INTRODUCTORY APPLIED MACHINE LEARNING

Yan-Fu Kuo

Dept. of Bio-industrial Mechatronics Engineering

National Taiwan University

Today:

- Artificial neural network

Outline

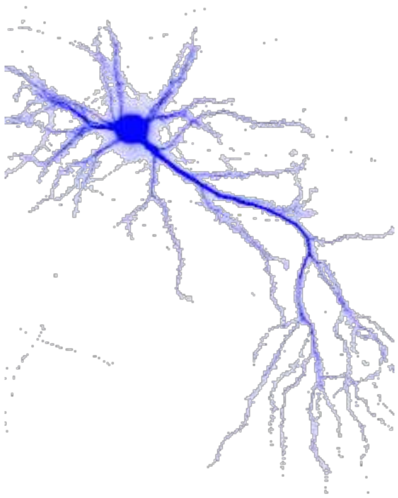
- Goals
- Introduction
- Single-layer perceptron networks
- Learning rules for single-layer perceptron networks
  - Perceptron learning rule
  - Adaline leaning rule
  - δ-leaning rule
- Multilayer perceptron
  - Back propagation learning algorithm

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

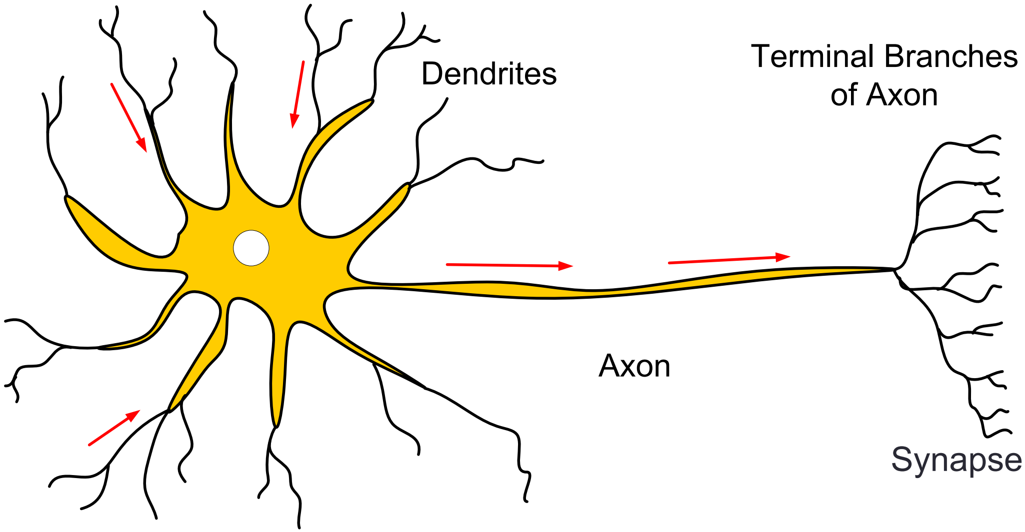
Introduction



Historical Background

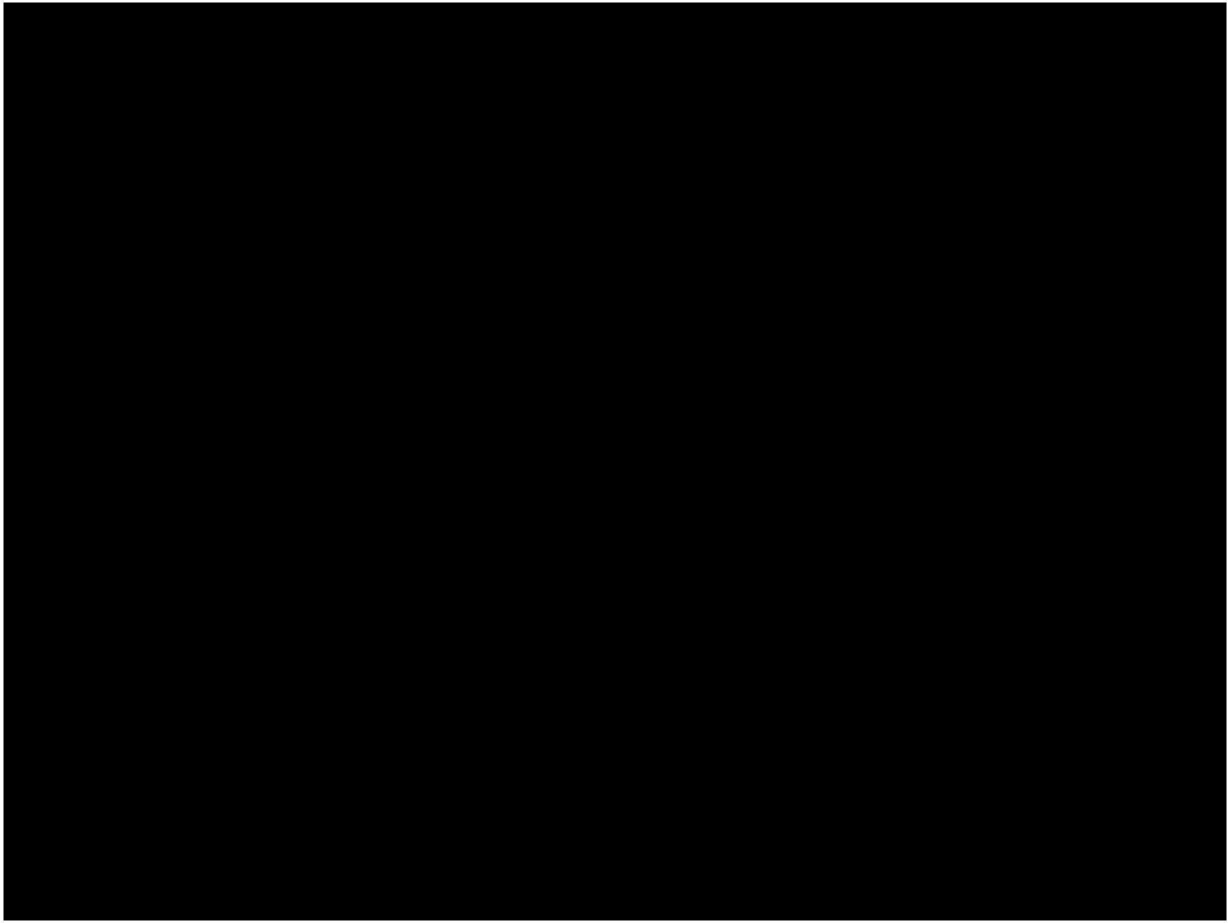
- 1943 McCulloch and Pitts proposed the first computational models of neuron
- 1949 Hebb proposed the first learning rule
- 1958 Rosenblatt introduced the simple single layer networks now called “perceptrons”
- 1969 Minsky and Papert’s exposed limitation of the theory
- 1986 The back-propagation learning 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 ~ 10^{11} neurons, each of which is connected ~ 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 aulaclty uesdnatnrd waht I was rdanieg. The phaonmneal pweor of the hmuan mnid, aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer 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.

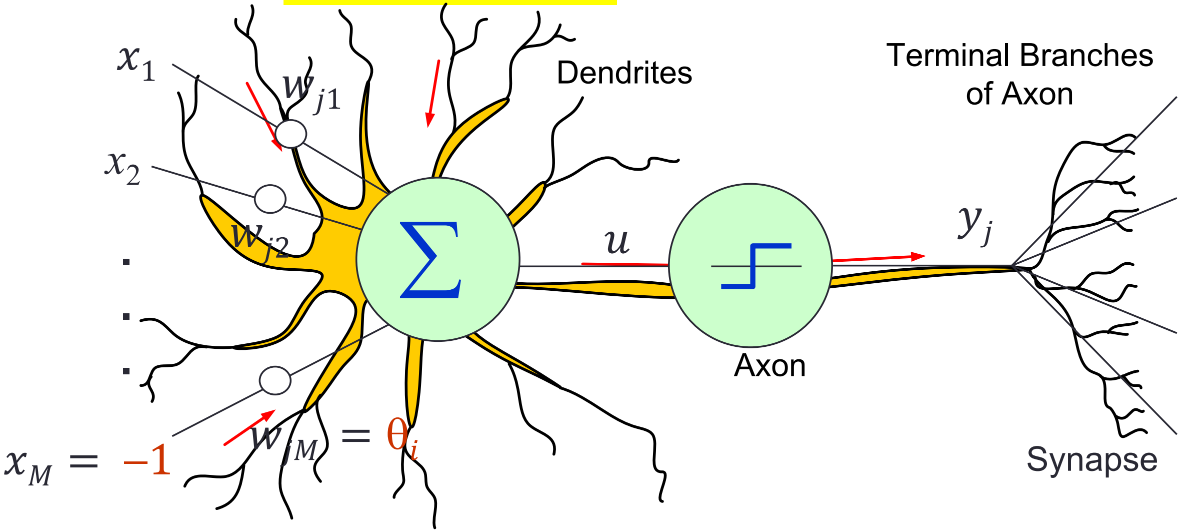
Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Amzanig huh? yaeh and I awlyas tghuhot slpeling was ipmorantt!

The McCulloch-Pitts Neuron

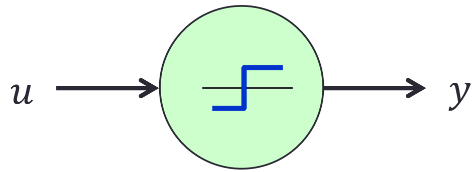
- Also known as a threshold logic unit
- A neuron j works like:

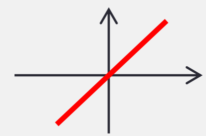
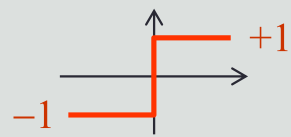
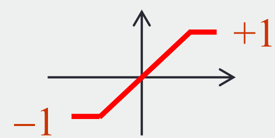
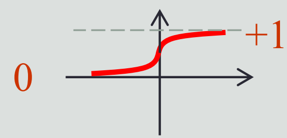
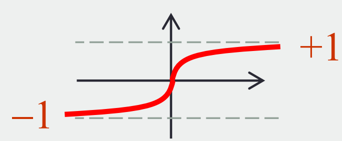
$$u = \sum_{l=1}^{M} w_{jl}\, x_l$$

$$y_j = a(u)$$



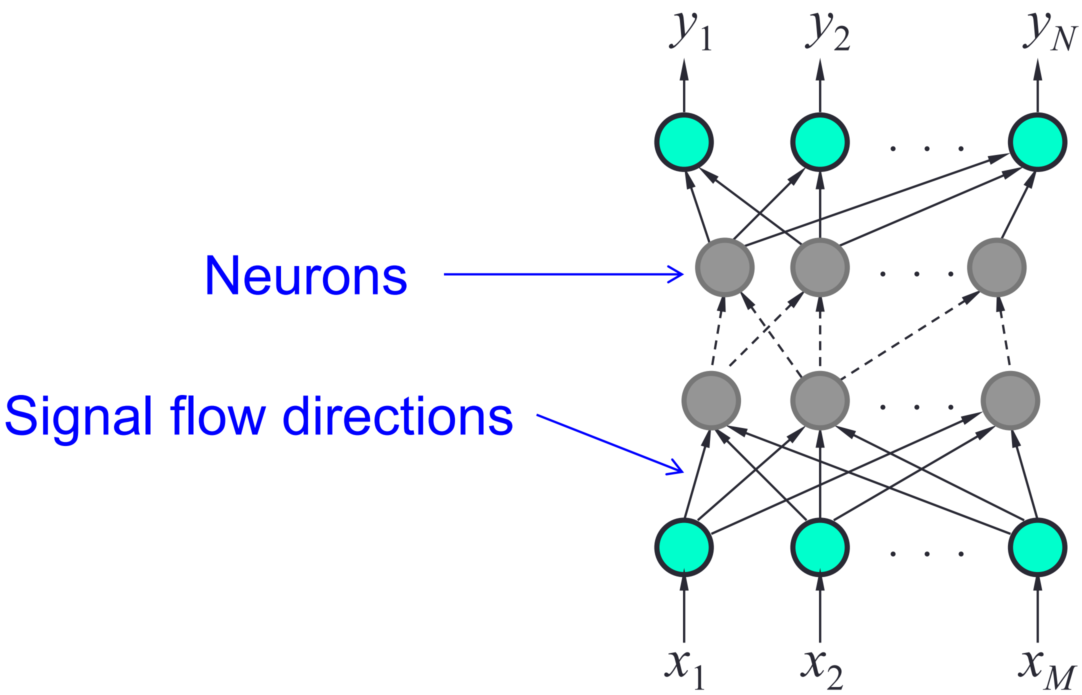
Typical Activation Function



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)

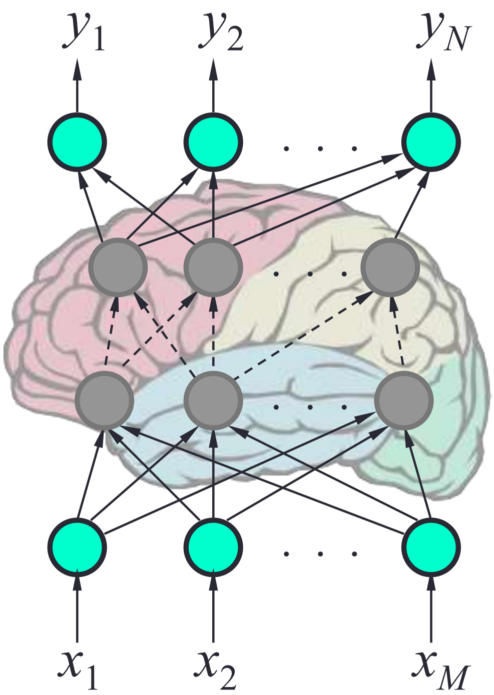


Layered Structure

Output Layer

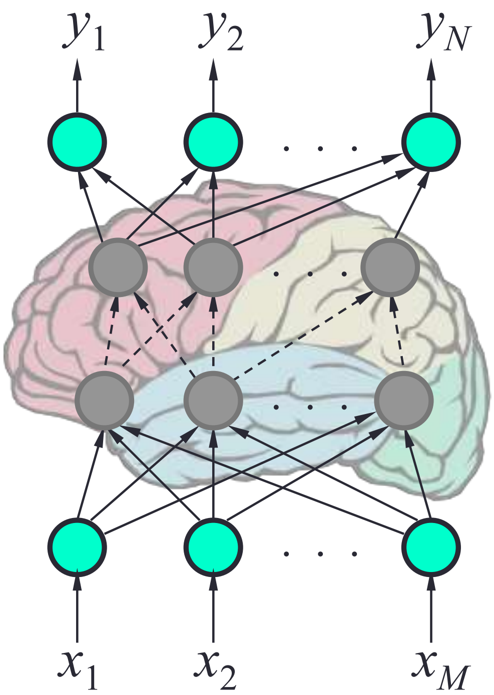
Hidden Layer(s)

Input Layer



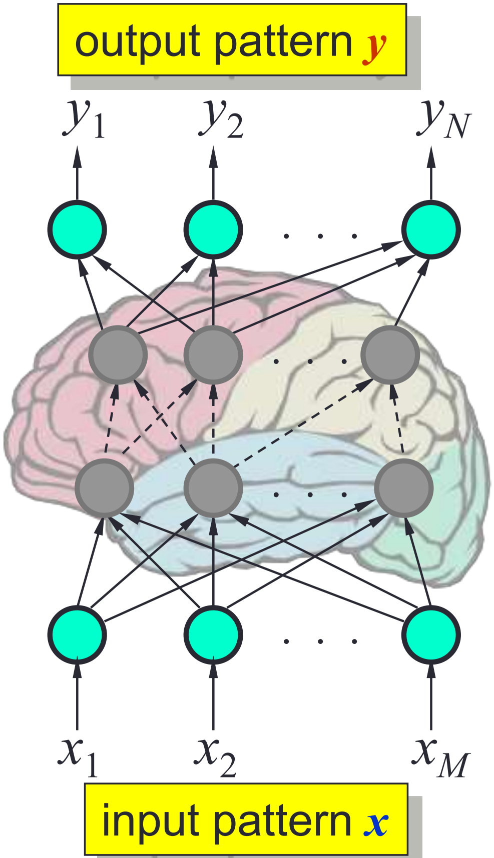
Knowledge and Memory

- 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: $\boldsymbol{x} \rightarrow \boldsymbol{y}$
- 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 then just a linear pattern classification



Training

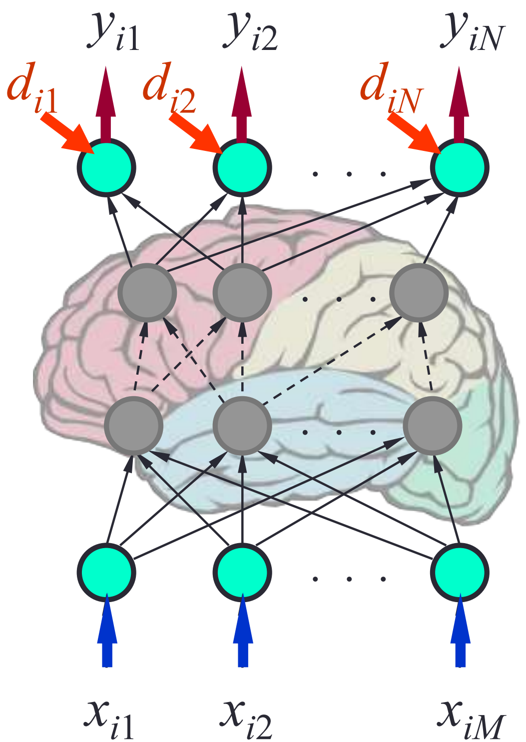
- Given a set of training samples $(\boldsymbol{x}_i, \boldsymbol{d}_i)$, where $i = 1 \ldots Q$

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

- The objective of training is to find a set of weights $\boldsymbol{w}$ that minimize the error, i.e.,

$$\boldsymbol{w} = \underset{\boldsymbol{w}}{\arg\min} \|\boldsymbol{y}_i - \boldsymbol{d}_i\|^2 ,$$

where $\boldsymbol{y}_i = (y_{i1}, \ldots, y_{iN})$

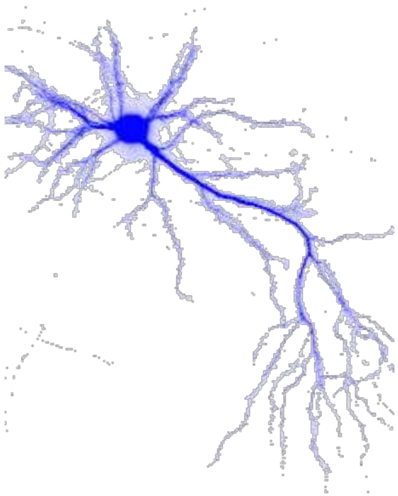


Artificial Neural Network

Single-layer perceptron networks

Learning rules

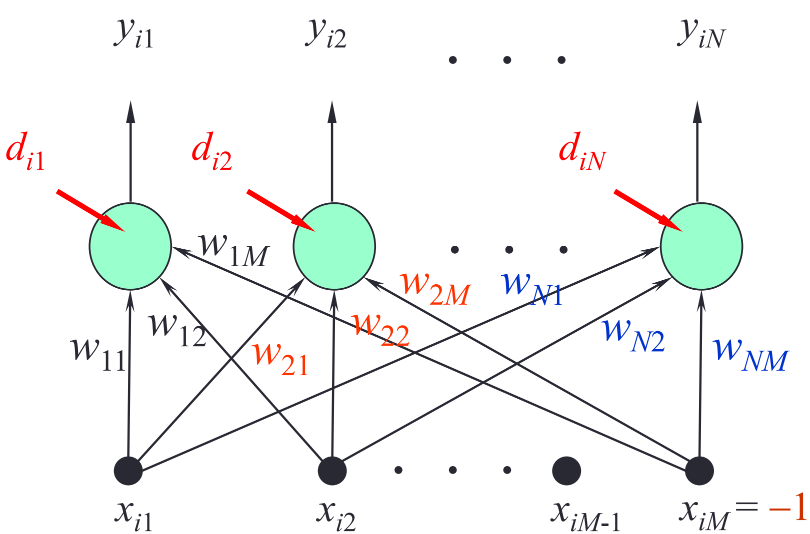
- Perceptron learning rule
- Adaline learning rule
- δ-leaning rule



Training a Single-layered Perceptron

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

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



Note:

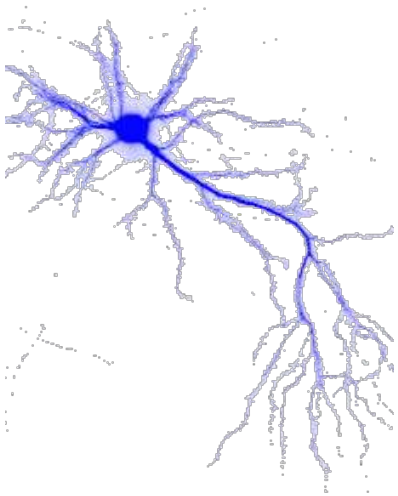
$$\begin{cases} \boldsymbol{x}_i = (x_{i1}, \ldots, x_{iM}) \\ \boldsymbol{d}_i = (d_{i1}, \ldots, d_{iN}) \end{cases}$$

Artificial Neural Network

Single-layer perceptron networks

Learning rules

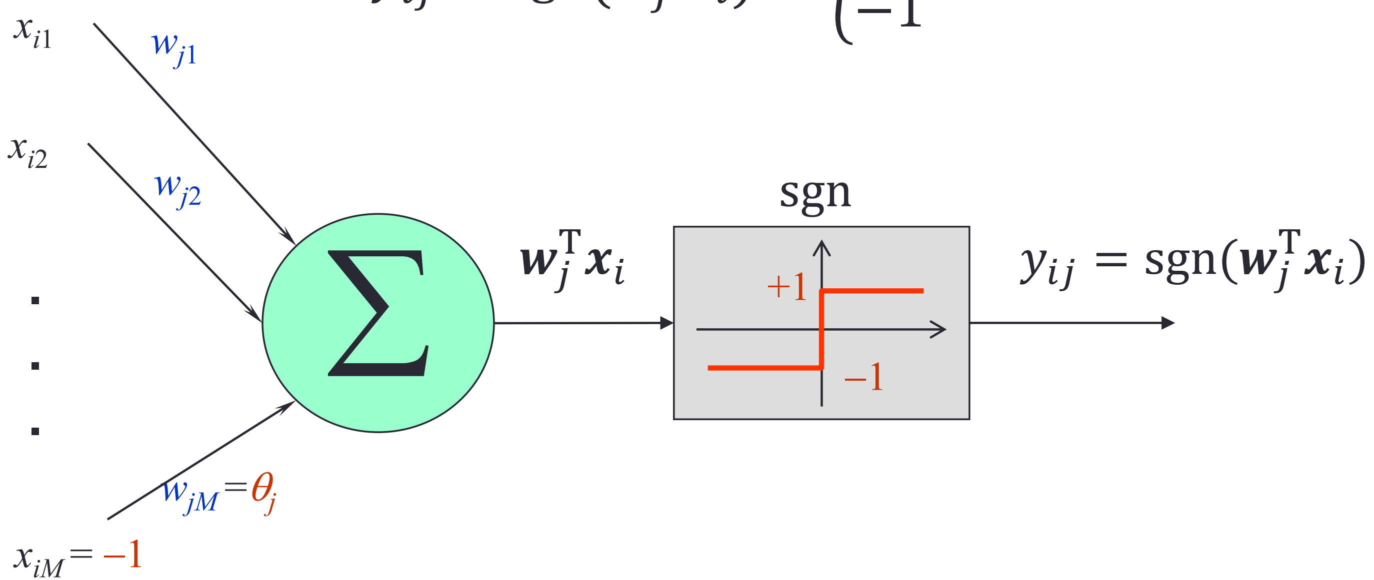
- Perceptron learning rule
- Adaline learning rule
- δ-leaning rule



Perceptron

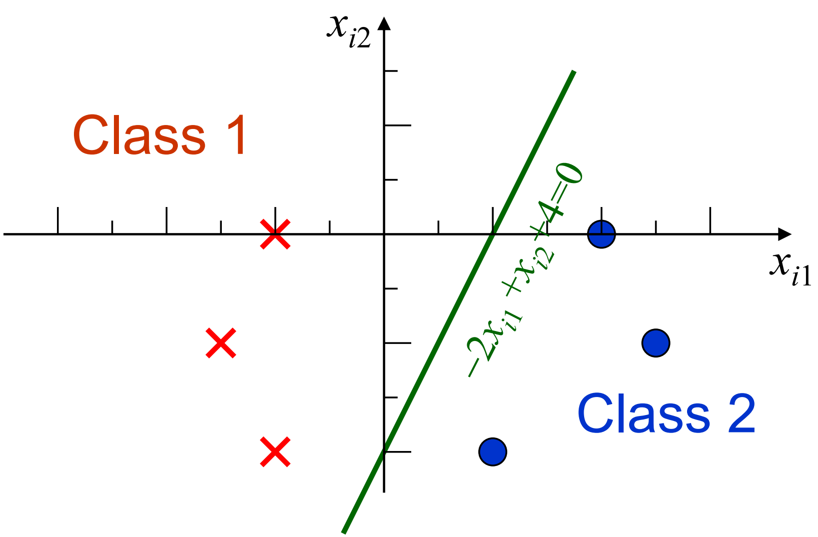
- Train an ANN for "classification"
- Hard limit threshold activation unit

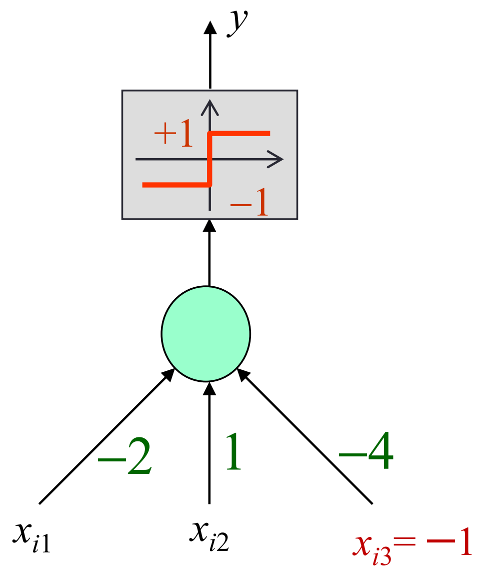
$$y_{ij} = \mathrm{sgn}\left(\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{x}_i\right) = \begin{cases} +1 \\ -1 \end{cases}$$



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: $y = \text{sgn}(-2x_{i1} + x_{i2} + 4)$





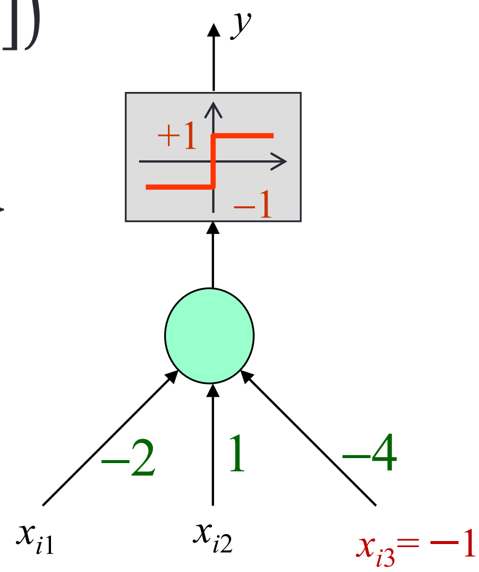
Augmented Input Vector

- Vector $\boldsymbol{\alpha}$ of any dimension can be augmented to $[\boldsymbol{\alpha} \ -1]$
- Class 1 (+1): $\{ [-1,0]^{\mathrm{T}}, [-1.5,-1]^{\mathrm{T}}, [-1,-2]^{\mathrm{T}} \}$

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

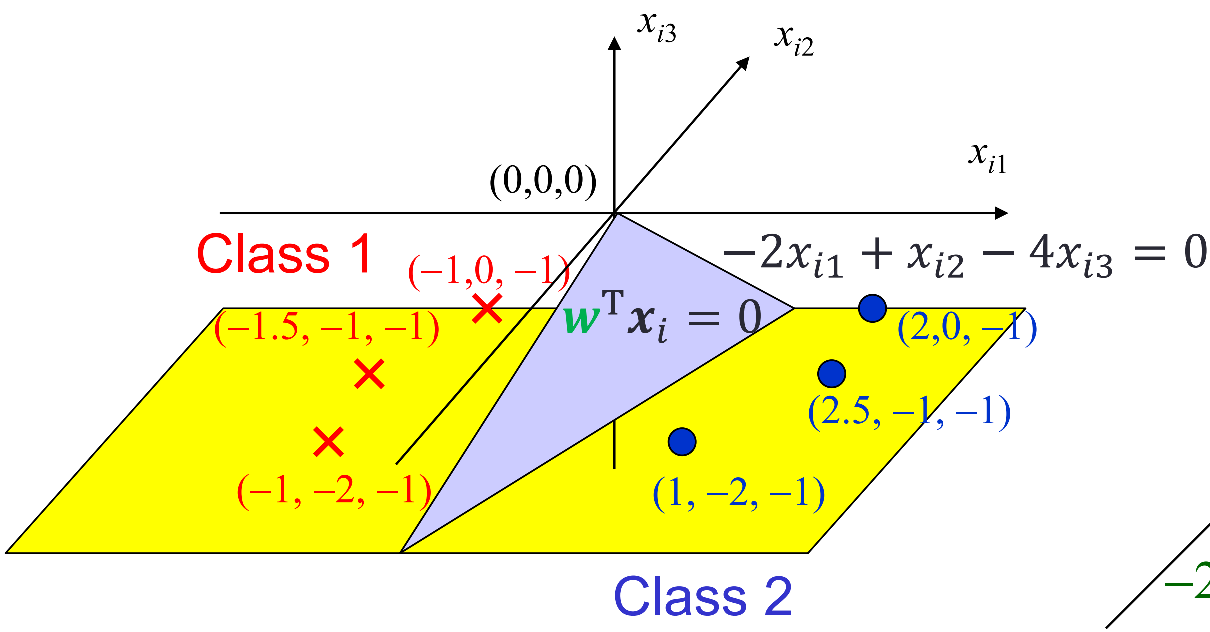
- Class 2 (-1): $\{ [2,0]^{\mathrm{T}}, [2.5,-1]^{\mathrm{T}}, [1,-2]^{\mathrm{T}} \}$

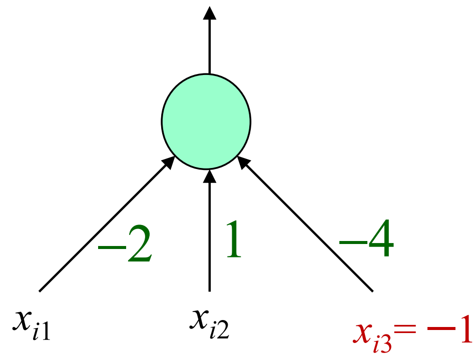
$$\Rightarrow \text{Class 2 } (-1): \left\{ \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 2.5 \\ -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix} \right\}$$



Augmented Input Vector (Cont'd)

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





Classification Problem Formulation

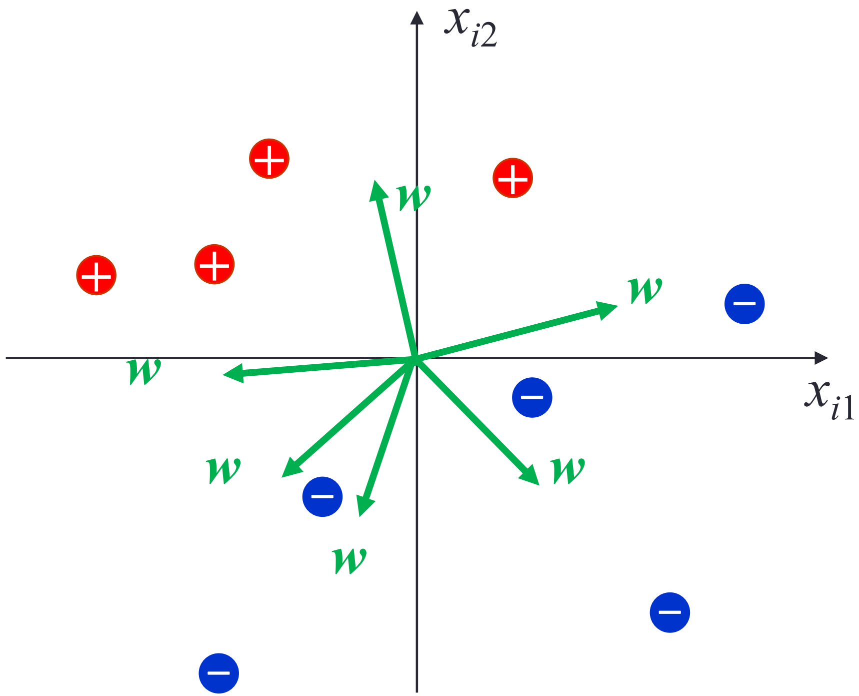
- Given training sets ($\boldsymbol{x} \in \Re^M$)
  - $T_1 = \{\boldsymbol{x}: \boldsymbol{x} \in d = +1 \}$
  - $T_2 = \{\boldsymbol{x}: \boldsymbol{x} \in d = +1 \}$
- Assume T_1 and T_2 are linearly separable
- For a single perceptron classifier, find $\boldsymbol{w} = (w_1, \ldots, w_M)^{\mathrm{T}}$ such that

$$y = \operatorname{sgn}\left(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}\right) = \begin{cases} +1, \boldsymbol{x} \in T_1 \\ -1, \boldsymbol{x} \in T_2 \end{cases}$$

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

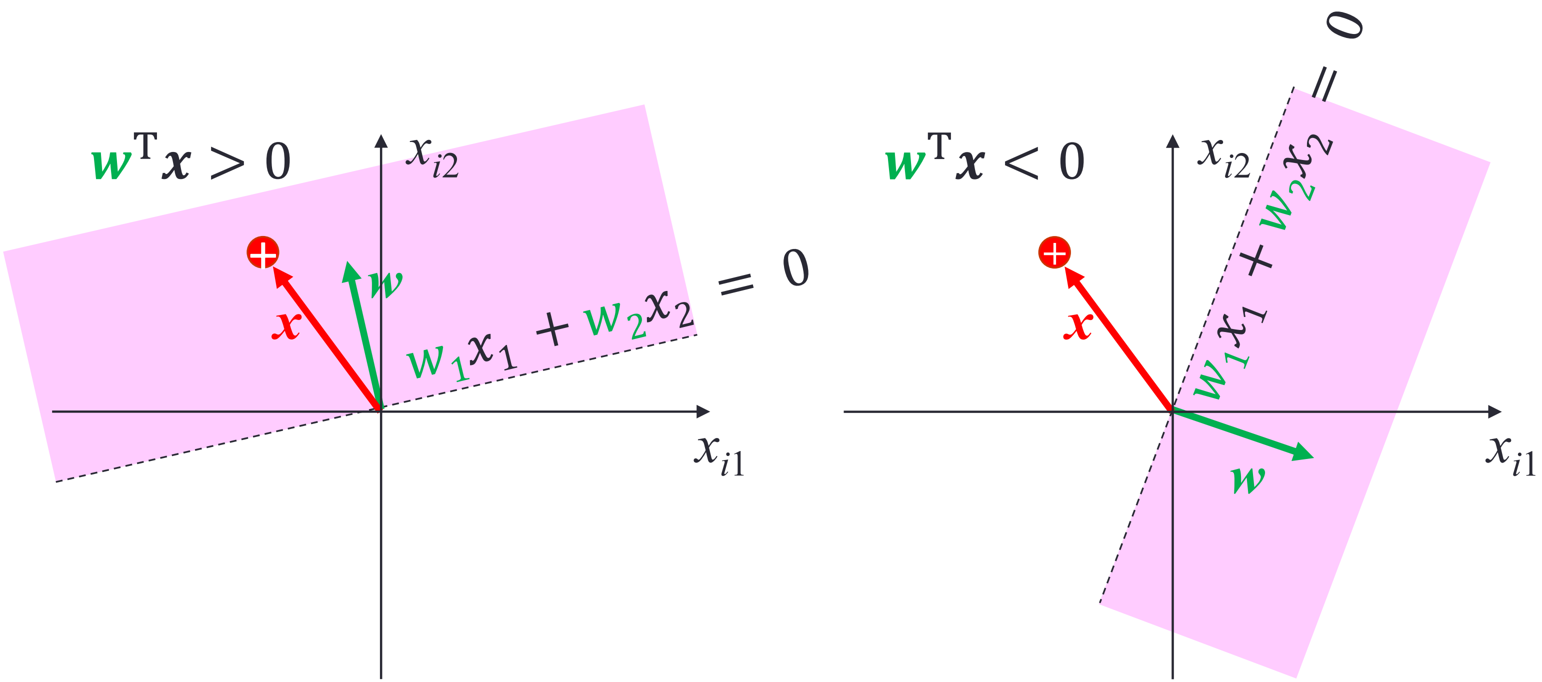
Objective – Finding Appropriate Weights

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



Inner Product between Weights and Inputs

- Classifier: $y = \mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$
- Objective: $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0$ for $\boldsymbol{x}$ ⊕



Learning Strategy

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

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

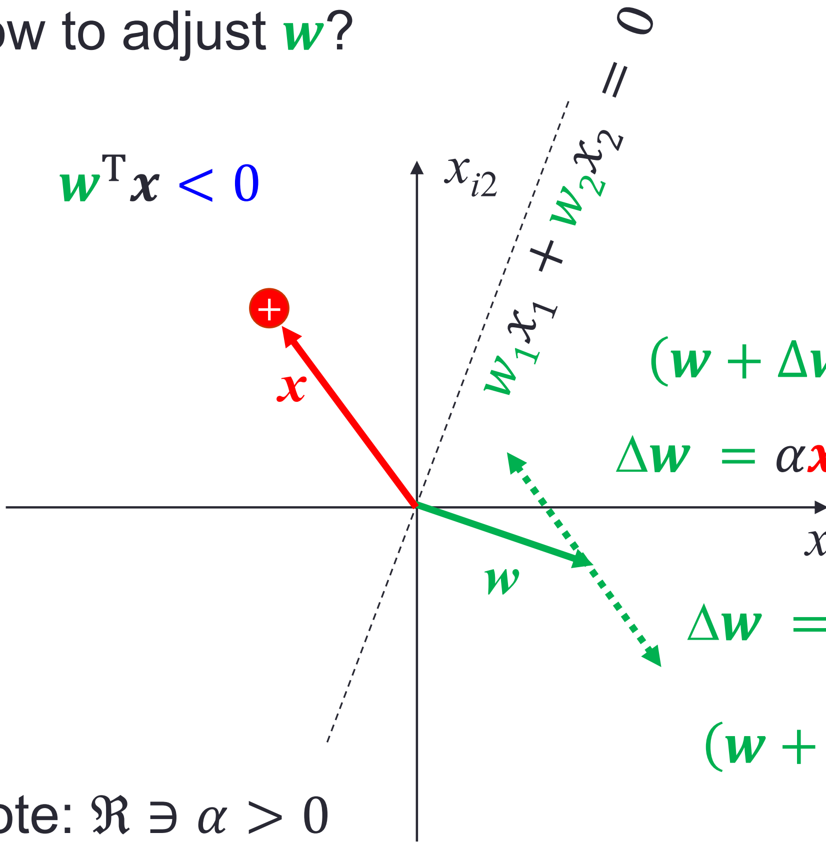
- In each iteration, a single sample is introduced, and the weight is adjust to minimize the error

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

- Keep what have been previously learned in the weights

How to Determine $\Delta \boldsymbol{w}_0$?

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



$\Delta \boldsymbol{w} = \alpha \boldsymbol{x}$

$\Delta \boldsymbol{w} = -\alpha \boldsymbol{x}$

Note: $\Re \ni \alpha > 0$

- Objective:

  $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0$ for $\boldsymbol{x}$ +

$$(\boldsymbol{w} + \Delta \boldsymbol{w})^{\mathrm{T}}\boldsymbol{x} = \underbrace{\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}_{<0} + \underbrace{\alpha \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}}_{>0}$$

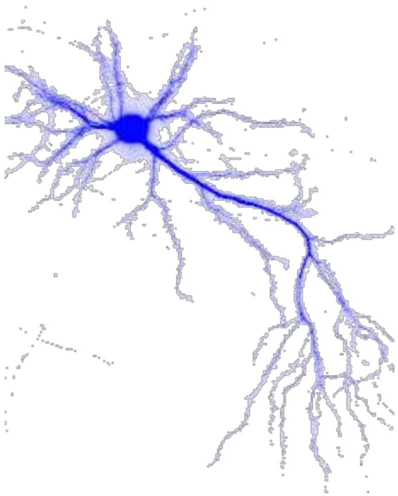
$$(\boldsymbol{w} + \Delta \boldsymbol{w})^{\mathrm{T}}\boldsymbol{x} = \underbrace{\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}_{<0} - \underbrace{\alpha \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}}_{>0}$$

Artificial Neural Network

Single-layer perceptron networks

Learning rule

- Perceptron learning rule
- Adaline learning rule
- δ-leaning rule



Perceptron Learning Rule

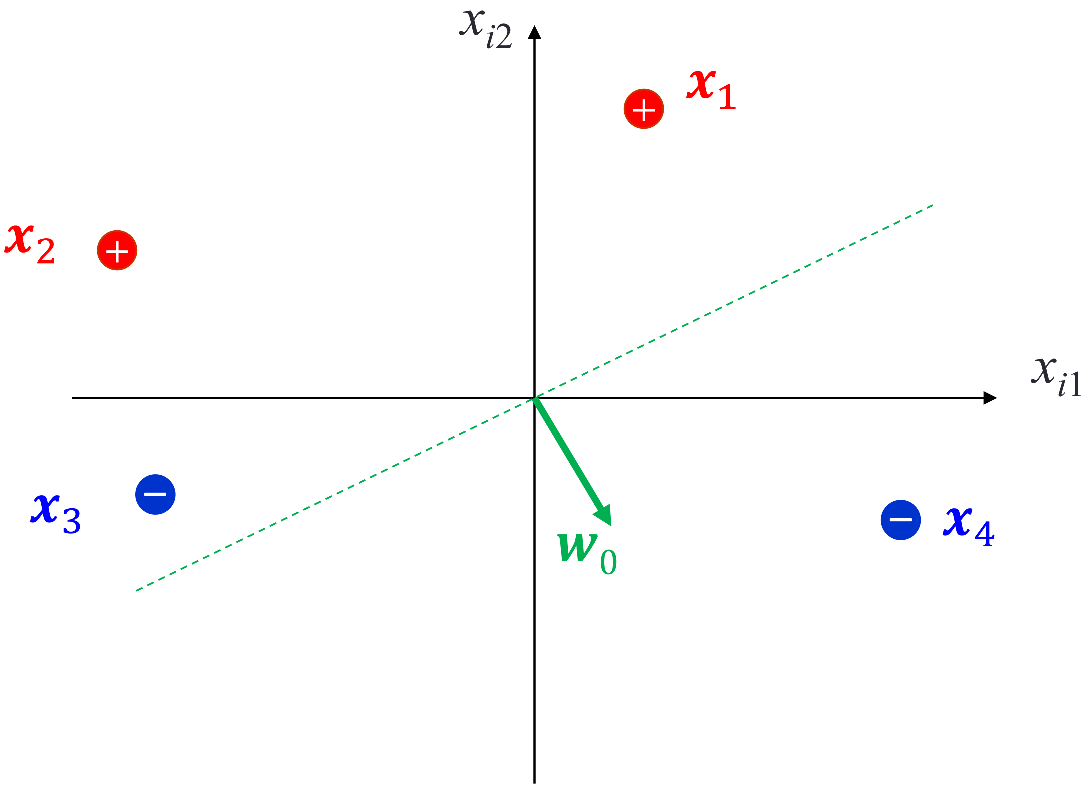
- Upon misclassification on (note: $\Re \ni \alpha > 0$)

$$\begin{cases} \Delta \boldsymbol{w} = \alpha \boldsymbol{x} & \text{for } d = +1 \\ \Delta \boldsymbol{w} = -\alpha \boldsymbol{x} & \text{for } d = -1 \end{cases}$$

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

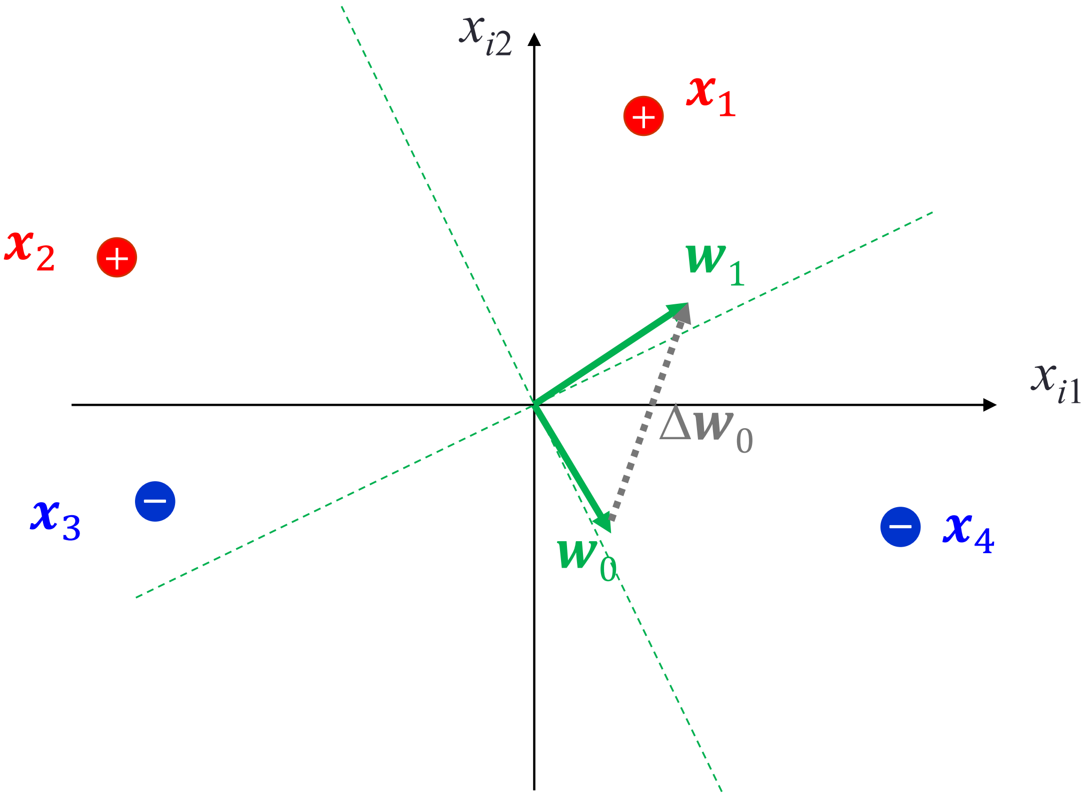
Example

- Arbitrary weight $\boldsymbol{w}_0$



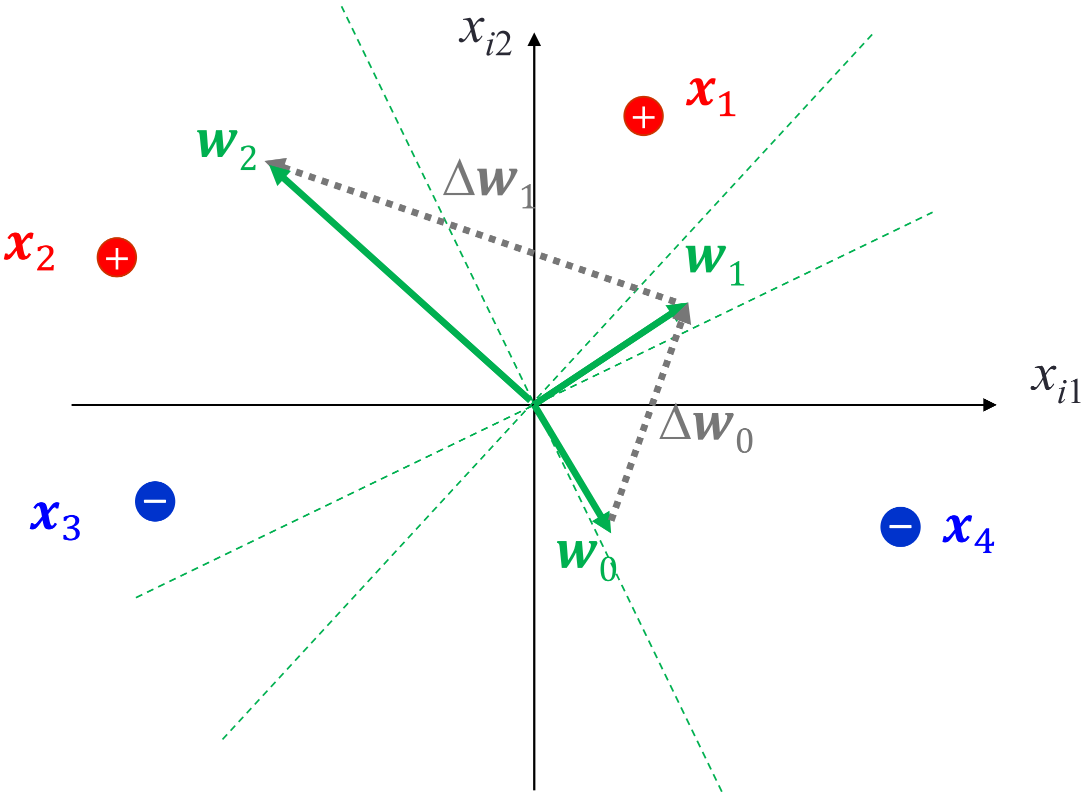
Example (Cont'd)

- Given input $\boldsymbol{x}_1$



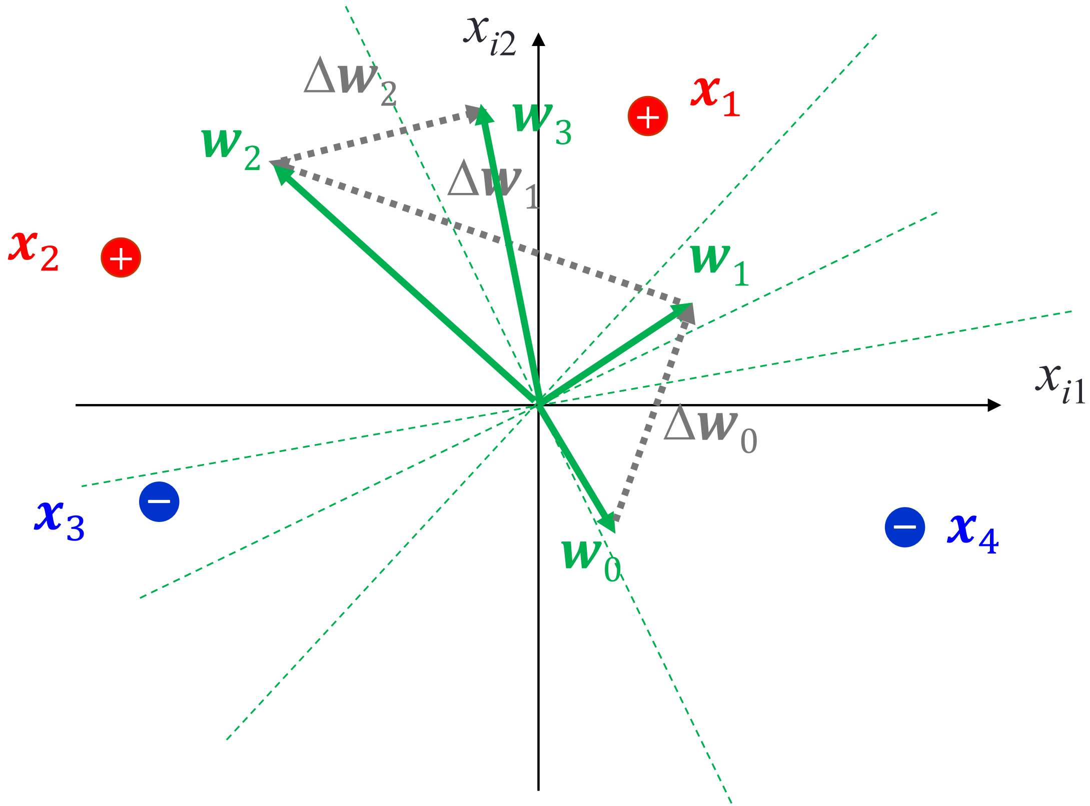
Example (Cont'd)

- Given input $\boldsymbol{x}_2$



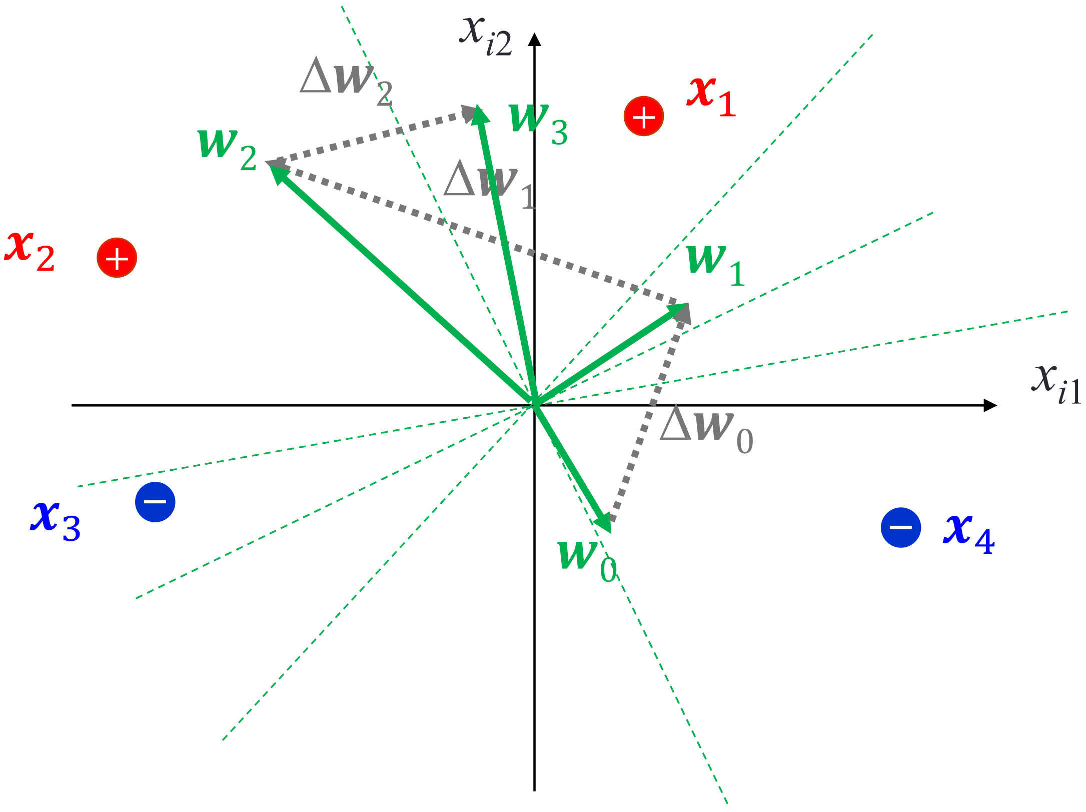
Example (Cont'd)

- Given input $\boldsymbol{x}_3$



Example (Cont'd)

- Given input $\boldsymbol{x}_4$, $\boldsymbol{w}_3$ is an appropriate weight



Perceptron Learning Rule

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

$$\begin{cases} \Delta \boldsymbol{w} = \alpha \boldsymbol{x} & \text{for } d = +1 \; \oplus \\ \Delta \boldsymbol{w} = -\alpha \boldsymbol{x} & \text{for } d = -1 \; \ominus \end{cases}$$

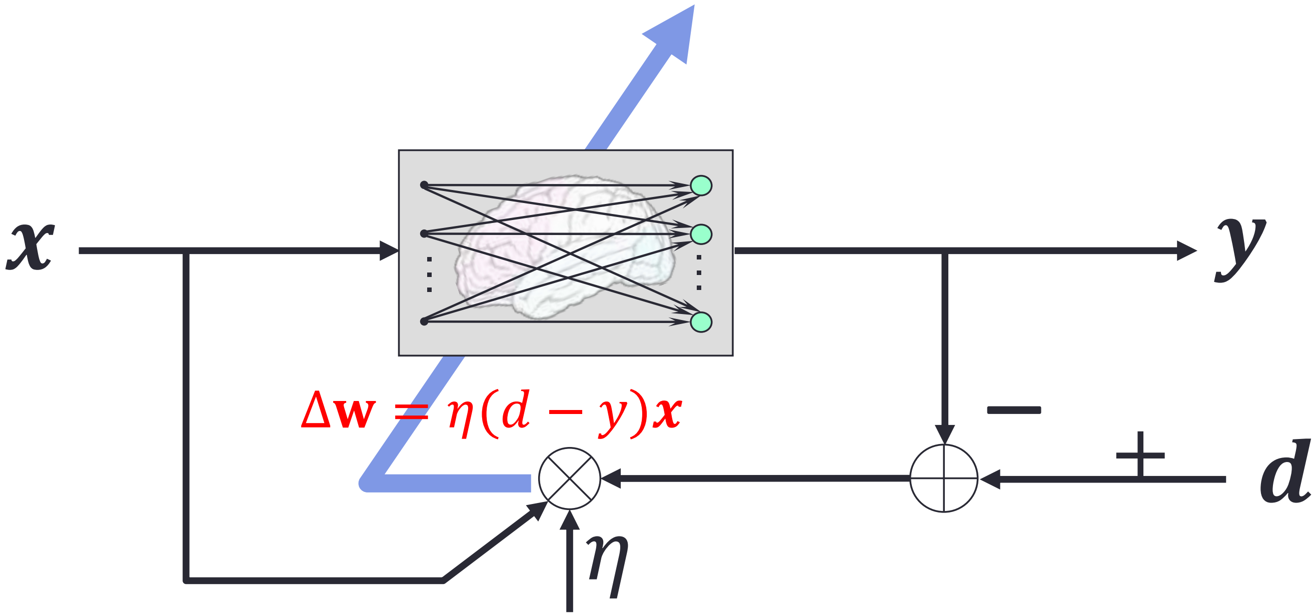
- Define error $r \in \Re$: $r = d - y = \begin{cases} +2 & \oplus \longrightarrow \ominus \\ -2 & \ominus \longrightarrow \oplus \\ 0 & \end{cases}$

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

  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

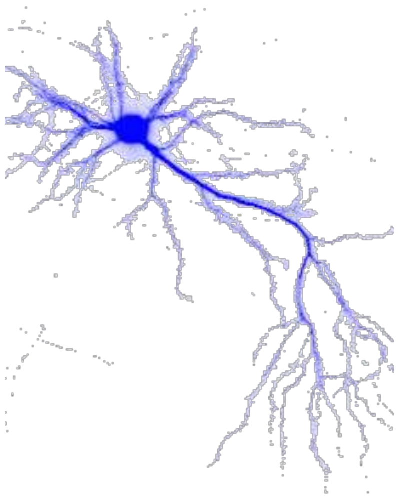


Artificial Neural Network

Single-layer perceptron networks

Learning rule

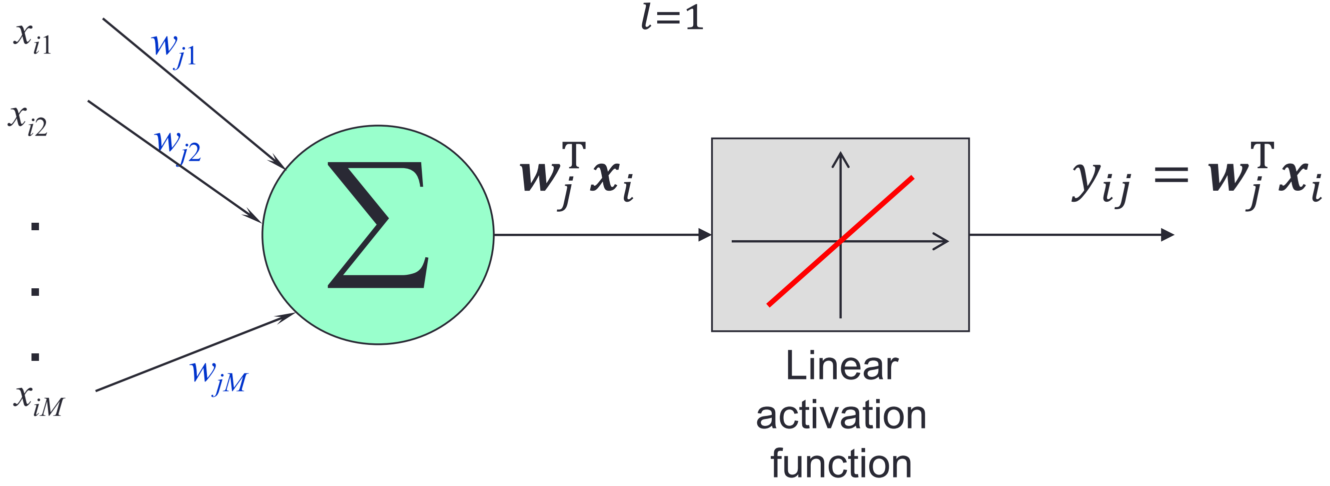
- Perceptron learning rule
- Adaline learning rule
- δ-leaning rule



Adaline (Adaptive Linear Element)

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

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



Cost Function

- Define misclassification cost function as:

$$E(\boldsymbol{w}_j) = \frac{1}{2}\sum_{i=1}^{Q}\left(d_{ij} - y_{ij}\right)^2 \in \Re$$

$$= \frac{1}{2}\sum_{i=1}^{Q}\left(d_{ij} - \boldsymbol{w}_j^{\mathrm{T}}\boldsymbol{x}_i\right)^2 = \frac{1}{2}\sum_{i=1}^{Q}\left(d_{ij} - \sum_{l=1}^{M} w_{jl}x_{il}\right)^2$$

Weight Adjustment

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

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

Adaline Learning Rule

- The gradient of the cost function

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

where

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

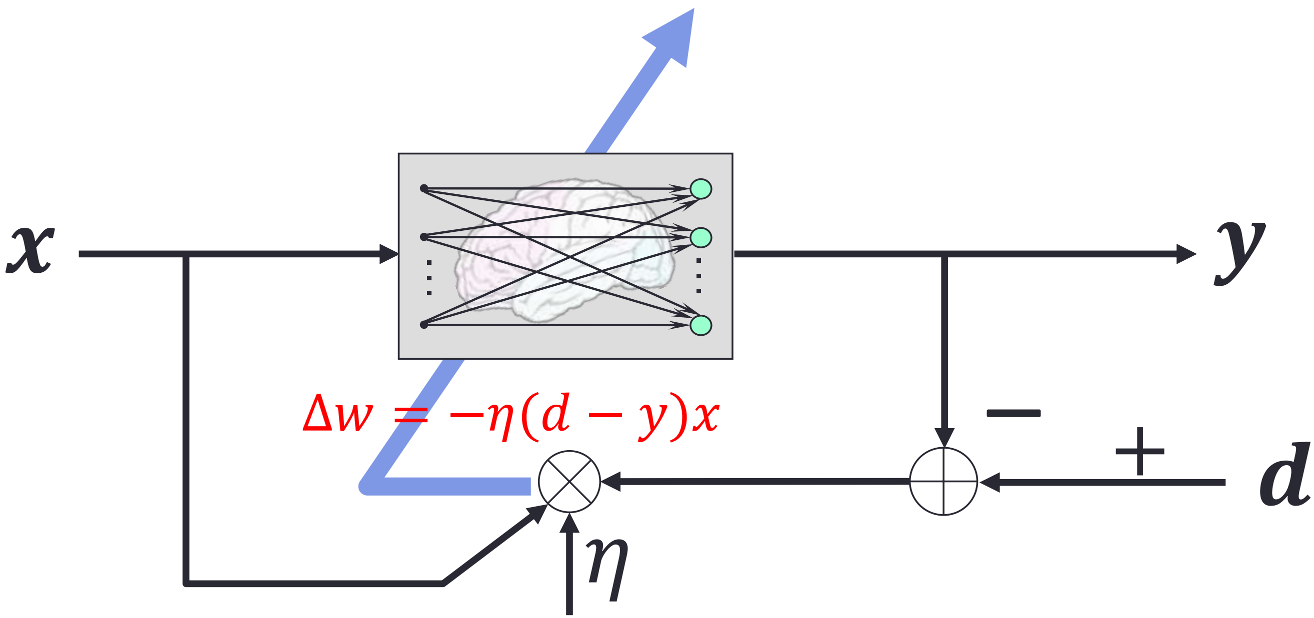
- Incremental Adaline learning rule:

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

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 $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_Q$ are statistically independent
2. At instance i, the input vector $\boldsymbol{x}_i$ is statistically independent of all previous samples of the desired response $\boldsymbol{d}_1, \boldsymbol{d}_2, \ldots, \boldsymbol{d}_{i-1}$
3. At instance i, the desired response $\boldsymbol{d}_i$ is dependent on $\boldsymbol{x}_i$, but statistically independent of all previous values of the desired response $\boldsymbol{d}_1, \boldsymbol{d}_2, \ldots, \boldsymbol{d}_{i-1}$
4. The input vector $\boldsymbol{x}_i$ and desired response $\boldsymbol{d}_i$ are drawn from Gaussian distributed populations

Adaline Convergence – η 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

$$\boldsymbol{R}_{\boldsymbol{x}} = \lim_{Q \to \infty} \frac{1}{Q} \sum_{i=1}^{Q} \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}}$$

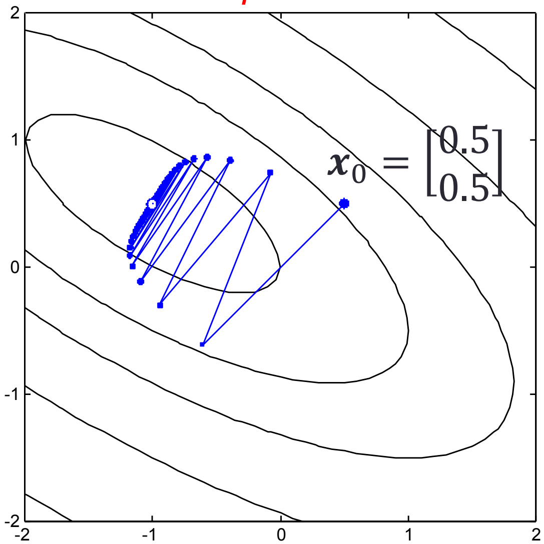
- $\lambda_{\max}$ is hardly available, usually the following convergence radius is used:

$$0 < \eta < \frac{2}{tr(\boldsymbol{R}_{\mathbf{x}})}$$

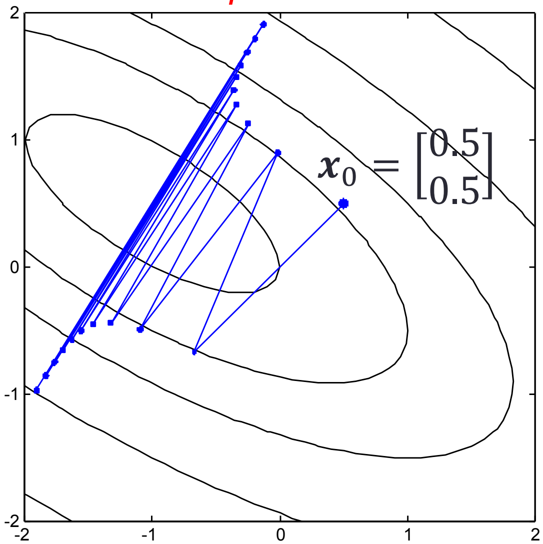
Convergence Example

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

$\eta = 0.37$



$\eta = 0.39$



Comparison

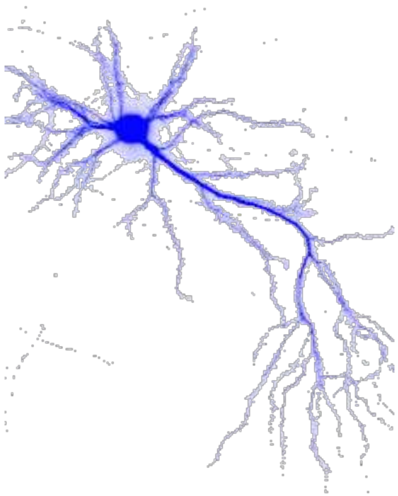
	Perceptron learning rule	Adaline learning rule (Widrow-Hoff)
Fundamental	Hebbian rule	Gradient decent
Convergence	In finite steps	Converge asymptotically
Constraint	Linearly separable	Linear independence

Artificial Neural Network

Single-layer perceptron networks

Learning rule

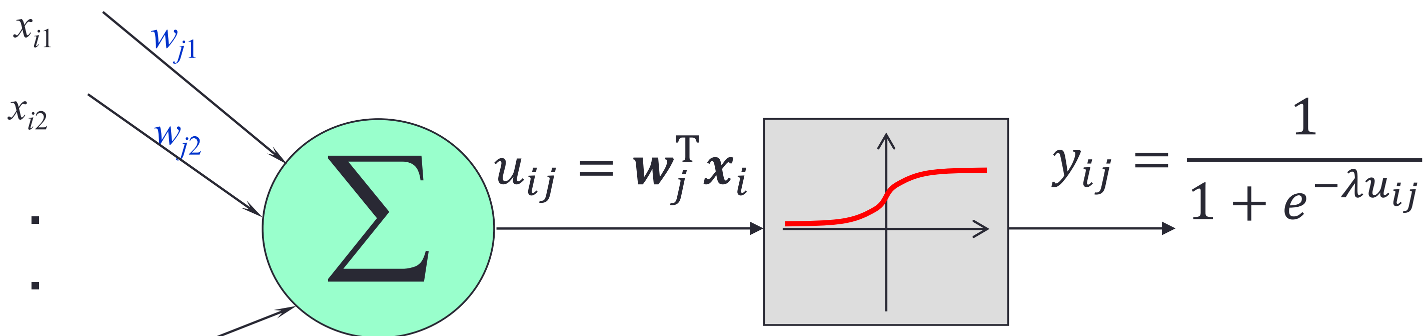
- Perceptron learning rule
- Adaline learning rule
- δ-leaning rule



Unipolar Sigmoid Activation Function

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

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

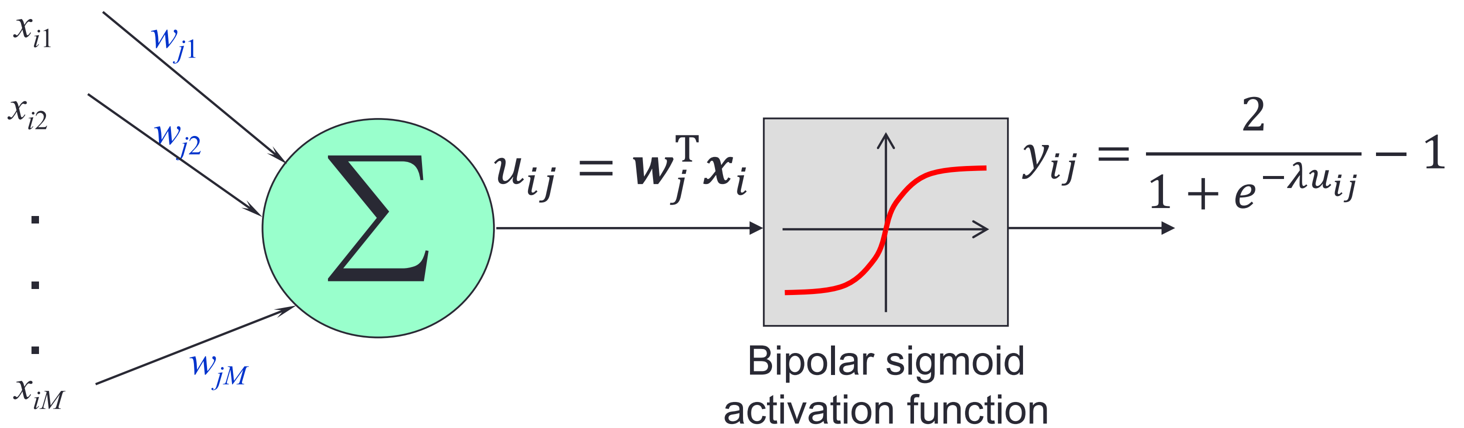


Unipolar sigmoid activation function

Bipolar Sigmoid Activation Function

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

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



Cost Function and Weight Adjustment

- Define misclassification cost function as:

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

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

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

Gradient of the Cost Function

- The gradient of the cost function

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

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

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

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

Depends on the activation function

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

δ Learning Rule

- The gradient of the cost function:

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

$$= \left(-\sum_{i=1}^{Q} (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} x_{i1}, \dots, -\sum_{i=1}^{Q} (d_{ij} - y_{ij}) \frac{\partial a(u_{ij})}{\partial u_{ij}} x_{iM} \right)^{\mathrm{T}}$$

- Incremental learning rule:

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

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 δ Learning Rule

- Unipolar sigmoid:

$$\Delta \boldsymbol{w}_j = \eta\left(d_{ij} - y_{ij}\right)\lambda y_{ij}\,(1 - y_{ij})\boldsymbol{x}_i$$

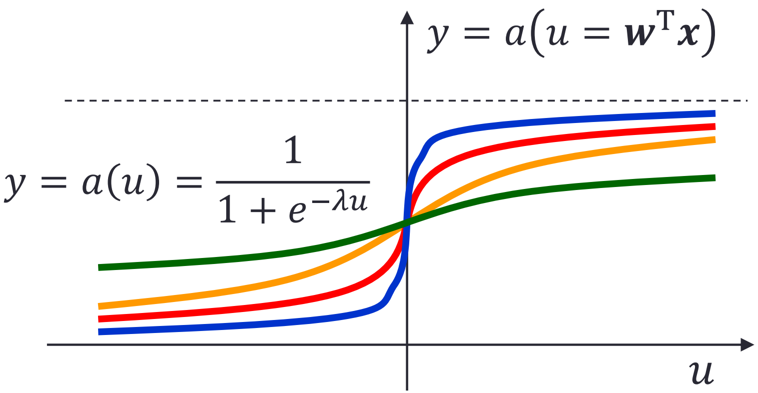
- Bipolar sigmoid:

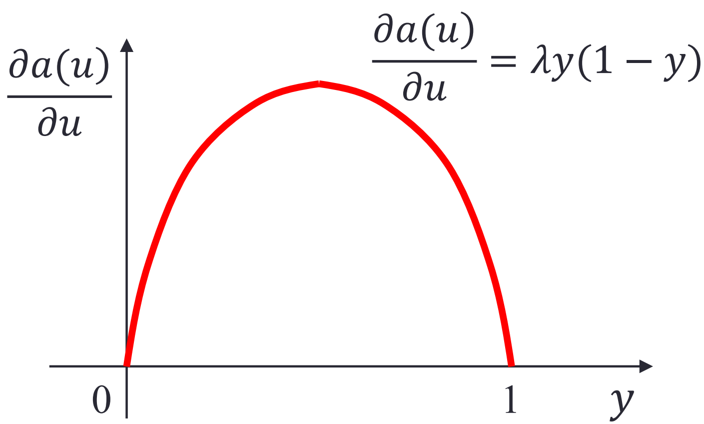
$$\Delta \boldsymbol{w}_j = 2\eta\left(d_{ij} - y_{ij}\right)\lambda y_{ij}\,(1 - y_{ij})\boldsymbol{x}_i$$

Saturation of Sigmoid

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

  2. Large $\boldsymbol{w}$ $\Rightarrow$ $y \sim 1$ $\Rightarrow$ $\frac{\partial a(u)}{\partial u} \sim 0$

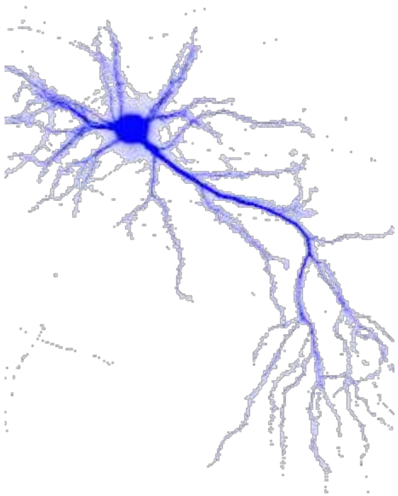




Artificial Neural Network

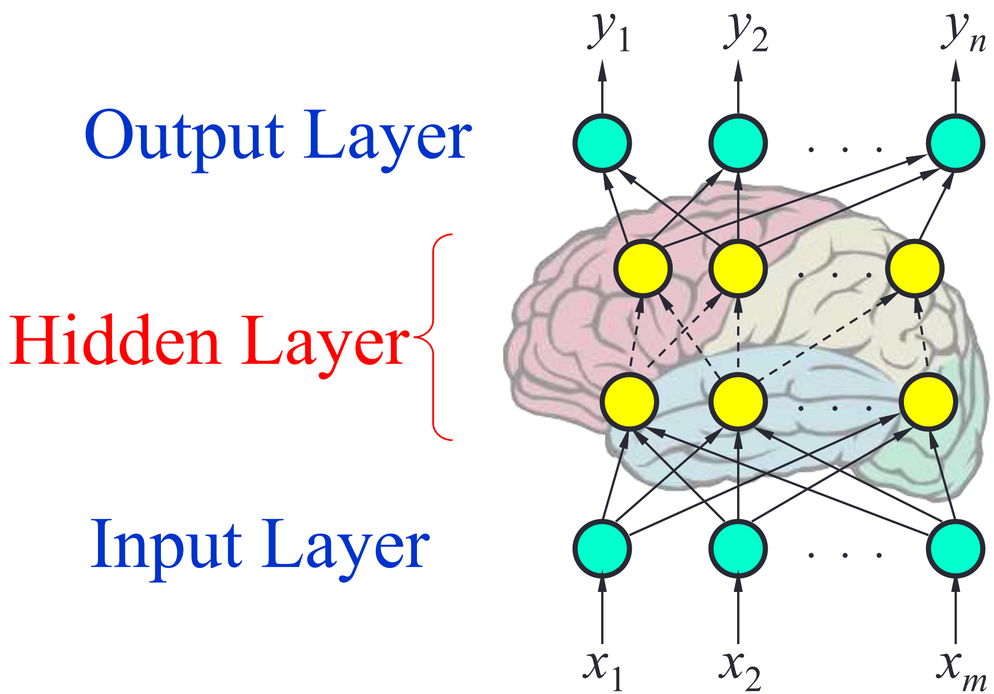
Multilayer perceptron

- Examples
- Back propagation learning algorithm

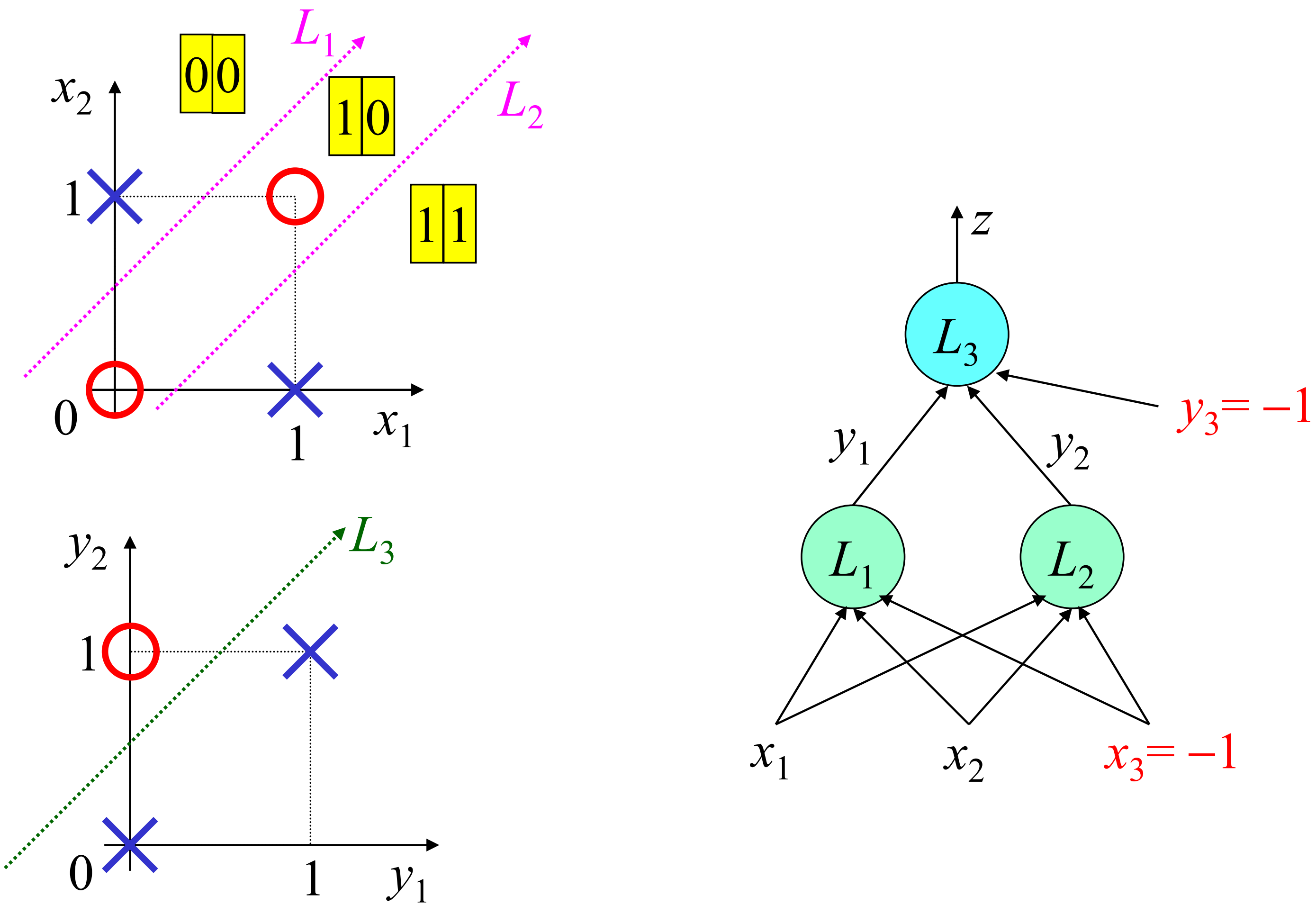


Multilayer Perceptron

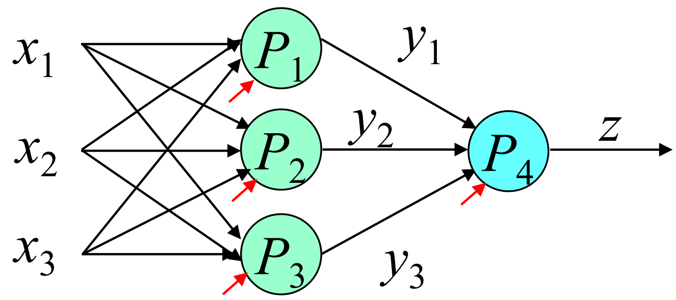
- Multilayer perceptron can handle problems that are not linearly separable

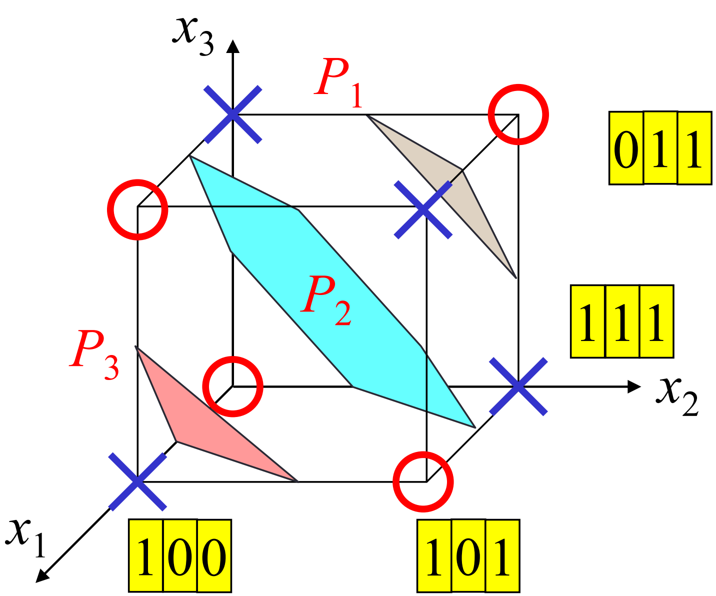


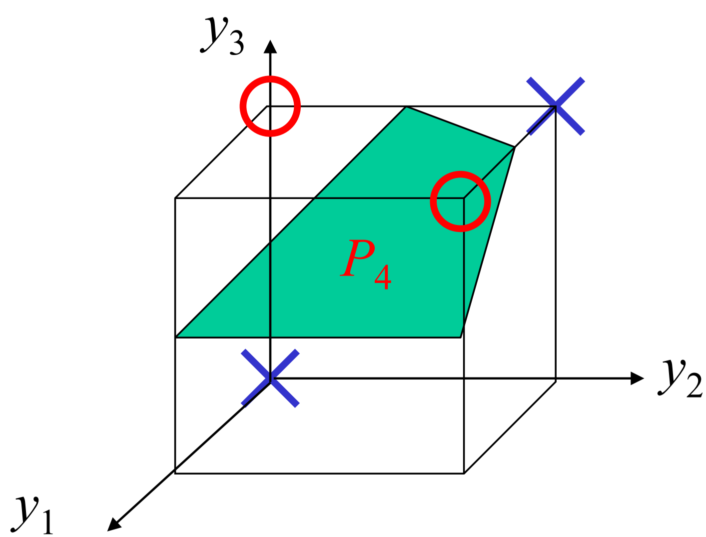
Example: XOR Problem



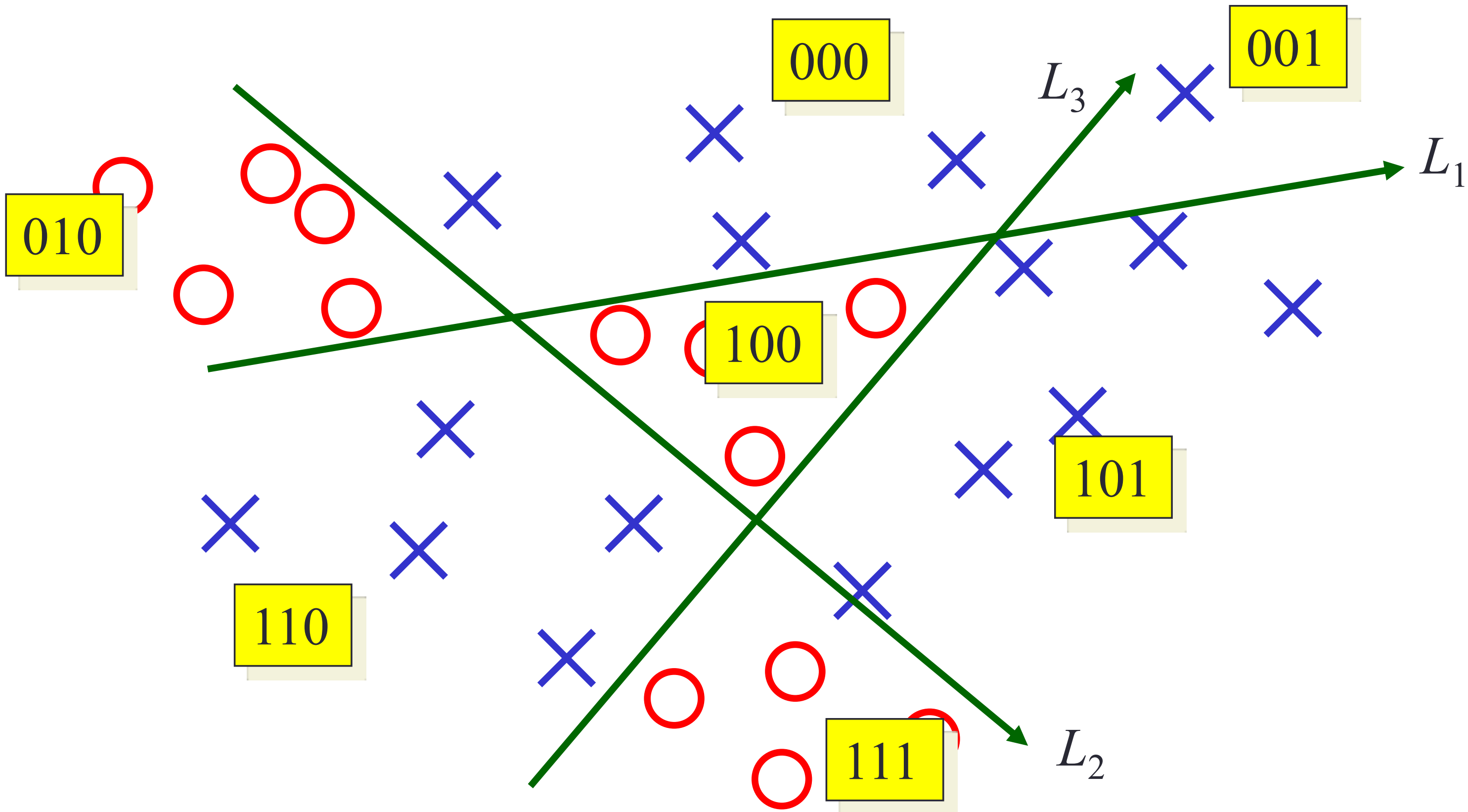
Another Example: Parity Problem



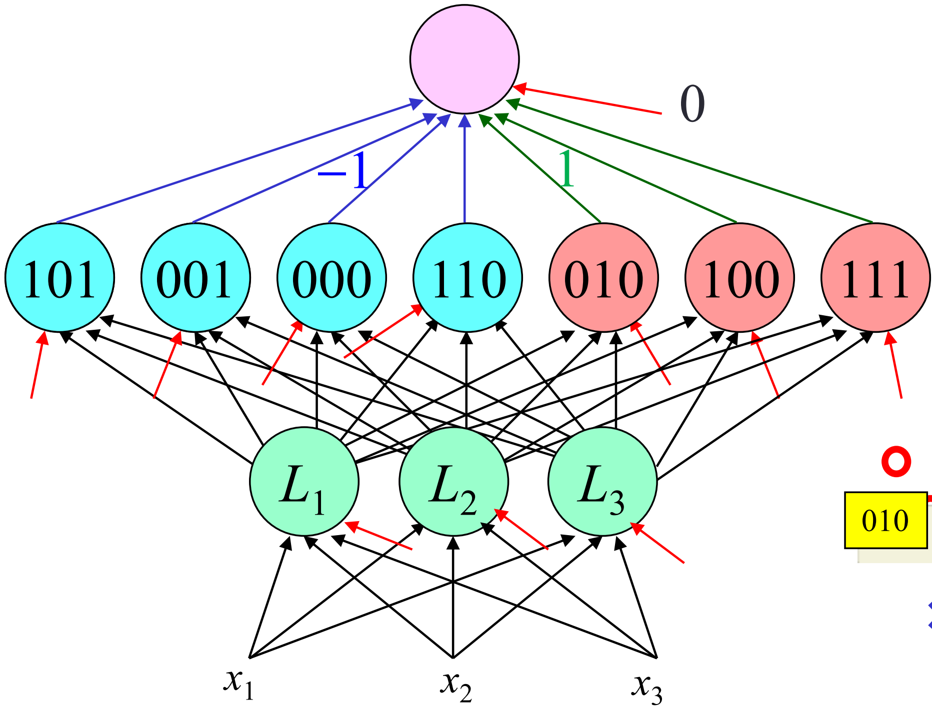


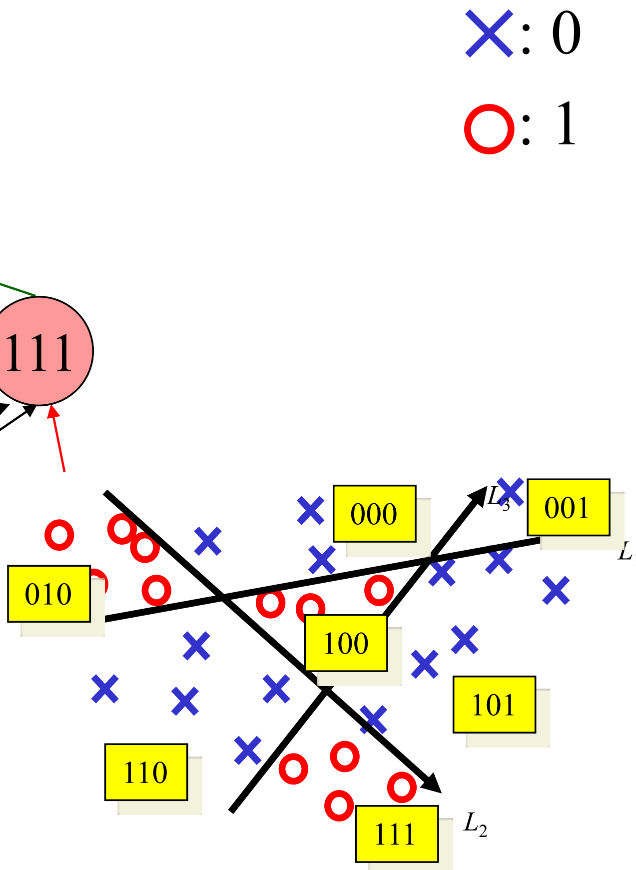


Another Example: Partition



Another Example: Partition (Cont'd)

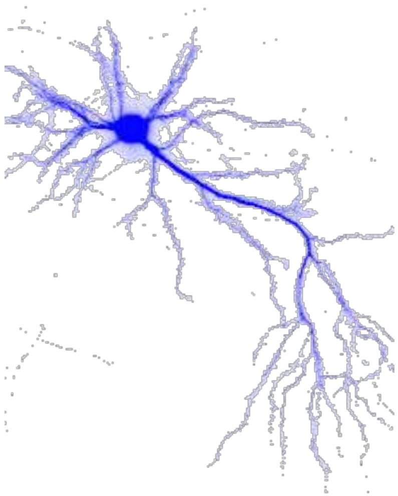




Artificial Neural Network

Multilayer perceptron

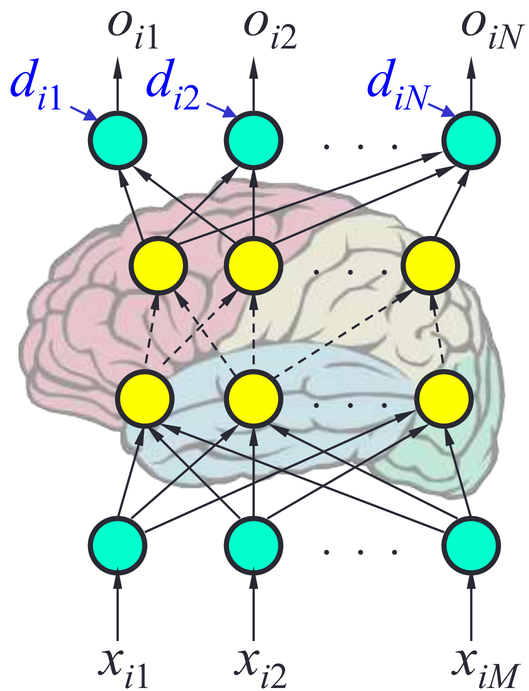
- Examples
- Back propagation learning algorithm



Supervised Learning

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

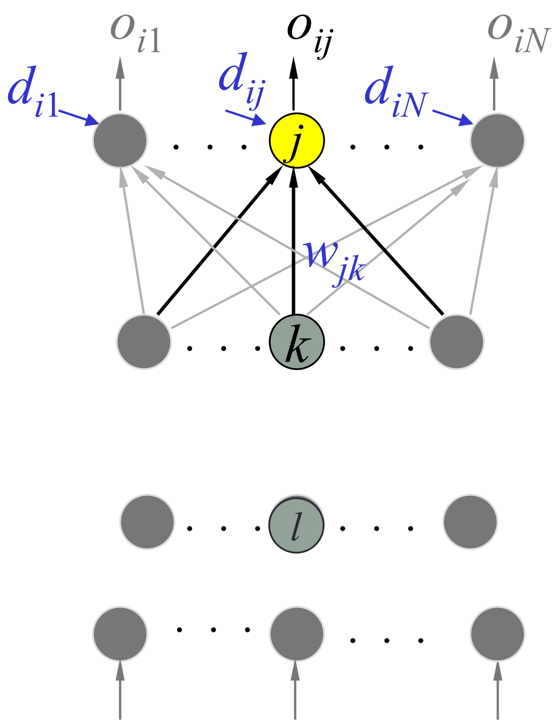
$$E = \sum_{i=1}^{Q} E_i = \sum_{i=1}^{Q} \left[\frac{1}{2} \sum_{j=1}^{N} (d_{ij} - o_{ij})^2 \right]$$



- Objective is to obtain a set of weights that minimize E for both the <u>output</u> and <u>hidden</u> neurons

Back Propagation

- Update the weights backward

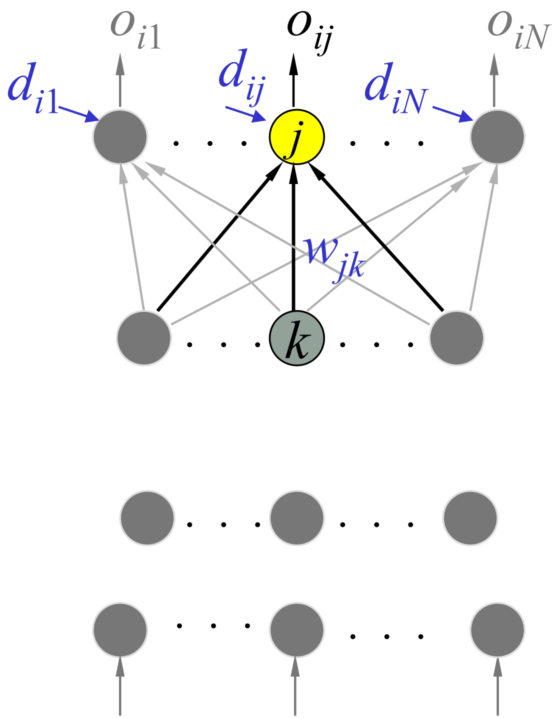


Notes:

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



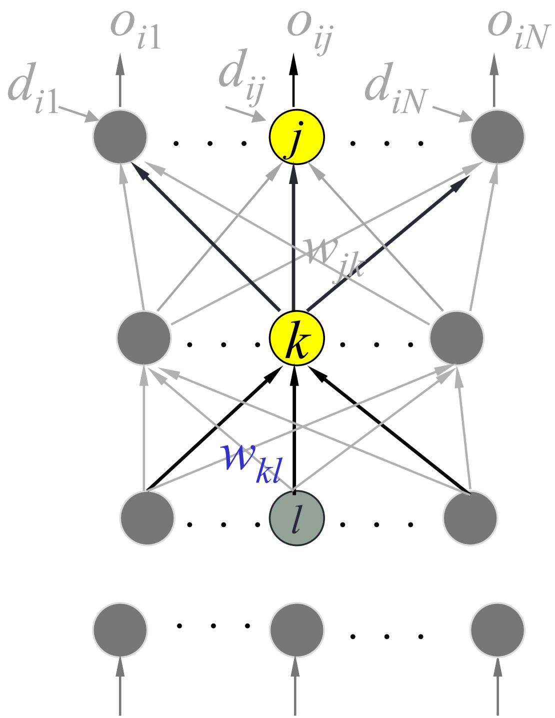
$$\frac{\partial E}{\partial w_{jk}} = \sum_{i=1}^{Q} \frac{\partial E_i}{\partial w_{jk}} = \sum_{i=1}^{Q} \frac{\partial E_i}{\partial o_{ij}} \frac{\partial o_{ij}}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial w_{jk}}$$

$$= \sum_{i=1}^{Q} \underbrace{-(d_{ij} - o_{ij}) \cdot \lambda o_{ij}(1 - o_{ij})}_{\delta_{ij}} \cdot o_{ik}$$

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

Learning on Hidden Neurons

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



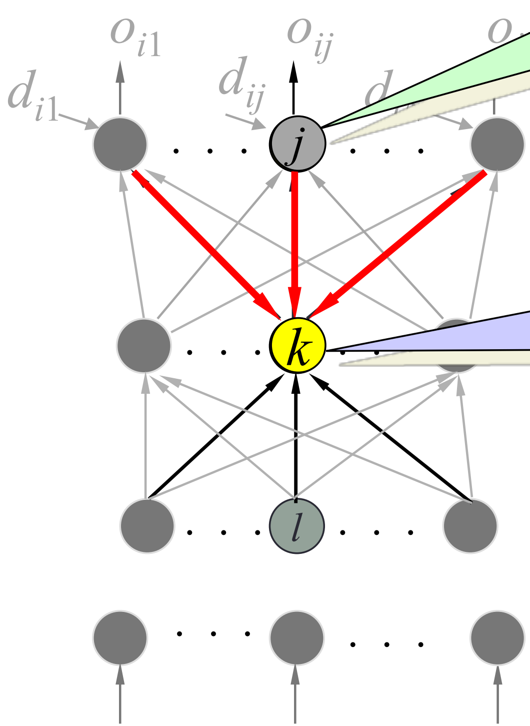
$$\frac{\partial E}{\partial w_{kl}} = \sum_{i=1}^{Q} \frac{\partial E_i}{\partial w_{kl}} = \sum_{i=1}^{Q} \frac{\partial E_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial u_{ik}} \frac{\partial u_{ik}}{\partial w_{kl}}$$

$$= \sum_{i=1}^{Q} \sum_{j} \frac{\partial E_i}{\partial u_{ij}} \frac{\partial u_{ij}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial u_{ik}} \frac{\partial u_{ik}}{\partial w_{kl}}$$

$$= \sum_{i=1}^{Q} \underbrace{\sum_{j} \delta_{ij} \cdot w_{jk} \cdot \lambda o_{ik}(1 - o_{ik})}_{\delta_{ik}} \cdot o_{il}$$

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

Back Propagation



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

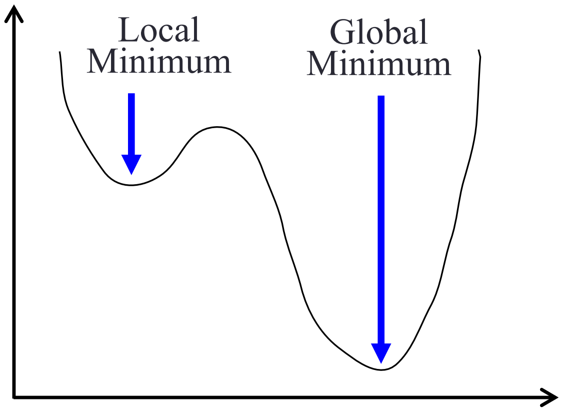
$$\Delta w_{jk} = -\eta \sum_{i=1}^{Q} \delta_{ij}\, o_{ik}$$

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

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

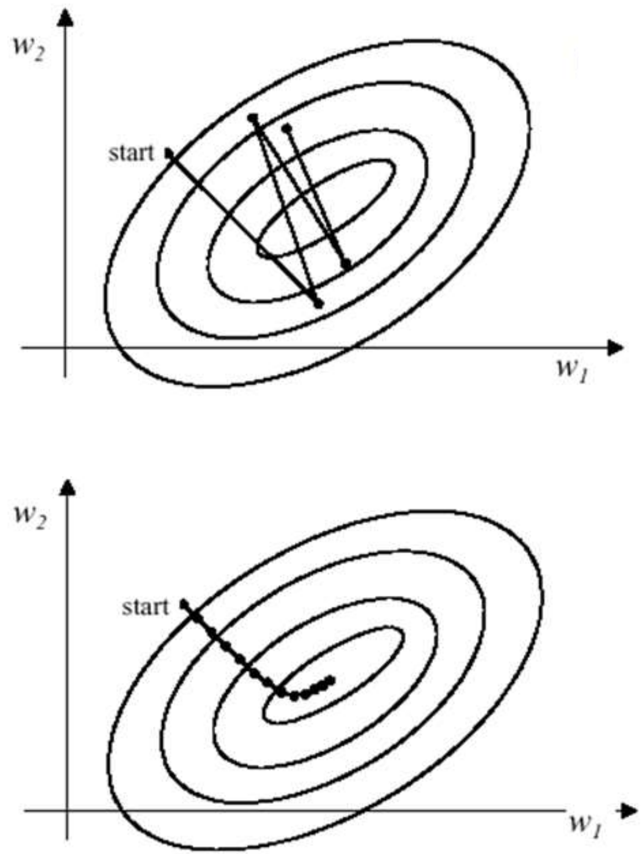
Back Propagation Using Gradient Descent

- 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

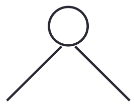
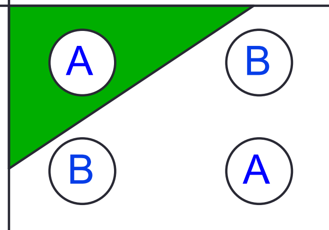
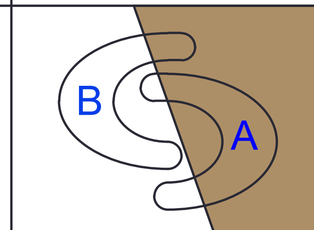
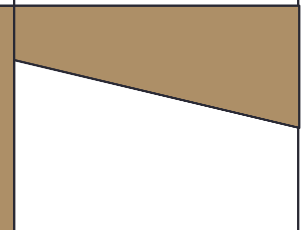
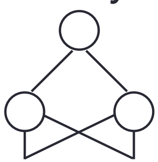
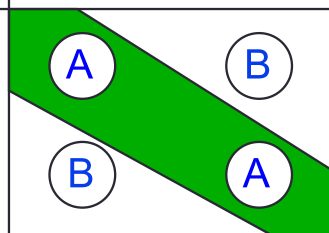
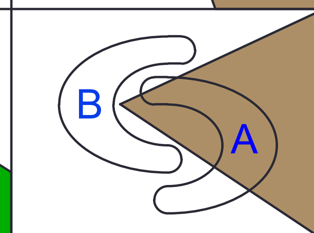
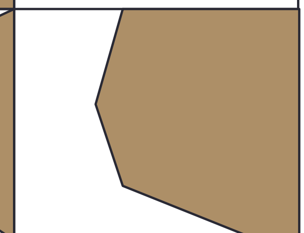
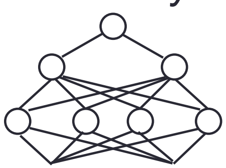
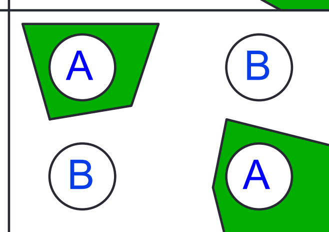
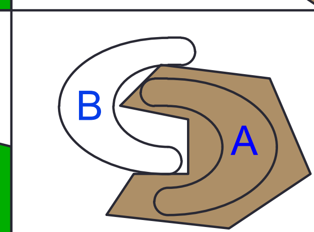
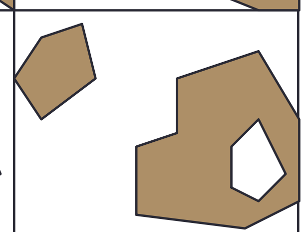
- Weight update rules
- Initial weight
- Learning rate η
- Number of nodes
- Number of hidden layers
- Stopping criterions



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)

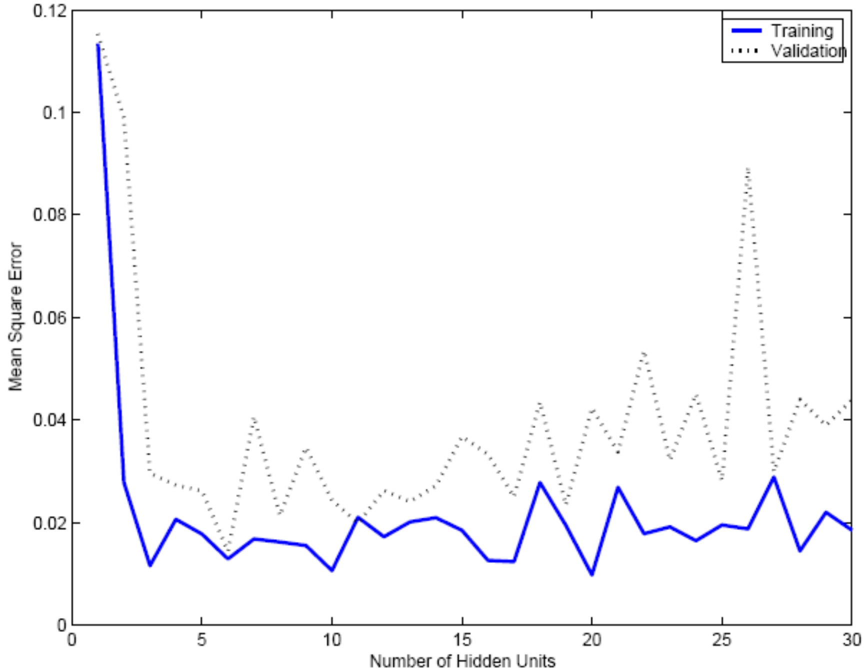
Rule of Thumb for Hidden Layers

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:

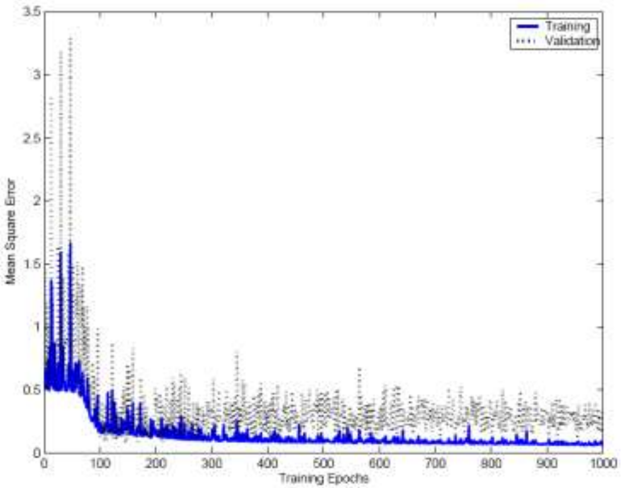
1. Hold out part of the sample
2. Cross-validation
3. Bootstrapping



Alpaydin, *Introduction to machine learning*

Stopping Criterions

- 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*

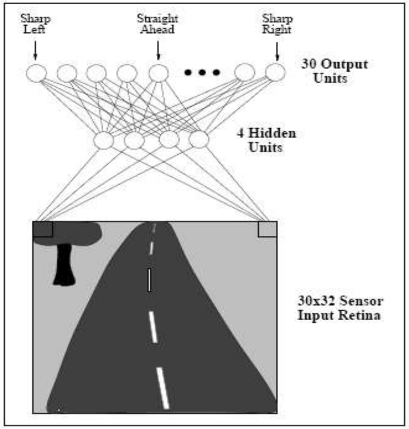
Autonomous Land Vehicle In a Neural Network

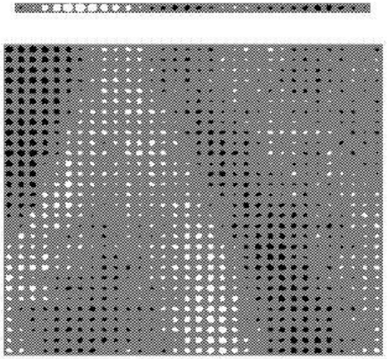
- Drives 70 mph on a public highway

30 outputs for steering

4 hidden units

30x32 pixels as inputs





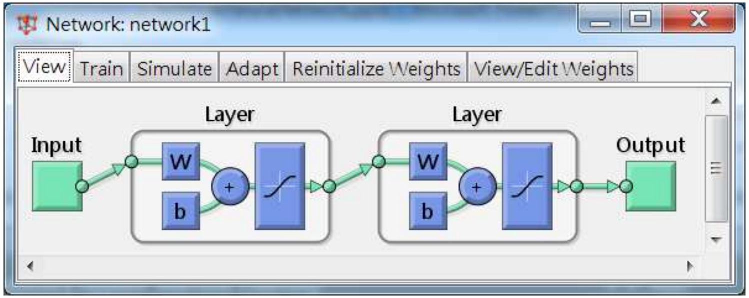
30x32 weights into one out of four hidden unit

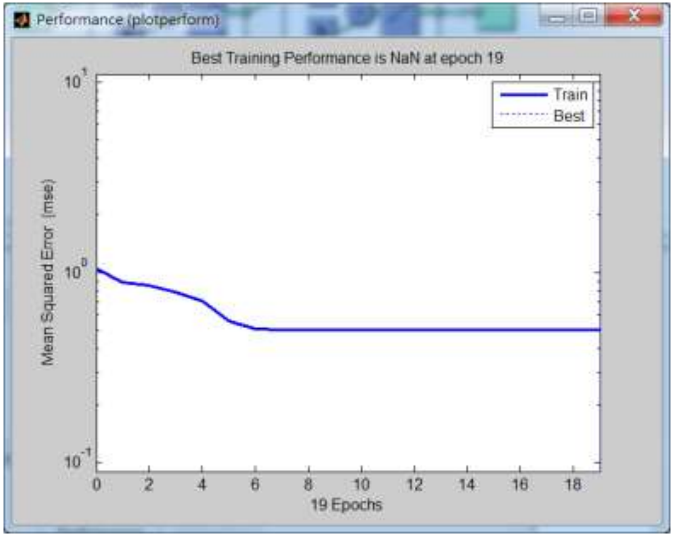
XOR Problem

- XOR:

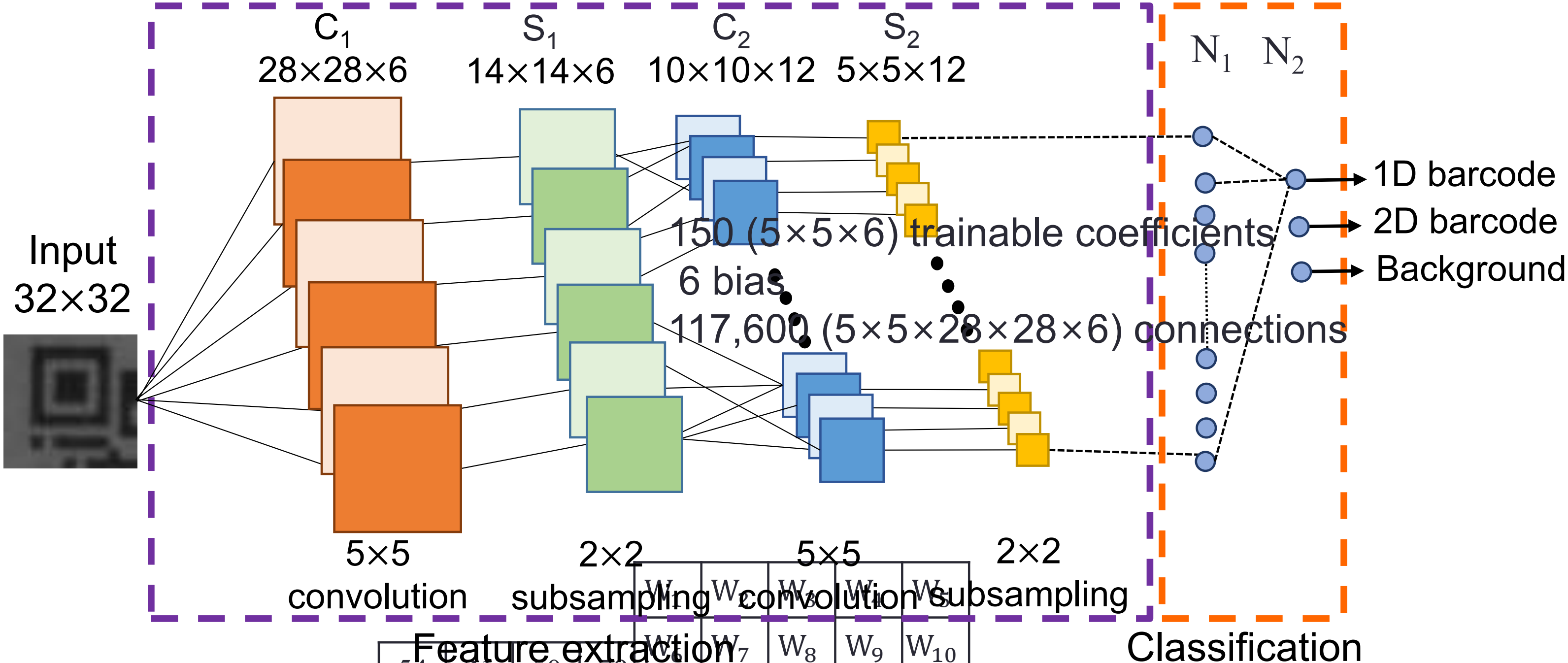
Input x	Output y
(+1, +1)	-1
(+1, -1)	+1
(-1, +1)	+1
(-1, -1)	-1

- Matlab ANN tool: nntool





Convolutional Neural Network



54	21	50	79
10	90	89	59
19	85	54	41
06	94	10	91

W_1	W_2	W_3	W_4	W_5
W_6	W_7	W_8	W_9	W_{10}
W_{11}	W_{12}	W_{13}	W_{14}	W_{15}
W_{16}	W_{17}	W_{18}	W_{19}	W_{20}
W_{21}	W_{22}	W_{23}	W_{24}	W_{25}

Layer C_1 :150 (5×5×6) trainable coefficients, 6 bias, sigmoid function
Layer C_2 :300 (5×5×12) trainable coefficients, 12 bias, sigmoid function
Layer N_2 :900 (5×5×12×3) 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_PDP86.pdf).
- C. Bishop, *Neural Networks for Pattern Recognition*

Acknowledgement

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