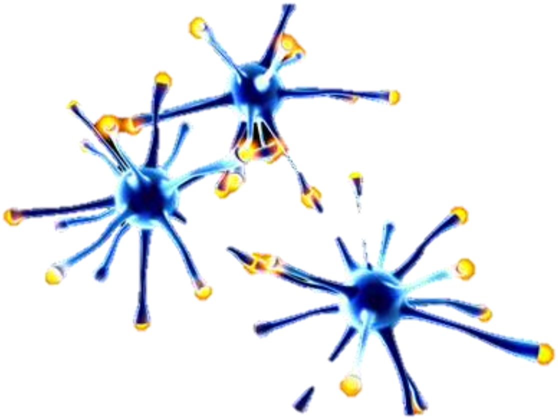
An Introduction to Artificial Neural Network

Dr Iman Ardekani

Unitec Institute of Technology

Content



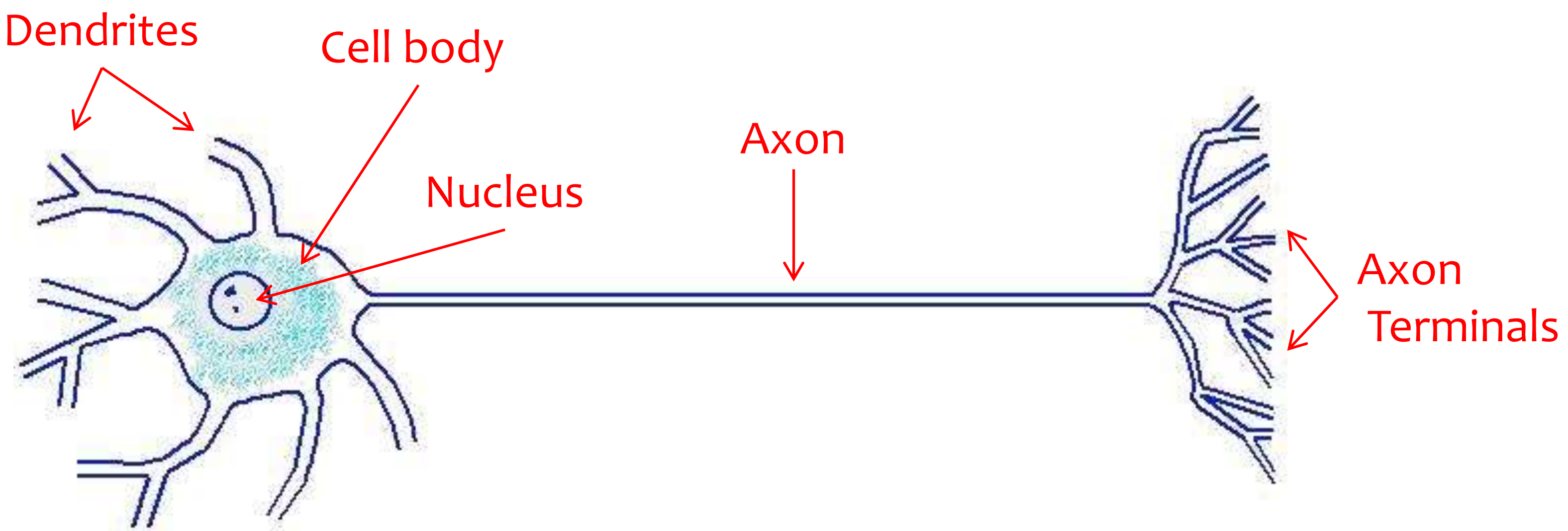
Biological Neurons

Ramón-y-Cajal (Spanish Scientist, 1852~1934):

1. **Brain is composed of individual cells called *neurons*.**
2. **Neurons are connected to each others by *synopses*.**

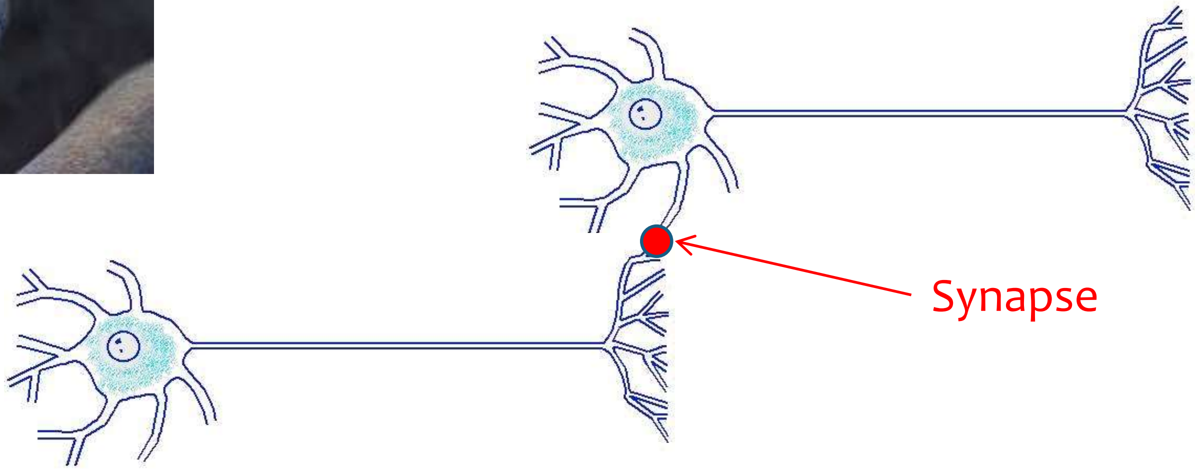
Biological Neurons

Neurons Structure (Biology)



Biological Neurons

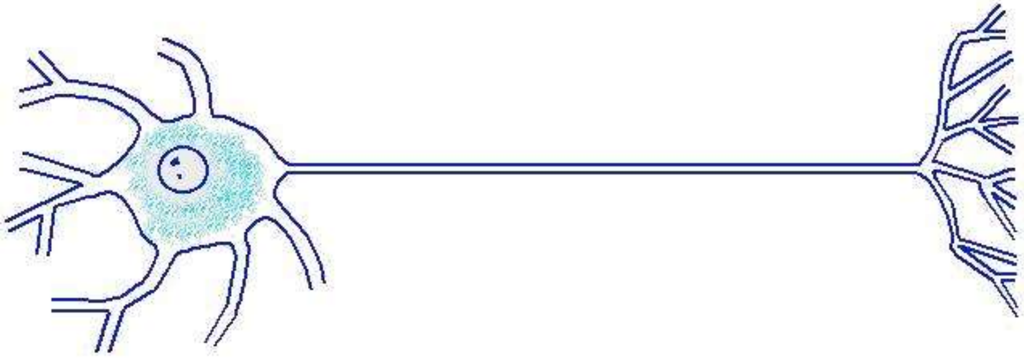
Synaptic Junction (Biology)



Biological Neurons

Neurons Function (Biology)

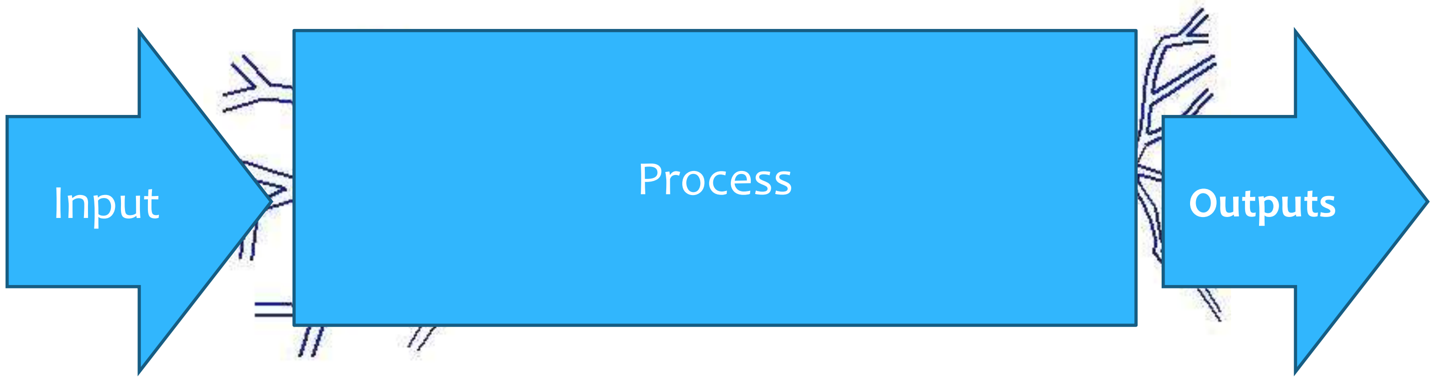
1. **Dendrites receive signal from other neurons.**
2. **Neurons can process (or transfer) the received signals.**
3. **Axon terminals deliverer the processed signal to other tissues.**



What kind of signals? Electrical Impulse Signals

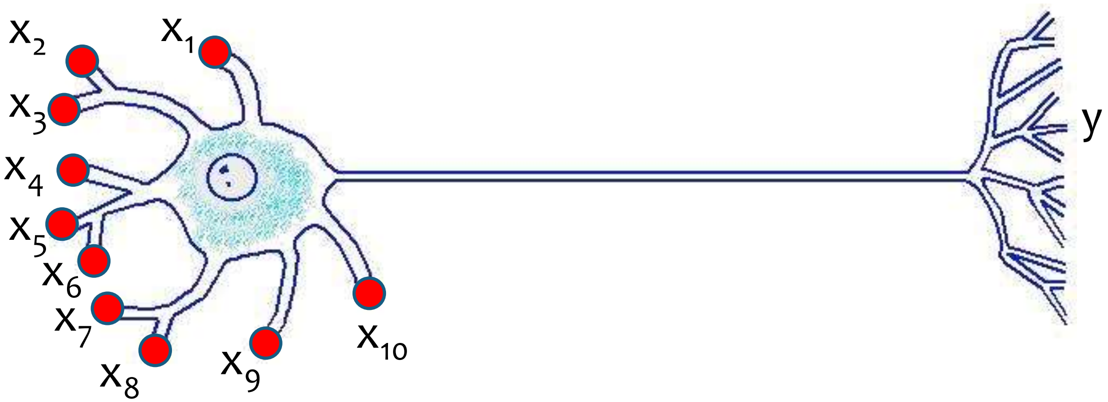
Biological Neurons

Modeling Neurons (Computer Science)



Biological Neurons

Modeling Neurons



Net input signal is a linear combination of input signals x_i.

Each Output is a function of the net input signal.

Modelling Neurons

- **McCulloch and Pitts (1943)** for introducing the idea of neural networks as computing machines
- **Hebb (1949**) for inventing the first rule for self-organized learning
- **Rosenblass (1958)** for proposing the perceptron as the first model for learning with a teacher

Modelling Neurons

Net input signal received through synaptic junctions is

$$net = b + \sum w_i x_i = b + W^T x$$

$$\text{Weight vector:} \quad W = [w_1\ w_2\ \dots\ w_m]^T$$

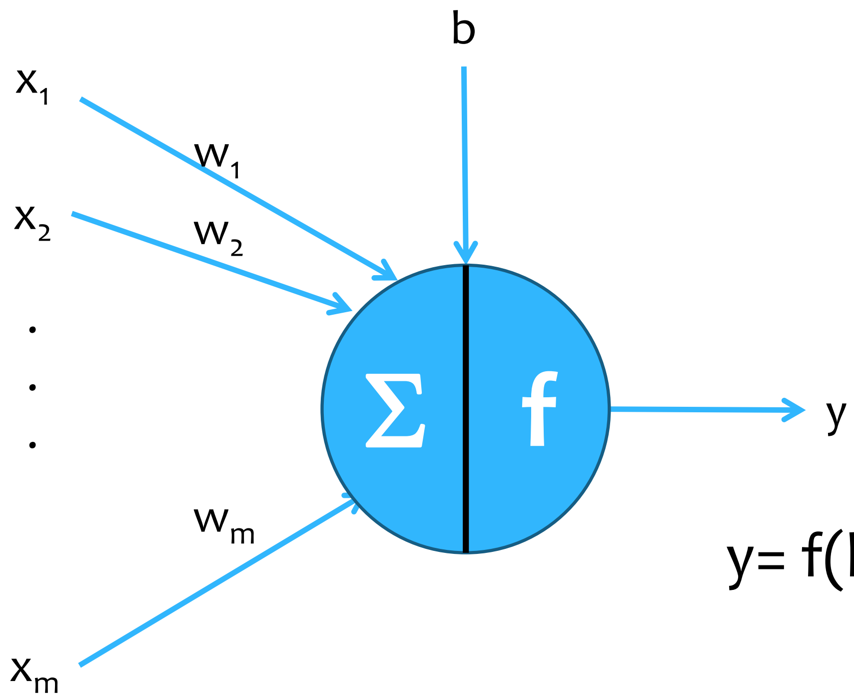
$$\text{Input vector:} \quad X = [x_1\ x_2\ \dots\ x_m]^T$$

Each output is a function of the net stimulus signal (f is called the activation function)

$$y = f\ (net) = f(b + W^T X)$$

Modelling Neurons

General Model for Neurons

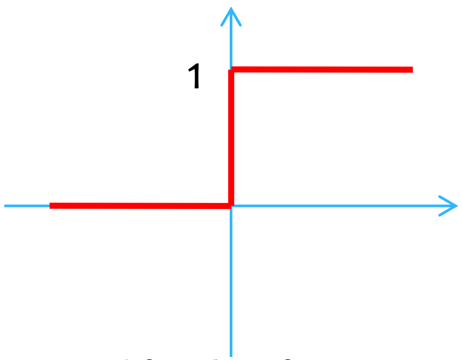


$$y = f(b + \Sigma w_n x_n)$$

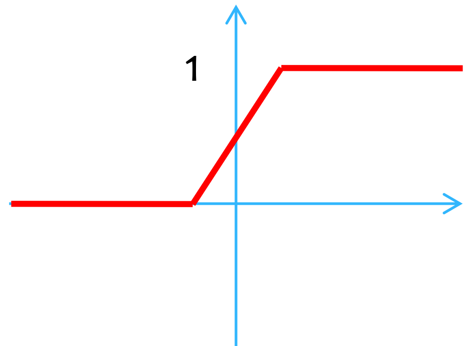
Modelling Neurons

Activation functions

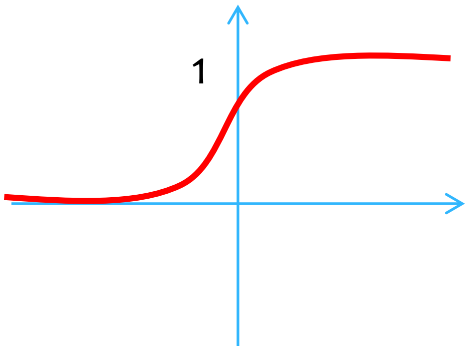
Threshold Function/ Hard Limiter



Linear Function



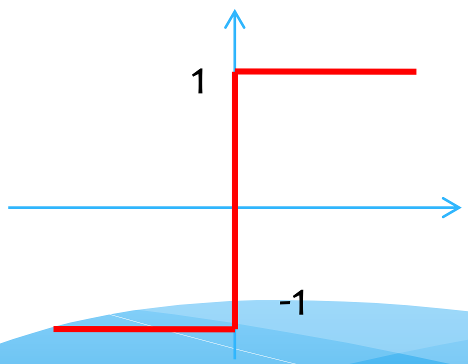
sigmoid Function

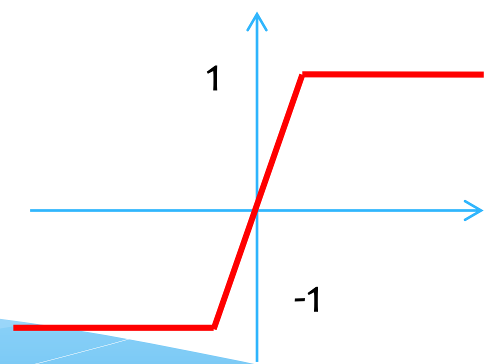


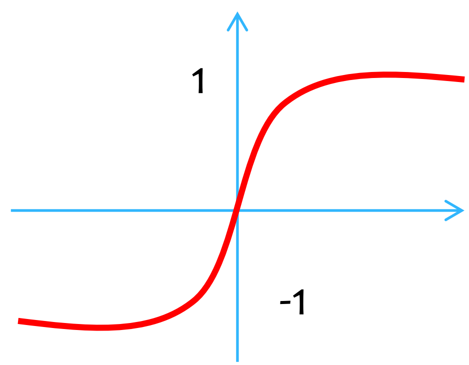
Good for classification

Simple computation

Continuous & Differentiable

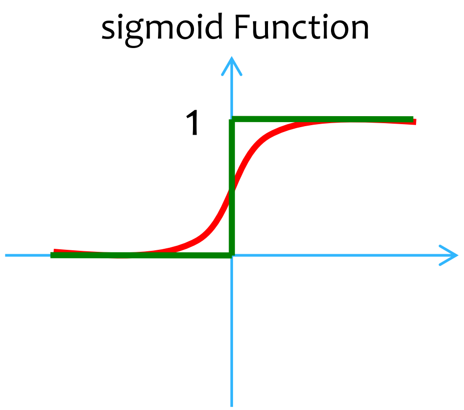






Modelling Neurons

Sigmoid Function

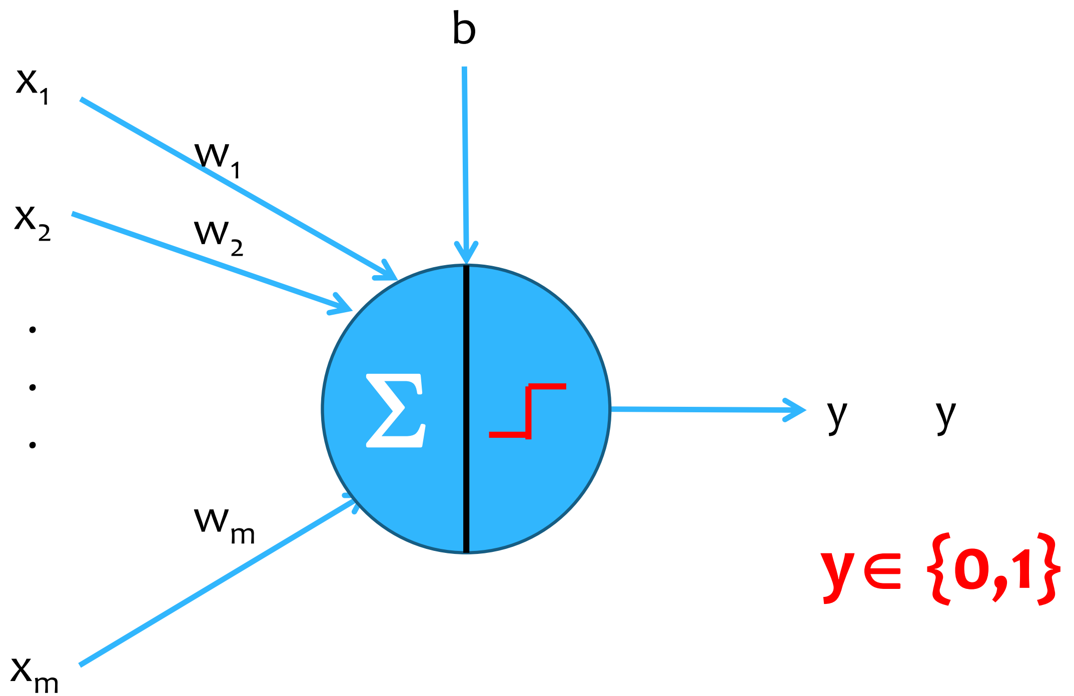


$$f(x) = \frac{1}{1 + e^{-ax}}$$

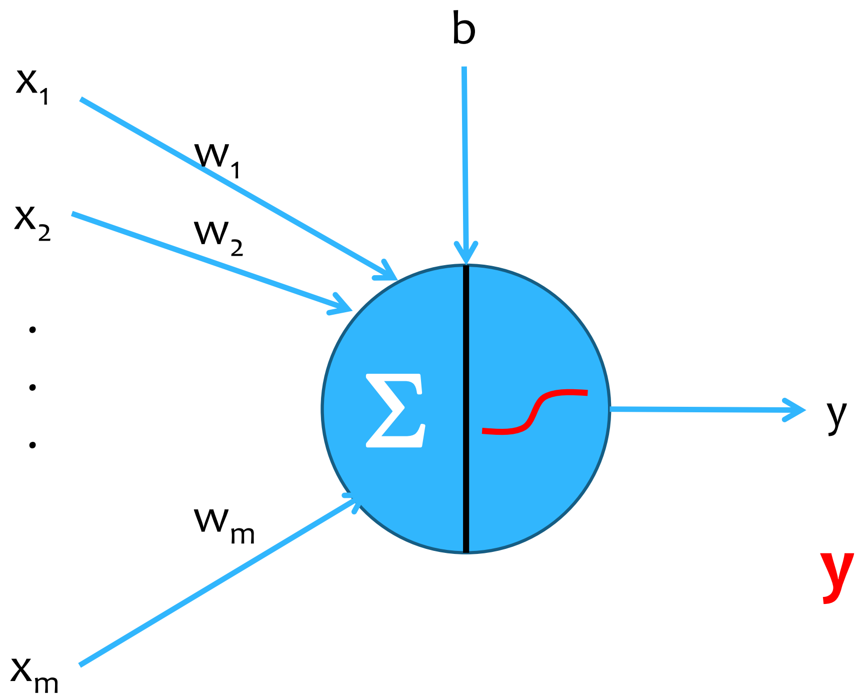
= threshold function
when *a* goes to infinity

Modelling Neurons

McCulloch-Pitts Neuron



Modelling Neurons

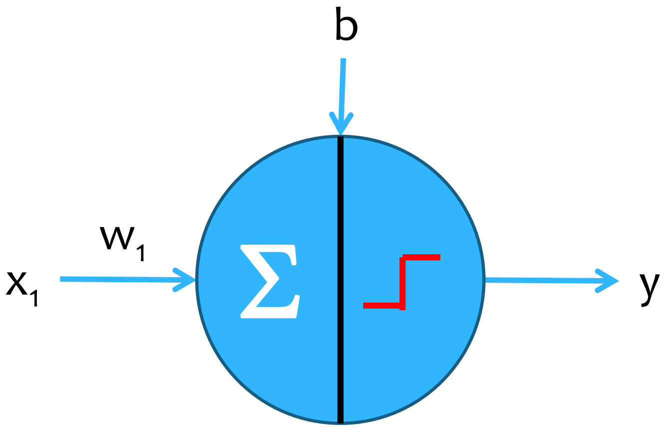


$y \in [0,1]$

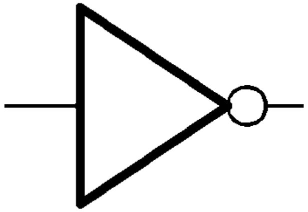
McCulloch-Pitts Neuron

Single-input McCulloch-Pitts neurode with $b=0$, $w_1=-1$ for binary inputs:

x_1	net	y
0	0	1
1	-1	0



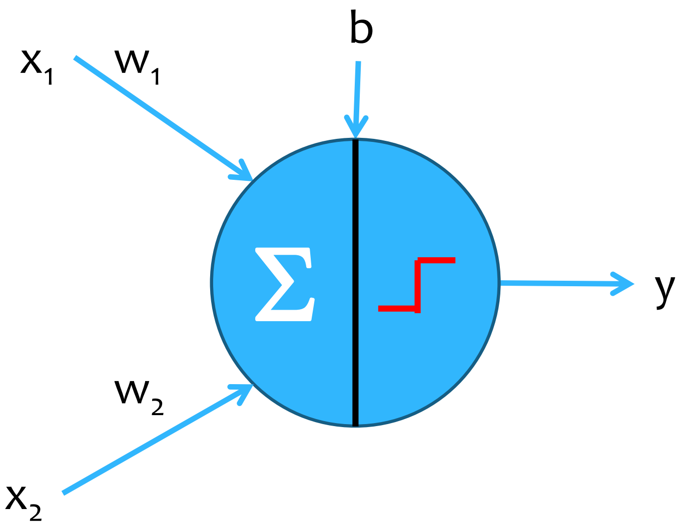
Conclusion?



McCulloch-Pitts Neuron

Two-input McCulloch-Pitts neurode with b=-1, $w_1=w_2=1$ for binary inputs:

x_1	x_2	net	y
0	0	?	?
0	1	?	?
1	0	?	?
1	1	?	?

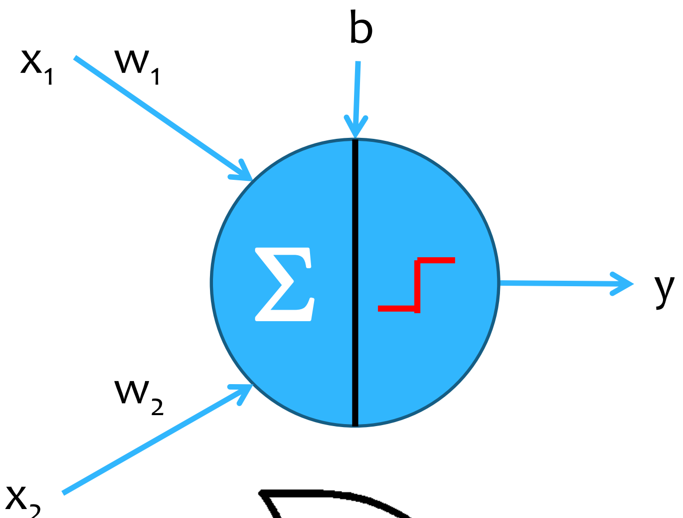


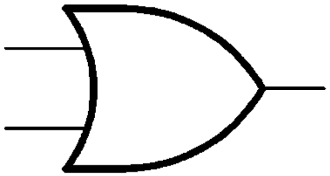
McCulloch-Pitts Neuron

CLASSROOM ACTVITIES

Two-input McCulloch-Pitts neurode with $b=-1$, $w_1=w_2=1$ for binary inputs:

x_1	x_2	net	y
0	0	-1	0
0	1	0	1
1	0	0	1
1	1	1	1

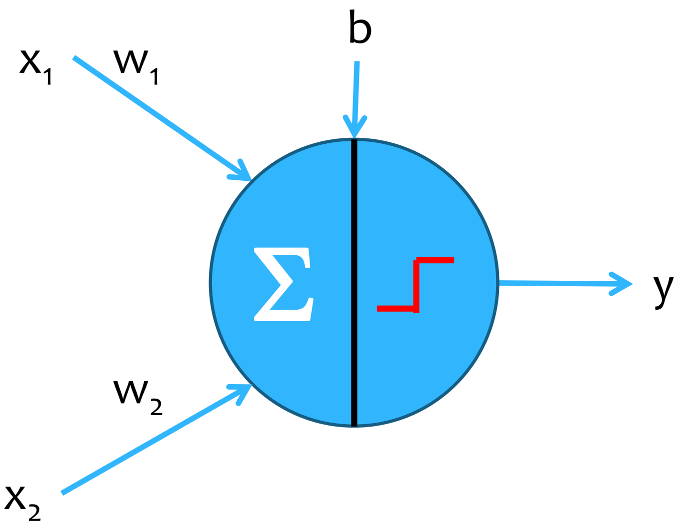




McCulloch-Pitts Neuron

Two-input McCulloch-Pitts neurode with b=-2, $w_1=w_2=1$ for binary inputs :

x_1	x_2	net	y
0	0	?	?
0	1	?	?
1	0	?	?
1	1	?	?

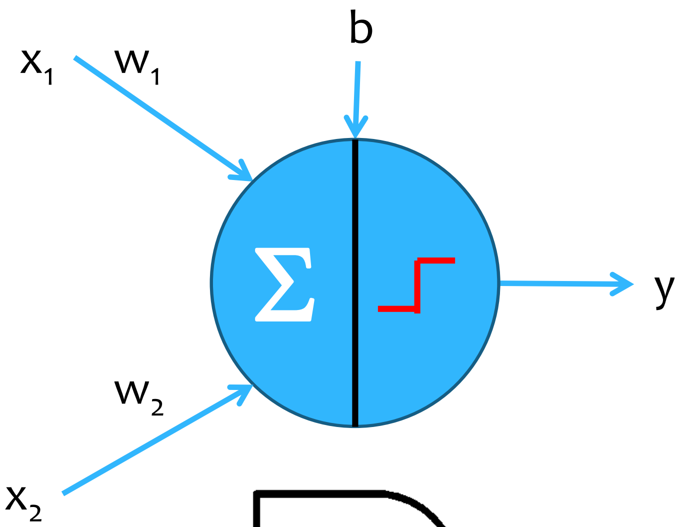


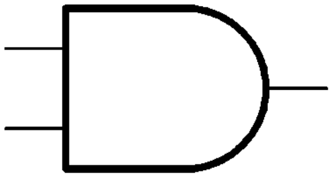
McCulloch-Pitts Neuron

CLASSROOM ACTVITIES

Two-input McCulloch-Pitts neurode with b=-2, w_1=w_2=1 for binary inputs :

x_1	x_2	net	y
0	0	-2	0
0	1	-1	0
1	0	-1	0
1	1	0	1





McCulloch-Pitts Neuron

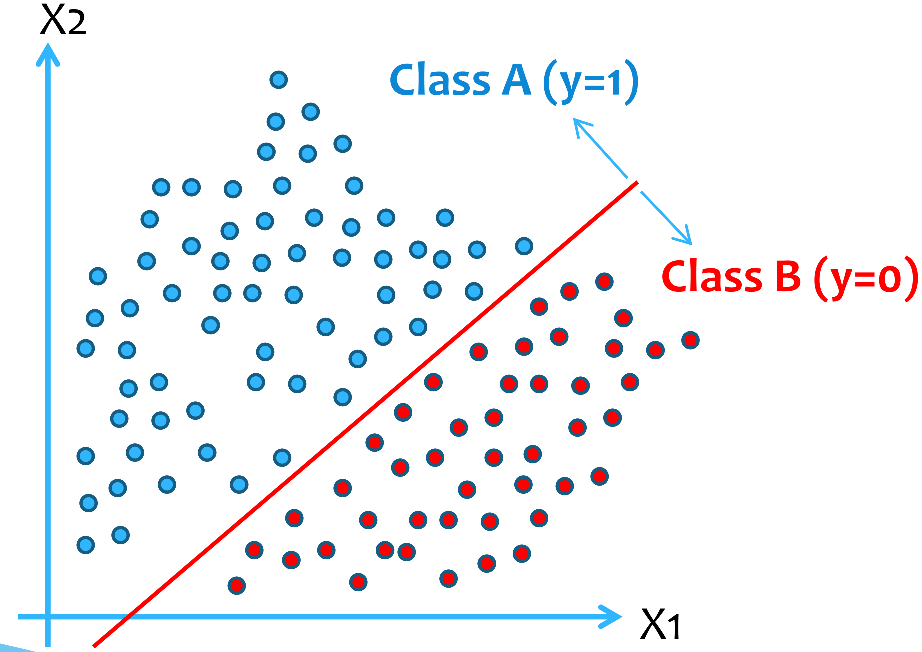
Every *basic* Boolean function can be implemented using combinations of McCulloch-Pitts Neurons.

McCulloch-Pitts Neuron

the McCulloch-Pitts neuron can be used as a classifier that separate the input signals into two classes (perceptron):

Class A $\Leftrightarrow$ y=?
y = 1 $\Leftrightarrow$ net = ?
net $\geq$ 0 $\Leftrightarrow$?
$\mathbf{b+w_1x_1+w_2x_2 \geq 0}$

Class B $\Leftrightarrow$ y=?
y = 0 $\Leftrightarrow$ net = ?
net < 0 $\Leftrightarrow$?
$\mathbf{b+w_1x_1+w_2x_2 < 0}$



McCulloch-Pitts Neuron

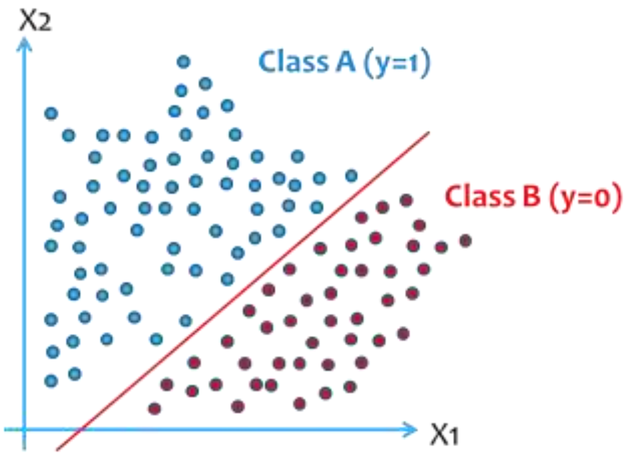
Class A $\Leftrightarrow$ $b+w_1x_1+w_2x_2 \geq 0$

Class B $\Leftrightarrow$ $b+w_1x_1+w_2x_2 < 0$

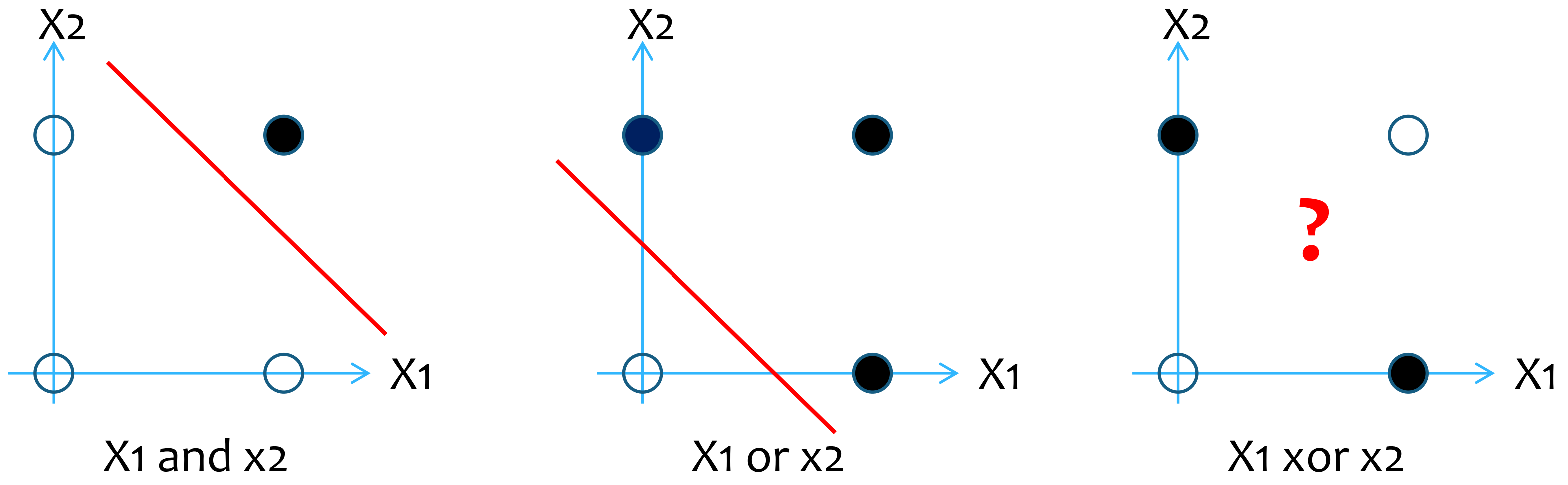
Therefore, the decision boundary is a hyperline given by

$$b+w_1x_1+w_2x_2 = 0$$

Where w_1 and w_2 come from?



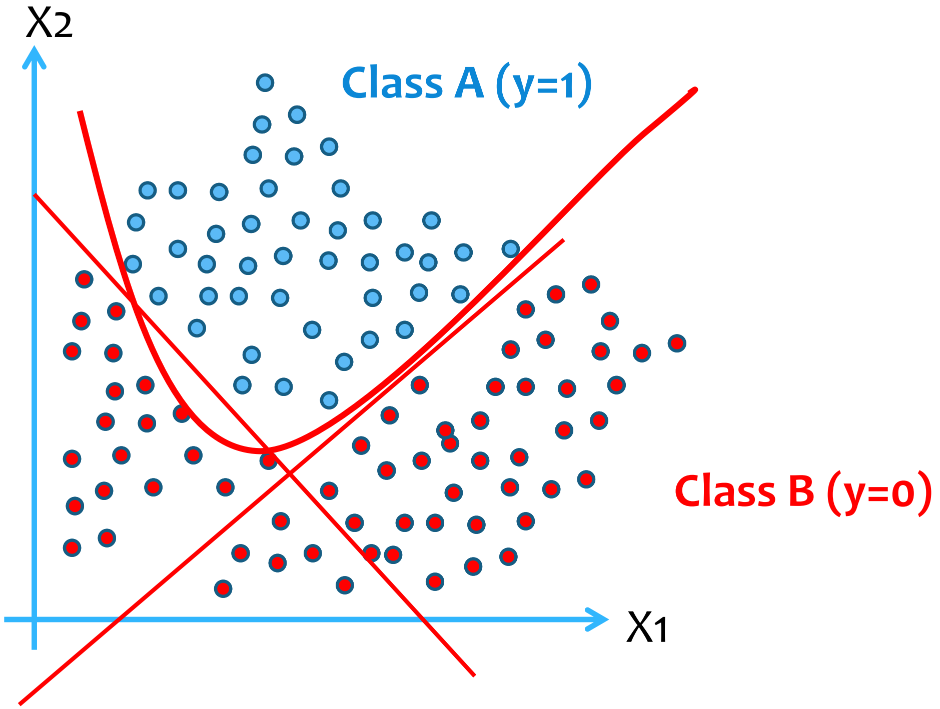
McCulloch-Pitts Neuron



Solution: More Neurons Required

McCulloch-Pitts Neuron

Nonlinear Classification

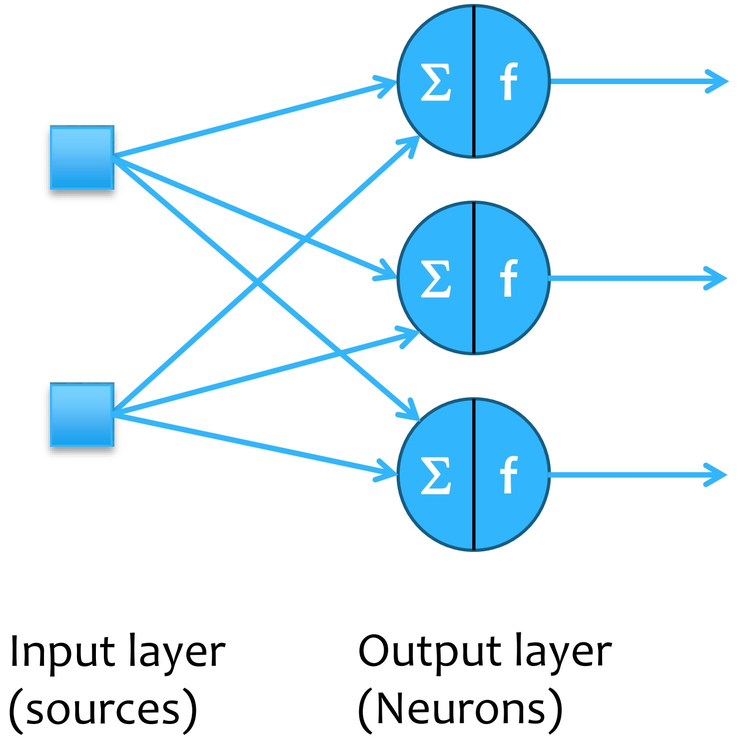


Network Architecture

Single Layer Feed-forward Network

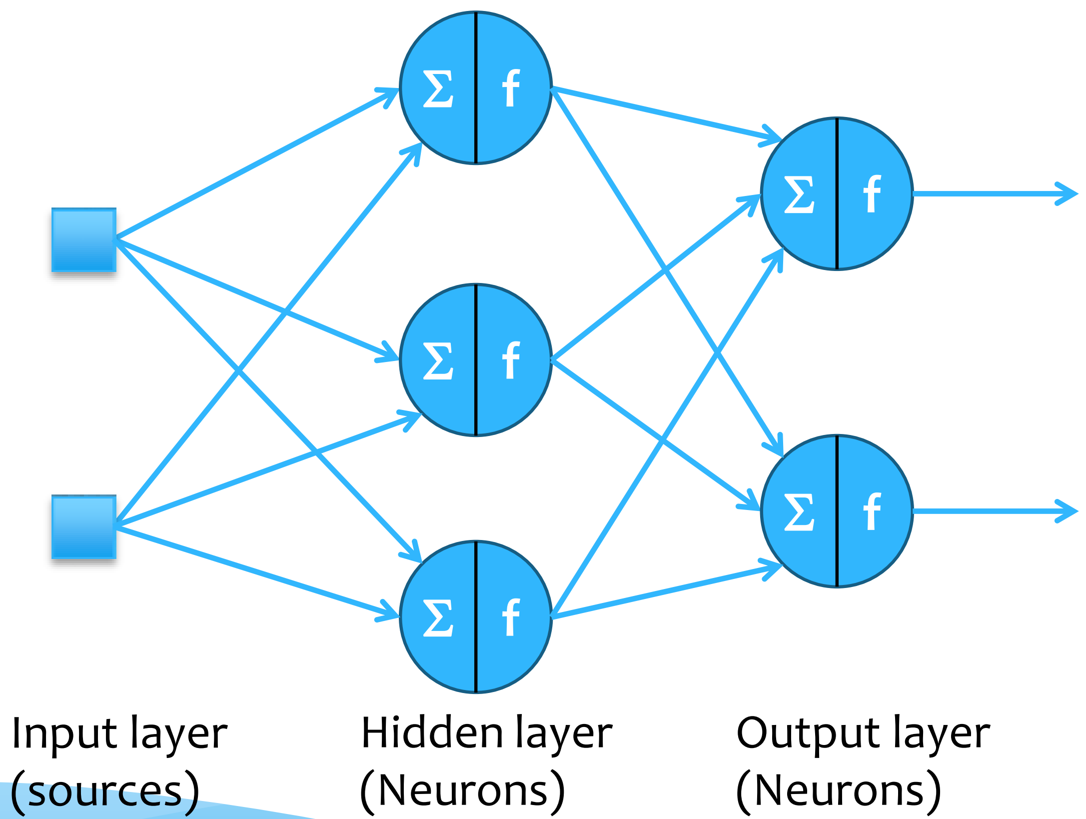
Single Layer:
There is only one computational layer.

Feed-forward:
Input layer projects to the output layer not vice versa.



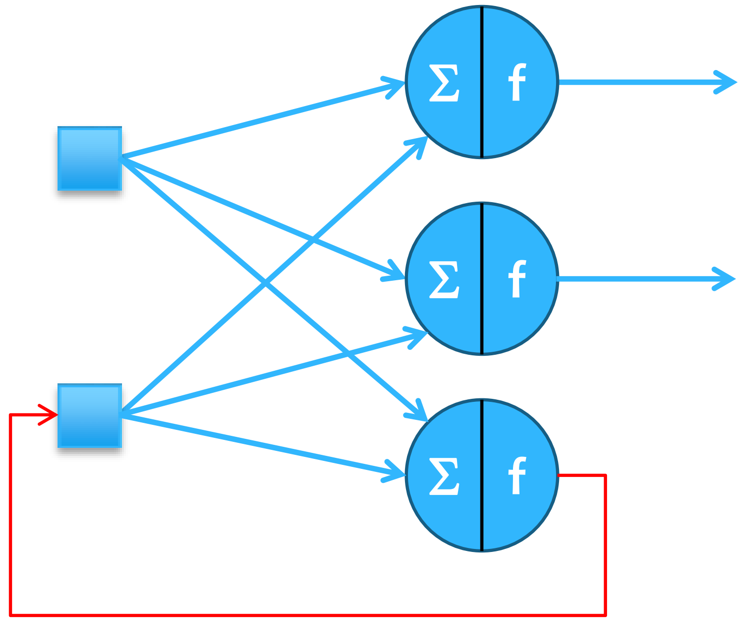
Network Architecture

Multi Layer Feed-forward Network



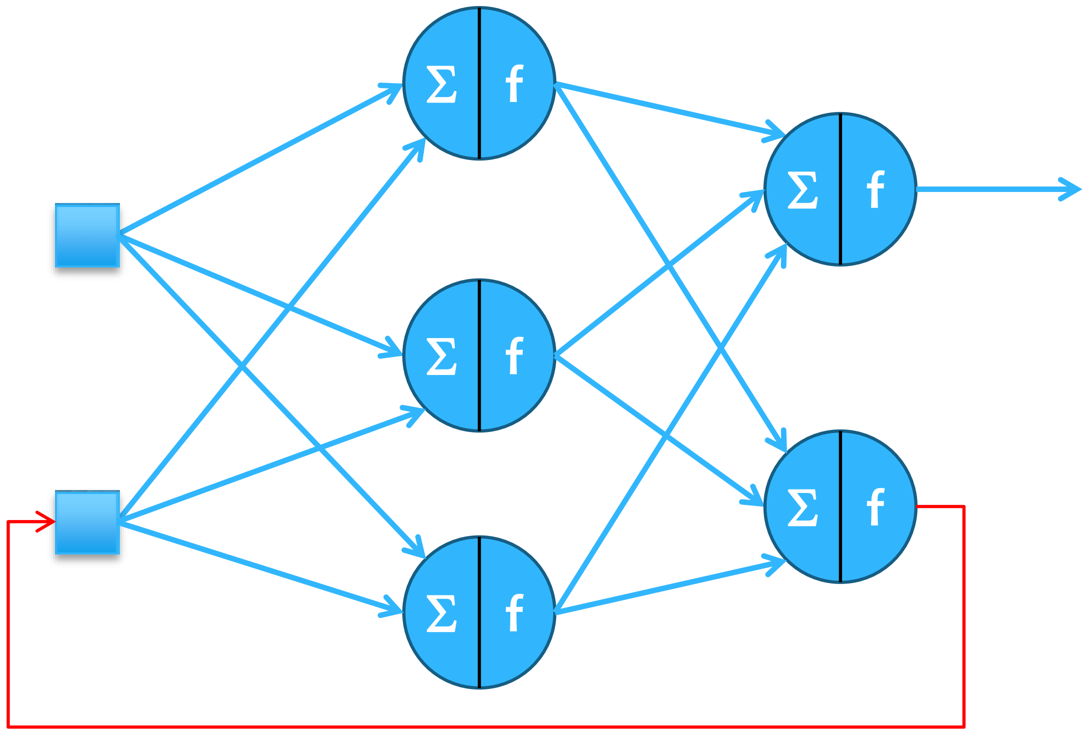
Network Architecture

Single Layer Recurrent Network



Network Architecture

Multi Layer Recurrent Network



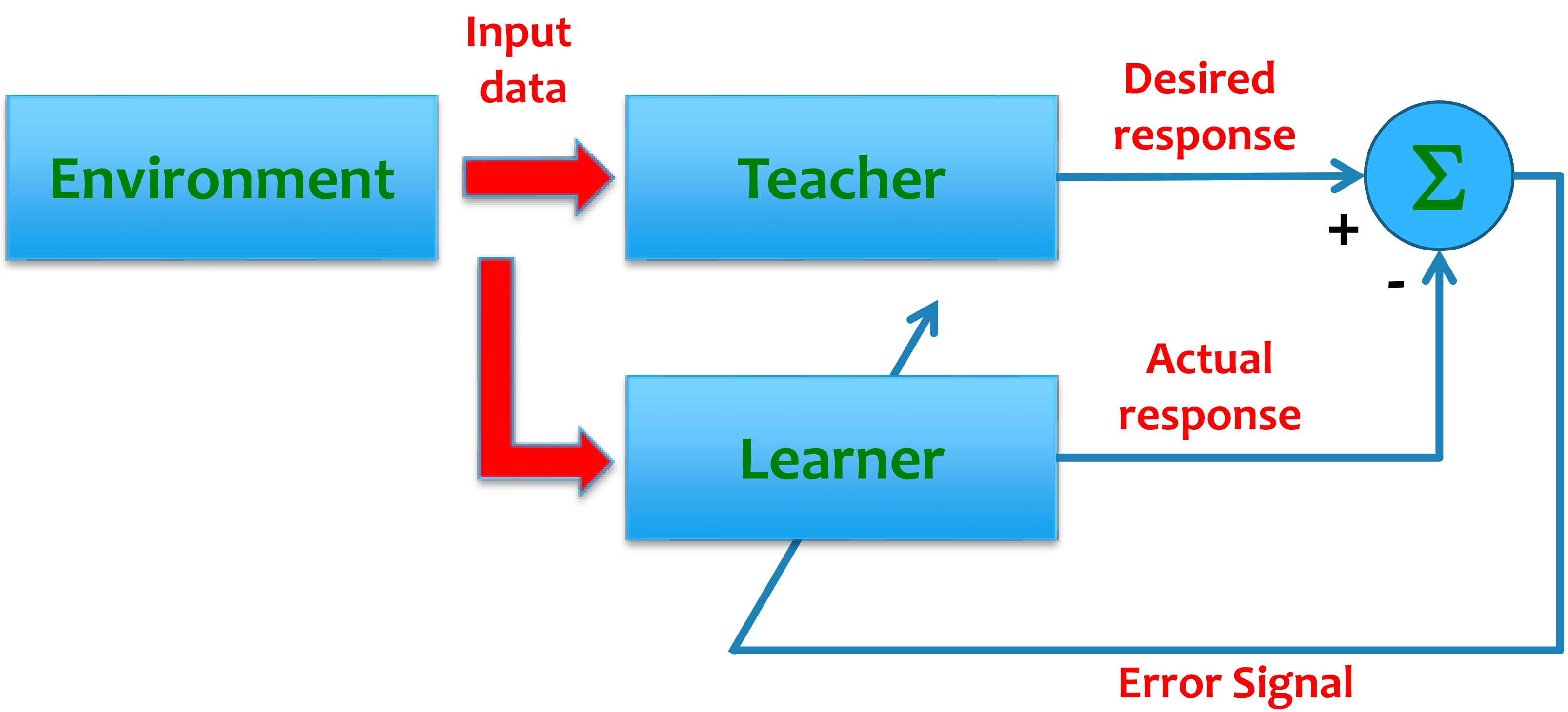
Learning Processes

The mechanism based on which a neural network can adjust its weights (synaptic junctions weights):

- Supervised learning: having a teacher
- Unsupervised learning: without teacher

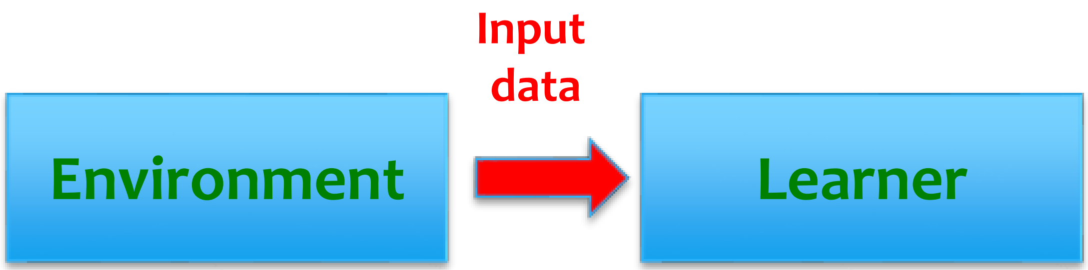
Learning Processes

Supervised Learning



Learning Processes

Unsupervised Learning

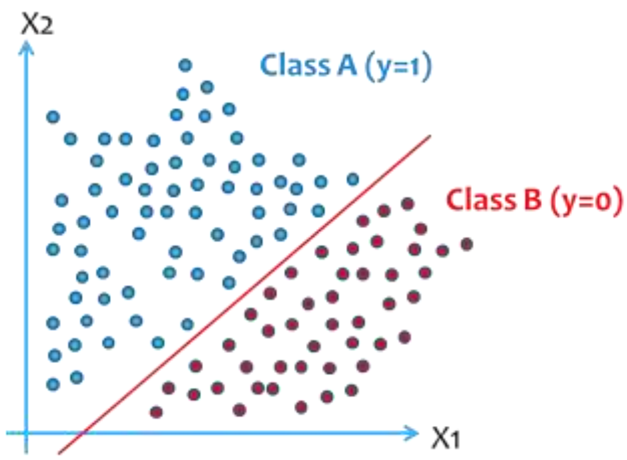


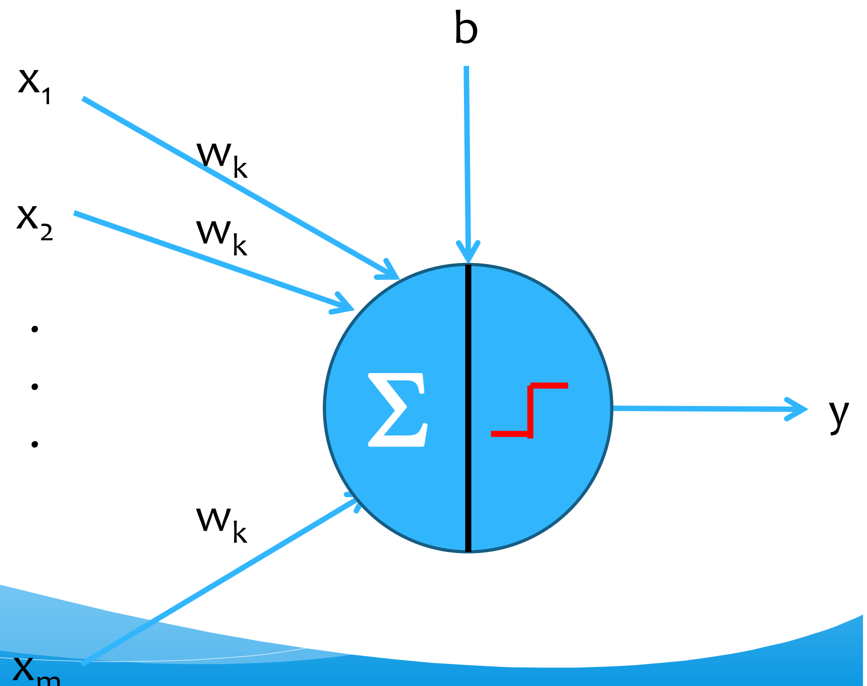
Neurons learn based on a competitive task.

A competition rule is required (competitive-learning rule).

Perceptron

- The goal of the perceptron to classify input data into two classes A and B
- Only when the two classes can be separated by a linear boundary
- The perceptron is built around the McCulloch-Pitts Neuron model
- A linear combiner followed by a hard limiter
- Accordingly the neuron can produce +1 and 0

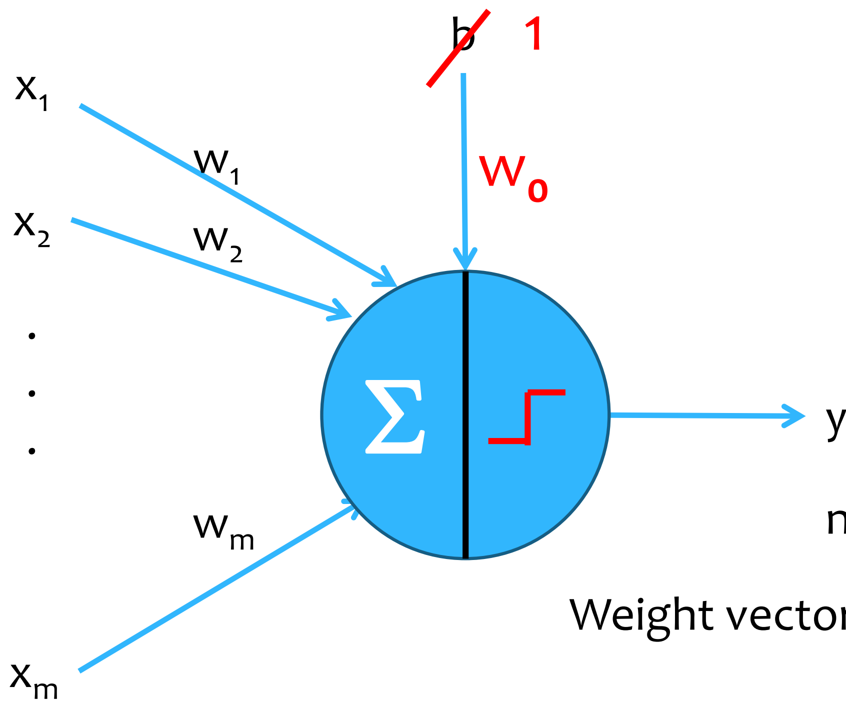




Iman Ardekani

Perceptron

Equivalent Presentation



net $= \mathbf{W}^T\mathbf{X}$

Weight vector: $\mathbf{W} = [w_0\ w_1 \dots\ w_m]^T$

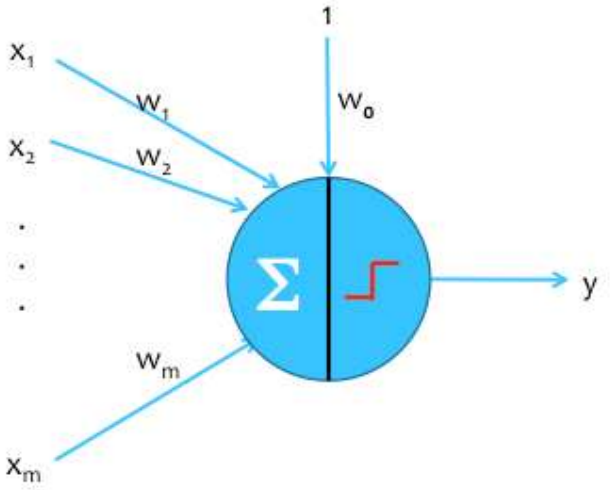
Input vector: $\mathbf{X} = [1\ x_1\ x_2 \dots\ x_m]^T$

Perceptron

There exist a weight vector **w** such that we may state

$\mathbf{W}^T\mathbf{x} > 0$ for every input vector **x** belonging to **A**

$\mathbf{W}^T\mathbf{x} \leq 0$ for every input vector **x** belonging to **B**



Perceptron

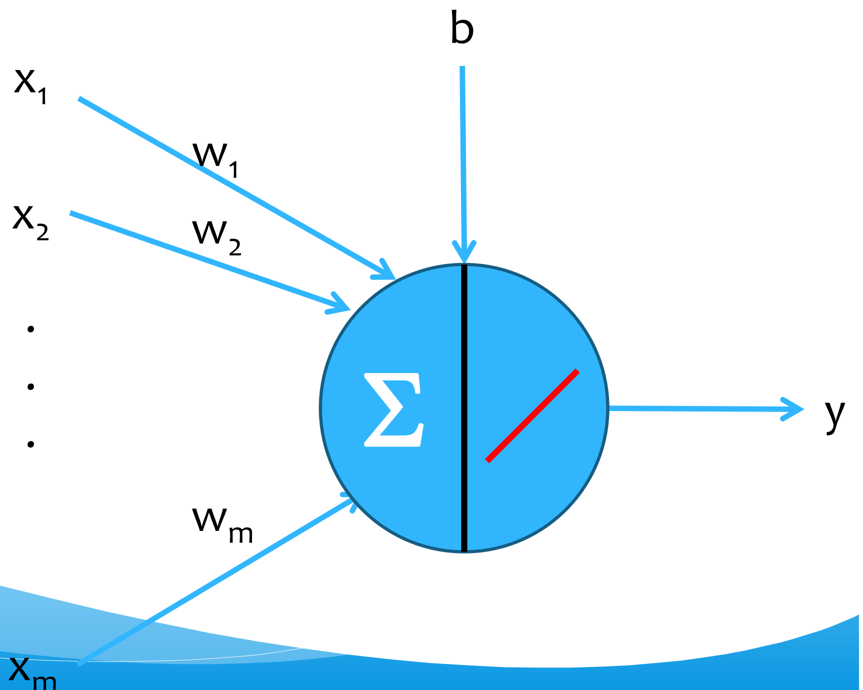
Elementary perceptron Learning Algorithm

1) Initiation: $\mathbf{w}(0)$ = a random weight vector
2) At time index n, form the input vector $\mathbf{x}(n)$
3) IF ($\mathbf{w}^T\mathbf{x} > 0$ and x belongs to A) or ($\mathbf{w}^T\mathbf{x} \leq 0$ and $\mathbf{x}$ belongs to B) THEN $\mathbf{w}(n)=\mathbf{w}(n-1)$ Otherwise $\mathbf{w}(n)=\mathbf{w}(n-1)-\eta\mathbf{x}(n)$
4) Repeat 2 until $\mathbf{w}(n)$ converges

1)

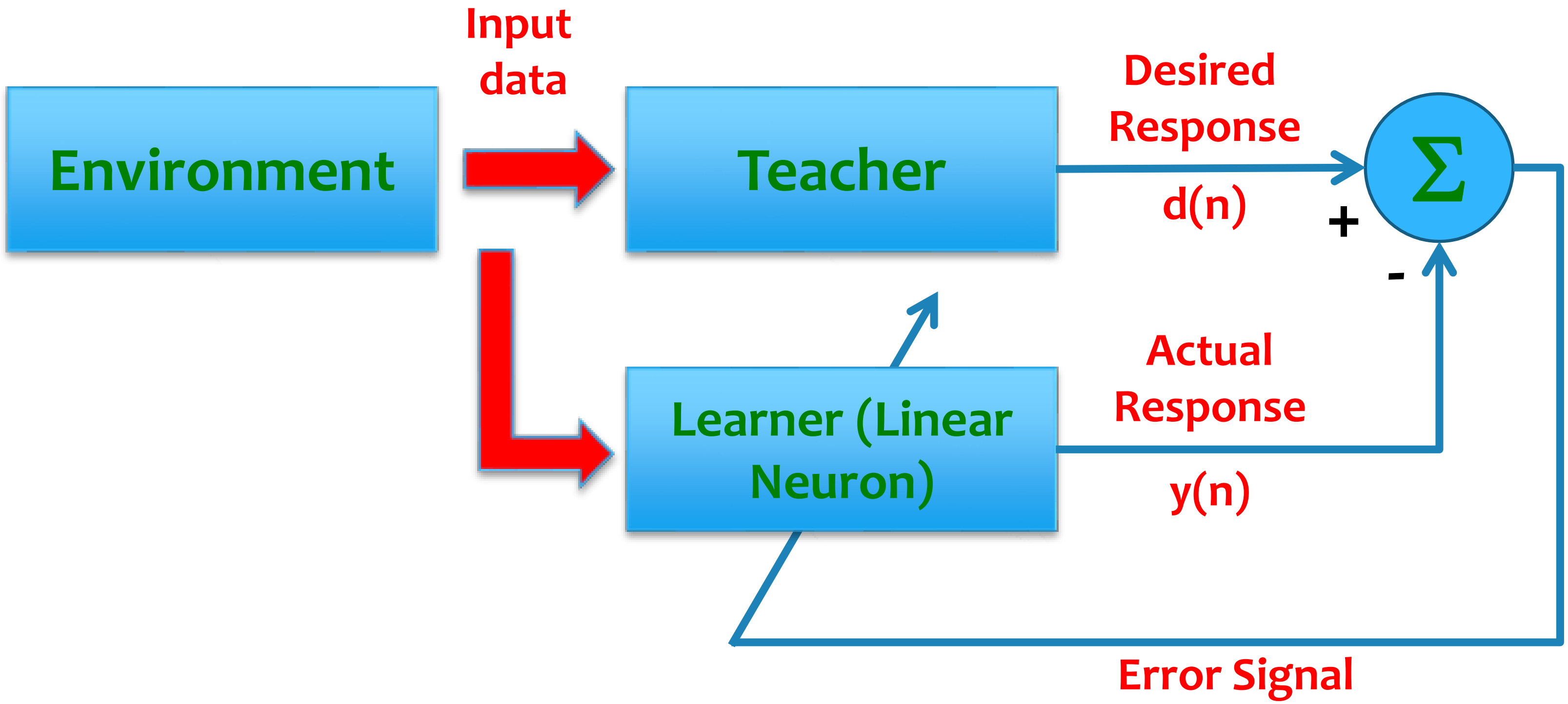
Linear Neuron

- when the activation function simply is f(x)=x the neuron acts similar to an adaptive filter.
- In this case: $y=net=\mathbf{w}^T\mathbf{x}$
- $\mathbf{w}=[w_1\ w_2\ \ldots\ w_m]^T$
- $\mathbf{x}=[x_1\ x_2\ \ldots\ x_m]^T$



Linear Neuron

Linear Neuron Learning (Adaptive Filtering)



Linear Neuron

LMS Algorithm

To minimize the value of the cost function defined as

$$E(\mathbf{w}) = 0.5\ e^2(n)$$

where e(n) is the error signal

$$e(n)=d(n)-y(n)=d(n)-\mathbf{w}^T(n)\mathbf{x}(n)$$

In this case, the weight vector can be updated as follows

$$w_i(n+1)=w_i(n-1)\ -\ \mu(\frac{dE}{dw_i})$$

Linear Neuron

LMS Algorithm (continued)

$$\frac{dE}{dw_i} = e(n)\frac{de(n)}{dw_i} = e(n)\frac{d}{dw_i}\{d(n)-\mathbf{w}^T(n)\mathbf{x}(n)\}$$

$$= -e(n)\, x_i(n)$$

$$w_i(n+1)=w_i(n)+\mu e(n)x_i(n)$$

$$\mathbf{w}(n+1)=\mathbf{w}(n)+\mu e(n)\mathbf{x}(n)$$

Linear Neuron

Summary of LMS Algorithm

1) Initiation: $\mathbf{w}(0)$ = a random weight vector
2) At time index n, form the input vector $\mathbf{x}(n)$
3) $y(n)=\mathbf{w}^T(n)\mathbf{x}(n)$
4) $e(n)=d(n)-y(n)$
5) $\mathbf{w}(n+1)=\mathbf{w}(n)+\mu e(n)\mathbf{x}(n)$
6) Repeat 2 until $\mathbf{w}(n)$ converges

Multilayer Perceptron

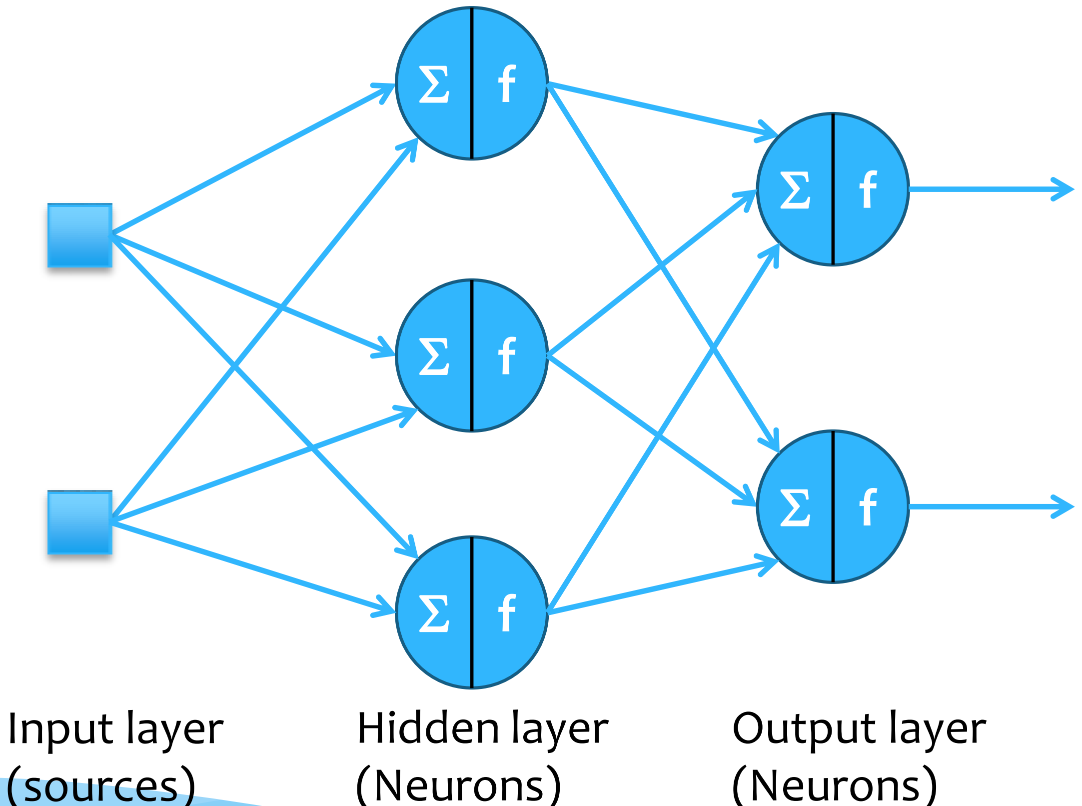
- To solve the xor problem
- To solve the nonlinear classification problem
- To deal with more complex problems

Multilayer Perceptron

- The activation function in multilayer perceptron is usually a Sigmoid function.
- Because Sigmoid is differentiable function, unlike the hard limiter function used in the elementary perceptron.

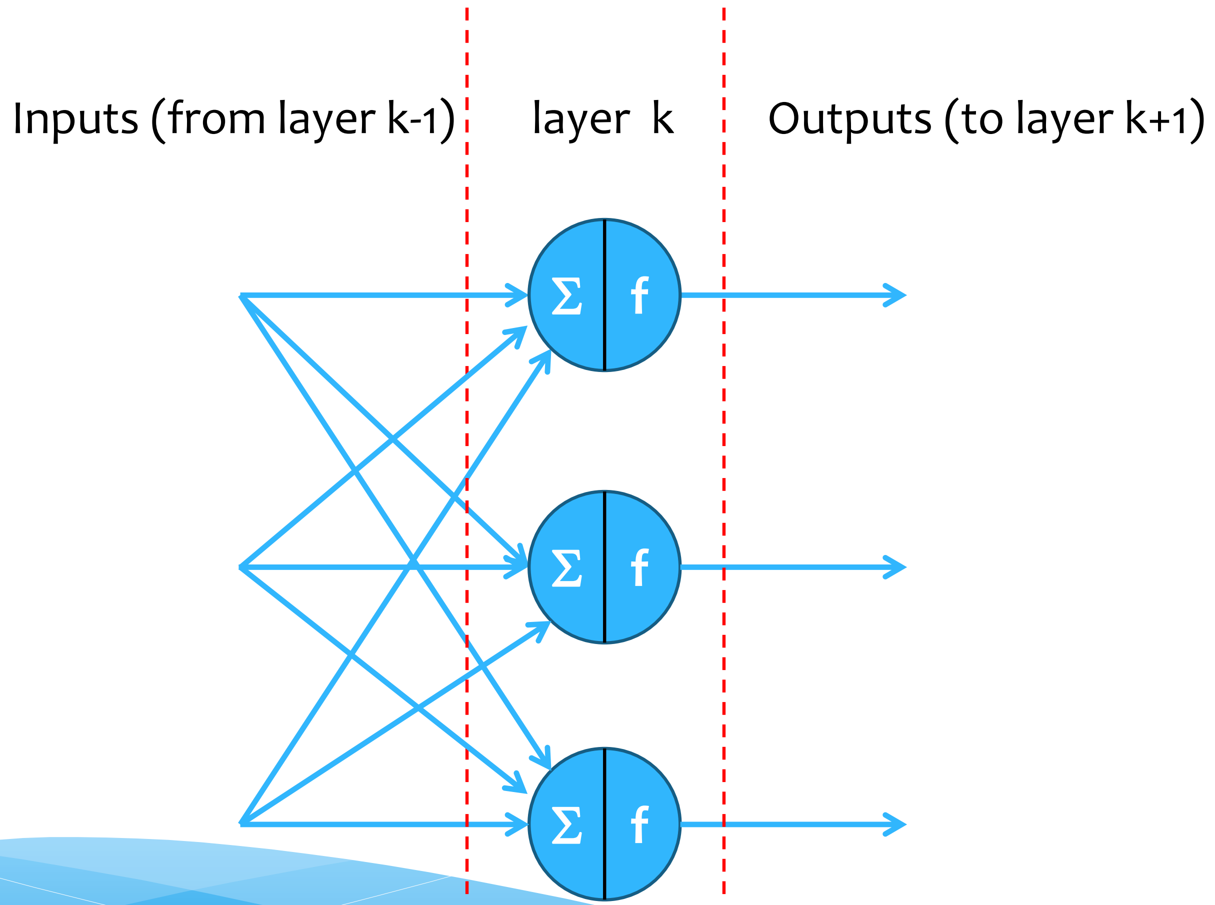
Multilayer Perceptron

Architecture



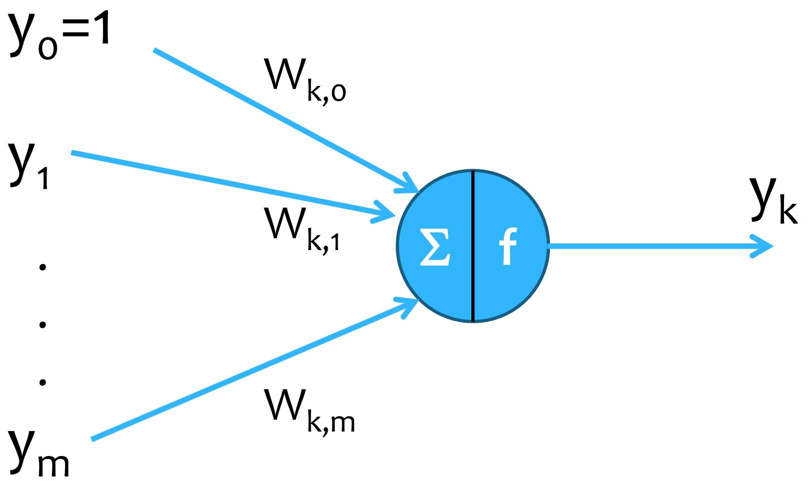
Multilayer Perceptron

Architecture



Multilayer Perceptron

Consider a single neuron in a multilayer perceptron (neuron k)

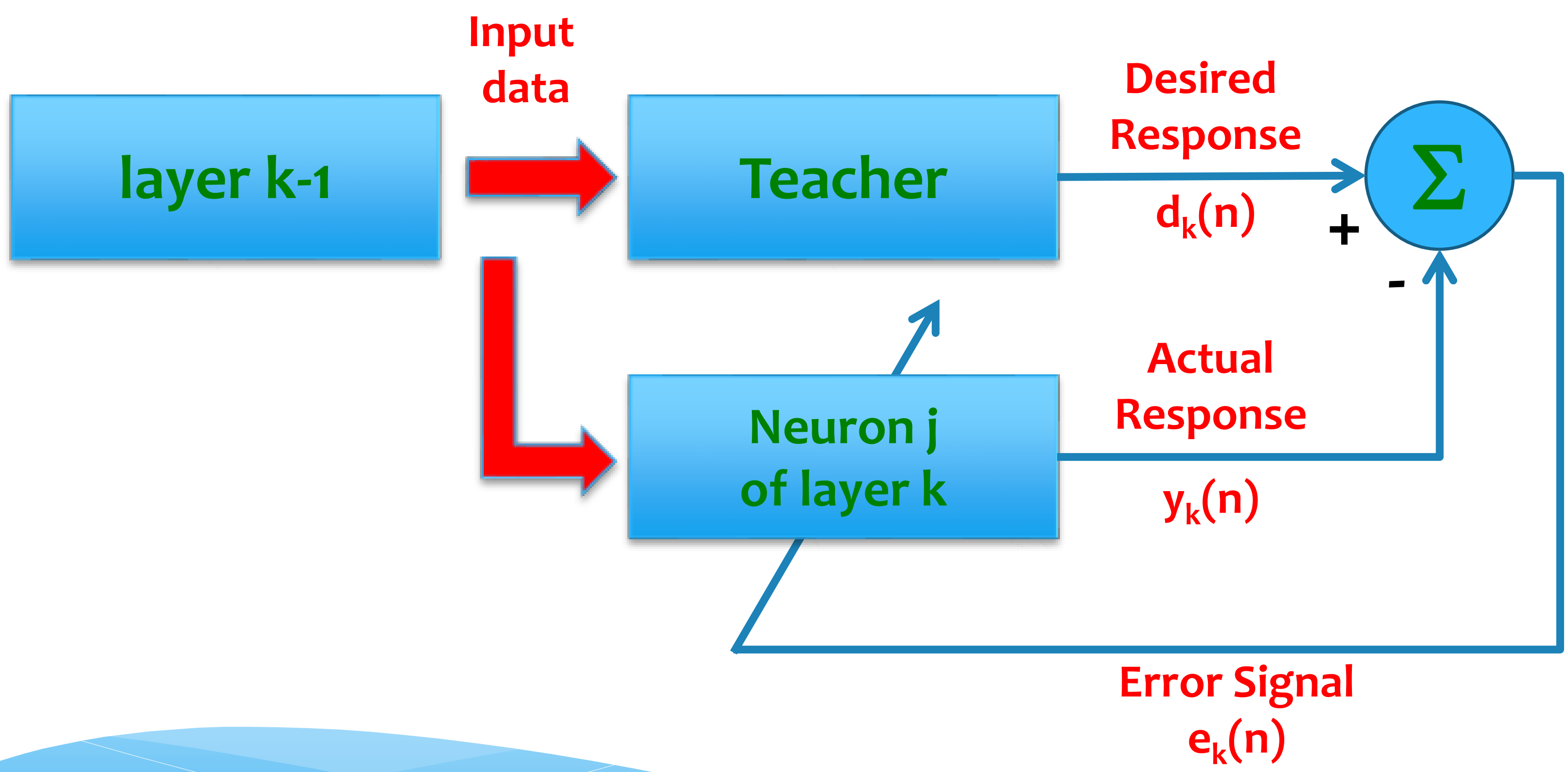


$$net_k = \sum w_{k,i} y_i$$

$$y_k = f\,(net_k)$$

Multilayer Perceptron

Multilayer perceptron Learning (Back Propagation Algorithm)



Multilayer Perceptron

Back Propagation Algorithm:

Cost function of neuron j:

$$E_k = 0.5\ e^2_j$$

Cost function of network:

$$E = \Sigma E_j = 0.5\Sigma e^2_j$$

$$e_k = d_k – y_k$$

$$e_k = d_k – f(net_k)$$

$$e_k = d_k – f(\Sigma w_{k,i} y_i)$$

Multilayer Perceptron

Back Propagation Algorithm:

Cost function of neuron k:

$$E_k = 0.5\, e^2_k$$

$$e_k = d_k - y_k$$

$$e_k = d_k - f(net_k)$$

$$e_k = d_k - f(\Sigma w_{k,i} y_i)$$

Multilayer Perceptron

Cost function of network:

$$E = \Sigma E_j = 0.5\Sigma e^2_j$$

Multilayer Perceptron

Back Propagation Algorithm:

To minimize the value of the cost function $E(w_{k,i})$, the weight vector can be updated using a gradient based algorithm as follows

$$w_{k,i}(n+1)=w_{k,i}(n) - \mu\left(\frac{dE}{dw_{k,i}}\right)$$

$$\frac{dE}{dw_{k,j}} = ?$$

Multilayer Perceptron

Back Propagation Algorithm:

$$\frac{dE}{dw_{k,i}} = \frac{dE}{de_k}\frac{de_k}{dy_k}\frac{dy_k}{dnet_k}\frac{dnet_k}{dw_{k,i}} =$$

$$\frac{dE}{dnet_k}\frac{dnet_k}{dw_{k,i}} = \delta_k\, y_k$$

$$\delta_k = \frac{dE}{de_k}\frac{de_k}{dy_k}\frac{dy_k}{dnet_k} = -e_k f'(net_k)$$

Local Gradient

$$\frac{dE}{de_k} = e_k$$

$$\frac{de_k}{dy_k} = -1$$

$$\frac{de_k}{dnet_k} = f'(net_k)$$

$$\frac{dnet_k}{dw_{k,i}} = y_i$$

Multilayer Perceptron

Back Propagation Algorithm:

Substituting $\frac{dE}{dw_{k,i}}$ into the gradient-based algorithm:

$w_{k,i}(n+1)=w_{k,i}(n) – \mu\, \delta_k\, y_k$

$\delta_k = -e_k f'(net_k) = - \{d_k - y_k\}\, f'(net_k) = ?$

If k is an output neuron we have all the terms of δ_k

When k is a hidden neuron?

Multilayer Perceptron

Back Propagation Algorithm:

When k is hidden

$$\delta_k = \frac{dE}{\cancel{de_k}} \frac{\cancel{de_k}}{dy_k} \frac{dy_k}{dnet_k} = \frac{dE}{dy_k} \frac{dy_k}{dnet_k}$$

$$\frac{de_k}{dnet_k} = f'(net_k)$$

$$= \frac{dE}{dy_k} f'(net_k)$$

$$\frac{dE}{dy_k} = ?$$

Multilayer Perceptron

$$\frac{dE}{dy_k} = ?$$

$$E = 0.5\Sigma e^2_j$$

$$\frac{dE}{dy_k} = \Sigma\, e_j\, \frac{de_j}{dy_k}$$

$$= \Sigma\, e_j\, \underbrace{\frac{de_j}{dnet_j}}_{-f'(net_j)}\, \overbrace{\frac{dnet_j}{dy_k}}^{w_{jk}}$$

$$= -\Sigma\, e_j\, f'(net_j)\, w_{jk} = -\Sigma\, \delta_j\, w_{jk}$$

Multilayer Perceptron

We had $\delta_k = -\dfrac{dE}{dy_k} f'(net_k)$

Substituting $\dfrac{dE}{dy_k} = \Sigma\, \delta_j\, w_{jk}$ into δ_k results in

$$\delta_k = -\, f'(net_k)\, \Sigma\, \delta_j\, w_{jk}$$

which gives the local gradient for the hidden neuron k

Multilayer Perceptron

Summary of Back Propagation Algorithm

1) Initiation: **w**(0) = a random weight vector

At time index n, get the input data and

2) Calculate net_k and y_k for all the neurons
3) For output neurons, calculate $e_k=d_k-y_k$
4) For output neurons, $\delta_k = -e_k f'(net_k)$
5) For hidden neurons, $\delta_k = - f'(net_k) \Sigma \delta_j w_{jk}$
6) Update every neuron weights by

$$w_{k,i}(NEW)=w_{k,i}(OLD) - \mu \delta_k y_k$$

7. Repeat steps 2~6 for the next data set (next time index)

THE END