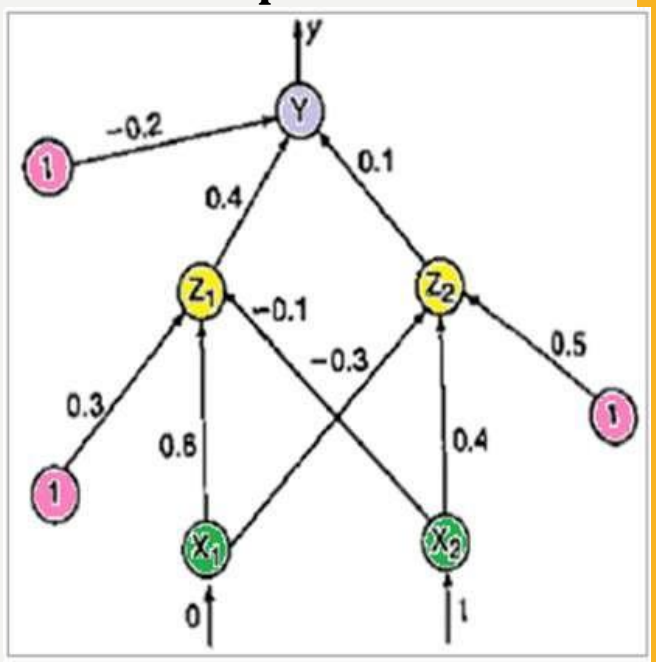
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.02117$$
$$= 0.12117$$

$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.00295$$
$$= 0.30295$$

$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$= 0.5 + 0.0006125 = 0.5006125$$

$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.02978$$
$$= -0.17022$$

Thus, the final weights have been computed for the network shown in Figure.



PROGRAM-4 BACK-PROPAGATION NETWORK

BACK-PROPAGATION NETWORK-TRAINING ALGORITHM

The error back-propagation learning algorithm can be outlined in the following algorithm:

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2–9 when stopping condition is false.

Step 2: Perform Steps 3–8 for each training pair.

Feed-forward phase (Phase I)

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit $z_j (j = 1$ to $p)$ sums its weighted input signals to calculate net input:

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

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

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

and send the output signal from the hidden unit to the input of output layer units.

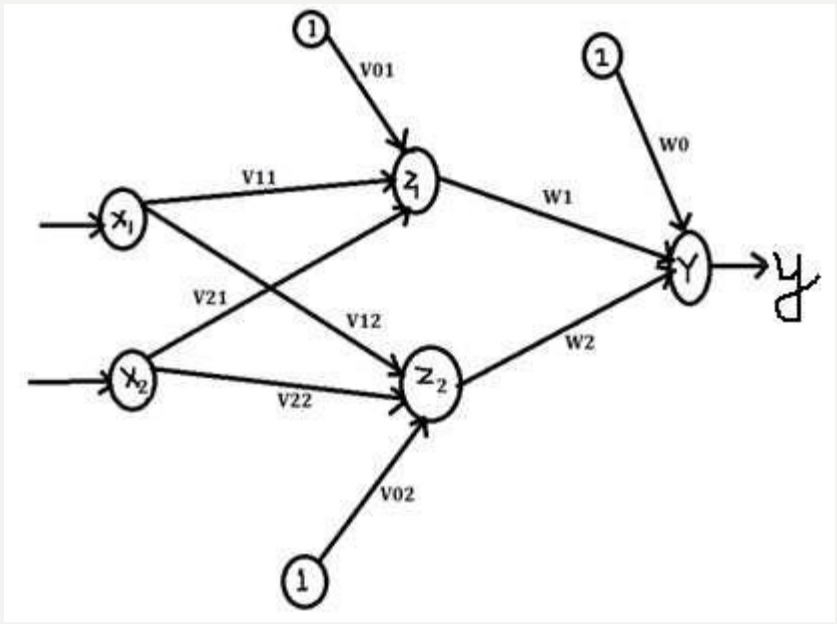
Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

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

Feed-forward phase (Phase I)



From the above diagram we can see that

Input vector to Z_1: $[v_{11}, v_{21}, v_{01}]$

Input vector to Z_2: $[v_{12}, v_{22}, v_{02}]$

Input vector to Y: $[w_1, w_2, w_0]$

The activation function is given by **$f(x)= 1/ (1 +e^{-x})$**

Input = [x1,x2] and target **t=y**

1: Calculate the net input weight for Z_1 and Z_2

$Z_{in1} = v_{01} + x_1 * v_{11} + x_2 * v_{21}$

$Z_{in2} = v_{02} + x_1 * v_{21} + x_2 * v_{22}$

2: Apply the Activation Function

$z_1 = f(Z_{in1}) = 1/1+e^{-Zin1}$

$z_2 = f(Z_{in2}) = 1/1+e^{-Zin2}$

3: Calculate the Net input of Output layer

$y_{in} = w_0 + z_i * w_1 + z_j * w_2$

4: Calculate the Net output using activation

$y = f(y_{in}) = 1/1+e^{-yin}$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Back-propagation of error (Phase II):

Step 6: Each output unit $y_k (k = 1 \text{ to } m)$ receives a target pattern corresponding to the input training pattern and computes the error correction term:

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

δ_k = (target − output) * derivative of $f(y_{in})$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

$$\Delta w_1 = \alpha(t - y_{in}) x_1$$

$$\Delta w_2 = \alpha(t - y_{in}) x_2$$

$$\Delta b = \alpha(t - y_{in})$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit $(z_j, j = 1 \text{ to } p)$ sums its delta inputs from the output units:

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

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

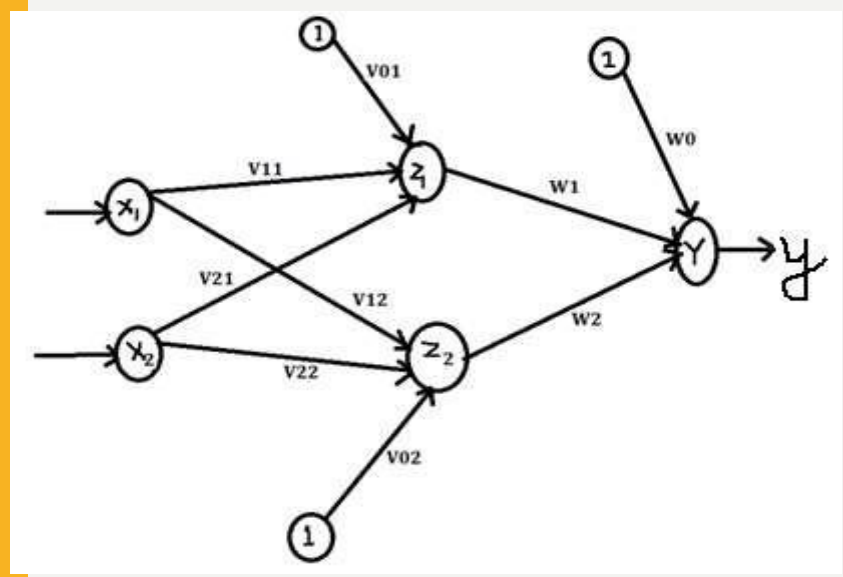
$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

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

BACK-PROPAGATION NETWORK-
TRAINING ALGORITHM

Back-propagation of error (Phase II):



5: Calculation of Error between hidden layer and output layer

$$\delta_k = (t_k - y_k) * \text{derivative } f(y_{in})$$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

$$\delta_1 = (t - y) * \text{derivative } f(y_{in})$$

Derivative of $f(y_{in}) = f(y_{in})(1 - f(y_{in}))$

6: Find the changes in weights between hidden and output layer:

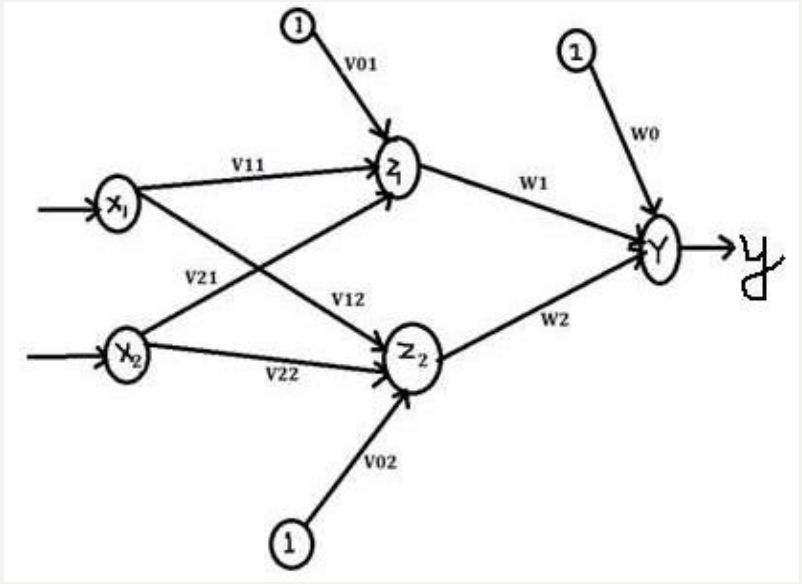
$$\Delta w_1 = \alpha \delta_1 z_1$$

$$\Delta w_2 = \alpha \delta_1 z_2$$

$$\Delta w_0 = \alpha \delta_1$$

BACK-PROPAGATION NETWORK-
TRAINING ALGORITHM

Back-propagation of error (Phase II):



Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj}) \qquad \delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11}$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21}$$

$$\text{Error, } \delta_1 = \delta_{in1} f'(z_{in1})$$

$$f'(z_{in1}) = f(z_{in1})[1 - f(z_{in1})]$$

$$\delta_1 = \delta_{in1} f'(z_{in1})$$

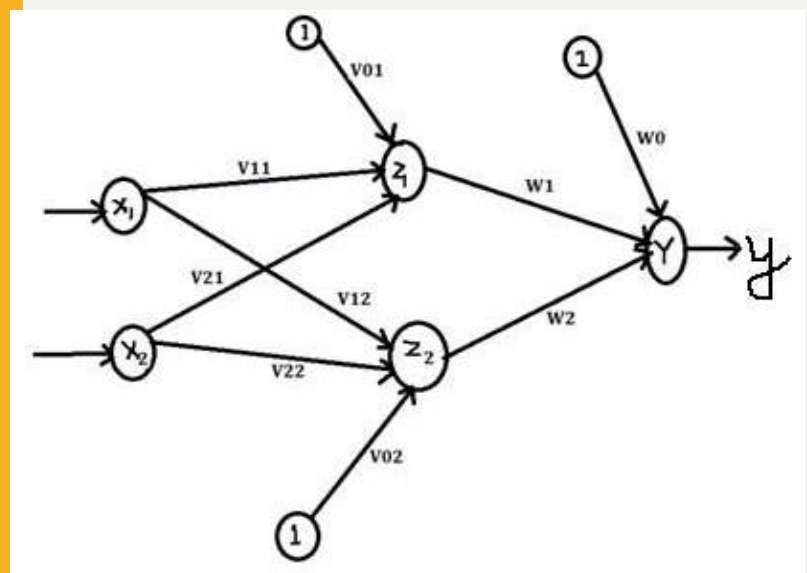
$$\text{Error, } \delta_2 = \delta_{in2} f'(z_{in2})$$

$$f'(z_{in2}) = f(z_{in2})[1 - f(z_{in2})]$$

$$\delta_2 = \delta_{in2} f'(z_{in2})$$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Back-propagation of error (Phase II):



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

$$\Delta v_{11} = \alpha \delta_1 x_1$$

$$\Delta v_{21} = \alpha \delta_1 x_2$$

$$\Delta v_{01} = \alpha \delta_1$$

$$\Delta v_{12} = \alpha \delta_2 x_1$$

$$\Delta' v_{22} = \alpha \delta_2 x_2$$

$$\Delta v_{02} = \alpha \delta_2$$

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Weight and bias updation (Phase III):

Step 8: Each output unit (y_k, $k = 1$ to m) updates the bias and weights:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

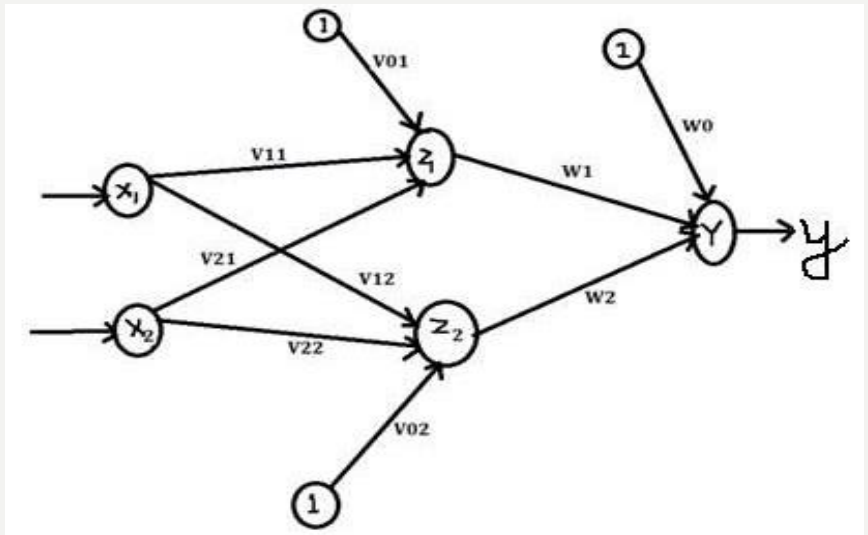
Each hidden unit (z_j, $j = 1$ to p) updates its bias and weights:

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

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

BACK-PROPAGATION NETWORK
TRAINING ALGORITHM

Weight and bias updation (Phase III):



$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01}$$
$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11}$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12}$$

$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$

$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2$$

BACK-PROPAGATION NETWORK

Q. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

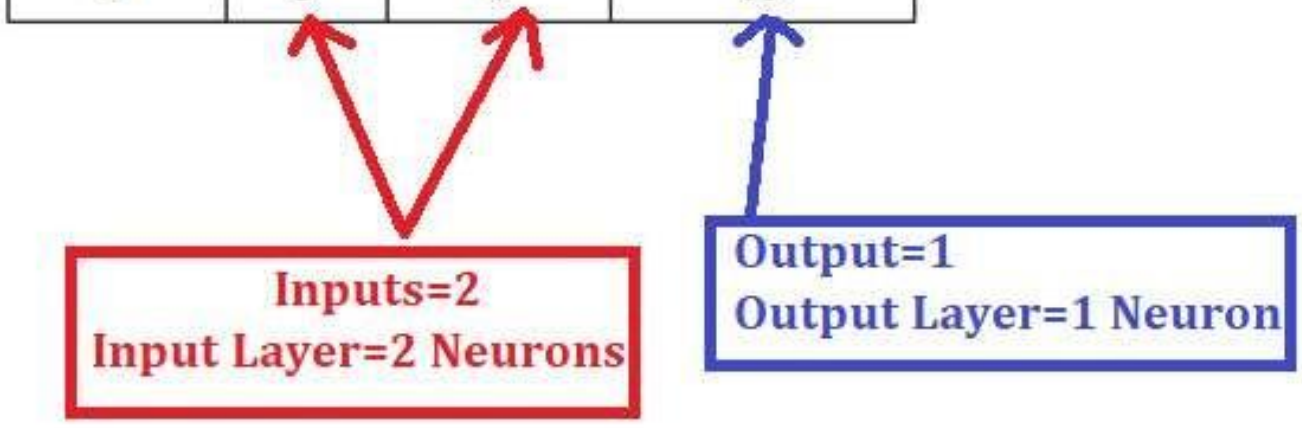
Training Examples:

Example	Sleep	Study	Expected % in Exams
1	2	9	92
2	1	5	86
3	3	6	89

NO. OF NEURONS IN EACH LAYER

Training Examples:

Example	Sleep	Study	Expected % in Exams
1	2	9	92
2	1	5	86
3	3	6	89



NO. OF NEURONS IN EACH LAYER

Number of hidden layers and corresponding result

0 - Only capable of representing linear separable functions or decisions.

1 - Can approximate any function that contains a continuous mapping from one finite space to another

2 - Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

More than 2 - Additional layers can learn complex representations but rare scenarios.

Number of neurons in hidden layers

Too few neurons in the hidden layers will result in underfitting, cannot model complicated data sets.

Too many neurons in the hidden layers may result in overfitting, works well on the training dataset and bad on other data. Also, increases the time it takes to train the network.

Rule of thumb guidelines:

- In order to secure the ability of the network to generalize the number of nodes has to be kept as low as possible.
- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

WARNING: Ultimately, the selection of an architecture for your neural network will come down to trial and error.

NO. OF NEURONS IN EACH LAYER

Inputs=2
Input Layer=2 Neurons

Hence, we use one hidden layer, No. of neurons in hidden layer=3

Output=1
Output Layer=1 Neuron

numpy

- **NumPy** stands for Numerical Python.
- **NumPy** is a Python library used for working with arrays.
- **NumPy** is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

numpy.exp( )→e^x

This mathematical function helps user to calculate exponential of all the elements in the input array.

```
# Python program explaining  exp() function
import numpy as np
in_array = [1, 3, 5]
print ("Input array : ", in_array)
out_array = np.exp(in_array)
print ("Output array : ", out_array)
```

Output :
Input array : [1, 3, 5]
Output array : [2.71828183 20.08553692 148.4131591]

numpy.random.uniform()

- Draw samples from a uniform distribution.
- Used to describe probability where every event has equal chances of occuring.
- E.g. Generation of random numbers.
- It has three parameters:
    1. a - lower bound - default 0 .0.
    2. b - upper bound - default 1.0.
    3. size - The shape of the returned array.

Example:
Create a 2x3 uniform distribution sample:

```
from numpy import random
x = random.uniform(size=(2, 3))
print(x)
```

```
Output:
[[0.52245093 0.0794727 0.7688373 ]
[0.61793475 0.19692874 0.3857731 ]]
```

np.dot()

- This function returns the dot product of two arrays. For 2-D vectors, it is the equivalent to matrix multiplication. For 1-D arrays, it is the inner product of the vectors. For N-dimensional arrays, it is a sum product over the **last axis of a** and the **second-last axis of b**.

```
import numpy.matlib
import numpy as np

a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
np.dot(a,b)
```

It will produce the following output –

```
[[37  40]
 [85  92]]
```

numpy.amax()

- This function Return the maximum of an array or maximum along an axis.

Example

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a)           # Maximum of the flattened array
3
>>> np.amax(a, axis=0)   # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1)   # Maxima along the second axis
array([1, 3])
```

PROGRAM NO. 4

```python
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) #maximum of X array longitudinally
y = y/100
```

Training Examples:

Example	Sleep	Study	Expected % in Exams
1	2	9	92
2	1	5	86
3	3	6	89

Normalize the input

Example	Sleep	Study	Expected % in Exams
1	2/3 = 0.66666667	9/9 = 1	0.92
2	1/3 = 0.33333333	5/9 = 0.55555556	0.86
3	3/3 = 1	6/9 = 0.66666667	0.89

PROGRAM

- Let's add our Sigmoid function (this will be part of forward propagation) and
- its derivative (this will be part of backpropagation)

```
#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)
```

- Now let's set how many ***epochs*** we want,
- what our learning rate ***lr*** will be,
- the number of features in our dataset (2),
- the number of hidden layer neurons (3) and
- the number of output neurons (1)

```
#Variable initialization
epoch=5 #Setting training iterations
lr=0.1 #Setting learning rate

inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
```

PROGRAM

- Now we can set our initial weights and biases.
- Let's let ***wh*** and ***bh*** be the weights and biases for the hidden layer neuron, and
- ***wout*** and ***bout*** be the weights and biases for the output neuron:

```
24 #weight and bias initialization
25 wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
26 bh=np.random.uniform(size=(1,hiddenlayer_neurons))
27 wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
28 bout=np.random.uniform(size=(1,output_neurons))
```

wh → weights from input layer(2) to hidden layers(3)
bh → bias weight for hidden neurons(3)

```
wh= [[0.61319388 0.75153121 0.75797285]
     [0.74895864 0.25677049 0.84938154]]

bh= [[0.69925111 0.04961435 0.80667877]]
```

wout → weights from hidden layers(3) to output layer(1)
bout → bias weight for output layer(1)

```
wout= [[0.56123295]
       [0.13551614]
       [0.69158675]]
bout= [[0.7462916]]
```

PROGRAM

- We have our data, our initial weights and biases are set, and our forward and backpropagation functions are ready to calculate.
- **Forward Propagation:**
  - First, we'll get the product of the **weight** and the **input values**.
  - Next, we'll add the **bias** to the products.
  - Then, well use our **sigmoid function** to transform/activate it.
  - We'll repeat this with the result in the output layer neuron.
- **Back Propagation:**
  - We'll calculate the error, the difference between *y* and the output.
  - Next, we'll find gradients of the output and transformation/activation.
  - Now let's find the delta, or the change factor at the output layer by multiplying it by the error.
  - Now the error will propagate back into the network, so we have to get the dot product of the delta and weight parameters between the second and third layer.
  - We'll do the same for the hidden layer, followed by updating the weights and biases.

PROGRAM

```
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
    #Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+bout
    output = sigmoid(outinp)

    #Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr   # dotproduct of nextlayererror and currentlayerop
    bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" ,X)
print("Actual Output: \n" ,y)
print("Predicted Output: \n" ,output)
```

Σ at hidden layer

Σ at output layer

OUTPUT

```
Input:
 [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
 [[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.80837019]
 [0.7977353 ]
 [0.81086374]]
```

QUESTIONS

1. List out the steps in Perceptron learning algorithm for single output classes. (5)
2. Compare supervised and unsupervised learning approaches in ANN. (5)
3. Implement NAND function using McCulloch-Pitts neuron model. (Use binary data representation).(5)
4. Explain the architecture and training algorithm of Back Propagation network. Describe the various terminologies used in the algorithm.(10)
5. With the help of an example, state the role of bias in determining the net output of an Artificial Neural Network. (5)
6. Illustrate the different steps involved in the training algorithm of Perceptrons. (5)
7. Design a Hebb network to realize logical OR function. (10)
8. Implement AND logical function using Perceptrons. (10)
9. How is the training algorithm performed in back-propagation neural networks? (6)
10. With graphical representations, explain the activation functions used in Artificial Neural Networks.(8)

QUESTIONS

11. Obtain the output of the neuron for a network with inputs are given as [x1,x2,x3]=[0.8,0.6,0.4] and the weights are [w1,w2,w3]=[0.1,0.3,-0.2] with bias= 0.35. Also find output for:

    i) Binary sigmoidal ii) Bipolar sigmoidal activation functions. (5)

12. List the stage involved in Back Propagation Algorithm? (5)
13. Discuss the concept of M P Neuron? Implement AND function using MP neuron (take binary data)?(10)
14. Design logical AND using Perceptron network for bipolar inputs and targets? (8)
15. Implement AND function using bipolar inputs and targets using Hebb rule method. (10)
16. Implement OR function using Perceptron training algorithm with binary inputs and bipolar targets. (10)
17. Implement AND function using McCulloch-Pitts neuron (using binary data representation). (5)
18. Compare and contrast biological neuron and artificial neuron (4 points) (3)
19. Obtain the output of the neuron for a network with inputs are given as [x1, x2] = [0.7, 0.8] and the weights are [w1, w2] = [0.2, 0.3] with bias = 0.9. Use i) Binary sigmoidal activation function ii) Bipolar sigmoid activation function (4)
20. State the training algorithm for multiple output classes in Perceptron. (5)
21. List any Five activation functions with their equations and graphs. (5)

QUESTIONS

22. Draw the flowchart of Hebb training algorithm. Design a Hebb net to implement NOR function using with bipolar inputs and targets. (10)
23. Find the weights required to perform the following classifications using Perceptron network: The vectors (1, 1, -1, -1) and (1,-1. 1, -1) are belonging to a class having target value 1. The vectors (-1, -1, -1, 1) and (-1, -1, 1, 1) are belonging to a class having target value -1. Assume learning rate 1 and initial weights as 0.
24. Draw the architecture of Back-Propagation network. Write its testing algorithm.
25. Use McCulloh Pitts neuron to design logic networks of AND and OR logic function.
26. Explain Back propagation neural network Architecture and its algorithm.
27. Implement McCulloh Pits neuron for (i) NAND (ii) XOR functions.
28. Discuss the Back propagation learning methods and algorithm in detail.
29. Explain the working of back propagation neural network with neat architecture and flowchart.
30. Discuss in detail various types of activation functions used in neural network with the aid of graphical as well as mathematical representation and output.
31. Determine the weights of a single layer perceptron for implementing the AND function. Consider the inputs and targets to be bipolar and $\alpha = 1$.