

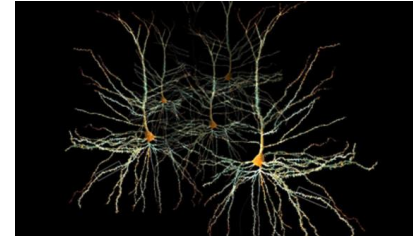
# LESSON 20

Neurons, Perceptrons  
Feedforward neural networks and  
their Learning Methods



# Outline

- Neurons
  - Natural neurons
  - Software/Artificial neurons
- The Perceptron
- Multilayer feedforward neural networks
- Other types of NNs
- Example using Matlab



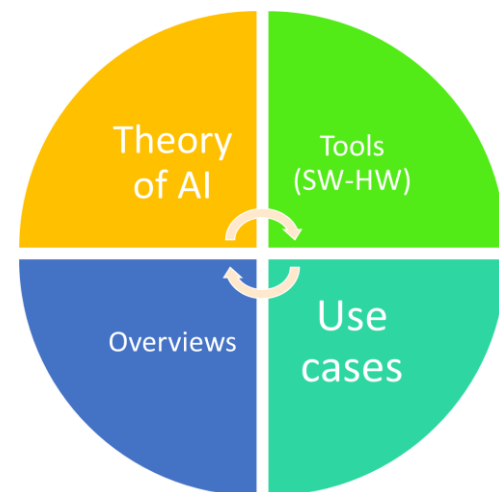
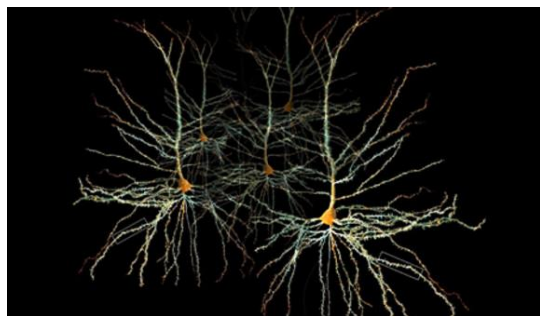


# Overview

## Introduction

## neural networks

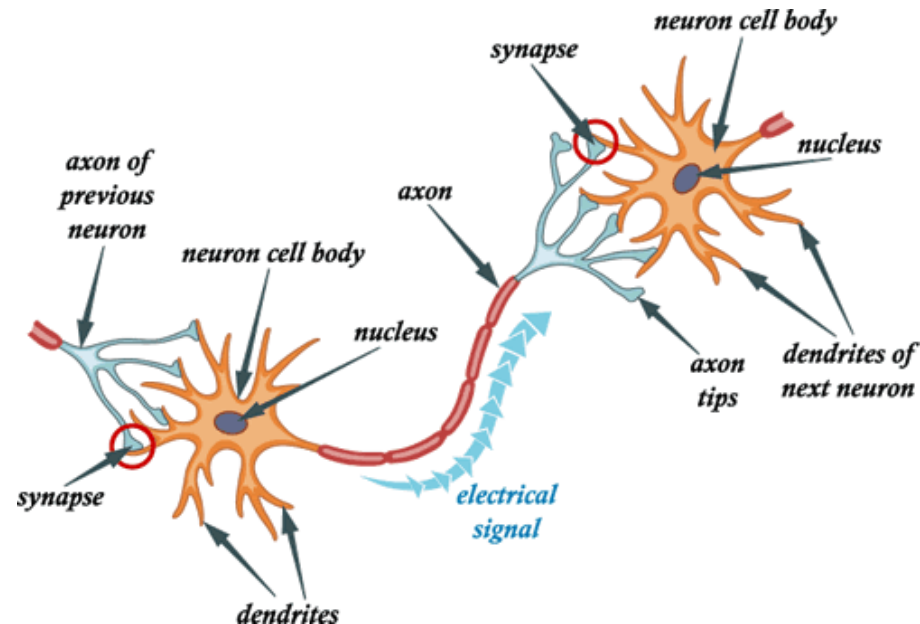
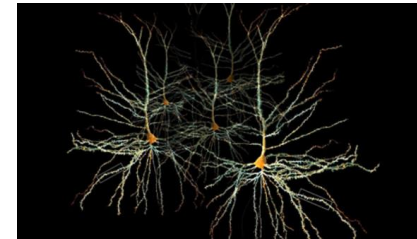
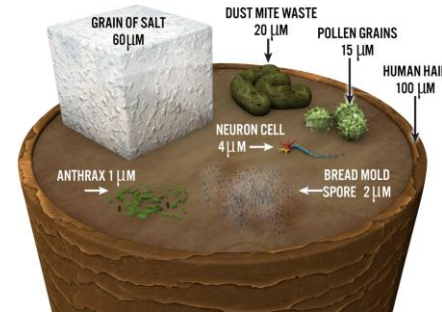
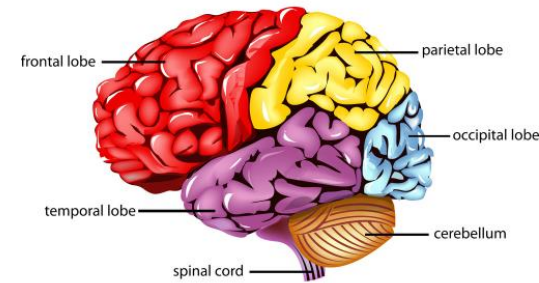
How the brain works with neural networks



# Neurons and synapses

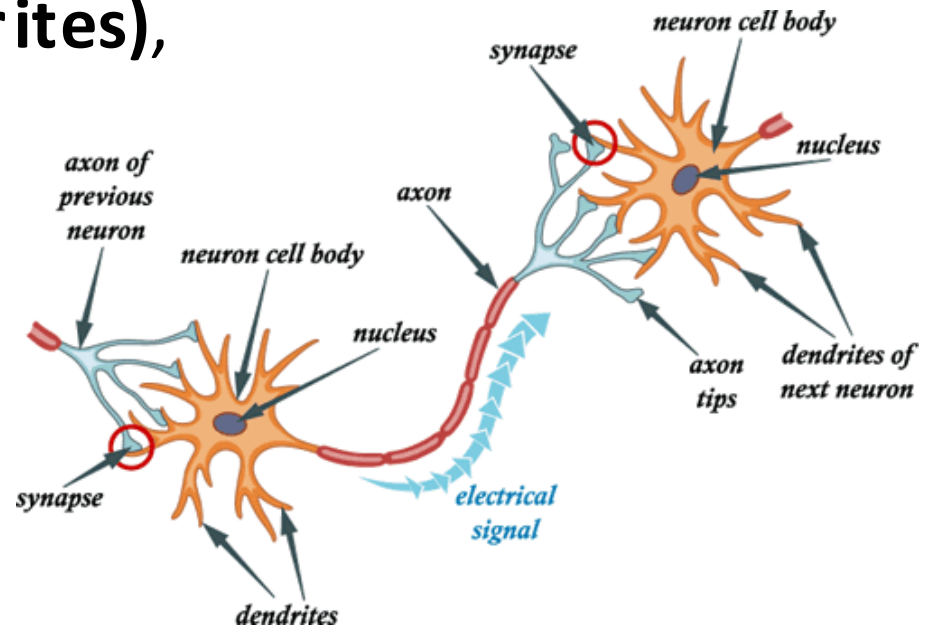
- A **neural network** can be defined as a model of reasoning based on the human brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called **neurons**.
- The human brain incorporates nearly **10 billion neurons** and **60 trillion connections (synapses)** between them. By using multiple neurons simultaneously, the brain can perform its functions much faster than the fastest computers.

Parts of the Human Brain



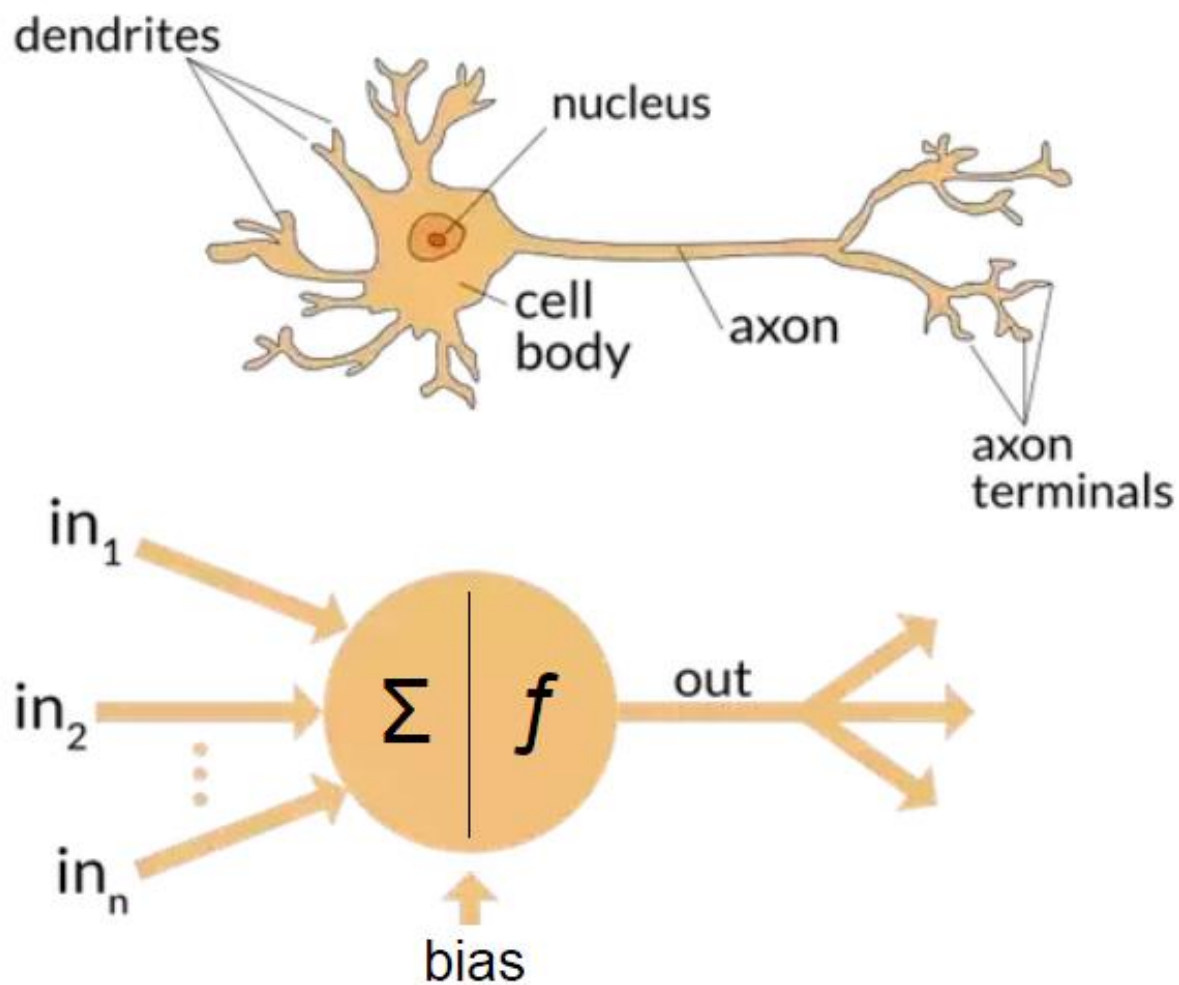
# Connections

- Each neuron has a very simple structure, but the ensemble of such elements constitutes a tremendous processing power.
- A neuron consists of a cell body (**soma**) a number of fibers (**dendrites**), and a single long fiber (**axon**).



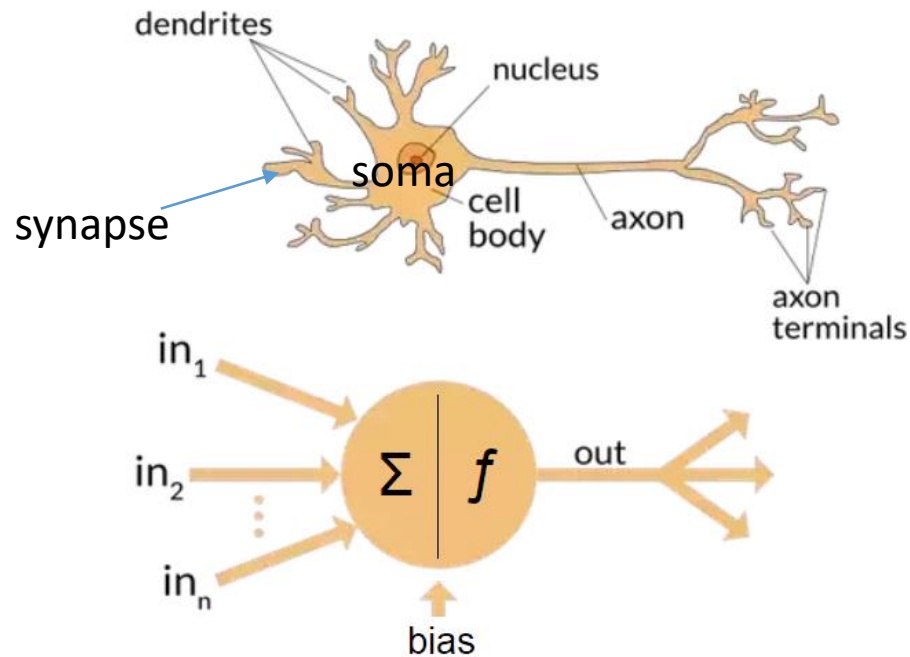


# Similitude

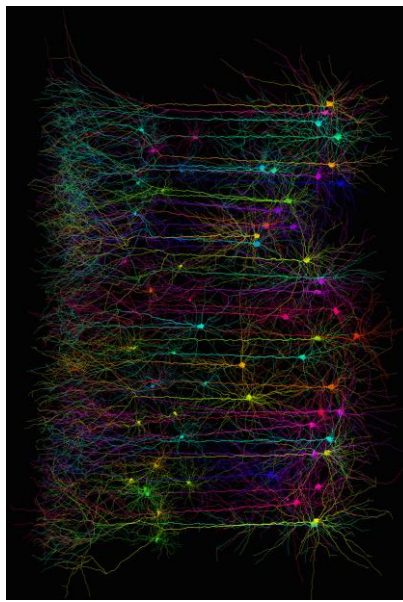


# Analogy between biological and artificial neural networks

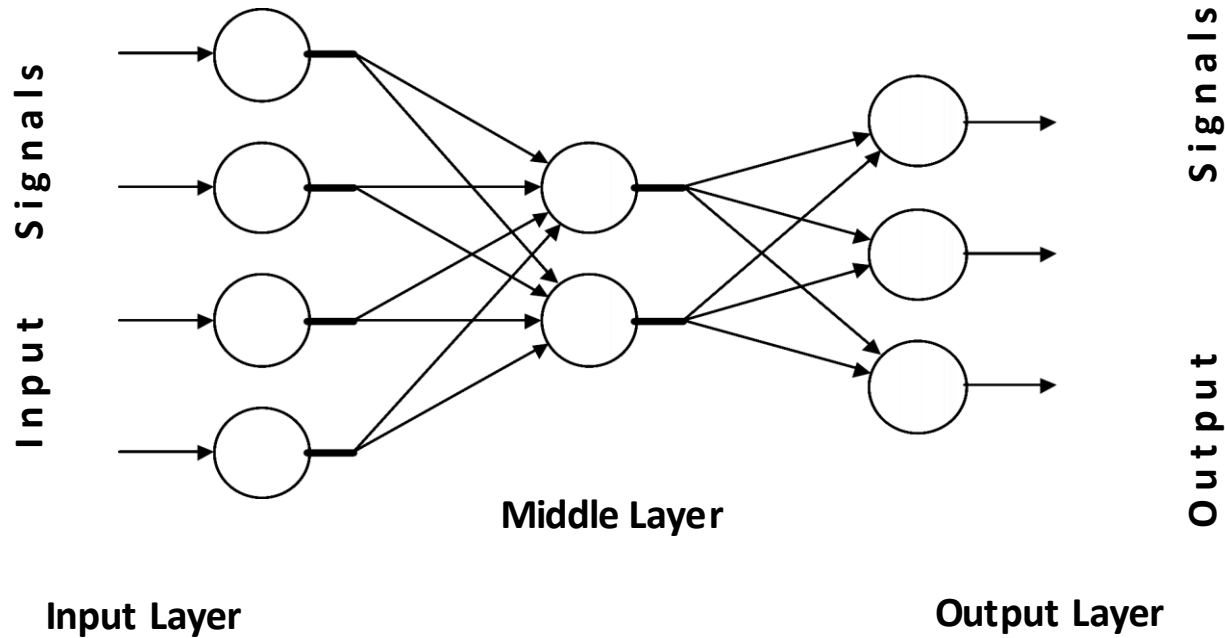
Biological Neural Network	Artificial Neural Network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight



Neural  
network



Artificial  
neural  
network





# Characteristics of artificial neural networks

- **Learning from experience**

Complex difficult to solve problems, but with plenty of data that describe the problem

- **Generalizing from examples**

Can interpolate from previous learning and give the correct response to unseen data

- **Rapid applications development**

NNs are generic machines and quite independent from domain knowledge

# Characteristics of artificial neural networks

- **Adaptability**

Adapts to a changing environment, if is properly designed

- **Computational efficiency**

Although the training off a neural network demands a lot of computer power, a trained network demands almost nothing in recall mode

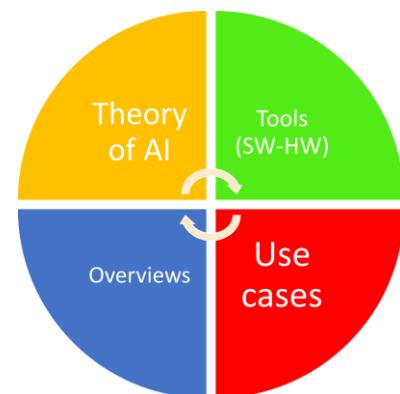
- **Non-linearity**

Not based on linear assumptions about the real word  
(→ advanced input processing)



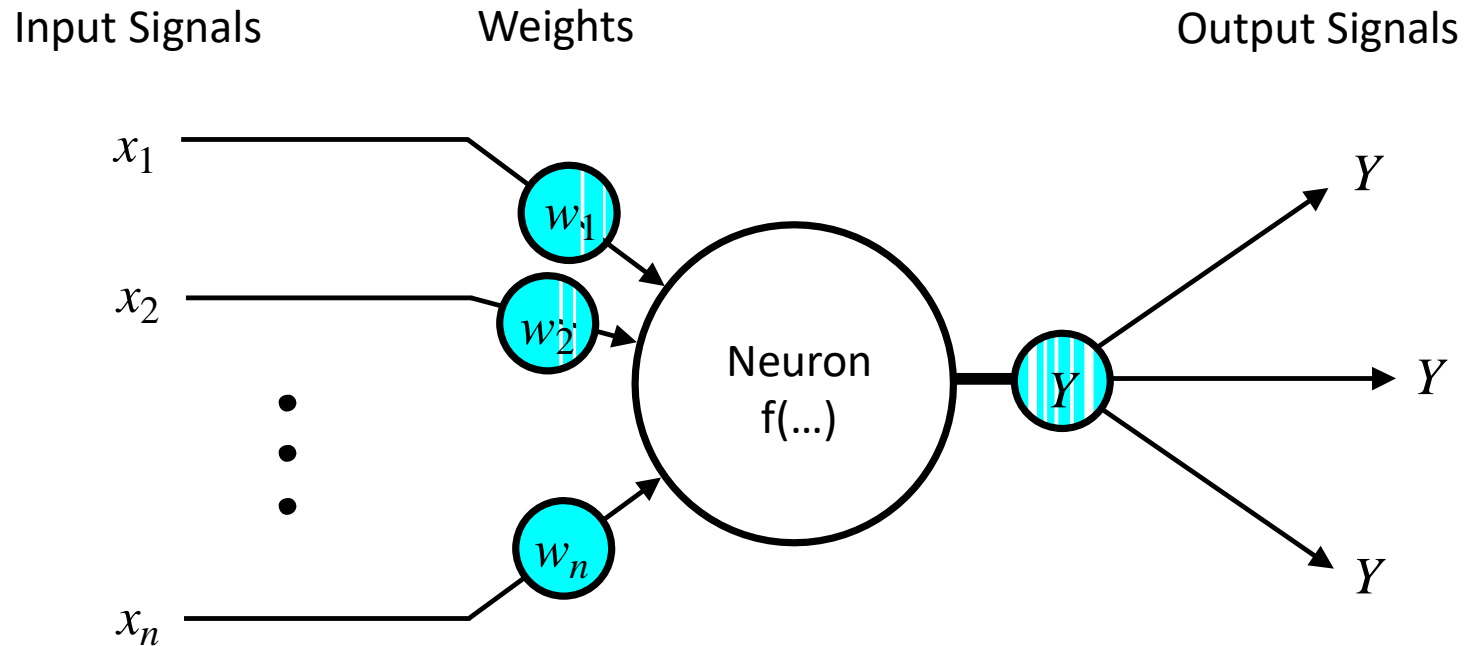
# THEORY

## The neuron as a simple computing element



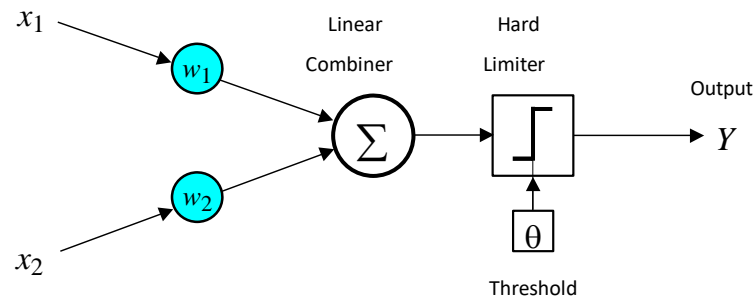
# The neuron as a simple computing element

## Diagram of a neuron



# Basic neuron model

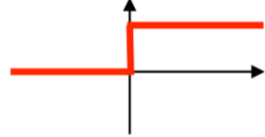
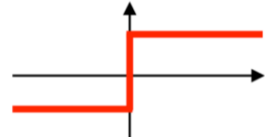
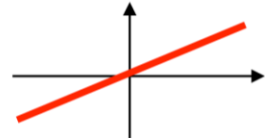


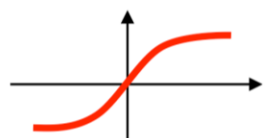
- The neuron computes the weighted sum of the input signals and compares the result with a **threshold value**,  $\vartheta$ . If the net input is less than the threshold, the neuron output is  $-1$ . But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value  $+1$ .
- The neuron uses the following transfer or **activation function**:



$$X = \sum_{i=1}^n x_i w_i$$
$$Y = \begin{cases} +1, & \text{if } X \geq \theta \\ -1, & \text{if } X < \theta \end{cases}$$

This type of activation function is called a **sign function**.

# Activation functions

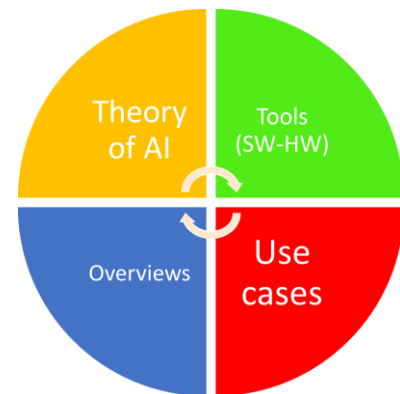
Activation function	Equation	1D Graph	
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$		Non linear Not «well» differentiable
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$		
Linear	$\phi(z) = z$		Non clipping
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$		Limited input range
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$		
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$		Manage «spikes»





# THEORY

## Perceptron

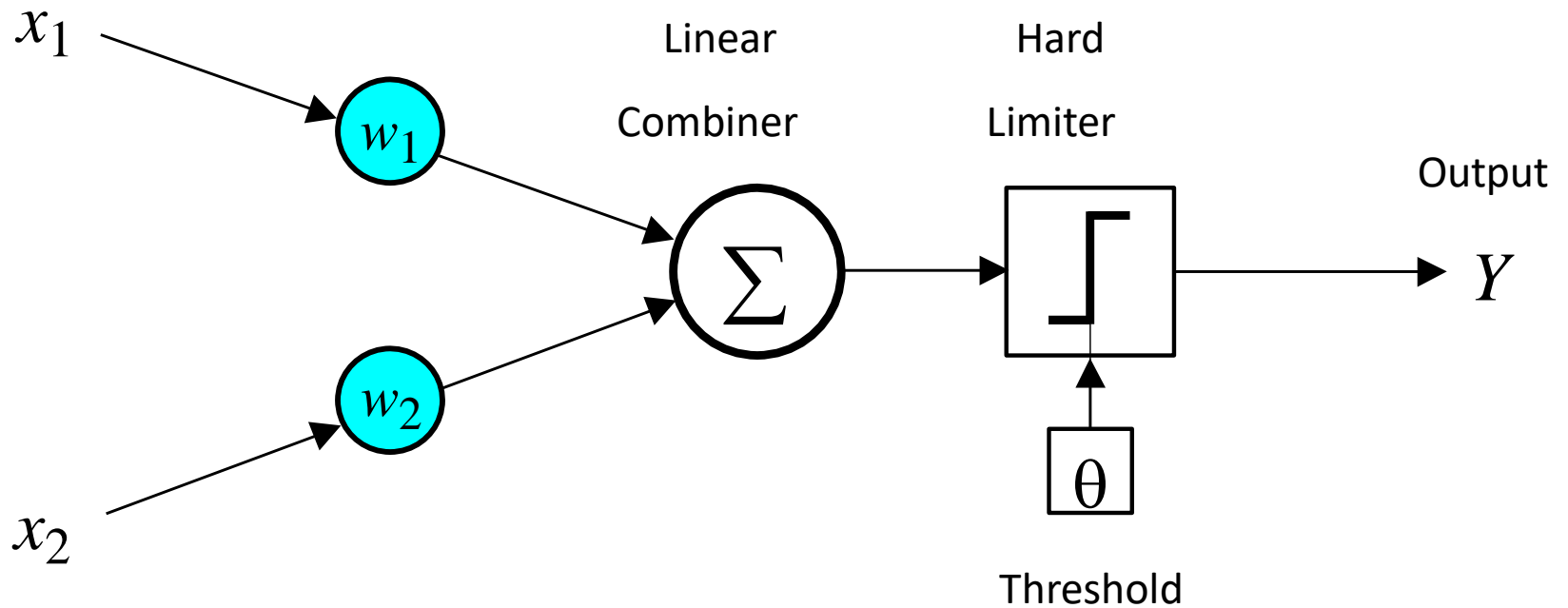


# Can a single neuron learn a task?

- In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron**.
- The perceptron is the simplest form of a neural network. It consists of a single neuron with **adjustable** synaptic weights and a **hard limiter**.

# Single-layer two-input perceptron

Inputs



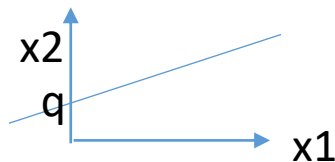
# The perceptron goal → Binary classification

- The aim of the perceptron is to classify inputs,  $x_1, x_2, \dots, x_n$ , into one of two classes, say  $A_1$  and  $A_2$ .
- In the case of an elementary perceptron, the  $n$ - dimensional space is divided by a **hyperplane** into two decision regions. The hyperplane is defined by the **linearly separable function**:

Example

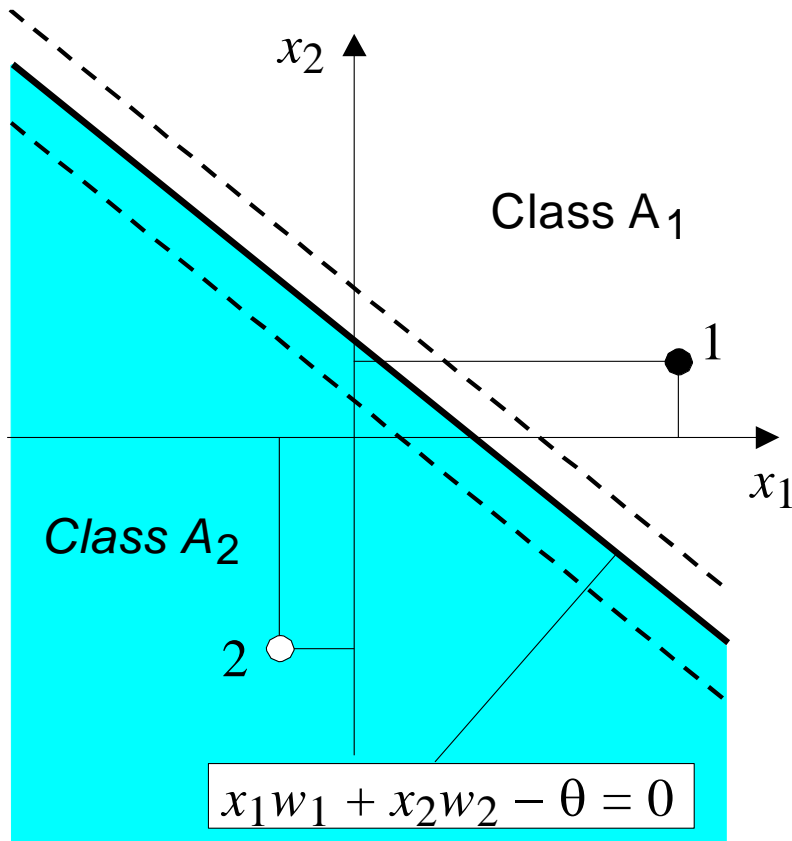
$$x_2 = m x_1 + q$$

$$x_2 - m x_1 - q = 0$$

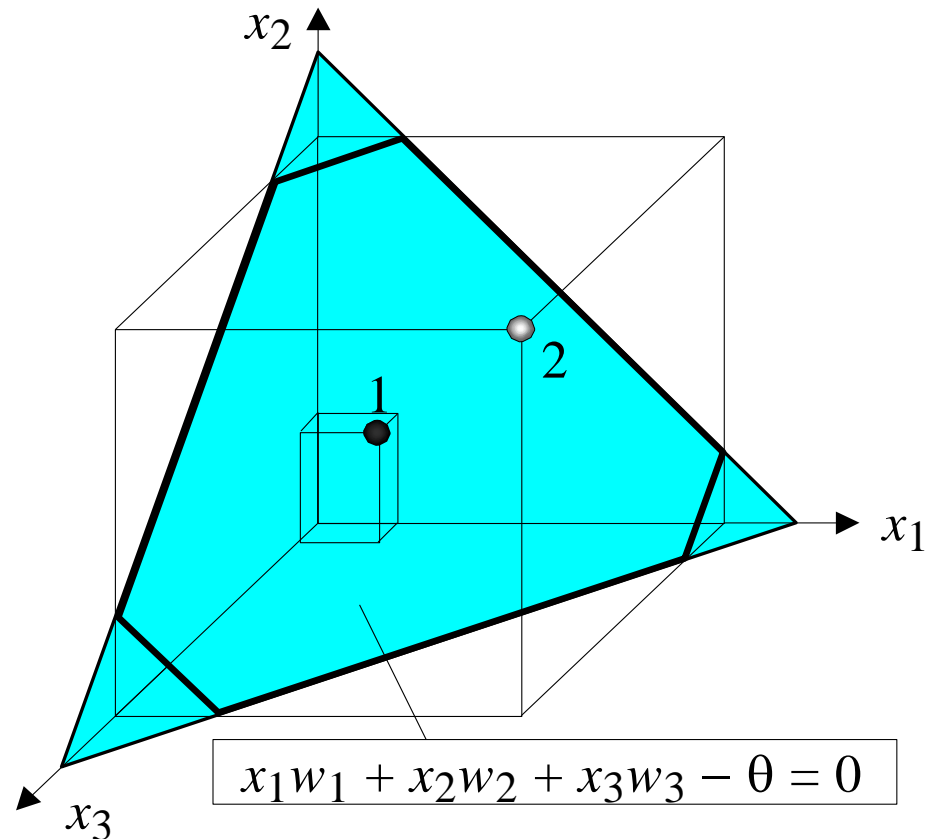


$$\sum_{i=1}^n x_i w_i - \theta = 0$$

# Linear separability in the perceptrons



(a) Two-input perceptron.



(b) Three-input perceptron.

# How does the perceptron learn its classification tasks? (1/2)

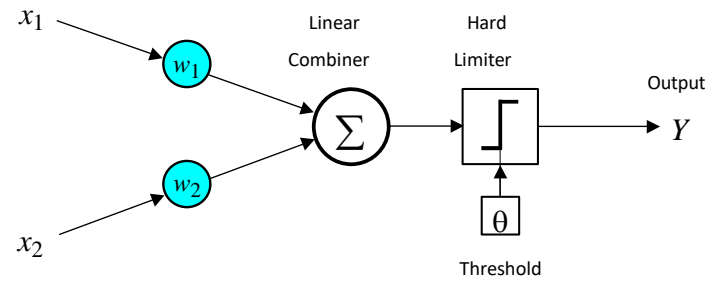
- **LEARNING MEANS.....**

making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron.

- The initial weights are randomly assigned, usually in the range  $[-0.5, 0.5]$ , and then updated to obtain the output consistent with the training examples.



# How does the perceptron learn its classification tasks? (2/2)



- If at **iteration**  $p$ , the actual output is  $Y(p)$  and the desired output is  $Y_d(p)$ , then the error is given by:

$$e(p) = Y_d(p) - Y(p)$$

where  $p = 1, 2, 3, \dots$

This not an epoch

One **epoch** is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE

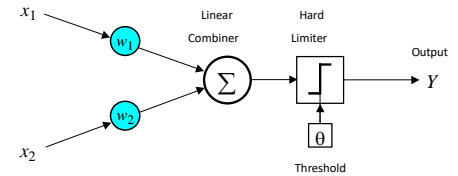


Iteration  $p$  here refers to the

**$p^{\text{th}}$  training example presented to the perceptron.**

- If the error,  $e(p)$ , is positive, we need to increase the perceptron output  $Y(p)$ , but if it is negative, we need to decrease  $Y(p)$ .

# The perceptron learning rule



“...if  $e(p) = Y_H(p) - Y(p)$ , is positive, we need to increase perceptron output  $Y(p)$ ”

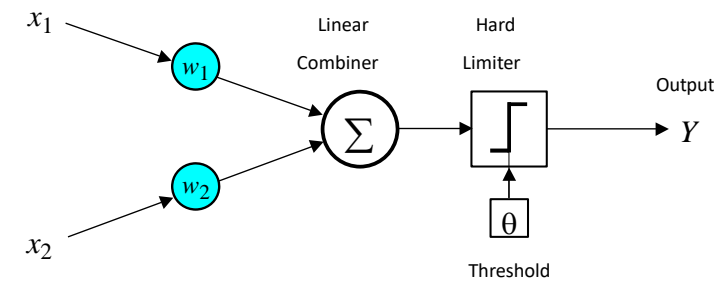
- $$\overset{\text{Next iteration}}{w_i(p+1)} = \overset{\text{Current}}{w_i(p)} + \alpha \cdot x_i(p) \cdot e(p)$$

where  $p = 1, 2, 3, \dots$

$\alpha$  is the **learning rate**, a positive constant less than unity.

- The perceptron learning rule was first proposed by Rosenblatt in 1960. Using this rule, we can derive the perceptron training algorithm for classification tasks.

# The 4 steps of the Perceptron's training algorithm



## Step 1: Initialisation

Set initial weights  $w_1, w_2, \dots, w_n$  and threshold  $\vartheta$  to random numbers in the range  $[-0.5, 0.5]$ .

## Step 2: Activation p=0 → first iteration

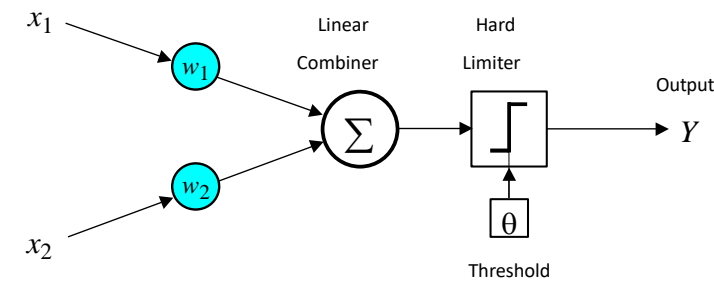
Activate the perceptron by applying inputs  $x_1(p), x_2(p), \dots, x_n(p)$  and desired output  $Y_d(p)$ .  
Calculate the actual output at iteration  $p = 1$

$$Y(p) = \text{step} \left\{ \sum_{i=0} [x_i(p) \cdot w_i(p)] - \theta \right\}$$

General Activation equation for the p-th iteration

where  $n$  is the number of the perceptron inputs, and *step* is a step activation function.

# Perceptron's training algorithm



## Step 3: Weight training

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p)$$

where  $\Delta w_i(p)$  is the weight correction at iteration  $p$ .

The weight correction is computed by the delta rule:

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Very important!

## Step 4: Iteration

Increase iteration  $p$  by one, go back to Step 2 and repeat the process until convergence.

This passage is  
the basis  
of the learning  
methods in NN!

# Example of perceptron learning: the logical operation AND

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Threshold:  $\theta = 0.2$ ; learning rate:  $\alpha = 0.1$

Epoch	Inputs		Desired output $Y_d$	Initial weights		Actual output $Y$	Error $e$	Final weights	
	$x_1$	$x_2$		$w_1$	$w_2$			$w_1$	$w_2$
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0

# Example of perceptron learning: the logical operation AND

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Epoch	Inputs		Desired output $Y_d$	Initial weights		Actual output $Y$	Error $e$	Final weights	
	$x_1$	$x_2$		$w_1$	$w_2$			$w_1$	$w_2$
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0

Threshold:  $\theta = 0.2$ ; learning rate:  $\alpha = 0.1$

$\Delta w_1 = \alpha \cdot x_1 \cdot e = 0.1 \cdot 1 \cdot (-1) = -0.1 \rightarrow \text{Final } w_1 = 0.3 - 0.1 = 0.2$



# Example of perceptron learning: the logical operation AND

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Epoch	Inputs		Desired output $Y_d$	Initial weights		Actual output $Y$	Error $e$	Final weights	
	$x_1$	$x_2$		$w_1$	$w_2$			$w_1$	$w_2$
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0

Threshold:  $\theta = 0.2$ ; learning rate:  $\alpha = 0.1$

$\Delta w_2 = \alpha x_2 e = 0.1 * 0 * (-1) = 0 \rightarrow$  Final  $w_2 = -0.1 - 0 = -0.1$  (no changes)

The input  $x_2 = 0 \rightarrow$  no delta

# Example of perceptron learning: the logical operation AND

Epoch	Inputs		Desired output $Y_d$	Initial weights		Actual output $Y$	Error $e$	Final weights	
	$x_1$	$x_2$		$w_1$	$w_2$			$w_1$	$w_2$
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	-1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	-1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	-1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

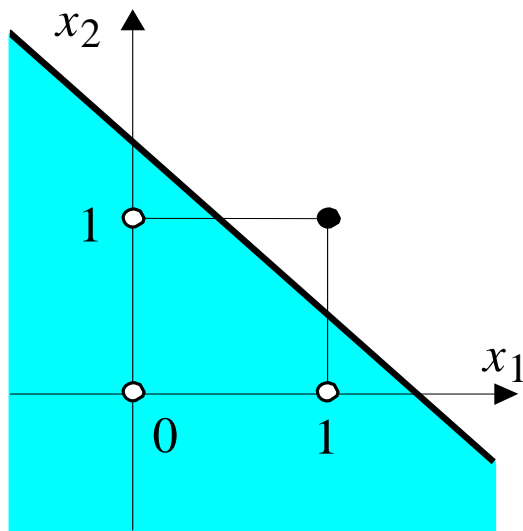
$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

error = 0

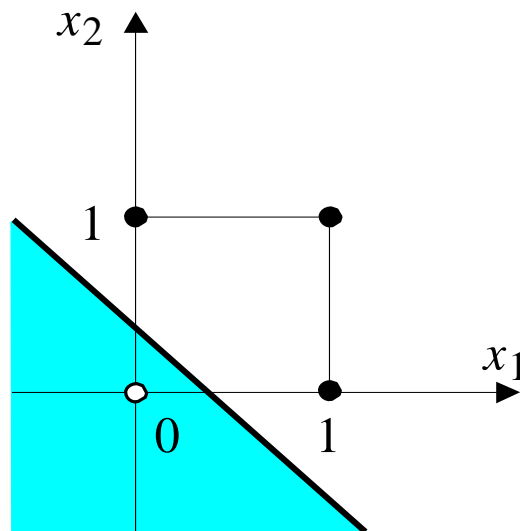
No more updates!

Threshold:  $\theta = 0.2$ ; learning rate:  $\alpha = 0.1$

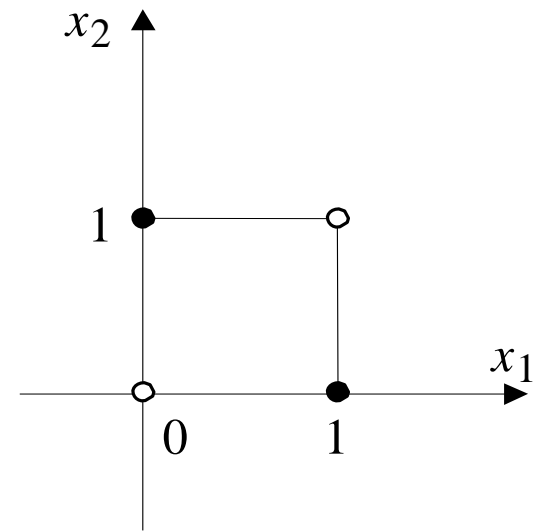
# Two-dimensional plots of basic logical operations



(a) *AND* ( $x_1 \cap x_2$ )



(b) *OR* ( $x_1 \cup x_2$ )



(c) *Exclusive-OR*  
( $x_1 \oplus x_2$ )

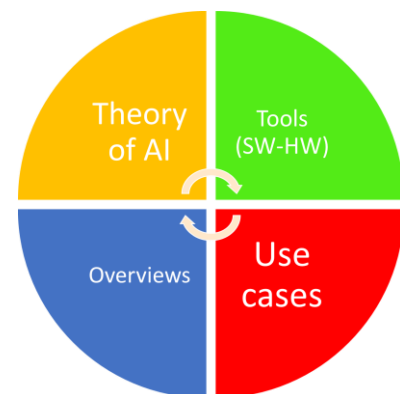
A perceptron can learn the operations *AND* and *OR*, but not *Exclusive-OR* → *Because the decision boundaries are not suitable for the datapoints in the training*



# THEORY

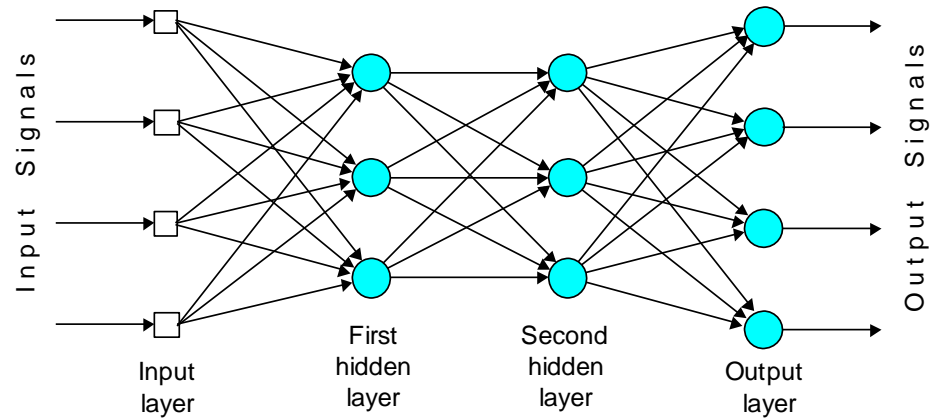
Multilayer neural networks

Back-propagation method

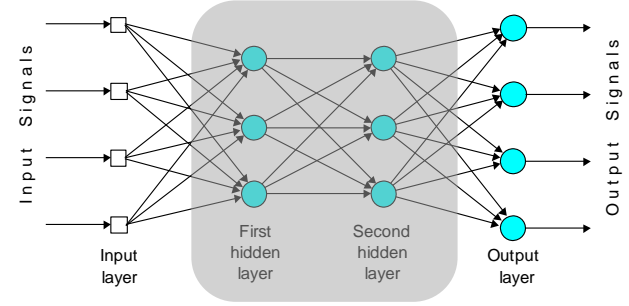


# Multilayer neural networks

- A multilayer perceptron is a feedforward neural network with one or more hidden layers.
- The network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons.
- The input signals are propagated in a forward direction on a layer-by-layer basis.



# What does the middle layer hide?



- A hidden layer “hides” its desired output.
  - Neurons in the hidden layer cannot be observed through the input/output behavior of the network.
  - There is no obvious way to know what the desired output of the hidden layer should be.
- **Commercial** ANNs typically incorporate 3-4 layers with 1-2 hidden layers. Each layer can contain from **10 to 1000 neurons**.
- **Experimental** neural networks may have 5+ layers, with 3+ hidden layers, and utilize **millions+ of neurons**.

## WARNING:

Convolutional Neural Networks (CNNs) have a different numbers of layers (>>) but the layers are quite different in their functioning!



# Learning multilayer network neural network

Learning in a multilayer network proceeds the same way as for a perceptron

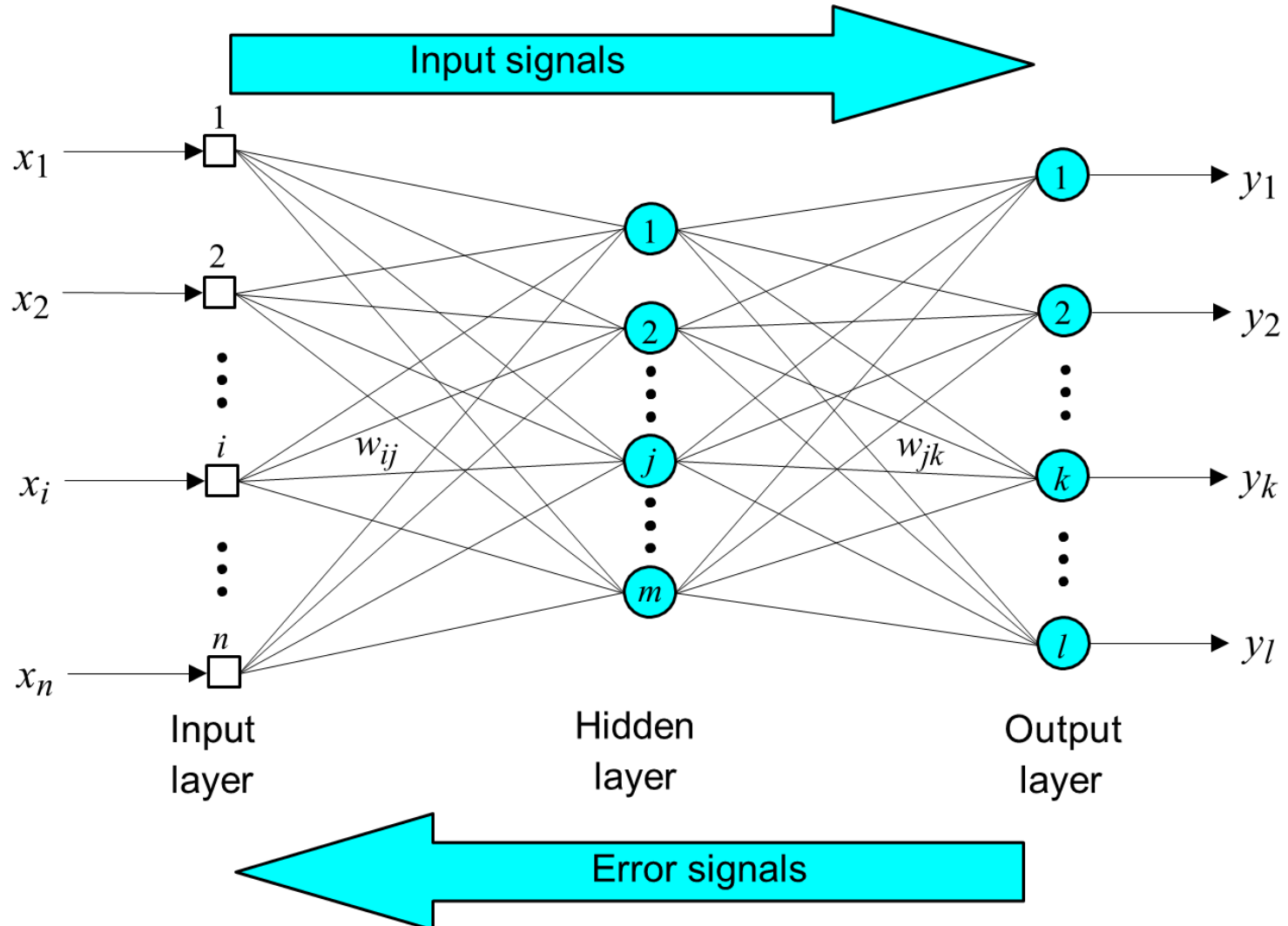
- A training set of input patterns is presented to the network
- The network computes its output pattern, and if there is an error - or in other words a difference between actual and desired output patterns - the weights are adjusted to reduce this error

# The back-propagation algorithm for neural network

The **back-propagation** neural network learning algorithm has two phases.

1. First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
2. If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

# Three-layer back-propagation neural network

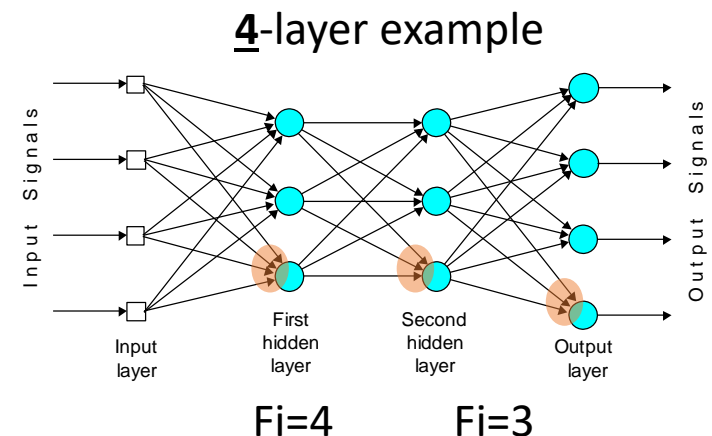


# The 4 steps of the back-propagation training algorithm

## Step 1: Initialisation (neuron-by-neuron basis)

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

$$\left( -\frac{2.4}{F_i}, +\frac{2.4}{F_i} \right)$$



where  $F_i$  is the total number of inputs of neuron  $i$  in the network.

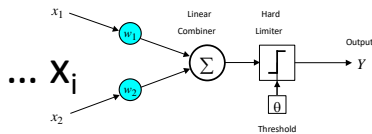
The **weight initialisation** is done on a **neuron-by-neuron basis**.

# The back-propagation training algorithm (2)

## Step 2: Activation

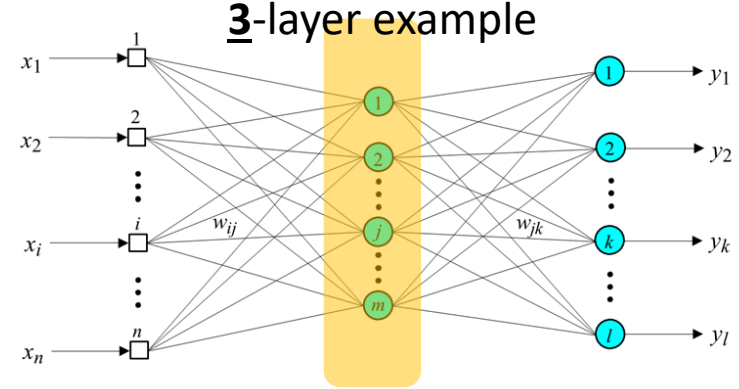
Activate the back-propagation neural network by applying inputs  $x_1(p), x_2(p), \dots, x_n(p)$  and desired outputs  $y_{d,1}(p), y_{d,2}(p), \dots, y_{d,l}(p)$ .

(a) Calculate the actual outputs of the neurons in the hidden layer:

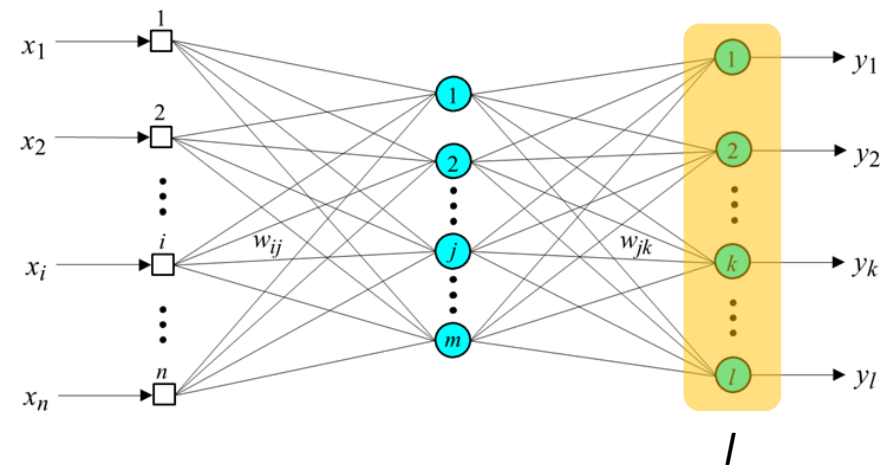


$$Y_j(\rho) = \text{sigmoid} \left\{ \sum_{i=1}^n [x_i(\rho) \cdot w_{i\mathbf{j}}(\rho)] - \theta_j \right\}$$

where  $n$  is the number of inputs of neuron  $j$  in the hidden layer, and sigmoid is the sigmoid activation function.



# The back-propagation training algorithm (3)



## Step 2: Activation (continued)

(b) Calculate the actual outputs of the neurons in the output layer:

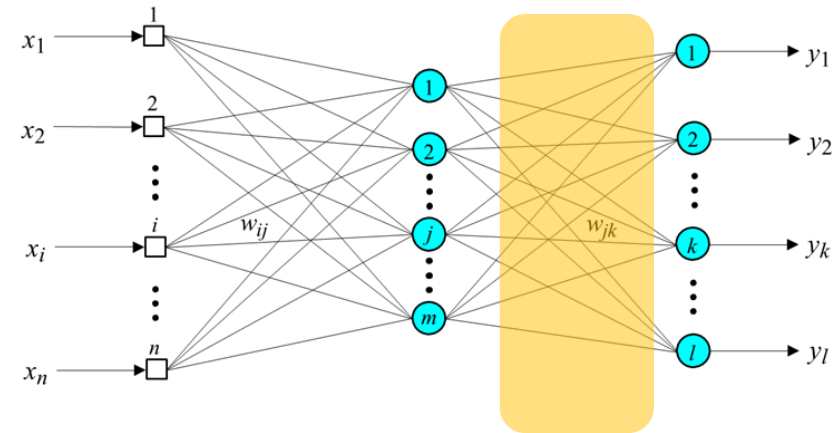
$$Y_k(\rho) = \text{sigmoid} \left\{ \sum_{j=1}^l [x_{jk}(\rho) \cdot w_{jk}(\rho)] - \theta_k \right\}$$

where  $m$  is the number of inputs of neuron  $k$  in the output layer.  $l$  (small L) is the number of outputs.

# The back-propagation training algorithm (4)

## Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.



Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

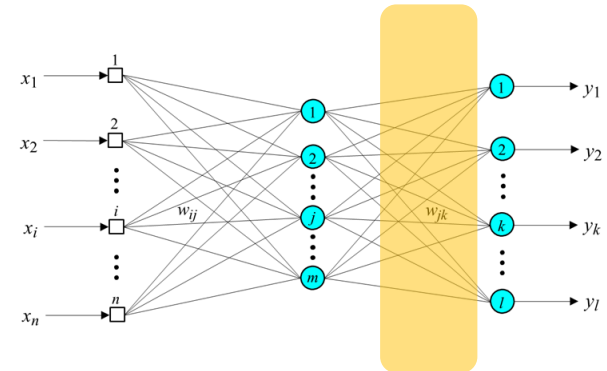
# The back-propagation training algorithm (4)

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Perceptron

## Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.



Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

Same eq.!  
but with  $\delta$



# The back-propagation training algorithm (4)

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

Perceptron

## Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the **error gradient** for the neurons in the output layer:

Not in  
the exam

$$\delta_k(p) = y_k(p) \cdot [1 - y_k(p)] \cdot e_k(p)$$

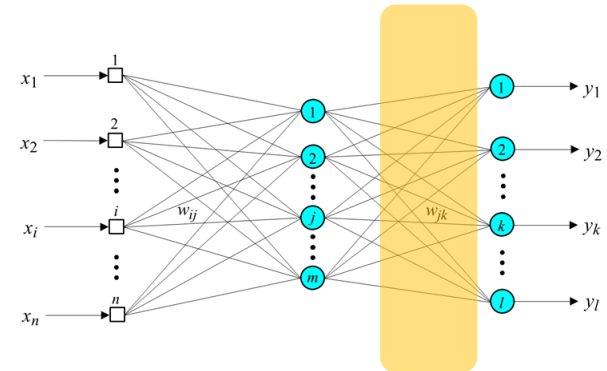
where  $e_k(p) = y_{d,k}(p) - y_k(p)$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)$$

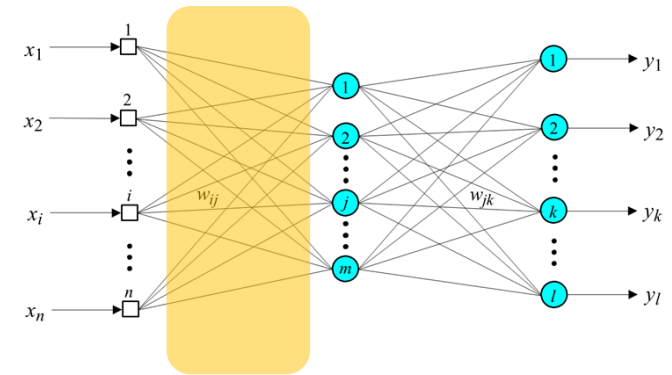
Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$



Similar eq.!  
y and  $\delta$

# The back-propagation training algorithm (5)



## Step 3: Weight training (continued)

(b) Calculate the error gradient for the neurons in the hidden layer:

Not in the exam →

$$\delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum_{k=1}^l \delta_k(p) w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \cdot x_i(p) \cdot \delta_j(p)$$



Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

# The back-propagation training algorithm (6/6)

## Step 4: Iteration

Increase iteration  $p$  by one, go back to *Step 2* and repeat the process until the selected error criterion is satisfied

- Reached the maximum number of epoch
- Errors is smaller than a fixed threshold
- No more gradient  $\rightarrow$  no improvement in the weights
- ...

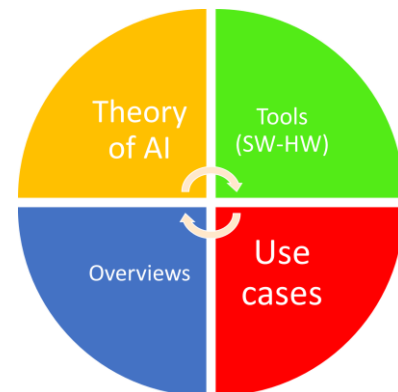


# THEORY

## Example of Back-propagation

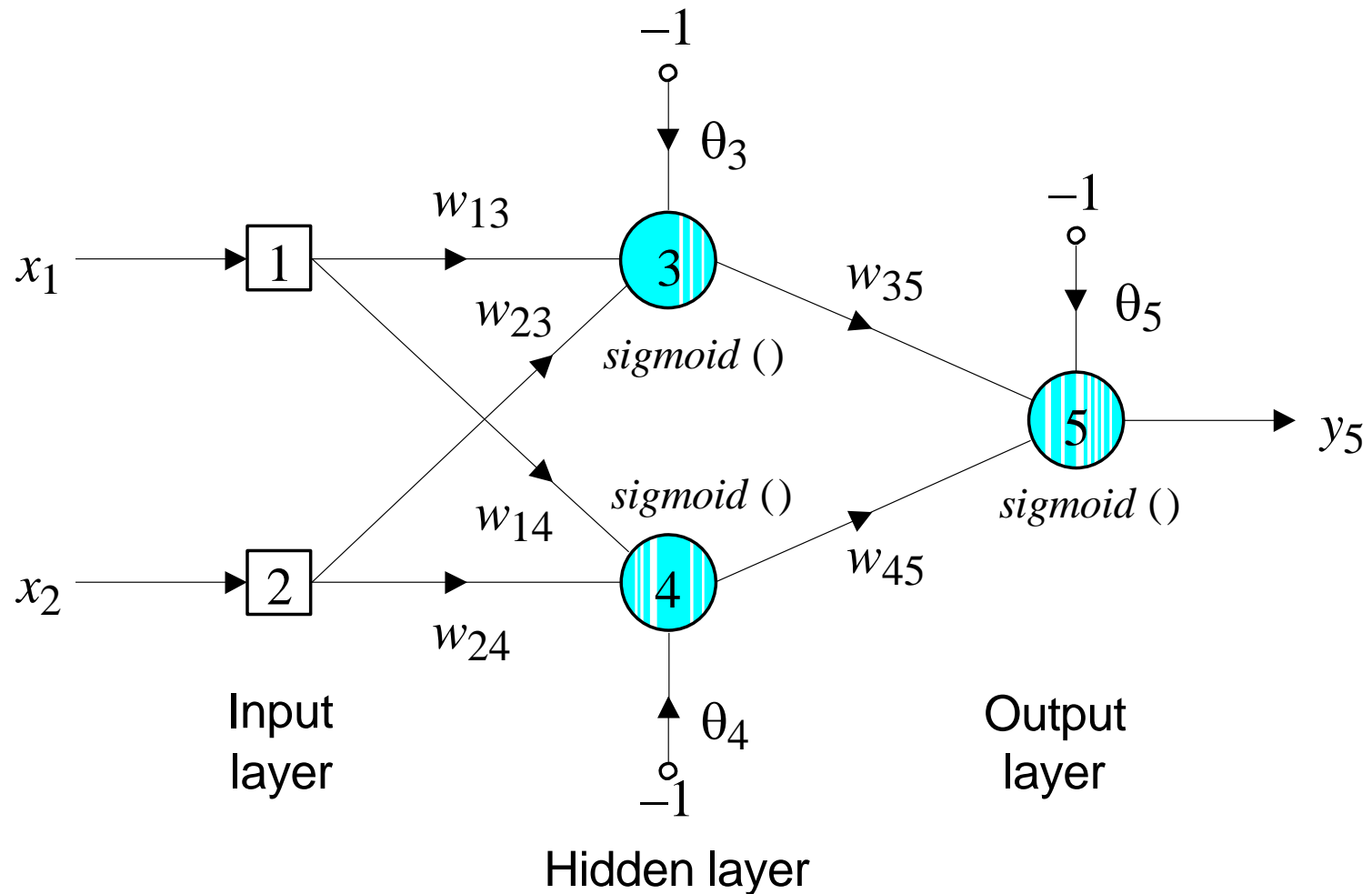
Not in the exam, but interesting  
to check the comprehension of the concepts.

We are going to process the back propagation  
with the “pencil”.



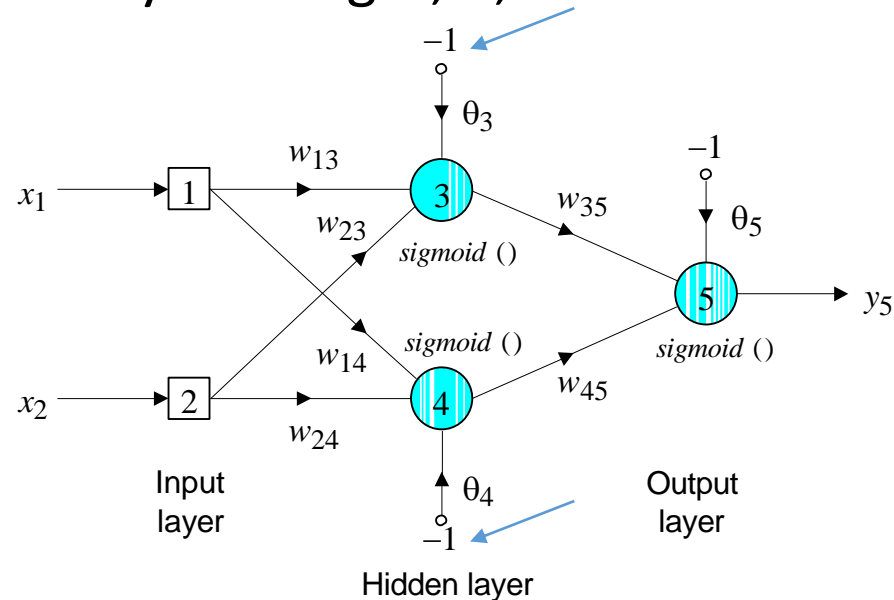
# Three-layer network for solving the Exclusive-OR operation (1)

This example is not in the exam,  
but it is interesting!



# Three-layer network for solving the Exclusive-OR operation (2)

- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight,  $\theta$ , connected to a fixed input equal to -1.



- The initial weights and threshold levels are set randomly as follows:

$$w_{13} = 0.5, w_{14} = 0.9, w_{23} = 0.4, w_{24} = 1.0, w_{35} = -1.2, \\ w_{45} = 1.1, \theta_3 = 0.8, \theta_4 = -0.1 \text{ and } \theta_5 = 0.3.$$

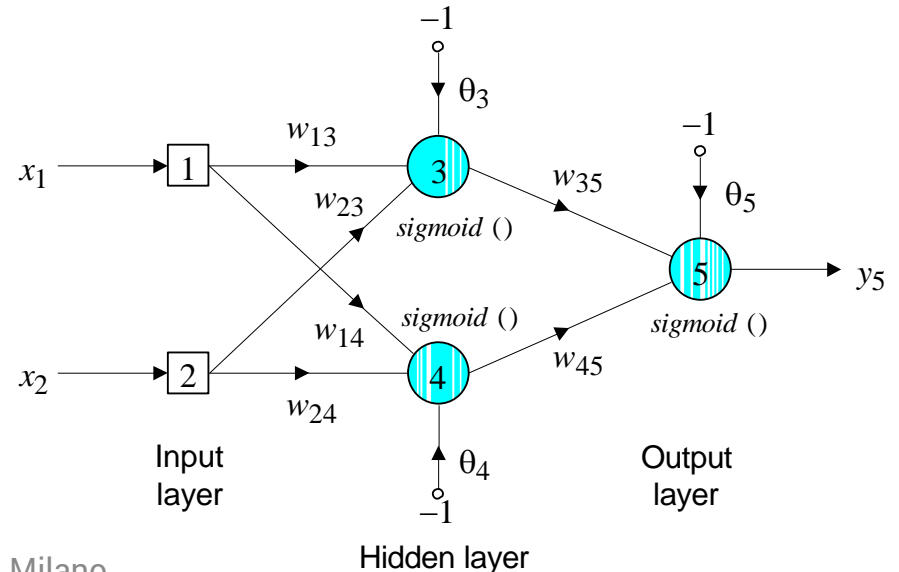
## Three-layer network for solving the Exclusive-OR operation (3)

- We consider a training set where inputs  $x_1$  and  $x_2$  are equal to 1 and desired output  $y_{d,5}$  is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = \text{sigmoid}(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1 / \left[ 1 + e^{-(1 \cdot 0.5 + 1 \cdot 0.4 - 1 \cdot 0.8)} \right] = 0.5250$$

$$y_4 = \text{sigmoid}(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1 / \left[ 1 + e^{-(1 \cdot 0.9 + 1 \cdot 1.0 + 1 \cdot 0.1)} \right] = 0.8808$$

Inputs		Desired output
$x_1$	$x_2$	$y_d$
1	1	0



## Three-layer network for solving the Exclusive-OR operation (4)

Inputs		Desired output
$x_1$	$x_2$	$y_d$
1	1	0

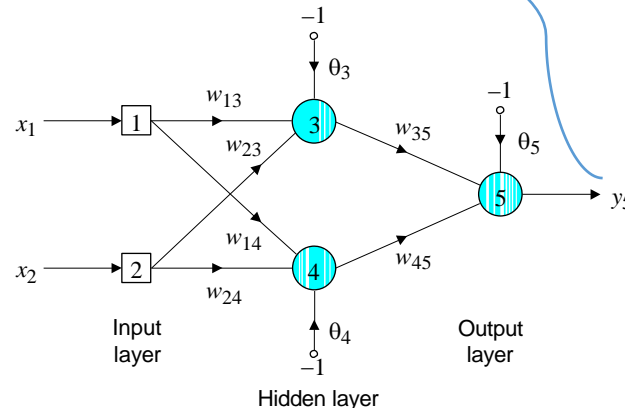
- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = \text{sigmoid}(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1 / \left[ 1 + e^{-(-0.5250 \cdot 1.2 + 0.8808 \cdot 1.1 - 1 \cdot 0.3)} \right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

Inputs		Desired output
$x_1$	$x_2$	$y_d$
1	1	0





# Three-layer network for solving the Exclusive-OR operation (5)

$$\Delta w_i(p) = \alpha \cdot y_i(p) \cdot e(p)$$

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error,  $e$ , from the output layer backward to the input layer.
- First, we calculate the error gradient for neuron 5 in the output layer:

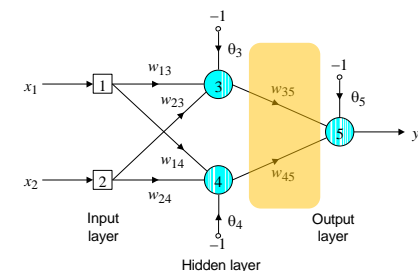
$$\delta_5 = y_5 (1 - y_5) e = 0.5097 \cdot (1 - 0.5097) \cdot (-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter,  $\alpha$ , is equal to 0.1:

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$



## Three-layer network for solving the Exclusive-OR operation (6)

$$\delta_j(p) = y_j(p) \cdot [1 - y_j(p)] \cdot \sum \delta_k(p) w_{jk}(p)$$

$$\Delta w_{ij}(p) = \alpha \cdot x_i(p) \cdot \delta_j(p)$$

- Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \cdot \delta_5 \cdot w_{35} = 0.5250 \cdot (1 - 0.5250) \cdot (-0.1274) \cdot (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \cdot \delta_5 \cdot w_{45} = 0.8808 \cdot (1 - 0.8808) \cdot (-0.1274) \cdot 1.1 = -0.0147$$

- We then determine the weight corrections:

$$\Delta w_{13} = \alpha \cdot x_1 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

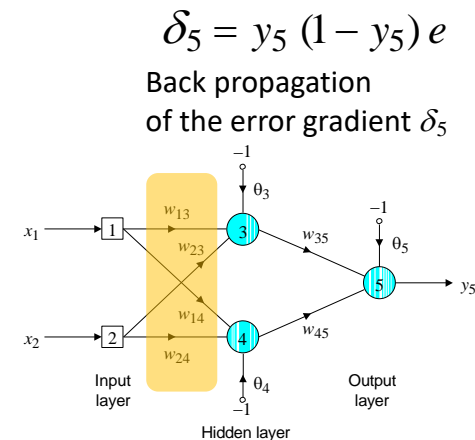
$$\Delta w_{23} = \alpha \cdot x_2 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \cdot (-1) \cdot \delta_3 = 0.1 \cdot (-1) \cdot 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \cdot x_1 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \cdot x_2 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \cdot (-1) \cdot \delta_4 = 0.1 \cdot (-1) \cdot (-0.0147) = 0.0015$$



# Three-layer network for solving the Exclusive-OR operation (7)

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

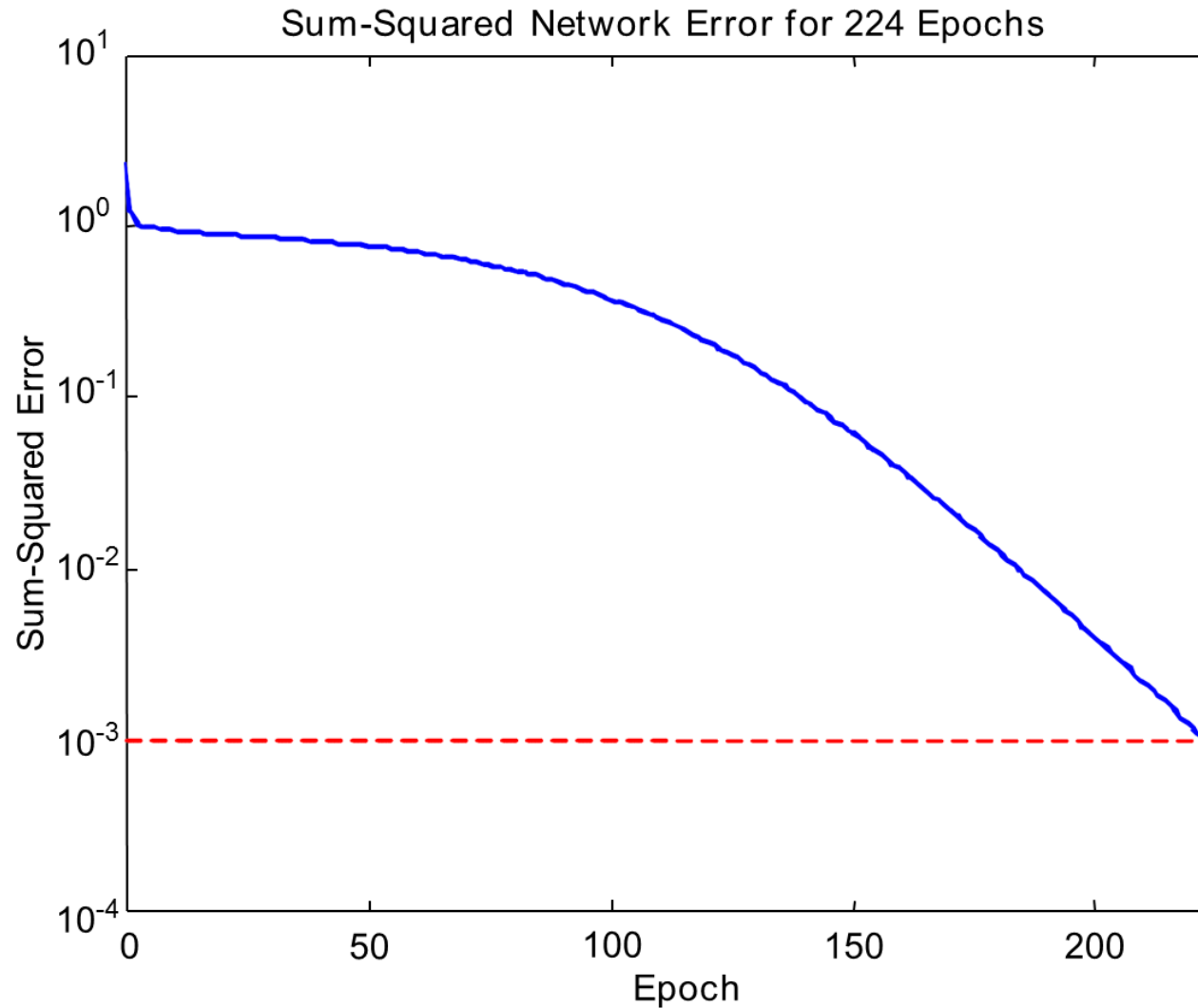
$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

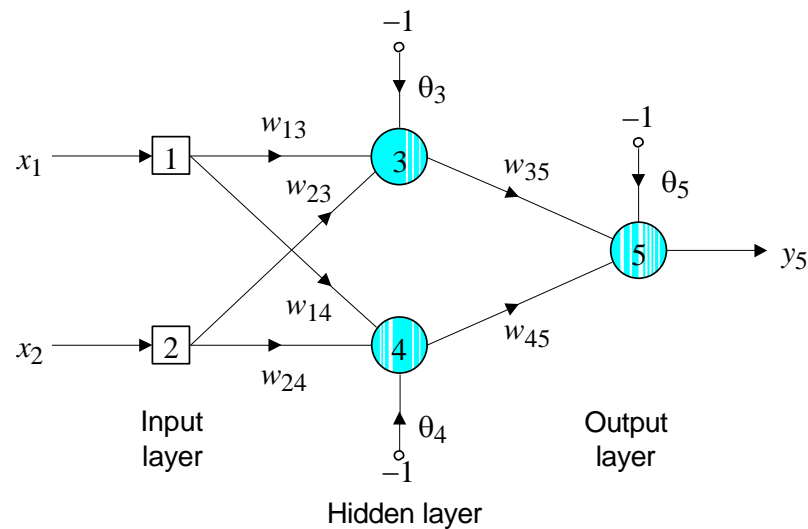
- The training process is repeated until the sum of squared errors is less than 0.001.

# Learning curve for operation Exclusive-OR



# Final results of three-layer network learning

Inputs		Desired output $y_d$	Actual output $y_5$	Error $e$	Sum of squared errors
$x_1$	$x_2$				
1	1	0	0.0155	-0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	-0.0175	

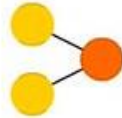


# Neural Networks

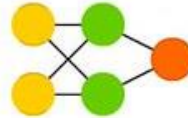
©2016 Fjodor van Veen - asimovinstitute.org

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

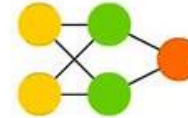
Perceptron (P)



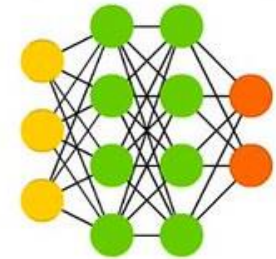
Feed Forward (FF)



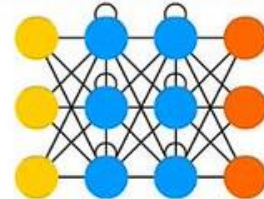
Radial Basis Network (RBF)



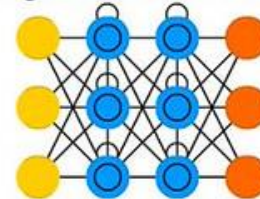
Deep Feed Forward (DFF)



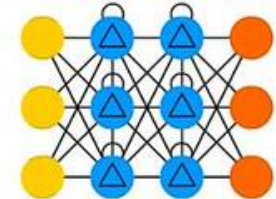
Recurrent Neural Network (RNN)



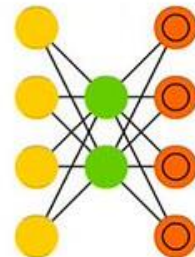
Long / Short Term Memory (LSTM)



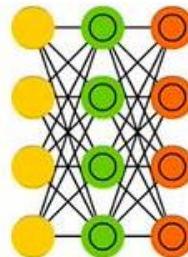
Gated Recurrent Unit (GRU)



Auto Encoder (AE)



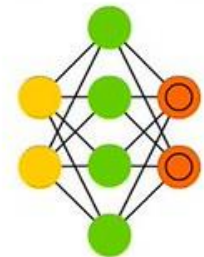
Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



Markov Chain (MC)



Hopfield Network (HN)



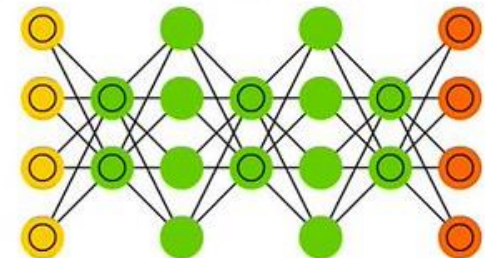
Boltzmann Machine (BM)



Restricted BM (RBM)



Deep Belief Network (DBN)





A mostly complete chart of

# Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

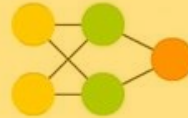
today!

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

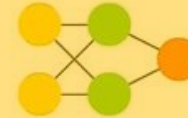
Perceptron (P)



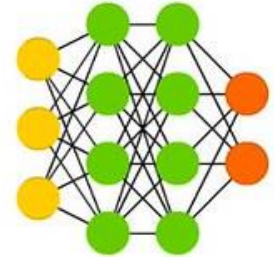
Feed Forward (FF)



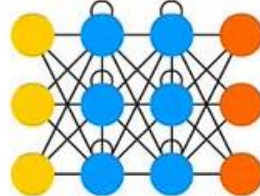
Radial Basis Network (RBF)



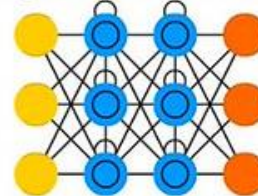
Deep Feed Forward (DFF)



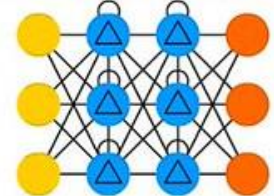
Recurrent Neural Network (RNN)



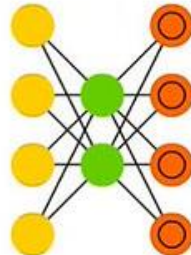
Long / Short Term Memory (LSTM)



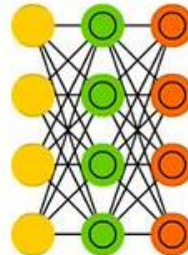
Gated Recurrent Unit (GRU)



Auto Encoder (AE)



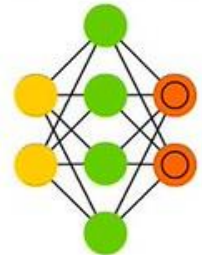
Variational AE (VAE)



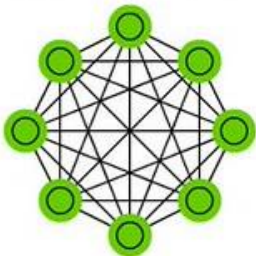
Denoising AE (DAE)



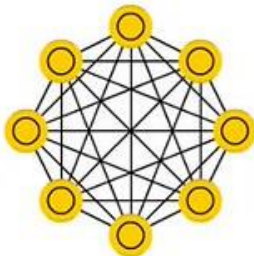
Sparse AE (SAE)



Markov Chain (MC)



Hopfield Network (HN)



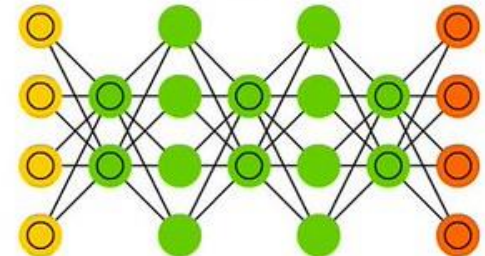
Boltzmann Machine (BM)



Restricted BM (RBM)



Deep Belief Network (DBN)

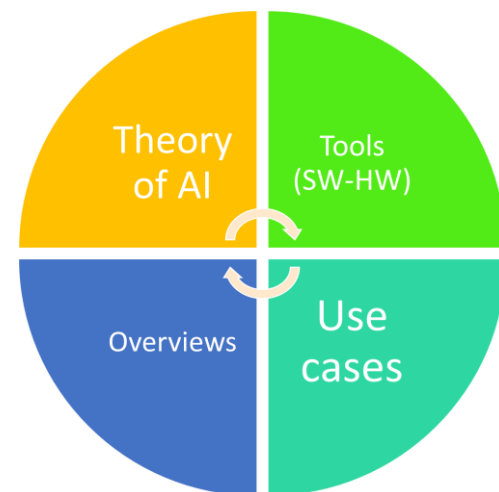




# Toolboxes

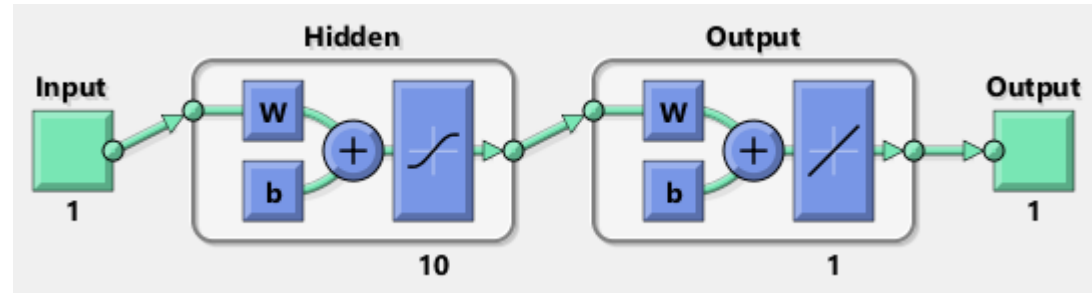
## Matlab

Feedforward neural networks





# Matlab



```
[x,t] = simplefit_dataset;  
net = feedforwardnet(10);  
net = train(net,x,t);  
view(net)  
y = net(x);  
perf = perform(net,y,t)
```

perf =

1.4639e-04

In Colab the information hiding  
is less evident...

There will be dedicated lab  
sessions about that.

# Main points



- The perceptron
  - Learning equations

$$\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)$$

- Multilayer neural networks
  - Theory
  - Application to a simple example
- Feedforward neural networks in Matlab (intro)

