

# Artificial Neural Network

Lecture Module 22



# Neural Networks

---

- Artificial neural network (ANN) is a machine learning approach that models human brain and consists of a number of artificial neurons.
- Neuron in ANNs tend to have fewer connections than biological neurons.
- Each neuron in ANN receives a number of inputs.
- An activation function is applied to these inputs which results in activation level of neuron (output value of the neuron).
- Knowledge about the learning task is given in the form of examples called training examples.



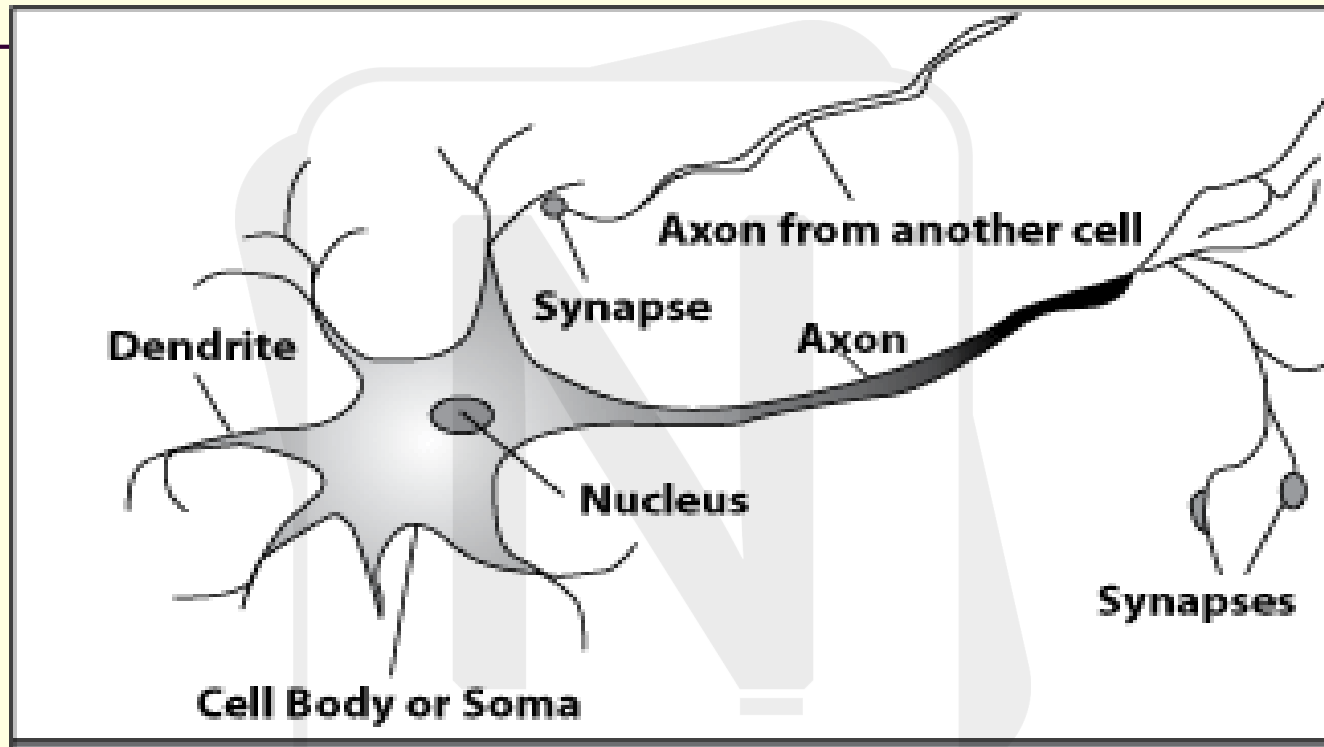
# How do our brains work?

- The Brain is a massively parallel information processing system.
- Our brains are a huge network of processing elements. A typical brain contains a network of 10 billion neurons.



# How do our brains work?

- A processing element

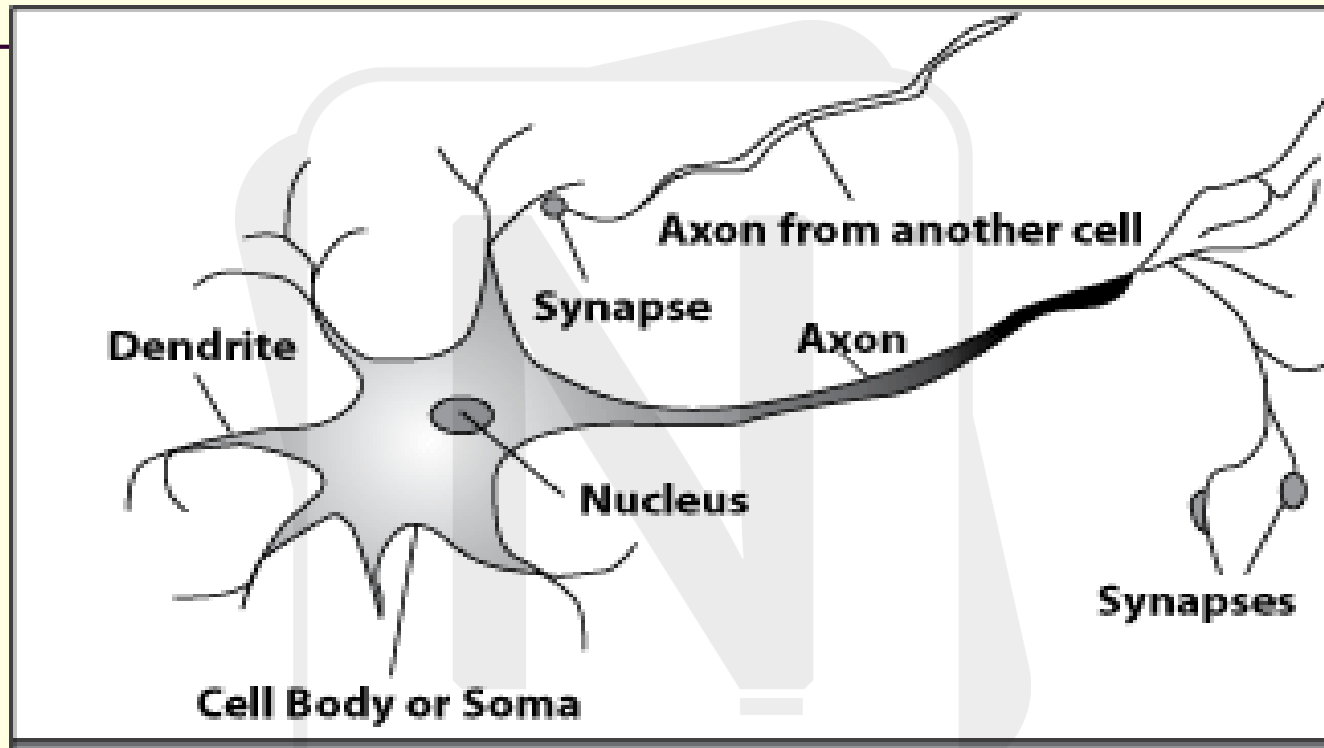


Dendrites: Input  
Cell body: Processor  
Synaptic: Link  
Axon: Output



# How do our brains work?

- A processing element

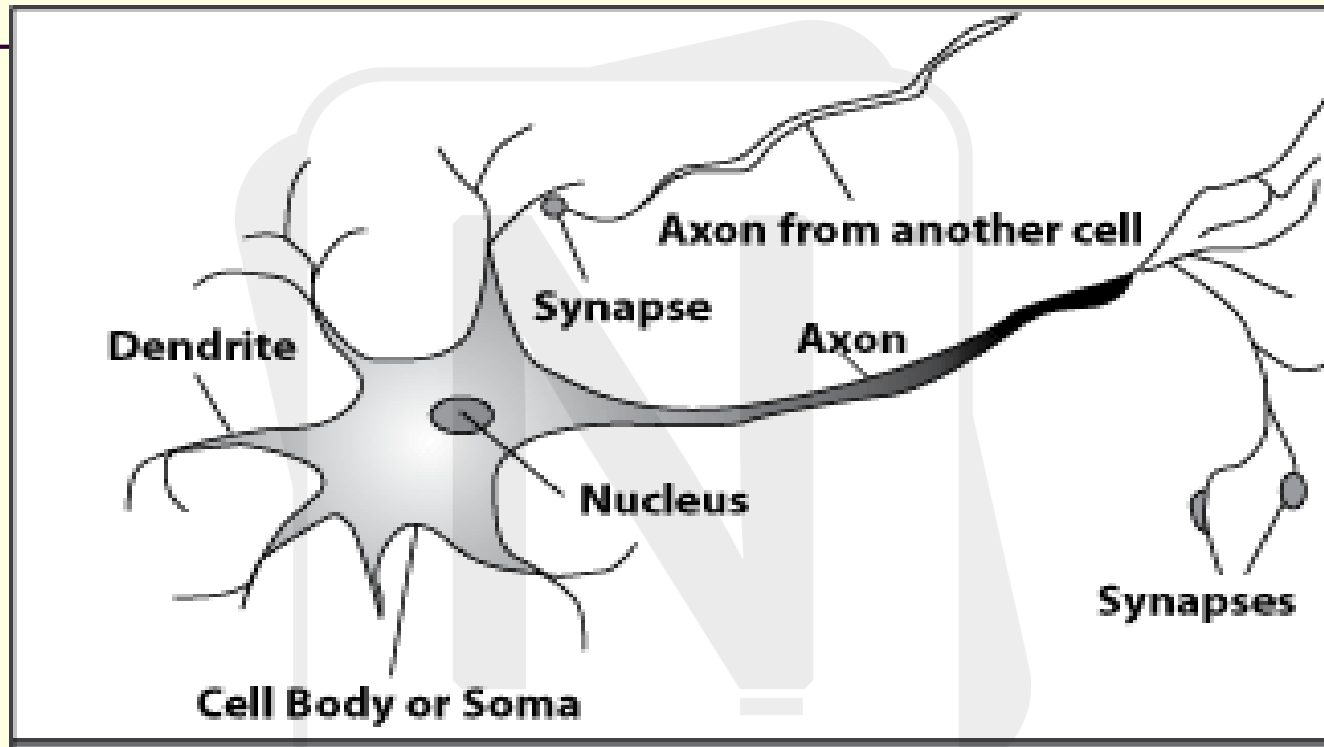


A neuron is connected to other neurons through about *10,000 synapses*



# How do our brains work?

- A processing element

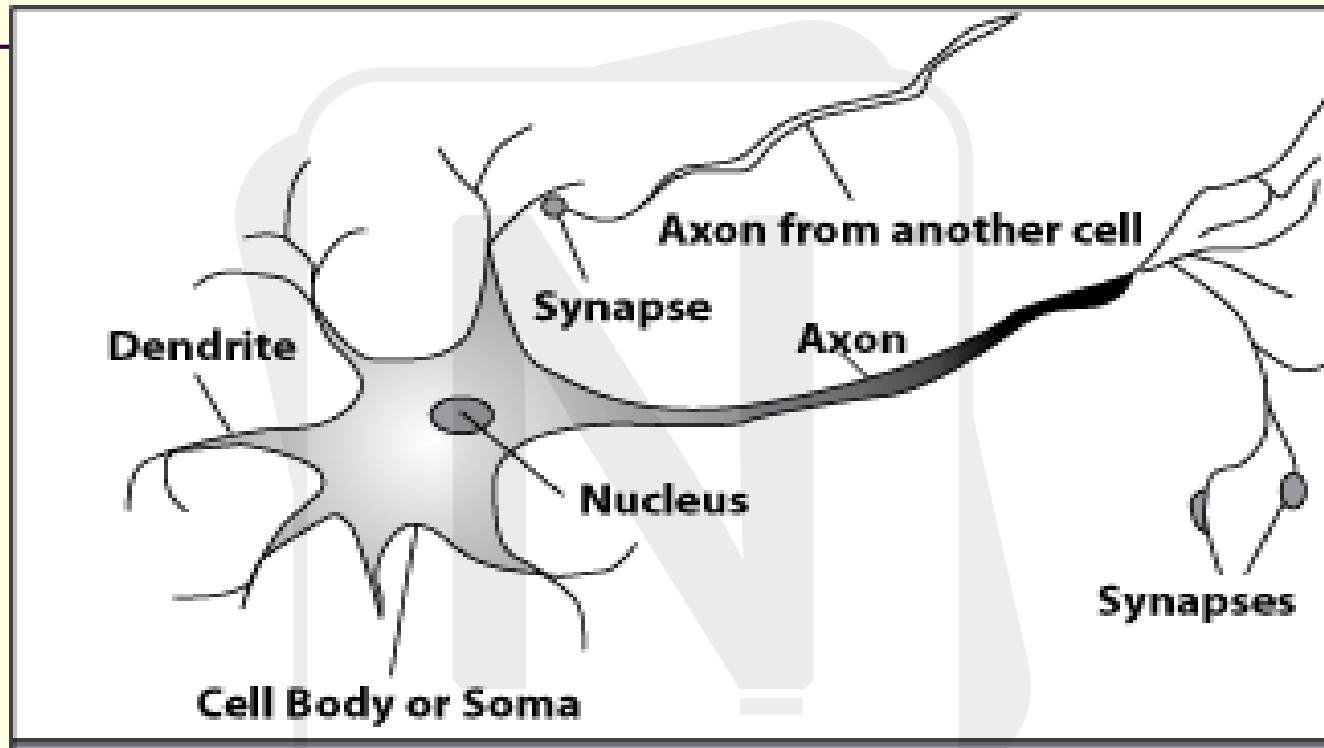


A neuron receives input from other neurons. Inputs are combined.



# How do our brains work?

- A processing element

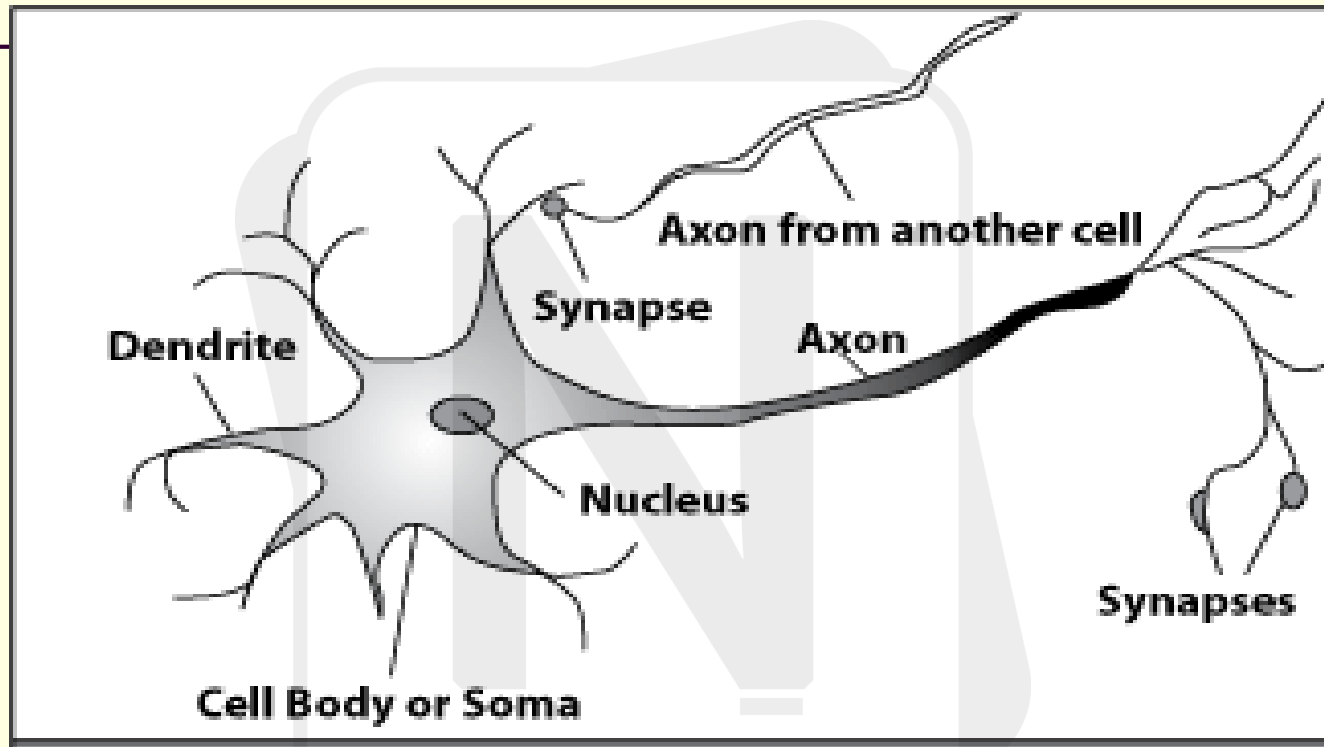


Once input exceeds a critical level, the neuron discharges a spike - an electrical pulse that travels from the body, down the axon, to the next neuron(s)



# How do our brains work?

- A processing element



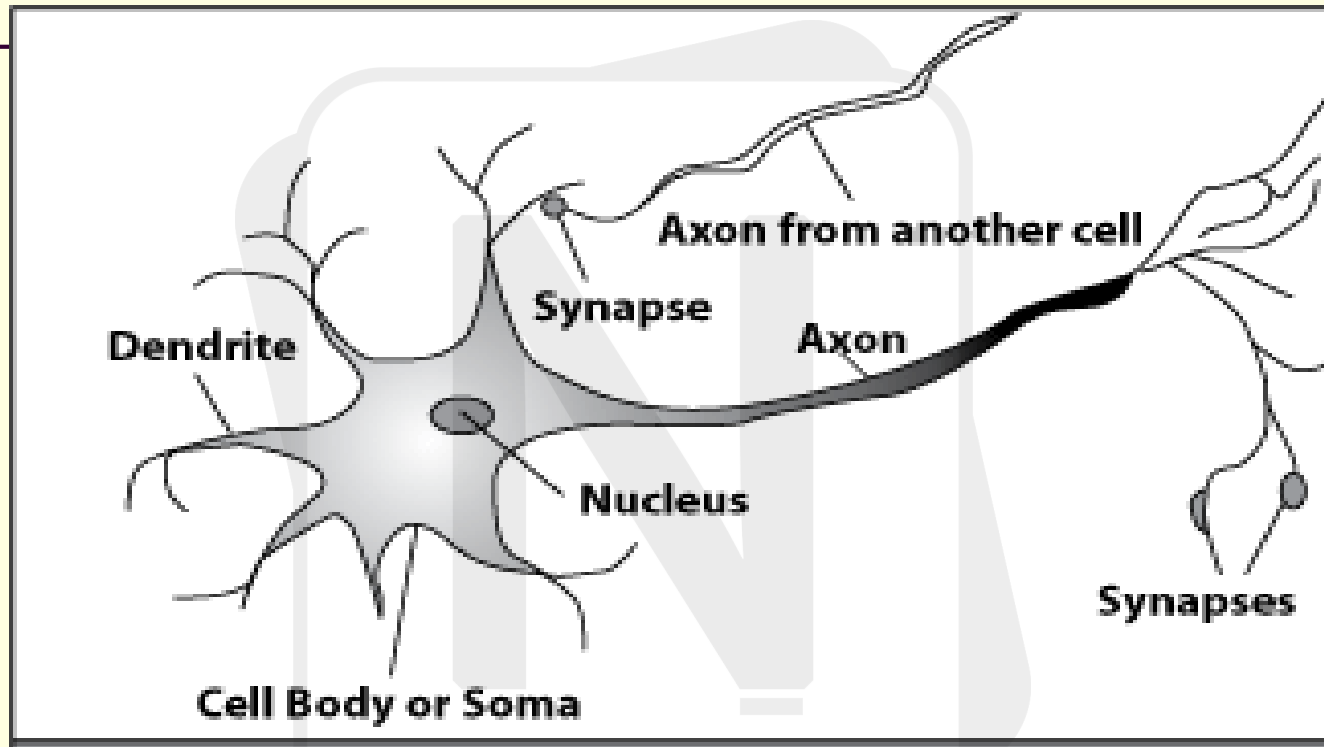
The axon endings almost touch the dendrites or cell body of the next neuron.





# How do our brains work?

- A processing element

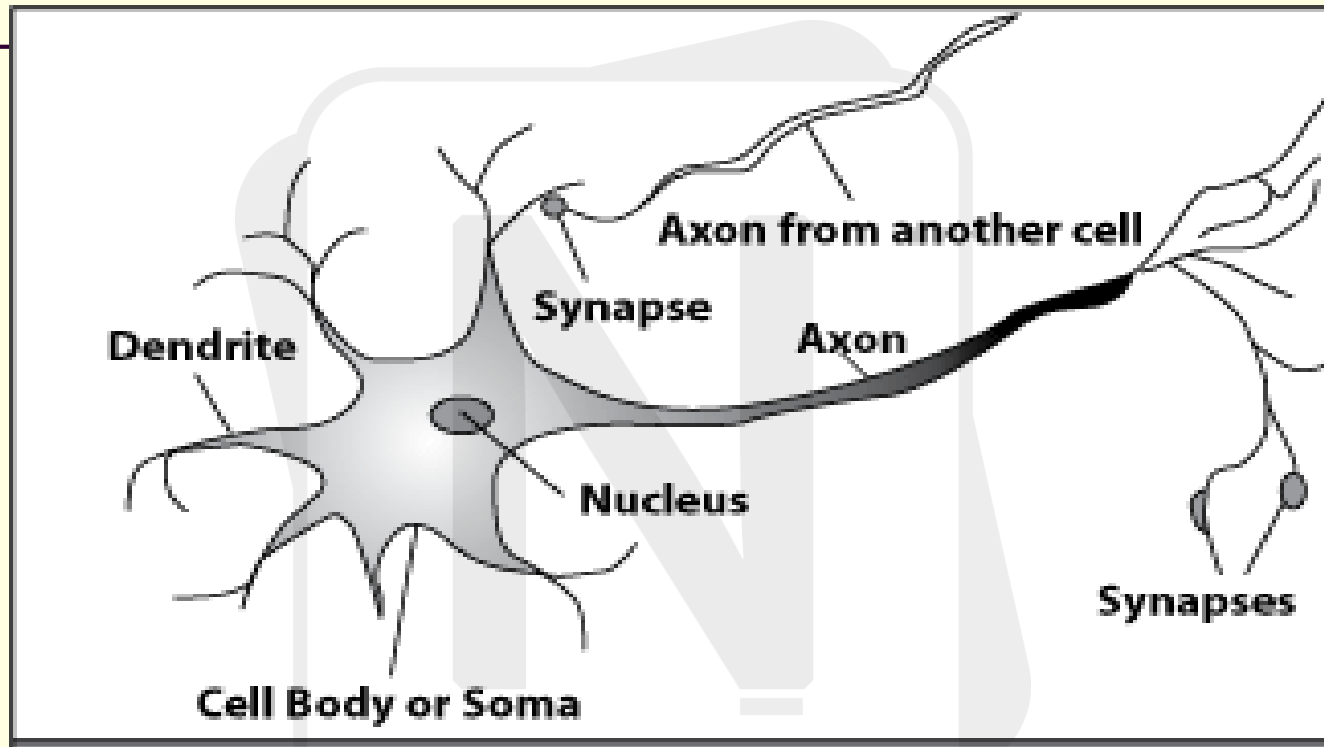


Transmission of an electrical signal from one neuron to the next is effected by neurotransmitters.



# How do our brains work?

- A processing element

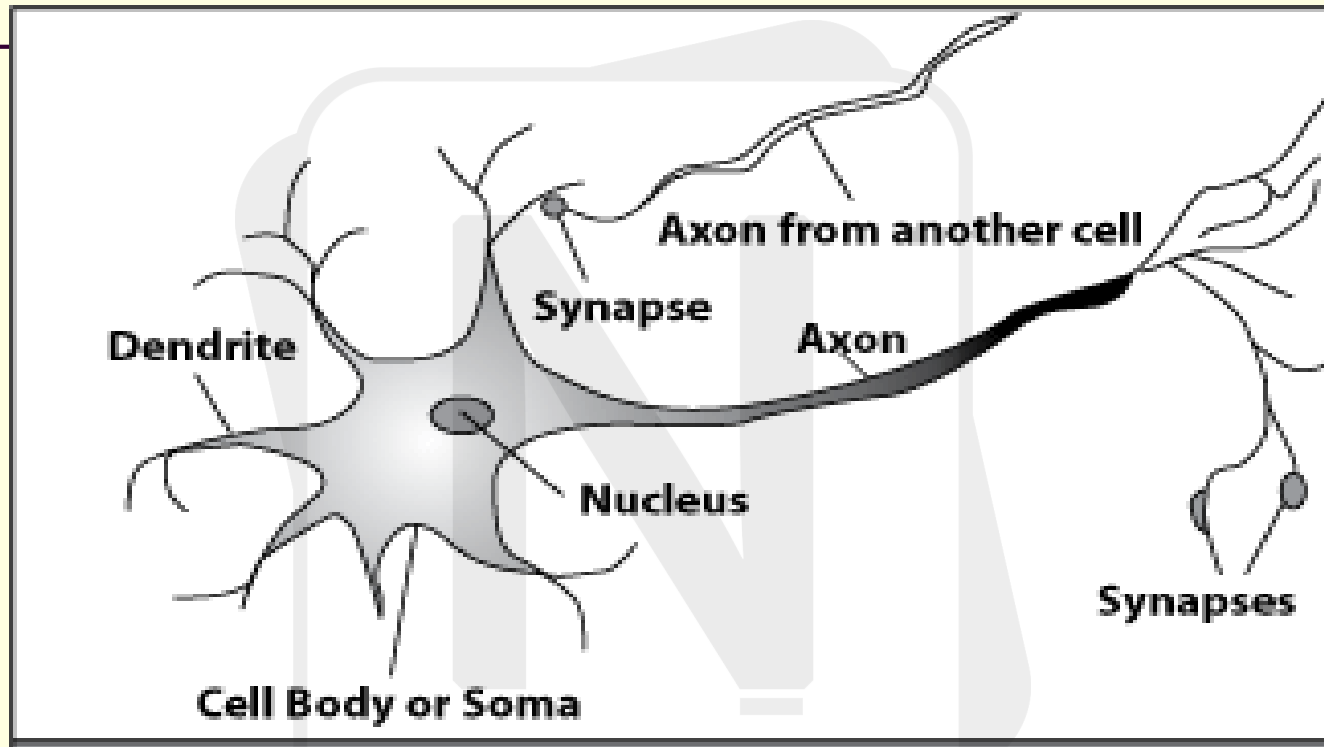


Neurotransmitters are chemicals which are released from the first neuron and which bind to the Second.



# How do our brains work?

- A processing element

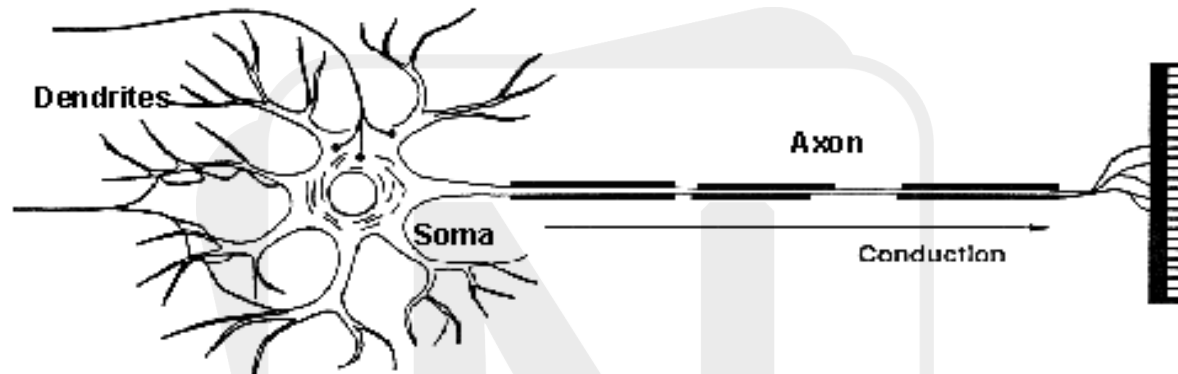


This link is called a synapse. The strength of the signal that reaches the next neuron depends on factors such as the amount of neurotransmitter available.

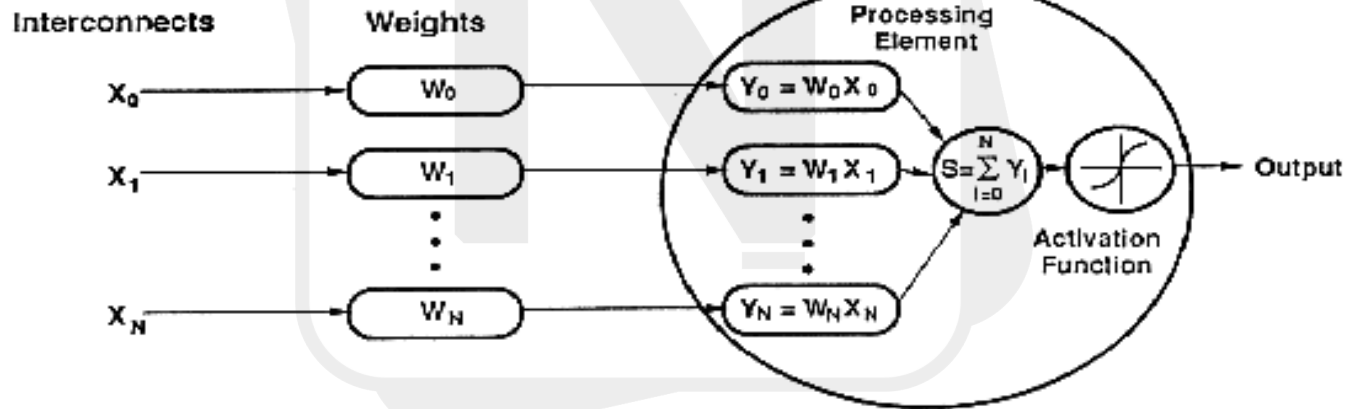


# How do ANNs work?

## Biological Neuron



## Artificial Neuron

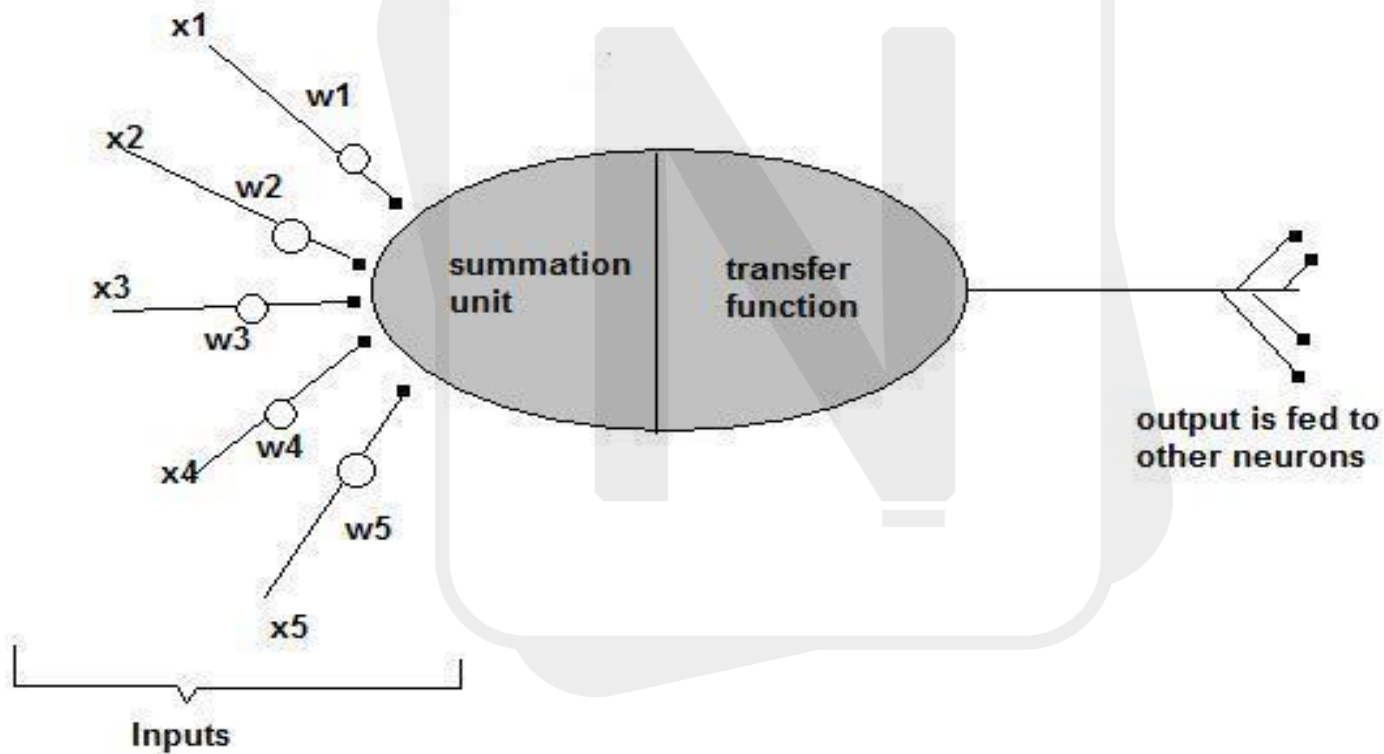


An artificial neuron is an imitation of a human neuron

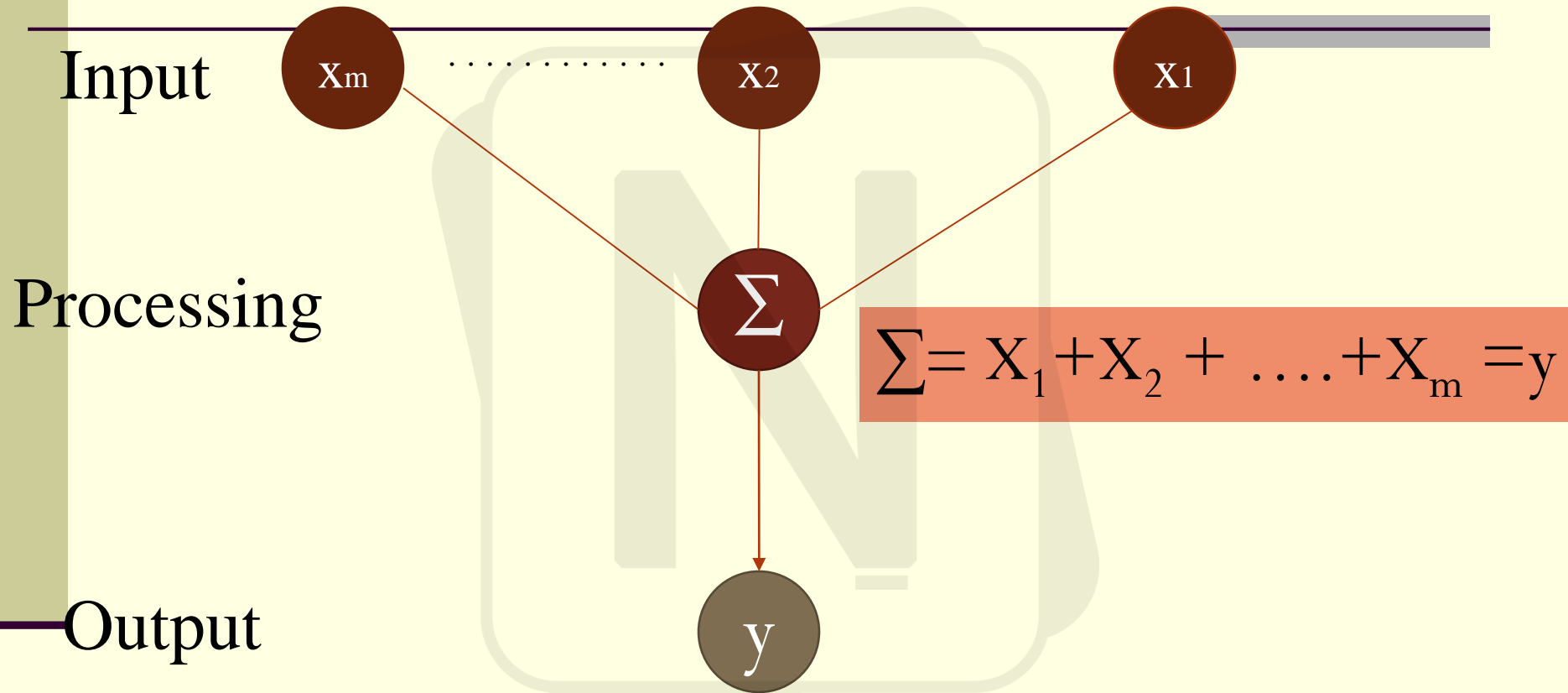
# How do ANNs work?

- Now, let us have a look at the model of an artificial neuron.

## A Single Neuron

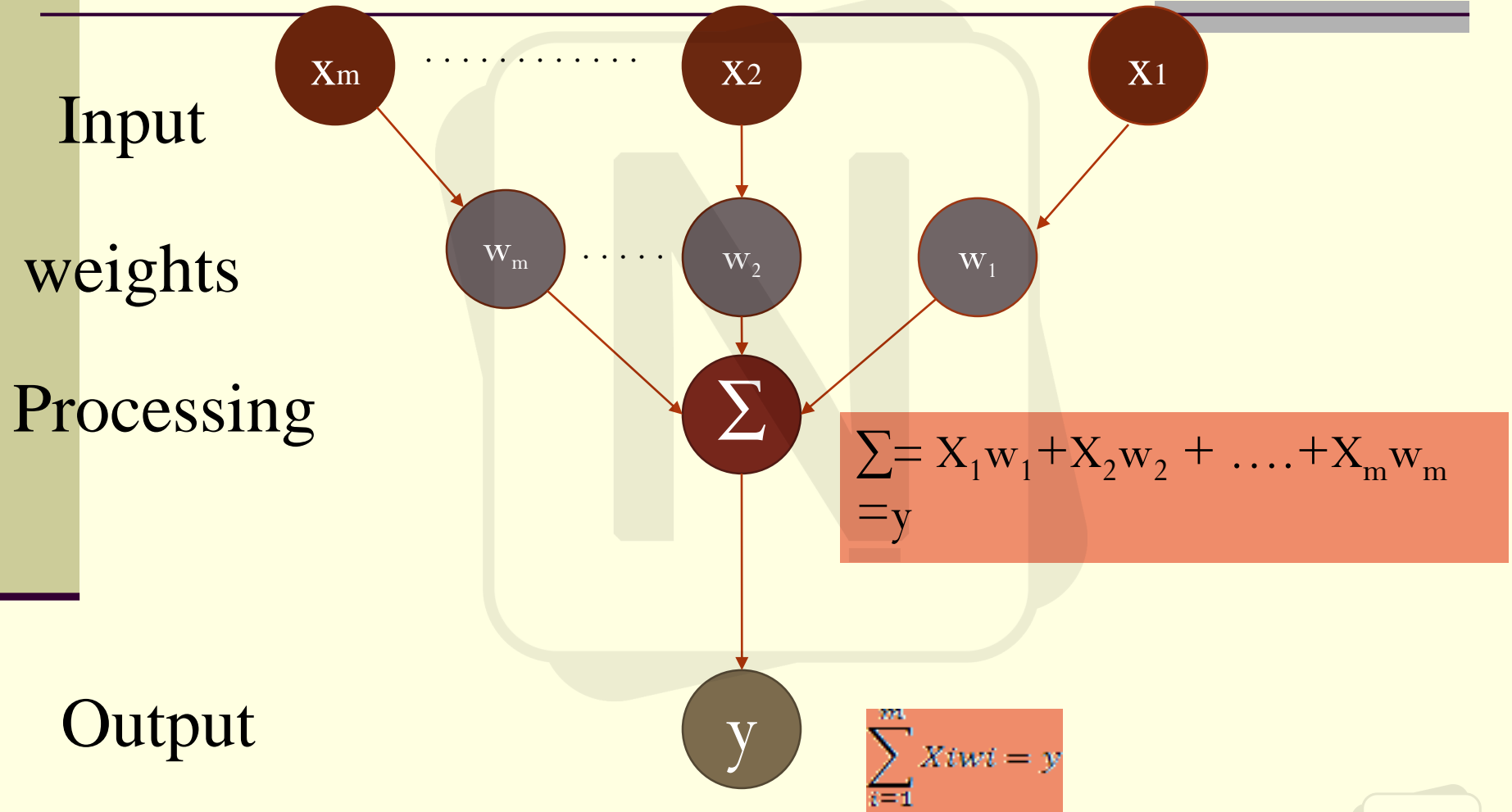


# How do ANNs work?



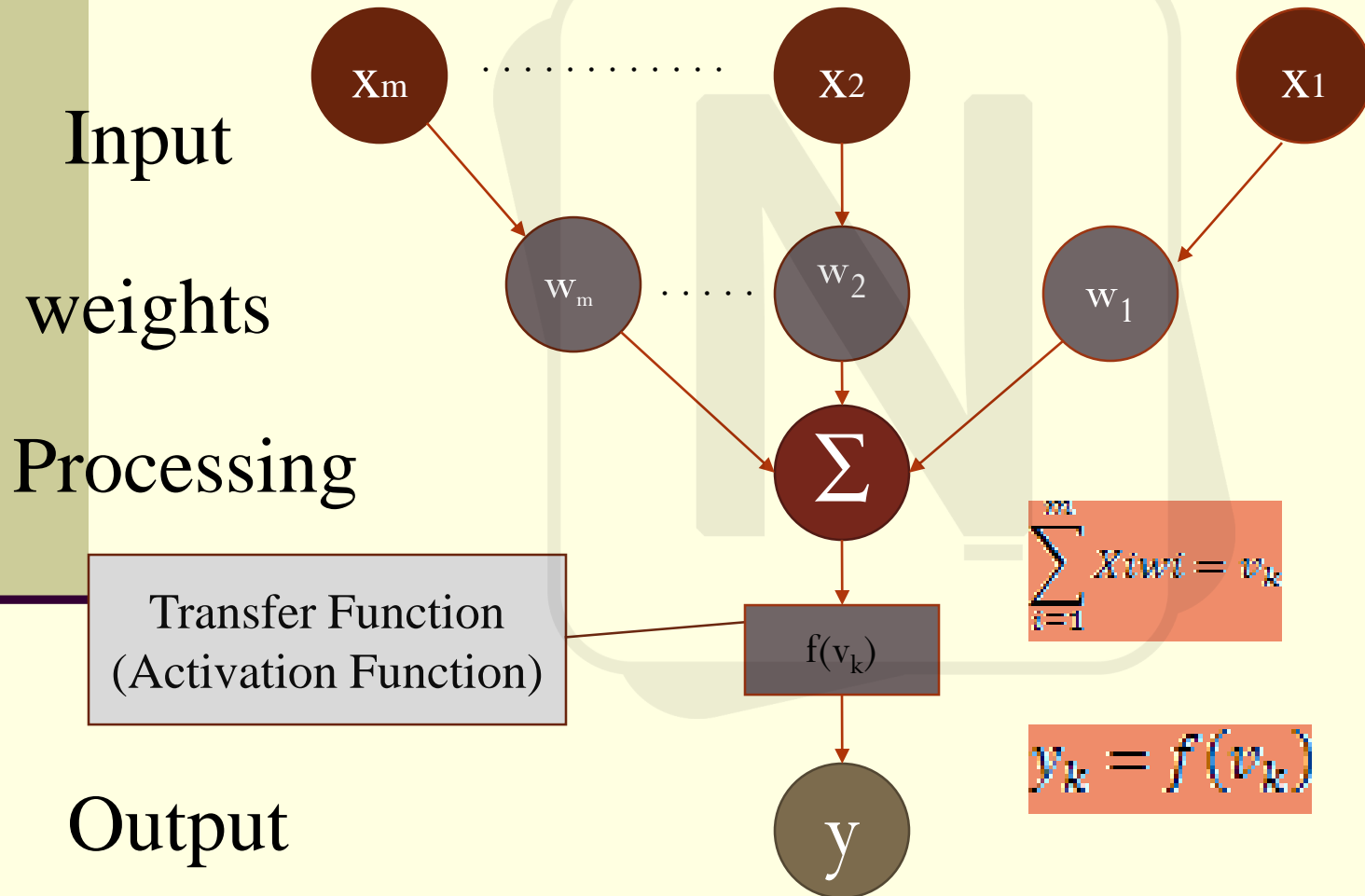
# How do ANNs work?

Not all inputs are equal



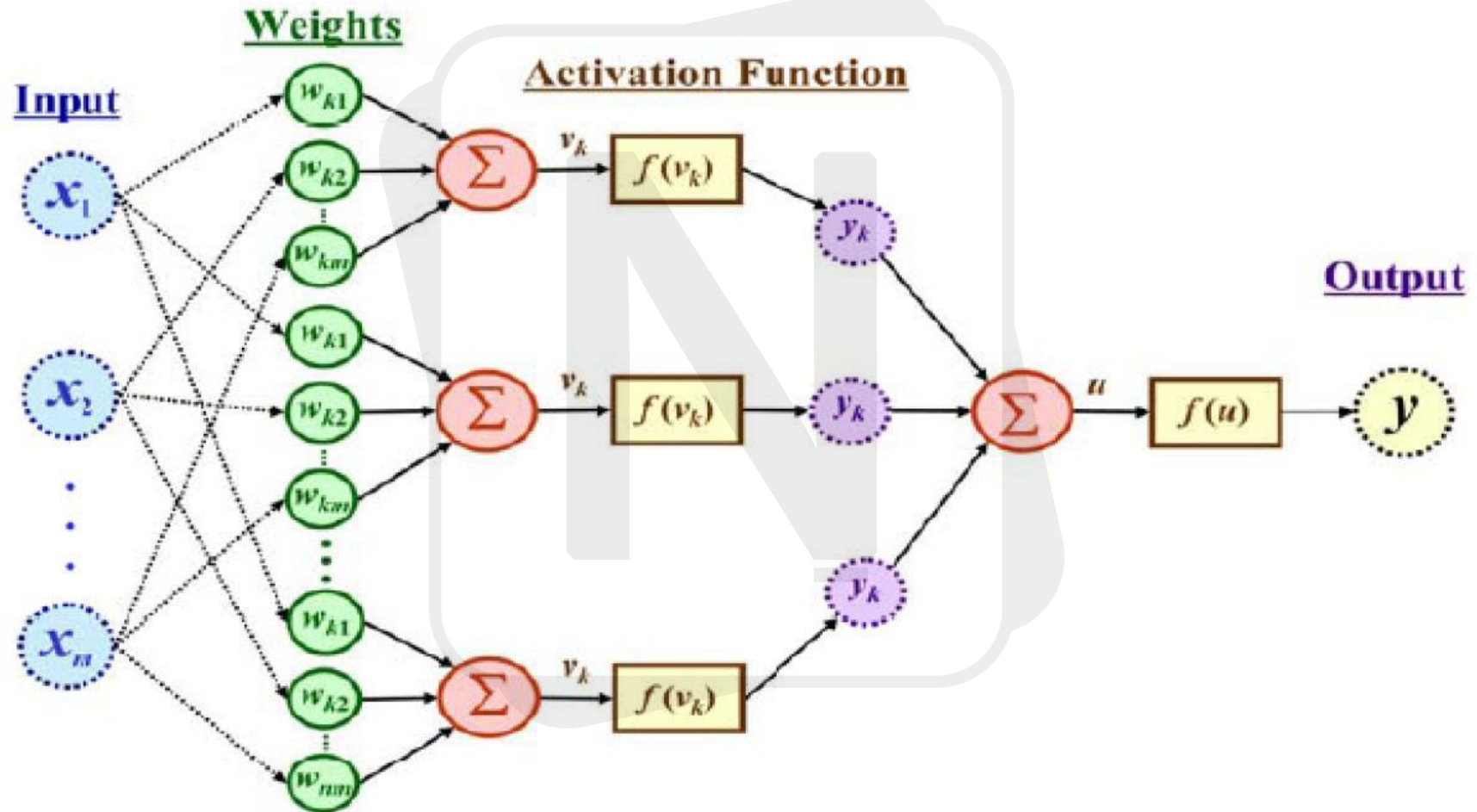
# How do ANNs work?

The signal is not passed down to the  
next neuron verbatim





The output is a function of the input, that is affected by the weights, and the transfer functions



# Artificial Neural Networks

---

- An ANN can:
  1. compute *any computable* function, by the appropriate selection of the network topology and weights values.
  2. learn from experience!
    - Specifically, by trial-and-error



# Contd..

---

- An Artificial Neural Network is specified by:
  - **neuron model**: the information processing unit of the NN,
  - **an architecture**: a set of neurons and links connecting neurons. Each link has a weight,
  - **a learning algorithm**: used for training the NN by modifying the weights in order to model a particular learning task correctly on the training examples.
- The aim is to obtain a NN that is trained and generalizes well.
- It should behaves correctly on new instances of the learning task.



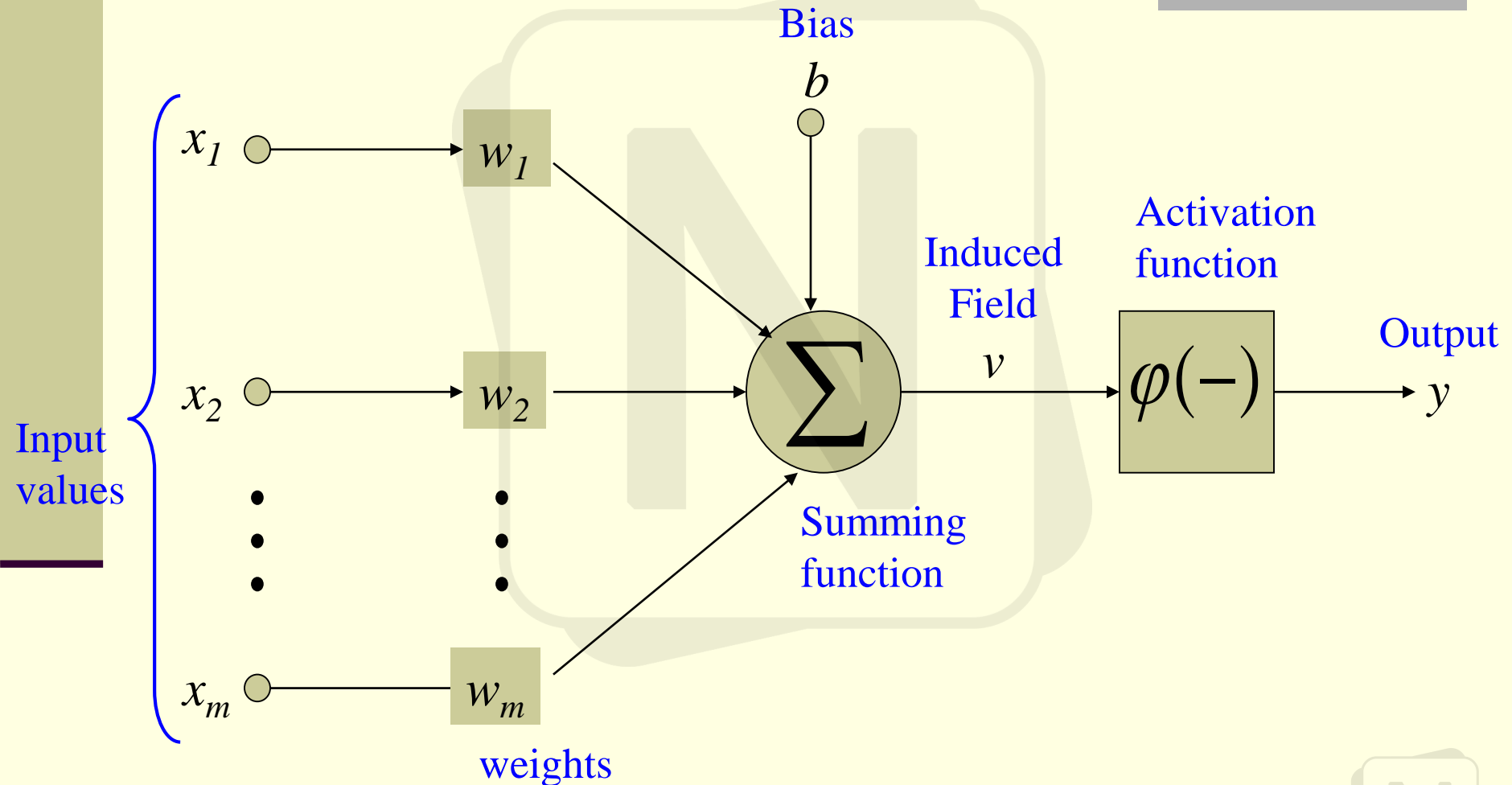
# Neuron

- The neuron is the basic information processing unit of a NN. It consists of:
  - 1 A set of **links**, describing the neuron inputs, with **weights**  $W_1, W_2, \dots, W_m$
  - 2 An **adder** function (linear combiner) for computing the weighted sum of the inputs:  
(real numbers)
  - 3 **Activation function**  $\phi$  for limiting the amplitude of the neuron output. Here 'b' denotes bias.

$$u = \sum_{j=1}^m w_j x_j$$

$$y = \phi(u + b)$$

# The Neuron Diagram



# Bias of a Neuron

- The bias  **$b$**  has the effect of applying a **transformation** to the weighted sum  **$u$**

$$v = u + b$$

- The bias is an external parameter of the neuron. It can be modeled by adding an extra input.
- **$v$**  is called **induced field** of the neuron

$$v = \sum_{j=0}^m w_j x_j$$

$$w_0 = b$$



# Neuron Models

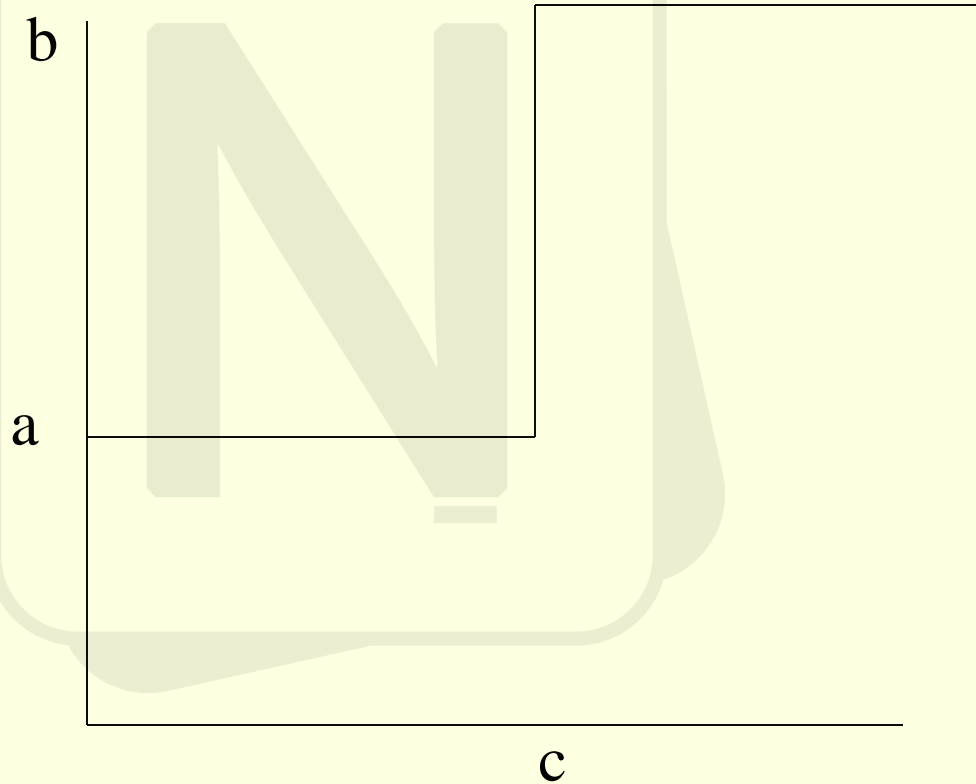
- The choice of activation function  $\varphi$  determines the neuron model.

## Examples:

- step function: 
$$\varphi(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > c \end{cases}$$
- ramp function: 
$$\varphi(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > d \\ a + ((v - c)(b - a) / (d - c)) & \text{otherwise} \end{cases}$$
- sigmoid function with z,x,y parameters 
$$\varphi(v) = z + \frac{1}{1 + \exp(-xv + y)}$$
- Gaussian function: 
$$\varphi(v) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{v - \mu}{\sigma}\right)^2\right)$$

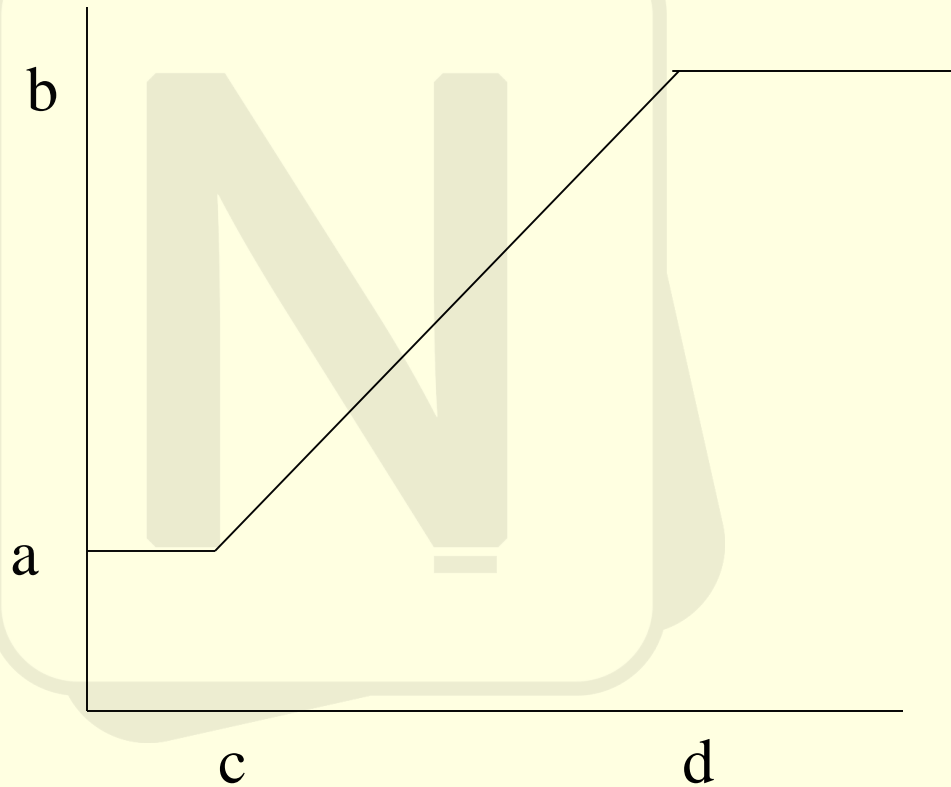


# Step Function

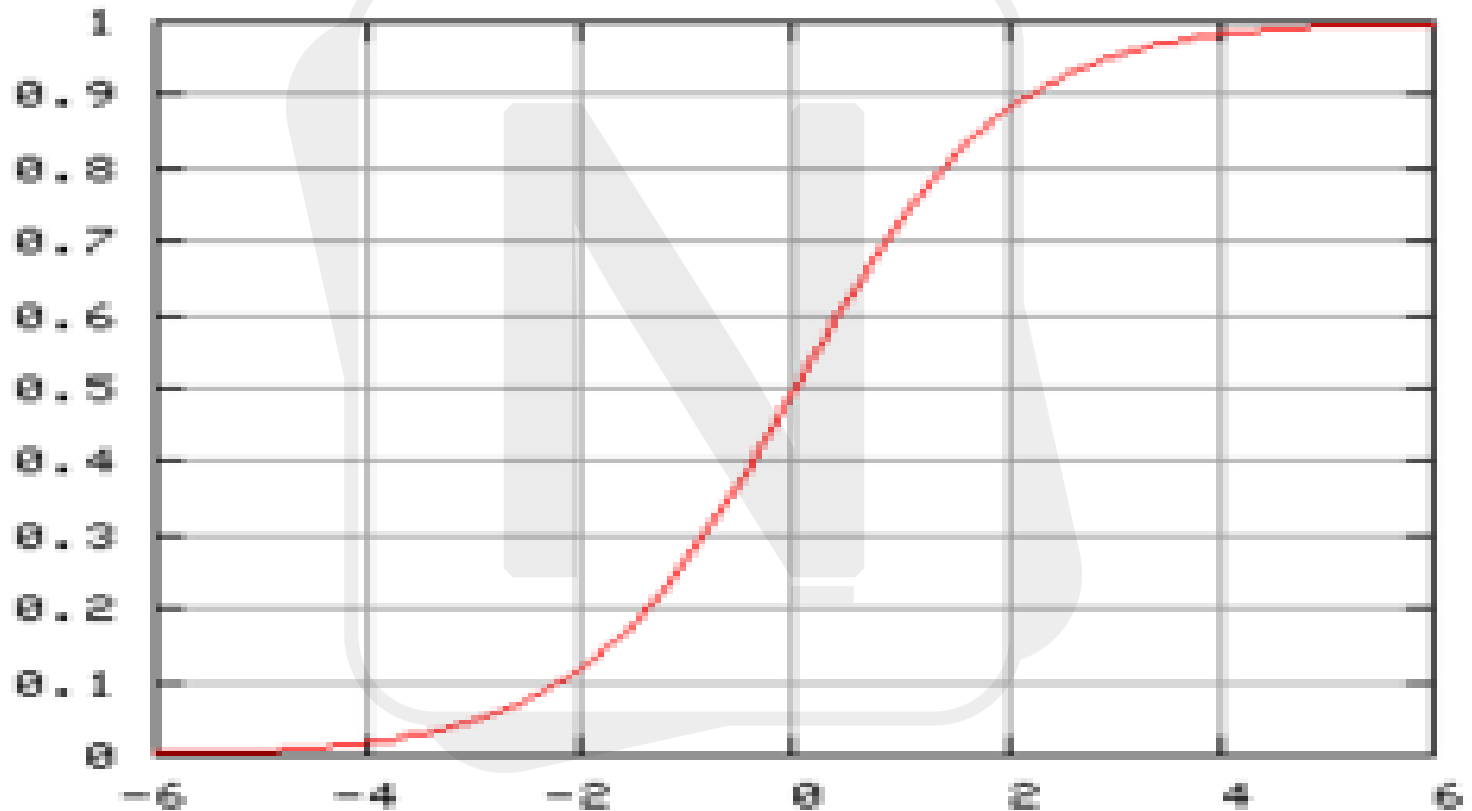




# Ramp Function

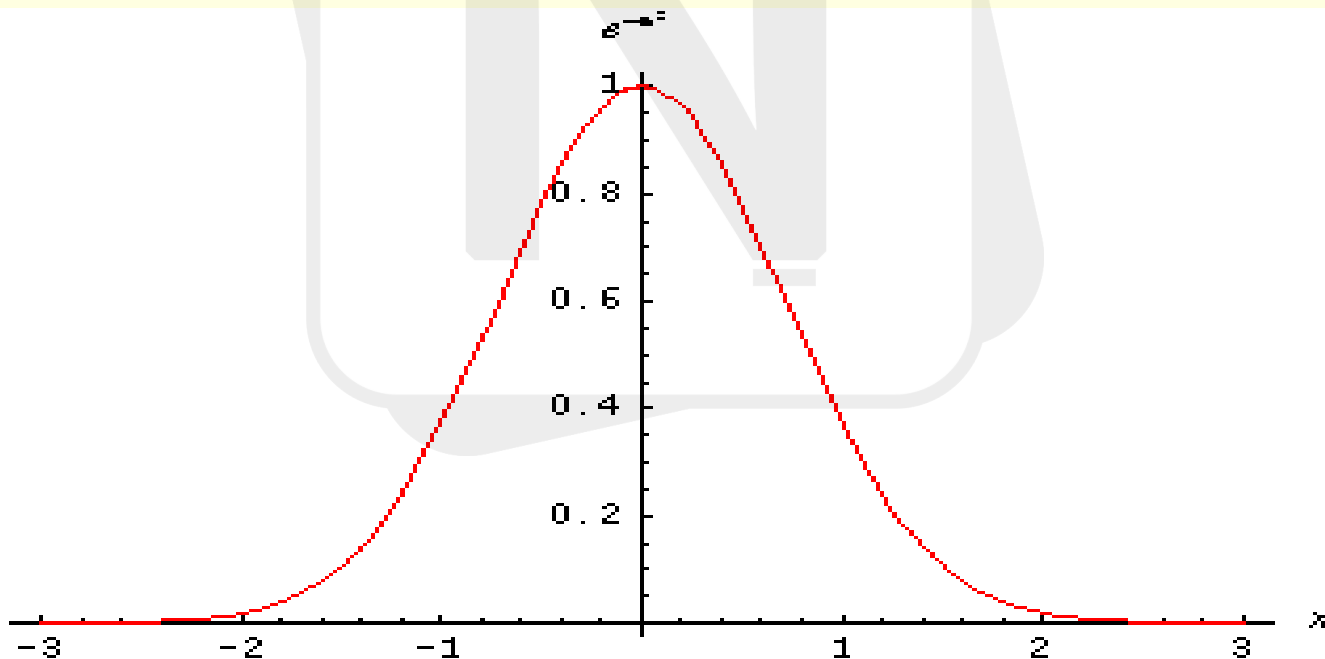


# Sigmoid function



- The **Gaussian function** is the probability function of the normal distribution. Sometimes also called the frequency curve.

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-(x-\mu)^2 / 2\sigma^2},$$



# Network Architectures

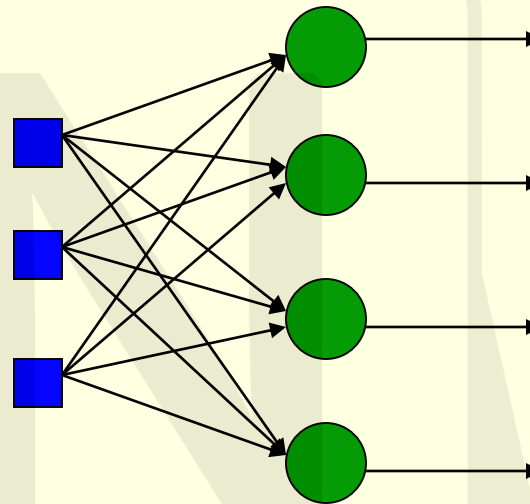
---

- Three different classes of network architectures
  - single-layer feed-forward
  - multi-layer feed-forward
  - recurrent
- The **architecture** of a neural network is linked with the learning algorithm used to train



# Single Layer Feed-forward

*Input layer  
of  
source nodes*



*Output layer  
of  
neurons*

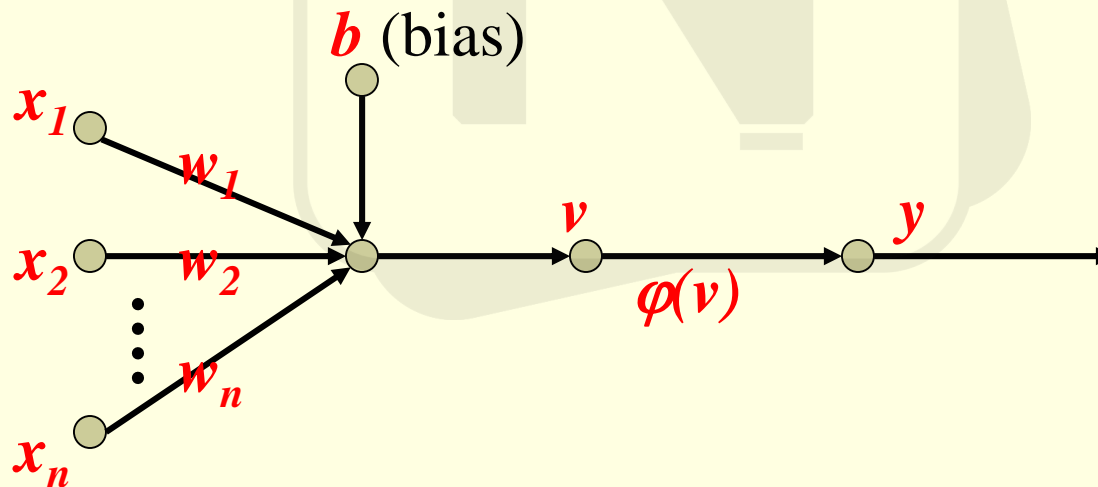


# Perceptron: Neuron Model

(Special form of single layer feed forward)

- The perceptron was first proposed by Rosenblatt (1958) is a simple neuron that is used to classify its input into one of two categories.
- A perceptron uses a **step function** that returns +1 if weighted sum of its input  $\geq 0$  and -1 otherwise

$$\varphi(v) = \begin{cases} +1 & \text{if } v \geq 0 \\ -1 & \text{if } v < 0 \end{cases}$$

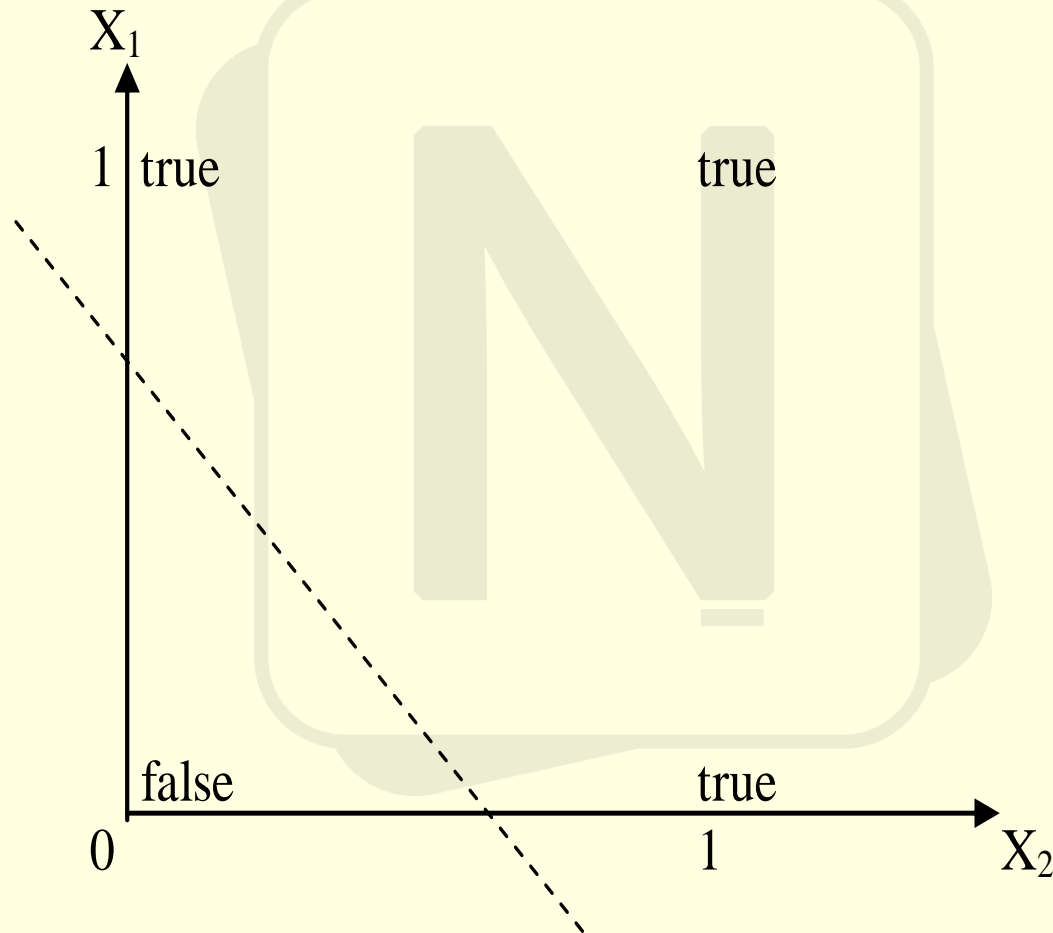


# Perceptron for Classification

- The perceptron is used for binary classification.
- First train a perceptron for a classification task.
  - Find suitable weights in such a way that the training examples are correctly classified.
  - Geometrically try to find a hyper-plane that separates the examples of the two classes.
- The perceptron can only model linearly separable classes.
- When the two classes are not linearly separable, it may be desirable to obtain a linear separator that minimizes the mean squared error.
- Given training examples of classes  $C_1$ ,  $C_2$  train the perceptron in such a way that :
  - If the output of the perceptron is +1 then the input is assigned to class  $C_1$
  - If the output is -1 then the input is assigned to  $C_2$



# Boolean function OR – Linearly separable





# Learning Process for Perceptron

- Initially assign random weights to inputs between -0.5 and +0.5
- Training data is presented to perceptron and its output is observed.
- If output is incorrect, the weights are adjusted accordingly using following formula.

$$w_i \leftarrow w_i + (a * x_i * e),$$
 where 'e' is error produced and 'a' ( $-1 < a < 1$ ) is learning rate

- 'a' is defined as 0 if output is correct, it is +ve, if output is too low and -ve, if output is too high.
- Once the modification to weights has taken place, the next piece of training data is used in the same way.
- Once all the training data have been applied, the process starts again until all the weights are correct and all errors are zero.
- Each iteration of this process is known as an epoch.



# Example: Perceptron to learn OR function

- Initially consider  $w_1 = -0.2$  and  $w_2 = 0.4$
- Training data say,  $x_1 = 0$  and  $x_2 = 0$ , output is 0.
- Compute  $y = \text{Step}(w_1 * x_1 + w_2 * x_2) = 0$ . Output is correct so weights are not changed.
- For training data  $x_1 = 0$  and  $x_2 = 1$ , output is 1
- Compute  $y = \text{Step}(w_1 * x_1 + w_2 * x_2) = 0.4 = 1$ . Output is correct so weights are not changed.
- Next training data  $x_1 = 1$  and  $x_2 = 0$  and output is 1
- Compute  $y = \text{Step}(w_1 * x_1 + w_2 * x_2) = -0.2 = 0$ . Output is incorrect, hence weights are to be changed.
- Assume  $a = 0.2$  and error  $e = 1$   
 **$w_i = w_i + (a * x_i * e)$  gives  $w_1 = 0$  and  $w_2 = 0.4$**
- With these weights, test the remaining test data.
- Repeat the process till we get stable result.



# Perceptron: Limitations

- The perceptron can only model linearly separable functions,
  - those functions which can be drawn in 2-dim graph and single straight line separates values in two part.
- Boolean functions given below are linearly separable:
  - AND
  - OR
  - COMPLEMENT
- It cannot model XOR function as it is non linearly separable.
  - When the two classes are not linearly separable, it may be desirable to obtain a linear separator that minimizes the mean squared error.



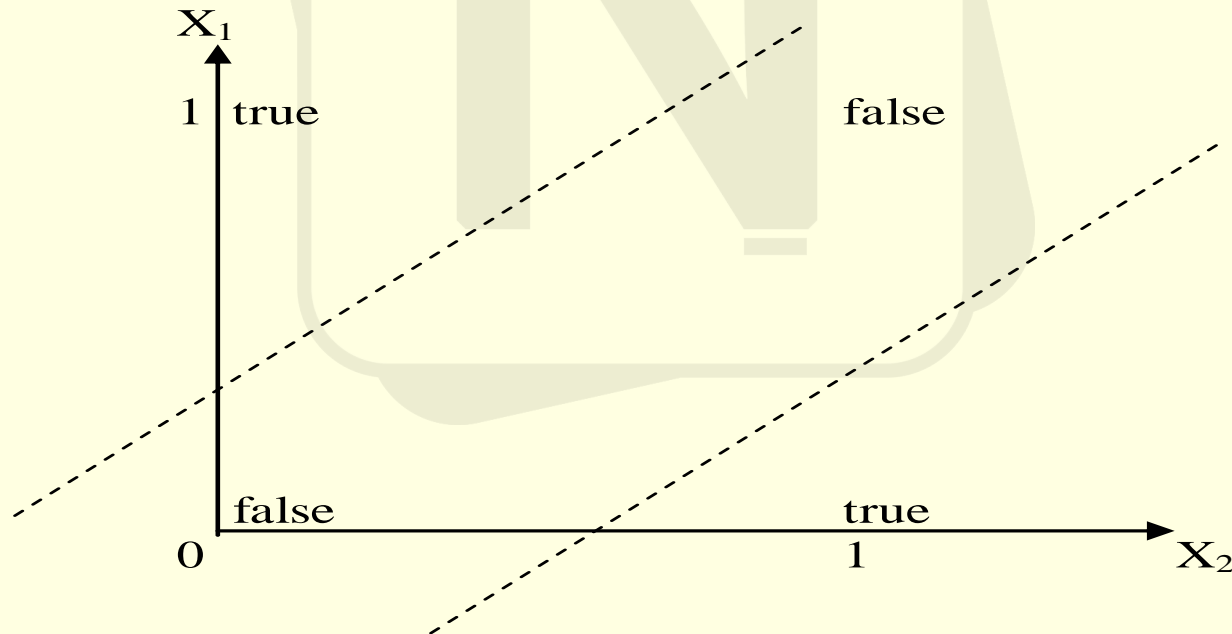
# XOR – Non linearly separable function

- A typical example of non-linearly separable function is the XOR that computes the logical **exclusive or**..
- This function takes two input arguments with values in  $\{0,1\}$  and returns one output in  $\{0,1\}$ ,
- Here 0 and 1 are encoding of the truth values **false** and **true**,
- The output is **true** if and only if the two inputs have different truth values.
- XOR is non linearly separable function which can not be modeled by perceptron.
- For such functions we have to use multi layer feed-forward network.



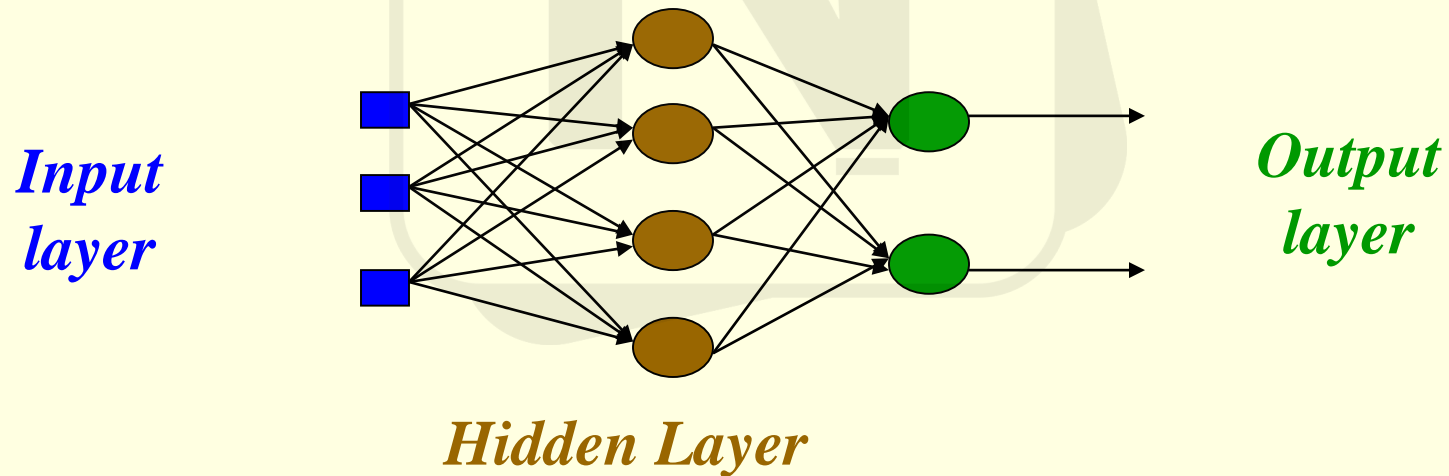
Input		Output
$X_1$	$X_2$	$X_1 \text{ XOR } X_2$
0	0	0
0	1	1
1	0	1
1	1	0

These two classes (true and false) cannot be separated using a line. Hence XOR is non linearly separable.



# Multi layer feed-forward NN (FFNN)

- FFNN is a more general network architecture, where there are hidden layers between input and output layers.
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.
- FFNNs overcome the limitation of single-layer NN.
- They can handle non-linearly separable learning tasks.

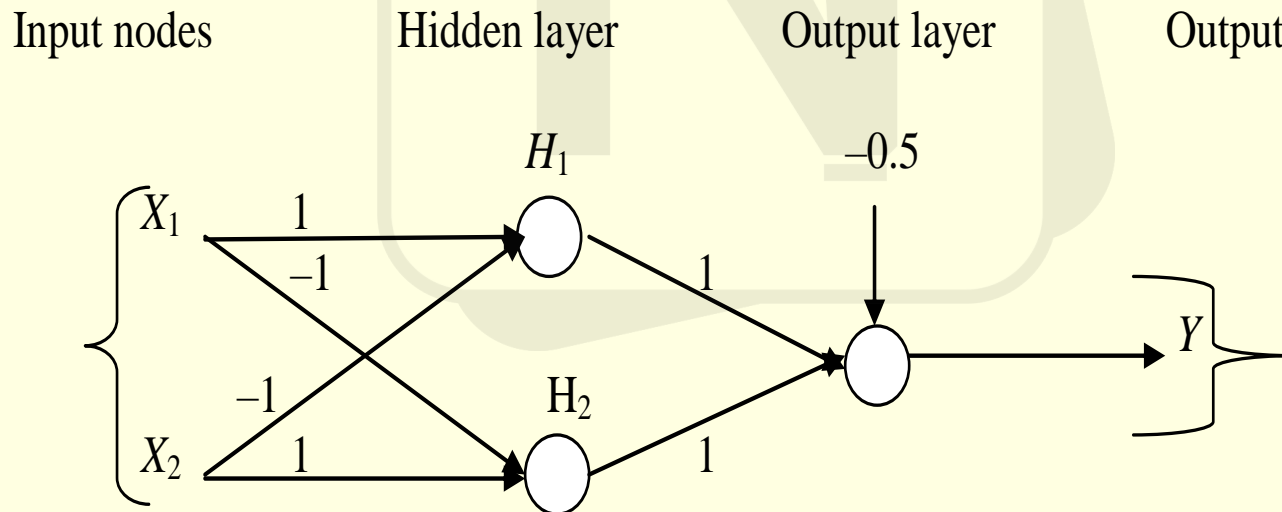


3-4-2 Network



# FFNN for XOR

- The ANN for XOR has two hidden nodes that realizes this non-linear separation and uses the sign (step) activation function.
- Arrows from input nodes to two hidden nodes indicate the directions of the weight vectors  $(1,-1)$  and  $(-1,1)$ .
- The output node is used to combine the outputs of the two hidden nodes.



Inputs		Output of Hidden Nodes		Output	$X_1 \text{ XOR } X_2$
$X_1$	$X_2$	$H_1$	$H_2$	Node	
0	0	0	0	$-0.5 \rightarrow 0$	0
0	1	$-1 \rightarrow 0$	1	$0.5 \rightarrow 1$	1
1	0	1	$-1 \rightarrow 0$	$0.5 \rightarrow 1$	1
1	1	0	0	$-0.5 \rightarrow 0$	0

Since we are representing two states by 0 (false) and 1 (true), we will map negative outputs ( $-1$ ,  $-0.5$ ) of hidden and output layers to 0 and positive output ( $0.5$ ) to 1.





# FFNN NEURON MODEL

- The classical learning algorithm of FFNN is based on the gradient descent method.
- For this reason the activation function used in FFNN are continuous functions of the weights, differentiable everywhere.
- The activation function for node  $i$  may be defined as a simple form of the **sigmoid function** in the following manner:

$$\phi(V_i) = \frac{1}{1 + e^{(-A * V_i)}}$$

where  $A > 0$ ,  $V_i = \sum W_{ij} * Y_j$ , such that  $W_{ij}$  is a weight of the link from node  $i$  to node  $j$  and  $Y_j$  is the output of node  $j$ .



# Training Algorithm: Backpropagation

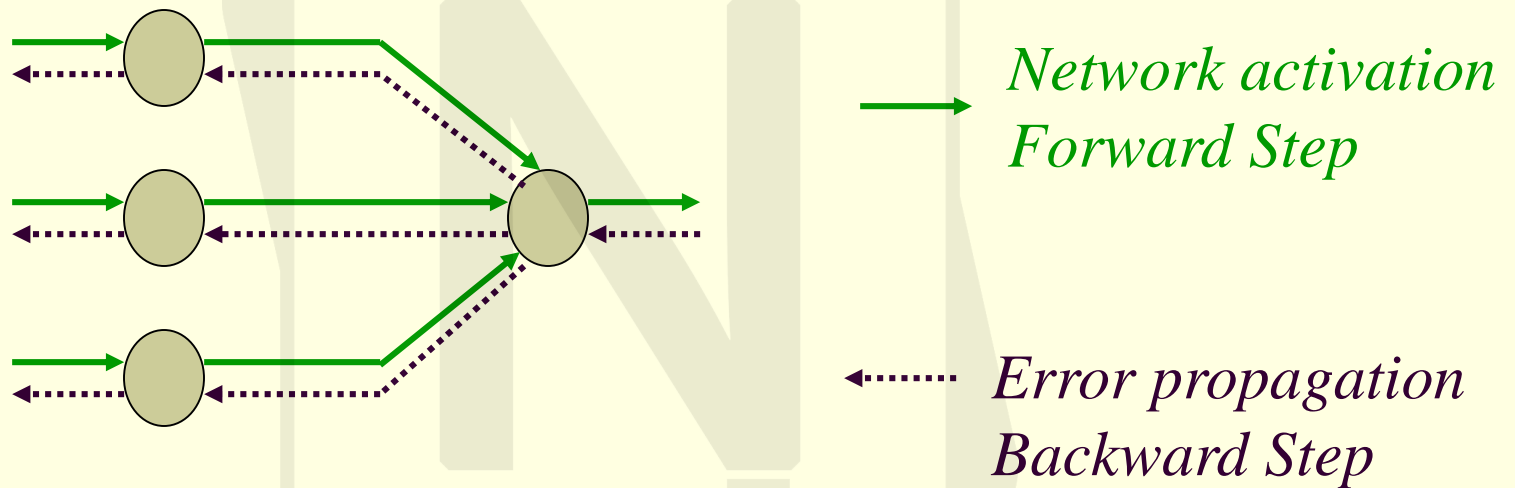
---

- The Backpropagation algorithm learns in the same way as single perceptron.
- It searches for weight values that minimize the total error of the network over the set of training examples (training set).
- Backpropagation consists of the repeated application of the following two passes:
  - **Forward pass:** In this step, the network is activated on one example and the error of (each neuron of) the output layer is computed.
  - **Backward pass:** in this step the network error is used for updating the weights. The error is propagated backwards from the output layer through the network layer by layer. This is done by recursively computing the local gradient of each neuron.



# Backpropagation

- Back-propagation training algorithm



- Backpropagation adjusts the weights of the NN in order to minimize the network total mean squared error.



## Contd..

- Consider a network of three layers.
- Let us use  $i$  to represent nodes in input layer,  $j$  to represent nodes in hidden layer and  $k$  represent nodes in output layer.
- $w_{ij}$  refers to weight of connection between a node in input layer and node in hidden layer.
- The following equation is used to derive the output value  $Y_j$  of node  $j$

$$Y_j = \frac{1}{1 + e^{-X_j}}$$

where,  $X_j = \sum x_i \cdot w_{ij} - \theta_j$ ,  $1 \leq i \leq n$ ;  $n$  is the number of inputs to node  $j$ , and  $\theta_j$  is threshold for node  $j$



# Total Mean Squared Error

- The error of output neuron  $k$  after the activation of the network on the  $n$ -th training example  $(x(n), d(n))$  is:

$$e_k(n) = d_k(n) - y_k(n)$$

- The network error is the sum of the squared errors of the output neurons:

$$E(n) = \sum e_k^2(n)$$

- The total mean squared error is the average of the network errors of the training examples.

$$E_{AV} = \frac{1}{N} \sum_{n=1}^N E(n)$$



# Weight Update Rule

- The Backprop weight update rule is based on the gradient descent method:
  - It takes a step in the direction yielding the maximum decrease of the network error E.
  - This direction is the opposite of the gradient of E.
- Iteration of the Backprop algorithm is usually terminated when the sum of squares of errors of the output values for all training data in an epoch is less than some threshold such as 0.01

$$w_{ij} = w_{ij} + \Delta w_{ij} \qquad \Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

# Backprop learning algorithm (incremental-mode)

n=1;

initialize **weights** randomly;

**while** (stopping criterion not satisfied or n < max\_iterations)

**for** each example (**x**, **d**)

- run the network with input **x** and compute the output **y**
- update the weights in backward order starting from those of the output layer:

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

with  $\Delta w_{ji}$  computed using the (generalized) Delta rule

**end-for**

  n = n+1;

**end-while;**



# Stopping criterions

---

- Total mean squared error change:
  - Back-prop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small (in the range  $[0.1, 0.01]$ ).
- Generalization based criterion:
  - After each epoch, the NN is tested for generalization.
  - If the generalization performance is adequate then stop.
  - If this stopping criterion is used then the part of the training set used for testing the network generalization will not be used for updating the weights.





# Recurrent Network

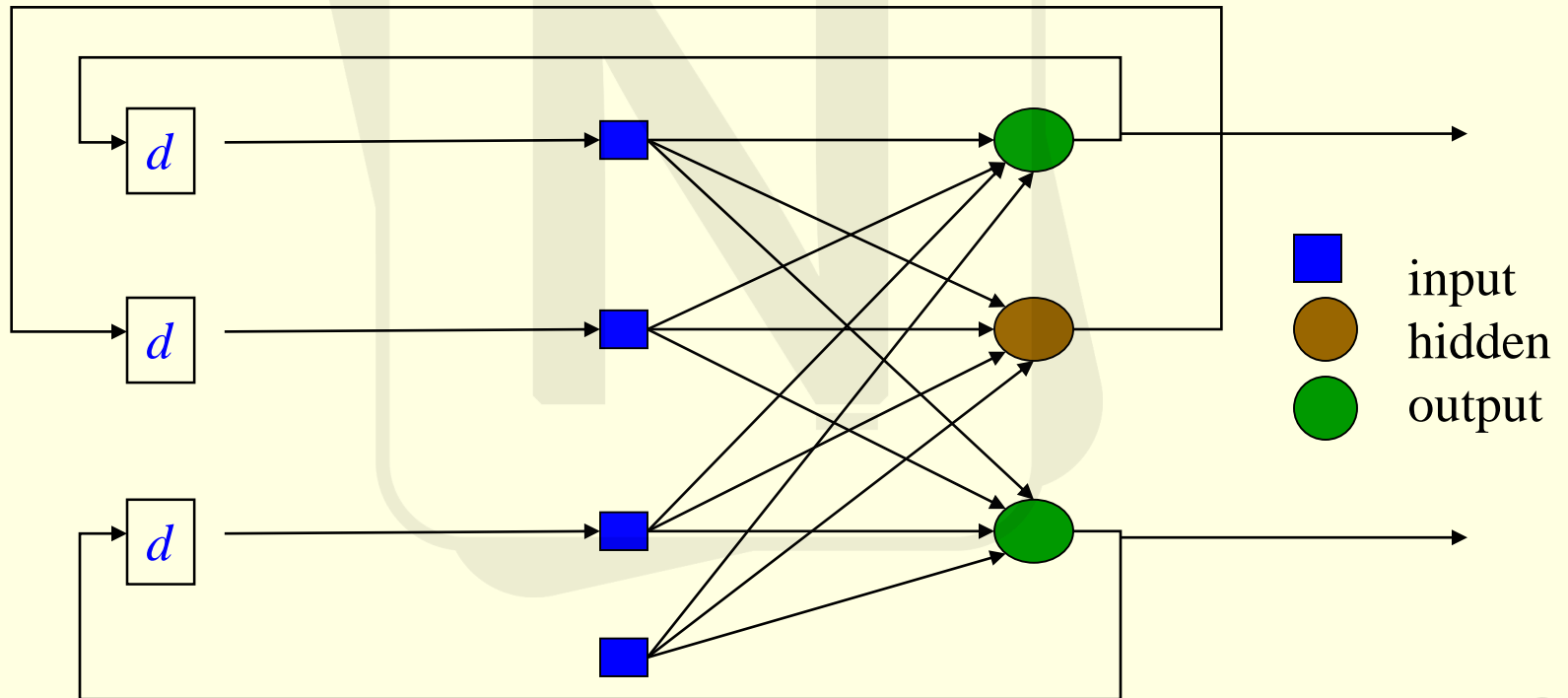
---

- **FFNN** is acyclic where data passes from input to the output nodes and not vice versa.
  - Once the FFNN is trained, its state is fixed and does not alter as new data is presented to it. It does not have memory.
- **Recurrent network** can have connections that go backward from output to input nodes and models dynamic systems.
  - In this way, a recurrent network's internal state can be altered as sets of input data are presented. It can be said to have memory.
  - It is useful in solving problems where the solution depends not just on the current inputs but on all previous inputs.
- **Applications**
  - predict stock market price,
  - weather forecast



# Recurrent Network Architecture

- Recurrent Network with **hidden neuron**: unit delay operator  $d$  is used to model a dynamic system



# Learning and Training

---

- During learning phase,
  - a recurrent network feeds its inputs through the network, including feeding data back from outputs to inputs
  - process is repeated until the values of the outputs do not change.
- This state is called equilibrium or stability
- Recurrent networks can be trained by using back-propagation algorithm.
- In this method, at each step, the activation of the output is compared with the desired activation and errors are propagated backward through the network.
- Once this training process is completed, the network becomes capable of performing a sequence of actions.



# Hopfield Network

- A **Hopfield network** is a kind of recurrent network as output values are fed back to input in an undirected way.
  - It consists of a set of  $N$  connected neurons with weights which are symmetric and no unit is connected to itself.
  - There are no special input and output neurons.
  - The activation of a neuron is binary value decided by the sign of the weighted sum of the connections to it.
  - A threshold value for each neuron determines if it is a firing neuron.
  - A firing neuron is one that activates all neurons that are connected to it with a positive weight.
  - The input is simultaneously applied to all neurons, which then output to each other.
  - This process continues until a stable state is reached.



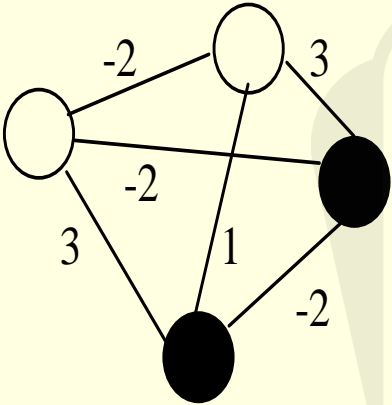
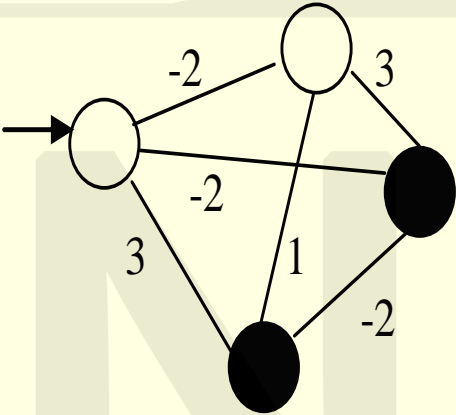
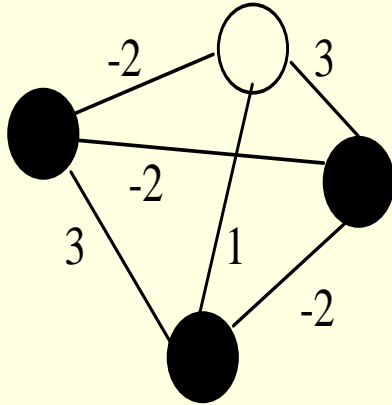
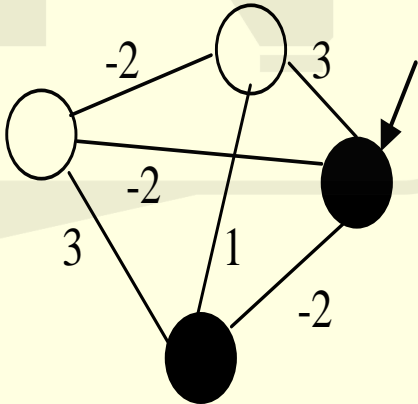
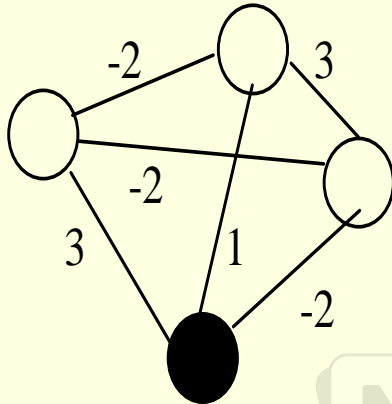
# Activation Algorithm

---

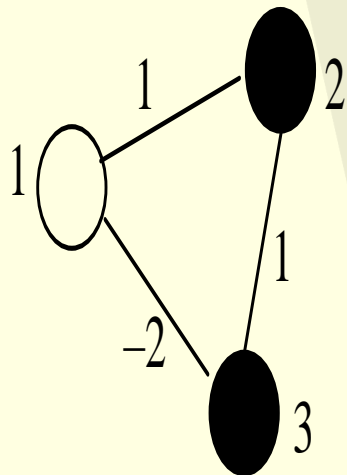
Active unit represented by 1 and inactive by 0.

- *Repeat*
  - Choose any unit randomly. The chosen unit may be active or inactive.
  - For the chosen unit, compute the sum of the weights on the connection to the active neighbours only, if any.
    - If  $\text{sum} > 0$  (threshold is assumed to be 0), then the chosen unit becomes active, otherwise it becomes inactive.
  - If chosen unit has no active neighbours then ignore it, and status remains same.
- *Until* the network reaches to a stable state

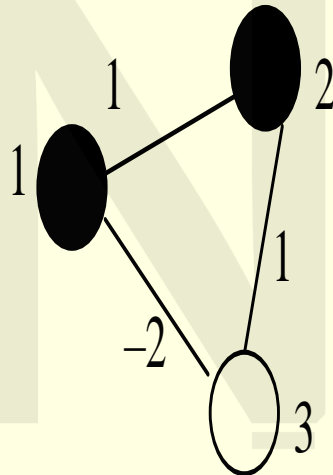


Current State	→ Selected Unit from current state	Corresponding New State
 <p>Here, the sum of weights of active neighbours of a selected unit is calculated.</p>		 <p><math>\text{Sum} = 3 - 2 = 1 &gt; 0;</math> activated</p>
		 <p><math>\text{Sum} = -2 &lt; 0;</math> deactivated</p>

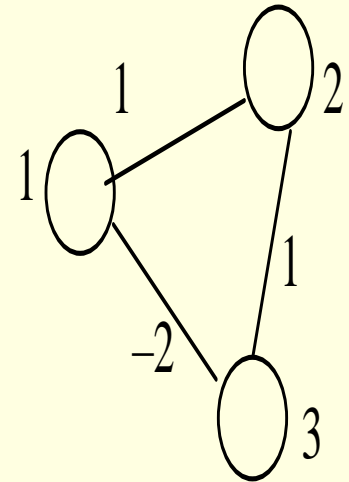
# Stable Networks



$$X = [0 \ 1 \ 1]$$



$$X = [1 \ 1 \ 0]$$



$$X = [0 \ 0 \ 0]$$

# Weight Computation Method

- Weights are determined using training examples.

$$W = \sum X_i \cdot (X_i)^T - M.I, \text{ for } 1 \leq i \leq M$$

- Here
  - $W$  is weight matrix
  - $X_i$  is an input example represented by a vector of  $N$  values from the set  $\{-1, 1\}$ .
    - Here,  $N$  is the number of units in the network; 1 and -1 represent active and inactive units respectively.
  - $(X_i)^T$  is the transpose of the input  $X_i$ ,
  - $M$  denotes the number of training input vectors,
  - $I$  is an  $N \times N$  identity matrix.





# Example

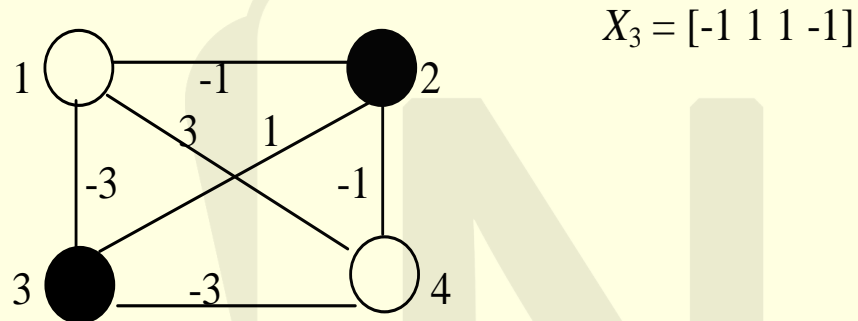
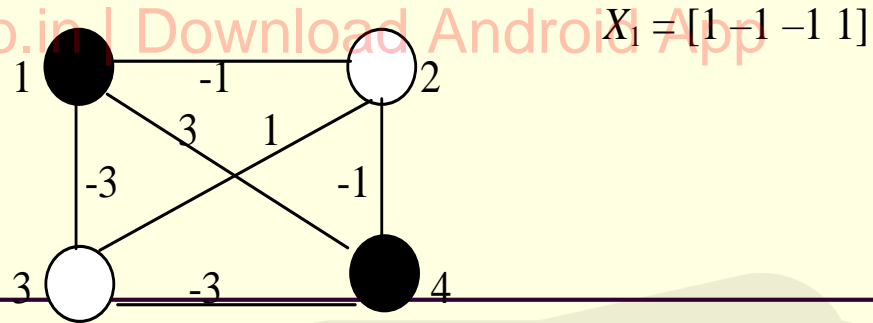
- Let us now consider a Hopfield network with four units and three training input vectors that are to be learned by the network.
- Consider three input examples, namely,  $X_1$ ,  $X_2$ , and  $X_3$  defined as follows:

$$X_1 = \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix}$$

$$X_2 = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix}$$

$$X_3 = \begin{pmatrix} -1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

$$W = X_1 \cdot (X_1)^T + X_2 \cdot (X_2)^T + X_3 \cdot (X_3)^T - 3.I$$



Stable positions of the network

$$W = \begin{pmatrix} 3 & -1 & -3 & 3 \\ -1 & 3 & 1 & -1 \\ -3 & 1 & 3 & -3 \\ 3 & -1 & -3 & 3 \end{pmatrix} - \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 0 & -1 & -3 & 3 \\ -1 & 0 & 1 & -1 \\ -3 & 1 & 0 & -3 \\ 3 & -1 & -3 & 0 \end{pmatrix}$$

## Contd..

---

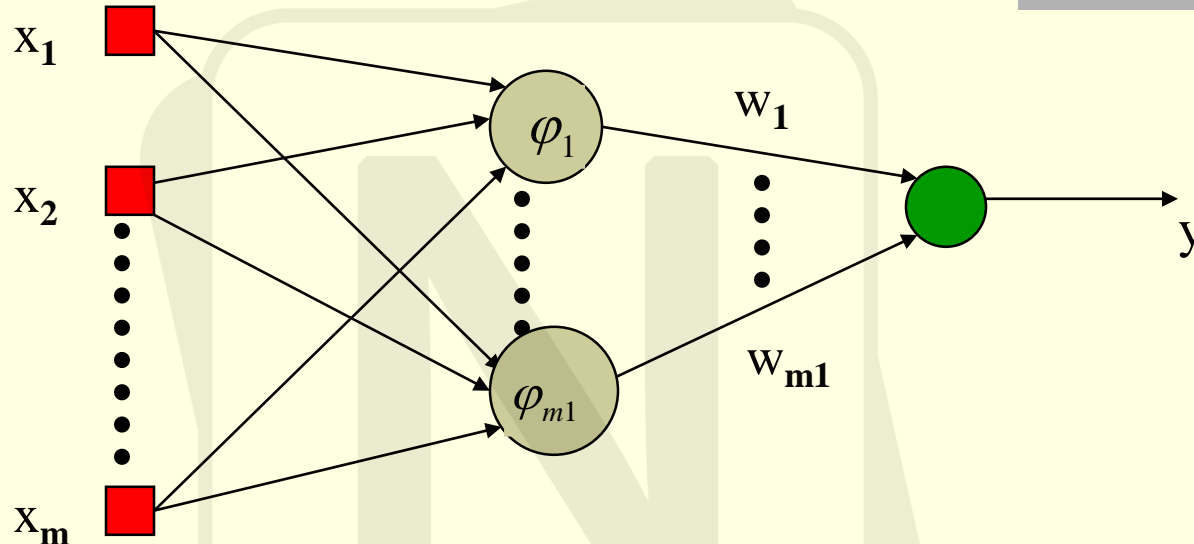
- The networks generated using these weights and input vectors are stable, except X2.
- X2 stabilizes to X1 (which is at hamming distance 1).
- Finally, with the obtained weights and stable states (X1 and X3), we can stabilize any new (partial) pattern to one of those



# Radial-Basis Function Networks

- A function is said to be a ***radial basis function*** (RBF) if its output depends on the distance of the input from a given stored vector.
  - The RBF neural network has an input layer, a hidden layer and an output layer.
  - In such RBF networks, the hidden layer uses neurons with RBFs as activation functions.
  - The outputs of all these hidden neurons are combined linearly at the output node.
- These networks have a wide variety of applications such as
  - function approximation,
  - time series prediction,
  - control and regression,
  - pattern classification tasks for performing complex (non-linear).

# RBF Architecture



- One hidden layer with RBF activation functions

$$\varphi_1 \dots \varphi_{m1}$$

- Output layer with linear activation function.

$$y = w_1 \varphi_1(\|x - t_1\|) + \dots + w_{m1} \varphi_{m1}(\|x - t_{m1}\|)$$

$\|x - t\|$  distance of  $x = (x_1, \dots, x_m)$  from center  $t$



## Cont...

- Here we require weights,  $w_i$  from the hidden layer to the output layer only.
- The weights  $w_i$  can be determined with the help of any of the standard iterative methods described earlier for neural networks.
- However, since the approximating function given below is linear w. r. t.  $w_i$ , it can be directly calculated using the matrix methods of linear least squares without having to explicitly determine  $w_i$  iteratively.

$$Y = f(X) = \sum_{i=1}^N w_i \varphi(\|X_i - t_i\|)$$

- It should be noted that the approximate function  $f(X)$  is differentiable with respect to  $w_i$ .



# Comparison

RBF NN	FF NN
<i>Non-linear layered feed-forward networks.</i>	<i>Non-linear layered feed-forward networks</i>
Hidden layer of RBF is <i>non-linear</i> , the output layer of RBF is <i>linear</i> .	Hidden and output layers of FFNN are usually <i>non-linear</i> .
One <i>single</i> hidden layer	May have <i>more</i> hidden layers.
Neuron model of the hidden neurons is <i>different</i> from the one of the output nodes.	Hidden and output neurons share a <i>common neuron model</i> .
Activation function of each hidden neuron in a RBF NN computes the <i>Euclidean distance</i> between input vector and the center of that unit.	Activation function of each hidden neuron in a FFNN computes the <i>inner product</i> of input vector and the synaptic weight vector of that neuron