

# INTRODUCTORY APPLIED MACHINE LEARNING

---

Yan-Fu Kuo

Dept. of Bio-industrial Mechatronics Engineering

National Taiwan University

Today:

- Artificial neural network

# Outline

- Goals
- Introduction
- Single-layer perceptron networks
- Learning rules for single-layer perceptron networks
  - Perceptron learning rule
  - Adaline leaning rule
  - $\delta$ -leaning rule
- Multilayer perceptron
  - Back propagation learning algorithm

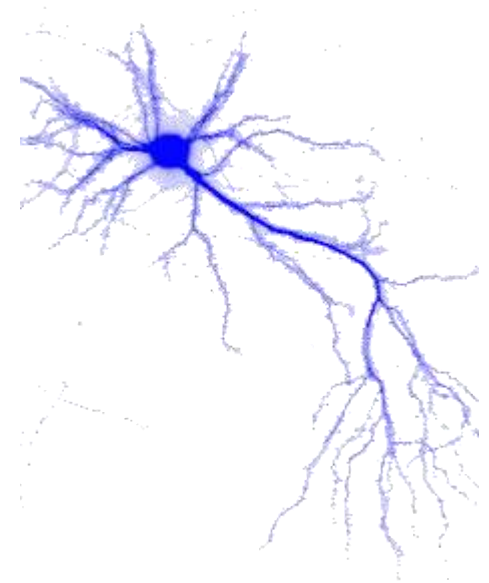
# Goals

- After this, you should be able to:
  - Understand the principles of artificial neural network (ANN)
  - Perform fundamental techniques to determine weights for single-layer ANN
  - Be familiar with common activation function for ANN

# Artificial Neural Network

---

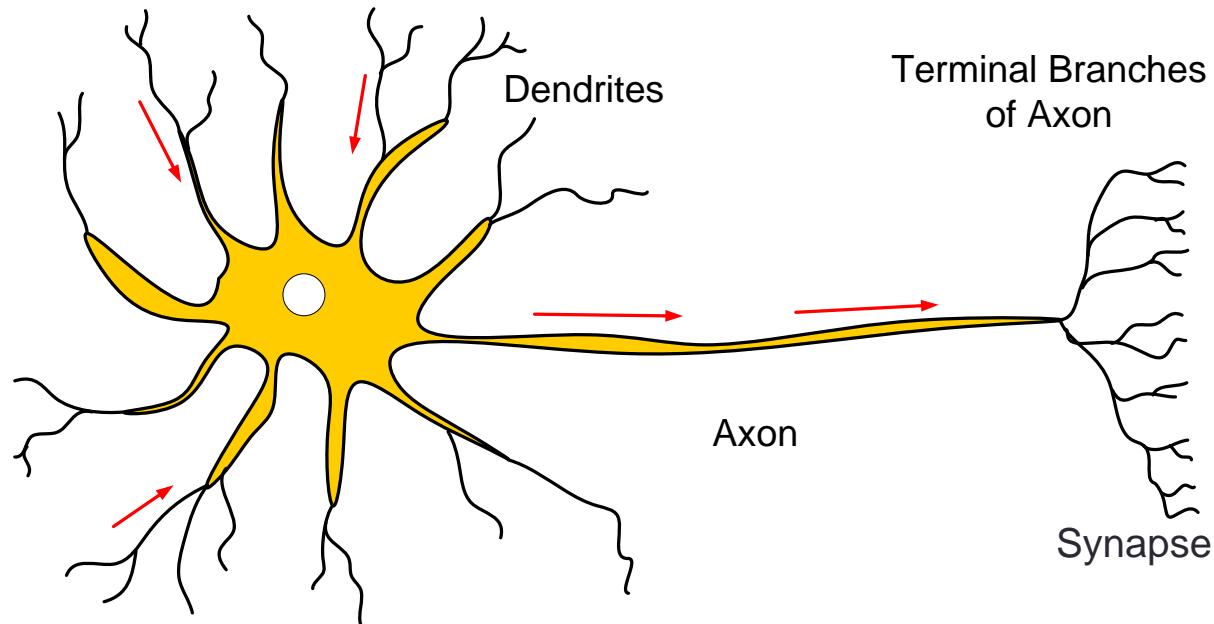
## Introduction



# Historical Background

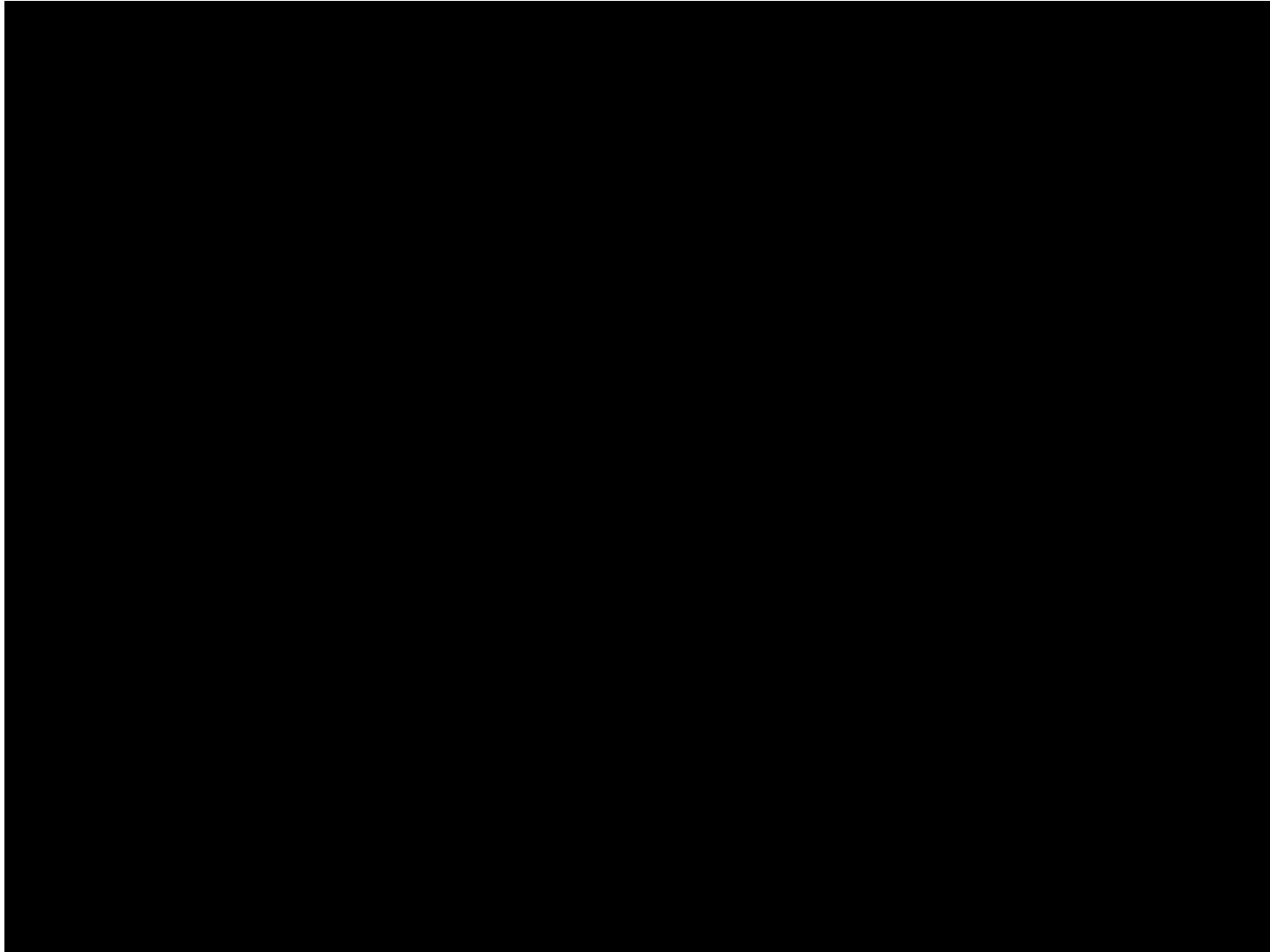
- **1943** McCulloch and Pitts proposed the first computational models of neuron
- **1949** Hebb proposed the first learning rule
- **1958** Rosenblatt introduced the simple single layer networks now called “perceptrons”
- **1969** Minsky and Papert’s exposed limitation of the theory
- **1986** The back-propagation learning algorithm for multi-layer perceptrons was re-discovered and the whole field took off again

# Neurons



- The main purpose of neurons is to receive, analyze and transmit further the information in a form of signals (electric pulses)
- When a neuron sends the information we say that a neuron “fires”

# Neuron Synapse



# Human Nervous System

- Human brain contains  $\sim 10^{11}$  neurons, each of which is connected  $\sim 10^4$  others
- Some scientists compared the brain with a “complex, nonlinear, parallel computer”
- The largest modern neural networks achieve the complexity comparable to a nervous system of a fly
- A neuron is much slower ( $10^{-3}$  sec) compared to a silicon logic gate ( $10^{-9}$  sec); however, the massive interconnection between neurons make up for the comparably slow rate
- Since individual neurons operate in a few milliseconds, calculations do not involve more than about 100 serial steps and the information sent from one neuron to another is very small (a few bits)



# Olny Srmat Poelpe Can Raed Tihs

I cdnuolt blveiee taht I cluod aulacly uesdnatnrd waht I was rdanieg. The phaonmneal pweor of the hmuan mnid, aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttair in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer be in the rghit pclae. The rset can be a taotl mses and you can sitll raed it wouthit a porbelm.

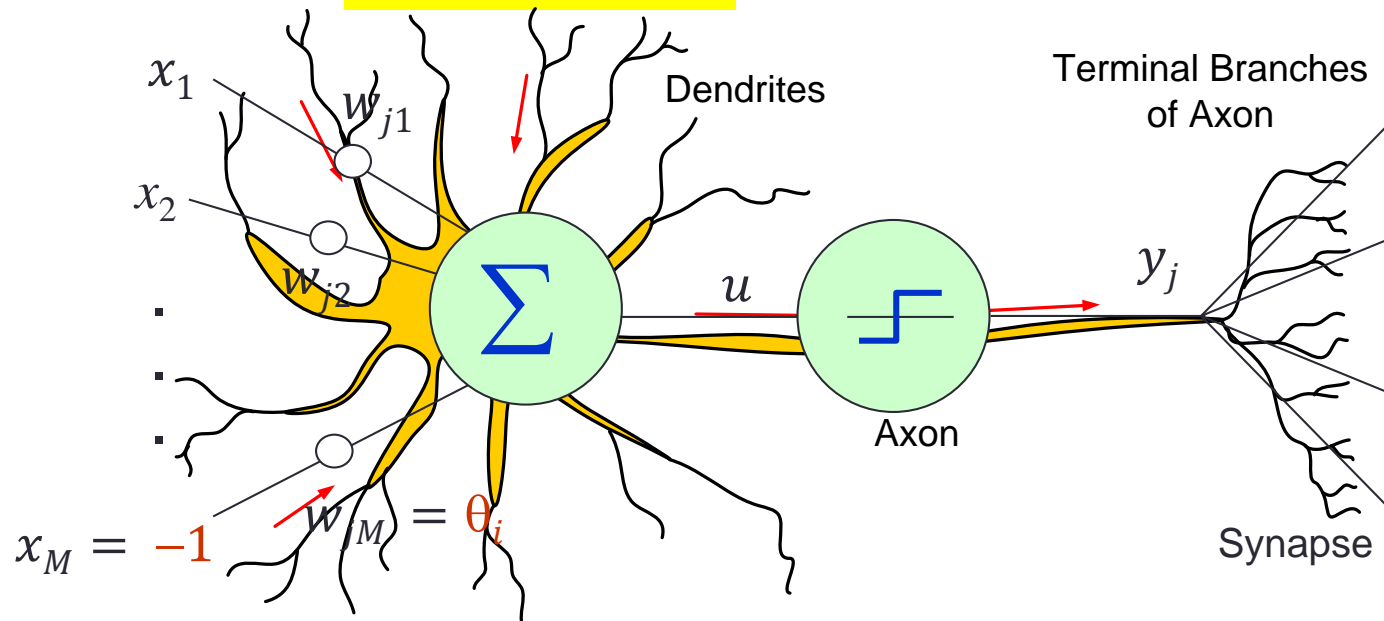
Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Amzanig huh? yaeh and I awlyas tghuhot slpeling was ipmorantt!

# The McCulloch-Pitts Neuron

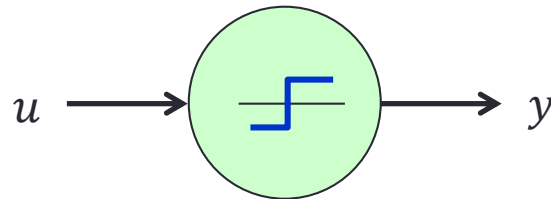
- Also known as a threshold logic unit
- A neuron  $j$  works like:


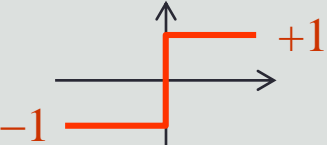

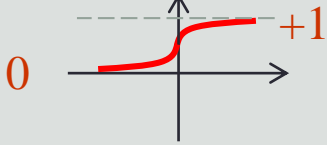
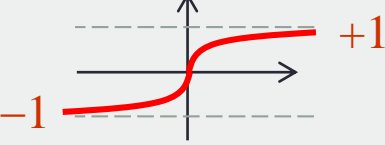
$$u = \sum_{l=1}^M w_{jl} x_l$$

$$y_j = a(u)$$



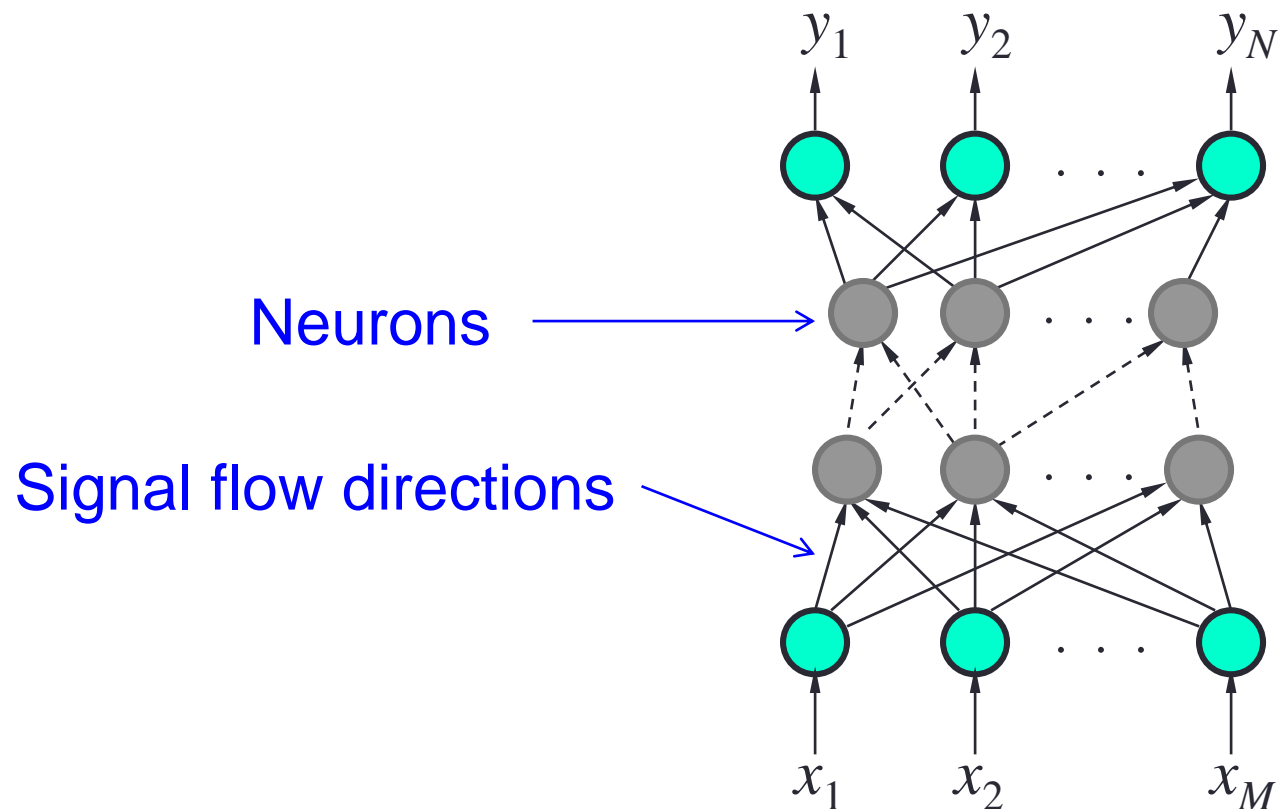
# Typical Activation Function



Linear	Unbounded	
Hard limit	Bounded in $[-1,1]$	
Saturating linear	Bounded in $[-1,1]$	
Unipolar sigmoid	Bounded in $[0,1]$	
Bipolar sigmoid	Bounded in $[-1,1]$	

# Feed-forward Neural Networks

- A neural network that does not contain cycles (feedback loops) is called a feed-forward network (or perceptron)

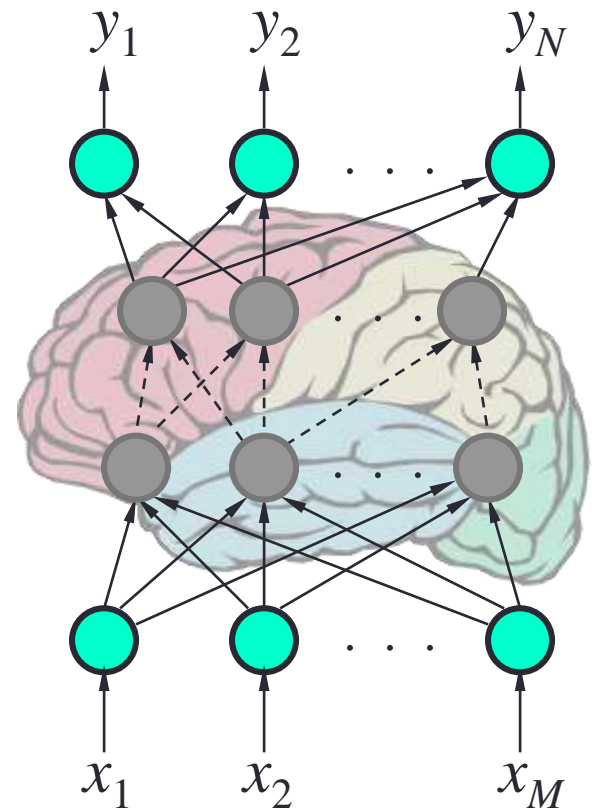


# Layered Structure

Output Layer —

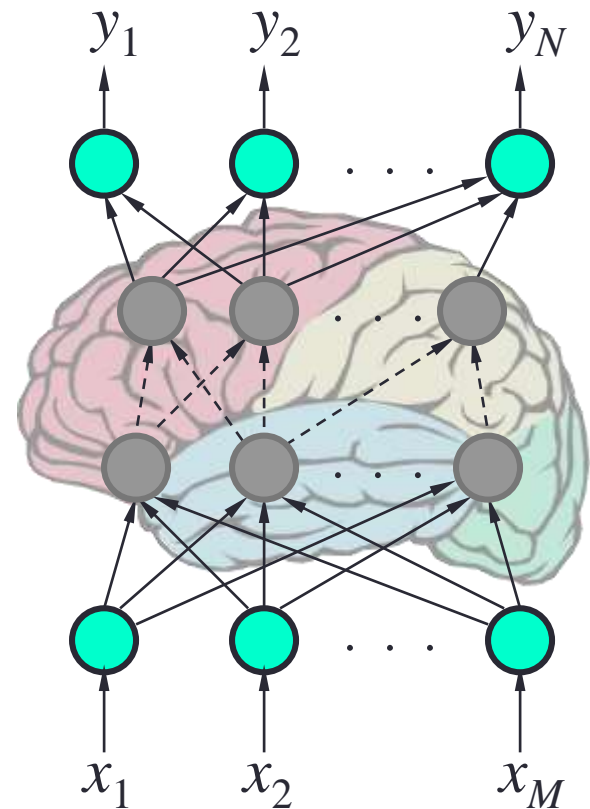
Hidden Layer(s) {

Input Layer —



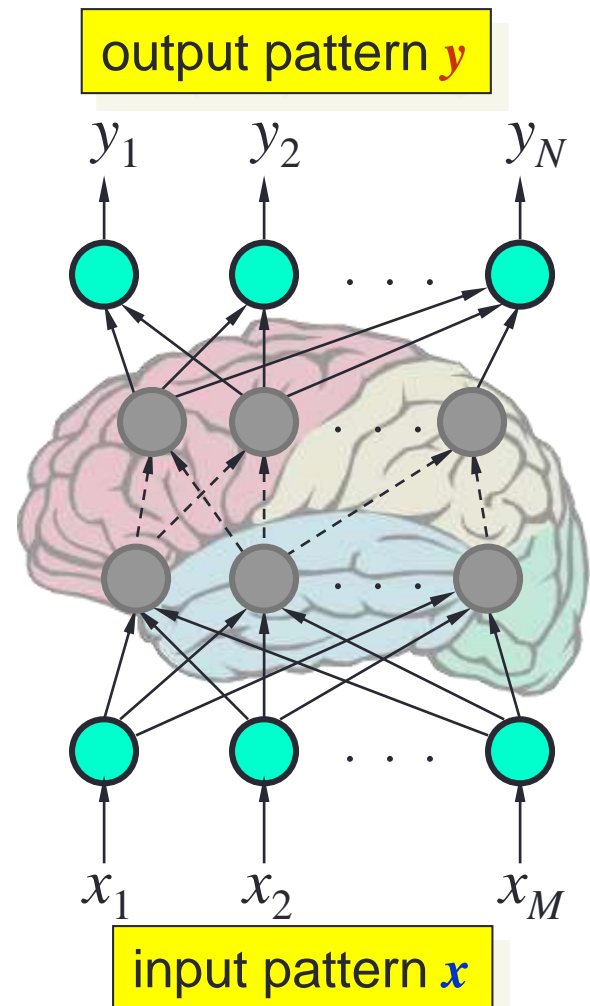
# Knowledge and Memory

- The output behavior of a network is determined by the weights
- Weights - the memory of an NN
- Knowledge - distributed across the network
- Large number of nodes are to increase the storage “capacity” and to ensure that the knowledge is robust



# Classification

- Function:  $x \rightarrow y$
- The NN's output is used to distinguish between and recognize different input patterns
- Different output patterns correspond to particular classes of input patterns
- Networks with hidden layers can be used for solving more complex problems than just a linear pattern classification

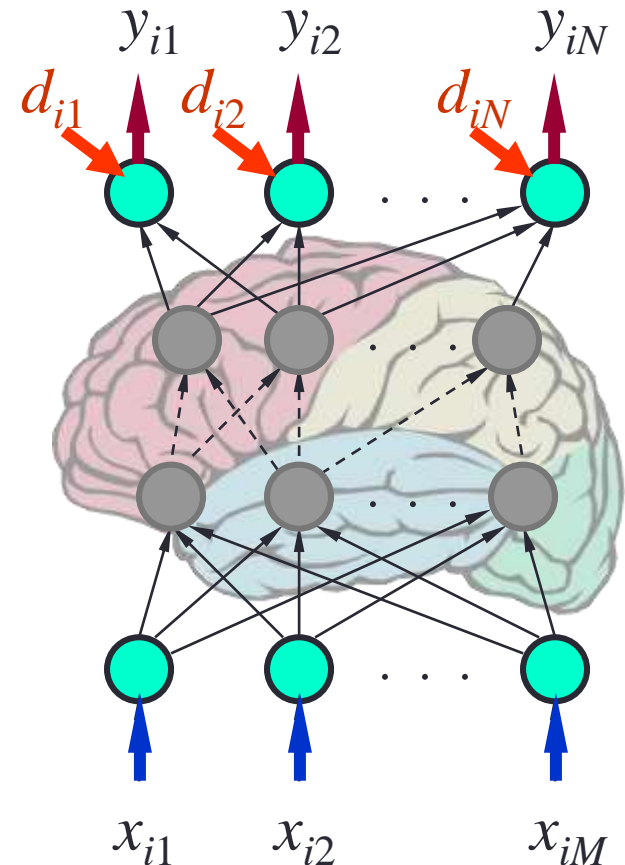


# Training

- Given a set of training samples  $(\mathbf{x}_i, \mathbf{d}_i)$ , where  $i = 1 \dots Q$ 
$$\begin{cases} \mathbf{x}_i = (x_{i1}, \dots, x_{iM}) \\ \mathbf{d}_i = (d_{i1}, \dots, d_{iN}) \end{cases}$$
- The objective of training is to find a set of weights  $\mathbf{w}$  that minimize the error, i.e.,

$$\mathbf{w} = \arg \min_{\mathbf{w}} \|\mathbf{y}_i - \mathbf{d}_i\|^2,$$

where  $\mathbf{y}_i = (y_{i1}, \dots, y_{iN})$





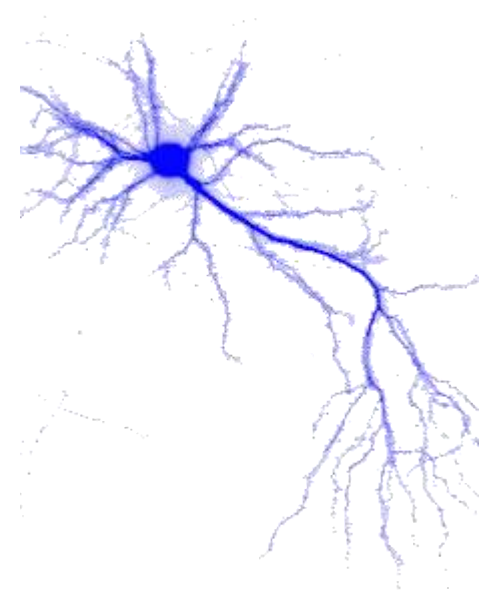
# Artificial Neural Network

---

Single-layer perceptron networks

Learning rules

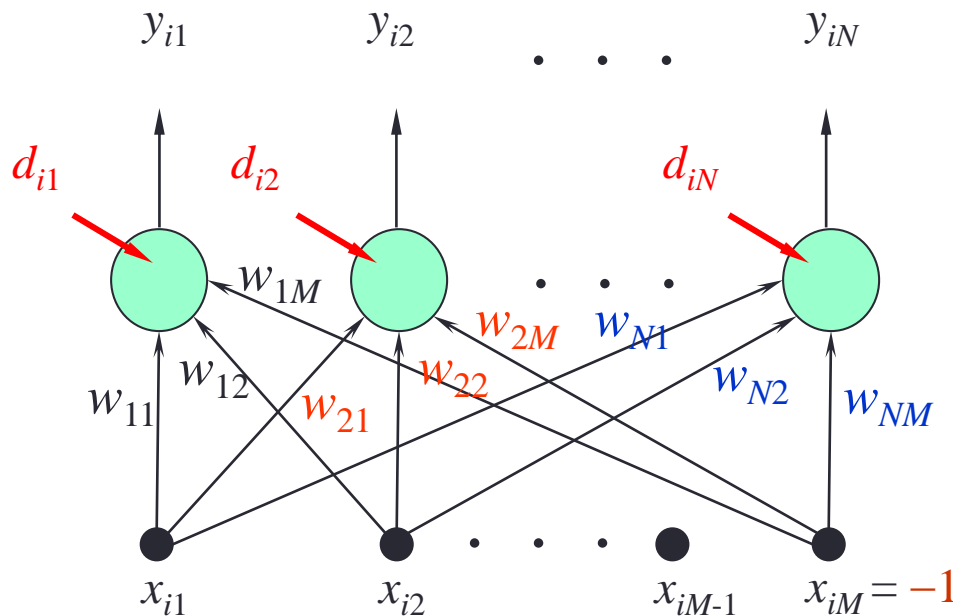
- Perceptron learning rule
- Adaline learning rule
- $\delta$ -learning rule



# Training a Single-layered Perceptron

- For a set of training samples  $(\mathbf{x}_i, \mathbf{d}_i)$ , where  $i = 1 \dots Q$

$$y_{ij} = a(\mathbf{w}_j^T \mathbf{x}_i) = a\left(\sum_{l=1}^M w_{jl} x_{il}\right) = d_{ij}$$



Note:

$$\begin{cases} \mathbf{x}_i = (x_{i1}, \dots, x_{iM}) \\ \mathbf{d}_i = (d_{i1}, \dots, d_{iN}) \end{cases}$$

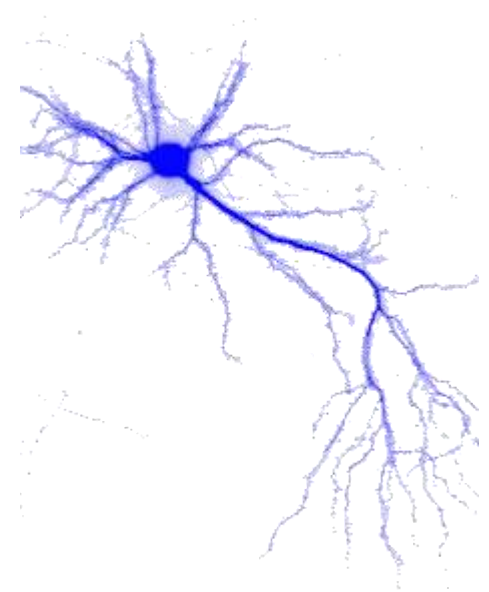
# Artificial Neural Network

---

Single-layer perceptron networks

Learning rules

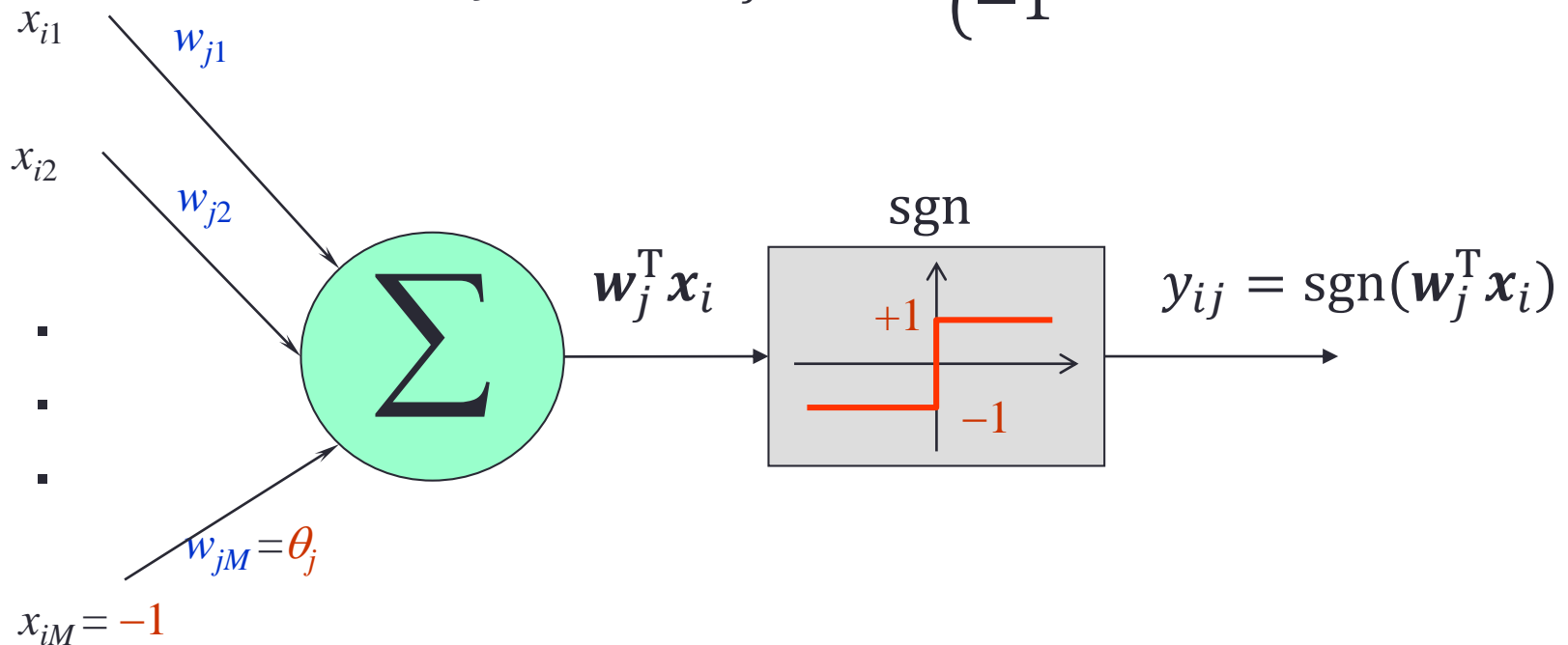
- Perceptron learning rule
- Adaline learning rule
- $\delta$ -learning rule



# Perceptron

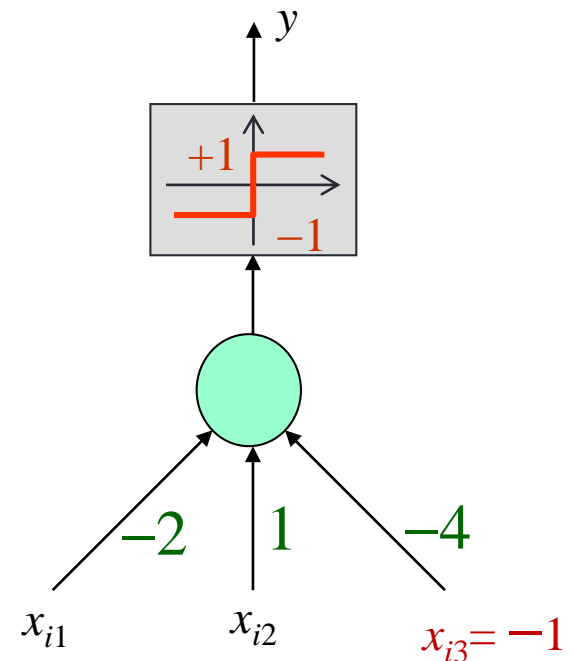
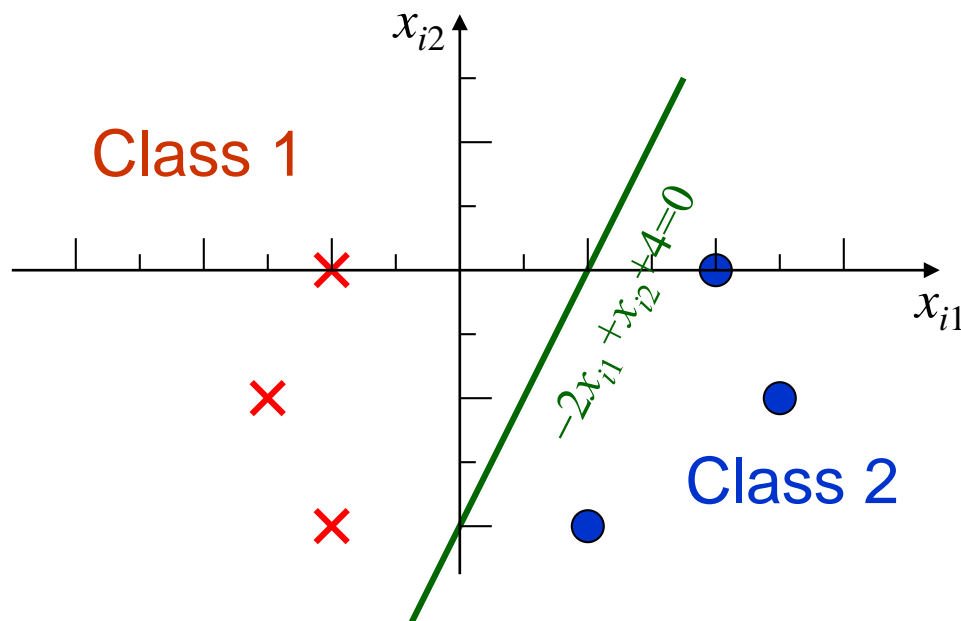
- Train an ANN for “classification”
- Hard limit threshold activation unit

$$y_{ij} = \text{sgn}(\mathbf{w}_j^T \mathbf{x}_i) = \begin{cases} +1 \\ -1 \end{cases}$$



# Example

- **Class 1 (+1):**  $\{ [-1,0]^T, [-1.5, -1]^T, [-1, -2]^T \}$
- **Class 2 (-1):**  $\{ [2,0]^T, [2.5, -1]^T, [1, -2]^T \}$
- **Classifier:**  $y = \text{sgn}(-2x_{i1} + x_{i2} + 4)$



# Augmented Input Vector

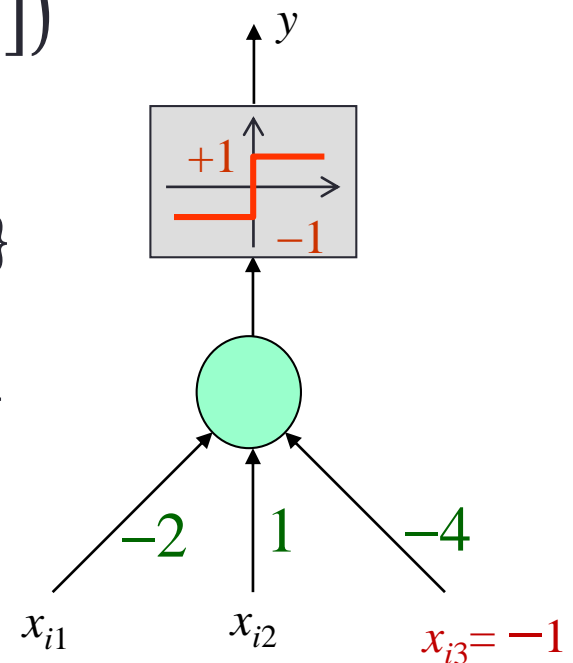
- Vector  $\alpha$  of any dimension can be augmented to  $[\alpha \ -1]$

- **Class 1 (+1):**  $\{ [-1,0]^T, [-1.5, -1]^T, [-1, -2]^T \}$

$$\Rightarrow \text{Class 1 (+1): } \left\{ \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1.5 \\ -1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ -2 \\ -1 \end{bmatrix} \right\}$$

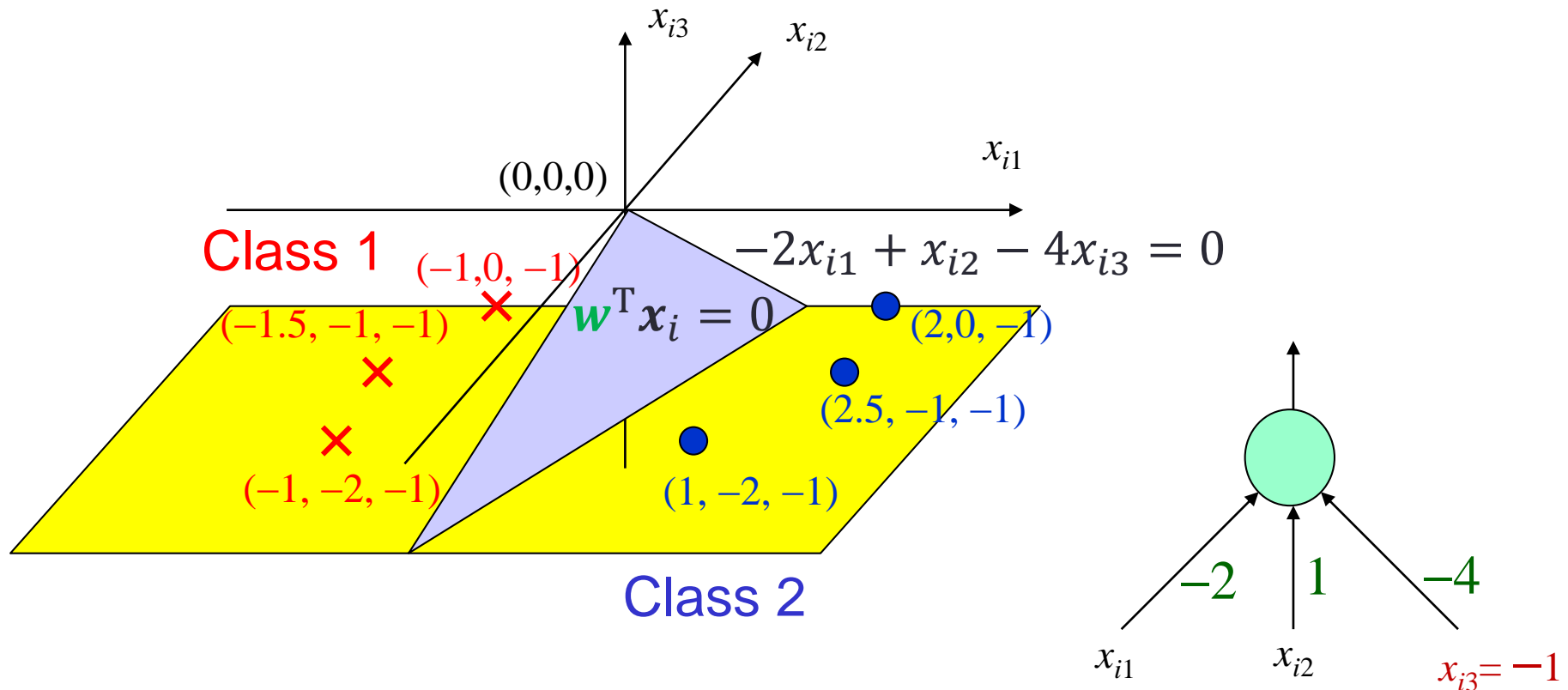
- **Class 2 (-1):**  $\{ [2,0]^T, [2.5, -1]^T, [1, -2]^T \}$

$$\Rightarrow \text{Class 2 (-1): } \left\{ \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 2.5 \\ -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix} \right\}$$



# Augmented Input Vector (Cont'd)

- A plane passes through the origin in the augmented input space with  $\mathbf{w}$  as its normal vector



# Classification Problem Formulation

- Given training sets ( $\mathbf{x} \in \mathbb{R}^M$ )
  - $T_1 = \{\mathbf{x}: \mathbf{x} \in d = +1 \text{ } \oplus \}$
  - $T_2 = \{\mathbf{x}: \mathbf{x} \in d = -1 \text{ } \ominus \}$
- Assume  $T_1$  and  $T_2$  are linearly separable
- For a single perceptron classifier, find  $\mathbf{w} = (w_1, \dots, w_M)^T$  such that

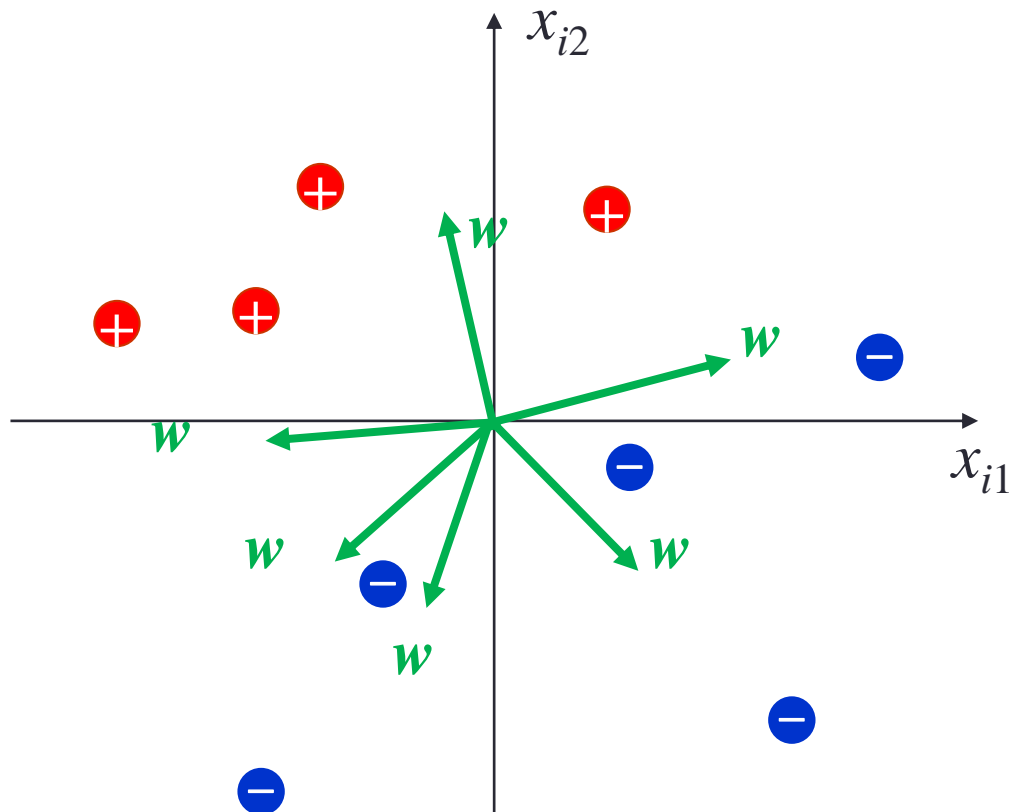
$$y = \text{sgn}(\mathbf{w}^T \mathbf{x}) = \begin{cases} +1, & \mathbf{x} \in T_1 \text{ } \oplus \\ -1, & \mathbf{x} \in T_2 \text{ } \ominus \end{cases}$$

- $\mathbf{w}^T \mathbf{x} = 0$  is a hyperplane passes through the origin of augmented input space



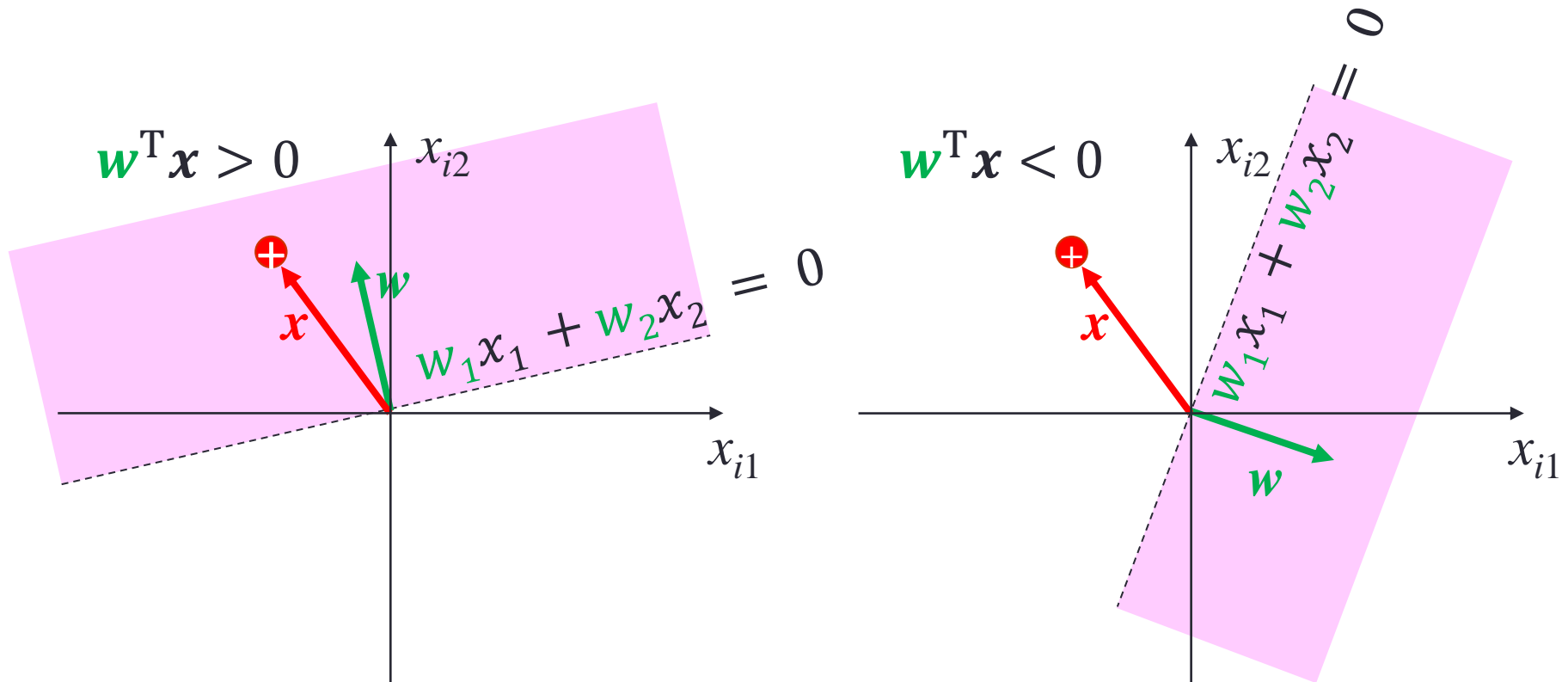
# Objective – Finding Appropriate Weights

- Want to learn a  $w$  from the training data points to discriminate the red and blue



# Inner Product between Weights and Inputs

- Classifier:  $y = \text{sgn}(\mathbf{w}^T \mathbf{x})$
- Objective:  $\mathbf{w}^T \mathbf{x} > 0$  for  $\mathbf{x} \oplus$



# Learning Strategy

- Starting from random initial weights  $\mathbf{w}_0$
- Learn from each individual instance at a time

$$\Delta \mathbf{w}_0 = \alpha \mathbf{x}_i, \quad \Re \ni \alpha > 0$$

- In each iteration, a single sample is introduced, and the weight is adjust to minimize the error

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \Delta \mathbf{w}_0$$

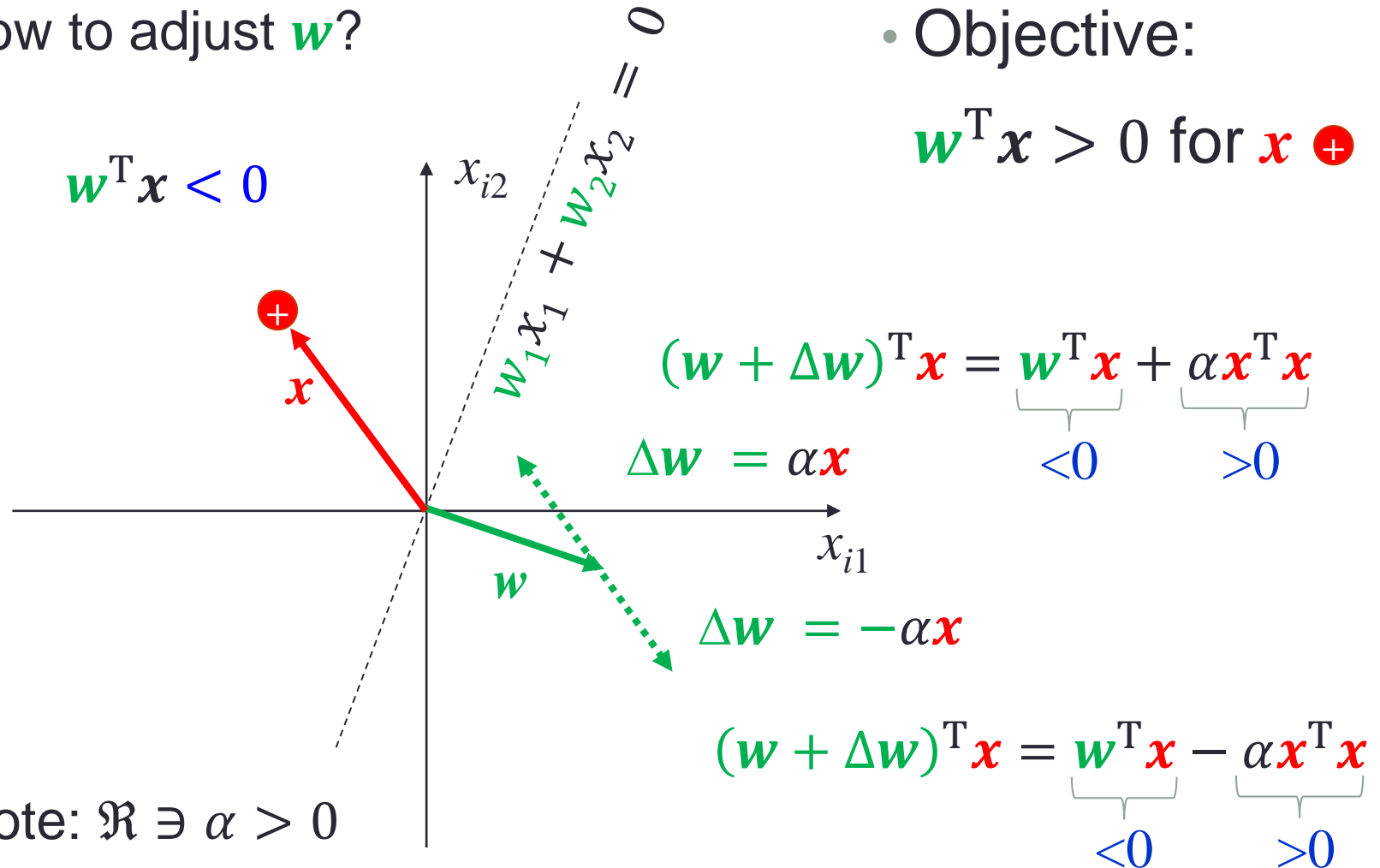
- Keep what have been previously learned in the weights

# How to Determine $\Delta \mathbf{w}_0$ ?

- How to adjust  $\mathbf{w}$ ?

- Objective:

$$\mathbf{w}^T \mathbf{x} > 0 \text{ for } \mathbf{x} \oplus$$



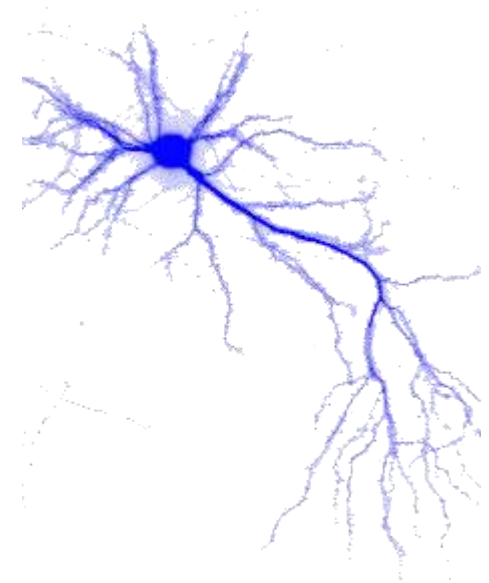
# Artificial Neural Network

---

Single-layer perceptron networks

Learning rule

- Perceptron learning rule
- Adaline learning rule
- $\delta$ -learning rule



# Perceptron Learning Rule

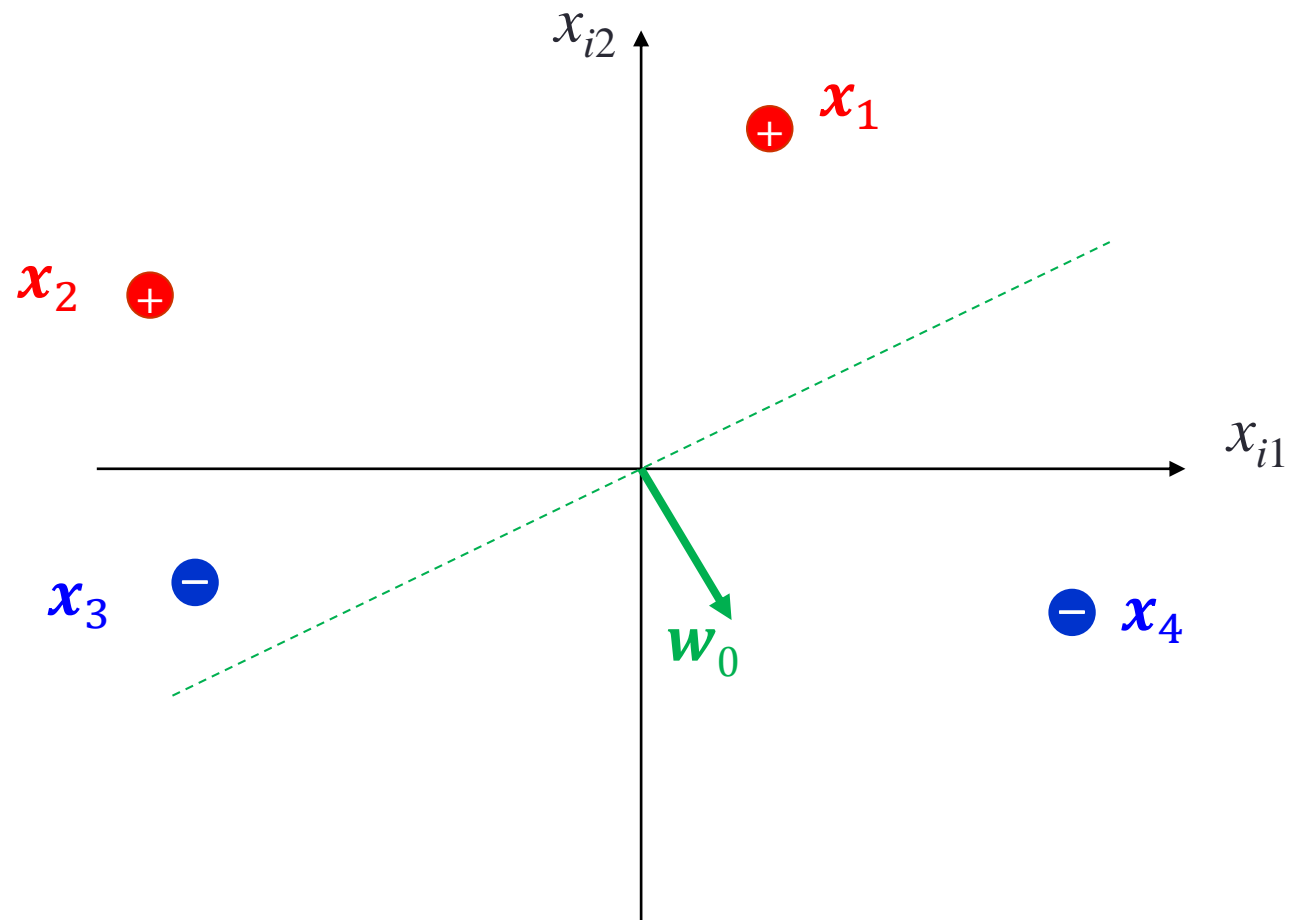
- Upon misclassification on (note:  $\Re \ni \alpha > 0$ )

$$\begin{cases} \Delta \mathbf{w} = \alpha \mathbf{x} & \text{for } d = +1 \text{ } \textcolor{red}{+} \\ \Delta \mathbf{w} = -\alpha \mathbf{x} & \text{for } d = -1 \text{ } \textcolor{blue}{-} \end{cases}$$

- If no misclassification,  $\Delta \mathbf{w} = \mathbf{0}$

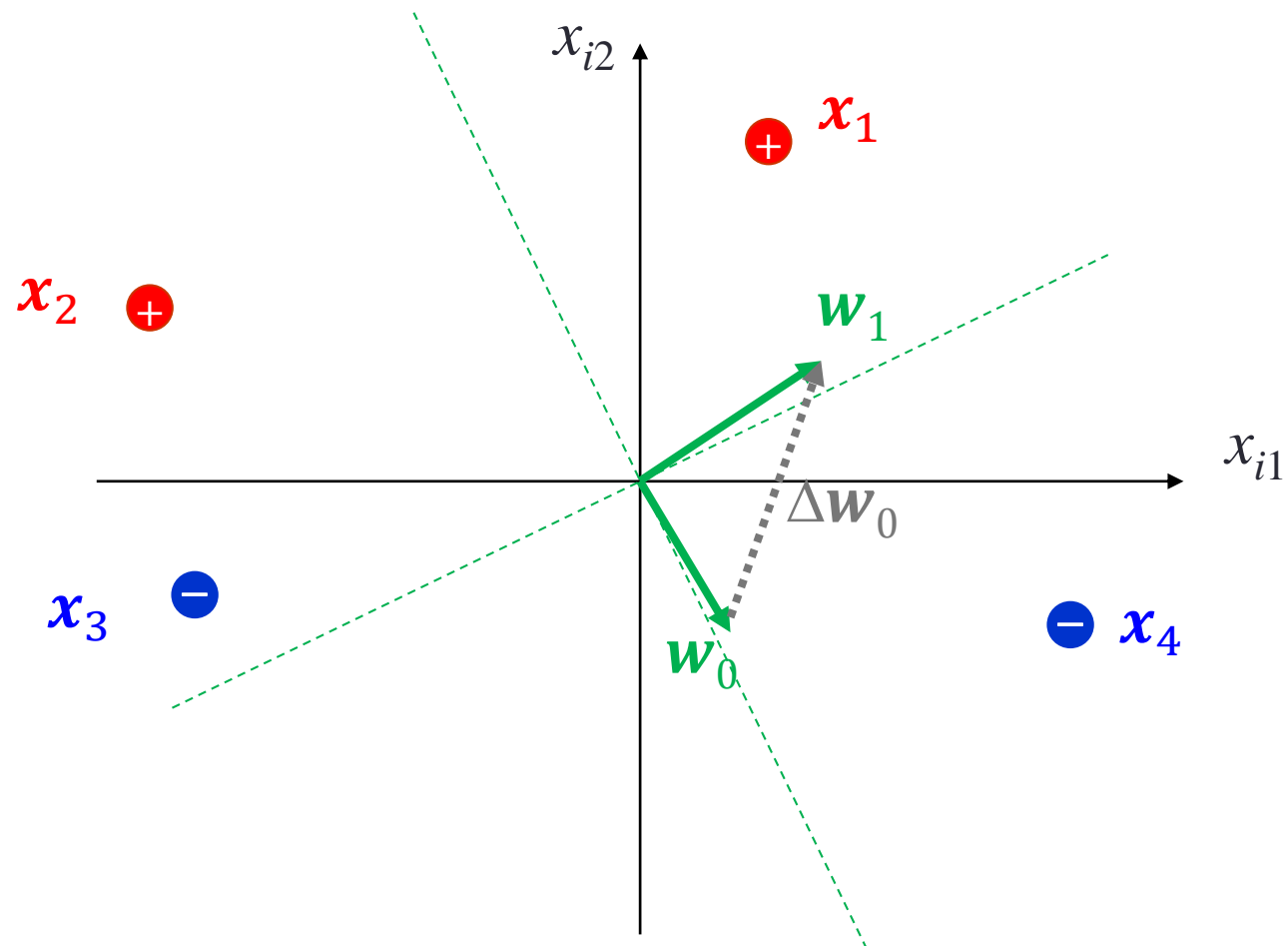
# Example

- Arbitrary weight  $w_0$



# Example (Cont'd)

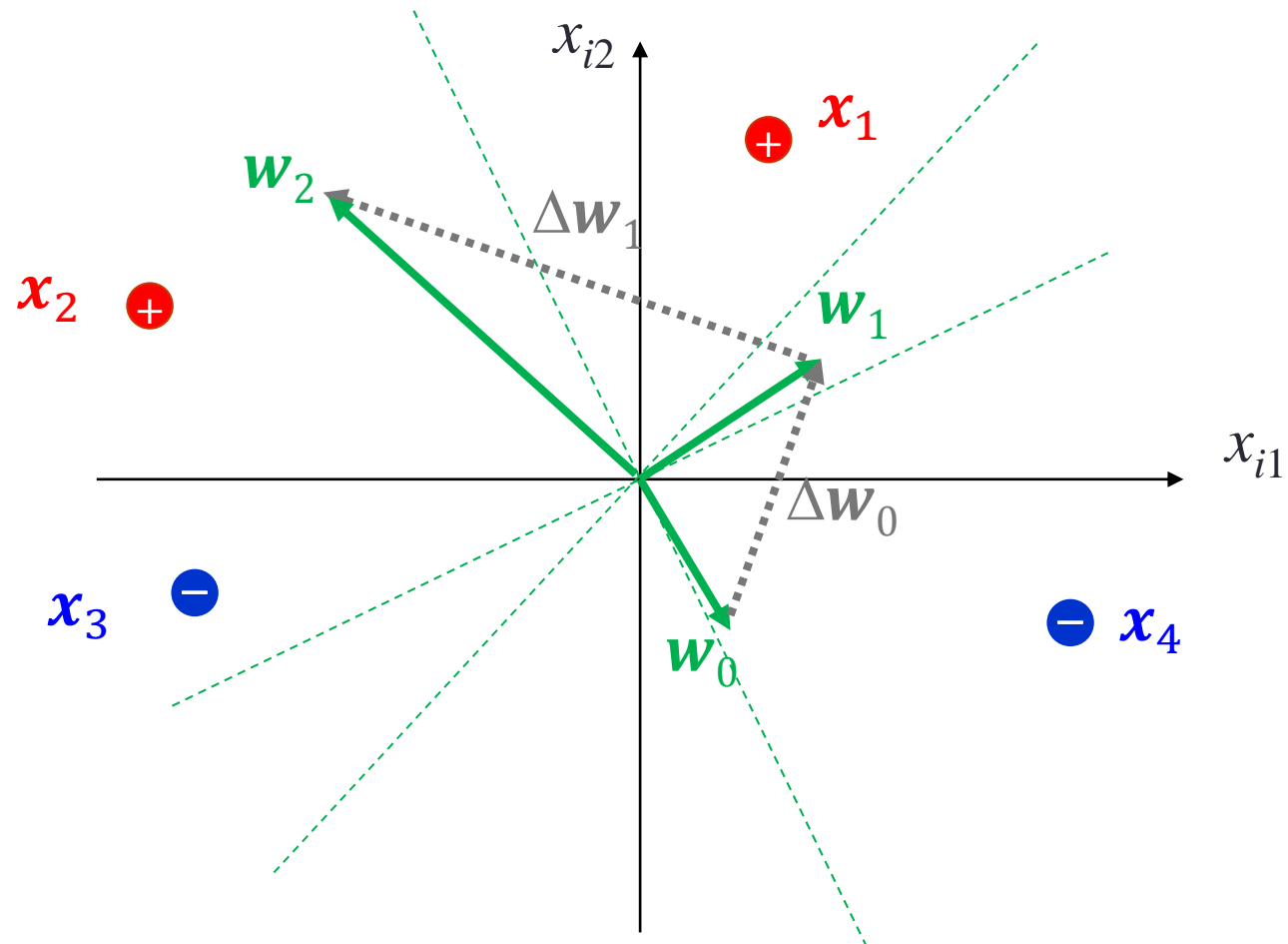
- Given input  $x_1$





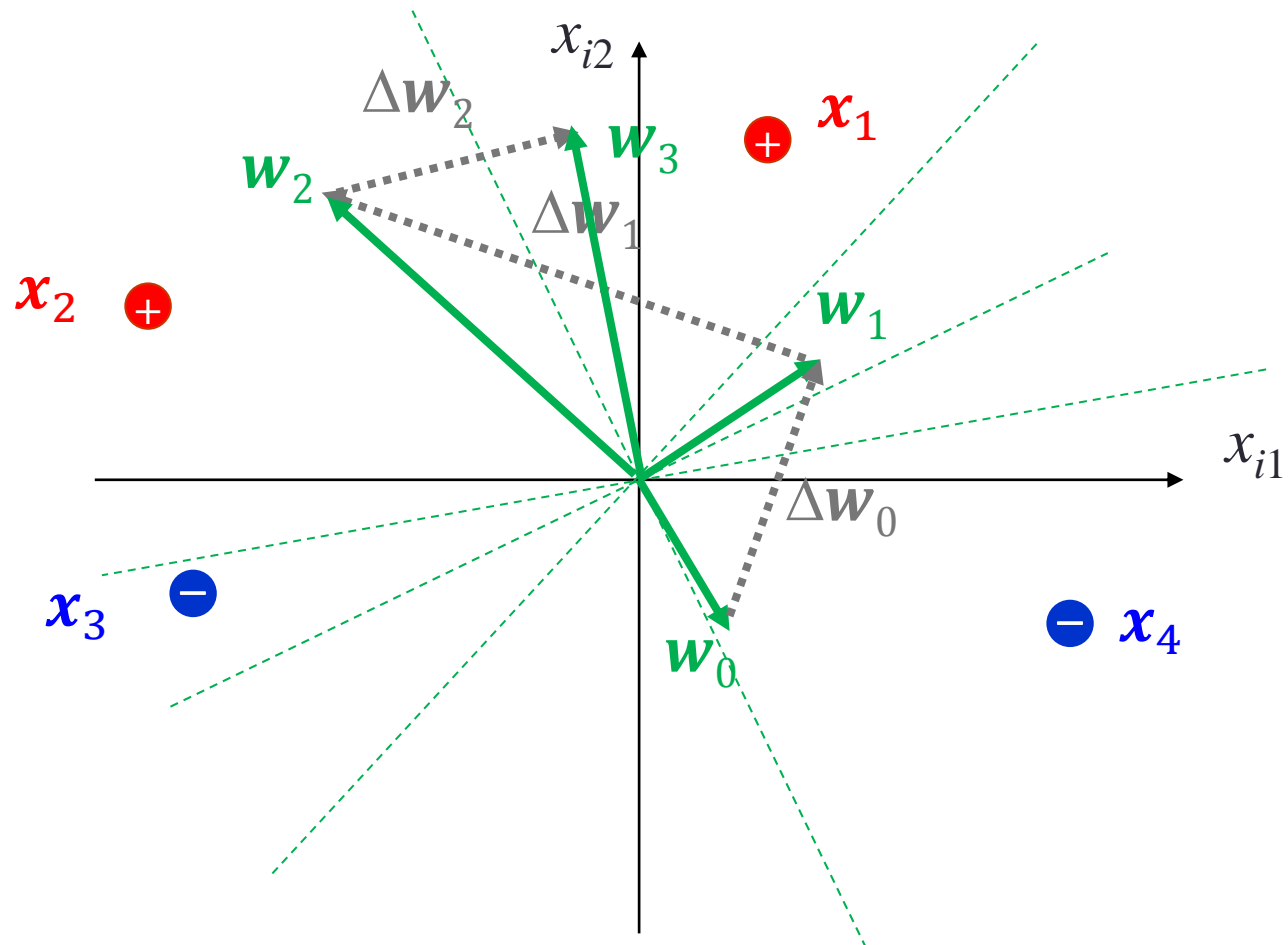
# Example (Cont'd)

- Given input  $x_2$



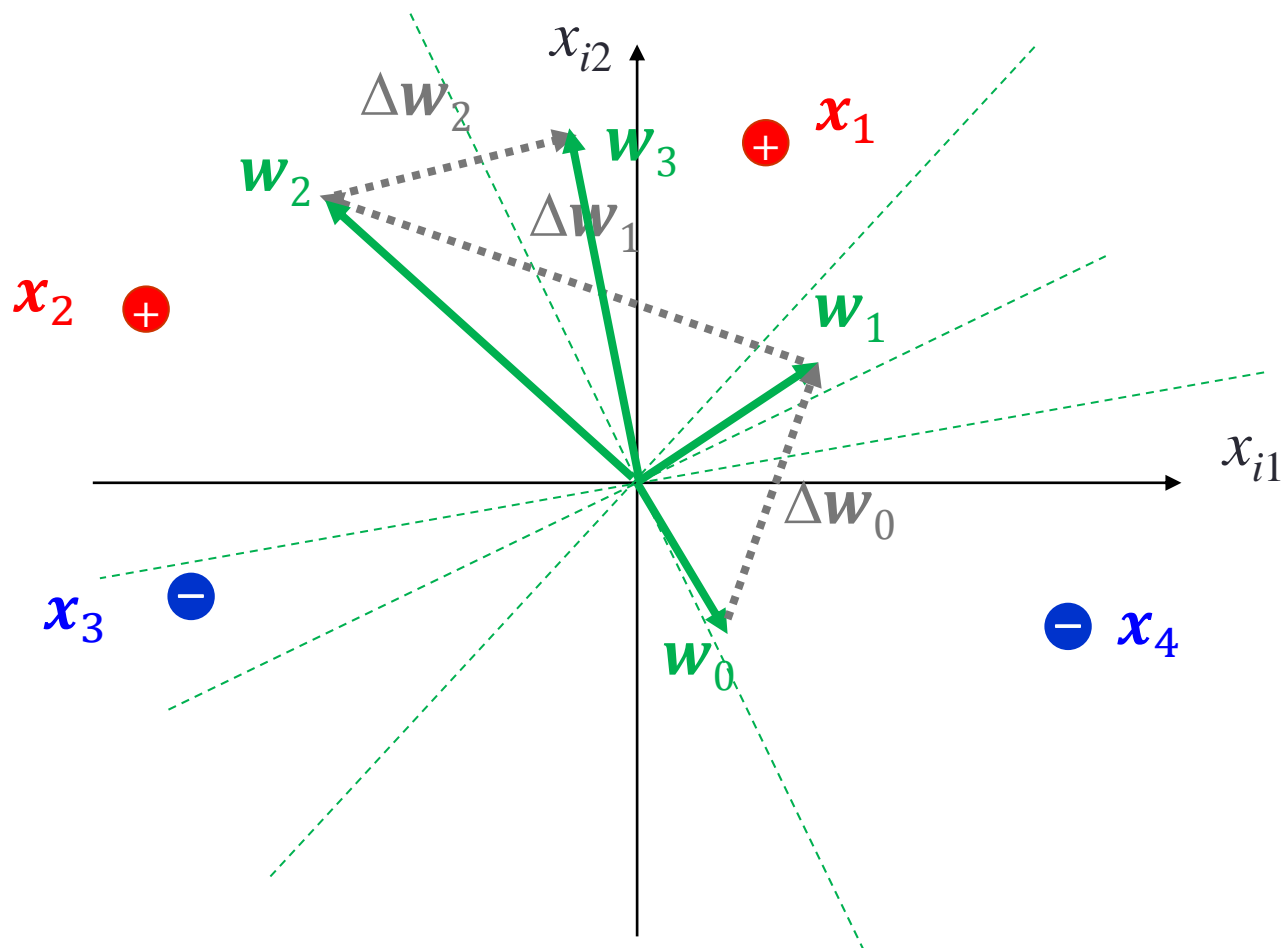
# Example (Cont'd)

- Given input  $x_3$



# Example (Cont'd)

- Given input  $x_4$ ,  $w_3$  is an appropriate weight



# Perceptron Learning Rule

- Upon misclassification on (note:  $\Re \ni \alpha > 0$ )

$$\begin{cases} \Delta \mathbf{w} = \alpha \mathbf{x} & \text{for } d = +1 \text{ (red +)} \\ \Delta \mathbf{w} = -\alpha \mathbf{x} & \text{for } d = -1 \text{ (blue -)} \end{cases}$$

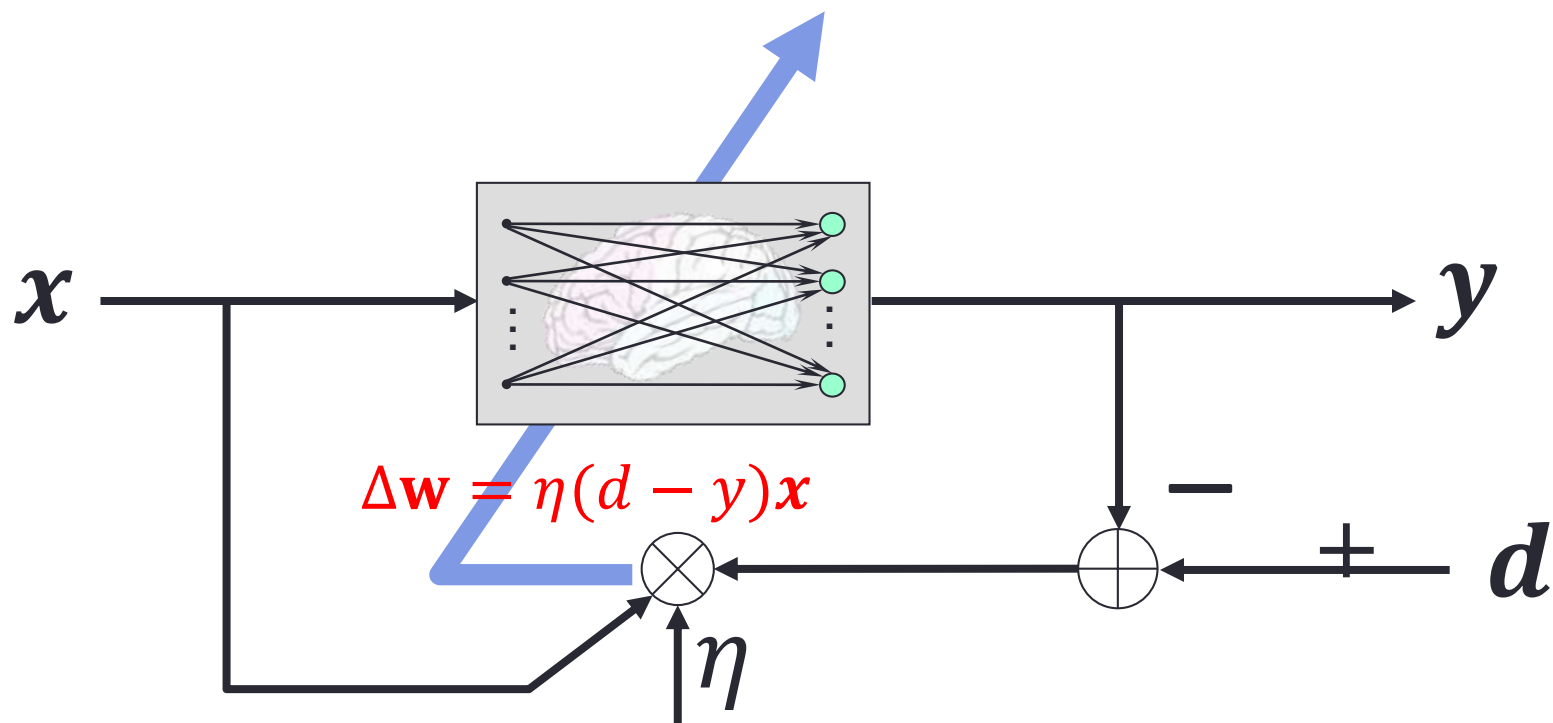
- Define error  $r \in \Re : r = d - y = \begin{cases} +2 & \text{red +} \rightarrow \text{blue -} \\ -2 & \text{blue -} \rightarrow \text{red +} \\ 0 & \end{cases}$

- Learning rule:  $\Delta \mathbf{w} = \eta r \mathbf{x} = \eta (d - y) \mathbf{x}$ ,

where  $0 < \eta \in \Re$  is the learning rate

# Perceptron Learning Rule Block Diagram

- Convergence theorem – if the given training set is linearly separable, the learning process will converge in a finite number of steps



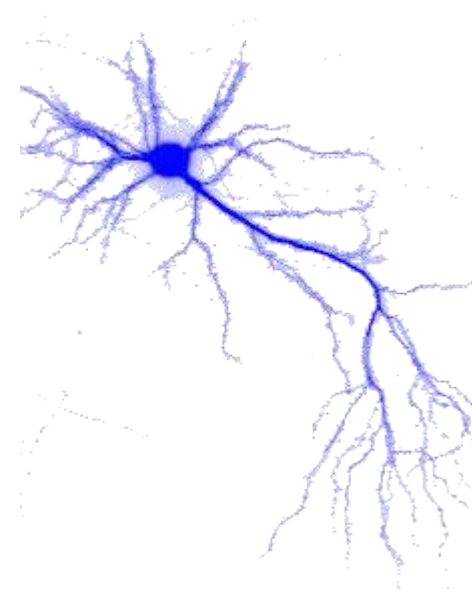
# Artificial Neural Network

---

Single-layer perceptron networks

Learning rule

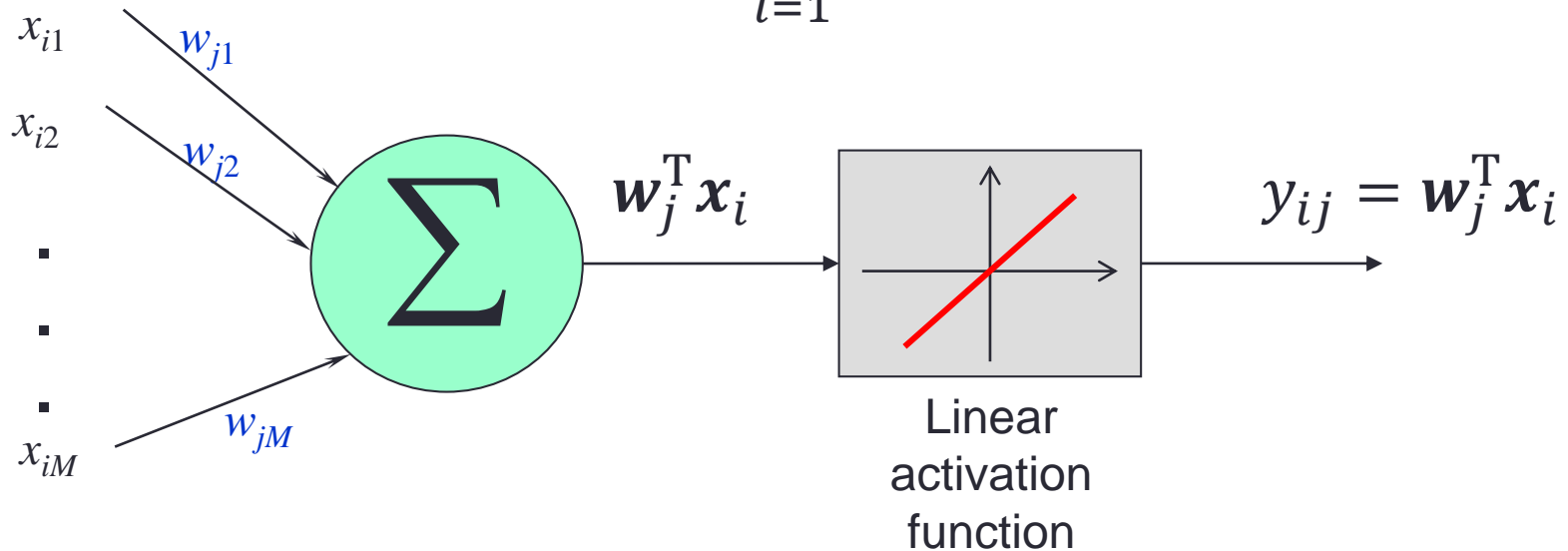
- Perceptron learning rule
- Adaline learning rule
- $\delta$ -learning rule



# Adaline (Adaptive Linear Element)

- Train an ANN for “prediction”
- For a set of training samples  $(\mathbf{x}_i, \mathbf{d}_i)$ , where  $i = 1 \dots Q$
- The output of the neuron  $j$ :

$$y_{ij} = \mathbf{w}_j^T \mathbf{x}_i = \sum_{l=1}^M w_{jl} x_{il} = d_{ij}$$



# Cost Function

- Define misclassification cost function as:

$$\begin{aligned} E(\mathbf{w}_j) &= \frac{1}{2} \sum_{i=1}^Q (d_{ij} - y_{ij})^2 \in \mathbb{R} \\ &= \frac{1}{2} \sum_{i=1}^Q (d_{ij} - \mathbf{w}_j^T \mathbf{x}_i)^2 = \frac{1}{2} \sum_{i=1}^Q \left( d_{ij} - \sum_{l=1}^M w_{jl} x_{il} \right)^2 \end{aligned}$$



# Weight Adjustment

- Objective of learning – minimizing the cost function
- Strategy – adjust the weights along the gradient of cost function:

$$\Delta \mathbf{w}_j = -\eta \nabla_{\mathbf{w}} E(\mathbf{w}_j)$$

# Adaline Learning Rule

- The gradient of the cost function

$$\nabla_{\mathbf{w}} E(\mathbf{w}_j) = \left( \frac{\partial E(\mathbf{w}_j)}{\partial w_{j1}}, \frac{\partial E(\mathbf{w}_j)}{\partial w_{j2}}, \dots, \frac{\partial E(\mathbf{w}_j)}{\partial w_{jM}} \right)^T \in \mathbb{R}^M$$

where

$$\frac{\partial E(\mathbf{w}_j)}{\partial w_{jp}} = - \sum_{i=1}^Q \left( d_{ij} - \sum_{l=1}^M w_{jl} x_{il} \right) x_{ip} = - \sum_{i=1}^Q (d_{ij} - y_{ij}) x_{ip}$$

- Incremental Adaline learning rule:

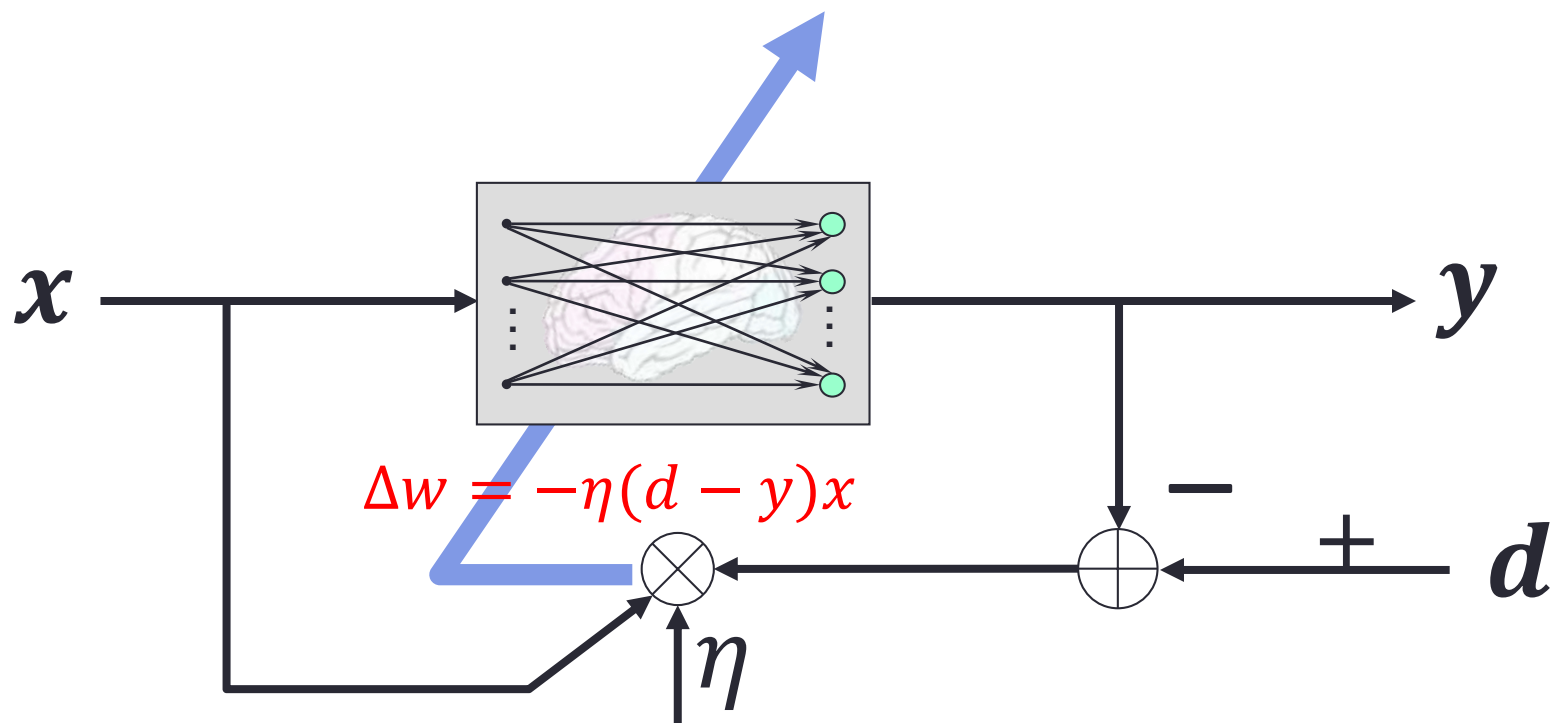
$$\Delta \mathbf{w}_j = -\eta \nabla_{\mathbf{w}} E(\mathbf{w}_j) = -\eta (d_{ij} - y_{ij}) \mathbf{x}_i$$

Incremental: the weight is updated sample point by sample point

Note: no summation term in the incremental learning rule

# Adaline Learning Rule Block Diagram

- Convergence theorem – if the given training set is linearly separable, the learning process will converge in a finite number of steps



# Adaline Convergence Condition

- Conditions conducted by Widrow (1976):
  1. The successive input vectors  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_Q$  are statistically independent
  2. At instance  $i$ , the input vector  $\mathbf{x}_i$  is statistically independent of all previous samples of the desired response  $\mathbf{d}_1, \mathbf{d}_2, \dots, \mathbf{d}_{i-1}$
  3. At instance  $i$ , the desired response  $\mathbf{d}_i$  is dependent on  $\mathbf{x}_i$ , but statistically independent of all previous values of the desired response  $\mathbf{d}_1, \mathbf{d}_2, \dots, \mathbf{d}_{i-1}$
  4. The input vector  $\mathbf{x}_i$  and desired response  $\mathbf{d}_i$  are drawn from Gaussian distributed populations

# Adaline Convergence – $\eta$ Radius

- It can be shown that LMS is convergent if

$$0 < \eta < \frac{2}{\lambda_{\max}}$$

where  $\lambda_{\max}$  is the largest eigenvalue of the correlation matrix for the inputs

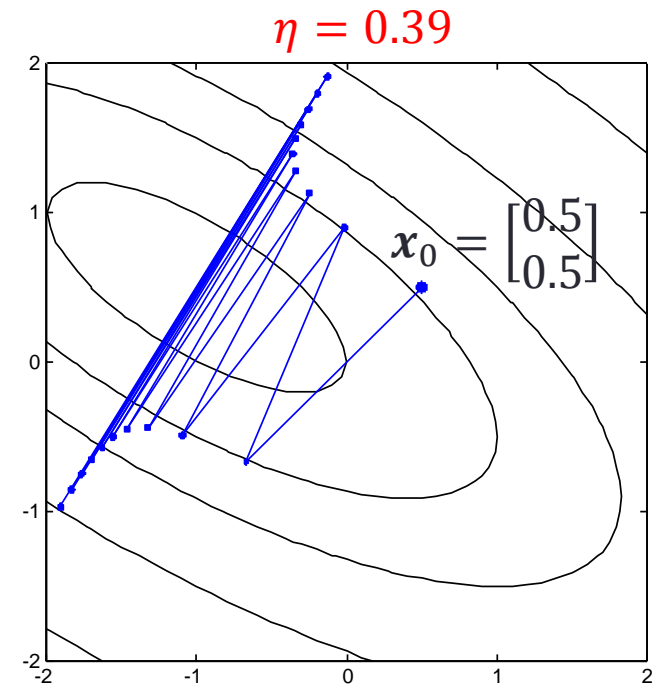
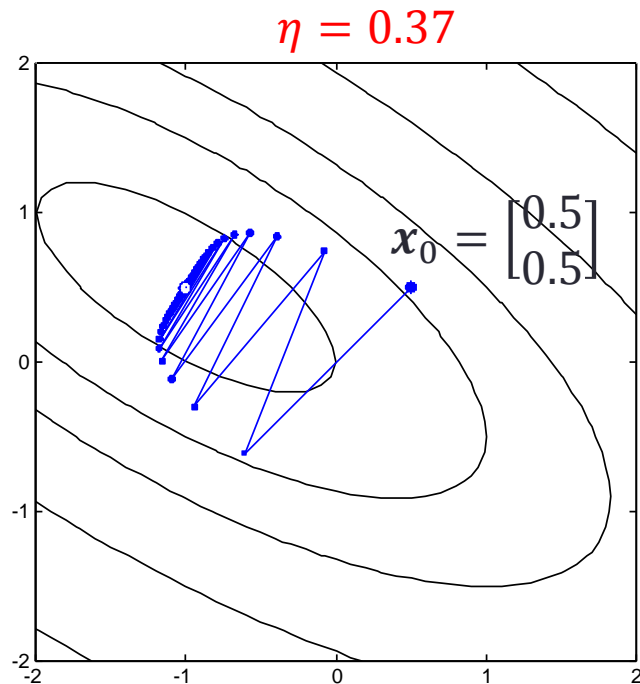
$$\mathbf{R}_x = \lim_{Q \rightarrow \infty} \frac{1}{Q} \sum_{i=1}^Q \mathbf{x}_i \mathbf{x}_i^T$$

- $\lambda_{\max}$  is hardly available, usually the following convergence radius is used:

$$0 < \eta < \frac{2}{\text{tr}(\mathbf{R}_x)}$$

# Convergence Example

- Gradient descent:  $\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla E(\mathbf{x}_n)$
- $E(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1 \Rightarrow \nabla^2 E(\mathbf{x}) = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix}$
- $\lambda_{\max} \left( \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \right) = 5.24 \Rightarrow \eta < \frac{2}{\lambda_{\max}} = 0.38$



# Comparison

	Perceptron learning rule	Adaline learning rule (Widrow-Hoff)
Fundamental	Hebbian rule	Gradient descent
Convergence	In finite steps	Converge asymptotically
Constraint	Linearly separable	Linear independence

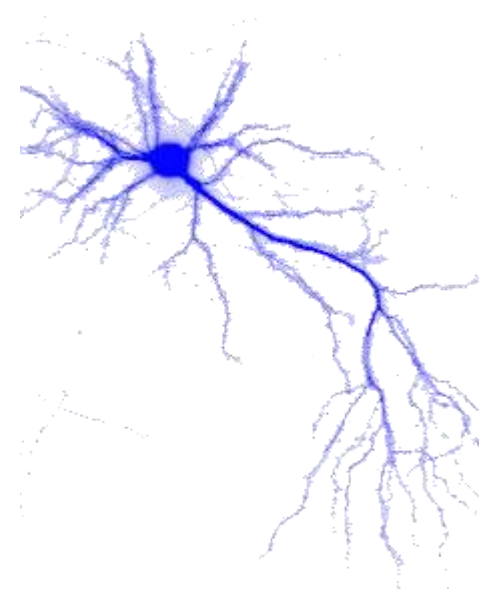
# Artificial Neural Network

---

Single-layer perceptron networks

Learning rule

- Perceptron learning rule
- Adaline learning rule
- $\delta$ -learning rule

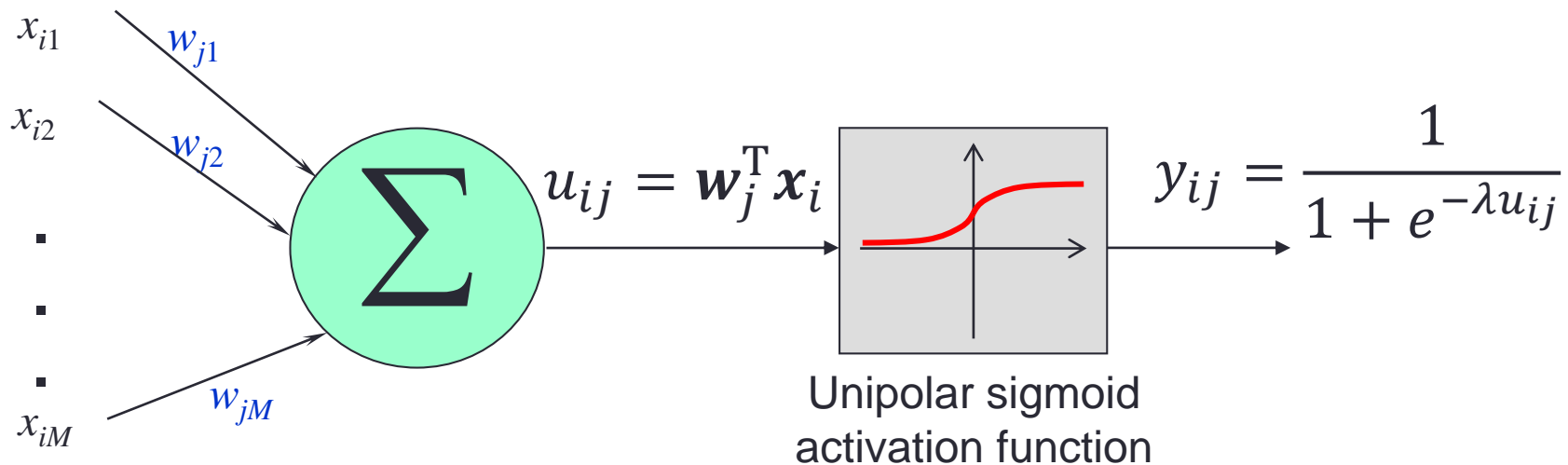




# Unipolar Sigmoid Activation Function

- Nonlinear activation function:  $y = a(u) = \frac{1}{1+e^{-\lambda u}}$
- For a set of training samples  $(\mathbf{x}_i, \mathbf{d}_i)$ , where  $i = 1 \dots Q$

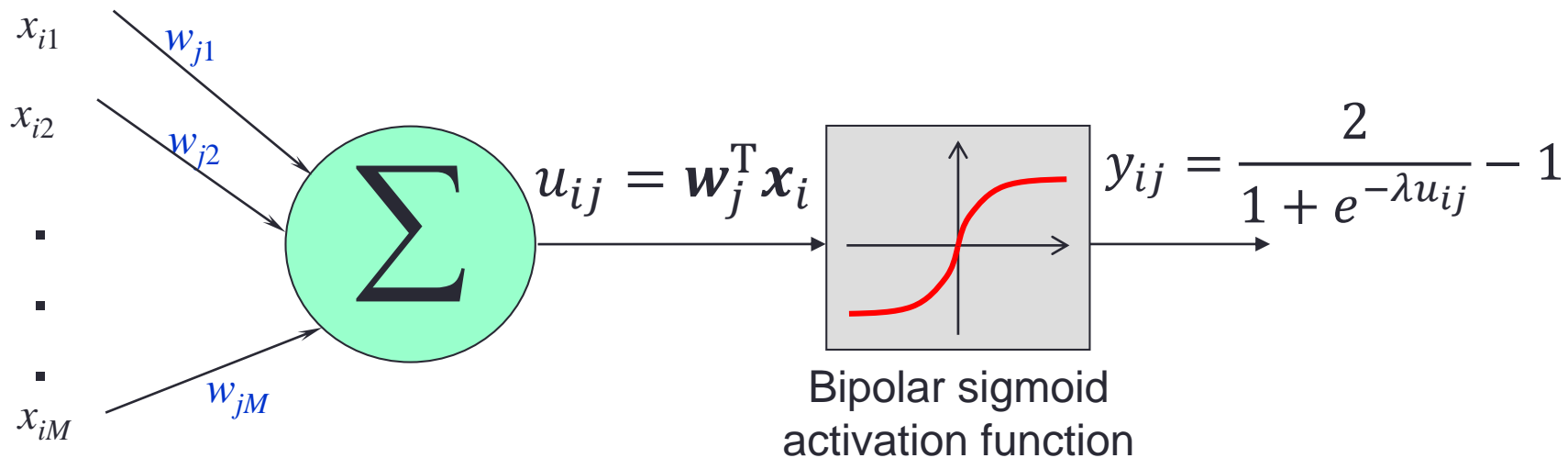
$$y_{ij} = a \left( u_{ij} = \mathbf{w}_j^T \mathbf{x}_i = \sum_{l=1}^M w_{jl} x_{il} \right) = \frac{1}{1 + e^{-\lambda u_{ij}}} = d_{ij}$$



# Bipolar Sigmoid Activation Function

- Nonlinear activation function:  $y = a(u) = \frac{2}{1+e^{-\lambda u}} - 1$
- For a set of training samples  $(\mathbf{x}_i, \mathbf{d}_i)$ , where  $i = 1 \dots Q$

$$y_{ij} = a \left( u_{ij} = \mathbf{w}_j^T \mathbf{x}_i = \sum_{l=1}^M w_{jl} x_{il} \right) = \frac{2}{1 + e^{-\lambda u_{ij}}} - 1 = d_{ij}$$



# Cost Function and Weight Adjustment

- Define misclassification cost function as:

$$E(\mathbf{w}_j) = \frac{1}{2} \sum_{i=1}^Q (d_{ij} - y_{ij})^2 = \frac{1}{2} \sum_{i=1}^Q (d_{ij} - a(u_{ij}))^2 \in \mathbb{R}$$

- Objective of learning – minimizing the cost function
- Strategy – adjust the weights along the gradient of cost function:

$$\Delta \mathbf{w}_j = -\eta \nabla_{\mathbf{w}} E(\mathbf{w}_j)$$

# Gradient of the Cost Function

- The gradient of the cost function

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_j) = \left( \frac{\partial E(\mathbf{w}_j)}{\partial w_{j1}}, \frac{\partial E(\mathbf{w}_j)}{\partial w_{j2}}, \dots, \frac{\partial E(\mathbf{w}_j)}{\partial w_{jM}} \right)^T \in \Re^M$$

- Partial derivative of the cost function against  $w_{jp}$ :

$$\frac{\partial E(\mathbf{w}_j)}{\partial w_{jp}} = - \sum_{i=1}^Q (d_{ij} - a(u_{ij})) \frac{\partial a(u_{ij})}{\partial w_{jp}}$$

$$= - \sum_{i=1}^Q (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial w_{jp}}$$

Depends on  
the  
activation  
function

$$u_{ij} = \mathbf{w}_j^T \mathbf{x}_i = \sum_{l=1}^M w_{jl} x_{il} \Rightarrow \frac{\partial u_{ij}}{\partial w_{jp}} = x_{ip}$$

# $\delta$ Learning Rule

- The gradient of the cost function:

$$\begin{aligned}\nabla_{\mathbf{w}_j} E(\mathbf{w}_j) &= \left( \frac{\partial E(\mathbf{w}_j)}{\partial w_{j1}}, \frac{\partial E(\mathbf{w}_j)}{\partial w_{j2}}, \dots, \frac{\partial E(\mathbf{w}_j)}{\partial w_{jM}} \right)^T \\ &= \left( -\sum_{i=1}^Q (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} x_{i1}, \dots, -\sum_{i=1}^Q (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} x_{iM} \right)^T\end{aligned}$$

- Incremental learning rule:

$$\Delta \mathbf{w}_j = -\eta \nabla_{\mathbf{w}_j} E(\mathbf{w}_j) = \eta (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} \mathbf{x}_i \in \mathbb{R}^M$$

# Partial Derivative of the Activation Function

- Partial derivative of the cost function:

$$\frac{\partial E(\mathbf{w}_j)}{\partial w_{jp}} = - \sum_{i=1}^Q (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} x_{ip}$$

Adaline	Unipolar sigmoid	Bipolar sigmoid
$a(u_{ij}) = u_{ij}$	$y_{ij} = a(u_{ij}) = \frac{1}{1 + e^{-\lambda u_{ij}}}$	$y_{ij} = a(u_{ij}) = \frac{2}{1 + e^{-\lambda u_{ij}}} - 1$
$\frac{\partial a(u_{ij})}{\partial u_{ij}} = 1$	$\frac{\partial a(u_{ij})}{\partial u_{ij}} = \lambda y_{ij} (1 - y_{ij})$	$\frac{\partial a(u_{ij})}{\partial u_{ij}} = 2\lambda y_{ij} (1 - y_{ij})$

# Incremental $\delta$ Learning Rule

- Unipolar sigmoid:

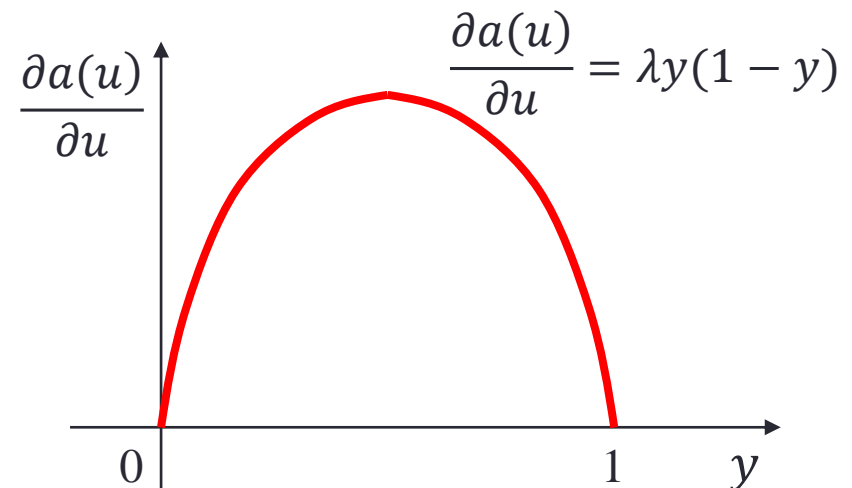
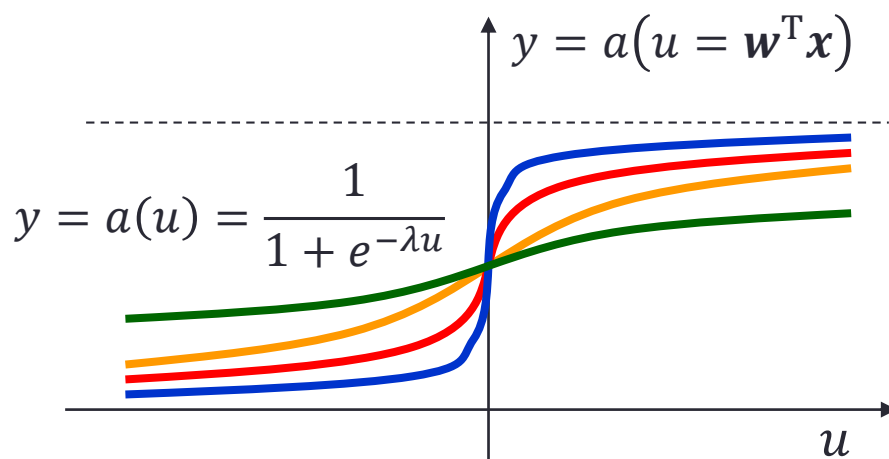
$$\Delta \mathbf{w}_j = \eta (d_{ij} - y_{ij}) \lambda y_{ij} (1 - y_{ij}) \mathbf{x}_i$$

- Bipolar sigmoid:

$$\Delta \mathbf{w}_j = 2\eta (d_{ij} - y_{ij}) \lambda y_{ij} (1 - y_{ij}) \mathbf{x}_i$$

# Saturation of Sigmoid

- The  $\lambda$  in the sigmoid function determines how fast the  $y$  saturates to the two extremes
- The initial training weight  $\mathbf{w}_0$  must close to zero (why?)
- Hint: 1. Learning rule:  $\Delta \mathbf{w}_j = \eta (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} \mathbf{x}_i$   
2. Large  $\mathbf{w} \Rightarrow y \sim 1 \Rightarrow \frac{\partial a(u)}{\partial u} \sim 0$



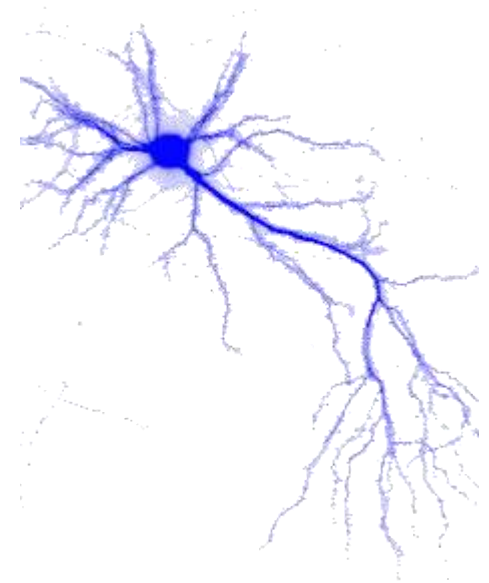


# Artificial Neural Network

---

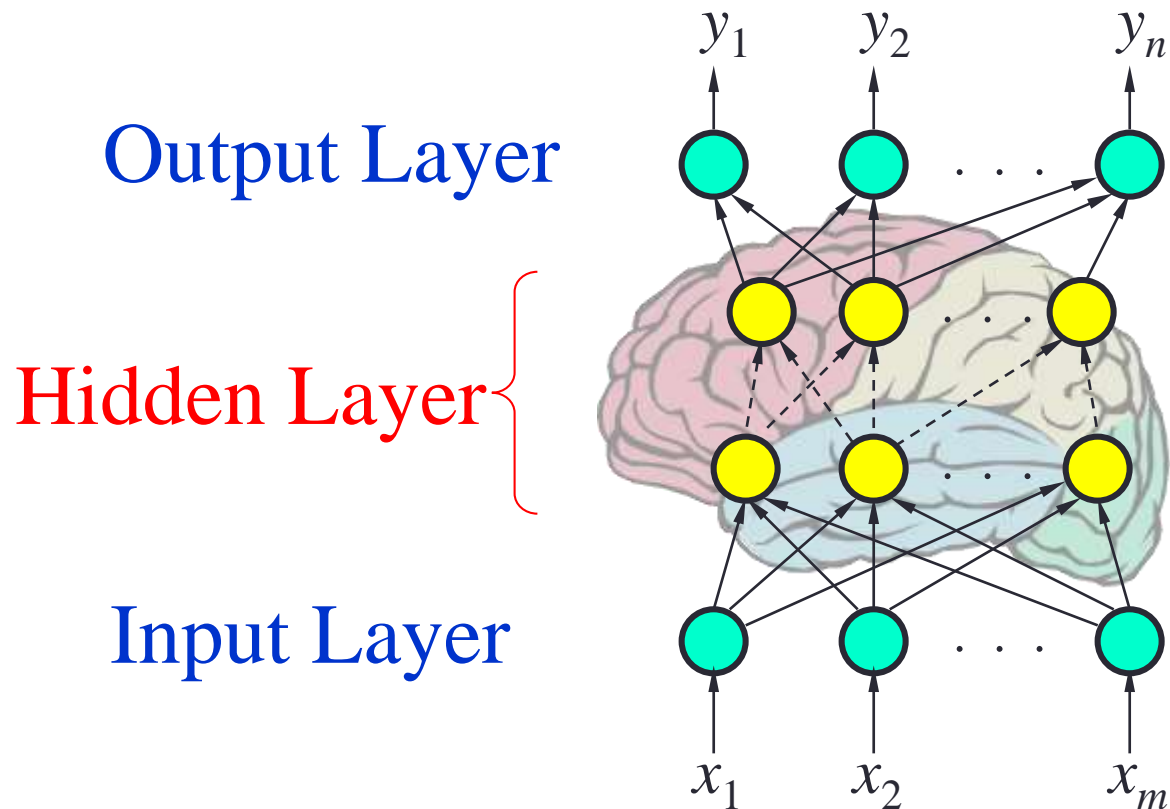
## Multilayer perceptron

- Examples
- Back propagation learning algorithm

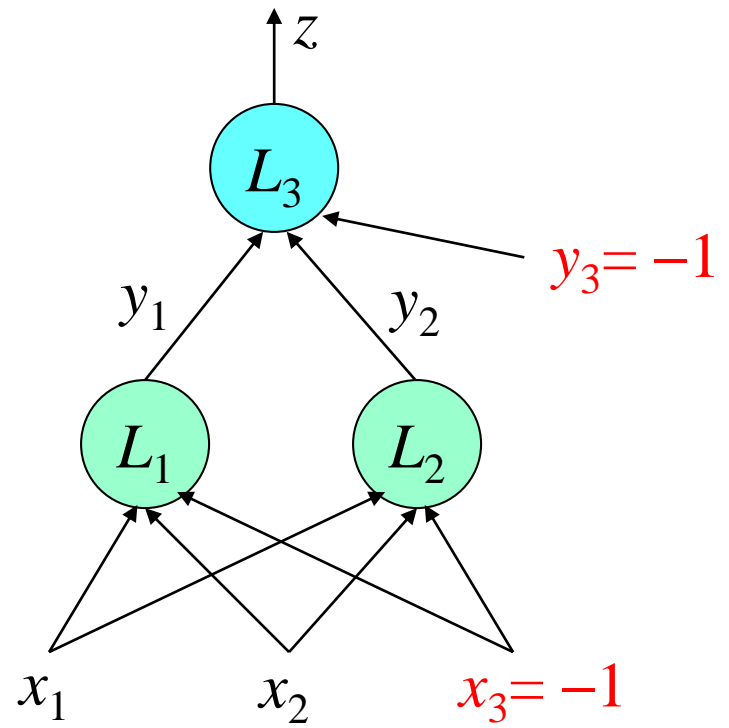
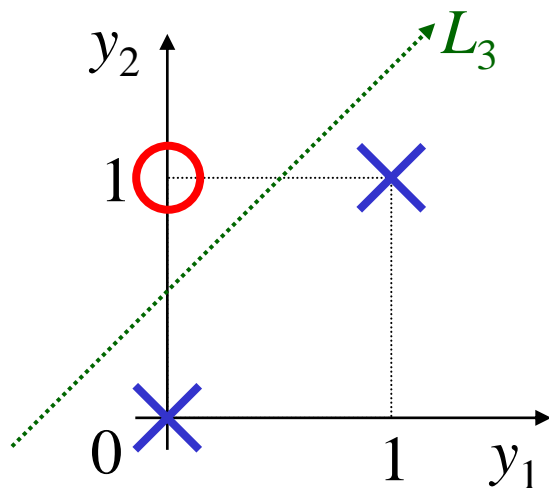
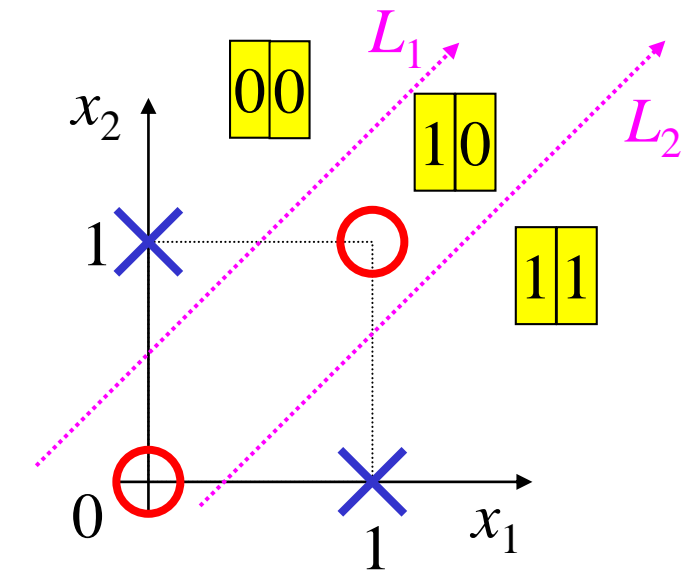


# Multilayer Perceptron

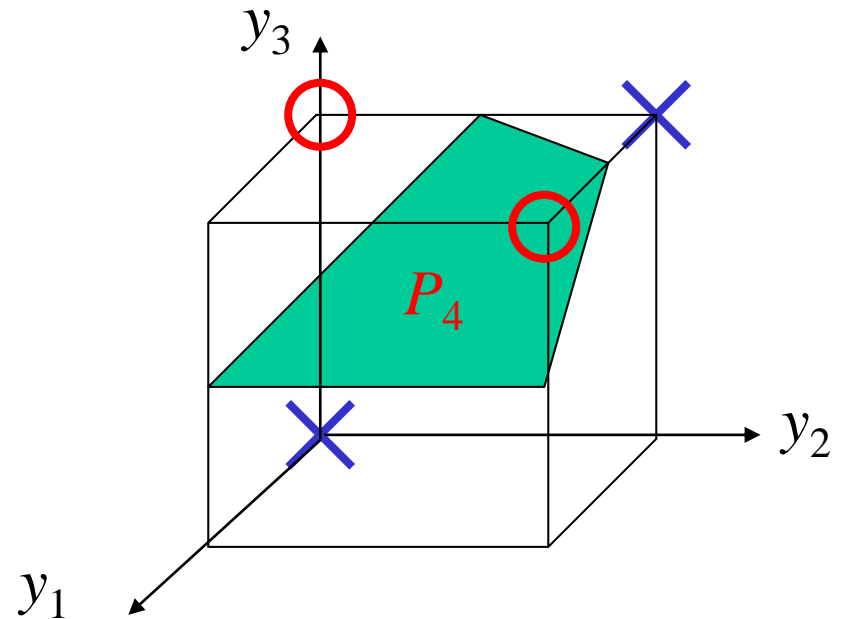
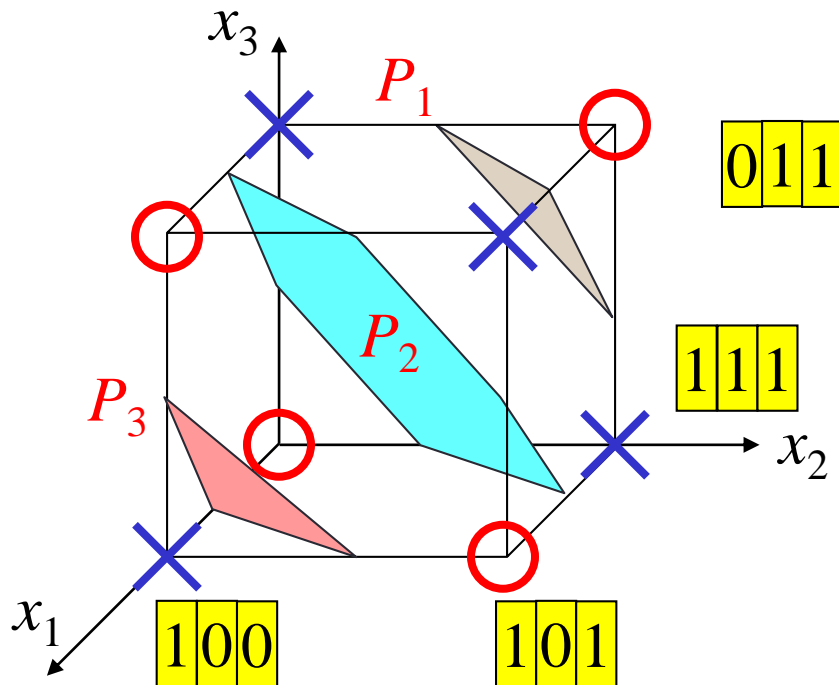
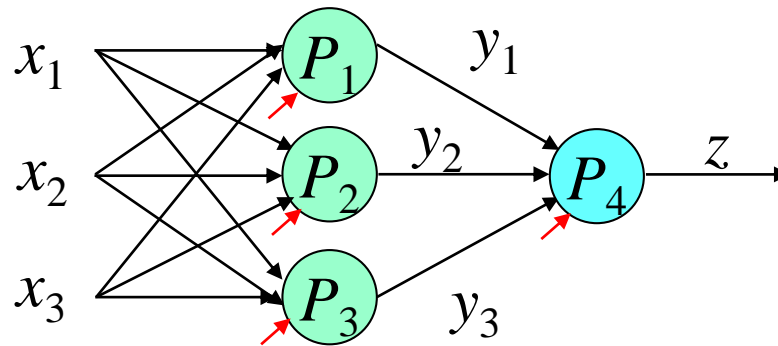
- Multilayer perceptron can handle problems that are not linearly separable



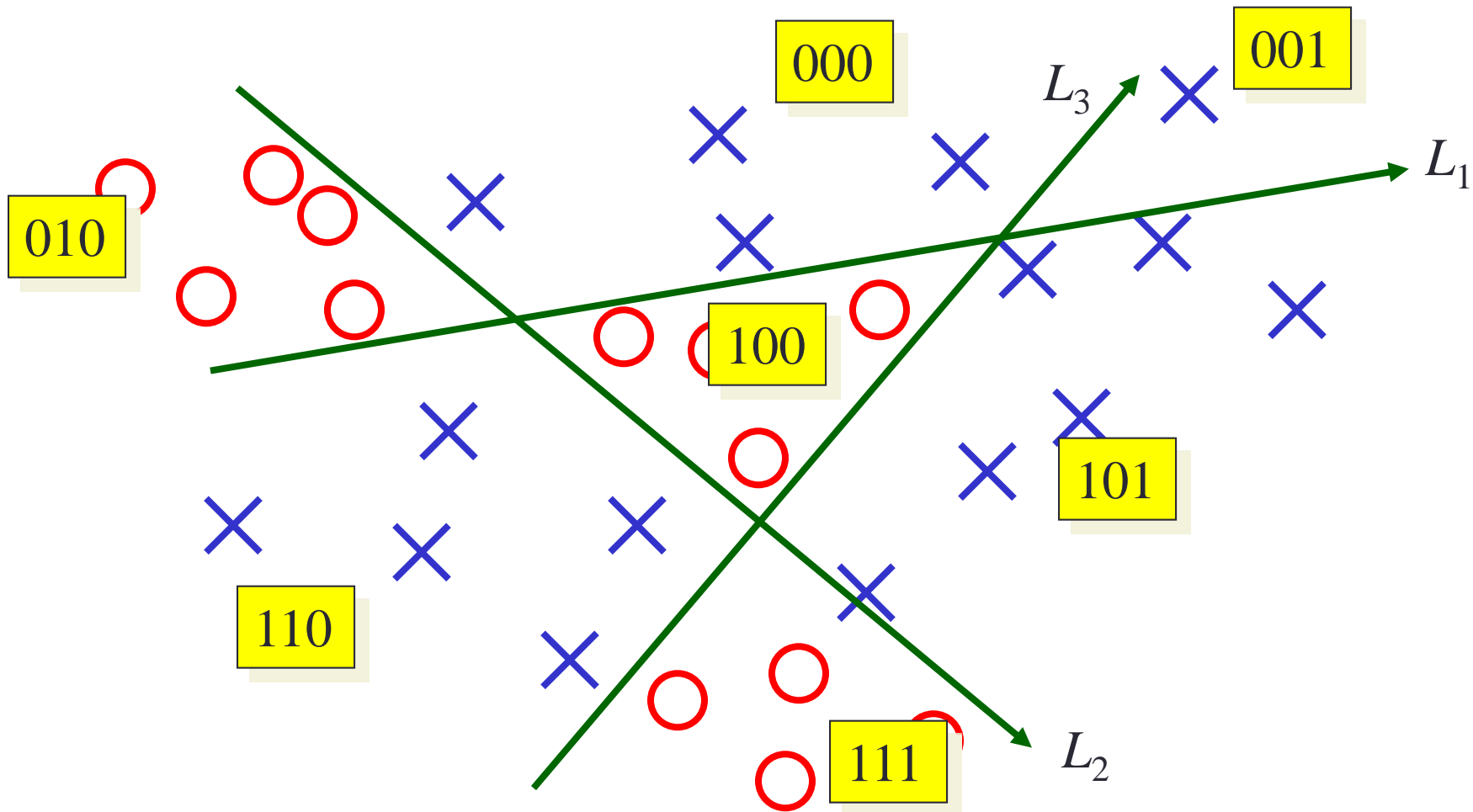
# Example: XOR Problem



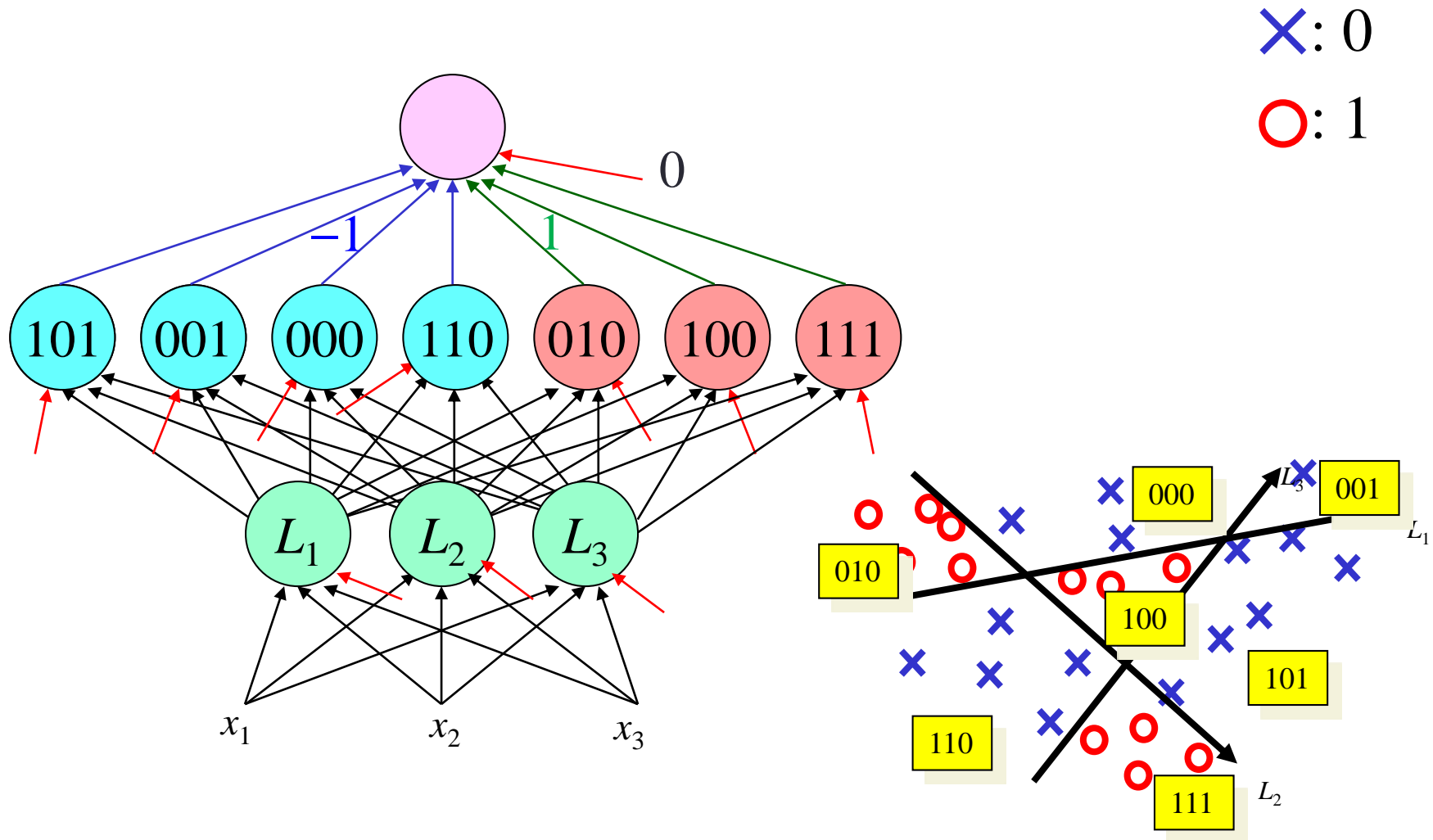
# Another Example: Parity Problem



# Another Example: Partition



# Another Example: Partition (Cont'd)

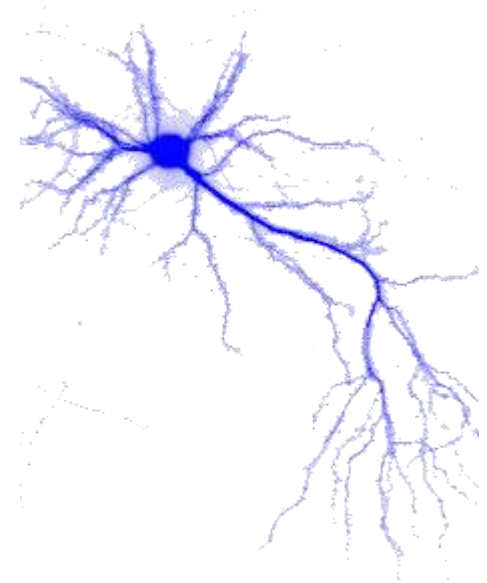


# Artificial Neural Network

---

## Multilayer perceptron

- Examples
- Back propagation learning algorithm

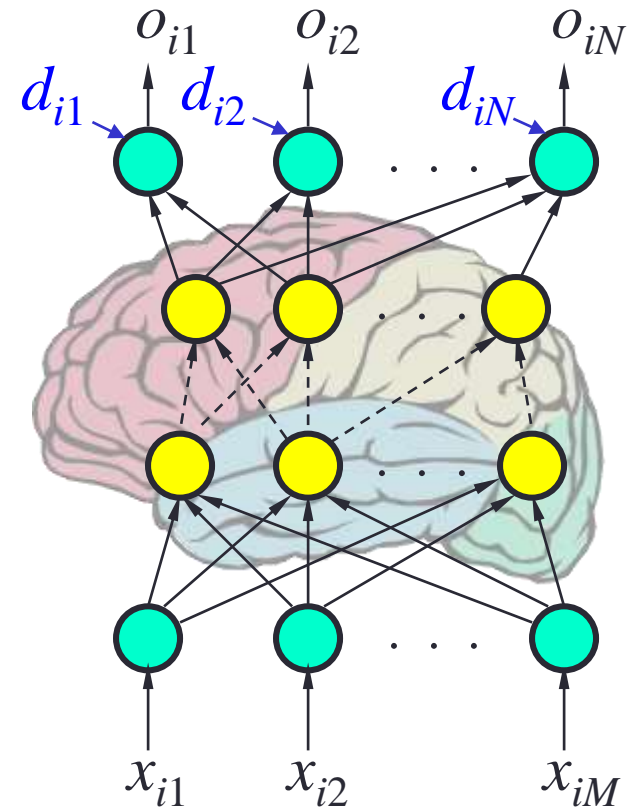


# Supervised Learning

- Given a set of training  $\{(\mathbf{x}_i, \mathbf{d}_i), i = 1 \dots Q\}$
- Define sum of squared error  $E$

$$E = \sum_{i=1}^Q E_i = \sum_{i=1}^Q \left[ \frac{1}{2} \sum_{j=1}^N (d_{ij} - o_{ij})^2 \right]$$

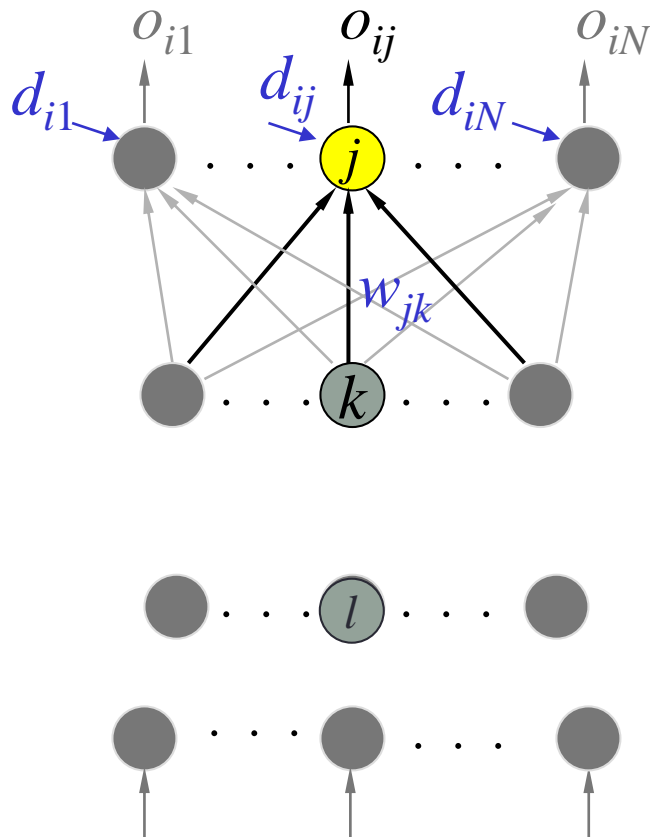
- Objective is to obtain a set of weights that minimize  $E$  for both the output and hidden neurons





# Back Propagation

- Update the weights backward

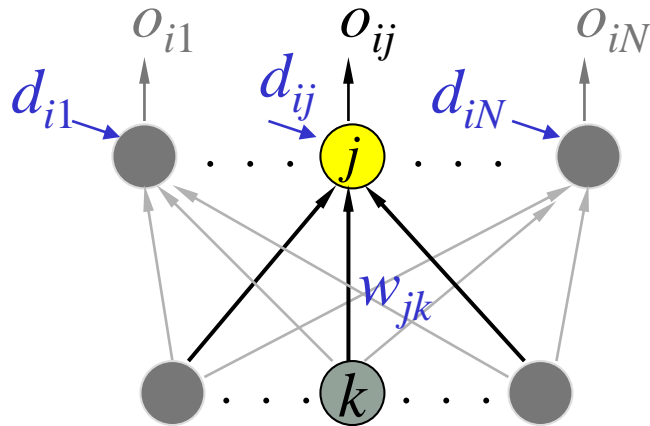


## Notes:

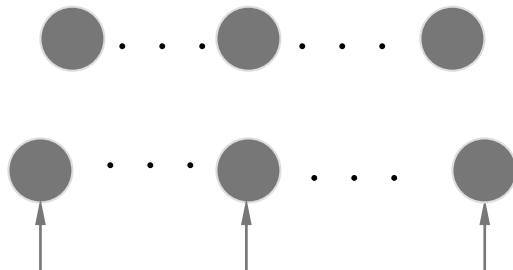
- $d_{ij}$ : actual outputs
- $o_{ij}$ : outputs of layer  $j$
- $o_{ik}$ : outputs of layer  $k$
- $o_{il}$ : outputs of layer  $l$
- $w_{jk}$ : weights connecting layers  $j$  and  $k$
- $w_{kl}$ : weights connecting layers  $k$  and  $l$

# Learning on Output Neurons

It is known:  $o_{ij} = a(u_{ij})$ ,  $E_i = \frac{1}{2} \sum_{j=1}^N (d_{ij} - o_{ij})^2$ ,  $u_{ij} = \sum w_{jk} o_{ik}$



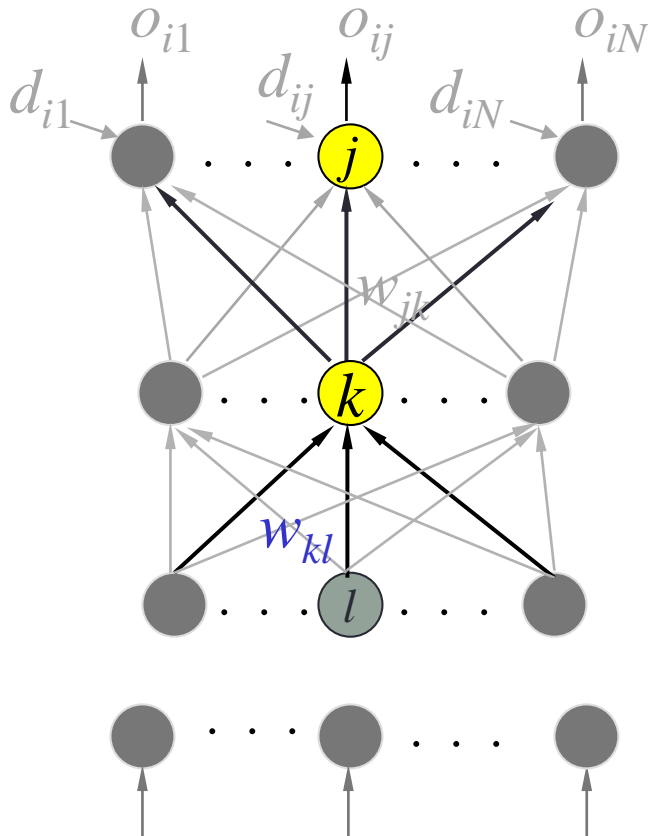
$$\begin{aligned} \frac{\partial E}{\partial w_{jk}} &= \sum_{i=1}^Q \frac{\partial E_i}{\partial w_{jk}} = \sum_{i=1}^Q \frac{\partial E_i}{\partial o_{ij}} \frac{\partial o_{ij}}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial w_{jk}} \\ &= \sum_{i=1}^Q \underbrace{-(d_{ij} - o_{ij}) \cdot \lambda o_{ij} (1 - o_{ij})}_{\delta_{ij}} \cdot o_{ik} \end{aligned}$$



$$\Rightarrow \Delta w_{jk} = -\eta \sum_{i=1}^Q \delta_{ij} o_{ik}$$

# Learning on Hidden Neurons

It is known:  $o_{ik} = a(u_{ik})$ ,  $E_i = \frac{1}{2} \sum_{j=1}^N (d_{ij} - o_{ij})^2$ ,  $u_{ik} = \sum w_{kl} o_{il}$



$$\begin{aligned}
 \frac{\partial E}{\partial w_{kl}} &= \sum_{i=1}^Q \frac{\partial E_i}{\partial w_{kl}} = \sum_{i=1}^Q \frac{\partial E_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial u_{ik}} \frac{\partial u_{ik}}{\partial w_{kl}} \\
 &= \sum_{i=1}^Q \sum_j \frac{\partial E_i}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial u_{ik}} \frac{\partial u_{ik}}{\partial w_{kl}} \\
 &= \sum_{i=1}^Q \sum_j \underbrace{\delta_{ij} \cdot w_{jk} \cdot \lambda o_{ik} (1 - o_{ik}) \cdot o_{il}}_{\delta_{ik}} \\
 &\Rightarrow \Delta w_{kl} = -\eta \sum_{i=1}^Q \delta_{ik} o_{il}
 \end{aligned}$$

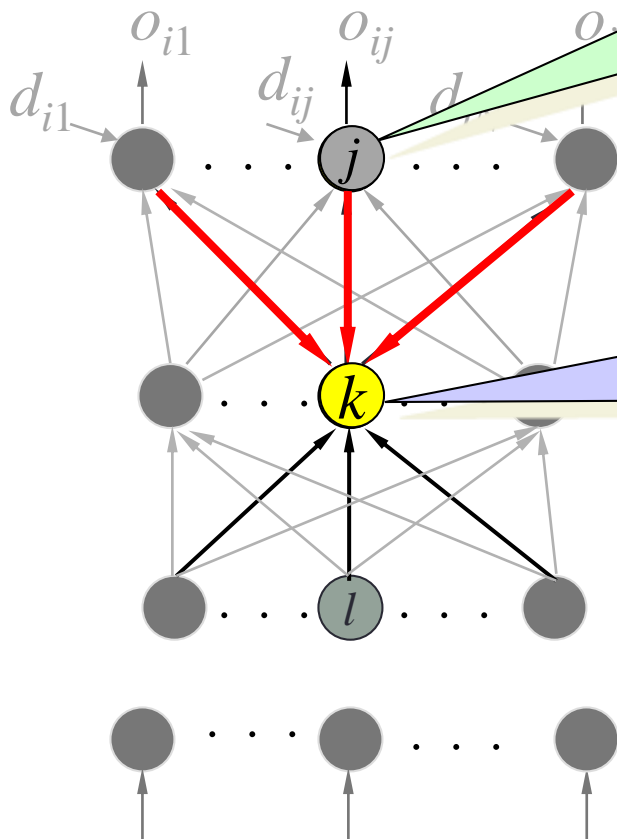
# Back Propagation

$$\delta_{ij} = \frac{\partial E_i}{\partial u_{ij}} = -(d_{ij} - o_{ij}) \cdot \lambda o_{ij}(1 - o_{ij})$$

$$\Delta w_{jk} = -\eta \sum_{i=1}^q \delta_{ij} o_{ik}$$

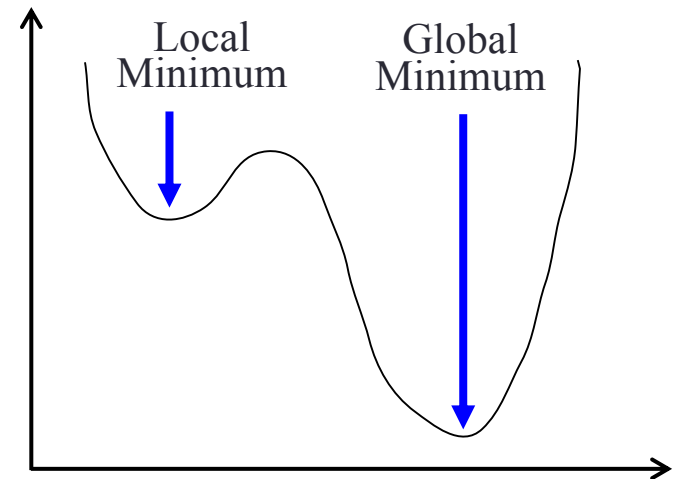
$$\delta_{ik} = \frac{\partial E_i}{\partial u_{ik}} = \sum_j \delta_{ij} \cdot w_{jk} \cdot \lambda o_{ik}(1 - o_{ik}) \cdot o_{il}$$

$$\Delta w_{kl} = -\eta \sum_{i=1}^q \delta_{ik} o_{il}$$



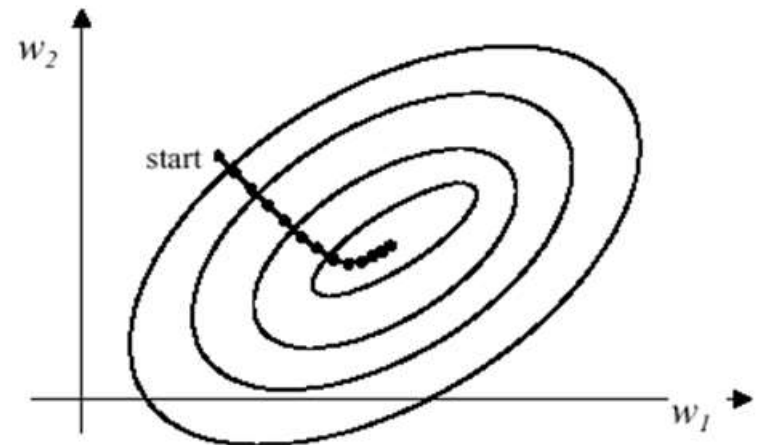
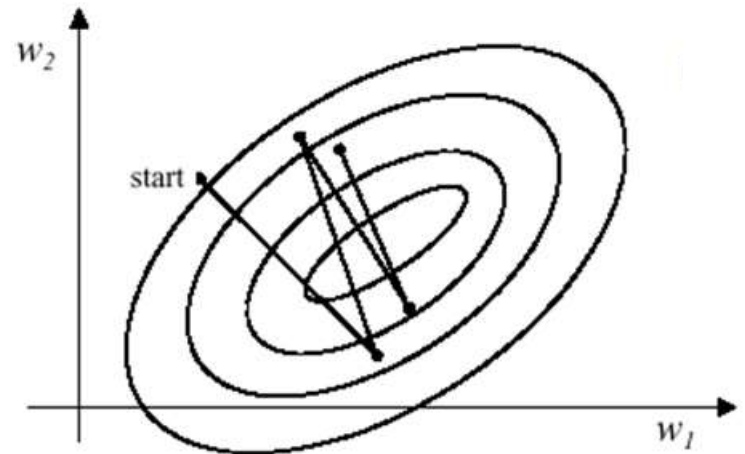
# Back Propagation Using Gradient Descent

- Advantages
  - Relatively simple implementation
  - Generally works well
- Disadvantages
  - Slow and inefficient
  - Can get stuck in local minima resulting in sub-optimal solutions
- Alternative
  - Simulated annealing
  - Genetic algorithms
  - Simplex algorithm



# Learning Parameters

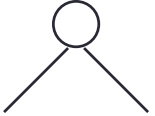
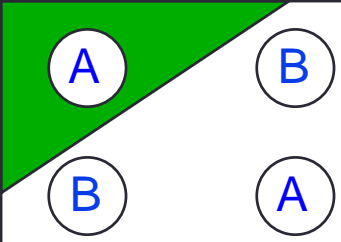
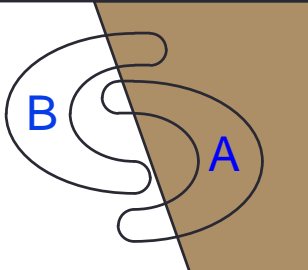

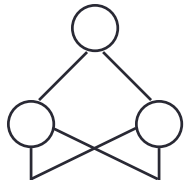
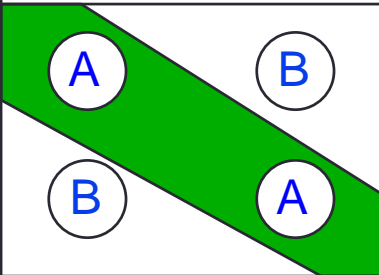
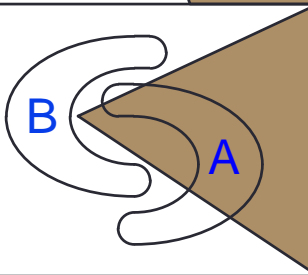
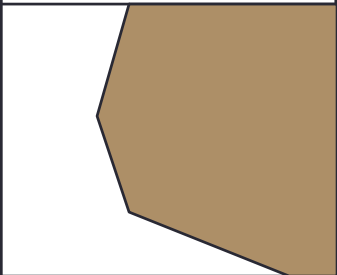
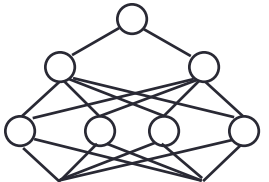
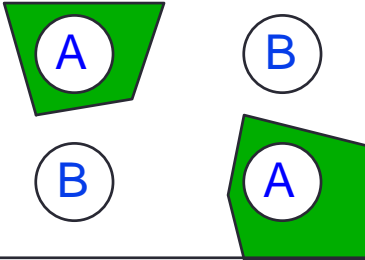
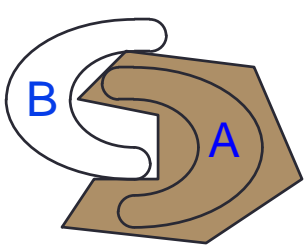
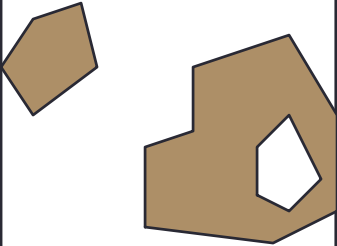
- Weight update rules
- Initial weight
- Learning rate  $\eta$
- Number of nodes
- Number of hidden layers
- Stopping criteria



# Number of Hidden Layers

- Multilayer feedforward networks with one hidden layer using arbitrary squashing functions are capable of approximating any function to any desired degree of accuracy, provided sufficiently many hidden units are available
  - G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function," Mathematics of Control, Signals, and Systems (1989)
  - K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," Neural Networks, 2:359-366 (1989)

# Rule of Thumb for Hidden Layers

Structure	Types of Decision Regions	Exclusive-OR Problem	Class Separation	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

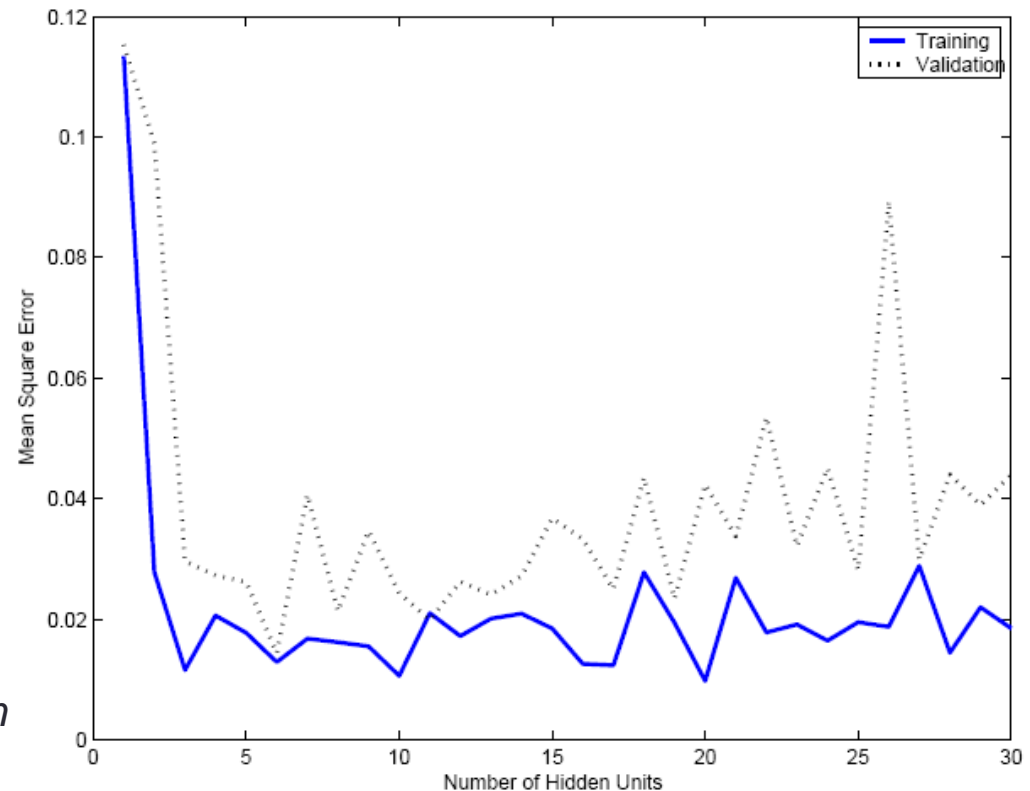


# Number of Hidden Layer Neurons

- Generally a trade-off between under-fitting and over-fitting
- Data-driven ways to determine the number of hidden layers:

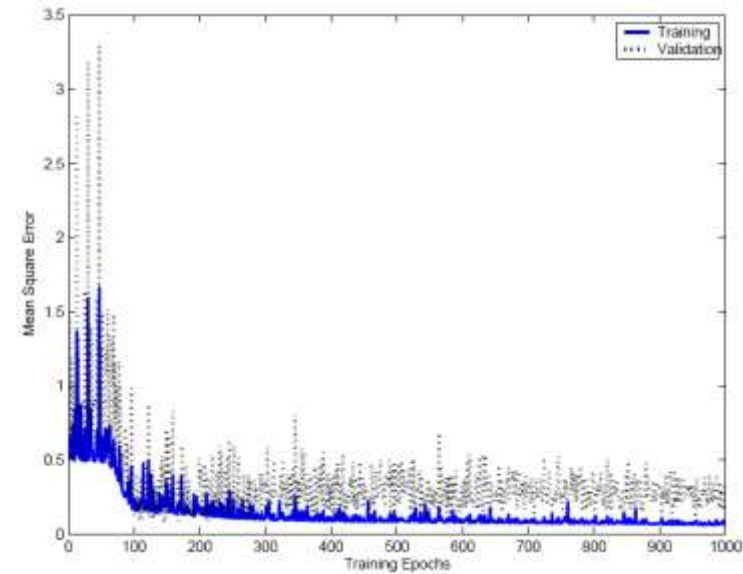
1. Hold out part of the sample
2. Cross-validation
3. Bootstrapping

Alpaydin, *Introduction to machine learning*



# Stopping Criteria

- Total mean squared error change:
  - Learning is considered to have converged when the absolute rate of change in the average squared error per iteration is sufficiently small
- Generalization based criterion:
  - After each iteration the ANN is tested for generalization using a different test sample set
  - Stop if the generalization performance is adequate



Alpaydin, *Introduction to machine learning*

# Autonomous Land Vehicle In a Neural Network

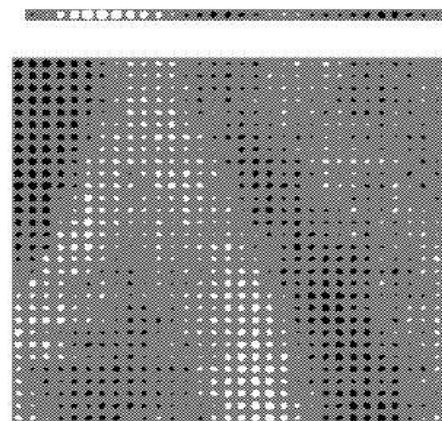
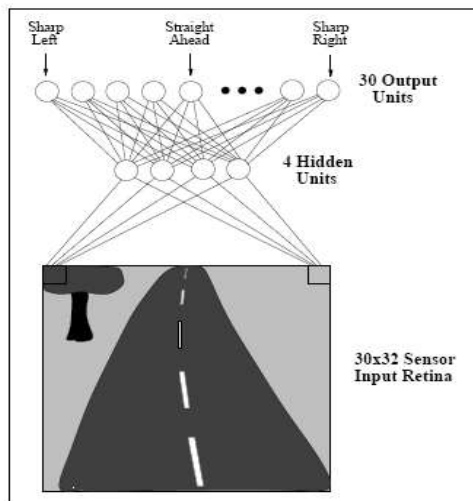
- Drives 70 mph on a public highway



30 outputs  
for steering

4 hidden  
units

30x32 pixels  
as inputs



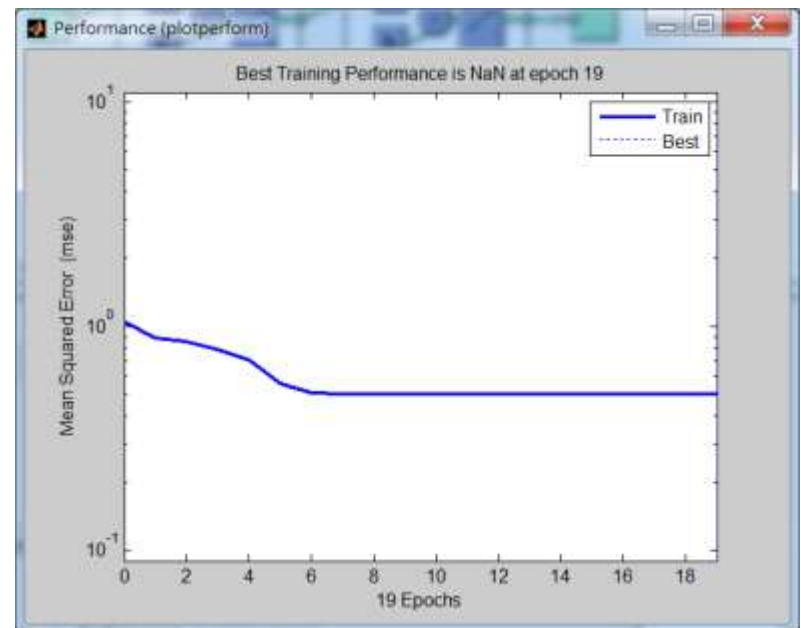
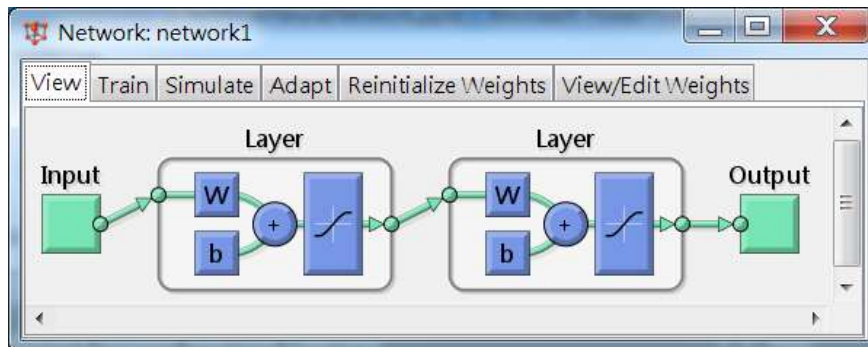
30x32 weights  
into one out of  
four hidden unit

# XOR Problem

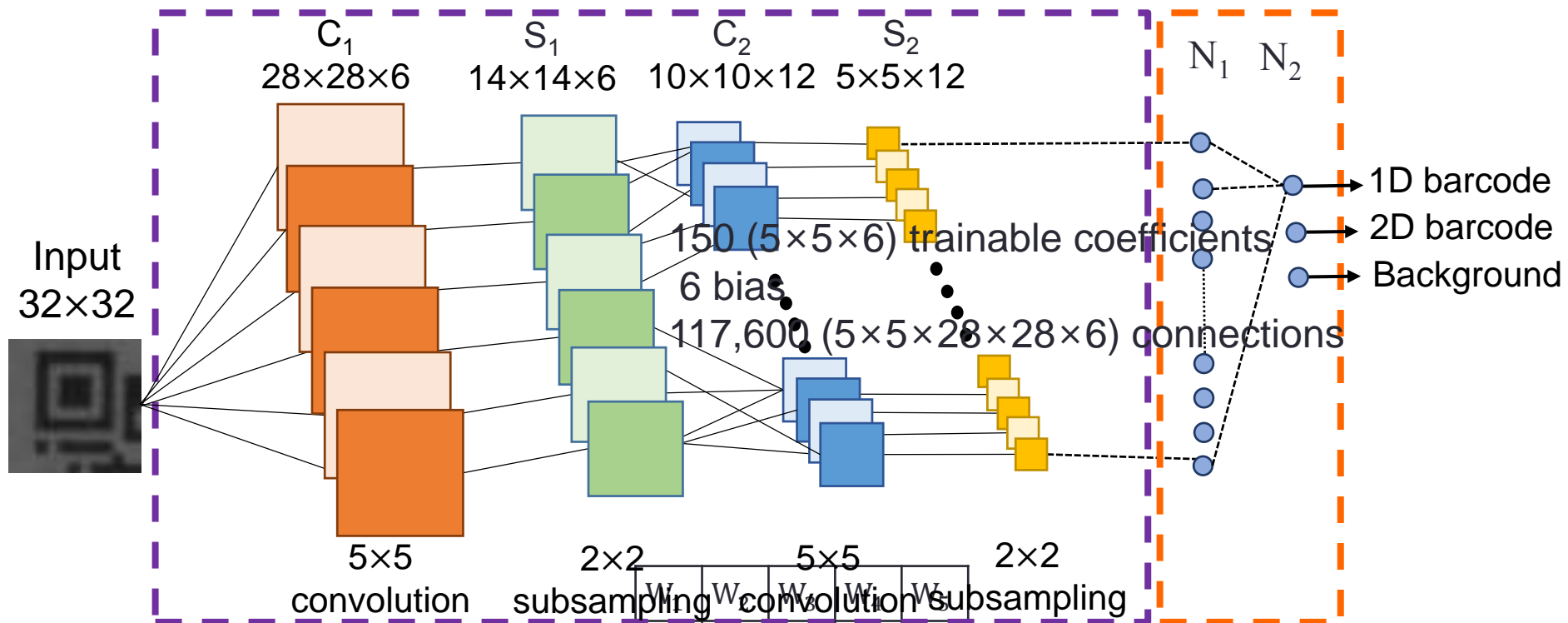
- XOR:

Input $x$	Output $y$
(+1, +1)	-1
(+1, -1)	+1
(-1, +1)	+1
(-1, -1)	-1

- Matlab ANN tool: nntool



# Convolutional Neural Network



**Feature extraction**

54	21	50	79	$W_7$	$W_8$	$W_9$	$W_{10}$
10	90	89	59	$W_{11}$	$W_{12}$	$W_{13}$	$W_{14}$
19	85	54	41	$W_{15}$	$W_{16}$	$W_{17}$	$W_{18}$
06	94	10	91	$W_{19}$	$W_{20}$	$W_{21}$	$W_{22}$
				$W_{23}$	$W_{24}$	$W_{25}$	

Layer  $C_1$  : 150 ( $5 \times 5 \times 6$ ) trainable coefficients, 6 bias, sigmoid function

Layer  $C_2$  : 300 ( $5 \times 5 \times 12$ ) trainable coefficients, 12 bias, sigmoid function

Layer  $N_2$  : 900 ( $5 \times 5 \times 12 \times 3$ ) trainable coefficients, 3 bias

Total of 1371 trainable parameter

# Reading Assignments

- S. Zhong and V. Cherkassky, "Factors Controlling Generalization Ability of MLP Networks," In Proc. IEEE Int. Joint Conf. on Neural Networks, vol. 1, pp. 625-630, Washington DC. July 1999.
- D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. I, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group. MIT Press, Cambridge, 1986.  
([http://psych.stanford.edu/~jlm/papers/PDP/Volume%201/Chap8\\_PD\\_P86.pdf](http://psych.stanford.edu/~jlm/papers/PDP/Volume%201/Chap8_PD_P86.pdf)).
- C. Bishop, *Neural Networks for Pattern Recognition*

# Acknowledgement

- Especially thank Dr. Tai-Wen Yue for sharing their valuable teaching material in this course