

# NEURAL NETWORKS

# NEURAL NETWORKS:

- A **neuron** is a cell in brain whose principle function is the collection, processing, and dissemination of electrical signals.
- Brains information processing capacity comes from networks of such neurons.
- Due to this reason some earliest AI work aimed to create such artificial networks.

# ARTIFICIAL NEURAL NETWORK (ANN)


- An artificial neural network (ANN) is **an information processing paradigm** that is **inspired by the way biological nervous systems**, as the brain, process information.
- **ANN learns like people, learn by example.**

# Why use neural networks?

- **Ability to derive meaning from complicated or imprecise data, can be used to extract patterns, data classification and detect trends that are too complex to be noticed by either humans or other computer techniques.**



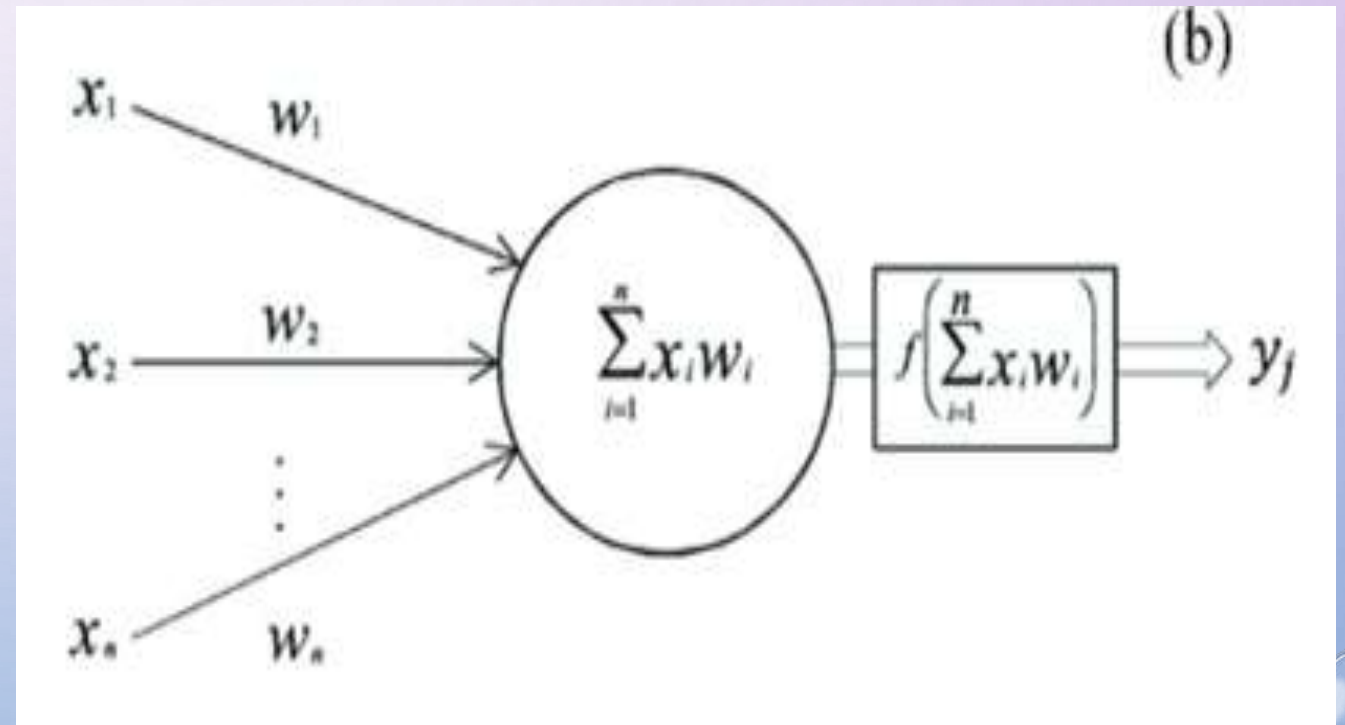
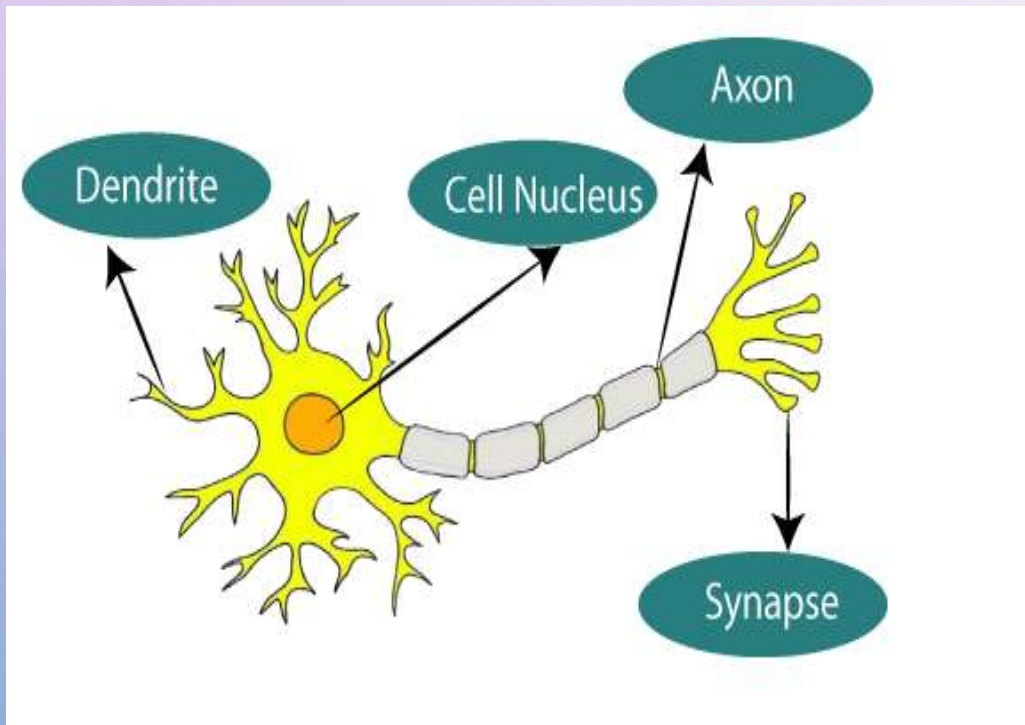
Other advantages include:

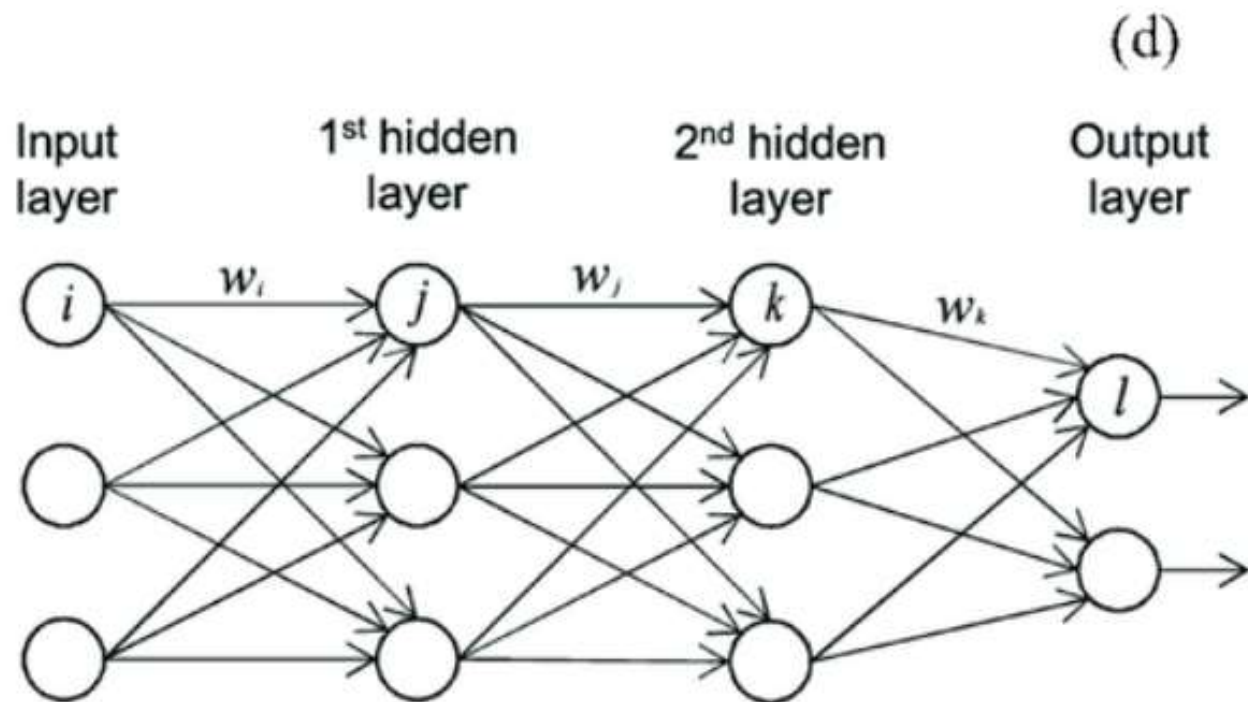
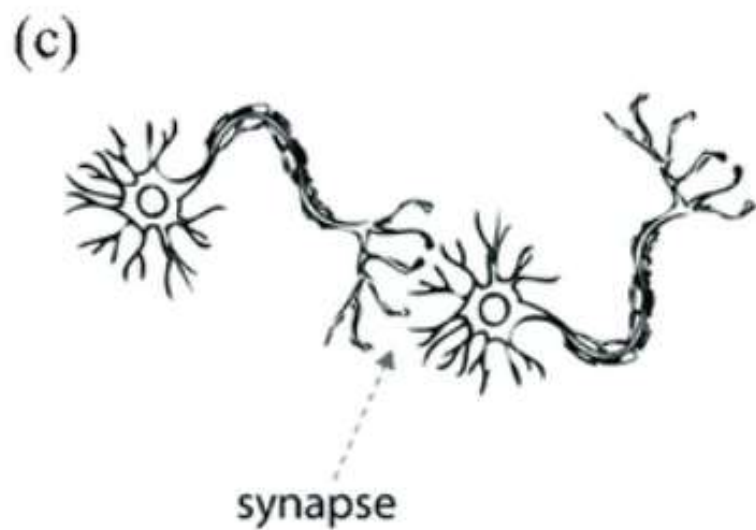
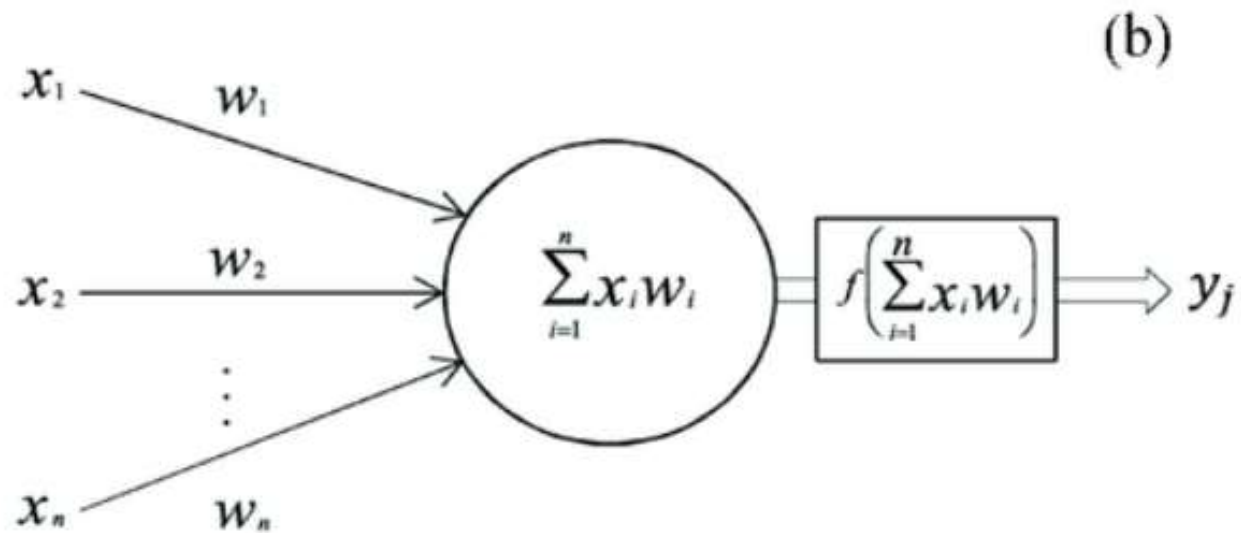
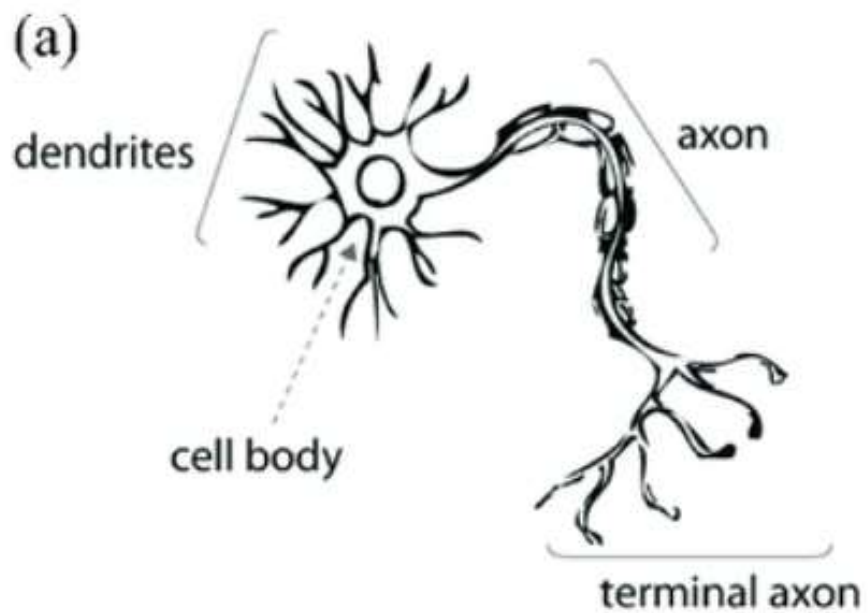
- **1. Adaptive learning:** an ability to learn how to do tasks based on the data given for training or initial experience.
  - **2. Self-Organization:** an ANN can create its own organization or representation of the information it receives during learning time.
- 



- **3. Real time operation:** ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
- **4. Fault tolerance via redundant information coding:** partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage

# BIOLOGICAL NEURAL NETWORKS VS. ARTIFICIAL NEURAL NETWORKS (ANN)







# BIOLOGICAL NEURAL NETWORK :

- Biological neural network (**BNN**) is a **structure that consists of synapse, dendrites, cell body, and axon.**
- In this neural network, the **processing is carried out by neurons.**
- **Dendrites** receive signals from other neurons, **NUCLEUS** sums all the incoming signals and **axon** transmits the signals to other cells.
- The **synapses** are the **input processing element.**



## Advantages:

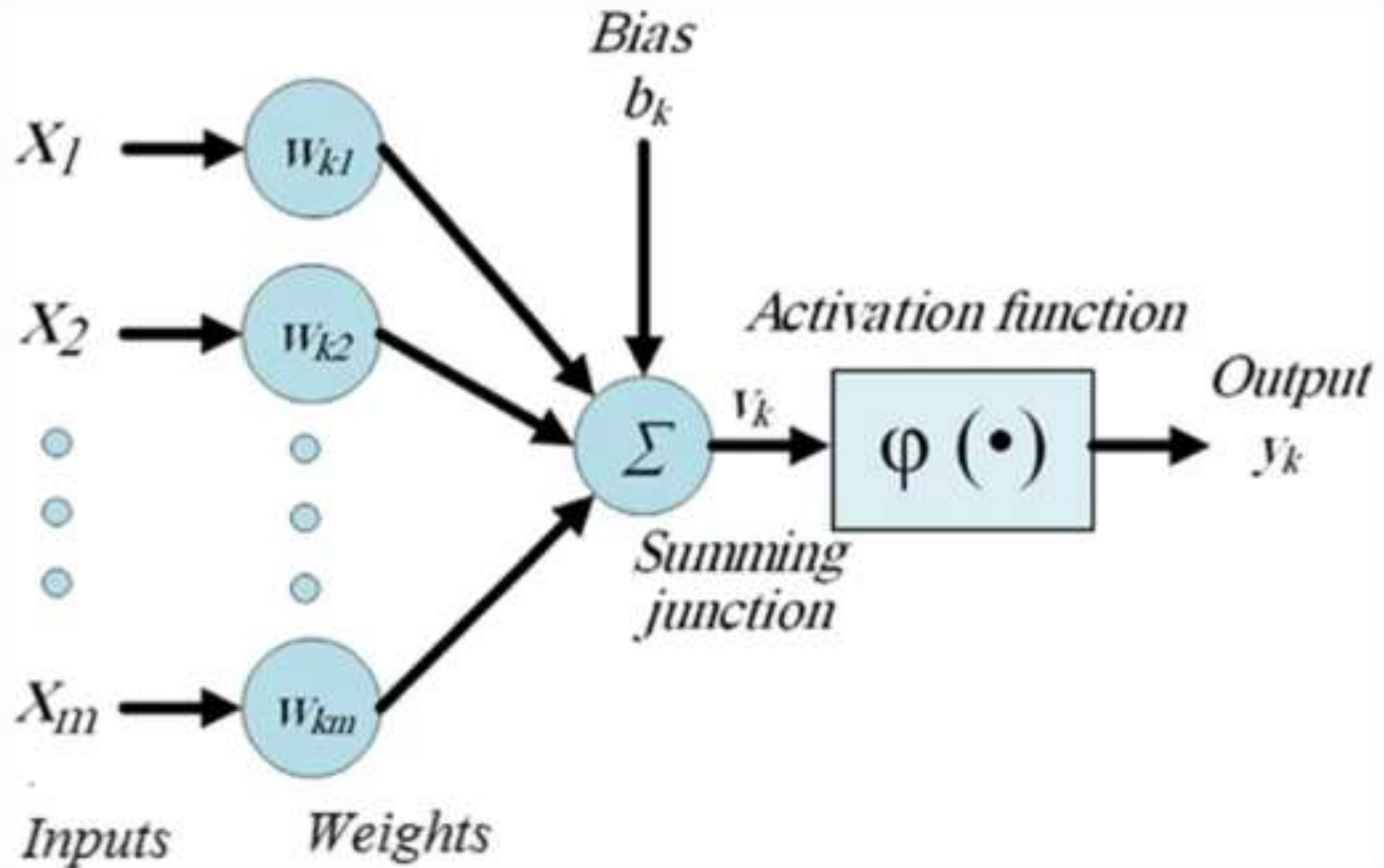
It is **able to process highly complex inputs.**

## Disadvantages :

- **Speed of processing is slow**
- 

## ARTIFICIAL NEURAL NETWORK :

- **Artificial neural network (ANN)** is a type of neural network which is based on a **feed-forward strategy**, because they **pass information through the nodes** continuously till it reaches the output node.
- This is **simplest type** of neural network.





# UNITS OF NEURAL NETWORK:

- **Nodes(units):** nodes represent a **cell of neural network**.
- **Links:** links are **directed arrows** that show **propagation of information from one node to another node**.
- **Weight:** each link has weight associated with it which determines **strength and sign of the connection**.


- **Activation function:** A **function** which is **used to derive output activation from the input activations to a given node** is called **activation function**.
- **Bias weight:** bias weight is **used to set the threshold for a unit**. Unit is activated when the weighted sum of real inputs exceeds the bias weight.



## Advantages of ANN :

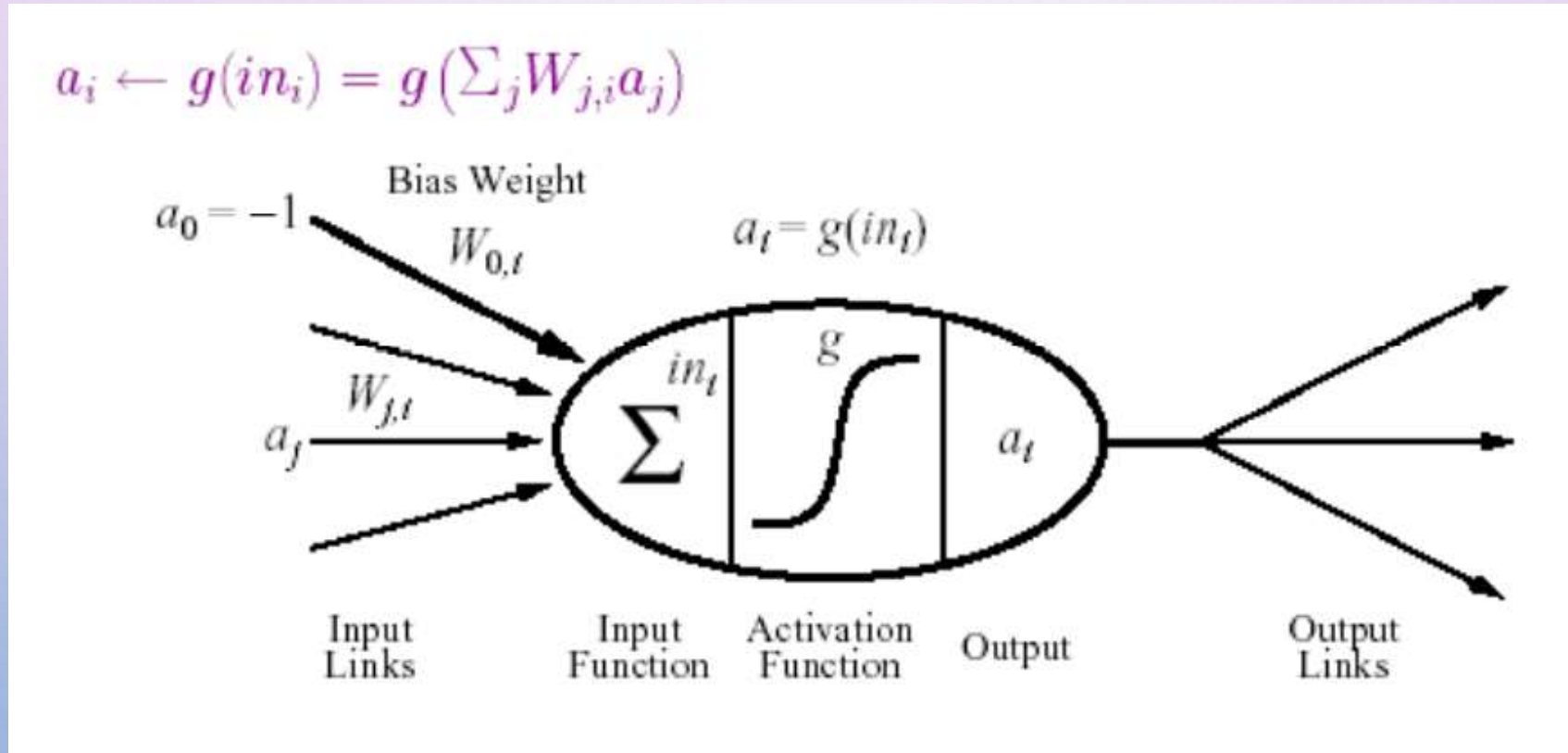
- Ability to **learn irrespective of the type of data** (linear or non-linear).
- ANN is highly **volatile** and serves **best in financial time series forecasting**.

## Disadvantages of ANN :

- The **simplest architecture** but it is **difficult to explain the behavior of the network**.
  - This network is **dependent on hardware**.
- 

# SIMPLE MODEL OF NEURAL NETWORK

- A simple **mathematical model** of neuron is **devised by McCulloch and pitt** is given in the figure given below:



- it fires when a linear combination of its inputs exceeds some threshold.**



- A neural network is **composed of nodes (units) connected by directed links**
- **A link from unit j to i** serve to **propagate the activation  $a_j$  from j to i.**
- Each link has some numeric **weight  $w_{j,i}$**  associated with it, which **determines strength and sign of connection.**
- Each unit first computes a **weighted sum of it's inputs**

$$in_i = \sum_{j=0}^n W_{j,i} a_j$$

- Then it applies **activation function g** to this **sum to derive the output:**

$$a_i = g(in_i) = g\left(\sum_{j=0}^n W_{j,i} a_j\right)$$

- Here,  **$a_j$  output activation from unit j** and  **$w_{j,i}$  is the weight on the link j to this node.**

# ACTIVATION FUNCTIONS

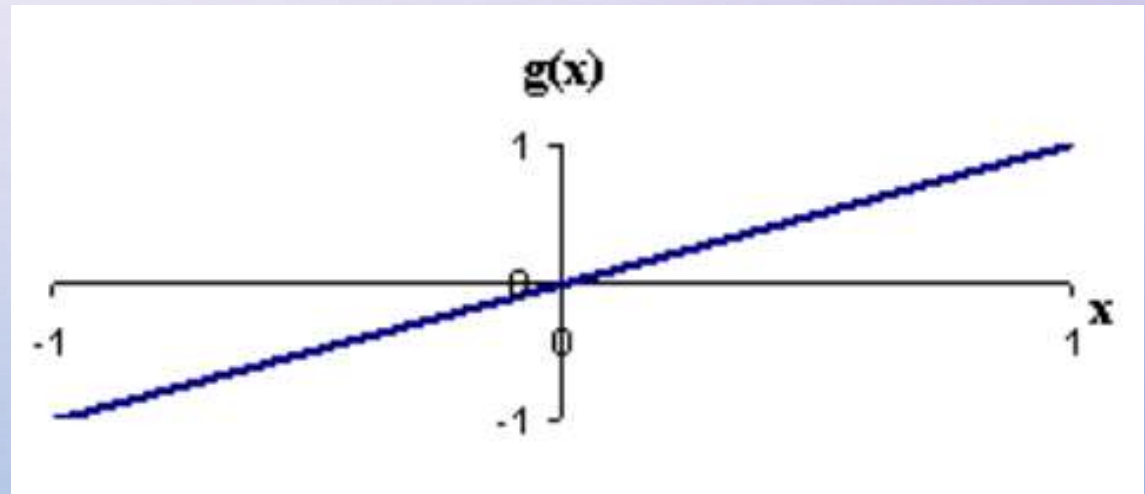
**ACTIVATION FUNCTION** TYPICALLY FALLS INTO ONE OF **THREE CATEGORIES**:

- ☐ **LINEAR**
- ☐ **THRESHOLD**
- ☐ **SIGMOID**
- ☐ **TANH**
- ☐ **SOFTMAX**
- ☐ **RELU**

# LINEAR ACTIVATION FUNCTIONS

- It is a **simple straight-line function** which is **directly proportional to the input** i.e. the **weighted sum of neurons**.
- It has the **equation:  $g(x) = kx$**

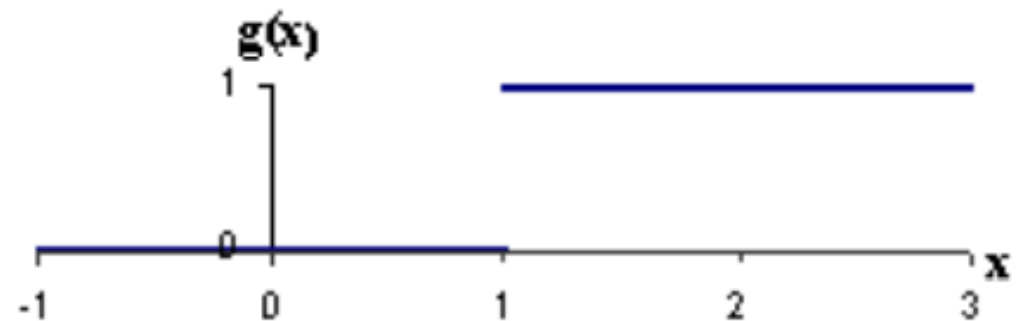
Where **k** is a constant.





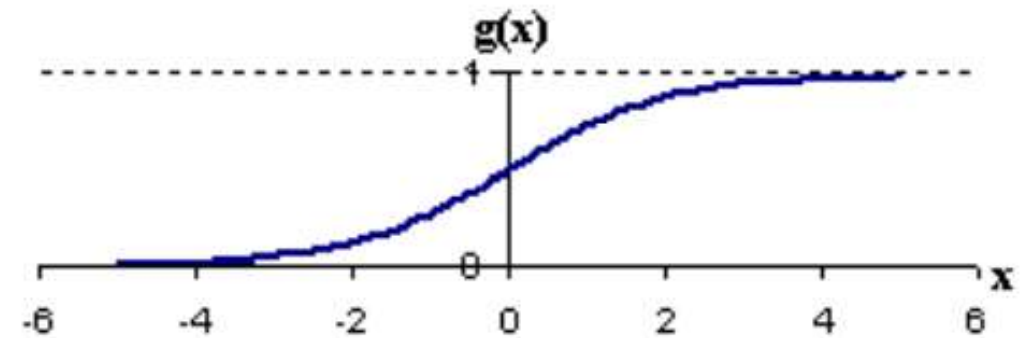
# Threshold activation functions

- For threshold activation functions, the **output are set at one of two levels**, depending on whether the **total input is greater or less than** some **threshold value**.
- **Function:-**  $G(x) = 1$  if  $x \geq k$   
 $= 0$  if  $x < k$



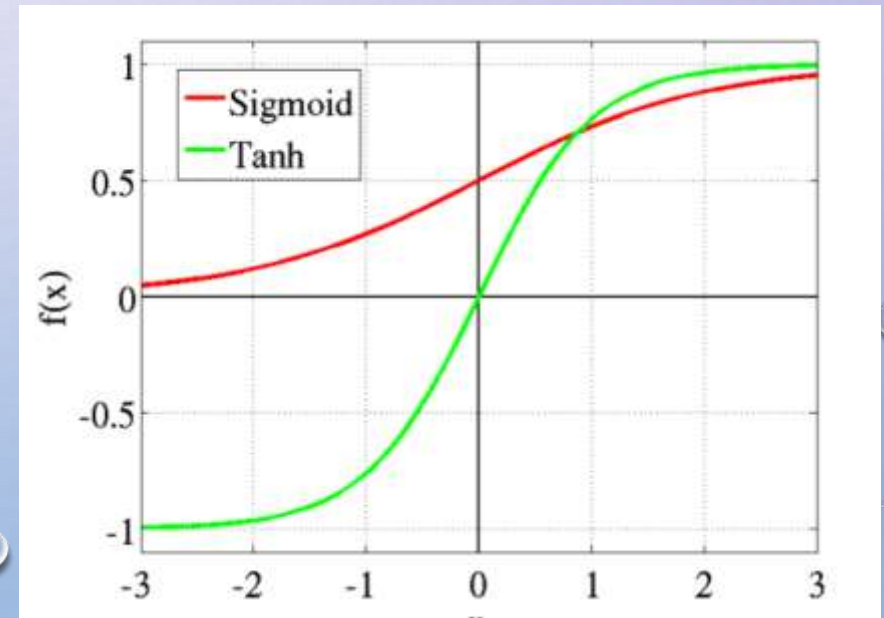
# SIGMOID ACTIVATION FUNCTIONS

- For sigmoid activation functions, the **output varies continuously but not linearly as the input changes.**
- Sigmoid units bear a **greater resemblance to real neurons** than do linear **or** threshold units.
- It has the **advantage of differentiable.**
- **Function:-**  $G(x) = 1 / (1 + e^{-x})$



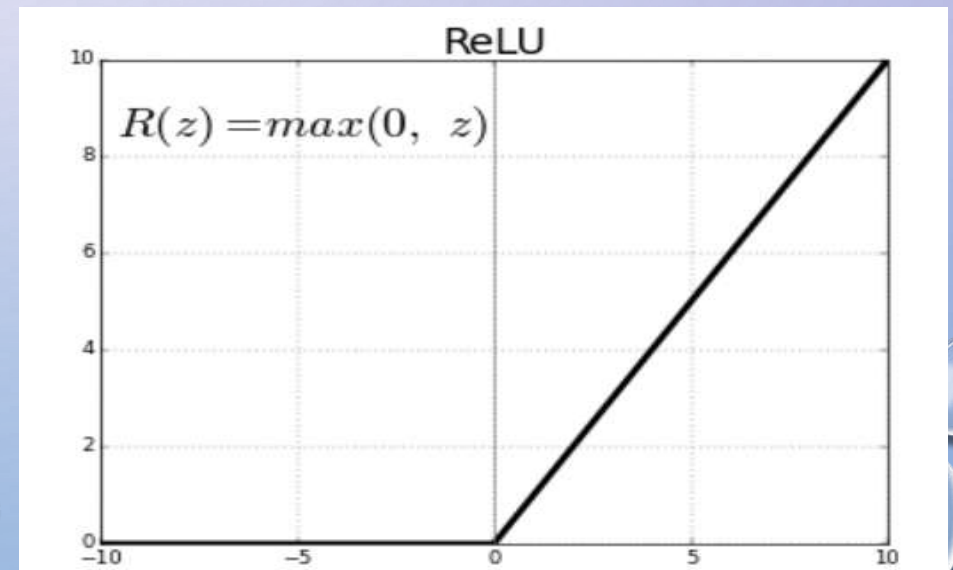
# TANH OR HYPERBOLIC TANGENT ACTIVATION FUNCTION

- tanh is also **like logistic sigmoid but better**. the **range of the tanh function** is from **(-1 to 1)**.
- tanh is **also sigmoidal (s - shaped)**.
- the **advantage** is that the **negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph**.



# RECTIFIED LINEAR UNIT (RELU) ACTIVATION FUNCTIONS

- The relu is the **most used activation function in the world right now**, since, it is **used in almost all the convolutional neural networks(CNN) or deep learning**.
- **$f(z)$  is zero when  $z$  is less than zero and  $f(z)$  is equal to  $z$  when  $z$  is above or equal to zero.**

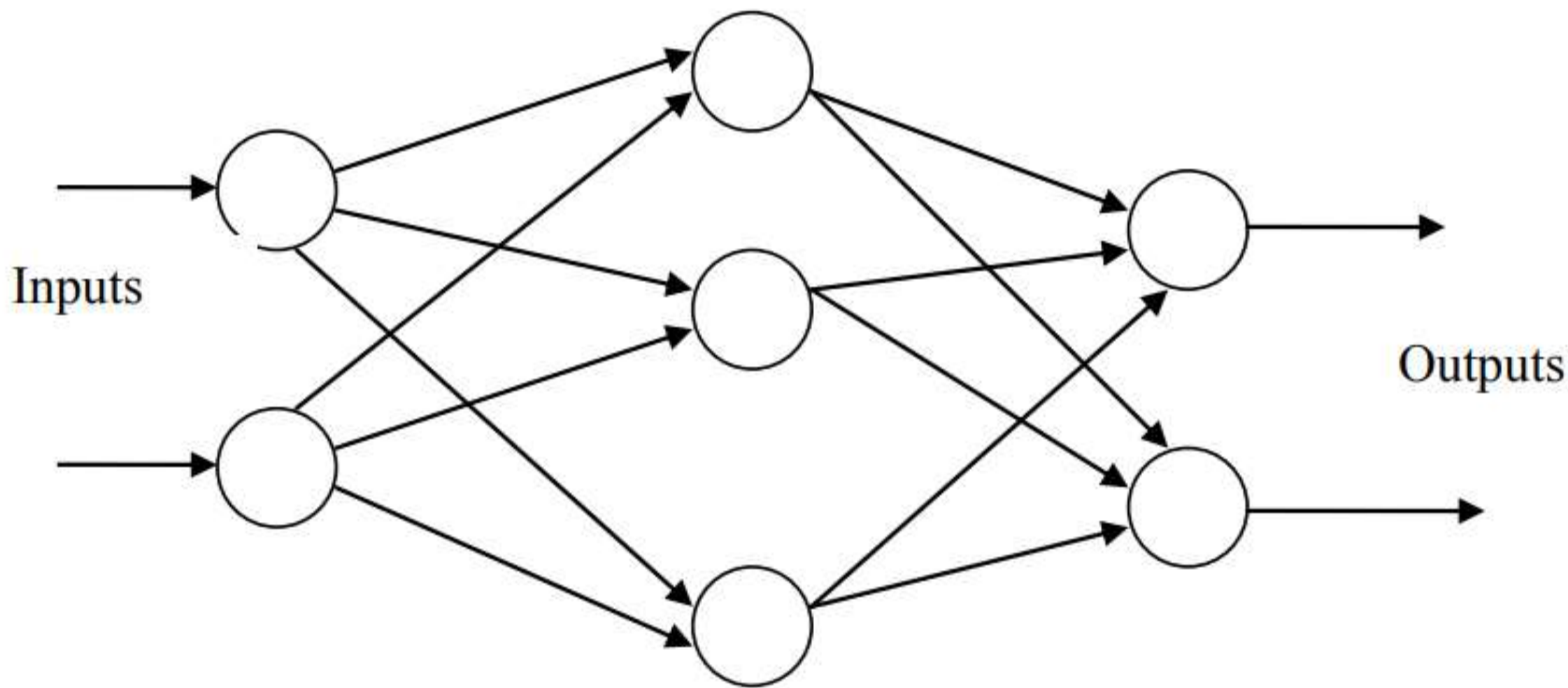




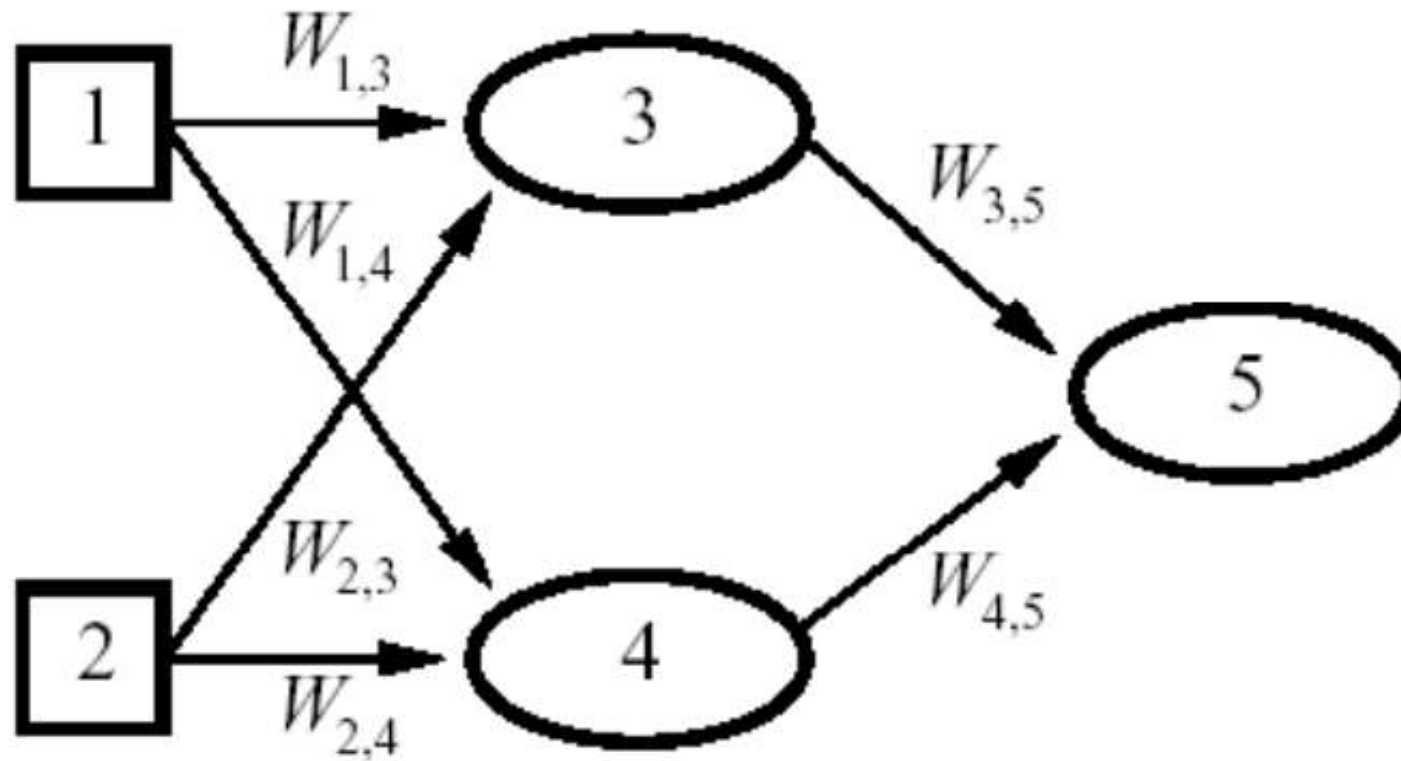
# NETWORK STRUCTURES

## Feed-forward networks:

- Feed-forward ANNs **allow signals to travel one way only**; from input to output.
- There is **no feedback (loops)** i.e. The **output of any layer does not affect that same layer**.
- They are **extensively used in pattern recognition**.



## Feed-forward example



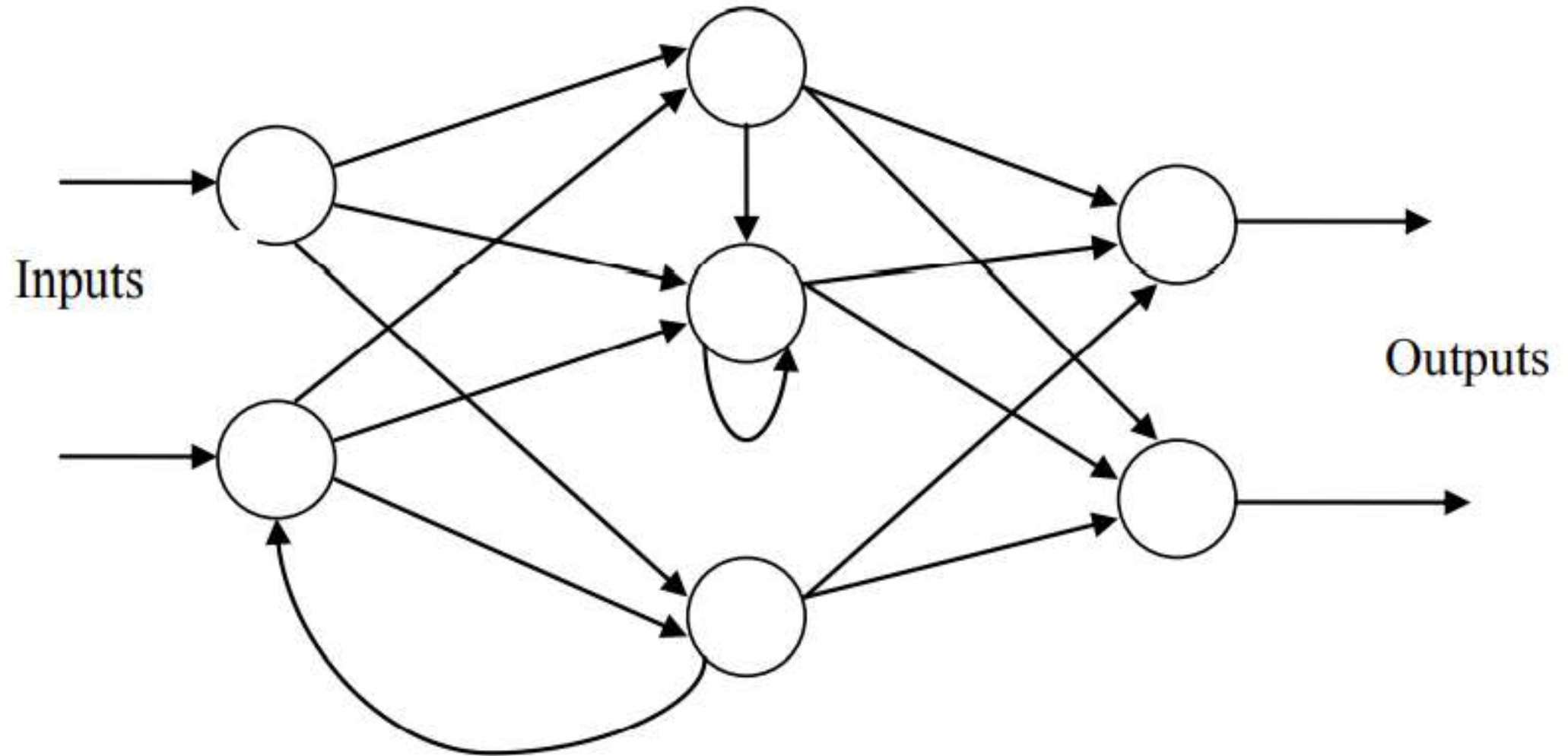
Here;

$$a_5 = g(W_{3,5} a_3 + W_{4,5} a_4)$$

$$= g(W_{3,5} g(W_{1,3} a_1 + W_{2,3} a_2) + W_{4,5} g(W_{1,4} a_1 + W_{2,4} a_2))$$

# FEEDBACK NETWORKS (RECURRENT NETWORKS):

- Feedback network **can have signals traveling in both directions** by introducing **loops in the network**.
- Works are very **powerful** and **can get extremely complicated**.
- Feedback networks are **dynamic**; their **'state'** is **changing continuously** until they **reach an equilibrium point**.
- They **remain at the equilibrium point until** the input changes and a new equilibrium needs to be found.
- Also referred to as **interactive or recurrent**.

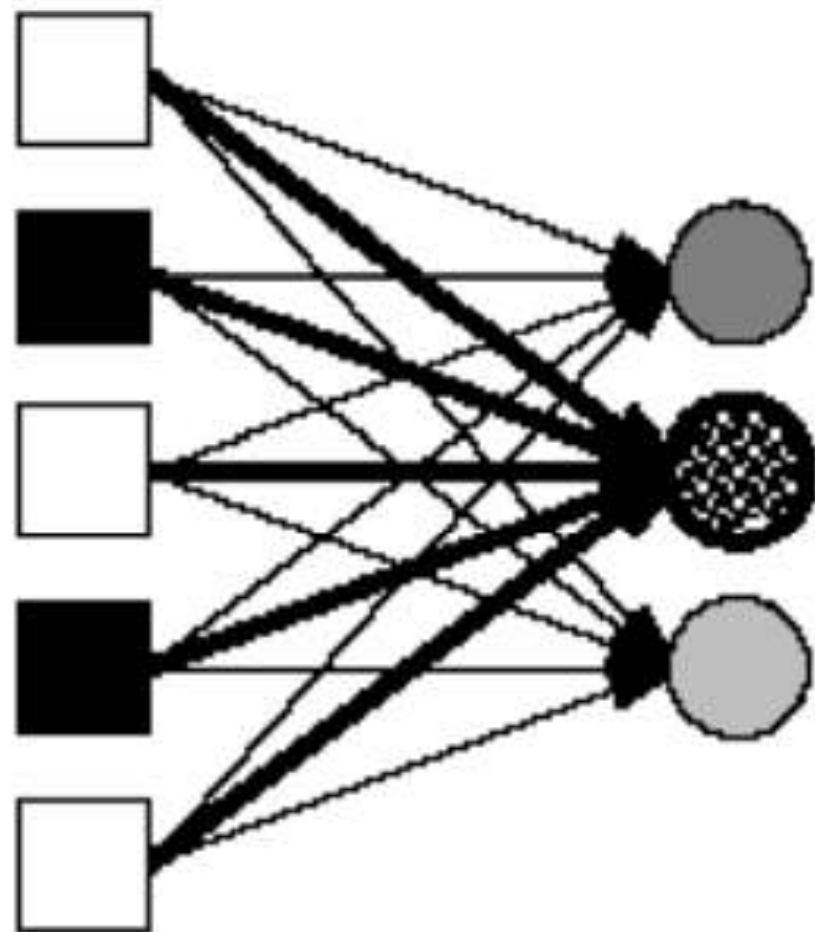




# TYPES OF FEED FORWARD NEURAL NETWORK:

## 1] Single-layer neural networks (perceptron)

- A neural network in which **all the inputs connected directly to the outputs** is called a **single-layer neural network**, or a **perceptron network**.
- Since each **output unit is independent of the others i.e each weight affects only one of the outputs.**



*Input  
Units*

$W_{j,i}$

*Output  
Units*

## 2] MULTILAYER NEURAL NETWORKS (PERCEPTRON)

- The neural network **which contains input layers, output layers and some hidden layers** is called multilayer neural network.
- The **advantage of adding hidden layers** is that **it enlarges the space of hypothesis**.
- **Layers of the network are normally fully connected.**

Output units

$a_l$

$W_{j,l}$

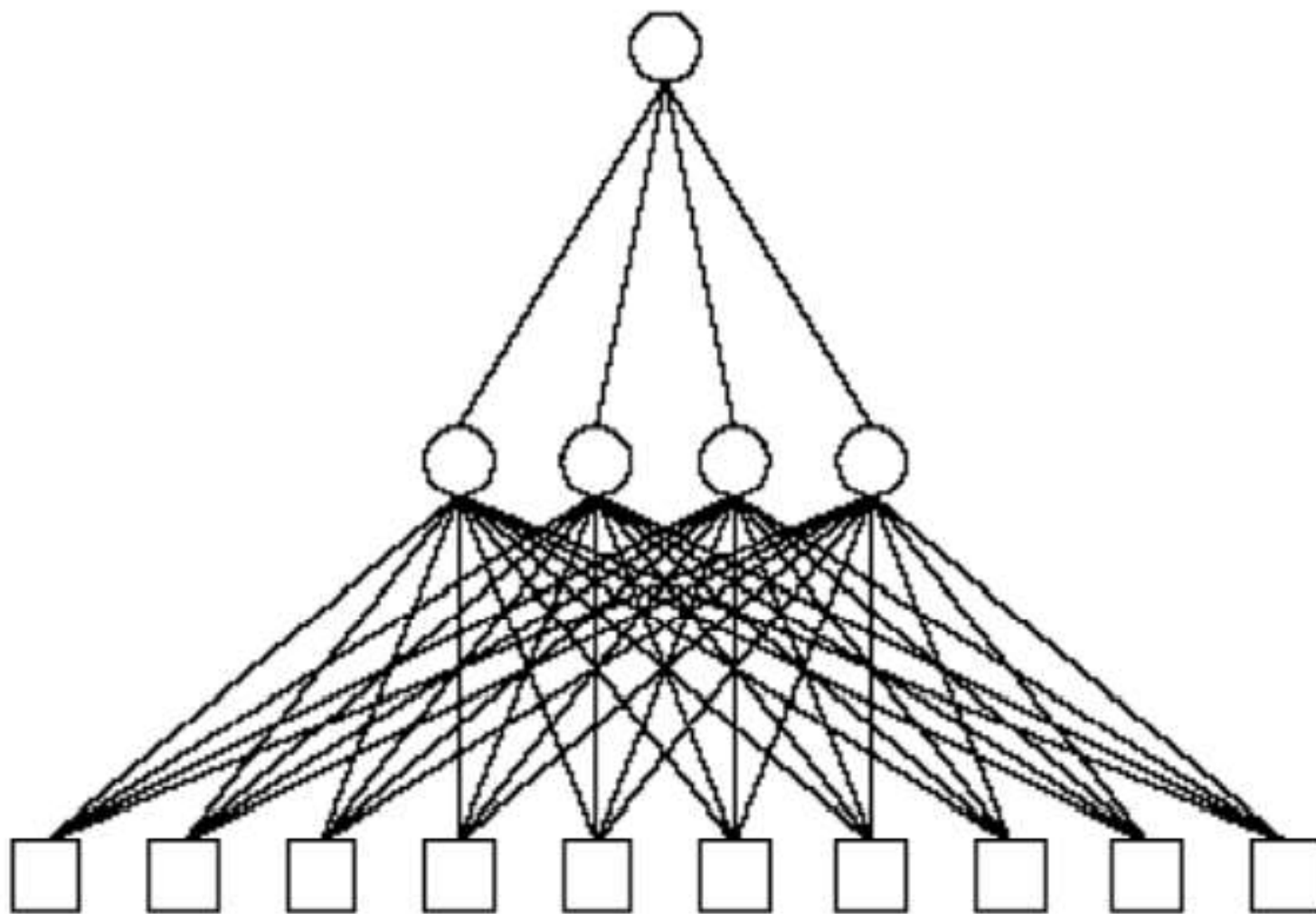
Hidden units

$a_j$

$W_{k,j}$

Input units

$a_k$



- Once the number of layers, and number of units in each layer, has been selected, **training** is used to set the network's weights and thresholds so as to minimize the prediction error made by the network
- **Training** is the process of adjusting weights and threshold to produce the desired result for different set of data.



# LEARNING IN NEURAL NETWORKS

- **Learning**: one of the powerful features of neural networks
- **Learning in neural networks** is carried out by **adjusting the connection weights** among neurons.
- It **is similar to a biological nervous system** in which learning is carried out by changing **synapses connection strengths**, among cells.

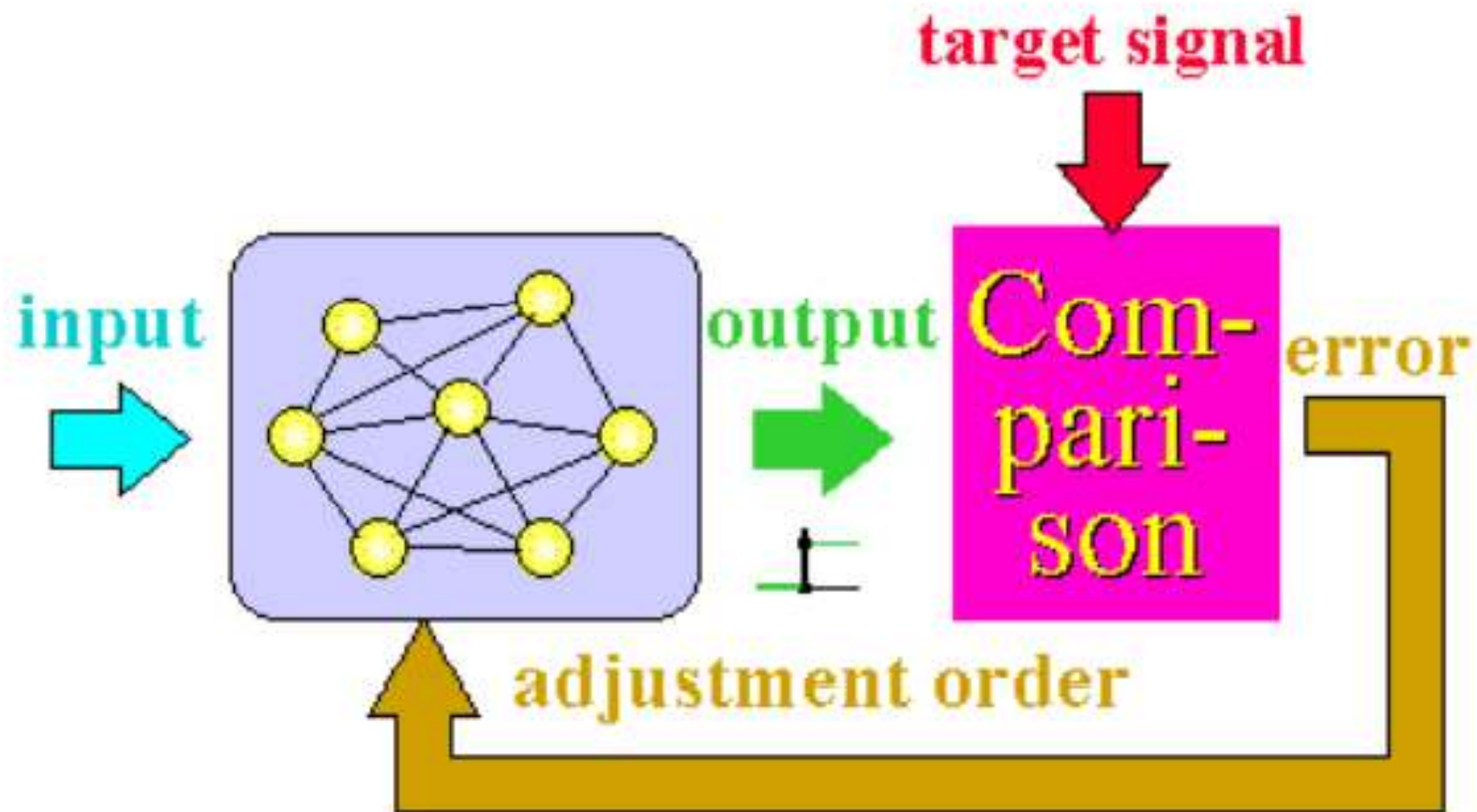
- The **operation of a neural network** is determined by the values of the **interconnection weights**.
- There is **no algorithm that determines how the weights should be assigned** in order to solve specific problems. Hence, the **weights are determined by a learning process**
- Learning may be **classified into two categories:**
  - (1) supervised learning**
  - (2) unsupervised learning**

## **SUPERVISED LEARNING:**

- In supervised learning, the **network is presented with inputs together with the target (teacher signal) outputs.**
- Then, the **neural network tries to produce an output as close as possible to the target signal by adjusting the values of internal weights.**

- The most common supervised learning method is the “**error correction method**”.
- Error correction method is **used for networks which their neurons have discrete output functions.**
- **Neural networks are trained with this method in order to reduce the error** (difference between the network's output and the desired output) **to zero**



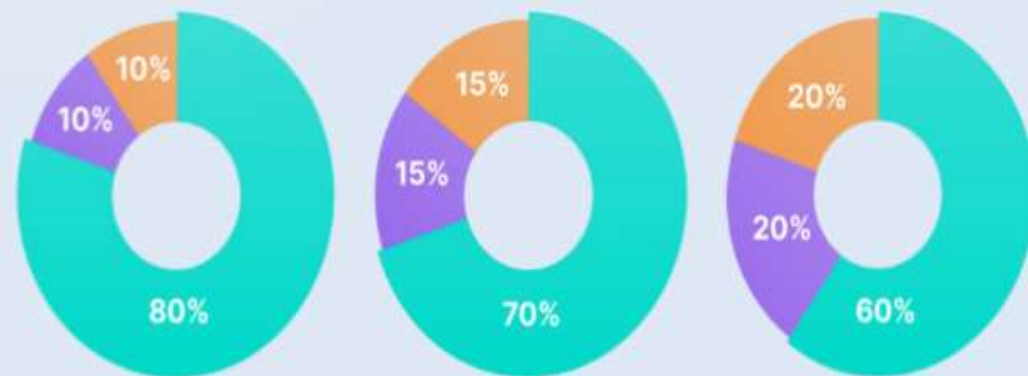




Student	Test1 marks	Test2 Marks	Study hours	Final result
1	30	35	4	Pass
2	42	45	6	Pass
3	20	17	1	Fail
4	45	48	6	Pass
5	25	22	2	Pass
6	34	40	2	Pass
7	49	47	6	Pass
8	17	10	0	Fail
9	25	20	1	Fail
10	35	38	3	Pass

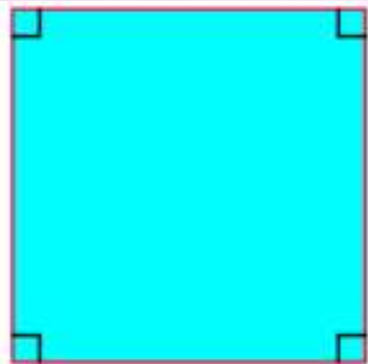
## Training Data Needs

● Training data
 ● Validation data
 ● Test data

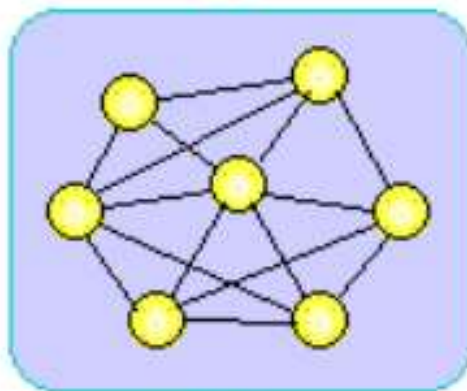


# UNSUPERVISED LEARNING:

- In unsupervised learning, there is **no teacher (target signal)** from outside and the network **adjusts its weights in response to only the input patterns.**
- A **typical example** of unsupervised learning is **hebbian learning.**



input



It has 4  
straight  
lines.

output



"It's a rectangular."

- Consider a machine (or living organism) which receives some **sequence of inputs  $x_1, x_2, x_3, \dots$** , where  **$x_t$  is the sensory input at time  $t$** .
- **In supervised learning** the machine is given a sequence of input & a **sequence of desired outputs  $y_1, y_2, \dots$** , and the **goal of the machine is to learn to produce the correct output given a new input**



- While, **in unsupervised learning** the machine simply receives inputs  $x_1, x_2, \dots$ , but obtains **neither supervised target outputs, nor rewards from its environment.**
- It may seem somewhat **mysterious** to imagine what the machine could possibly learn given that it doesn't get any feedback from its environment
- Unsupervised learning can be thought of as **finding patterns** in the data above and beyond what would be considered pure unstructured noise.



# HEBBIAN LEARNING:

- The **oldest and most famous** of all learning rules is hebb's postulate of learning:
- **When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells** such that **efficiency** as one of the cells firing B is increased

- The **weight between two neurons increases** if the two neurons activate **simultaneously** and **reduces** if they activate separately.
- **Nodes that tend to be either both positive or both negative at the same time** have **strong positive weights**, while those that tend to be opposite have **strong negative weights**.

# HEBB'S ALGORITHM

- given a training input  $s$  with its target output  $t$

- **Step 0: initialize all weights to 0**

- **Step 1: set the activations of the input units:**

$$x_i = s_i$$

- **Step 2: set the activation of the output unit to the target value:**

$$y = t$$

- **Step 3: adjust the weights:**

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

- **Step 4: adjust the bias (just like the weights)**

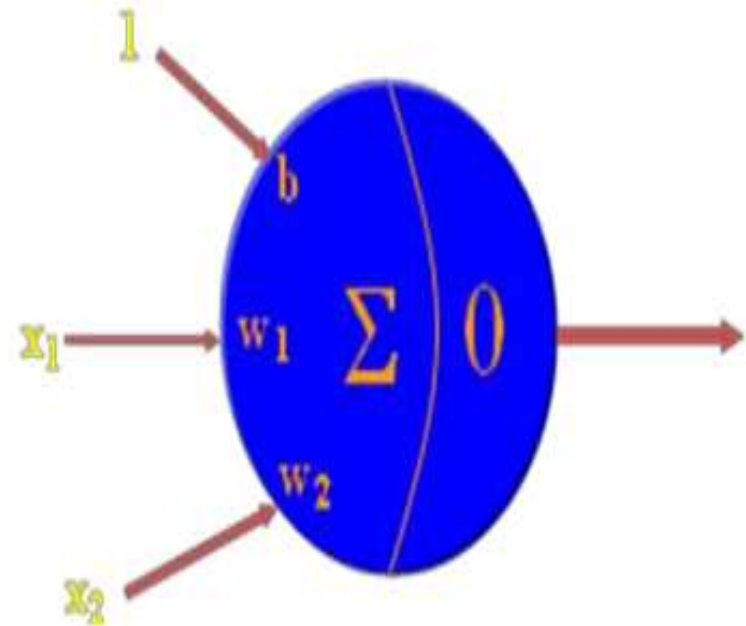
$$b(\text{new}) = b(\text{old}) + y$$

### Example:

**PROBLEM:** Construct a Hebb Net which performs like an AND function, that is, only when both features are “active” will the data be in the target class.

**TRAINING SET** (with the bias input always at 1):

x1	x2	bias	Target
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

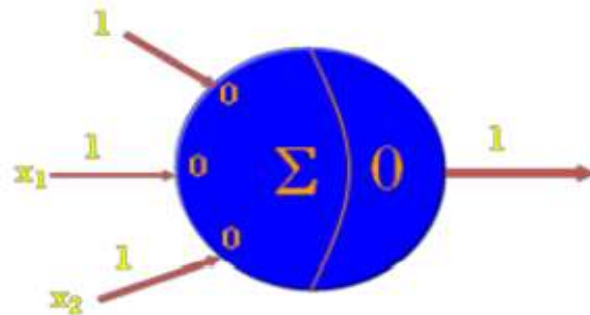




## Training-First Input:

- Initialize the weights to 0

**Present the first input:  
(1 1 1) with a target of 1**

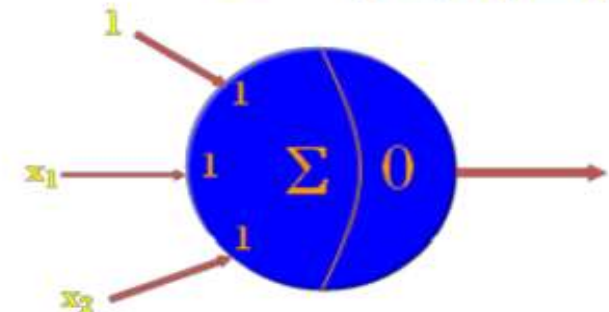


**Update the weights:**

$$\begin{aligned} w_1(\text{new}) &= w_1(\text{old}) + x_1 t \\ &= 0 + 1 = 1 \end{aligned}$$

$$\begin{aligned} w_2(\text{new}) &= w_2(\text{old}) + x_2 t \\ &= 0 + 1 = 1 \end{aligned}$$

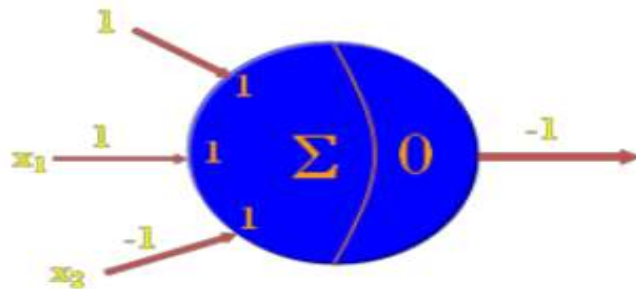
$$\begin{aligned} b(\text{new}) &= b(\text{old}) + t \\ &= 0 + 1 = 1 \end{aligned}$$





## Training- Second Input:

- **Present the second input:  
(1 -1 1) with a target of -1**

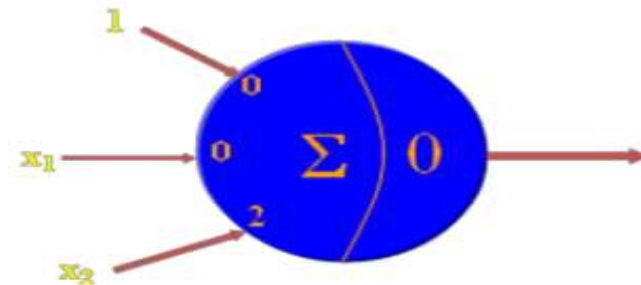


**Update the weights:**

$$\begin{aligned} w_1(\text{new}) &= w_1(\text{old}) + x_1 t \\ &= 1 + 1(-1) = 0 \end{aligned}$$

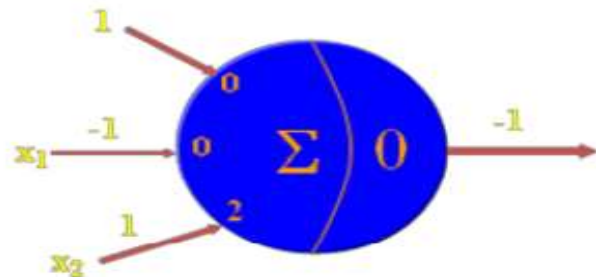
$$\begin{aligned} w_2(\text{new}) &= w_2(\text{old}) + x_2 t \\ &= 1 + (-1)(-1) = 2 \end{aligned}$$

$$\begin{aligned} b(\text{new}) &= b(\text{old}) + t \\ &= 1 + (-1) = 0 \end{aligned}$$



### Training- Third Input:

- **Present the third input:**  
 **$(-1 \ 1 \ 1)$  with a target of  $-1$**

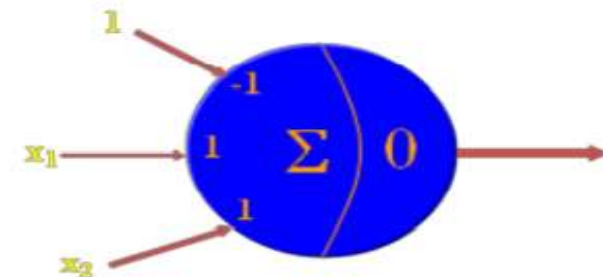


### **Update the weights:**

$$\begin{aligned} w_1(\text{new}) &= w_1(\text{old}) + x_1 t \\ &= 0 + (-1)(-1) = 1 \end{aligned}$$

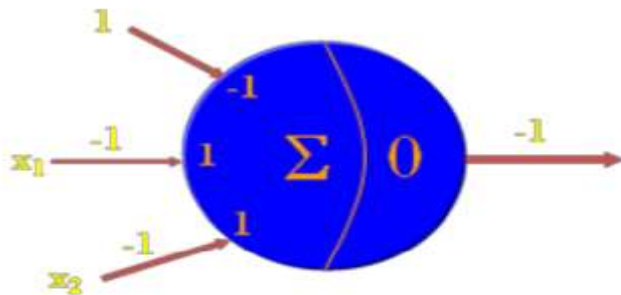
$$\begin{aligned} w_2(\text{new}) &= w_2(\text{old}) + x_2 t \\ &= 2 + 1(-1) = 1 \end{aligned}$$

$$\begin{aligned} b(\text{new}) &= b(\text{old}) + t \\ &= 0 + (-1) = -1 \end{aligned}$$



### Training- Fourth Input:

- **Present the fourth input:  
(-1 -1 1) with a target of -1**

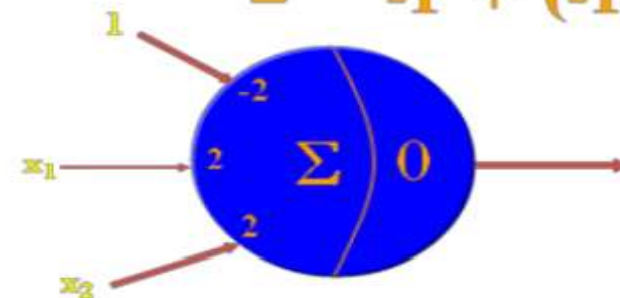


### **Update the weights:**

$$\begin{aligned} w_1(\text{new}) &= w_1(\text{old}) + x_1 t \\ &= 1 + (-1)(-1) = 2 \end{aligned}$$

$$\begin{aligned} w_2(\text{new}) &= w_2(\text{old}) + x_2 t \\ &= 1 + (-1)(-1) = 1 \end{aligned}$$

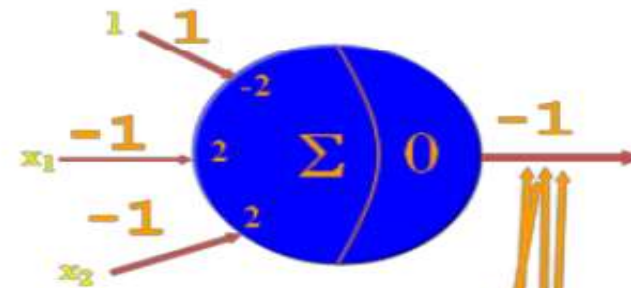
$$\begin{aligned} b(\text{new}) &= b(\text{old}) + t \\ &= -1 + (-1) = -2 \end{aligned}$$



## Final Neuron:

- This neuron works:

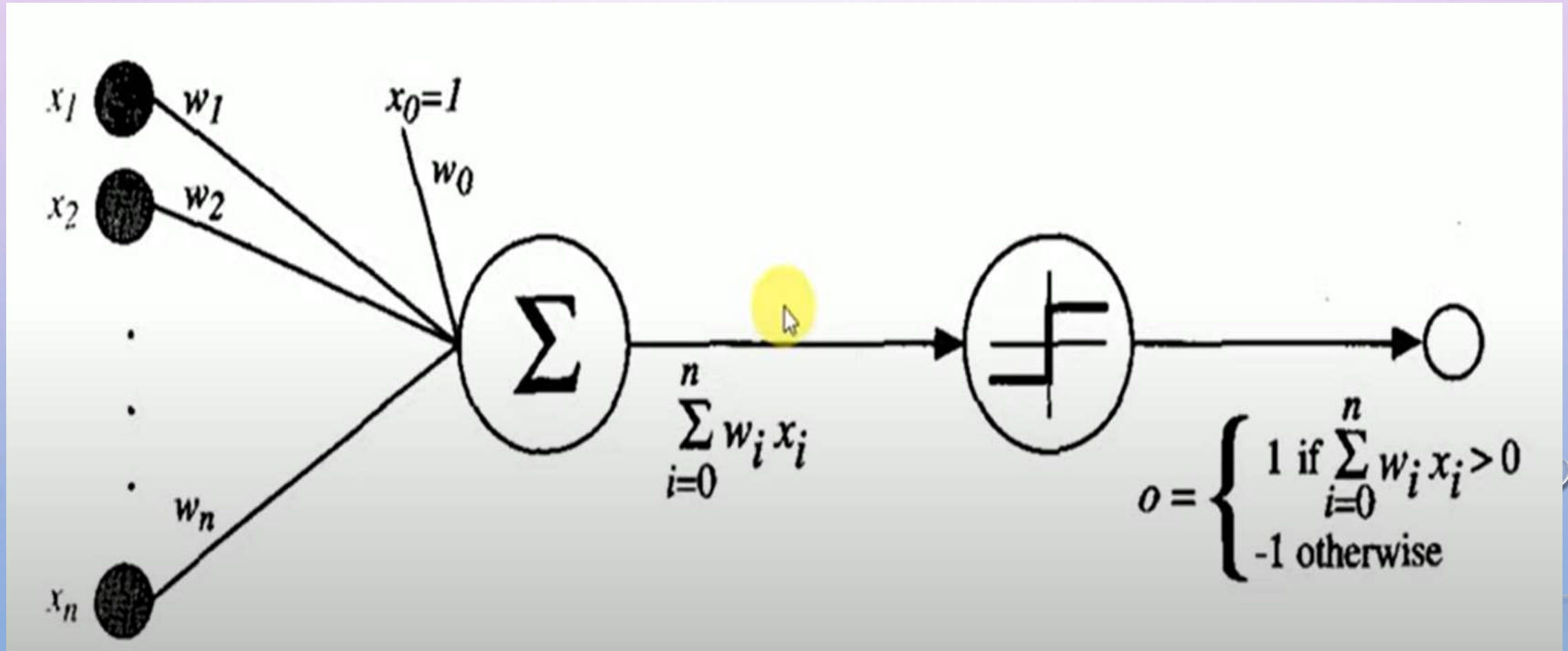
x1	x2	bias	Target
1	1	1	1
1	-1	1	-1
-1	1	1	1
-1	-1	1	-1



$$\begin{aligned} 1 * 2 + 1 * 2 + 1 * (-2) &= 2 > 0 \\ (-1) * 2 + 1 * 2 + 1 * (-2) &= -2 < 0 \\ 1 * 2 + (-1) * 2 + 1 * (-2) &= -2 < 0 \\ (-1) * 2 + (-1) * 2 + 1 * (-2) &= -6 < 0 \end{aligned}$$



# PERCEPTRON LEARNING THEORY:






# PERCEPTRON LEARNING THEORY:

- The term **"perceptrons"** was **coined by Frank Rosenblatt in 1962** and is **used to describe the connection of simple neurons into networks.**
- These networks are **simplified versions of the real nervous system** where **some properties are exaggerated** and **others are ignored.**
- For the moment **we will concentrate on single layer perceptrons.**



So how can we achieve learning in our model neuron?

- We **need to train them** so they can do things that are useful.
  - To do this we must **allow the neuron to learn from its mistakes.**
  - A learning paradigm that achieves this, is known as **supervised learning** and works in the following manner.
- 

- **Step-I.** Set the **weight and thresholds** of the neuron to random values.
- **Step-II.** Present an **input**.
- **Step-III.** **Calculate the output** of the neuron.
- **Step-IV.** **Alter the weights to reinforce correct decisions and discourage wrong decisions**, hence **reducing the error**. So **for the network to learn** we shall **increase the weights** on the **active inputs** when we want the **output to be active**, and **to decrease** them when we want the **output to be inactive**.
- **Step-V.** Now **present the next input** and **repeat steps iii. - V.**

# Perceptron learning algorithm:

- The algorithm for perceptron learning is based on the supervised learning procedure discussed previously.

## Algorithm:

**Step i:- Initialize weights and threshold.**

Set  $w_i(t)$ , ( $0 \leq i \leq n$ ), to be the weight  $i$  at time  $t$ , and  $\theta$  to be the threshold value in the output node. Set  $w_0$  to be  $-\theta$ , the bias, and  $x_0$  to be always 1. Set  $w_i(t)$  to small random values, thus initializing the weights and threshold.

**Step ii:- Present input and desired output**

**present input**  $x_0, x_1, x_2, \dots, x_n$  and **desired output**  $d(t)$



**Step iii:- Calculate the actual output**

$$y(t) = g [w_0(t)x_0(t) + w_1(t)x_1(t) + \dots + w_n(t)x_n(t)]$$

**Step iv:- Adapts weights**

$W_i(t+1) = w_i(t) + \alpha[d(t) - y(t)]x_i(t)$  , where  $0 \leq \alpha \leq 1$  (**learning rate**) is a **positive gain function that controls the adaption rate.**

- **Step v:- Steps iii. And iv are repeated until the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed.**
- Please **note** that the **weights only change if an error is made and hence this is only when learning shall occur**



# PERCEPTRON TRAINING RULE – ANN

*Perceptron\_training\_rule* ( $X, \eta$ )

initialize  $\mathbf{w}$  ( $w_i \leftarrow$  an initial (small) random value)

repeat

    for each training instance  $(\mathbf{x}, \mathbf{tx}) \in X$

        compute the real output  $\mathbf{ox} = \text{Activation}(\text{Summation}(\mathbf{w} \cdot \mathbf{x}))$

        if  $(\mathbf{tx} \neq \mathbf{ox})$

            for each  $w_i$

$w_i \leftarrow w_i + \Delta w_i$

$\Delta w_i \leftarrow \eta (\mathbf{tx} - \mathbf{ox}) x_i$

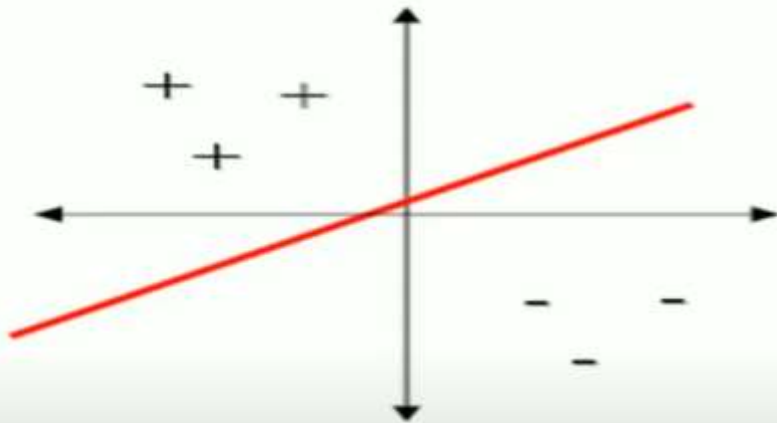
            end for

        end if

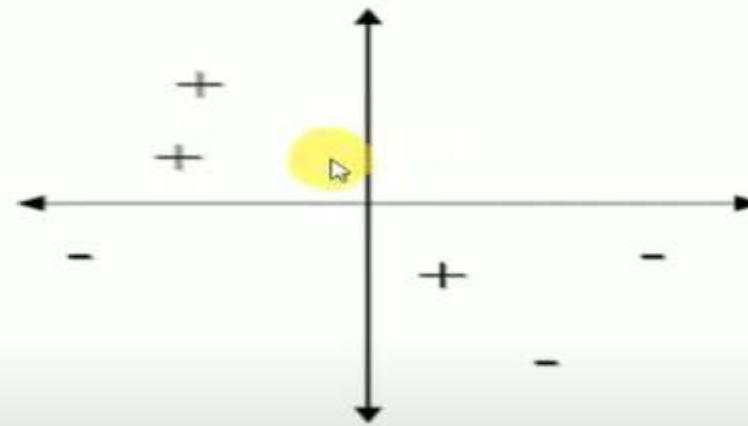
    end for

until all the training instances in  $X$  are correctly classified

return  $\mathbf{w}$



Linearly separable



Non-linearly separable

- Perceptron rule finds a successful weight vector when the training examples are linearly separable, but it can fail to converge if the examples are not linearly separable.
- A second training rule, called the **delta rule**, is designed to overcome this difficulty.
- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use **gradient descent** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.
- This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units.

## DELTA RULE:

- The delta rule is a **gradient descent learning rule** for updating the weights of the artificial neurons in a **single-layer perceptron**.
- It is a **special case** of the more general **backpropagation algorithm**.
- For a neuron 'J' with activation function  $g(x)$  the **delta rule** for 'J' th weight  $W_{j,i}$  is given by

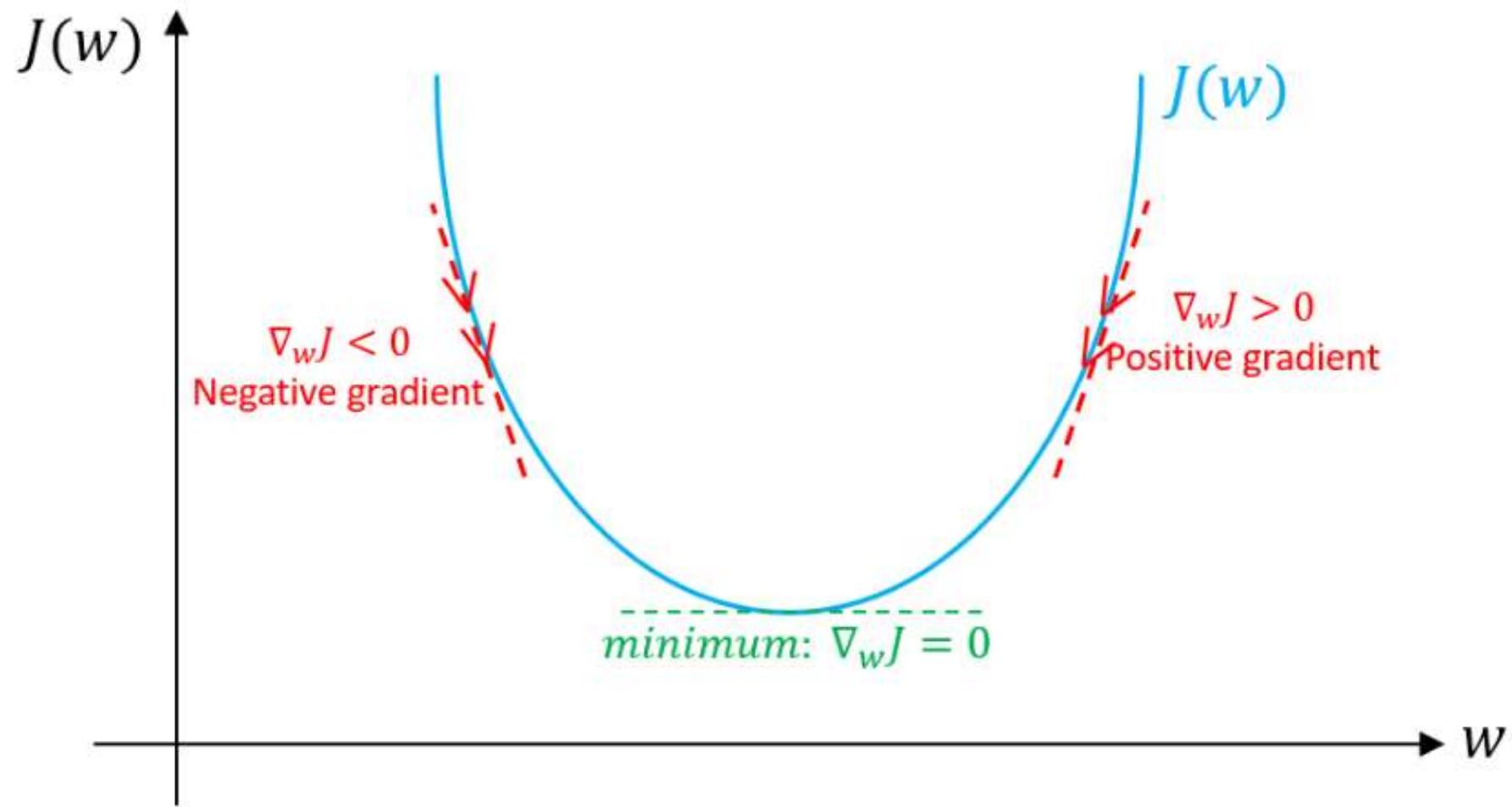
$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i,$$



where  $\alpha$  is a small constant called *learning rate*,  $g(x)$  is the neuron's activation function,  $t_j$  is the target output,  $h_j$  is the weighted sum of the neuron's inputs,  $y_j$  is the actual output, and  $x_i$  is the  $i$ th input. It holds  $h_j = \sum x_i w_{ji}$  and  $y_j = g(h_j)$ .

The delta rule is commonly stated in simplified form for a perceptron with a linear activation function as

$$\Delta w_{ji} = \alpha(t_j - y_j)x_i$$





- This vector derivative is called the *gradient* of  $E$  with respect to  $\vec{w}$ , written *as*  $\nabla E(\vec{w})$ .

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

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

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

where

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

# GRADIENT DESCENT ALGORITHM

- Gradient Descent and the Delta Rule is used separate the Non-Linearly Separable data.
- Weights are updated using the following rule,

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

- Where,

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

# GRADIENT DESCENT ALGORITHM

GRADIENT-DESCENT(*training\_examples*,  $\eta$ )

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where  $\vec{x}$  is the vector of input values, and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).*

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $\langle \vec{x}, t \rangle$  in *training\_examples*, Do
    - \* Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - \* For each linear unit weight  $w_i$ , Do

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

- For each linear unit weight  $w_i$ , Do

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

The key practical difficulties in applying gradient descent are

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



# BACKPROPAGATION

- It is a **supervised learning method**, and is an implementation of the **delta rule**
- The term is an **abbreviation** for "**backwards propagation of errors**".
- Backpropagation **requires** that the **activation function** used by the **artificial neurons** (or "nodes") is **differentiable**
- As the algorithm's name implies, the **errors** (or learning) **propagate backwards from the output nodes to the inner nodes**

- This **gradient** is **almost always used in a simple stochastic gradient descent algorithm**, is a general **optimization algorithm**, but is typically used to fit the parameters of a machine learning model, to find weights that minimize the error.

- Backpropagation is used to calculate the **gradient of the error** of the network with respect to the network's **modifiable weights**
- Backpropagation usually allows **quick convergence on satisfactory local minima** for error in the kind of networks to which it is suited.
- Backpropagation networks are necessarily **multilayer perceptrons** (usually with one input, one hidden, and one output layer).
- In order for the hidden layer to serve any useful function, **multilayer networks must have non-linear activation functions for the multiple layers**: a multilayer network using only linear activation functions is equivalent to some single layer, linear network.

## **SUMMARY OF THE BACKPROPAGATION TECHNIQUE:**

- **1. Present a training sample to the neural network.**
- **2. Compare the network's output** to the desired output from that sample.  
**Calculate the error in each output neuron.**
- **3. For each neuron, calculate what the output should have been, and a scaling factor, how much lower or higher the output must be adjusted to match the desired output. This is the local error.**



- 4. **Adjust the weights of each neuron to lower the local error.**
- 5. **Assign "blame" for the local error to neurons at the previous level**, giving greater responsibility to neurons connected by stronger weights.
- 6. **Repeat from step 3 on the neurons at the previous level**, using each one's "blame" as its error.

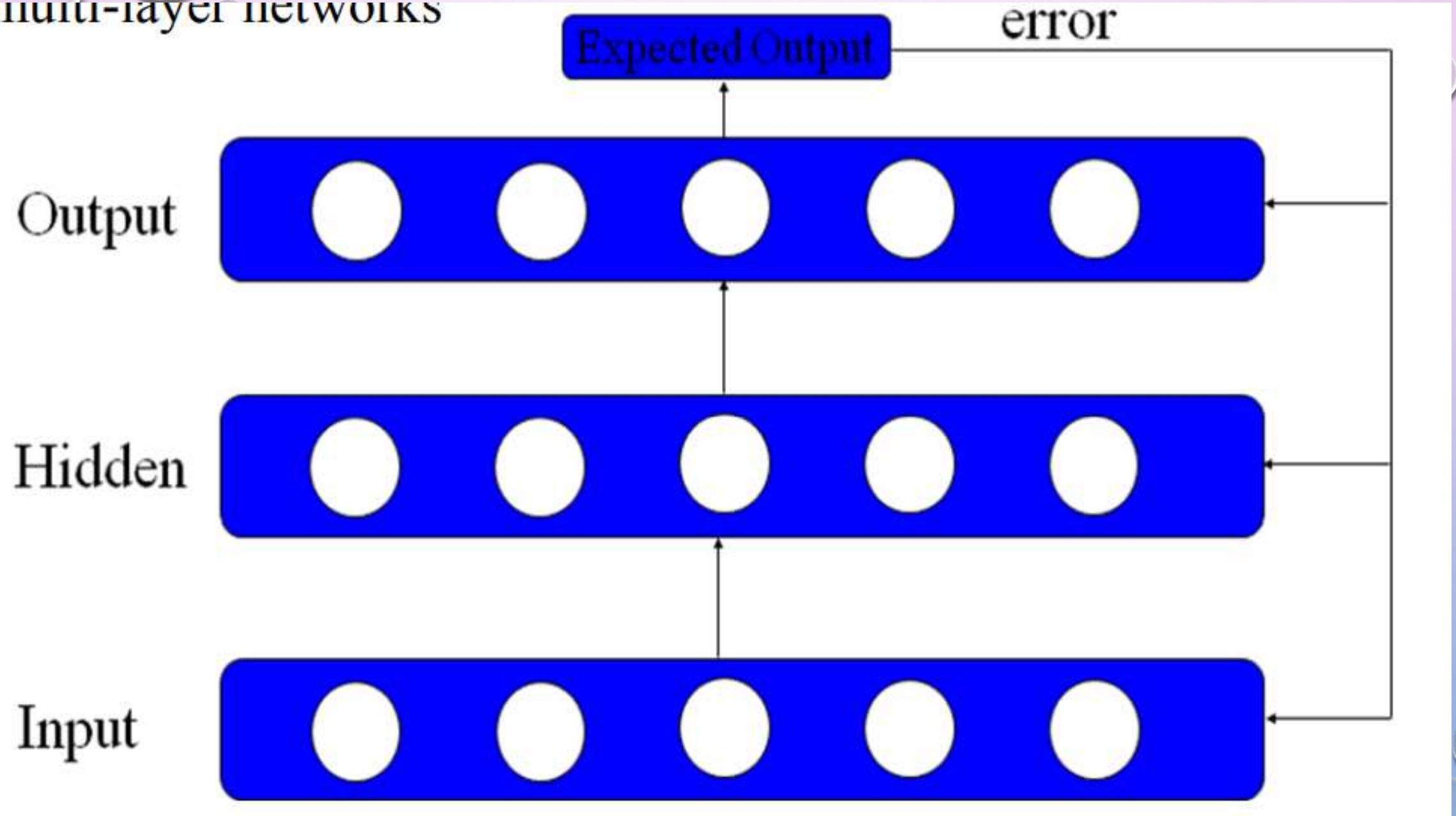
# Characteristics:

- A multi-layered perceptron **has three distinctive characteristics**
  - the **network contains one or more layers of hidden neurons**
  - the **network exhibits a high degree of connectivity**
    - **each neuron has a smooth (differentiable everywhere) nonlinear activation function,**  
the most common is the **sigmoidal nonlinearity**:

$$y_j = \frac{1}{1 + e^{s_j}}$$

The backpropagation algorithm **provides a computational efficient method for training multi-layer networks**

# multi-layer networks



## Algorithm:

**Step 0:** Initialize the weights to small random values

**Step 1:** Feed the training sample through the network and determine the final output

**Step 2:** Compute the error for each output unit, for unit k it is:

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

Diagram illustrating the error calculation for unit k:

- $t_k$  is labeled "Required output" (with an arrow pointing to  $t_k$ )
- $y_k$  is labeled "Actual output" (with an arrow pointing to  $y_k$ )
- $f'(y_{in_k})$  is labeled "Derivative of f" (with an arrow pointing to  $f'$ )

**Step 3:** Calculate the weight correction term for each output unit, for unit k it is:

$$\Delta w_{jk} = \alpha \delta_k z_j$$

Diagram illustrating the weight correction term calculation:

- $\alpha$  is labeled "A small constant" (with an arrow pointing to  $\alpha$ )
- $\delta_k$  is labeled "Hidden layer signal" (with an arrow pointing to  $\delta_k$ )
- $z_j$  is labeled "Hidden layer signal" (with an arrow pointing to  $z_j$ )



**Step 4:** Propagate the delta terms (errors) back through the weights of the hidden units where the delta input for the  $j^{\text{th}}$  hidden unit is:

$$\delta_{\text{in}_j} = \sum_{k=1}^m \delta_k w_{jk}$$

The delta term for  $j^{\text{th}}$  hidden unit is:  $\delta_j = \delta_{\text{in}_j} f'(z_{\text{in}_j})$

**Step 5:** Calculate the weight correction term for the hidden units:  $\Delta w_{ij} = \alpha \delta_j x_i$

**Step 6:** Update the weights:  $w_{ik}(\text{new}) = w_{ik}(\text{old}) + \Delta w_{ik}$

**Step 7:** Test for stopping (maximum cycles, small changes, etc)

**Note:** There are a number of options in the design of a backprop system;

- Initial weights – best to set the initial weights (and all other free parameters) to random numbers inside a small range of values (say  $-0.5$  to  $0.5$ )
- Number of cycles – tend to be quite large for backprop systems
- Number of neurons in the hidden layer – as few as possible

# Back Propagation Algorithm

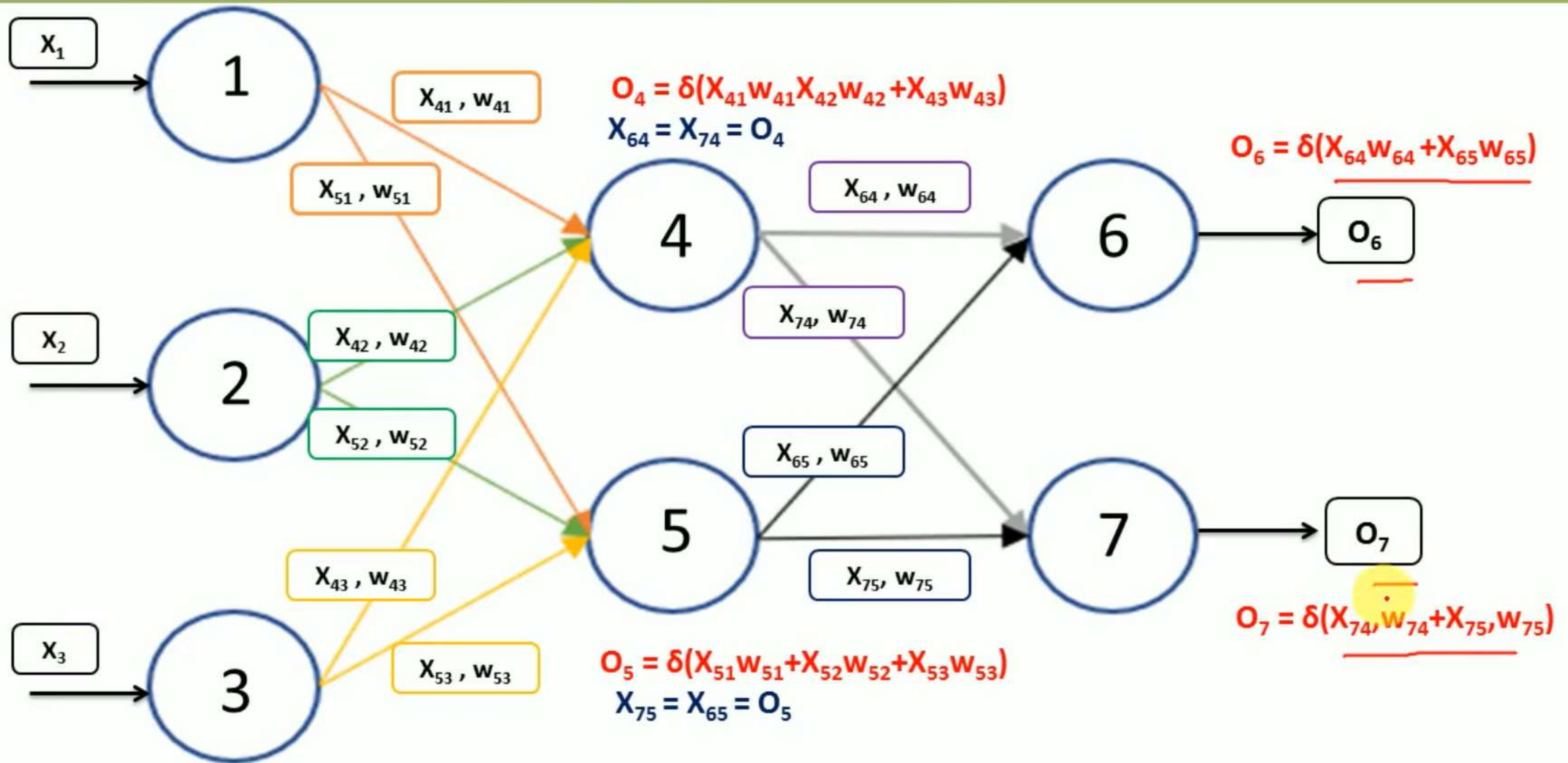
**BACKPROPAGATION** (training\_example,  $\eta$ ,  $n_{in}$ ,  $n_{out}$ ,  $n_{hidden}$  )

- Each training example is a pair of the form  $(x, t)$ , where  $(x)$  is the vector of network input values, and  $(t)$  is the vector of target network output values.
- $\eta$  is the learning rate (e.g., 0.05).
- $n_i$ , is the number of network inputs,
- $n_{hidden}$  the number of units in the hidden layer, and
- $n_{out}$  the number of output units.
- The input from unit  $i$  into unit  $j$  is denoted  $x_{ji}$ , and the weight from unit  $i$  to unit  $j$  is denoted

$w_{ji}$



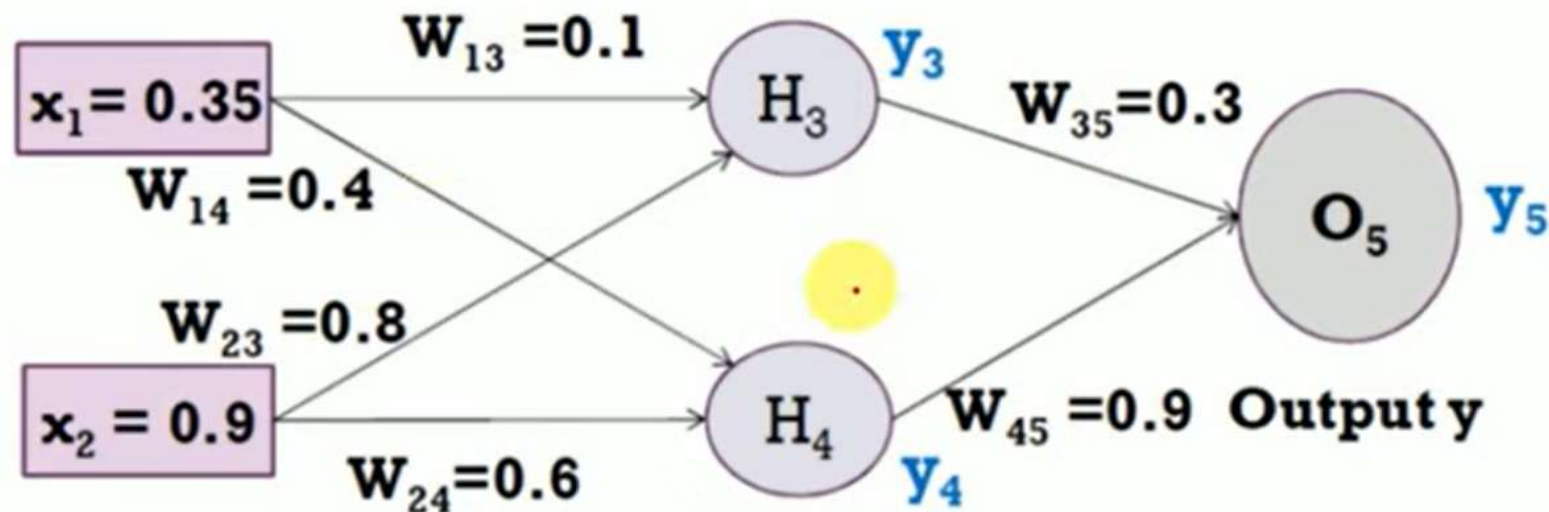
# Back Propagation Algorithm





# Back Propagation Solved Example - 1

- Assume that the neurons have a sigmoid activation function, perform a forward pass and a backward pass on the network. Assume that the actual output of  $y$  is 0.5 and learning rate is 1. Perform another forward pass.



- Forward Pass: Compute output for  $y_3$ ,  $y_4$  and  $y_5$ .

$$a_j = \sum_i (w_{i,j} * x_i) \quad y_j = F(a_j) = \frac{1}{1 + e^{-a_j}}$$

$$\begin{aligned} a_1 &= (w_{13} * x_1) + (w_{23} * x_2) \\ &= (0.1 * 0.35) + (0.8 * 0.9) = 0.755 \\ y_3 &= f(a_1) = 1 / (1 + e^{-0.755}) = 0.68 \end{aligned}$$

$$\begin{aligned} a_2 &= (w_{14} * x_1) + (w_{24} * x_2) \\ &= (0.4 * 0.35) + (0.6 * 0.9) = 0.68 \\ y_4 &= f(a_2) = 1 / (1 + e^{-0.68}) = 0.6637 \end{aligned}$$

$$\begin{aligned} a_3 &= (w_{35} * y_3) + (w_{45} * y_4) \\ &= (0.3 * 0.68) + (0.9 * 0.6637) = 0.801 \\ y_5 &= f(a_3) = 1 / (1 + e^{-0.801}) = \mathbf{0.69 \text{ (Network Output)}} \end{aligned}$$

$$\text{Error} = y_{\text{target}} - y_5 = -0.19$$

- Each weight changed by:

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

$$\delta_j = o_j(1 - o_j)(t_j - o_j) \quad \text{if } j \text{ is an output unit}$$

$$\delta_j = o_j(1 - o_j) \sum_k \delta_k w_{kj} \quad \text{if } j \text{ is a hidden unit}$$

- where  $\eta$  is a constant called the learning rate
- $t_j$  is the correct teacher output for unit  $j$
- $\delta_j$  is the error measure for unit  $j$

Backward Pass: Compute  $\delta_3$ ,  $\delta_4$  and  $\delta_5$ .

For output unit:

$$\begin{aligned} \delta_5 &= y(1-y)(y_{\text{target}} - y) \\ &= 0.69 * (1 - 0.69) * (0.5 - 0.69) = -0.0406 \end{aligned}$$

For hidden unit:

$$\begin{aligned} \delta_3 &= y_3(1-y_3) w_{35} * \delta_5 \\ &= 0.68 * (1 - 0.68) * (0.3 * -0.0406) = -0.00265 \end{aligned}$$

$$\begin{aligned} \delta_4 &= y_4(1-y_4) w_{45} * \delta_5 \\ &= 0.6637 * (1 - 0.6637) * (0.9 * -0.0406) = -0.0082 \end{aligned}$$



## Compute new weights

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

$$\Delta w_{45} = \eta \delta_5 y_4 = 1 * -0.0406 * 0.6637 = -0.0269$$

$$w_{45}(\text{new}) = \Delta w_{45} + w_{45}(\text{old}) = -0.0269 + (0.9) = 0.8731$$

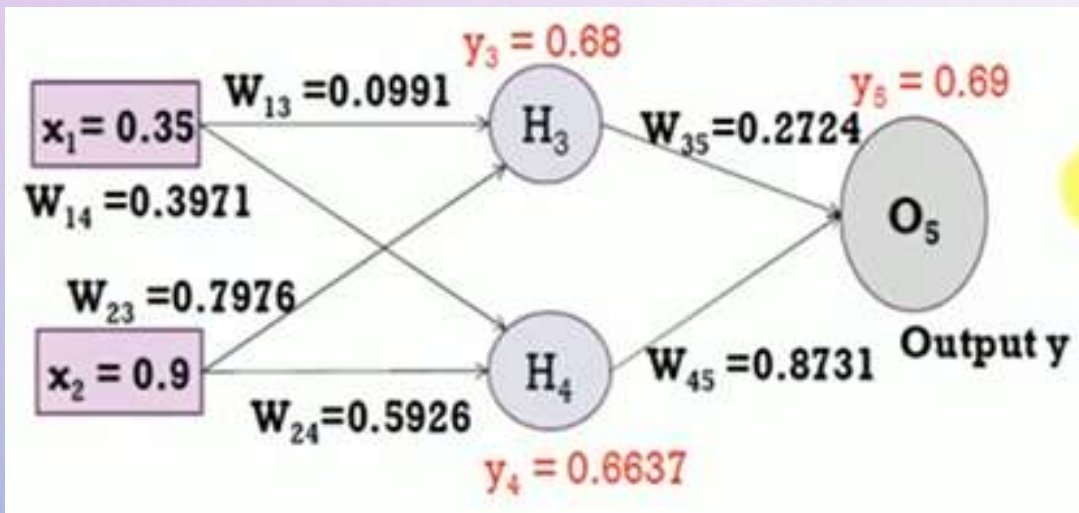
$$\Delta w_{14} = \eta \delta_4 x_1 = 1 * -0.0082 * 0.35 = -0.00287$$

$$w_{14}(\text{new}) = \Delta w_{14} + w_{14}(\text{old}) = -0.00287 + 0.4 = 0.3971$$



- Similarly, update all other weights

<b>i</b>	<b>j</b>	<b><math>w_{ij}</math></b>	<b><math>\delta_i</math></b>	<b><math>x_i</math></b>	<b><math>\eta</math></b>	<b>Updated <math>w_{ij}</math></b>
1	3	0.1	-0.00265	0.35	1	0.0991
2	3	0.8	-0.00265	0.9	1	0.7976
1	4	0.4	-0.0082	0.35	1	0.3971
2	4	0.6	-0.0082	0.9	1	0.5926
3	5	0.3	-0.0406	0.68	1	0.2724
4	5	0.9	-0.0406	0.6637	1	0.8731



$$\text{Error} = y_{\text{target}} - y_5 = -0.182$$

- Forward Pass: Compute output for  $y_3$ ,  $y_4$  and  $y_5$ .

$$a_j = \sum_i (w_{ij} * x_i) \quad y_j = F(a_j) = \frac{1}{1 + e^{-a_j}}$$

$$a_1 = (w_{13} * x_1) + (w_{23} * x_2) \\ = (0.0991 * 0.35) + (0.7976 * 0.9) = 0.7525$$

$$y_3 = f(a_1) = 1 / (1 + e^{-0.7525}) = \underline{0.6797}$$

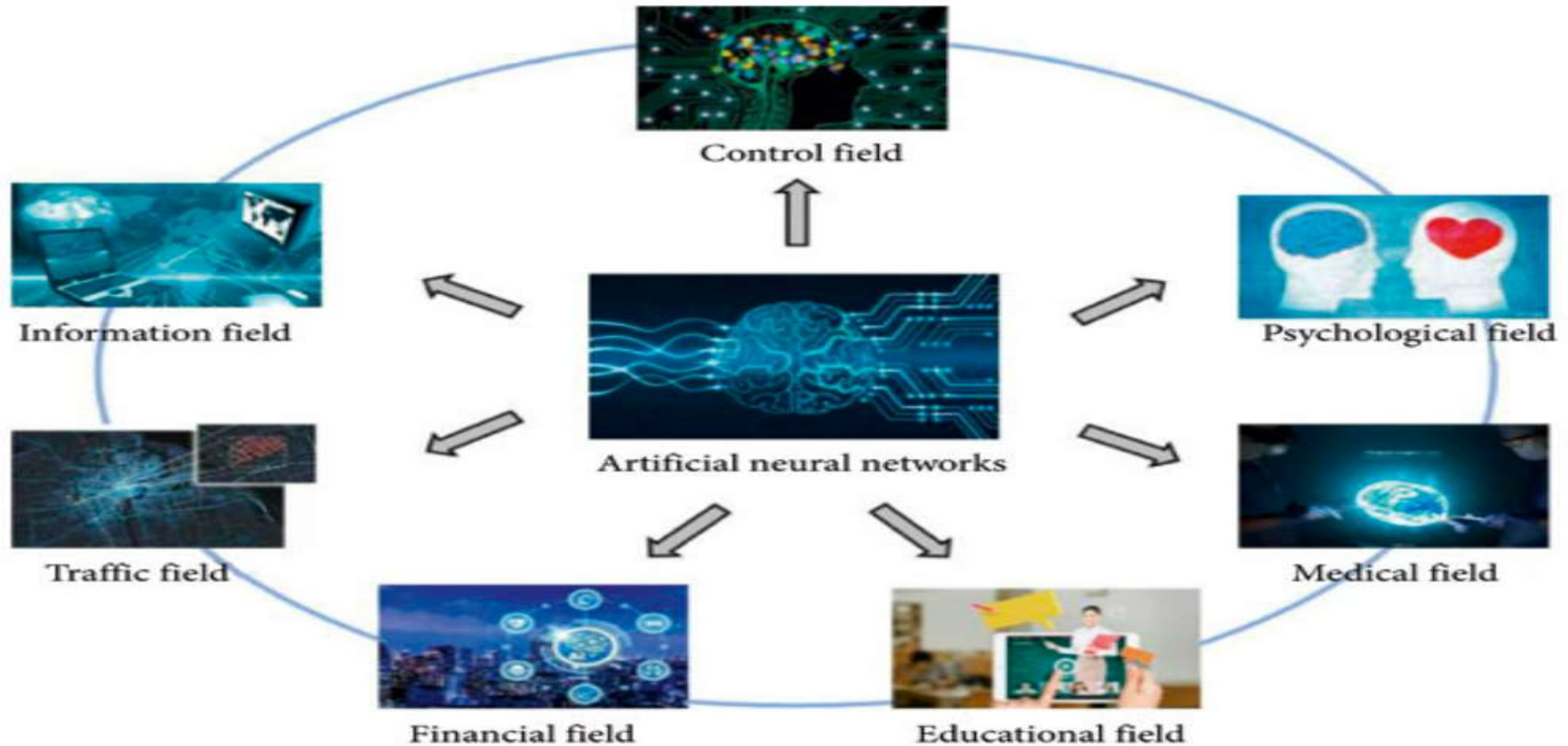
$$a_2 = (w_{14} * x_1) + (w_{24} * x_2) \\ = (0.3971 * 0.35) + (0.5926 * 0.9) = 0.6723$$

$$y_4 = f(a_2) = 1 / (1 + e^{-0.6723}) = \underline{0.6620}$$

$$a_3 = (w_{35} * y_3) + (w_{45} * y_4) \\ = (0.2724 * 0.6797) + (0.8731 * 0.6620) = 0.7631$$

$$y_5 = f(a_3) = 1 / (1 + e^{-0.7631}) = \underline{0.6820} \text{ (Network Output)}$$

# APPLICATION OF ARTIFICIAL NEURAL NETWORKS





# APPLICATION OF ARTIFICIAL NEURAL NETWORKS

