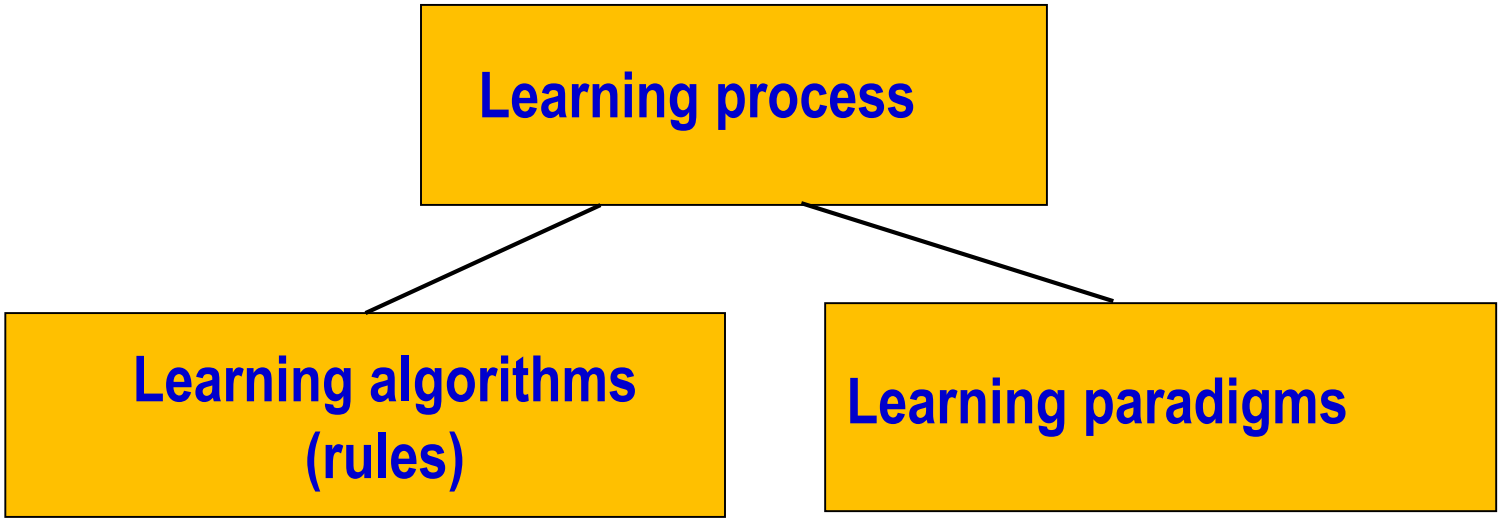
Neural Network Theory and Applications

Bao-Liang Lu（吕宝粮）
Dept. of Computer Science & Engineering
Shanghai Jiao Tong University
Office: 3-431 Dianxin Bld, Minhang
Tel: 3420-5422
bllu@sjtu.edu.cn

Outline of Lecture Two

- A taxonomy of the learning process
- Basic learning rules
- Notation and definition
- Chapter 3: An illustrative example
- Chapter 4: Perceptron learning rule
  - Perceptron architecture
  - Perceptron learning rule
  - Proof of convergence
- Examples of Perceptron learning rule
- Least Mean Square Algorithm
- Back-propagation algorithm

A taxonomy of the learning process



- Memory-based
- Error-correct
- Hebbian Postulate
- Competitive

- Supervised
- Unsupervised
- Reinforcement

Memory-Based Learning -1

- In *memory-based learning*, all (or most) of past experiences are explicitly stored in a large memory.
- This consists of correctly classified input-output examples $\{(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), \ldots, (\mathbf{x}_N, d_N)\}$.
- Again, $\mathbf{x}_i$ denotes i-th input vector and d_i the corresponding desired response.
- Without loss of generality, d_i can be restricted to be a scalar.
- Typically, d_i is the number of pattern class.
- Consider now classification of a test vector $\mathbf{x}_{test}$ not seen before.
- This is done by retrieving and analyzing the training data in a local neighborhood of $\mathbf{x}_{test}$.
- All memory-based learning algorithms involve two parts:

Memory-Based Learning -2

1. Criterion used for defining the local neighborhood of the test vector $\mathbf{x}_{test}$.
2. Learning rule applied to the training examples in the local neighborhood of the test vector $\mathbf{x}_{test}$.

- A simple yet effective memory-based learning algorithm is known as the **nearest-neighbor rule**.
- The vector $\mathbf{x}'_N$ belonging to the set of training vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ is the nearest neigbor of $\mathbf{x}_{test}$ if

$$\min_i d(\mathbf{x}_i, \mathbf{x}_{test}) = d(\mathbf{x}'_N, \mathbf{x}_{test})$$

  where $d(\mathbf{x}_i, \mathbf{x}_{test})$ is the Euclidean distance between the vectors $\mathbf{x}_i$ and $\mathbf{x}_{test}$.
- $\mathbf{x}_{test}$ is classified into the same class as its nearest neigbor $\mathbf{x}'_N$.
- Nearest neigbor rule is independent of the underlying distribution.

Memory-Based Learning -3

- Assume that:
  - The training samples are independently and identically distributed;
  - The sample size N is infinitely large.
- One can then show that the probability of error in nearest neighbor classification is at most twice the *Bayes probability of error* (Cover and Hart, 1967).
- The Bayes error is the smallest possible (optimal one).

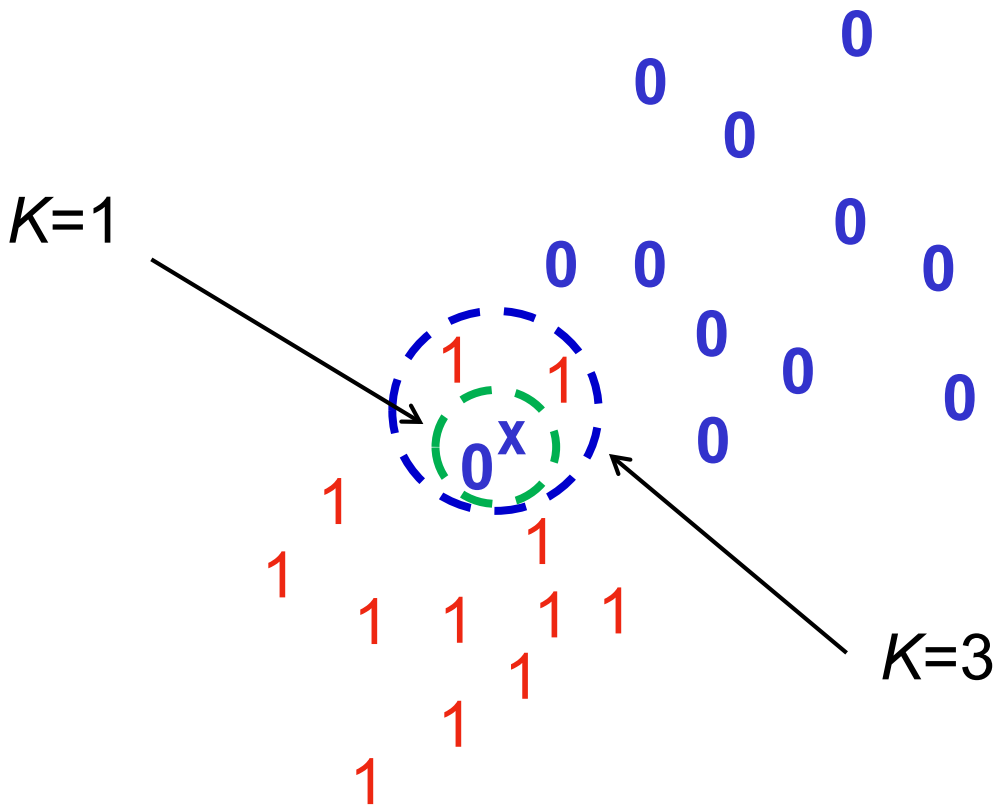
$$P_B \le P_{NN} \le P_B\left(2 - \frac{M}{M-1} P_B\right) \le 2P_B$$

- Thus half the classification information in an infinitely large training set is contained in the nearest neighbor.
- **k-nearest neighbor classifier**:
  $\mathbf{x}_{test}$ is classified to the class which is most frequently represented in its k-nearest neighbors (a majority vote).

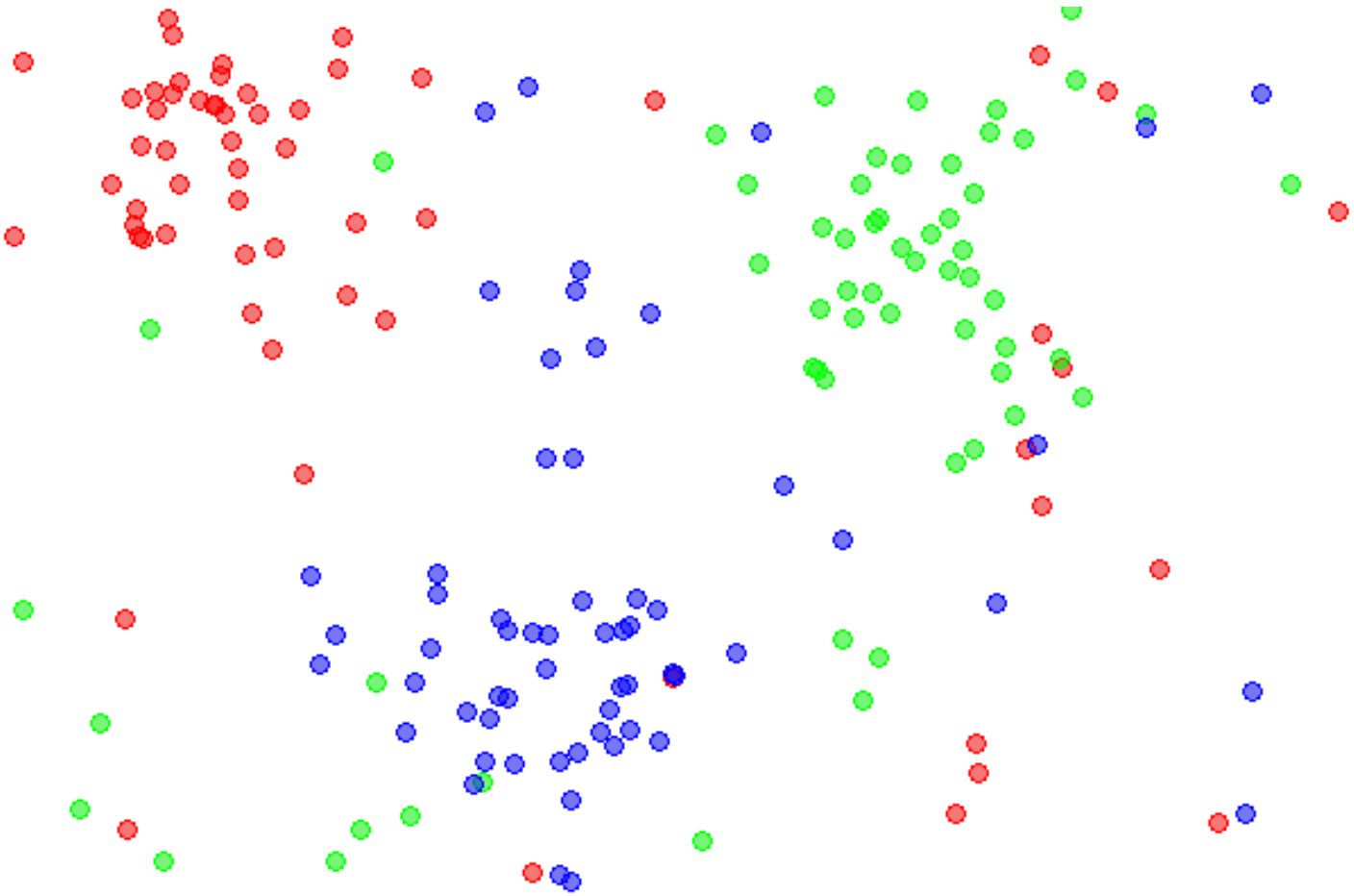
Memory–Based Learning –4

- The k-nearest neigbor classifier averages information, rejecting single outliers.
- *outlier* = exceptional, often erroneous observation.

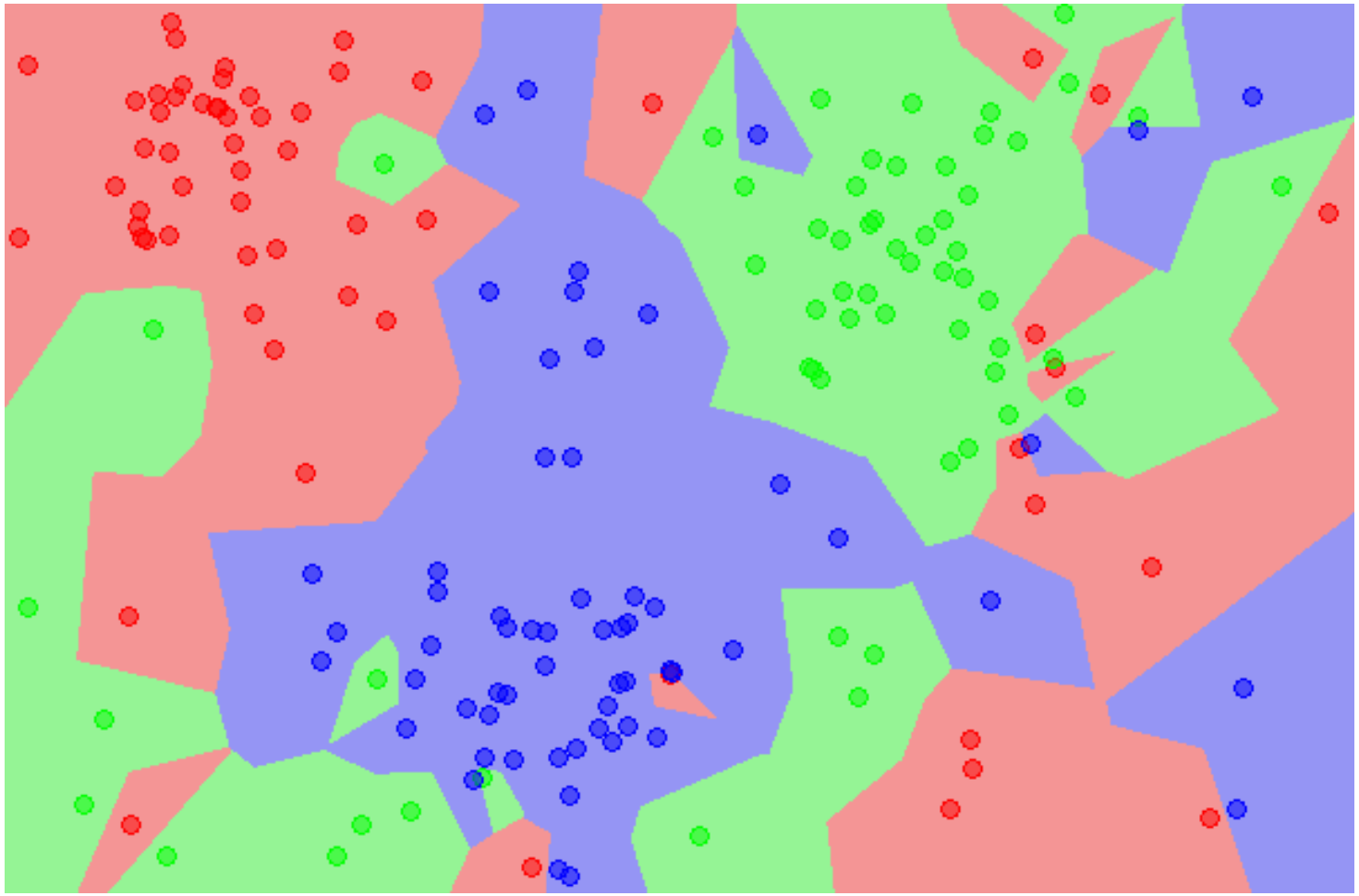
K-Nearest Neighbor Classifier



A three-class problem



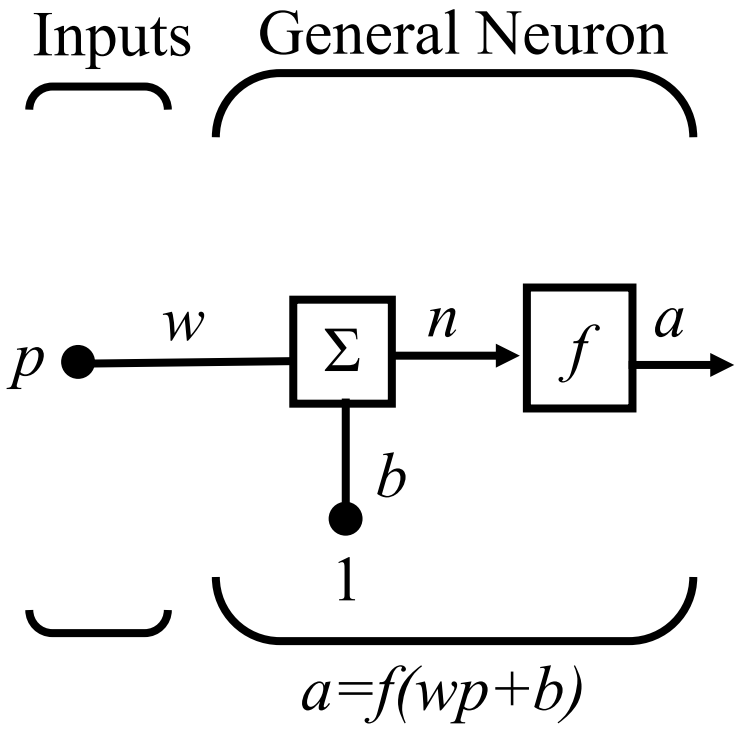
Result of NN Classifier (*k*=1)



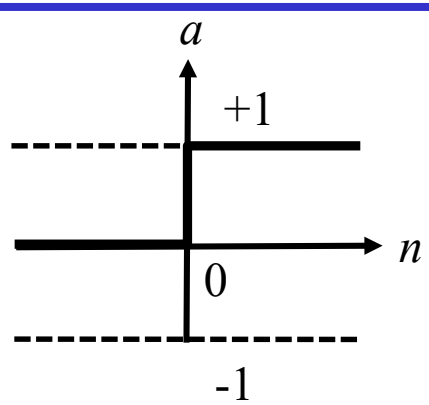
Voronoi Tessellation (棋盘形分布）

Notation and definition

Single-Input Neuron

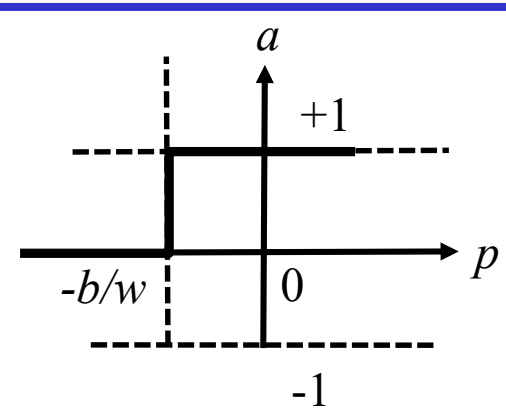


Transfer Functions



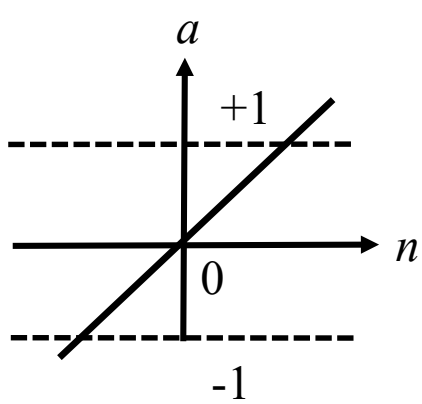
a=hardlim(n)

Hard Limit Transfer Function



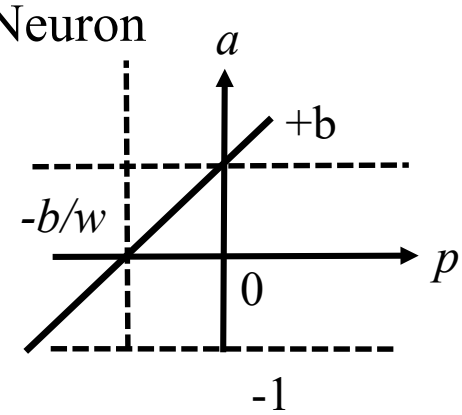
a=hardlim(wp+b)

Single-Input *hardlim* Neuron



a=purelin(n)

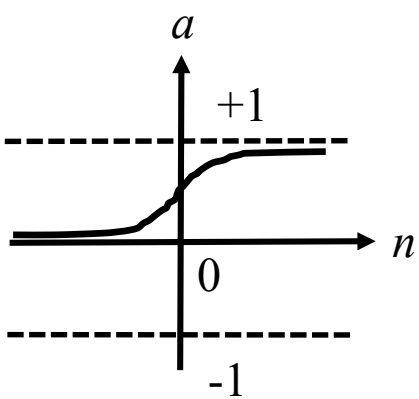
Linear Transfer Function



a=purelin(wp+b)

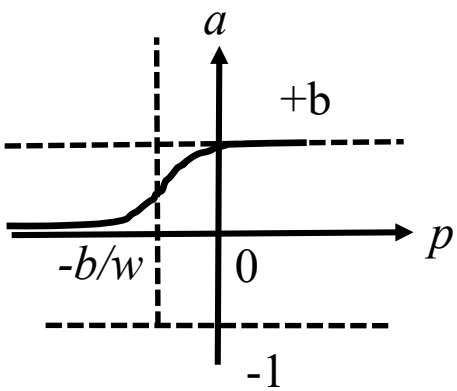
Single-Input *purelin* Neuron

Transfer Functions



a=logsig(n)

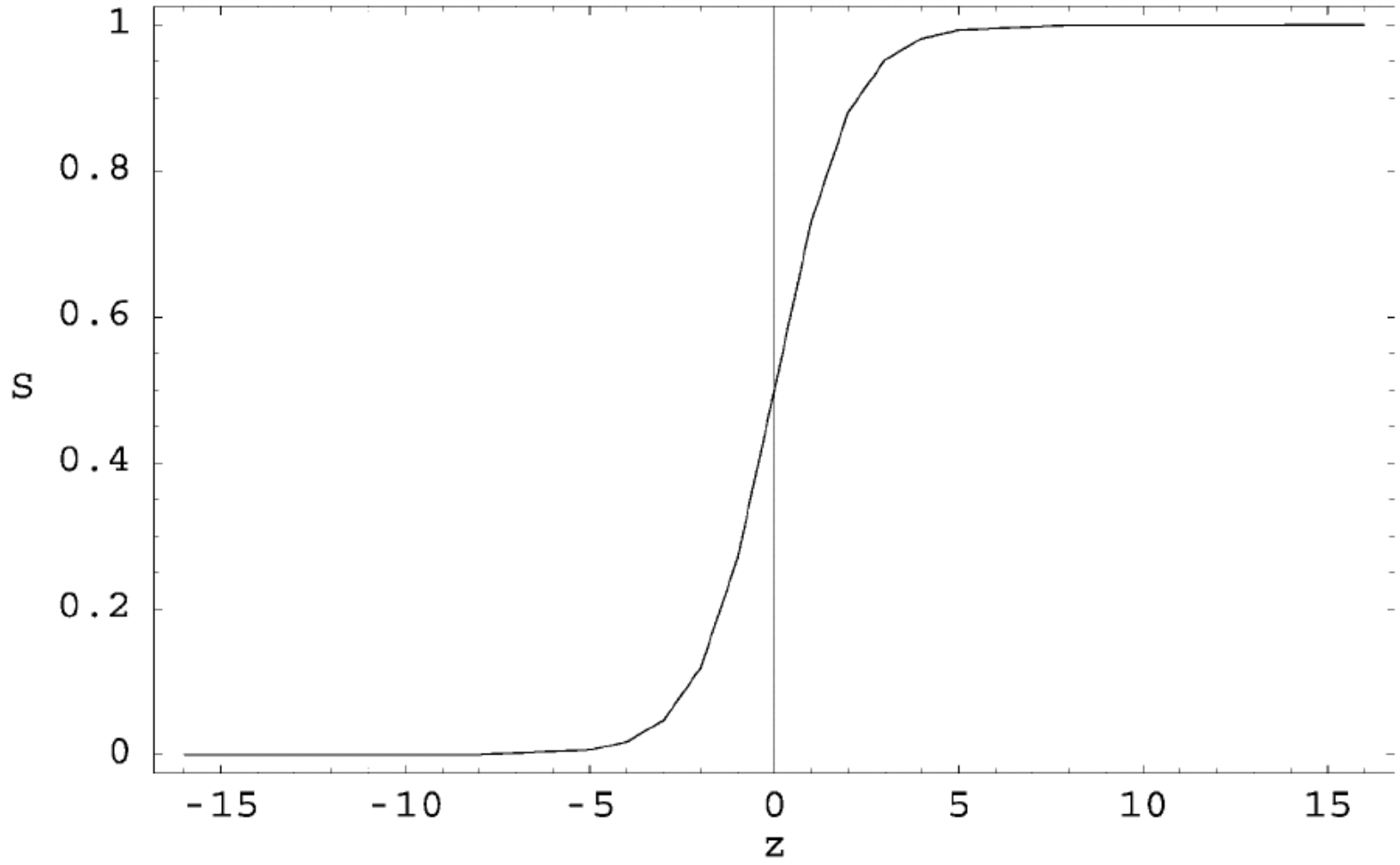
Log-Sigmoid Transfer Function



a=logsig(wp+b)

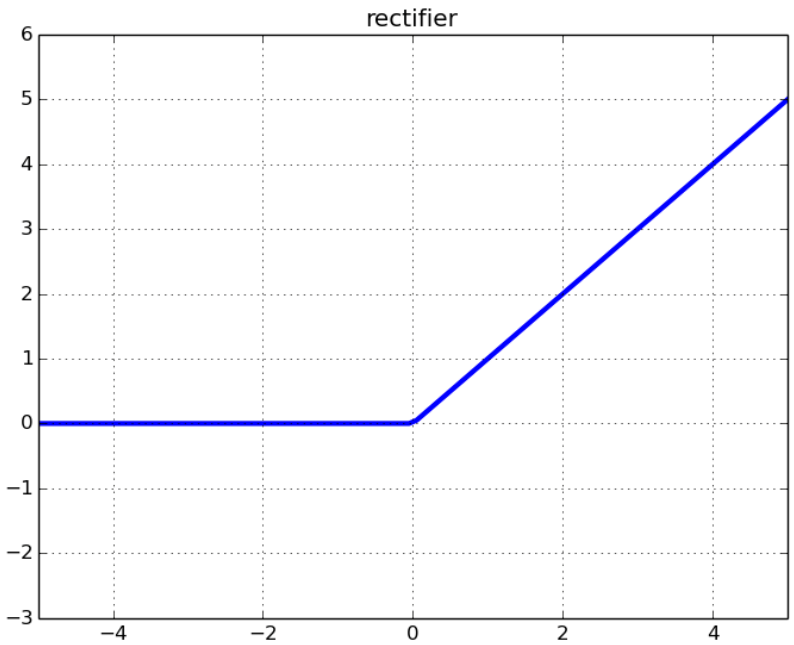
Single-Input *logsig* Neuron

Sigmoidal Activation Function

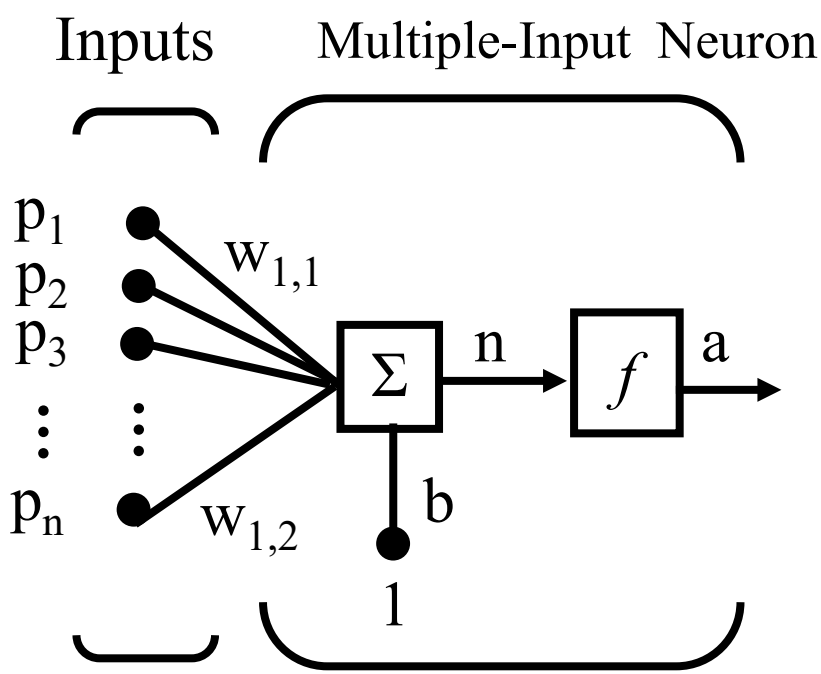


Rectified Linear Unit: ReLU

- *ReLU:* **Rectified Linear Unit**
- $ReLU = \max(0, x) = \begin{cases} 0, if\ x \leq 0 \\ x, if\ x > 0 \end{cases}$

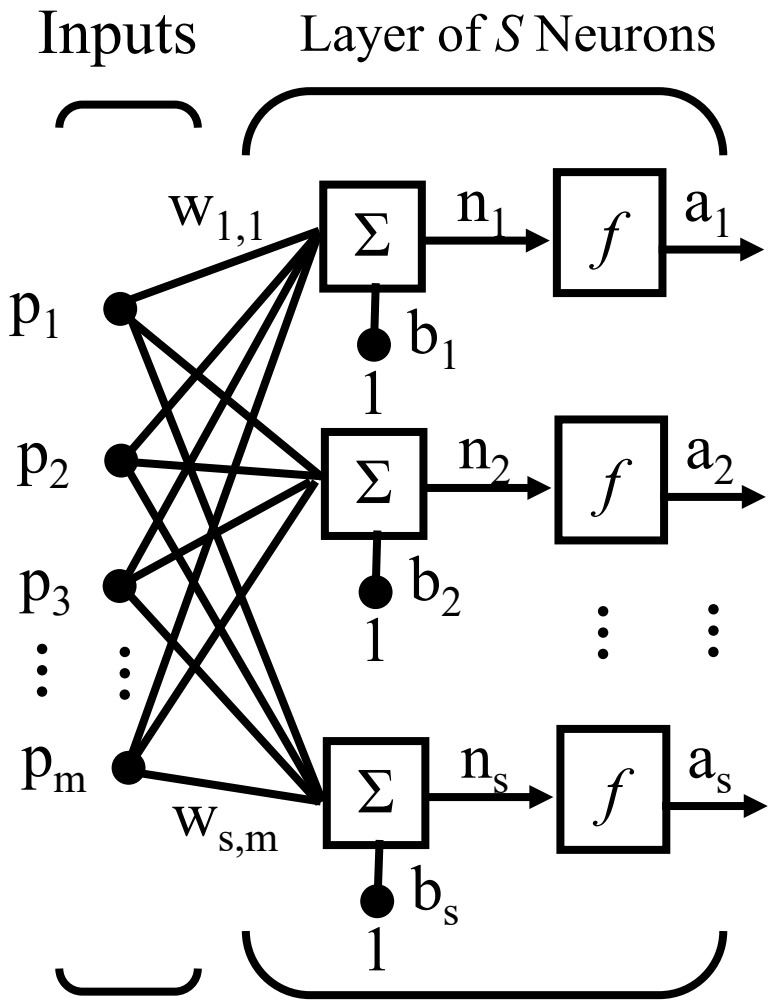


Multiple-Input Neuron



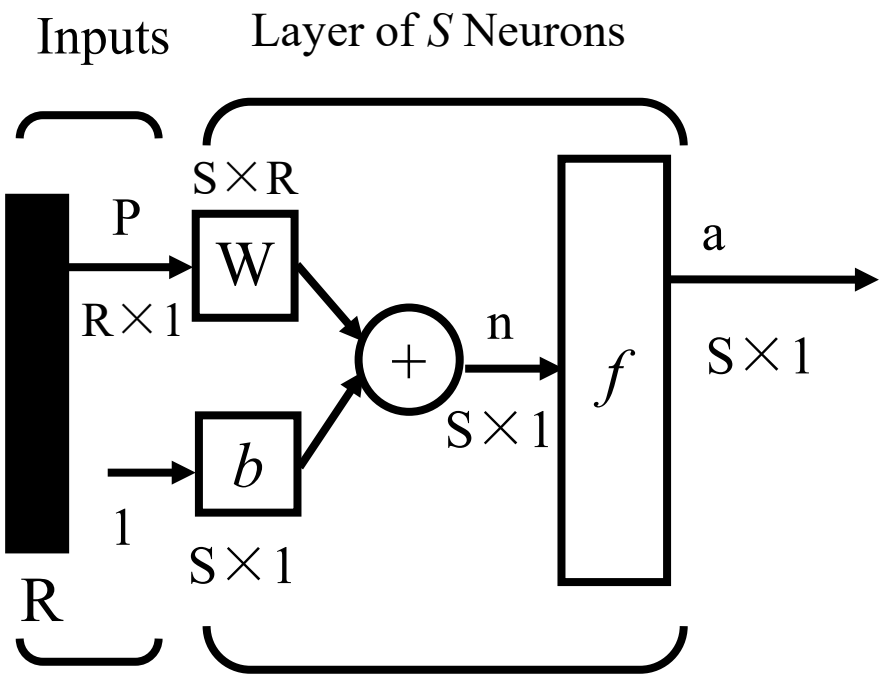
$a = f(Wp+b)$

Layer of Neurons



$a = f(Wp + b)$

Abbreviated Notation

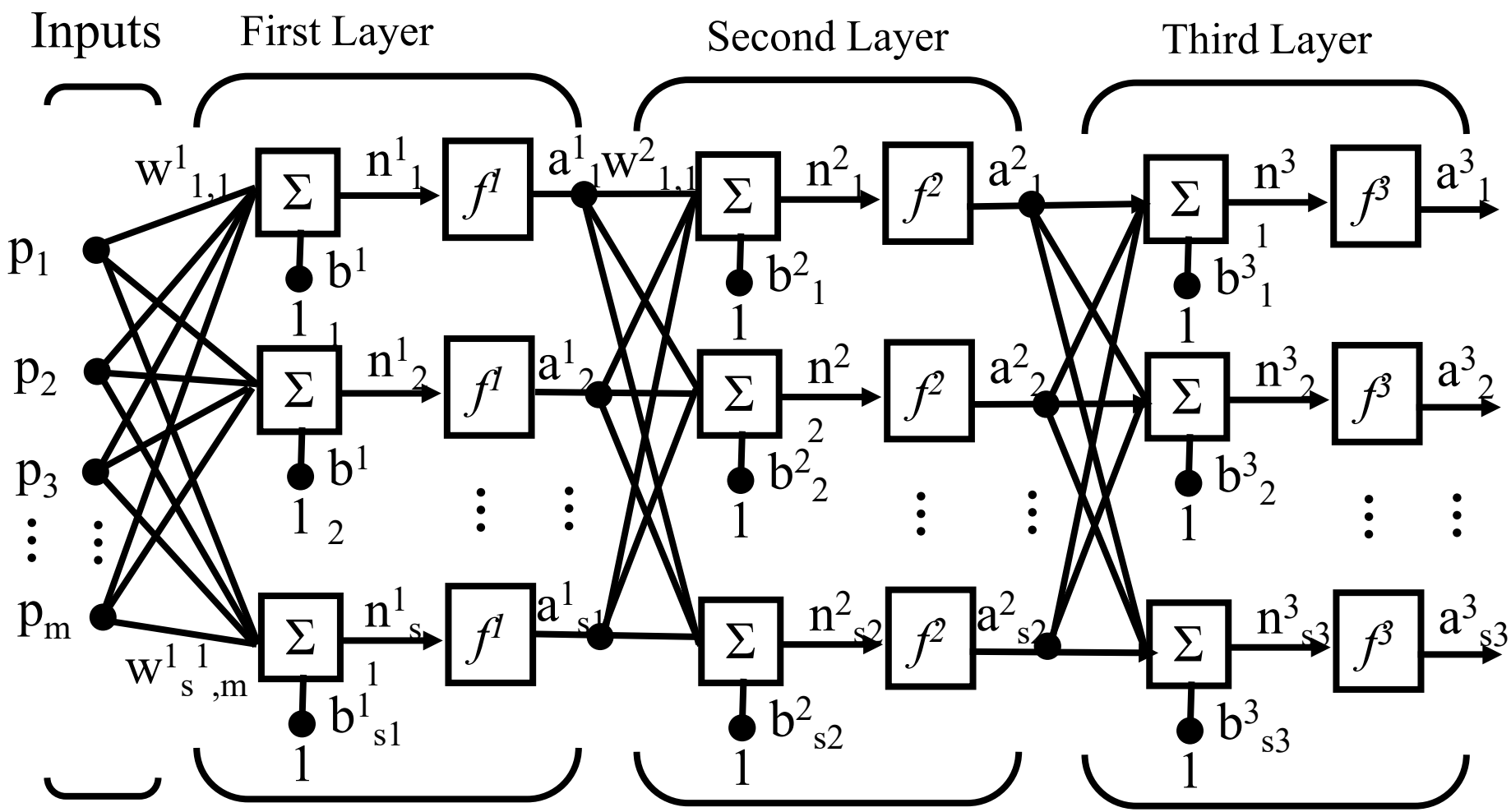


$$a = f(Wp + b)$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

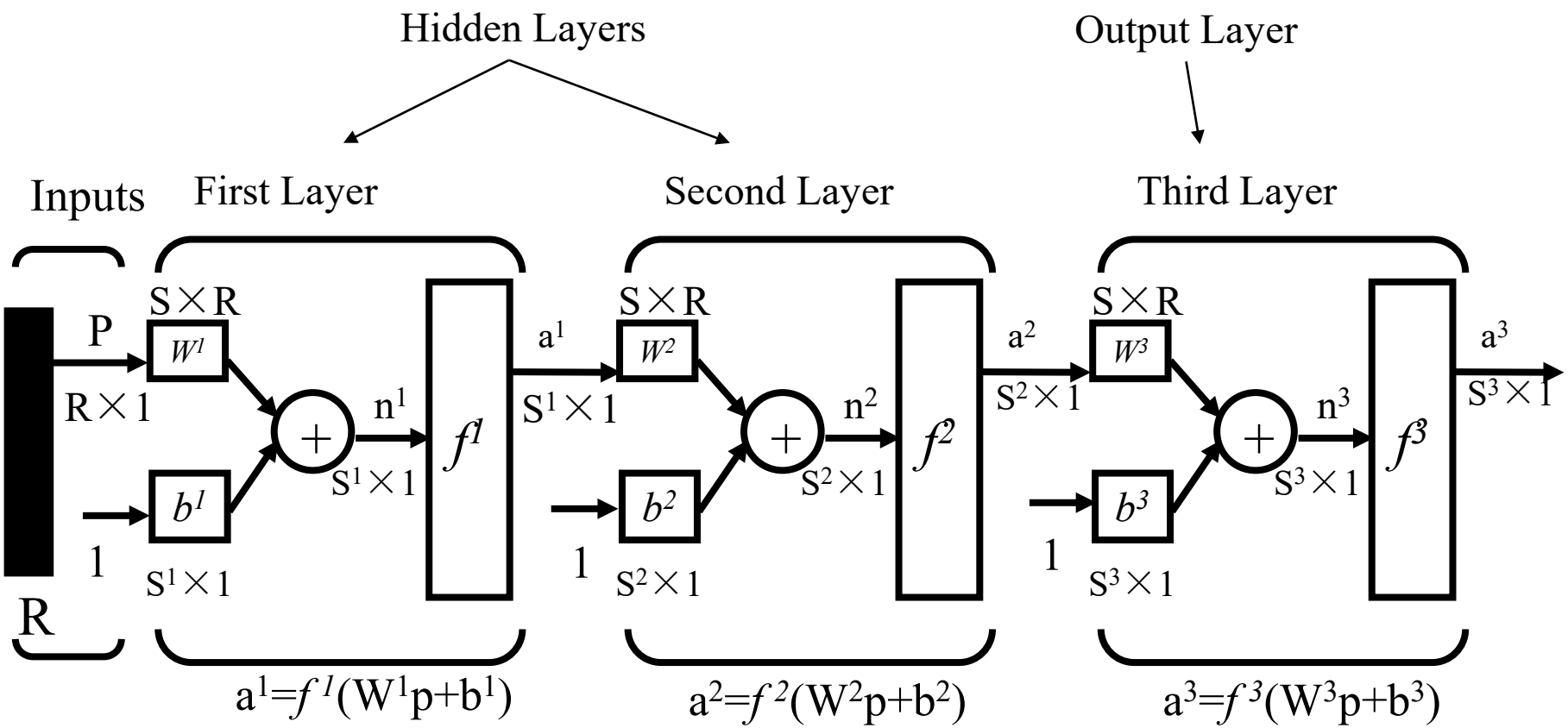
Multilayer Network



$a^1 = f^1 (W^1 p + b)$ $\quad$ $a^2 = f^2 (W^2 p + b)$ $\quad$ $a^3 = f^3 (W^3 p + b)$

$$a^3 = f^3 (W^3 f^2 (W^2 f^1 (W^1 p + b1) + b2) + b3)$$

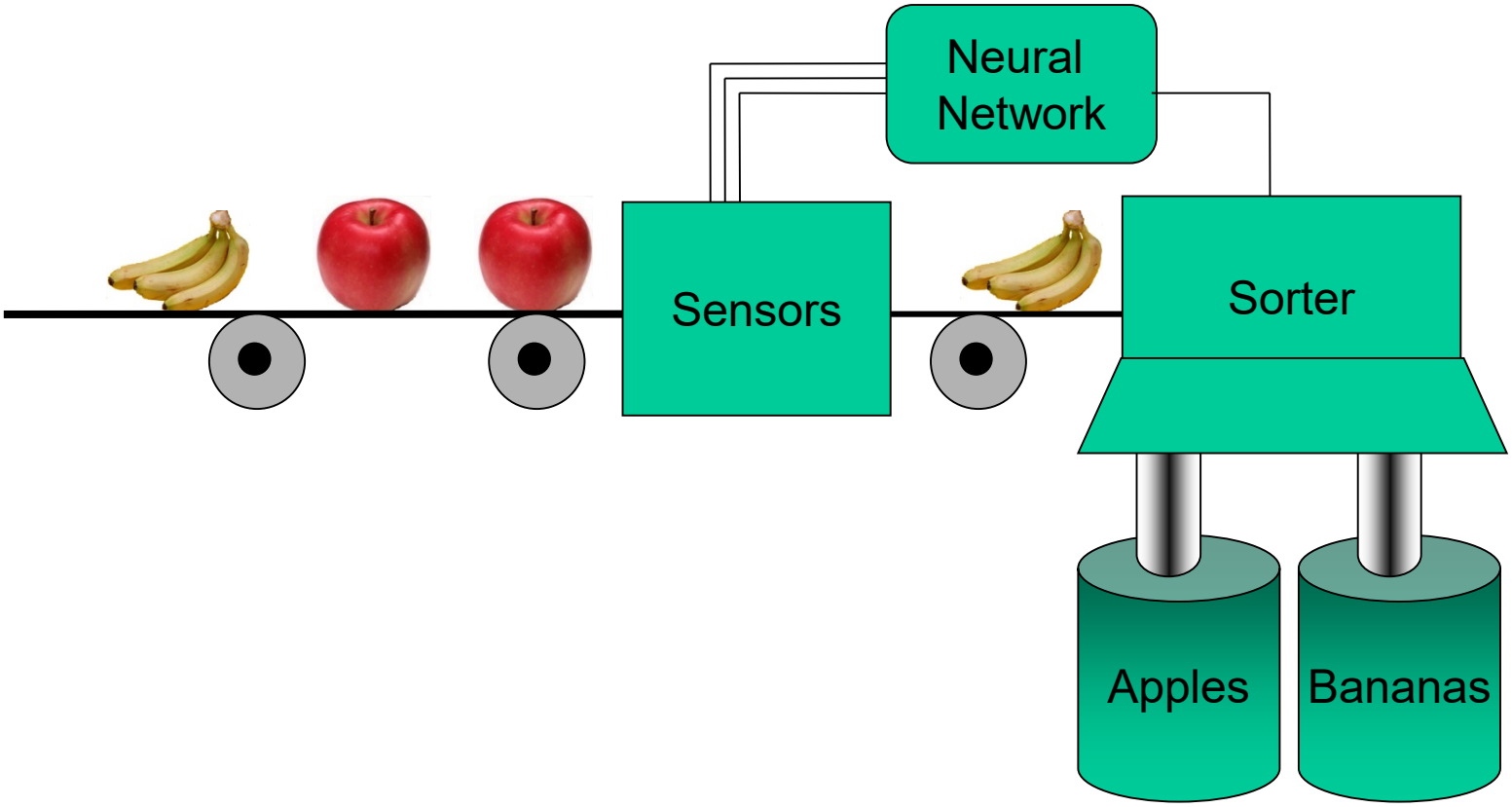
Abbreviated Notation



$$a^3 = f^3\,(W^3 f^2\,(W^2 f^1\,(W^1 p + b1) + b2) + b3)$$

An Illustrative Example

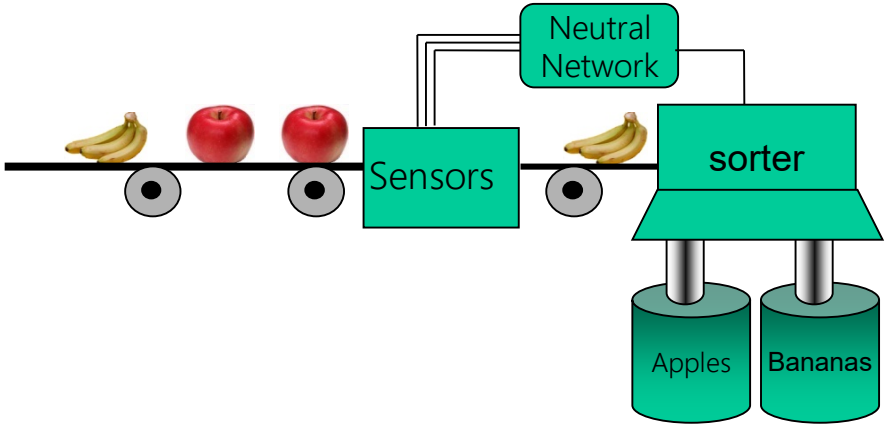
Apple/Banana Sorter



Prototype Vectors

Measurement Vector

$$\mathbf{p} = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix}$$



Shape: {1 : round ; -1 : eliptical}
Texture: {1 : smooth ; -1 : rough}
Weight: {1 : > 1 lb. ; -1 : < 1 lb.}

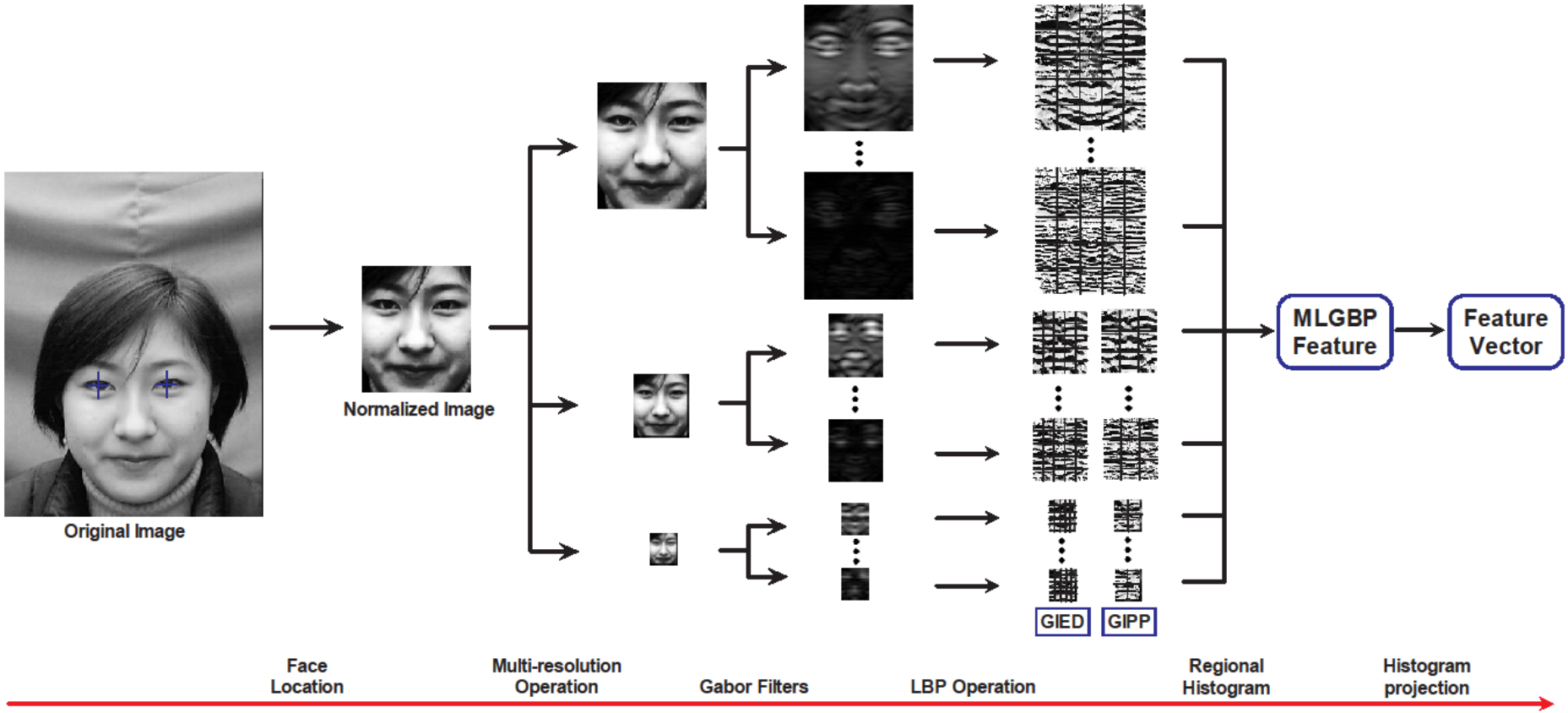
Prototype Banana

$$\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

Prototype Apple

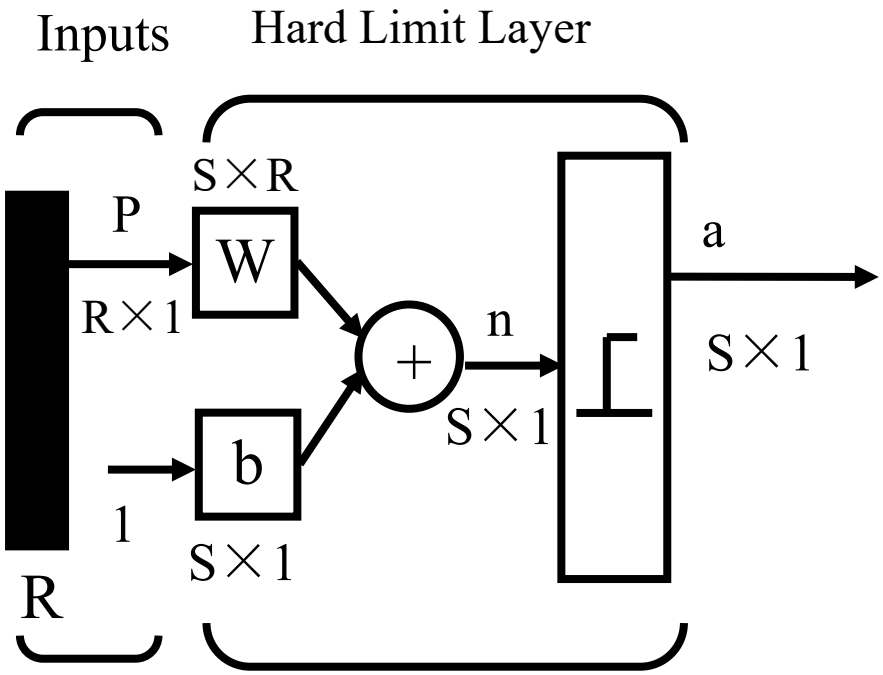
$$\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

Feature extraction vs Feature selection



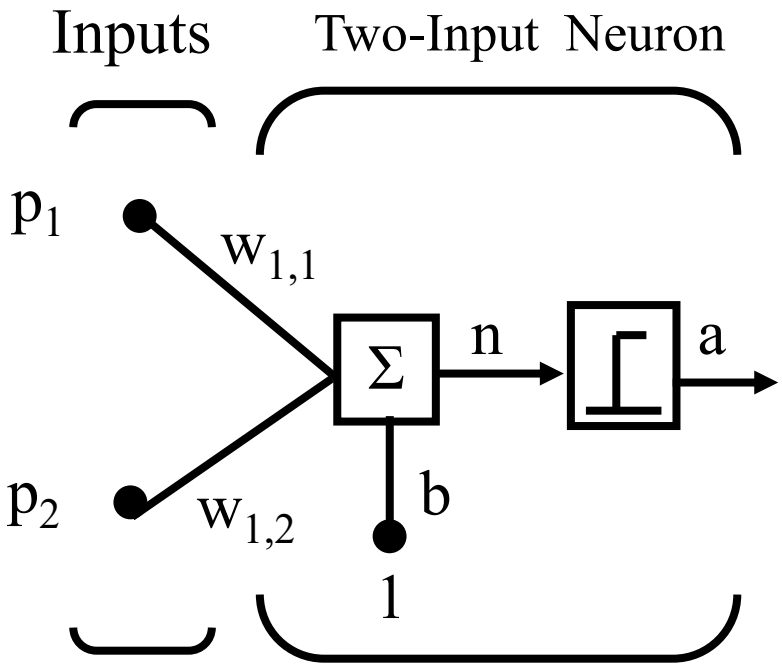
Process of extracting MLGBP

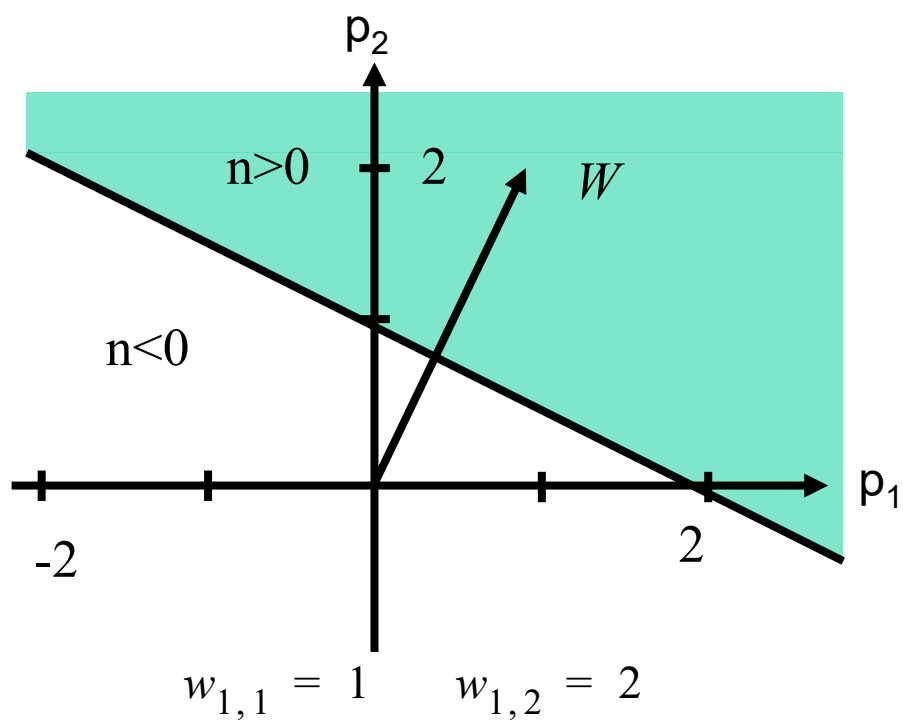
Perceptron



a=*hardlim* (Wp+b)

Two-Input Case





a=*hardlim* (Wp+b)

$$a = hardlims(n) = hardlims(\begin{bmatrix}1 & 2\end{bmatrix}\mathbf{p} + (-2))$$

Decision Boundary

$$\mathbf{W}\mathbf{p} + b = 0 \qquad \begin{bmatrix}1 & 2\end{bmatrix}\mathbf{p} + (-2) = 0$$

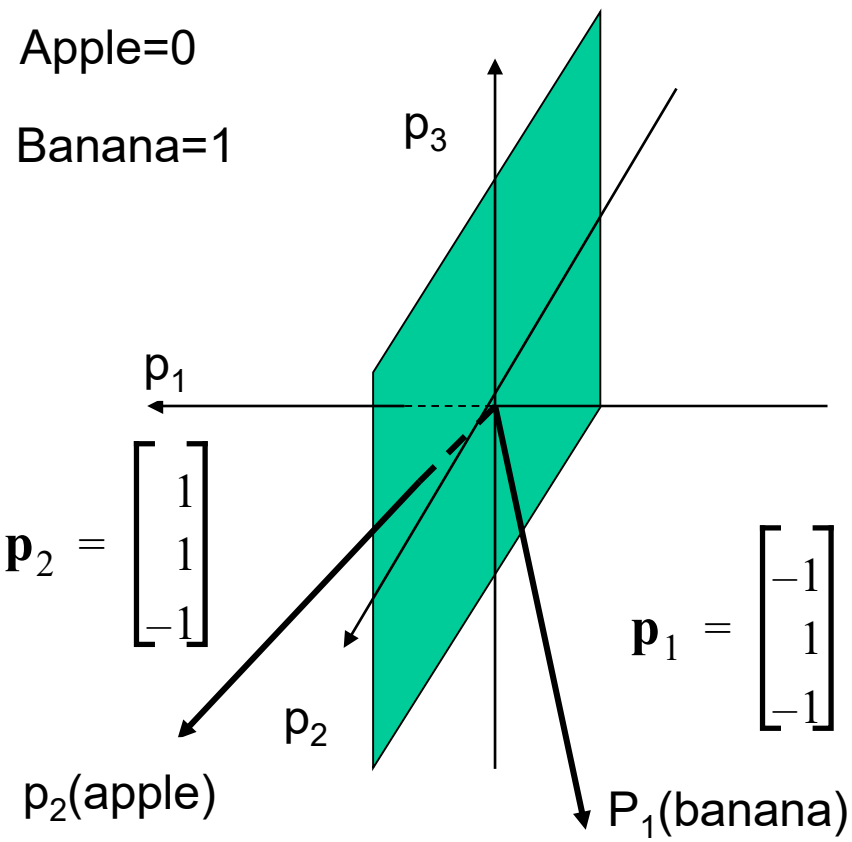
Hyperplane

Apple/Banana Example

$$a = hardlims\left(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + b\right)$$

Apple=0

Banana=1



$$\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \qquad \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

The decision boundary should separate the prototype vectors.

$$p_1 = 0$$

The weight vector should be orthogonal to the decision boundary, and should point in the direction of the vector which should produce an output of 1.

The bias determines the position of the boundary

$$\begin{bmatrix} -1 & 0 & 0 \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + 0 = 0$$

Testing the Network

Measurement Vector

$$\mathbf{p} = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix}$$

Shape: {1 : round ; -1 : eliptical}
Texture: {1 : smooth ; -1 : rough}
Weight: {1 : > 1 lb. ; -1 : < 1 lb.}

Banana:

$$a = hardlims\left(\begin{bmatrix} -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} + 0 \right) = 1 (\text{banana})$$

Apple:

$$a = hardlims\left(\begin{bmatrix} -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0 \right) = -1 \ (\text{apple})$$

"Rough" Banana:

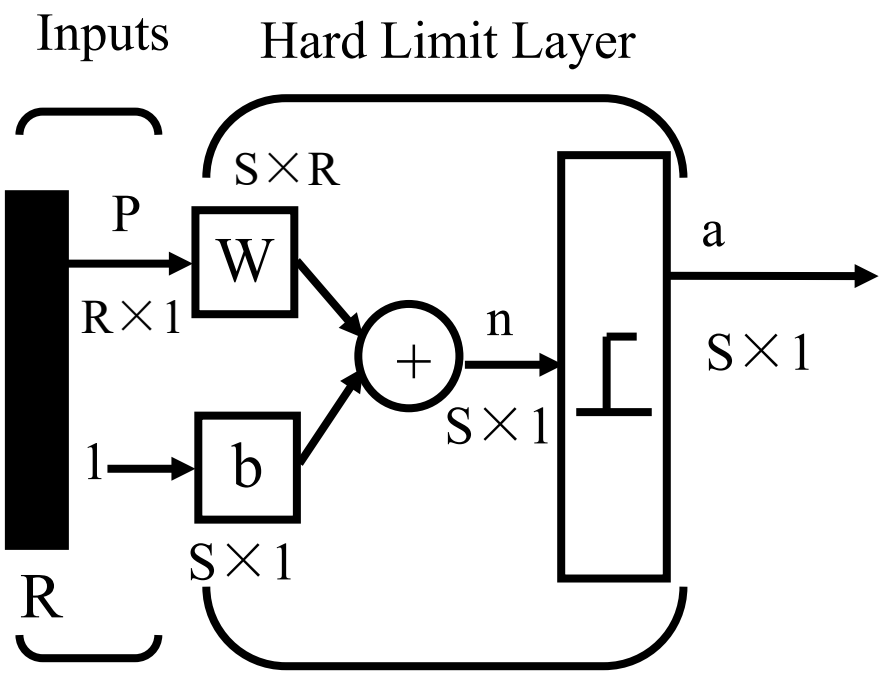
$$a = hardlims\left(\begin{bmatrix} -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + 0 \right) = 1 (\text{banana})$$

Summary

- Perceptron
  - Feedforward Network
  - Linear Decision Boundary
  - One Neuron for Each Decision

Perceptron Learning Rule

Perceptron Architecture



a=*hardlim* (Wp+b)

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

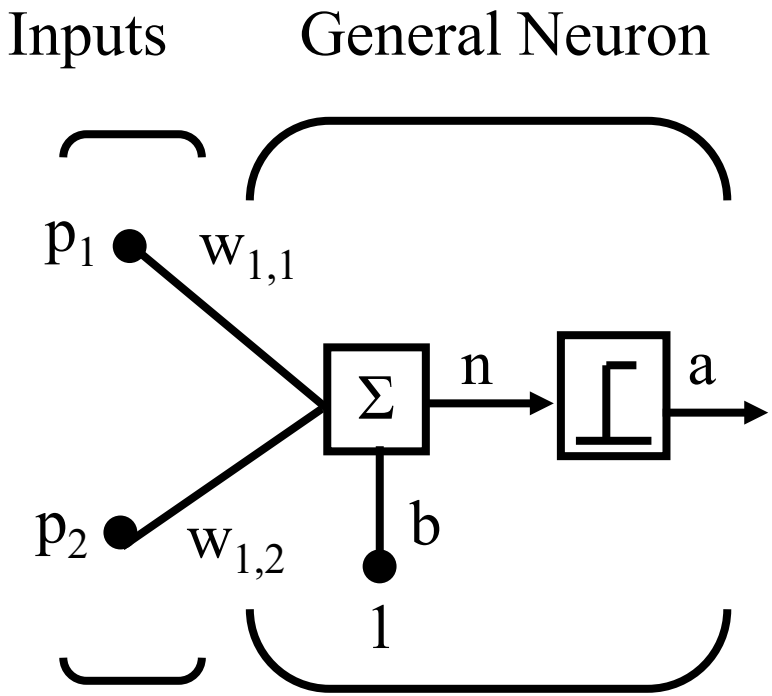
$$_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}$$

*i*th row of W

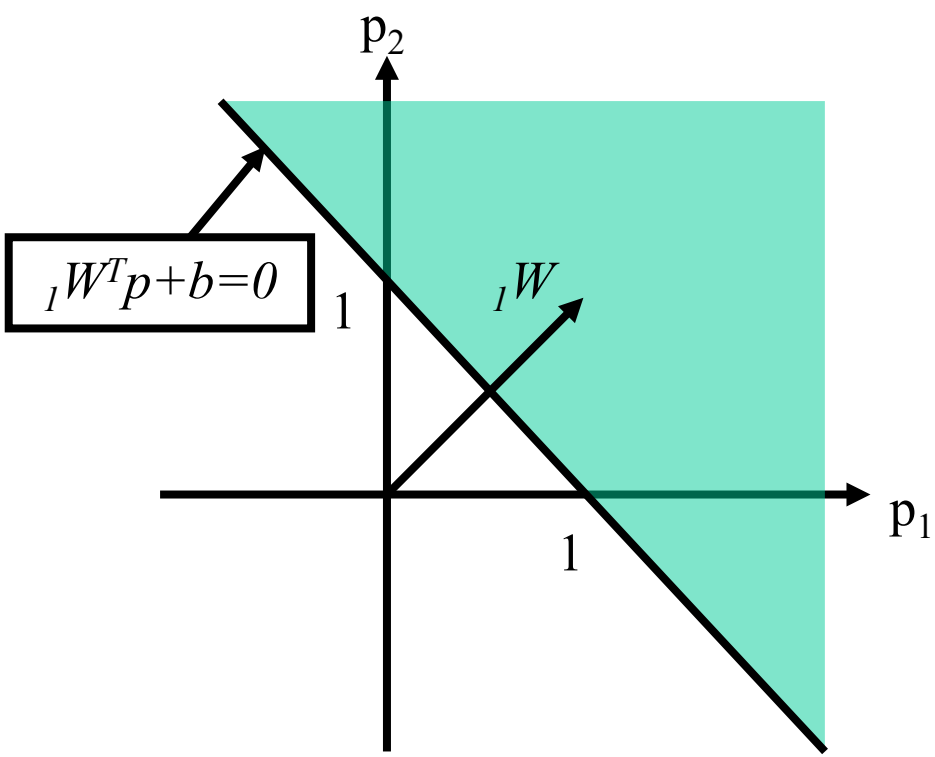
$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i)$$

Single-Neuron Perceptron

$w_{1,1} = 1 \qquad w_{1,2} = 1 \qquad b = -1$



a=*hardlim* (Wp+b)

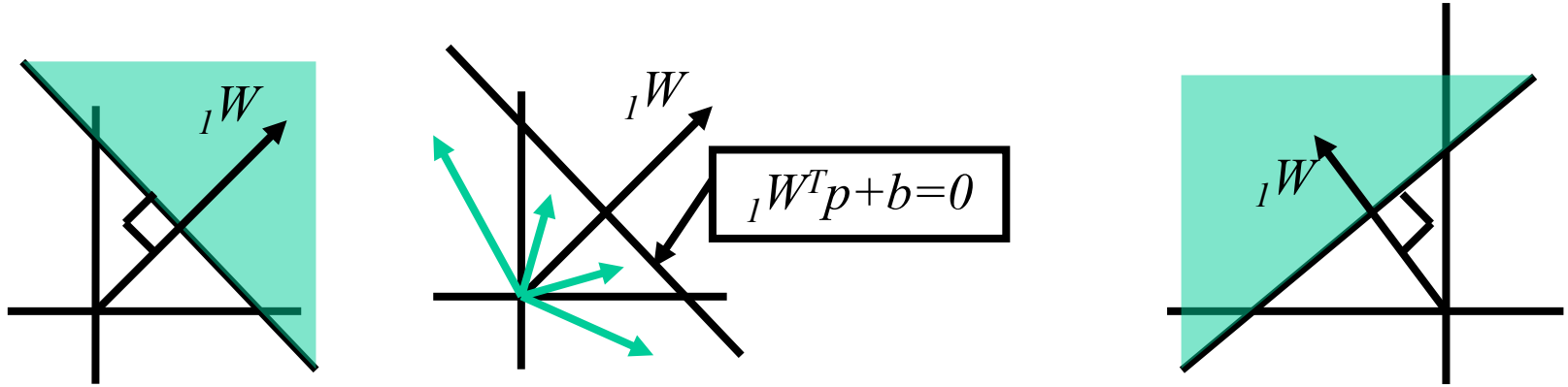


$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

Decision Boundary

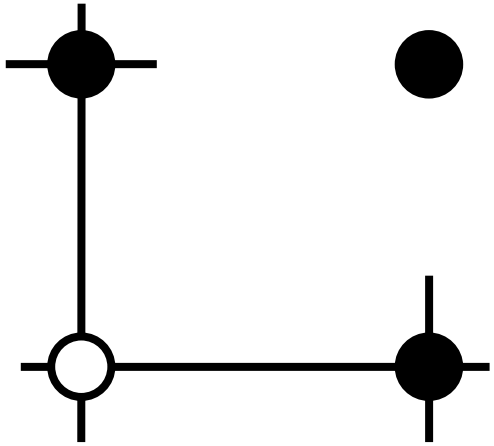
$$ {}_1\mathbf{w}^{\mathrm{T}}\mathbf{p} + b = 0 \qquad\qquad {}_1\mathbf{w}^{\mathrm{T}}\mathbf{p} = -b $$

- All points on the decision boundary have the same inner product with the weight vector.
- Therefore they have the same projection onto the weight vector, and they must lie on a line orthogonal to the weight vector

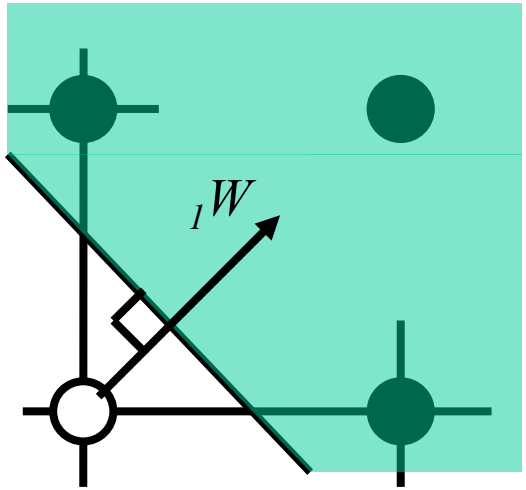


Example - OR

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \quad \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \quad \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}$$



OR Solution



Weight vector should be orthogonal to the decision boundary.

$$ {}_1\mathbf{w} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} $$

Pick a point on the decision boundary to find the bias.

$$ {}_1\mathbf{w}^{\mathrm{T}}\mathbf{p} + b = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} + b = 0.25 + \mathrm{b} = 0 \qquad \Rightarrow \qquad b = -0.25 $$

Multiple-Neuron Perceptron

Each neuron will have its own decision boundary.

$$ {}_i\mathbf{w}^T\mathbf{p} + b_i = 0 $$

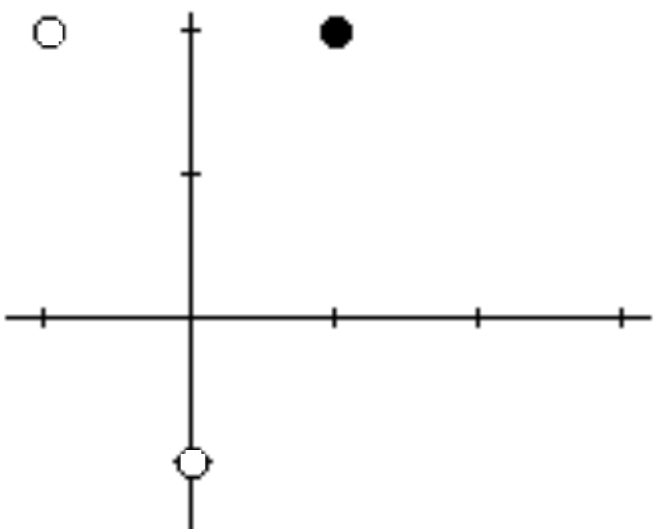
A single neuron can classify input vectors into two categories.

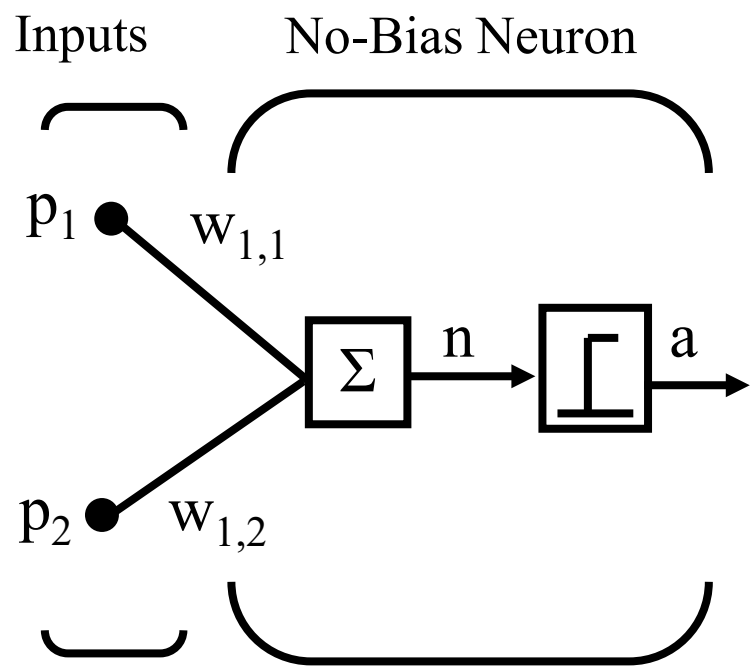
A multi-neuron perceptron can classify input vectors into 2^S categories.

Learning Rule Test Problem

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\} \quad \left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}$$



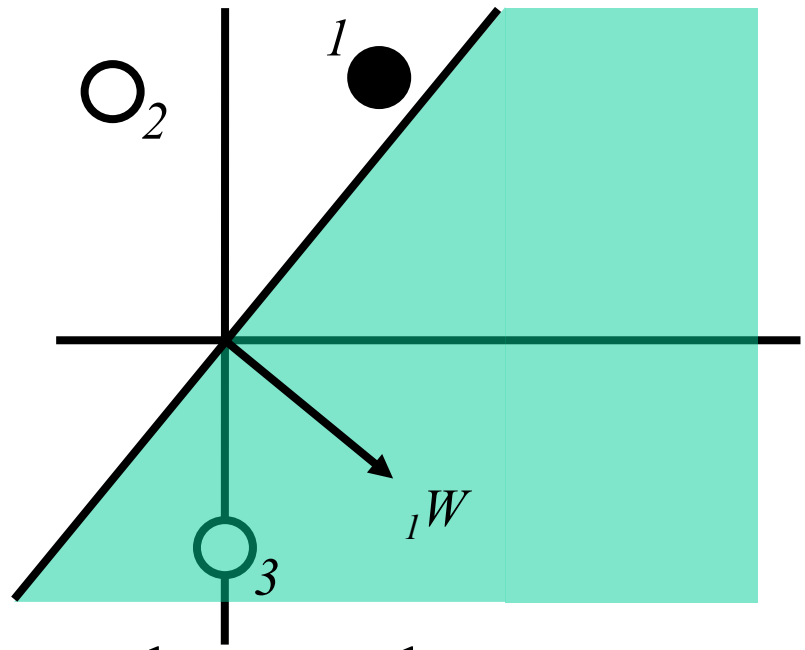


a=*hardlim* (Wp)

Starting Point

Random initial weight:

$$ {}_1\mathbf{w} = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} $$



$$ \left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_1 = 1 \right\} $$

Present $\mathbf{p}_1$ to the network:

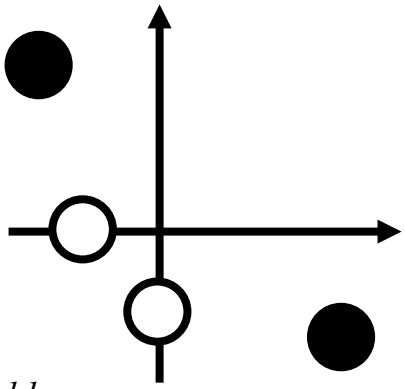
$$ a = hardlim({}_1\mathbf{w}^T \mathbf{p}_1) = hardlim\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) $$

$$ a = hardlim(-0.6) = 0 $$

Incorrect Classification.

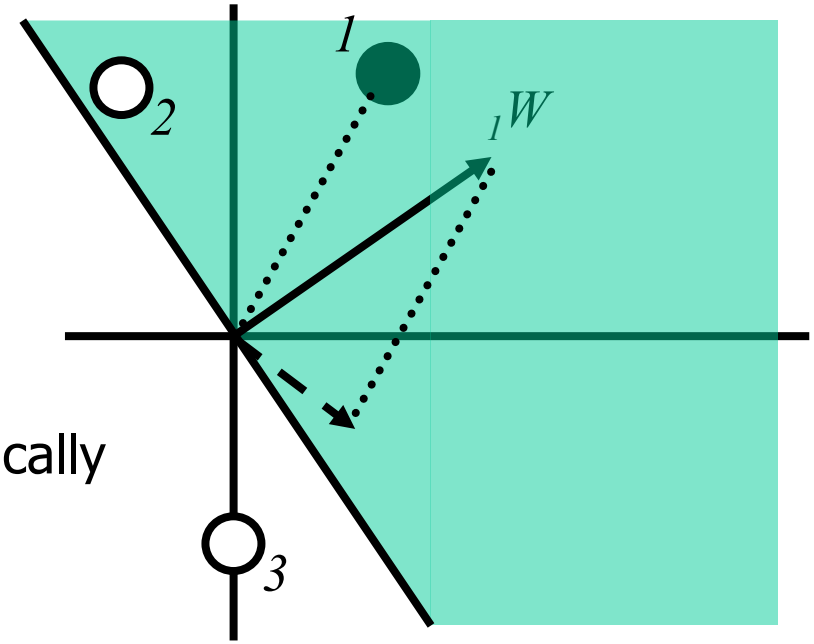
Tentative Learning Rule

- Set ${}_1\mathbf{w}$ to $\mathbf{p}_1$ ✗
  - Not stable
- Add $\mathbf{p}_1$ to ${}_1\mathbf{w}$ ✓



Tentative Rule: If $t = 1$ and $a = 0$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix}$$



Repeated presentations of p_1 would cause the direction of ${}_1W$ to asymptotically approach the direction of p_1

Second Input Vector

$$\left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0 \right\}$$

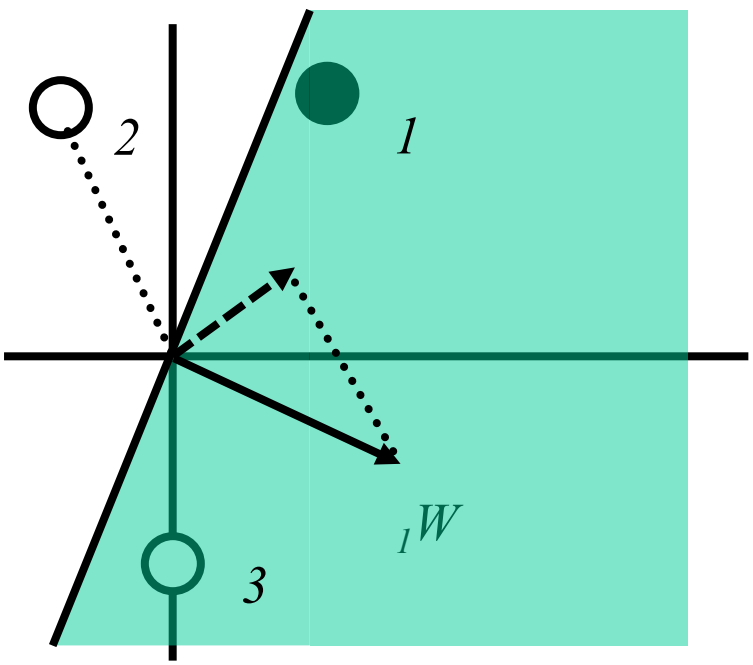
$$a = hardlim({}_1\mathbf{w}^{\mathrm{T}}\mathbf{p}_2) = hardlim\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right)$$

$$a = hardlim(0.4) = 1$$ (Incorrect Classification)

Modification to Rule: $\text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}$$

Since we would like to move the weight vector ${}_1W$ away from the input, we can simply change the addition to subtraction



Third Input Vector

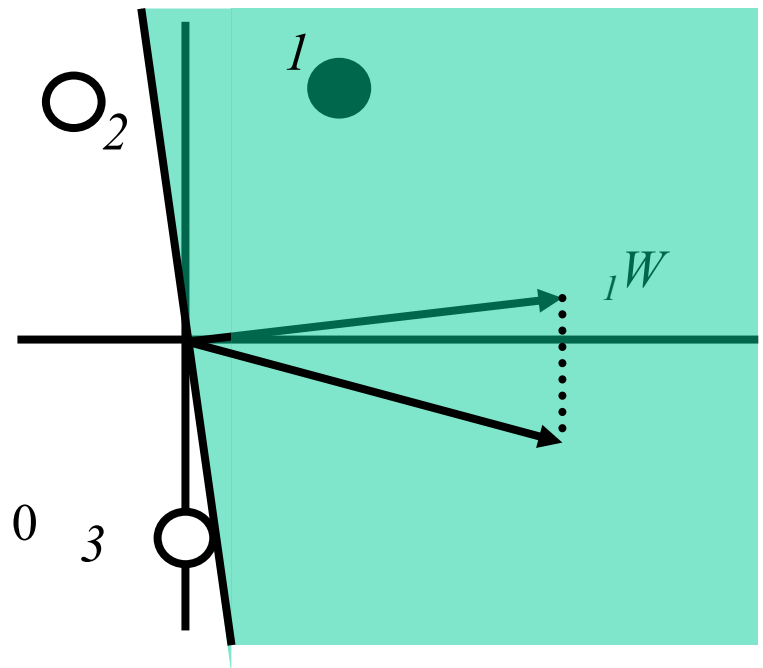
$$\left\{ \mathbf{p}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, t_3 = 0 \right\}$$

$$a = hardlim({}_1\mathbf{w}^T \mathbf{p}_3) = hardlim\left(\begin{bmatrix} 3.0 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right)$$

$$\mathrm{a} = \mathrm{hardlim}(0.8) = 1$$

(Incorrect Classification)

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}$$



$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_1 = 1 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, t_3 = 0 \right\}$$

$$a = hard\lim({}_1W^T P_3) = had\lim([3.0\ 0.2]\begin{bmatrix} 0 \\ -1 \end{bmatrix}) = 0$$

Patterns are now correctly classified.

If $t = a$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$.

Unified Learning Rule

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$$

$$\text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$$

$$\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$$

$$e = t - a$$

$$\text{If } e = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$$

$$\text{If } e = -1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$$

$$\text{If } e = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$$

$$\boxed{\begin{aligned} {}_1\mathbf{w}^{new} &= {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p} \\ b^{new} &= b^{old} + e \end{aligned}}$$

A bias is a weight with an input of 1.

Multiple-Neuron Perceptrons

To update the *i*th row of the weight matrix:

$$ {}_i\mathbf{w}^{new} = {}_i\mathbf{w}^{old} + e_i\mathbf{p} $$

$$ b_i^{new} = b_i^{old} + e_i $$

Matrix form:

$$ \boxed{\begin{aligned} \mathbf{W}^{new} &= \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T \\ \mathbf{b}^{new} &= \mathbf{b}^{old} + \mathbf{e} \end{aligned}} $$

Apple/Banana Example

Training Set

Prototype Banana $\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \end{bmatrix} \right\}$ $\left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \end{bmatrix} \right\}$ Prototype Apple

Initial Weights

$$\mathbf{W} = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \qquad b = 0.5$$

First Iteration

$$a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} + 0.5 \right)$$

$$a = hardlim(-0.5) = 0 \qquad e = t_1 - a = 1 - 0 = 1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (1)\begin{bmatrix} -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = 0.5 + (1) = 1.5$$

Second Iteration

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \end{bmatrix} \right\} \qquad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \end{bmatrix} \right\}$$

$$a = hardlim\,(\mathbf{W}\mathbf{p}_2 + b) = hardlim\,(\begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (1.5))$$

$$a = hard\lim(2.5) = 1$$

$$e = t_2 - a = 0 - 1 = -1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & -1.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = 1.5 + (-1) = 0.5$$

Check

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 1 \end{bmatrix} \right\} \qquad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \end{bmatrix} \right\}$$

$$a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} + 0.5\right)$$

$$a = hardlim(1.5) = 1 = t_1$$

$$a = hardlim(\mathbf{W}\mathbf{p}_2 + b) = hardlim\left(\begin{bmatrix} -1.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0.5\right)$$

$$a = hardlim(-1.5) = 0 = t_2$$

Perceptron Rule Capability

The perceptron rule will always converge to weights which accomplish the desired classification, assuming that such weights exist.

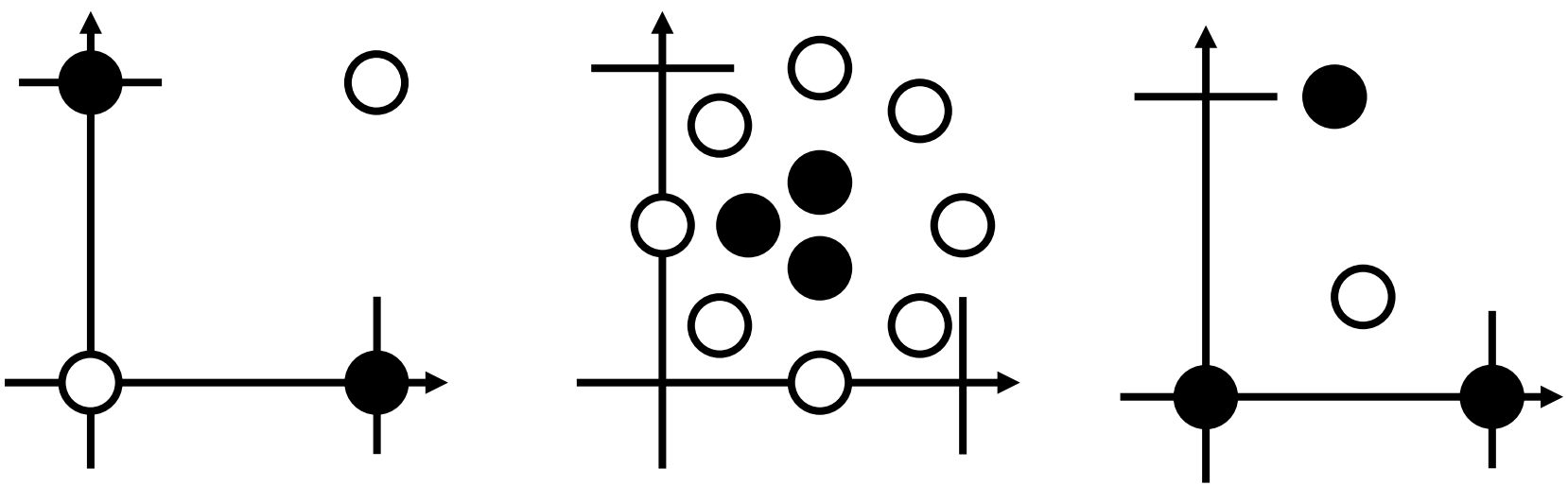
Proof of Convergence

Perceptron Limitations

Linear Decision Boundary

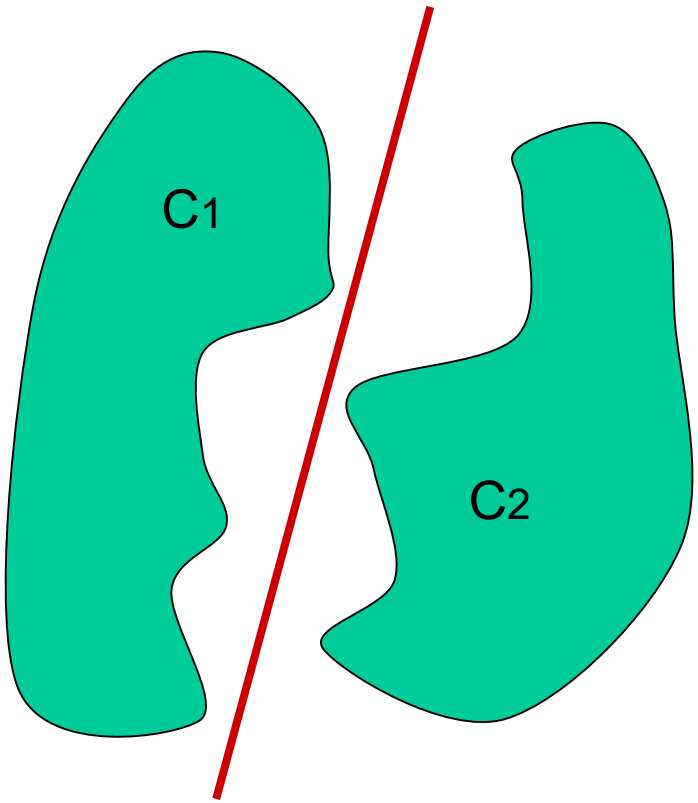
$$_1\mathbf{w}^T\mathbf{p} + b = 0$$

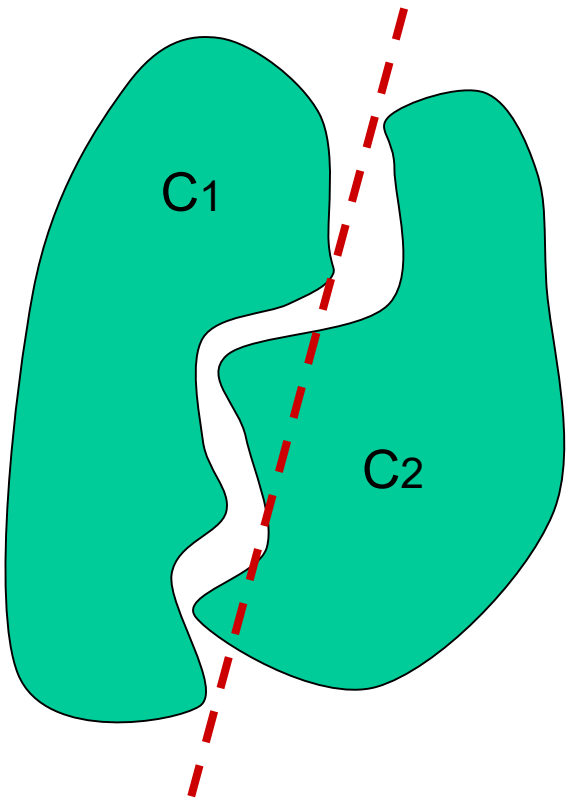
Non-linearly Separable Problems



Linear separable and non-separable problems

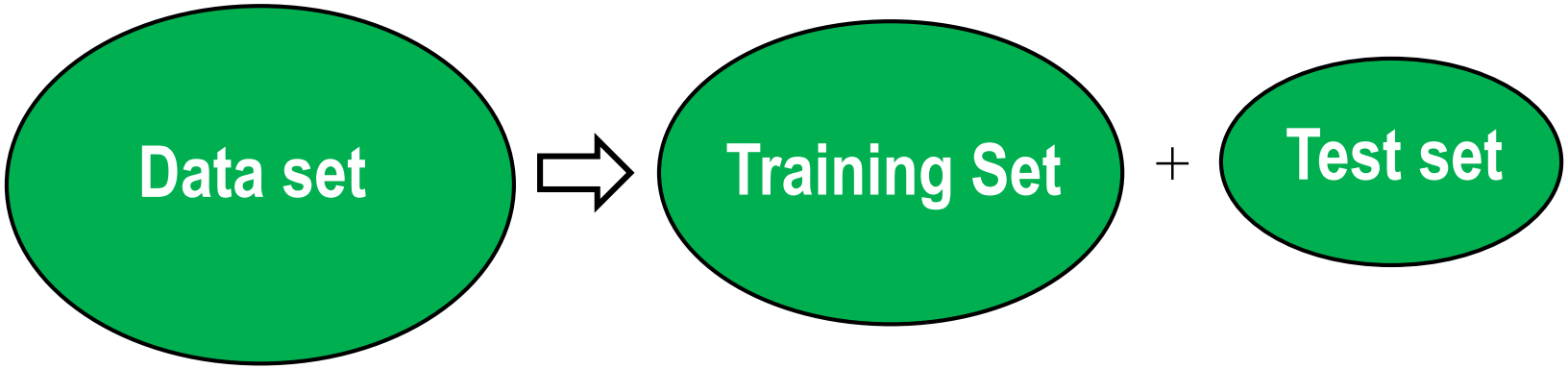
- Linear Separable Problem
- Linear Non-separable Problem





Two phases of supervised learning

- Training phase
  - Training data set
- Test phase or recognition phase
  - Test data set
- Generalization performance

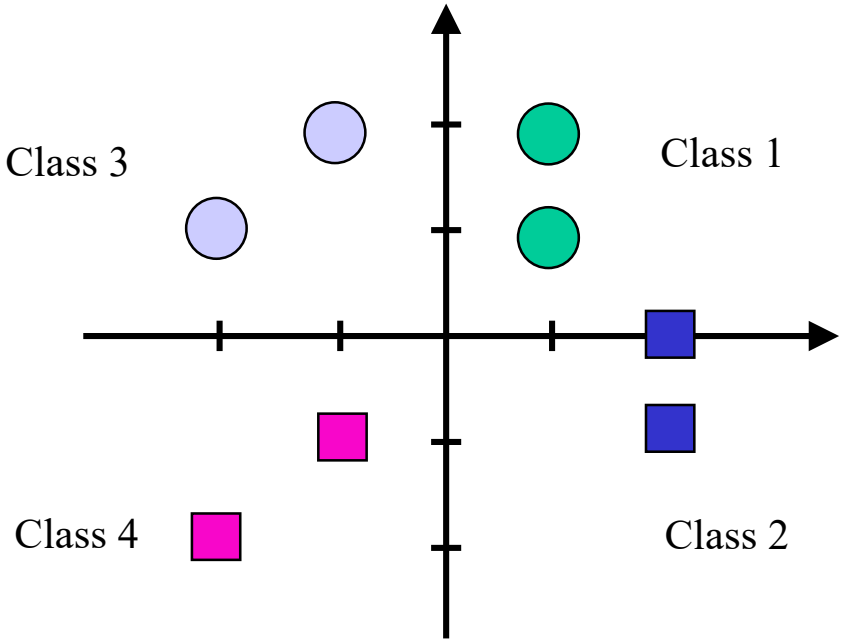


$$T_{All} = T_{Train} \bigcup T_{Test} \; ; \quad T_{Train} \bigcap T_{Test} = \Phi$$

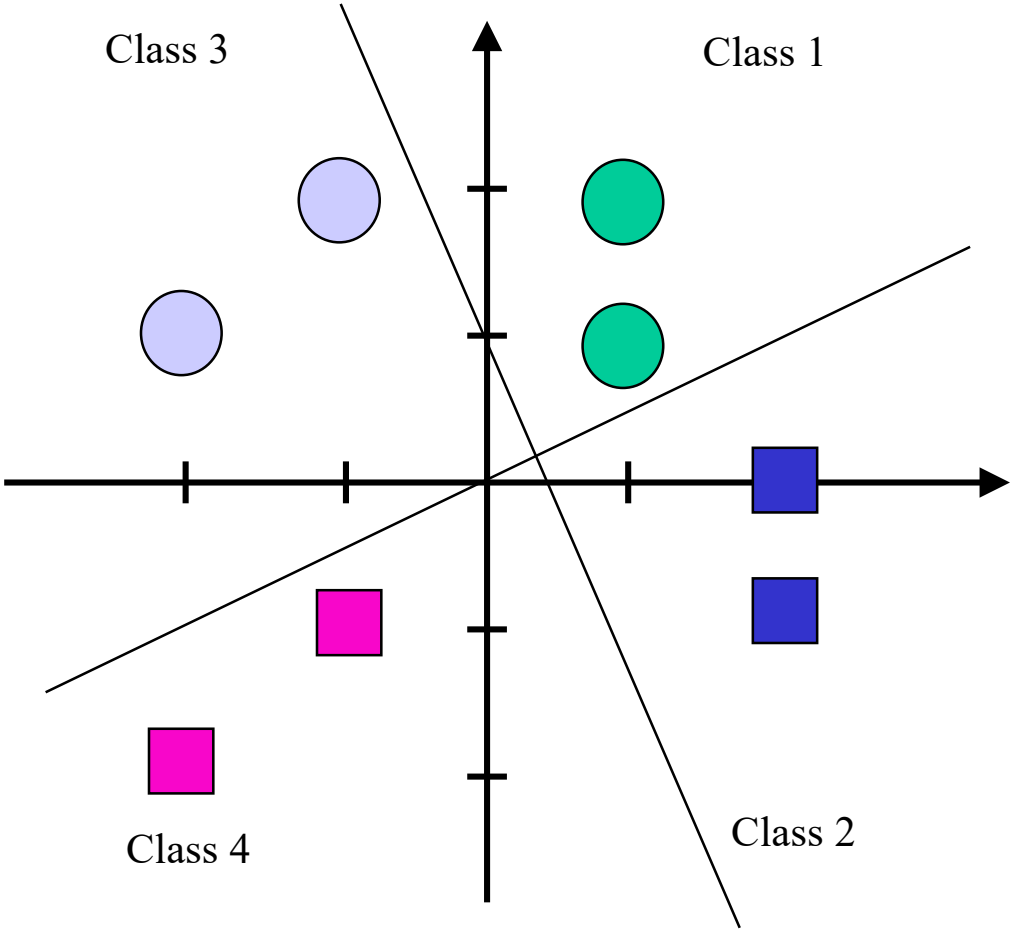
Four-Class Problem: Training Set

$$\text{Class 1}: \left\{ p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \; p_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\}, \qquad \text{Class 2}: \left\{ p_3 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \; p_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\},$$

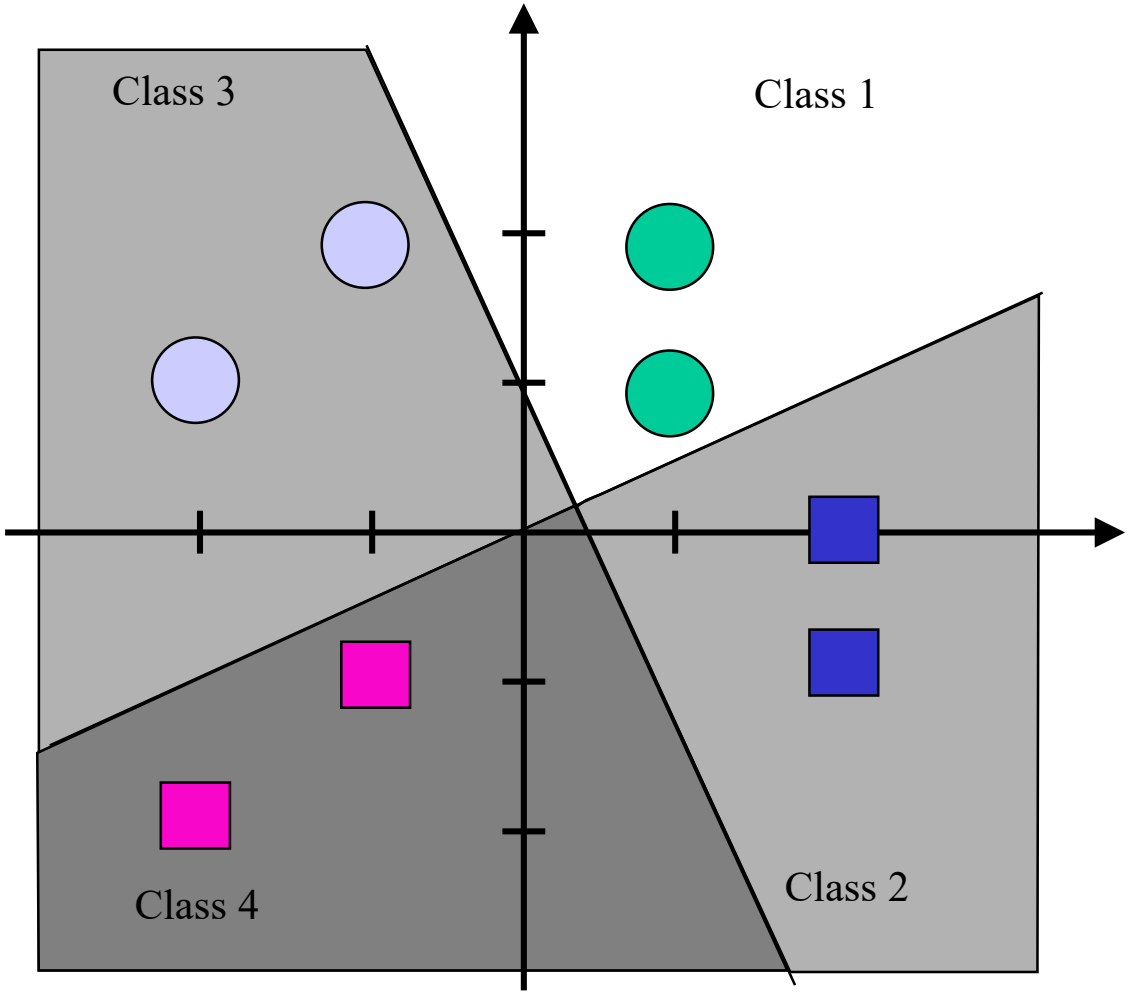
$$\text{Class 3}: \left\{ p_5 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \; p_6 = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \right\}, \quad \text{Class 4}: \left\{ p_7 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \; p_8 = \begin{bmatrix} -2 \\ -2 \end{bmatrix} \right\}.$$



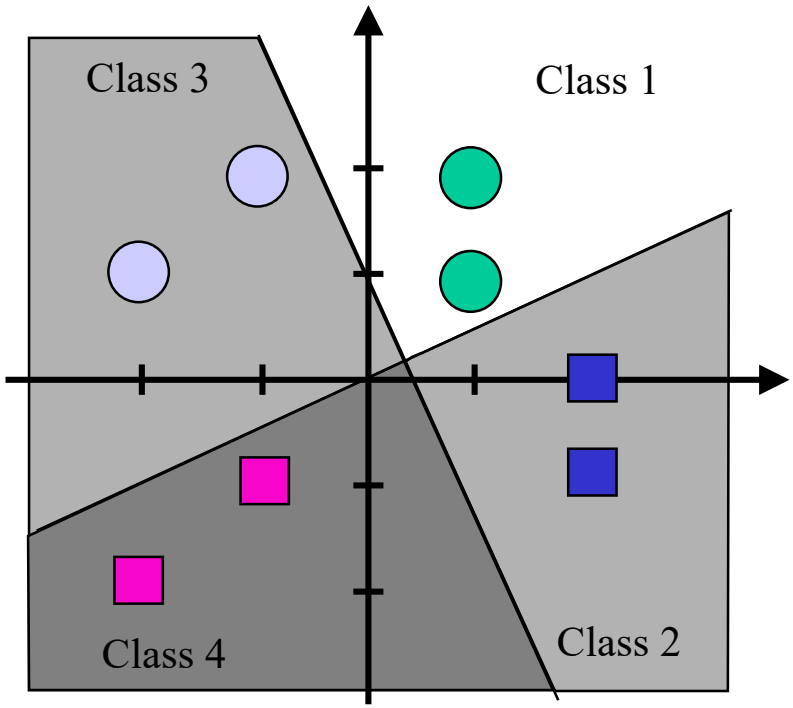
Tentative Decision Boundaries



Decision Regions



Target Representation



$$\text{Class 1}: \left\{ t_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ t_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}, \quad \text{Class 2}: \left\{ t_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ t_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\},$$

$$\text{Class 3}: \left\{ t_5 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ t_6 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \quad \text{Class 4}: \left\{ t_7 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ t_8 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Solution

$$ {}_1W = \begin{bmatrix} -3 \\ -1 \end{bmatrix} \text{ and } {}_2W = \begin{bmatrix} 1 \\ -2 \end{bmatrix} $$

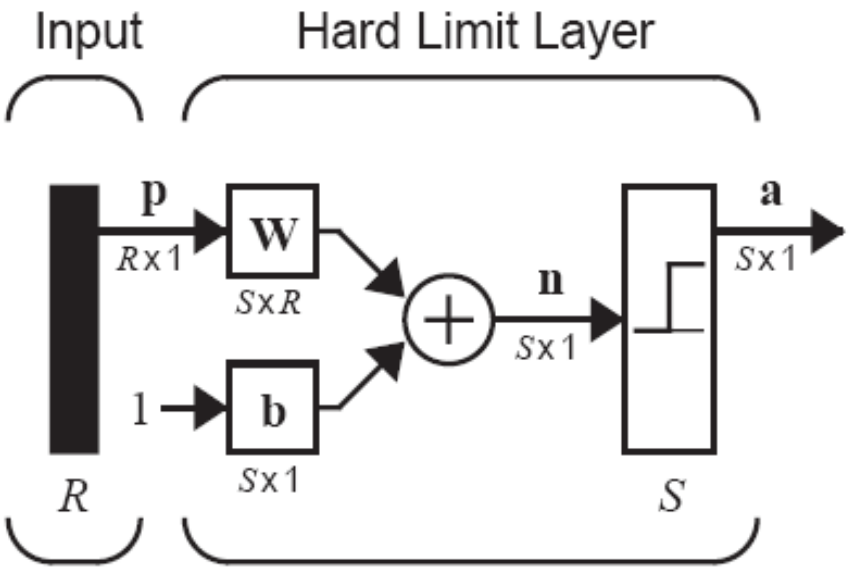
$$ b_1 = -{}_1W^T p = -\begin{bmatrix} -3 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1 $$

$$ b_2 = -{}_2W^T p = -\begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0 $$

$$ W = \begin{bmatrix} {}_1W^T \\ {}_2W^T \end{bmatrix} = \begin{bmatrix} -3 & -1 \\ 1 & -2 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1 \\ 0 \end{bmatrix} $$

Summary

- Perceptron Architecture



$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp}+\mathbf{b})$$

a = hardlim(Wp + b)

$$W = \begin{bmatrix} {}_1w^T \\ {}_2w^T \\ \vdots \\ {}_Sw^T \end{bmatrix}$$

$$a_i = hard\lim(n_i) = hard\lim({}_iw^T p + b_i)$$

Summary (2)

- Decision Boundary

$$ {}_i w^T p + b_i = 0 $$

The decision boundary is always orthogonal to the weight vector.
Single-layer perceptrons can only classify linearly separable vectors.

- Perceptron Learning Rule

$$ W^{new} = W^{old} + ep^T $$

$$ b^{new} = b^{old} + e $$

$$ \text{where } e = t - a $$

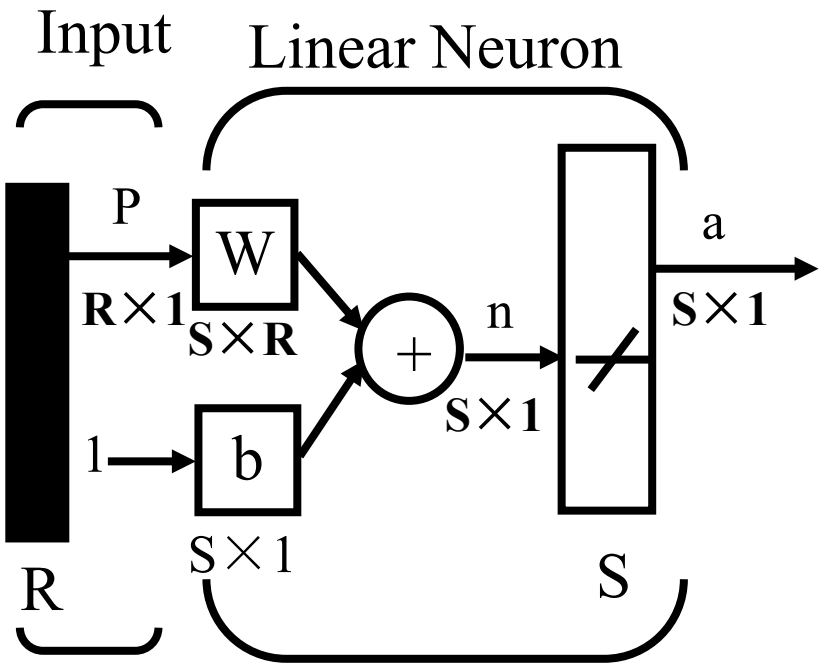
Least Mean Square Algorithm

Widrow–Hoff Learning

(Least Mean Square Algorithm)

最小均方算法

ADALINE (ADAptive LInear NEuron) Network



$$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp} + \mathbf{b}) = \mathbf{Wp} + \mathbf{b}$$

a=purelin(Wp+b)

$$a_i = purelin(n_i) = purelin({}_i\mathbf{w}^T\mathbf{p} + b_i) = {}_i\mathbf{w}^T\mathbf{p} + b_i$$

$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

Taylor Series Expansion

$$F(x) = F(x^*) + \frac{d}{dx}F(x)\Big|_{x=x^*}(x - x^*)$$

$$+ \frac{1}{2}\frac{d^2}{dx^2}F(x)\Big|_{x=x^*}(x - x^*)^2 + \cdots$$

$$+ \frac{1}{n!}\frac{d^n}{dx^n}F(x)\Big|_{x=x^*}(x - x^*)^n + \cdots$$

Example

$$F(x) = e^{-x}$$

Taylor series of $F(x)$ about $x^* = 0$:

$$F(x) = e^{-x} = e^{-0} - e^{-0}(x-0) + \frac{1}{2}e^{-0}(x-0)^2 - \frac{1}{6}e^{-0}(x-0)^3 + \ldots$$

$$F(x) = 1 - x + \frac{1}{2}x^2 - \frac{1}{6}x^3 + \ldots$$

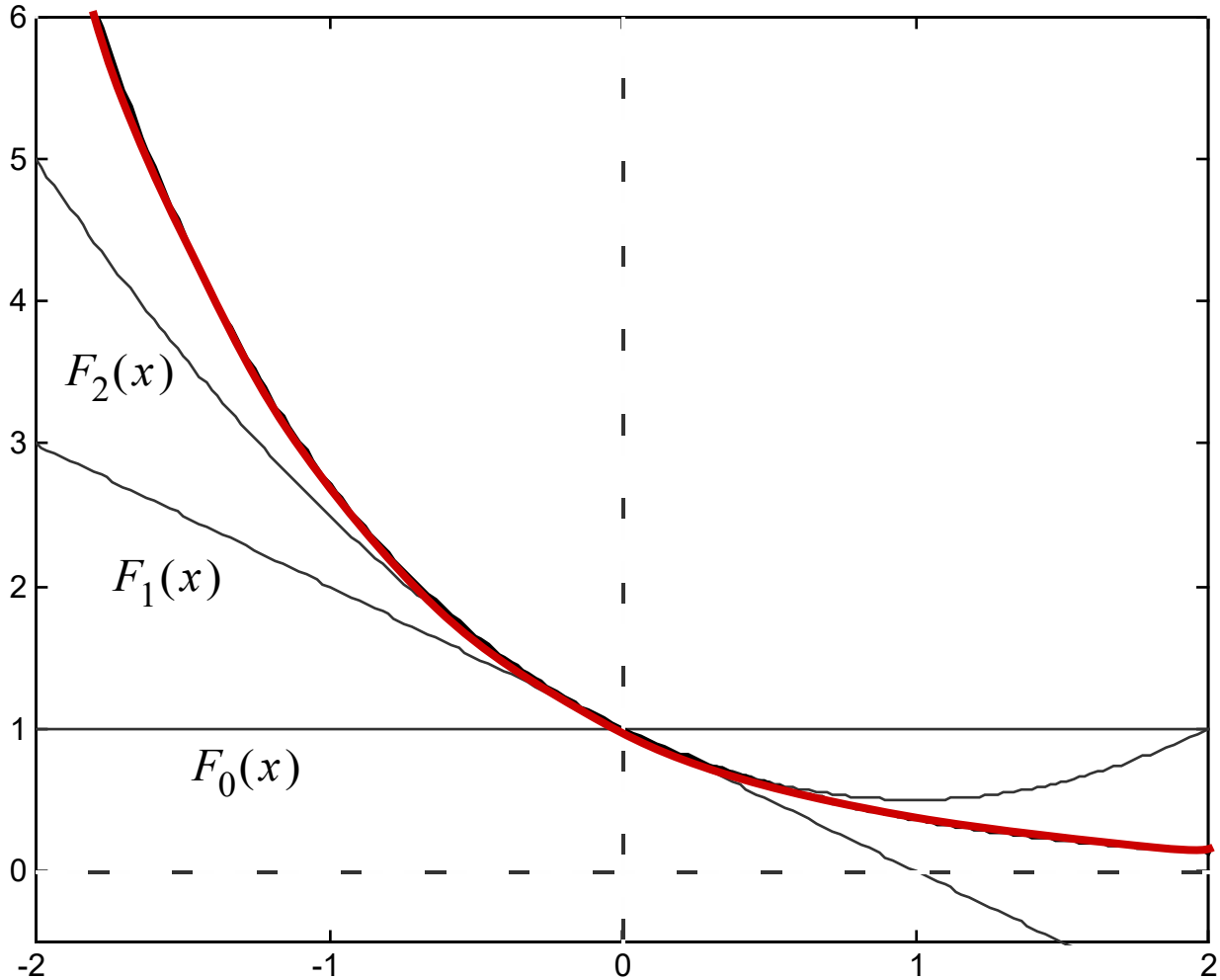
Taylor series approximations:

$$F(x) \approx F_0(x) = 1$$

$$F(x) \approx F_1(x) = 1 - x$$

$$F(x) \approx F_2(x) = 1 - x + \frac{1}{2}x^2$$

Plot of Approximations



Vector Case

$$F(x) = F(x_1, x_2, \ldots x_n)$$

$$F(x) = F(x^*) + \frac{d}{dx} F(x)\big|_{x=x^*} (x - x^*)$$

$$+ \frac{1}{2} \frac{d^2}{dx^2} F(x)\big|_{x=x^*} (x - x^*)^2 + \ldots$$

$$+ \frac{1}{n!} \frac{d^n}{dx^n} F(x)\big|_{x=x^*} (x - x^*)^n + \ldots$$

Matrix Form

$$F(x) = F(x^*) + \nabla F(x)^T \big|_{x=x^*} (x - x^*)$$
$$+ \frac{1}{2}(x - x^*)^T \nabla^2 F(x)^T \big|_{x=x^*} (x - x^*) + \cdots$$

Gradient

$$\nabla F(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(x) \\ \frac{\partial}{\partial x_2} F(x) \\ \vdots \\ \frac{\partial}{\partial x_n} F(x) \end{bmatrix}$$

Hessian

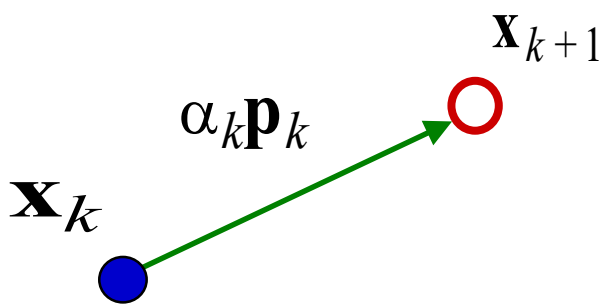
$$\nabla^2 F(x) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^{\,2}} F(x) & \frac{\partial^2}{\partial x_1 \partial x_2} F(x) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} F(x) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(x) & \frac{\partial^2}{\partial x_2^{\,2}} F(x) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} F(x) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F(x) & \frac{\partial^2}{\partial x_n \partial x_2} F(x) & \cdots & \frac{\partial^2}{\partial x_n^{\,2}} F(x) \end{bmatrix}$$

Basic Optimization Algorithm

$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k}$$

or

$$\Delta \mathbf{x}_k = (\mathbf{x}_{k+1} - \mathbf{x}_k) = \alpha_k \mathbf{p}_k$$



$\mathbf{p}_k$ - Search Direction

α_k - Learning Rate

Steepest Descent（最速下降）

Choose the next step so that the function decreases:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

For small changes in $\mathbf{x}$ we can approximate $F(\mathbf{x})$:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_k}$$

If we want the function to decrease:

$$\mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0$$

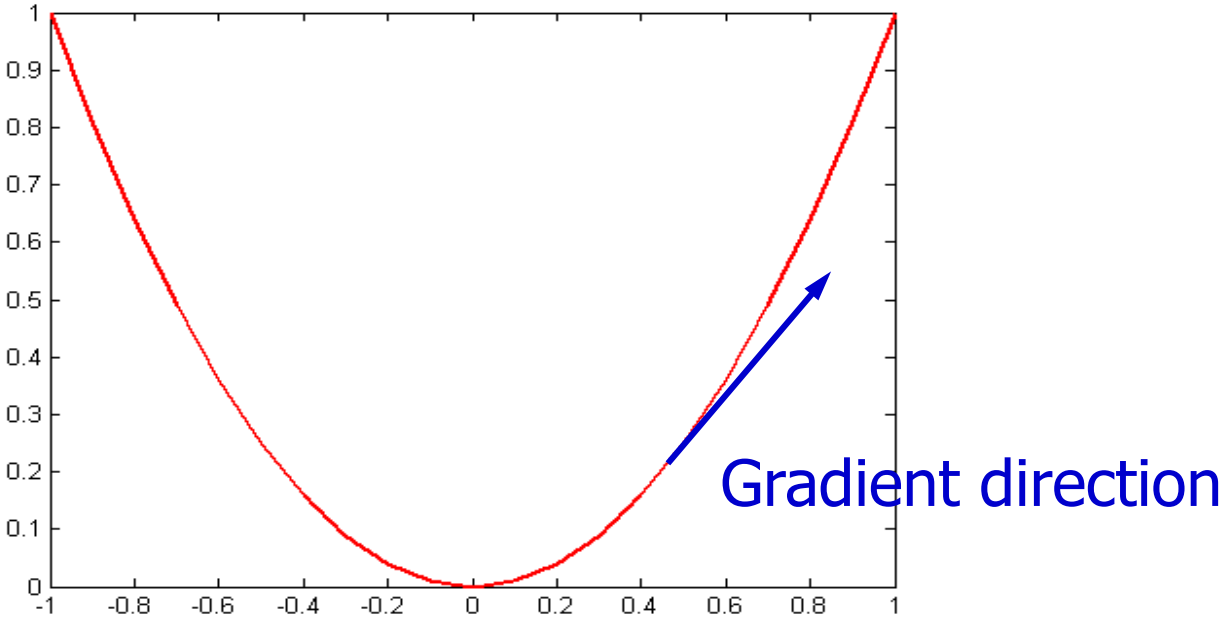
We can maximize the decrease by choosing:

$$\mathbf{p}_k = -\mathbf{g}_k$$

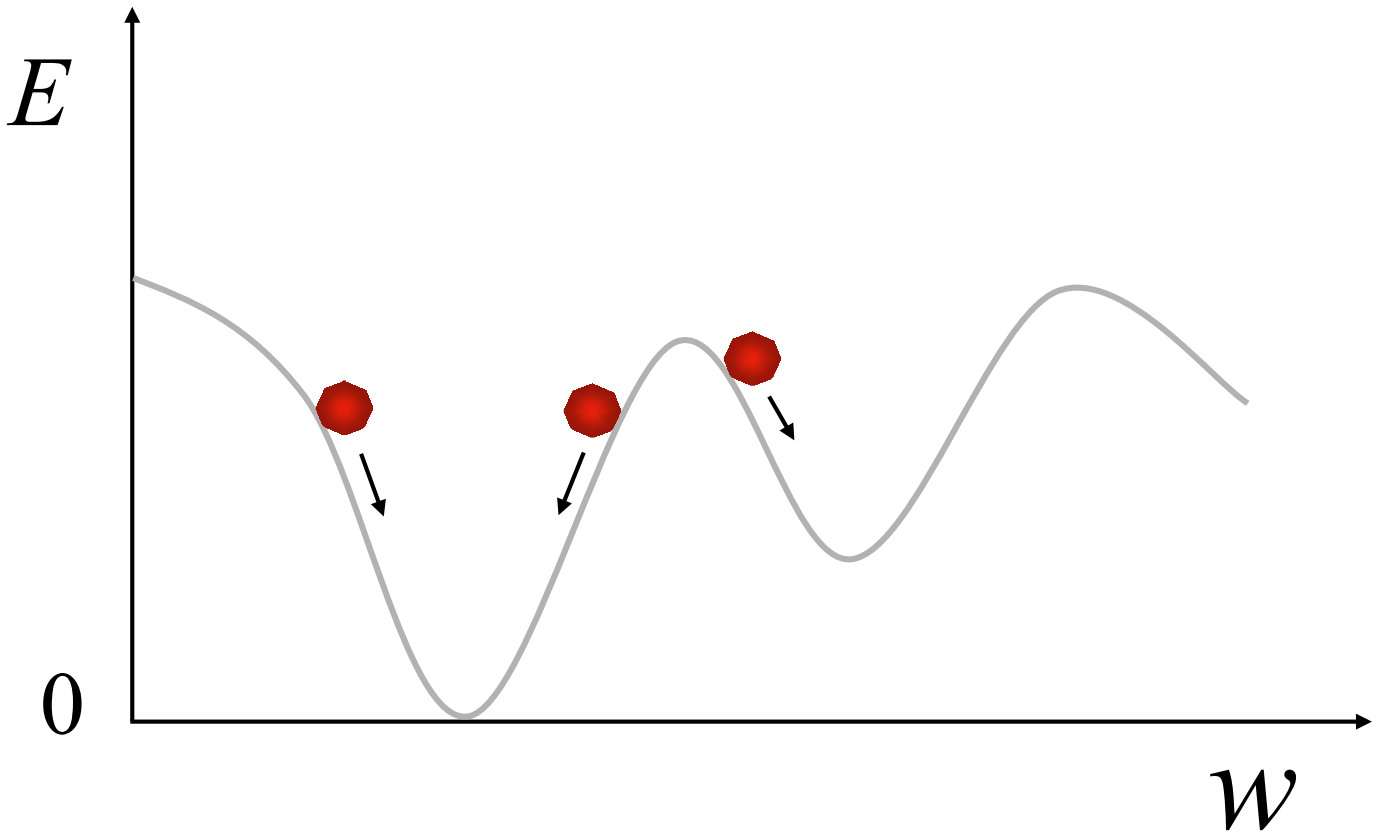
$$\boxed{\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k}$$

Gradient Method

Gradient method could be thought of as a ball rolling down from a hill: the ball will roll down and finally stop at the valley



Local Minimum Problem



Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

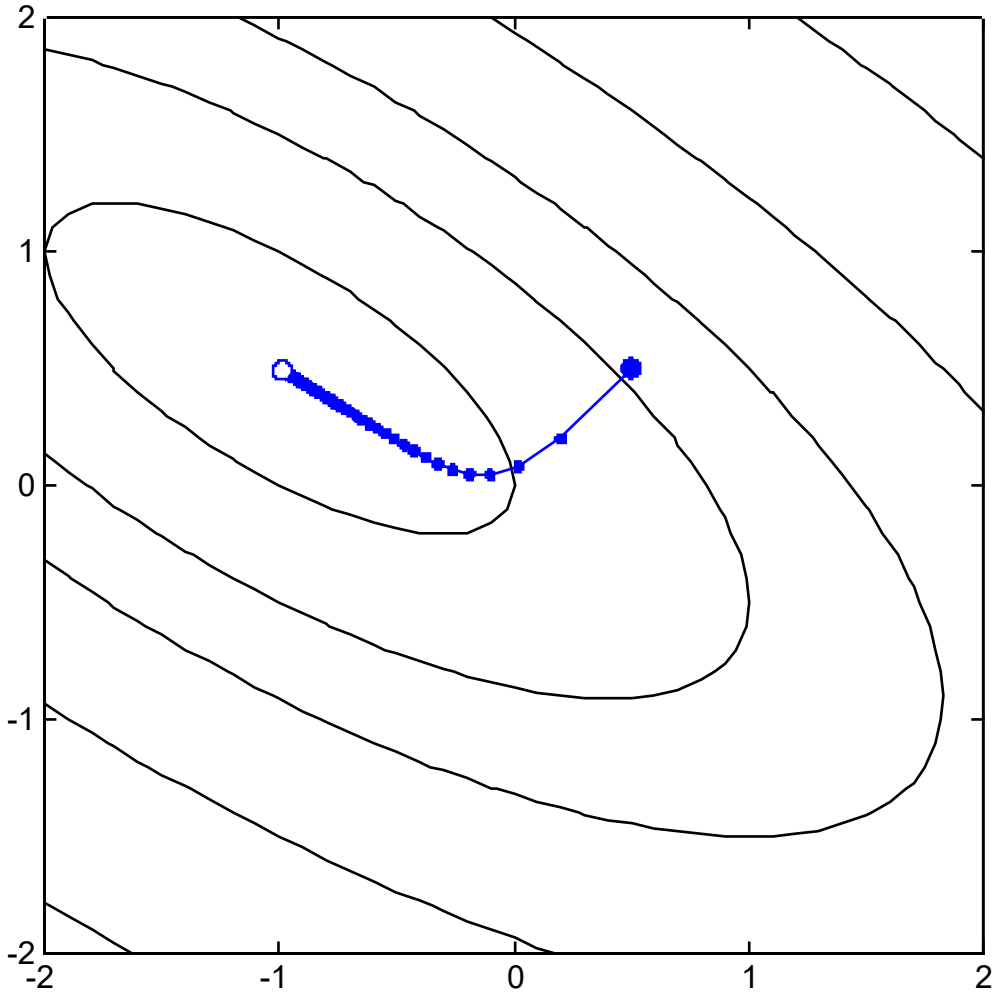
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

Plot



Stable Learning Rates (Quadratic)

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c$$

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\,\mathbf{g}_k = \mathbf{x}_k - \alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}) \quad \Longrightarrow \quad \mathbf{x}_{k+1} = \underbrace{[\mathbf{I} - \alpha\mathbf{A}]}_{\text{Stability is determined by the eigenvalues of this matrix.}}\mathbf{x}_k - \alpha\mathbf{d}$$

This is linear dynamic system, which will be stable if the eigenvalues of the matrix [**I** - α**A**] are less than one in magnitude .

Stable Learning Rates (Quadratic) -2

$$[\mathbf{I} - \alpha\mathbf{A}]\mathbf{z}_i = \mathbf{z}_i - \alpha\mathbf{A}\mathbf{z}_i = \mathbf{z}_i - \alpha\lambda_i\mathbf{z}_i = \underbrace{(1 - \alpha\lambda_i)}_{\text{Eigenvalues of } [\mathbf{I} - \alpha\mathbf{A}].}\mathbf{z}_i$$

(λ_i - eigenvalue of $\mathbf{A}$)

Stability Requirement:

$$\left|(1 - \alpha\lambda_i)\right| < 1 \qquad \alpha < \frac{2}{\lambda_i}$$

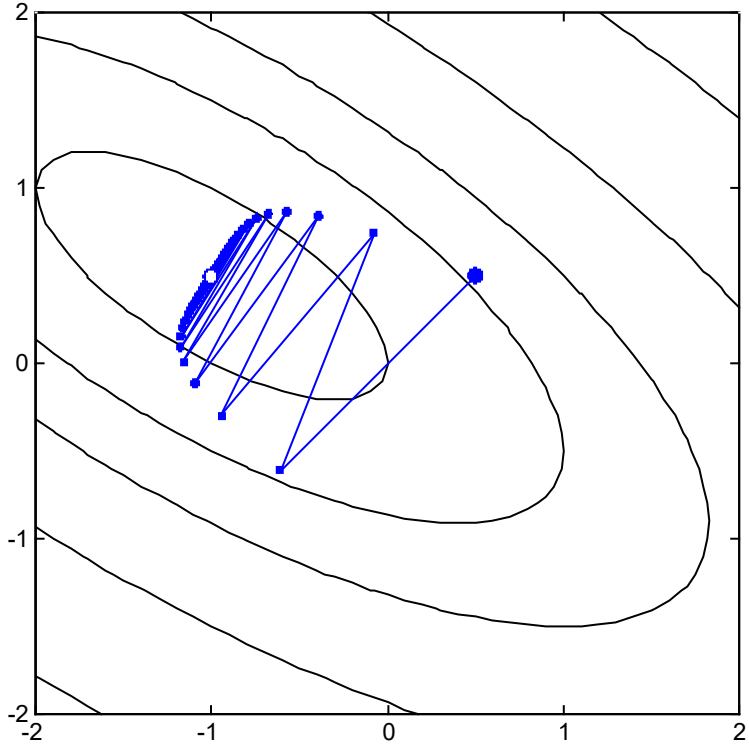
$$\boxed{\alpha < \frac{2}{\lambda_{max}}}$$

Example

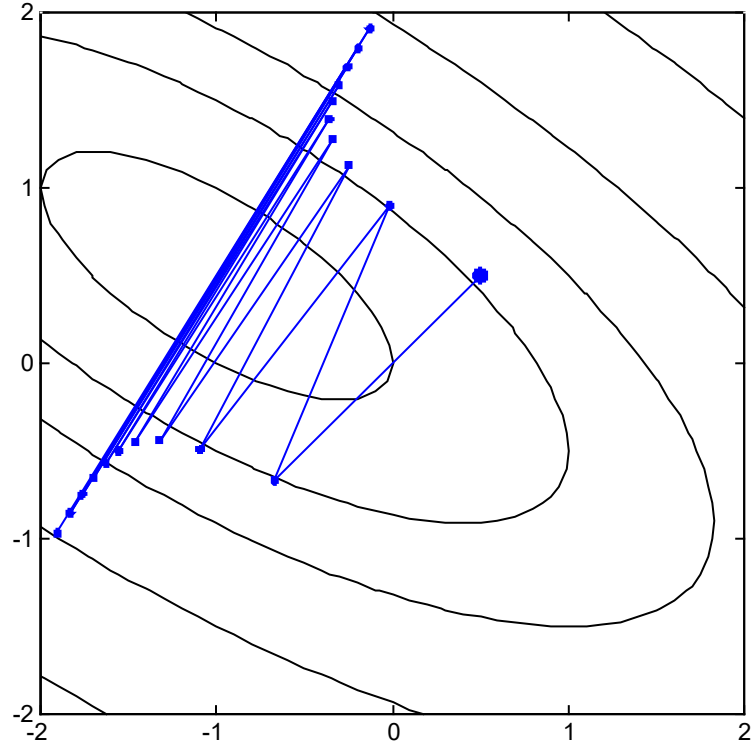
$$\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 2 & 4 \end{bmatrix} \qquad \left\{ (\lambda_1 = 0.764), \left(\mathbf{z}_1 = \begin{bmatrix} 0.851 \\ -0.526 \end{bmatrix} \right) \right\}, \left\{ \lambda_2 = 5.24, \left(\mathbf{z}_2 = \begin{bmatrix} 0.526 \\ 0.851 \end{bmatrix} \right) \right\}$$

$$\alpha < \frac{2}{\lambda_{max}} = \frac{2}{5.24} = 0.38$$

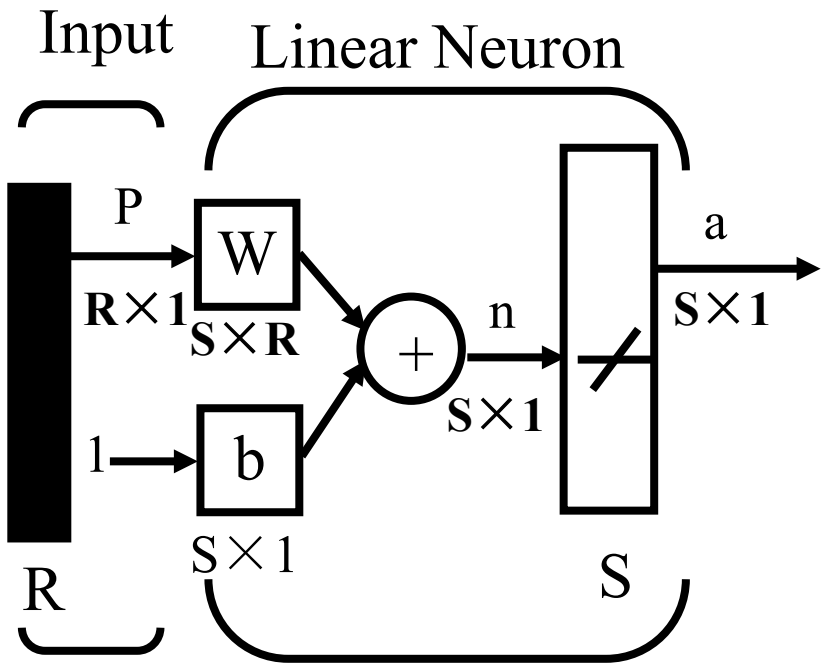
$\alpha = 0.37$



$\alpha = 0.39$



ADALINE (ADAptive LInear NEuron) Network



$$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp} + \mathbf{b}) = \mathbf{Wp} + \mathbf{b}$$

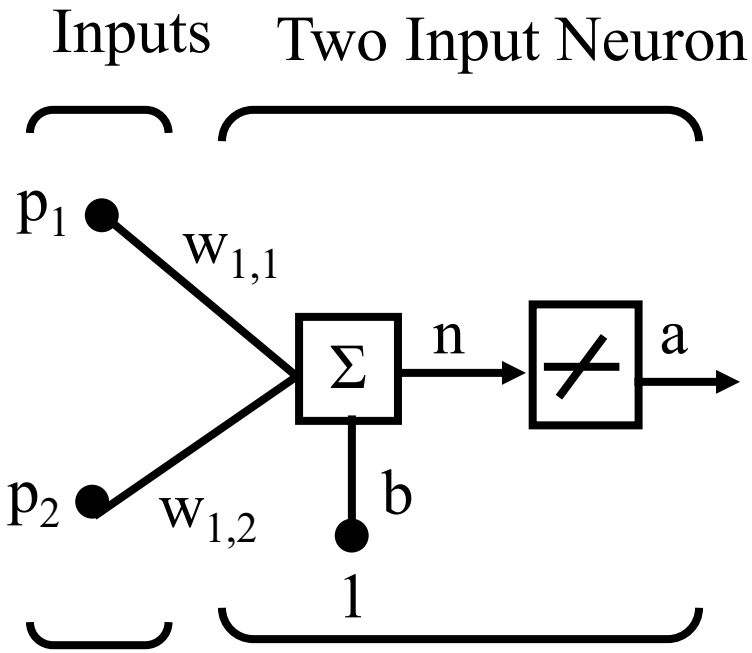
a=purelin(Wp+b)

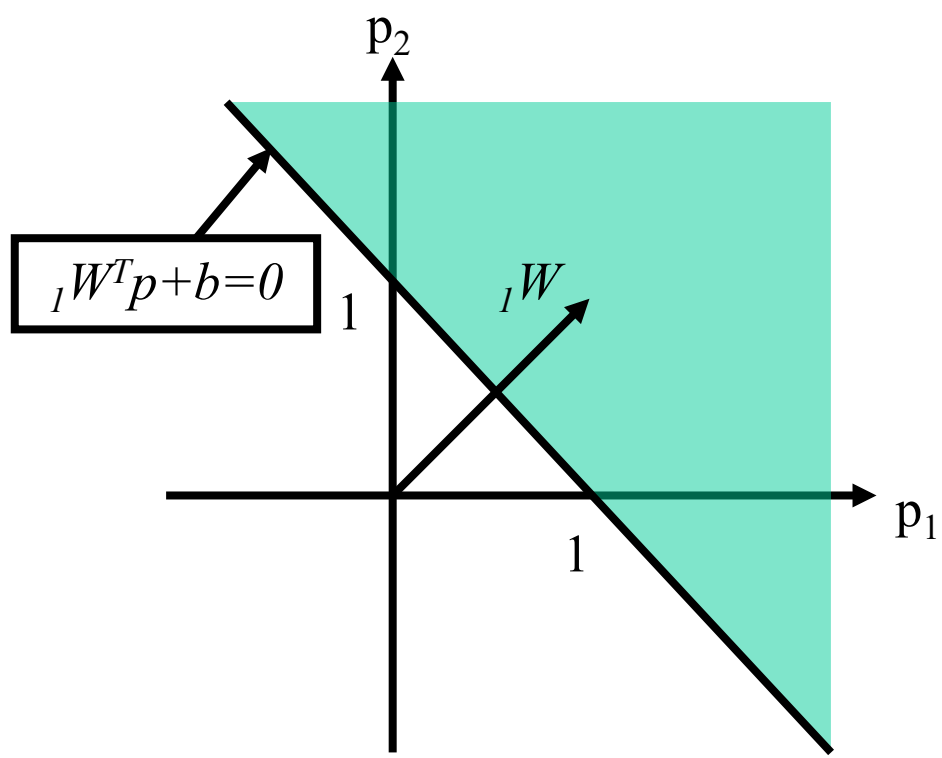
$$a_i = purelin(n_i) = purelin({}_i\mathbf{w}^T\mathbf{p} + b_i) = {}_i\mathbf{w}^T\mathbf{p} + b_i$$

$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

Two-Input ADALINE

$w_{1,1} = 1 \qquad w_{1,2} = 1 \qquad b = -1$





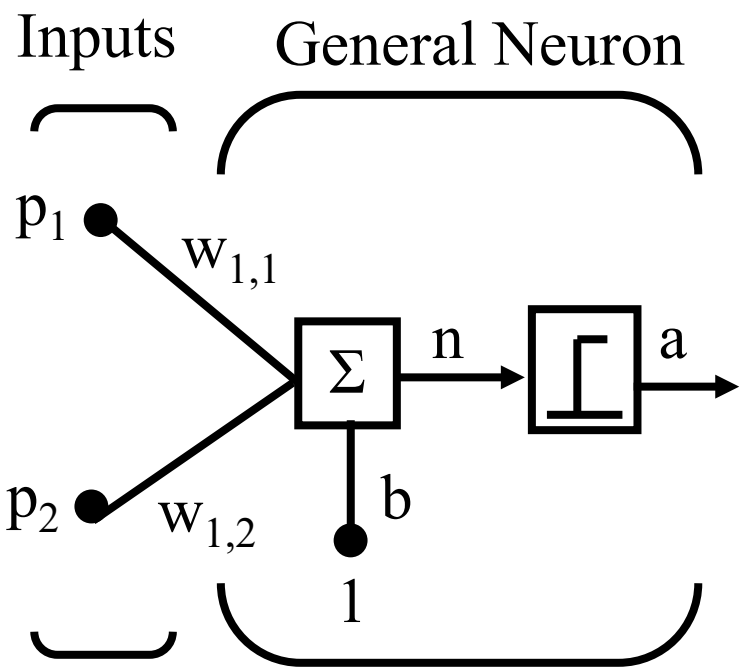
a=*purelin*(Wp+b)

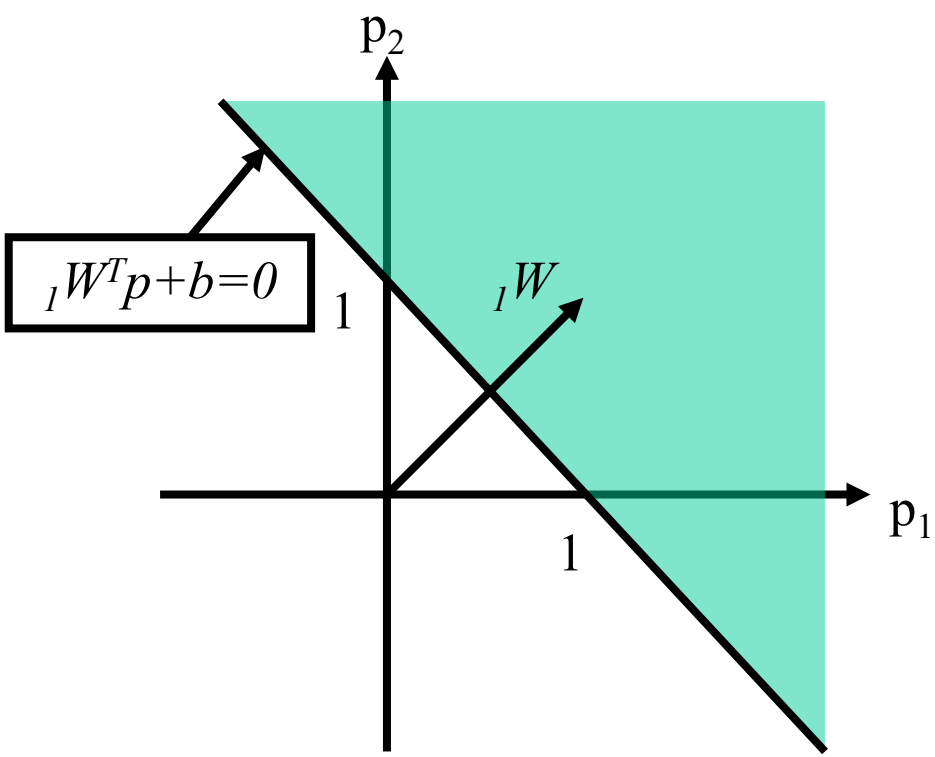
$$a = purelin(n) = purelin({}_1\mathbf{w}^T\mathbf{p} + b) = {}_1\mathbf{w}^T\mathbf{p} + b$$

$$a = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Single-Neuron Perceptron

$w_{1,1} = 1 \qquad w_{1,2} = 1 \qquad b = -1$





a=*hardlim*(Wp+b)

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

Mean Square Error

Training Set:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Input: $\mathbf{p}_q$ Target: $\mathbf{t}_q$

Notation:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \qquad a = {}_1\mathbf{w}^T\mathbf{p} + b \quad \Rightarrow \quad a = \mathbf{x}^T\mathbf{z}$$

Mean Square Error (均方误差):

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2]$$

$E[\,]$ 表示期望值

Error Analysis

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2]$$

$$F(\mathbf{x}) = E[t^2 - 2t\mathbf{x}^T\mathbf{z} + \mathbf{x}^T\mathbf{z}\mathbf{z}^T\mathbf{x}]$$

$$F(\mathbf{x}) = E[t^2] - 2\mathbf{x}^T\mathrm{E}[t\mathbf{z}] + \mathbf{x}^T E[\mathbf{z}\mathbf{z}^T]\mathbf{x}$$

$$\boxed{F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}}$$

$$c = E[t^2] \qquad \mathbf{h} = E[t\mathbf{z}] \qquad \mathbf{R} = E[\mathbf{z}\mathbf{z}^T]$$

The mean square error for the ADALINE Network is a quadratic function:

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} \qquad \mathbf{d} = -2\mathbf{h} \qquad \mathbf{A} = 2\mathbf{R}$$

向量h给出输入向量和对应目标输出之间的相关系数

R 是输入的相关矩阵

Quadratic Functions

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{d}^T\mathbf{x} + c \qquad \text{(Symmetric } \mathbf{A}\text{)}$$

Gradient and Hessian:

Useful properties of gradients:

$$\nabla(\mathbf{h}^T\mathbf{x}) = \nabla(\mathbf{x}^T\mathbf{h}) = \mathbf{h}$$

$$\nabla\mathbf{x}^T\mathbf{Q}\mathbf{x} = \mathbf{Q}\mathbf{x} + \mathbf{Q}^T\mathbf{x} = 2\mathbf{Q}\mathbf{x} \quad \text{(for symmetric } \mathbf{Q}\text{)}$$

Gradient of Quadratic Function:

$$\boxed{\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}}$$

Hessian of Quadratic Function:

$$\boxed{\nabla^2 F(\mathbf{x}) = \mathbf{A}}$$

Stationary Point

Hessian Matrix:

$$\mathbf{A} = 2\mathbf{R}$$

The correlation matrix **R** must be at least positive semidefinite. If there are any zero eigenvalues, the performance index will either have a weak minumum or else no stationary point, otherwise there will be a unique global minimum **x***.

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{h} + 2\mathbf{R}\mathbf{x}$$

$$-2\mathbf{h} + 2\mathbf{R}\mathbf{x} = \mathbf{0}$$

If R is positive definite:

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$$

Approximate Steepest Descent

Approximate mean square error (one sample):

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2]$$

$$\hat{F}(\mathbf{x}) = (t(k)-a(k))^2 = e^2(k)$$

Approximate gradient:

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k)$$

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}} \qquad j = 1, 2, \ldots, R$$

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Approximate Gradient Calculation

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial \mathrm{w}_{1,j}} [t(k) - ({}_1\mathbf{w}^T \mathbf{p}(k) + b)]$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} \left[t(k) - \left(\sum_{i=1}^{R} w_{1,i} p_i(k) + b \right) \right]$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \qquad \frac{\partial e(k)}{\partial b} = -1$$

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$$

LMS Algorithm

LMS Learning Rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k)$$

$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + 2\alpha e(k)\mathbf{p}(k)$$

$$b(k+1) = b(k) + 2\alpha e(k)$$

Perceptron Learning Rule

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p}$$

$$b^{new} = b^{old} + e$$

Multiple-Neuron Case

$$ {}_i\mathbf{w}(k+1) = {}_i\mathbf{w}(k) + 2\alpha e_i(k)\mathbf{p}(k) $$

$$ b_i(k+1) = b_i(k) + 2\alpha e_i(k) $$

Matrix Form:

$$ \mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k)\mathbf{p}^T(k) $$

$$ \mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k) $$

Example

Banana $\left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} -1 \end{bmatrix} \right\}$

Apple $\left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \end{bmatrix} \right\}$

$$\mathbf{R} = E[\mathbf{p}\mathbf{p}^T] = \frac{1}{2}\mathbf{p}_1\mathbf{p}_1^T + \frac{1}{2}\mathbf{p}_2\mathbf{p}_2^T$$ (Input Correlation Matrix)

$$\mathbf{R} = \frac{1}{2}\begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}\begin{bmatrix} -1 & 1 & -1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$\lambda_1 = 1.0, \qquad \lambda_2 = 0.0, \qquad \lambda_3 = 2.0$$

$$\alpha < \frac{1}{\lambda_{max}} = \frac{1}{2.0} = 0.5$$

Iteration One

Banana

$$a(0) = \mathbf{W}(0)\mathbf{p}(0) = \mathbf{W}(0)\mathbf{p}_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = 0$$

$$e(0) = t(0) - a(0) = t_1 - a(0) = -1 - 0 = -1$$

$$\mathbf{W}(1) = \mathbf{W}(0) + 2\alpha e(0)\mathbf{p}^T(0)$$

$$\mathbf{W}(1) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} + 2(0.2)(-1) \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix}$$

Iteration Two

Apple $\quad a(1) = \mathbf{W}(1)\mathbf{p}(1) = \mathbf{W}(1)\mathbf{p}_2 = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = -0.4$

$$e(1) = t(1) - a(1) = t_2 - a(1) = 1 - (-0.4) = 1.4$$

$$\mathbf{W}(2) = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} + 2(0.2)(1.4) \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}^T = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix}$$

Iteration Three

$$a(2) = \mathbf{W}(2)\mathbf{p}(2) = \mathbf{W}(2)\mathbf{p}_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64$$

$$e(2) = t(2) - a(2) = t_1 - a(2) = -1 - (-0.64) = -0.36$$

$$\mathbf{W}(3) = \mathbf{W}(2) + 2\alpha e(2)\mathbf{p}^T(2) = \begin{bmatrix} 1.1040 & 0.0160 & -0.0160 \end{bmatrix}$$

$$\mathbf{W}(\infty) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Perceptron rule stops as soon as the patterns are correctly classified, even though some patterns may be close to the boundaries.

The LMS algorithm minimizes the mean square error. Therefore it tries to move the decision boundaries as far from the reference patterns as possible.

Backpropagation Algorithm

Multilayer Perceptron with Two Hidden Layers

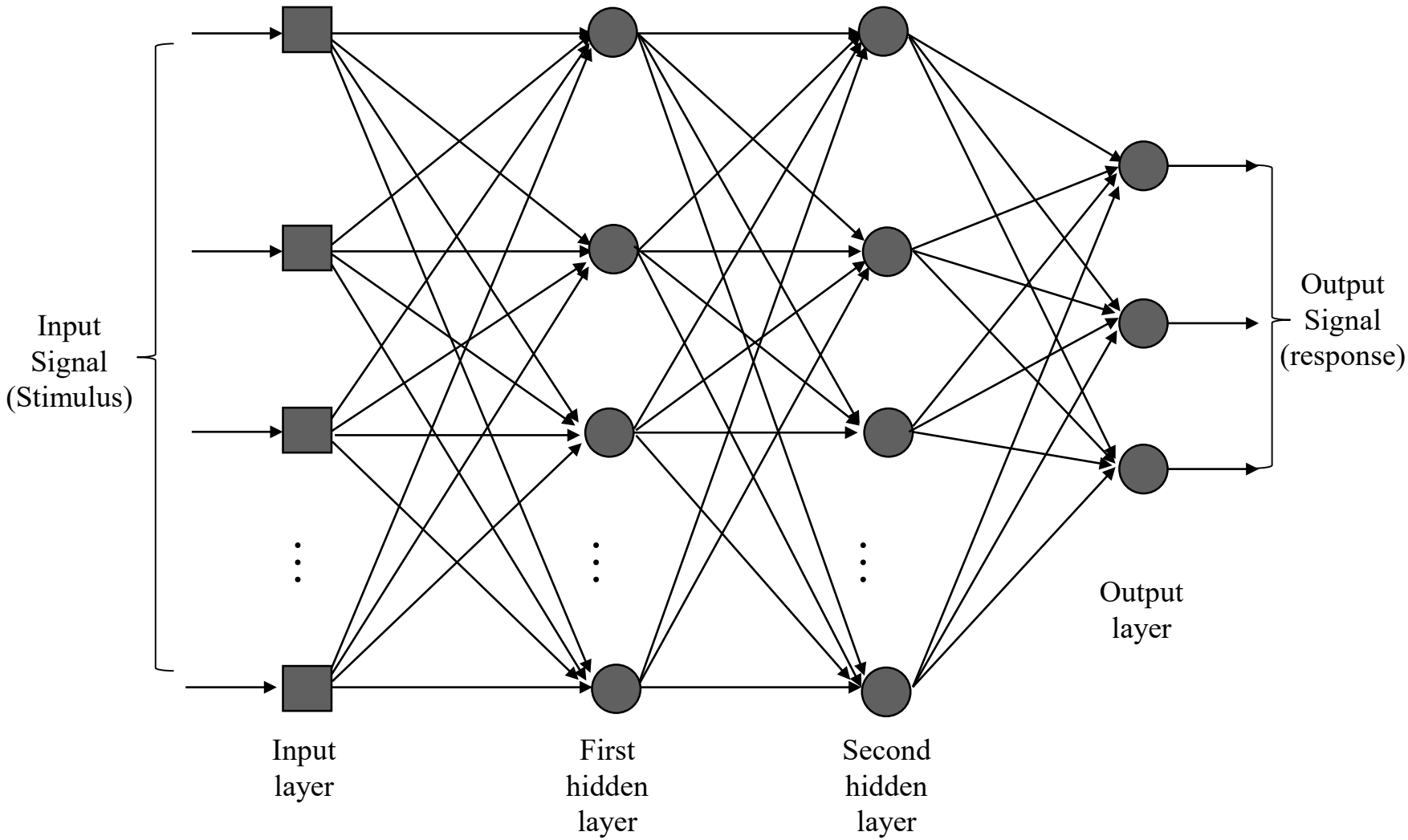
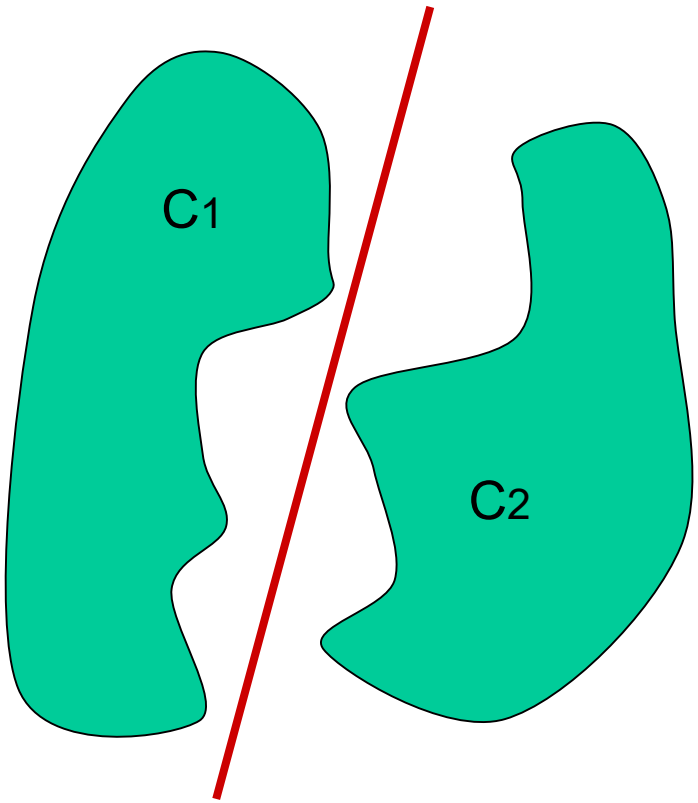
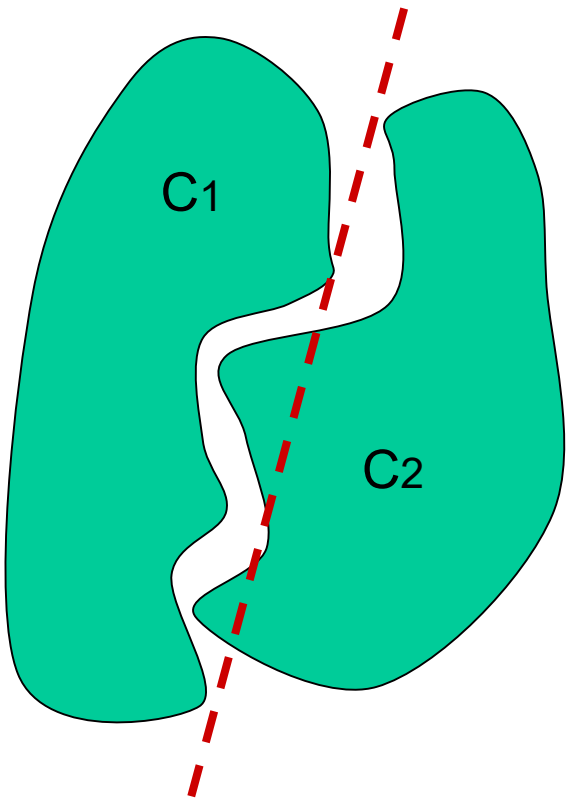


Figure 4.1

Linear separable and non-separable Problems

- Linear Separable Problem
- Linear Non-separable Problem





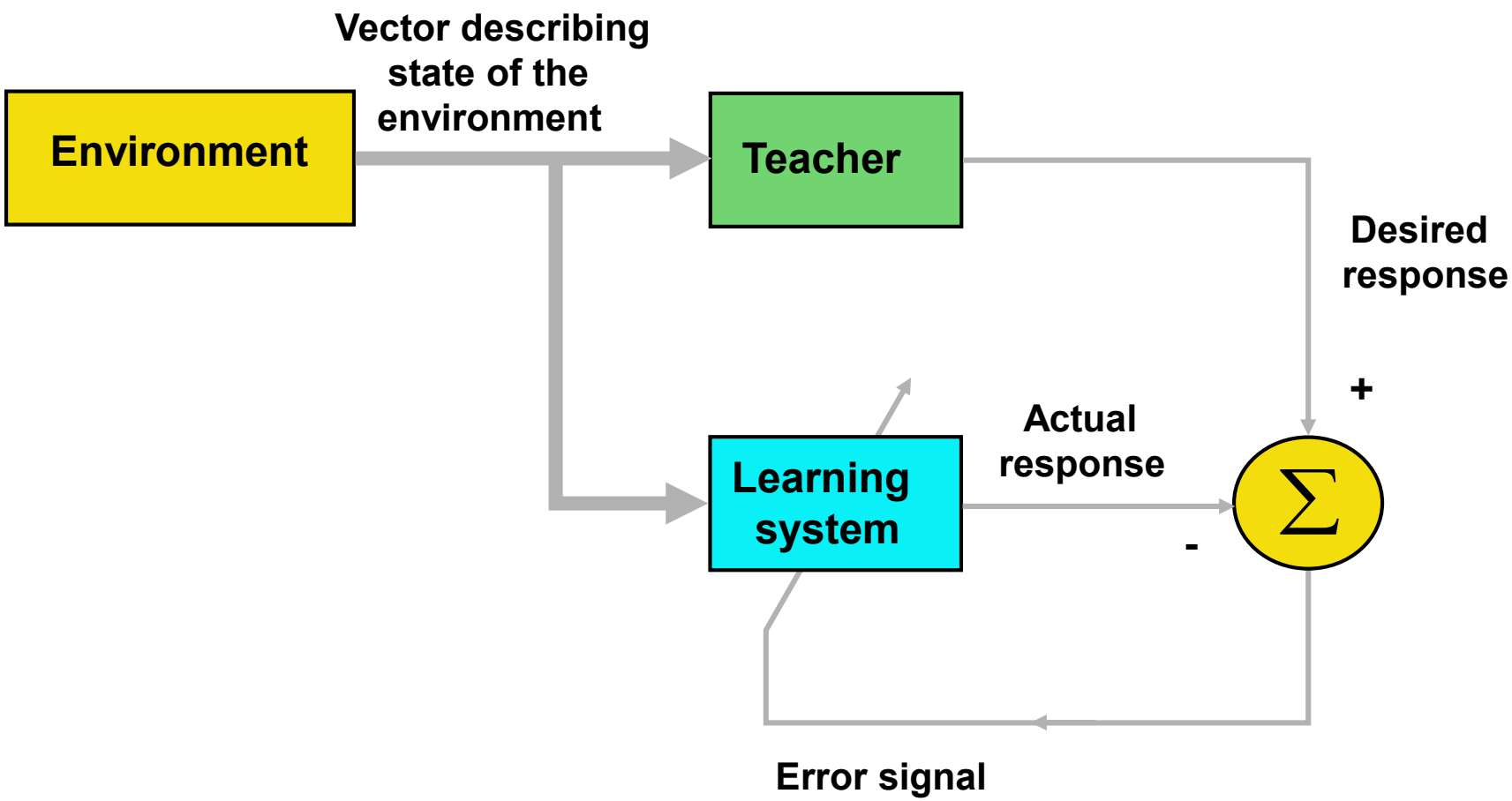
Neural Network History

- 1943: McCulloch and Pitts Model
- 1949: Hebb's rule
- 1962: Rosenblatt's learning Perceptron
- 1969: Minsky and Papert published their book "Perceptrons" on the limitations of perceptrons
- 1970s: Kohonen, Anderson, Grossberg, Amari, Fukushima
- 1980s: Rumelhart and McClelland, Hopfield, etc.
- 1990s: NNs have found many practical applications
- 2006: Hinton, LeCun, Bengio, etc. deep learning

Multilayer Perceptrons

- A multilayer feedforward network consists of an input layer, one or more hidden layers, and an output layer.
- Computations take place in the hidden and output layers only.
- The input signal propagates through the network in a forward direction, layer-by-layer.
- Such neural networks are called *multilayer perceptrons* (MLPs).
- They have been successfully applied to many difficult and diverse problems.
- Multilayer perceptrons are typically trained using so-called error *back-propagation algorithm*.
- This is a supervised error-correction learning algorithm.

Error-Corrective Learning



Multilayer Perceptrons -2

- It can be viewed as a generalization of the LMS algorithm.
- Back-propagation learning consists of two passes through the different layers of a MLP network.
- In the *forward pass*, the output (response) of the network to an input vector is computed.
- Here all the synaptic weights are kept fixed.
- During the *backward pass*, the weights are adjusted using an error-correction rule.
- The error signal is propagated backward through the network.
- After adjustment, the output of the network should have moved closer to the desired response in a statistical sense.

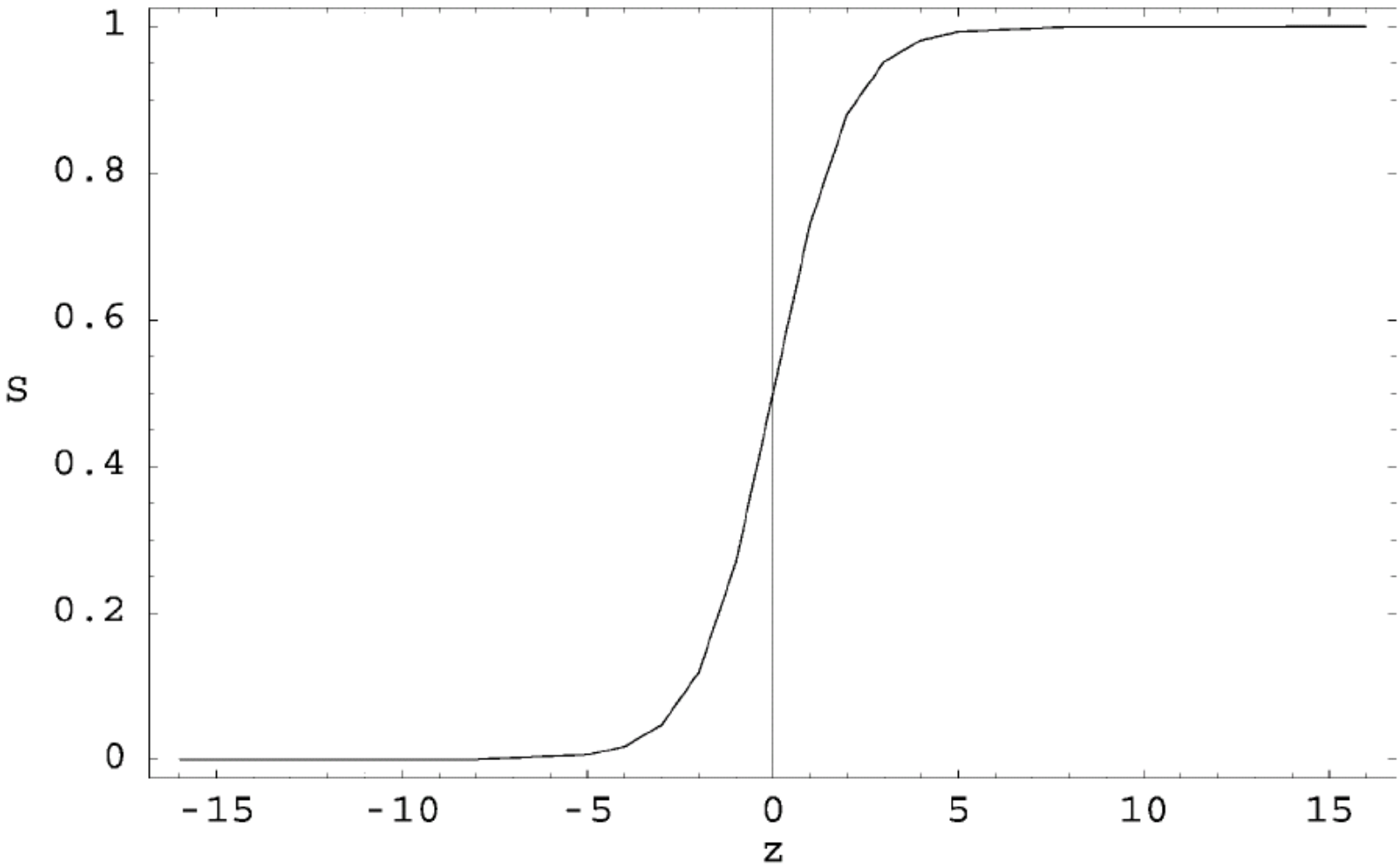
Properties of MLP

- Each neuron has a *smooth* (differentiable everywhere) *nonlinear activation function.*
- This is usually a sigmoidal nonlinearity defined by the *logistic function*

$$y_j = \frac{1}{1 + \exp(-v_j)}$$

  where v_j is the local field (weighted sum of inputs plus bias).
- Nonlinearities are important: otherwise the network could be reduced to a linear single-layer perceptron.

Sigmoidal Activation Function



Properties of MLP -2

- The network contains hidden layer(s), enabling learning complicated tasks and mappings.
- The network has a high connectivity.
- These properties give the multilayer perceptron its computational power.
- On the other hand, distributed nonlinearities make the theoretical analysis of a MLP network difficult.
- Back-propagation learning is more difficult and in its basic form slow because of the hidden layer(s).

Some Preliminaries

- Figure 4.1 shows an architectural graph of a multilayer perceptron with two hidden layers and an output layer.
- Recall that in the input layer, no computations take place; the input vector is only fed in componentwise.
- The network is *fully connected*.
- Two kinds of signals appear in the MLP network

1. *Function Signals.* Input signals propagating forward through the network, producing in the last phase output signals.
2. *Error signals.* Originate at output neurons, and propagate layer by layer backward through the network.

Two Basic Signal Flows

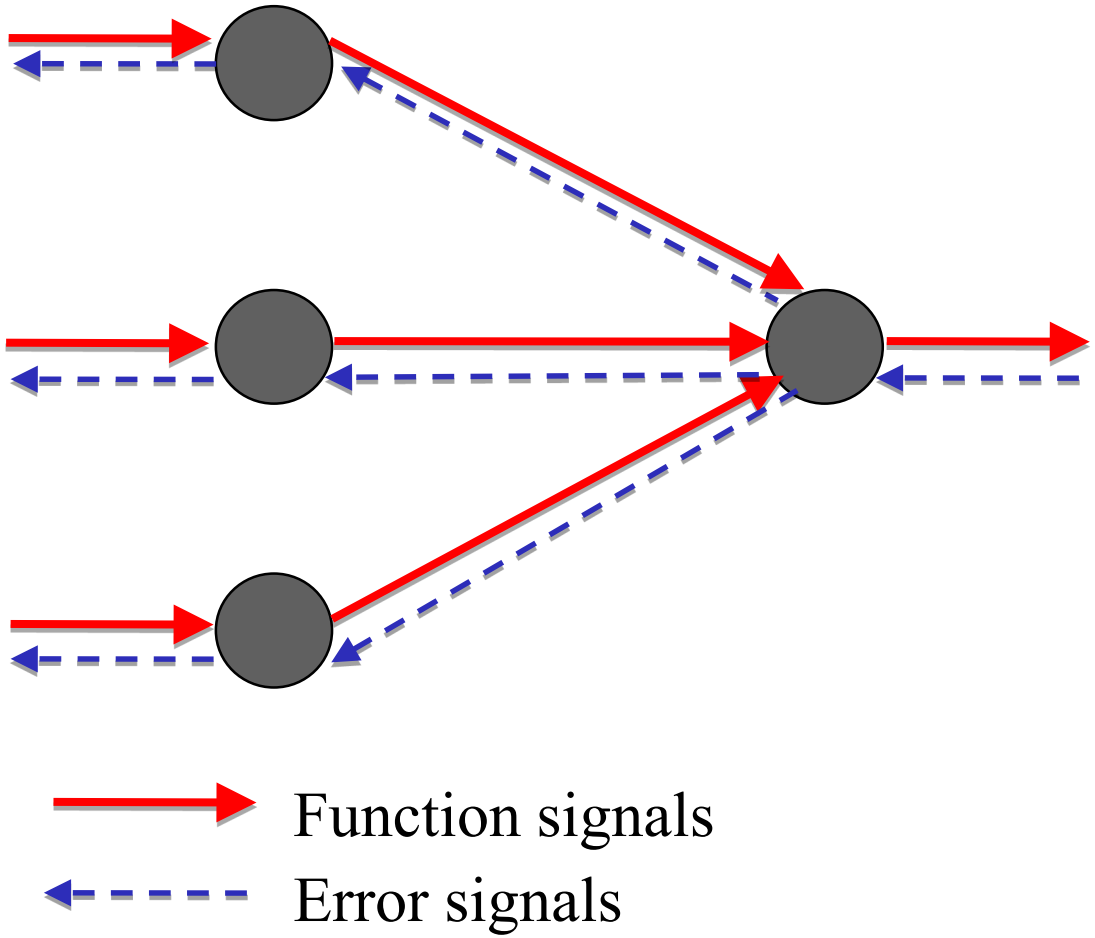


Figure 4.2

Some Preliminaries -2

- Each hidden or output neuron performs two computations:
  1. The computation of the function signal appearing at its output. This is a nonlinear function of the input signal and synaptic weights of that neuron.
  2. The computation of an estimate of the gradient vector, needed in the backward pass.
- The derivation of the back-propagation algorithm is rather involved.
- Before doing that, let us introduce the notation used.

Notation

- The indices i, j and k refer to neurons in different layers. Neuron j lies in the layer right to neuron i, and neuron k right to neuron j.
- In iteration n, the nth training vector is presented to the network.
- The symbol $\mathcal{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration n.
  - $\mathcal{E}_{av}$ is the average of $\mathcal{E}(n)$ over all n.
- $e_j(n)$ is the error signal at the output of neuron j for iteration n.
- $d_j(n)$ is the desired response for neuron j.
- $y_j(n)$ is the function signal at the output of neuron j for iteration n.

Notation -2

- $w_{ji}(n)$ is the weight connecting the output of neuron i to the input of neuron j at iteration n.
  - The correction applied to this weight is denoted by $\Delta w_{ji}(n)$.
- $v_j(n)$ denotes the local field of neuron j at iteration n.
  - It is the weighted sum of inputs plus bias of that neuron.
- The activation function (nonlinearity) associated with neuron j is denoted by $\varphi_j(.)$.
- b_j denotes the bias applied to neuron j, corresponding to the weight $w_{j0} = b_j$ and a fixed input $+1$.
- $x_i(n)$ denotes the ith element of the input vector.

Notation -3

- $o_k(n)$ denotes the kth element of the overall output vector.
- η denotes the learning-rate parameter.
- m_l denotes the number of neurons in layer l.
  - The network has L layers.
  - For output layer, the notation $m_L = M$ is also used.

Signal-flow graph of output neuron *k*

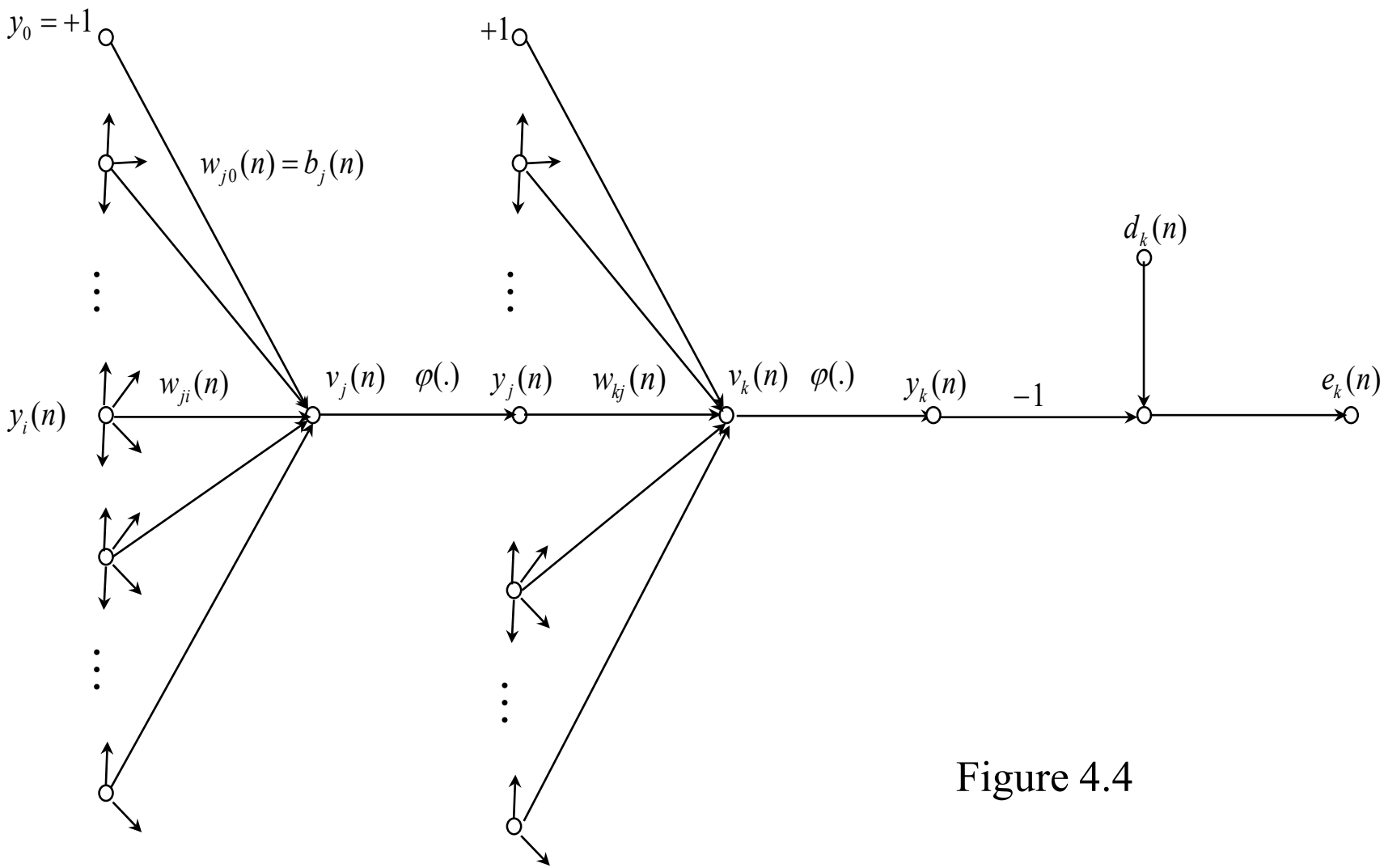


Figure 4.4

Back-Propagation Algorithm

- The error signal at the output of neuron j at iteration n is defined by

$$e_j(n) = d_j(n) - y_j(n), \text{ neuron j is an output node} \tag{1}$$

- The instantaneous value of the error energy for neuron j is defined by $e_j^2(n)/2$.
- The $\boxed{\text{total instantaneous error energy } \mathcal{E}(n)}$ for all the neurons in the output layer is therefore

$$\mathcal{E}(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n) \tag{2}$$

where the set C contains all the neurons in the output layer.

Back-Propagation Algorithm -2

- Let N be the total number of training vectors (examples, patterns).

- Then the *average squared error energy* is

$$\mathcal{E}_{av} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{E}(n) \tag{3}$$

- For a given training set, $\mathcal{E}_{av}$ is the *cost function* which measures the learning performance.

- It depends on all the free parameters (weights and biases) of the network.

- The objective is to derive a learning algorithm for minimizing $\mathcal{E}_{av}$ with respect to the free parameters.

Back-Propagation Algorithm -3

- In the basic back-propagation, a similar training method as in the LMS algorithm is used.
- Weights are updated on a pattern-by-pattern basis during each epoch.
- *Epoch* is one complete presentation of the entire training set.
- In other words, instantaneous stochastic gradient based on a single sample only is used for getting simple adaptive update formulas.
- The average of these updates over one epoch estimates the gradient of $\mathcal{E}_{av}$.

Signal-flow graph of output neuron *j*

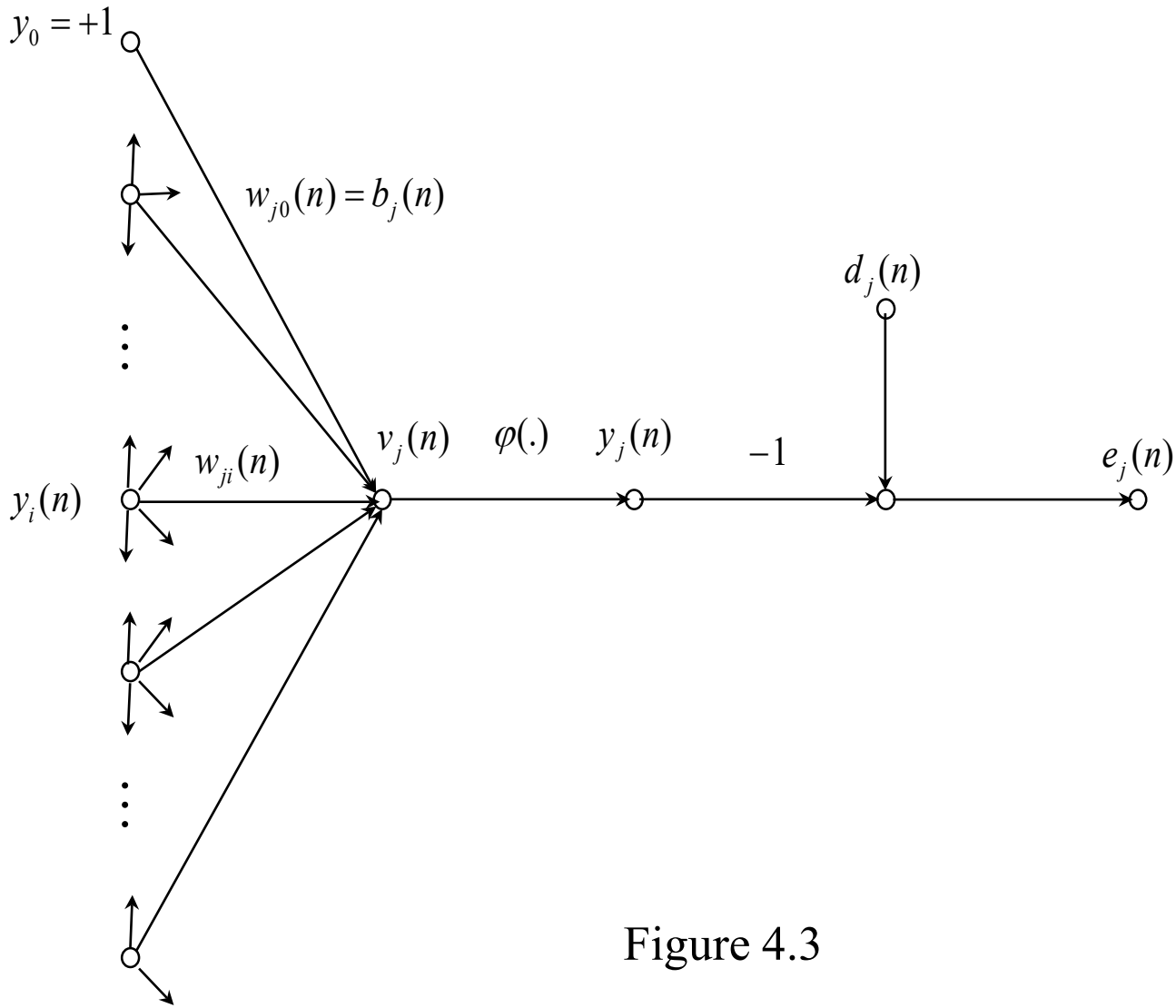


Figure 4.3

Back-Propagation Algorithm -4

- Consider now Figure 4.3 showing neuron j.
- It is fed by a set of function signals produced by a layer of neurons to its left.
- The local field $v_j(n)$ of neuron j is clearly

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{4}$$

- The function signal $y_j(n)$ appearing at the output of neuron j at iteration n is then

$$y_j(n) = \varphi_j(v_j(n)). \tag{5}$$

Back-Propagation Algorithm -5

- The correction $\Delta w_{ji}(n)$ made to the synaptic weight $w_{ji}(n)$ is proportional to the partial derivative $\partial\mathcal{E}(n)/\partial w_{ji}(n)$ of the instantaneous error.
- Using the *chain rule* of calculus, this gradient can be expressed as follows:

$$\frac{\partial\mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial\mathcal{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}\frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{6}$$

- The partial derivative $\partial\mathcal{E}(n)/\partial w_{ji}(n)$ represents a *sensitivity factor*.
- It determines the direction of search for the weight $w_{ji}(n)$.

Back-Propagation Algorithm -6

- Differentiating both sides of Eq. (2) with respect to $e_j(n)$, we get

$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n) \tag{7}$$

- Differentiating Eq. (1) with respect to $y_j(n)$ yields

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{8}$$

- Differentiating Eq. (5) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{9}$$

where φ_j' denotes the derivative of φ_j.

Back-Propagation Algorithm -7

- Finally, differentiating (4) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n). \tag{10}$$

- Inserting these partial derivatives into (6) yields

$$\boxed{\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)} \tag{11}$$

- The correction $\Delta w_{ji}(n)$ applied to the weight $w_{ji}(n)$ is defined by the *delta rule*:

$$\boxed{\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}} \tag{12}$$

where η is the learning-rate parameter of the back-propagation algorithm.

Back-Propagation Algorithm -8

- The minus sign comes from using *gradient descent* in learning for minimizing the error $\mathcal{E}(n)$.
- Inserting (11) into (12) yields

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_j(n) \tag{13}$$

where the *local gradient* is defined by

$$\boxed{\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\varphi_j'(v_j(n))} \tag{14}$$

- We note that a key factor in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j.
- This error signal depends on the location of the neuron in the MLP network.

Back-Propagation Algorithm -9

Case 1: Neuron j is an Output Node

- Computation of the error $e_j(n)$ is straightforward in this case.
- The desired response $d_j(n)$ for the neuron j is directly available.
- One can use the previous formulas (13) and (14).

Case 2: Neuron j is a Hidden Node

- Now there is no desired response available for neuron j.
- Question: how to compute the responsibility of this neuron for the error made at the output?
- This is the *credit-assignment problem*

Back-Propagation Algorithm -10

- The error signal for a hidden neuron must be determined recursively in terms of the error signals of all neurons connected to it.
- Here the development of the back-propagation algorithm gets complicated.
- Figure 4.4 illustrates the situation where neuron j is a hidden node.
- Using Eq. (14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as follows:

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n)) \qquad (15)$$

Signal-flow graph of output neuron k

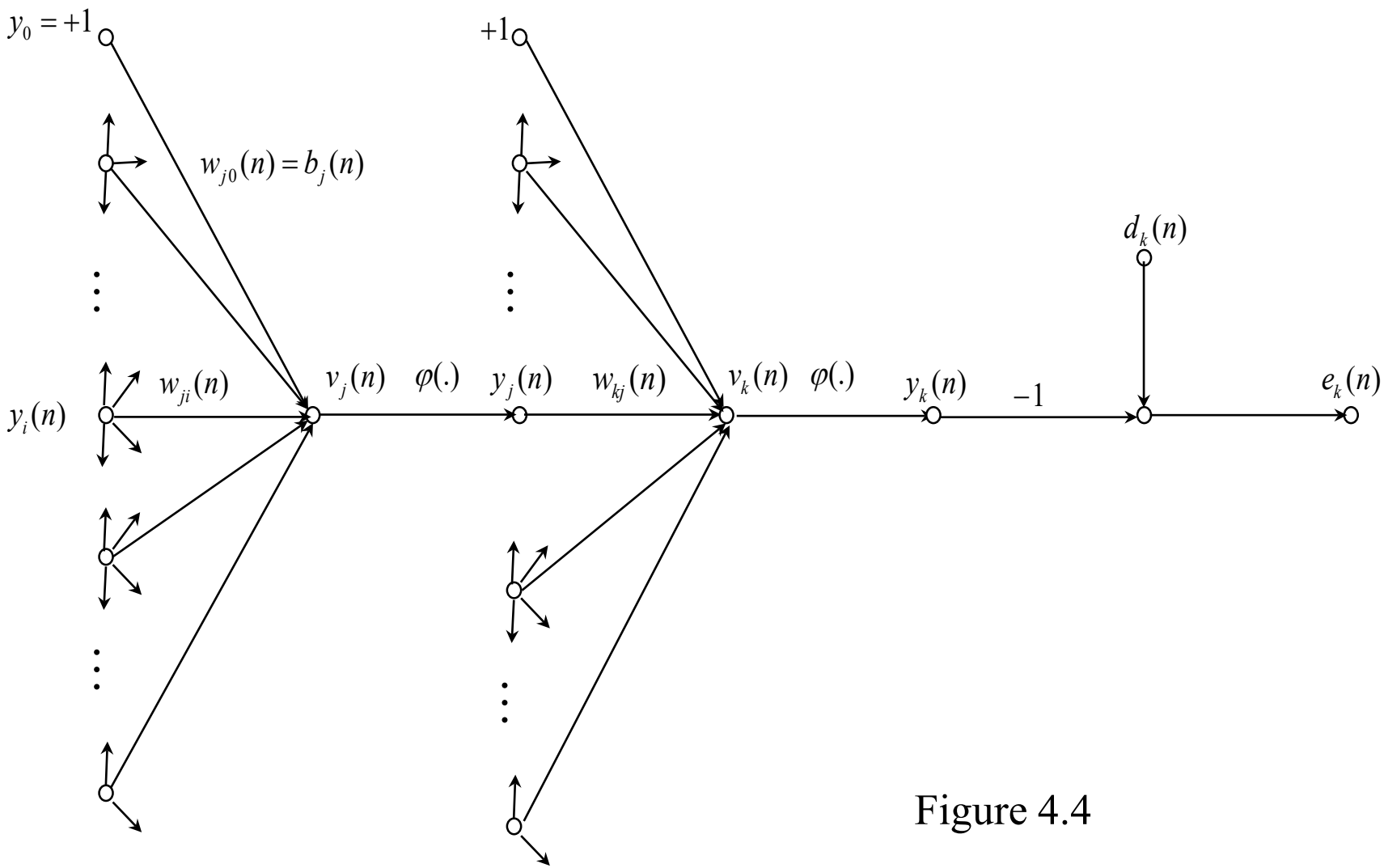


Figure 4.4

Back-Propagation Algorithm -11

- The partial derivative $\partial\mathcal{E}(n)/\partial y_j(n)$ may be calculated as follows.

- From Figure 4.4 we see that

$$\mathcal{E}(n) = \frac{1}{2}\sum_{k\in C} e_k^2(n), \text{ neuron k is an output node} \qquad (16)$$

- Differentiating this with respect to the function signal $y_j(n)$ and using the chain rule we get

$$\frac{\partial\mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \qquad (17)$$

Back-Propagation Algorithm -12

- From Figure 4.4 we note that when the neuron k is an output node

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \tag{19}$$

so that

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{20}$$

- Figure 4.4 shows also that the local field of neuron k is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{21}$$

where the bias term is again included as the weight $w_{k0}(n)$.

- Differentiating this with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{22}$$

Back-Propagation Algorithm -13

- Inserting these expressions into (3) we get the desired partial derivative

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) = -\sum_k \delta_k(n)w_{kj}(n) \quad (23)$$

- Here again $\delta_k(n)$ denotes the local gradient for neuron k.

- Finally, inserting (8) into (1) yields the *back-propagation formula* for the local gradient $\delta_j(n)$:

$$\boxed{\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n)} \quad (24)$$

- This holds when neuron j is hidden.

Back-propagation of local gradient

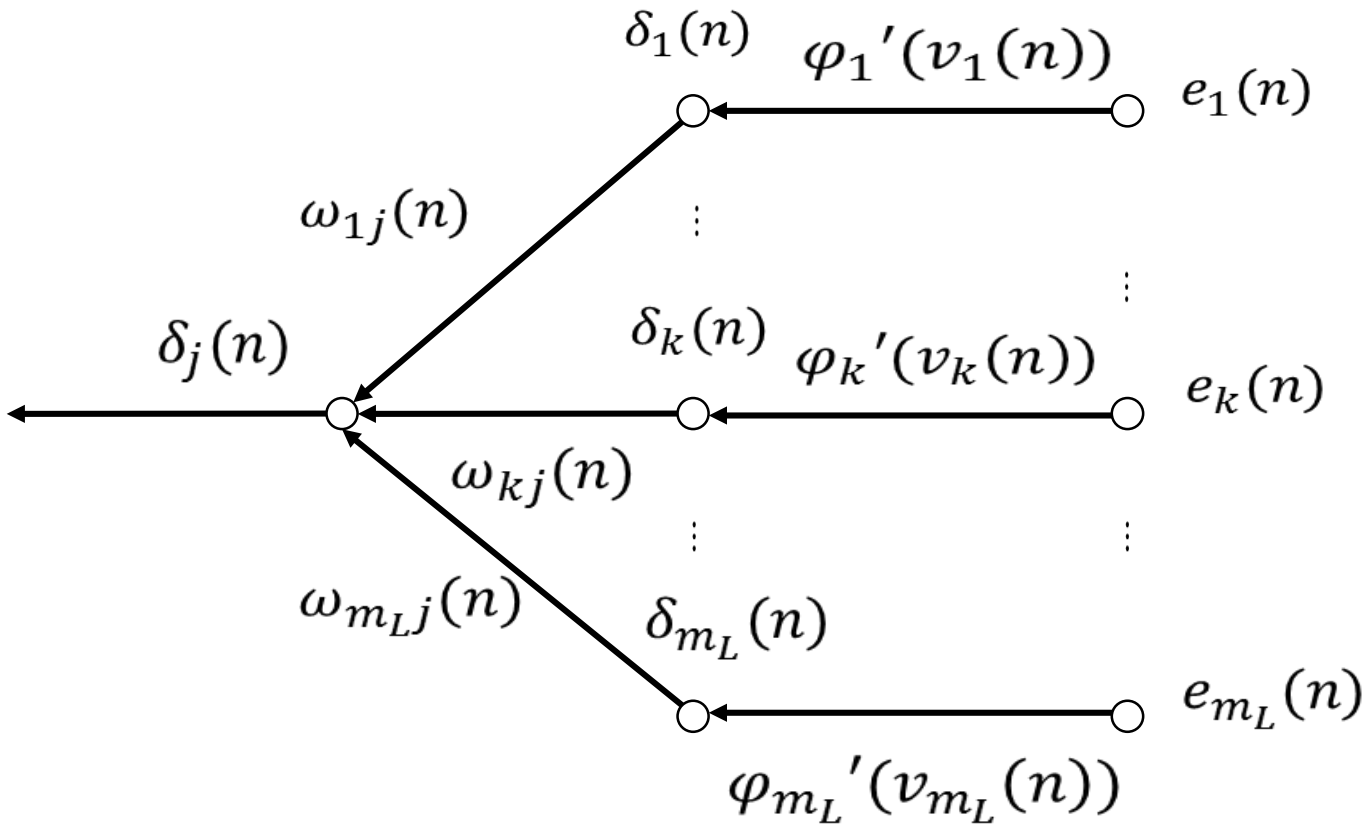


Figure 4.5

Back-Propagation Algorithm -14

- Let us briefly study the factors in this formula:
  - $\varphi_j'(v_j(n))$ depends solely on the activation function $\varphi_j(.)$ of the hidden neuron j.
  - The local gradients $\delta_k(n)$ require knowledge of the error signals $e_k(n)$ of the neurons in the next (right-hand side) layer.
  - The synaptic weights $w_{kj}(n)$ describe the connections of neuron j to the neurons in the next layer to the right.

Back-Propagation Algorithm -15

- We may summarize the results derived thus far in this section as follows:
- The correction $\Delta w_{ji}(n)$ of the weight connecting neuron i to neuron j is described by Eq. (25) (above Figure 4.5).
- The local gradient $\delta_j(n)$ is computed from Eq. (14) if neuron j lies in the output layer.
- If neuron j lies in the hidden layer, the local gradient is computed from Eq. (24)

$$\begin{pmatrix} Weight \\ correction \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} Learning \\ parameter \\ \eta \end{pmatrix} \begin{pmatrix} Local \\ gradient \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} Input\ signal \\ of\ neuron\ j \\ y_i(n) \end{pmatrix} \qquad (25)$$

Back-Propagation Algorithm -16

- The local gradient is given by

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n)) \tag{14}$$

when the neuron j is in the output layer.

- In the hidden layer, the local gradient is

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{24}$$

computed recursively from the local gradients of the following layer, back-propagating error

The Two-Passes of Computation

- In applying the back-propagation algorithm, two distinct passes of computation are distinguished.
- **Forward pass**
  - The weights are not changed in this phase.
  - The function signal appearing at the output of neuron j is computed as

$$y_j(n) = \varphi(v_j(n)) \tag{26}$$

  - Here the local field $v_j(n)$ of neuron j is

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{27}$$

The Two-Passes of Computation -2

- In the first hidden layer, $m = m_0$ is the number of input signals $x_i(n)$, $i = 1, \ldots, m_0$, and in (27)

$$y_i(n) = x_i(n)$$

- In the output layer, $m = m_L$ is the number of outputs (26).
- The outputs (components of the output vector) are denoted by

$$y_j(n) = o_j(n)$$

- These outputs are then compared with the respective desired responses $d_j(n)$, yielding the error signals $e_j(n)$.
- In the forward pass, computation starts from the first hidden layer and terminates at the output layer.

The Two-Passes of Computation -3

- **Backward pass**
  - In the backward pass, computation starts at the output layer, and ends at the first hidden layer.
  - The local gradient δ is computed for each neuron by passing the error signal through the network layer by layer.
  - The delta rule of Eq. (25) is used for updating the synaptic weights.
  - The weight updates are computed recursively layer by layer.
- The input vector is fixed through each round-trip (forward pass followed by a backward pass).
- After this, the next training (input) vector is presented to the network.

Learning Rate

- Back-propagation approximates steepest descent method.
- A small learning-rate parameter η leads to a slow learning rate.
- Generally, basic back-propagation suffers from very slow learning if the network is large (several layers, a lot of nodes).
- On the other hand, choosing too large a learning parameter may lead to oscillatory behavior.
- A simple method of improving the learning speed without oscillatory behavior:
- Use a *generalized delta rule* including a momentum term:

$$\boxed{\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)} \tag{39}$$

- Here α is a positive *momentum constant*.
- If $\alpha = 0$, the corresponding momentum term vanishes.

Learning Rate-2

- Then (39) reduces to the standard delta rule derived earlier.
- The effect of the momentum term is analyzed somewhat in Haykin's book on p. 170.
- The conclusions are:
  1. The momentum constant should be in the interval $0 \leq \alpha < 1$.
  2. The momentum term tends to accelerate descent in steady downhill direction.
  3. In directions where the partial derivative $\partial \mathcal{E}(t)/\partial w_{ji}(t)$ oscillates in sign, the momentum term has a stabilizing effect.

Learning Rate-3

- In deriving the back-propagation algorithm, it was assumed that the learning parameter η is a constant.
- In practice, it is better to use a *connection-dependent* learning parameter η_{ij}.
- This will be discussed later on in this chapter.

Sequential and Batch Modes of Training

- Recall that one complete presentation of the entire training set is called an epoch.
- The learning process is continued over several/many epochs.
- Learning is stopped when the weight values and biases stabilize, and the average squared error converges to some minimum value.
- It is useful to present the training samples in a randomized order during each epoch.
- In back-propagation, one may use either sequential (on-line, stochastic) or batch learning mode.

Sequential and Batch Modes of Training -2

1. Sequential Mode

- The weights are updated after presenting each training example (input vector).
- The derivation before was given for this mode.

2. Batch Mode

- Here the weights are updated after each epoch only.
- All the training examples are presented once before updating the weights and biases.
- In batch mode, the cost function is the average squared error

$$\mathcal{E}_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n) \tag{13}$$

Sequential and Batch Modes of Training -3

- The synaptic weight is updated using the batch delta rule

$$\Delta w_{ji} = -\eta \frac{\partial \mathcal{E}_{av}}{\partial w_{ji}} = -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}} \tag{43}$$

- The partial derivative $\partial e_j(n)/\partial w_{ji}$ may be computed as in the sequential mode.

- *Advantages of sequential mode:*

  - requires less storage
  - less likely to get trapped in a local minimum.

- *Advantages of the batch mode:*

  - Provides an accurate estimate of the gradient vector.
  - Convergence to a local minimum at least is guaranteed.

Sequential and Batch Modes of Training -4

- The sequential mode of back-propagation has several disadvantages.
- In spite of that, it is highly popular for two important practical reasons:
  - The algorithm is simple to implement.
  - It provides effective solutions to large and difficult problems.

Stopping Criteria

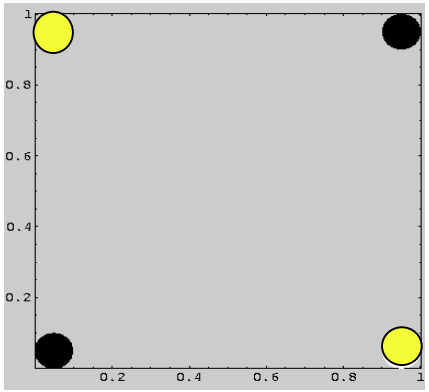
- In general, the back-propagation algorithm cannot be shown to converge.
- There are no well-defined criteria for stopping its operation.
- However, there are reasonable practical criteria for terminating learning.
- Consider first the unique properties of a *local* or *global minimum* of the error surface.
- Denote by $\mathbf{w}^*$ a weight vector at a local or global minimum.
- Necessary condition: the gradient vector $\mathbf{g}(\mathbf{w})$ vanishes at the minimum point $\mathbf{w}^*$.

Stopping Criteria -2

- This yield the following criterion for the convergence of back-propagation learning:
- *Stop learning when the Euclidean norm* $\| \mathbf{g}(\mathbf{w}) \|$ *of the gradient vector is below a certain threshold.*
- Drawbacks of this stopping criterion:
  - Learning times may be long.
  - Requires computation of the gradient vector $\mathbf{g}(\mathbf{w})$.
- Another criterion is based on the stationarity of the average squared error measure $\mathcal{E}_{av}$ at the point $\mathbf{w} = \mathbf{w}^*$:
- *Stop learning when the absolute rate of change in the average squared error per epoch is sufficiently small.*
- A small rate of change is usually taken to be 0.1% - 1% per epoch.

XOR Problem

- The patterns in the first class are (1,1) and (0,0).
- The patterns in the second class are (0,1) and (1,0).
- A single-layer perceptron is not sufficient for solving this problem.
- Reason: the classes are not linearly separable.
- However, the problem may be solved by adding a hidden layer.
- McCulloch-Pitts neuron model (a hard-limiting nonlinearity) is used here.



Heuristics for the BP Algorithm

- Design of a MLP network using back-propagation learning is partly art, not science.
- Numerous heuristic methods have been proposed for improving the learning speed and performance of back-propagation.

1. Sequential versus batch update

- Sequential learning mode is computationally faster than the batch mode.
- This is especially true when the training data set is large and highly redundant.

Heuristics for the BP Algorithm -2

2. Maximizing information content

- Every training example should be chosen so that it contains as much as possible useful information for the learning task.
- Two ways of achieving this aim are:
  - Using an example that results in the largest training error.
  - Using an example that is radically different from the previously used ones.
- The training examples should be presented in randomized order in different epochs.
- A more refined technique is to emphasize diffucult patterns in learning.
- However, this has problems also:
  - Distribution of the training data is distorted.
  - Outliers may have a catastrophic effect on performance.

Heuristics for the BP Algorithm -3

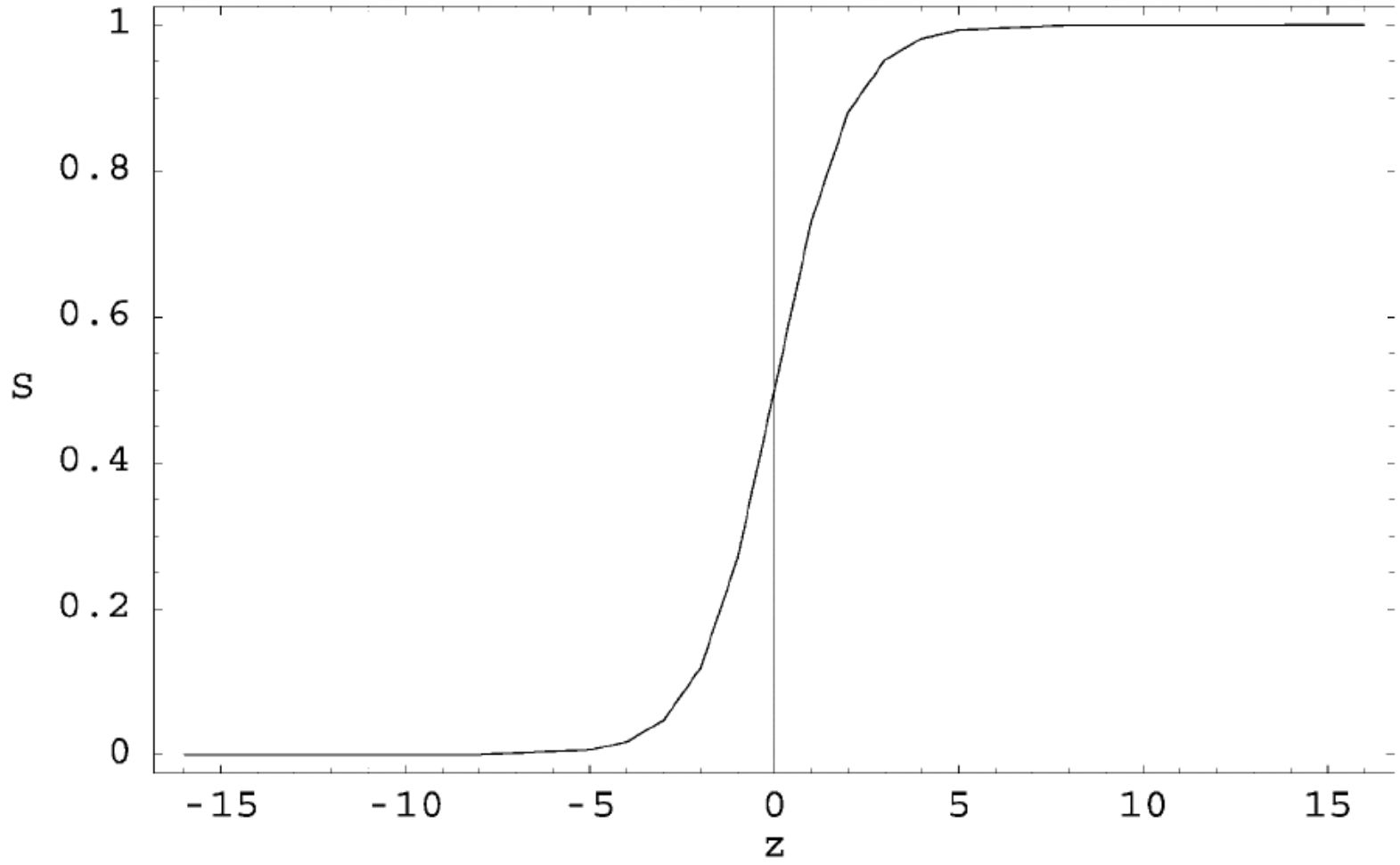
3. Activation function

- A MLP network trained with backpropagation typically learns faster if an antisymmetric sigmoid function is used.
- An activation function $\varphi(v)$ is *antisymmetric (odd)* if $\varphi(-v) = -\varphi(v)$.
- The standard logistic function $a/[1 + \exp(-bv)]$ is nonsymmetric, but $\tanh(bv)$ is antisymmetric.
- A good choice for an activation function:

$$\varphi(v) = a \tanh(bv)$$

  where $a = 1.7159$ and $b = 2/3$.
- Then $\varphi(1) = 1$, $\varphi(-1) = -1$, and the first and second derivatives of $\varphi(v)$ have suitable values.

Sigmoidal Activation Function



Heuristics for the BP Algorithm -4

4. Target values

- The target values (desired responses) should be chosen within the range of the sigmoid activation function.
- The desired responses should be somewhat smaller than the extremal (saturation) values of the activation function.
- Otherwise, the back-propagation algorithm tends to drive the free parameters of the networks to infinity.
- This slows down the learning process by driving the hidden neurons into saturation.
- For example, for the activation function $\varphi(v) = 1.716 \tanh(0.667v)$ discussed before, convenient target values are $d_j = \pm 1$ (see Fig. 4.10a).

Heuristics for the BP Algorithm -5

5. Normalizing the inputs

- For speeding up back-propagation learning, the input vectors (variables) should be *preprocessed*.
- Recommended preprocessing steps for the training patterns:
  1. The mean value of the training vectors should be made zero (or small enough).
     - prevents slow, zigzagging type learning.
  2. The input variables (different components of training vectors) should be uncorrelated.
     - Can be realized using Principal Components Analysis (Chapter 8).
     - Removes second-order statistical redundancies.
  3. The decorrelated input variables should be scaled to have approximately the same variances.
     - Ensures that different synaptic weights learn with roughly the same speed.

Normalization of Inputs and Outputs

$$\bar{x}_{ik} = X_{\min} + \frac{x_{ik} - x_{k,\min}}{x_{k,\max} - x_{k,\min}}(X_{\max} - X_{\min}) = \alpha_k + \beta_k x_{ik},$$

where $x_{ik} \in \mathbf{R}$ is the kth element of the ith training input, $x_{k,\min} = \min\{x_{ik} | i = 1, 2, \ldots, M\}$, $x_{k,\max} = \max\{x_{ik} | i = 1, 2, \ldots, M\}$, $\bar{x}_{ik}$ is the kth element of the ith normalized training input, $\alpha_k = X^k_{\min} - x_{k,\min}(X_{\max} - X_{\min})/(x_{k,\max} - x_{k,\min})$, and $\beta_k = (X_{\max} - X_{\min})/(x_{k,\max} - x_{k,\min})$.

Heuristics for the BP Algorithm -6

6. Initialization

- Good initial values for the synaptic weights and thresholds (biases) of the network can help tremendously in designing a good network.
- Assume first that the synaptic weights have large initial values.
- Then it is likely that the neurons will be driven into saturation.
- Results in slow learning.
- Assume now that synaptic weights are assigned small initial values.

Heuristics for the BP Algorithm -7

- Assume now that:
  - The input variables have zero mean and unit variance.
  - They are mutually uncorrelated.
  - The tanh nonlinearity is used.
  - The thresholds (biases) are set to zero for all neurons.
  - The initial values of the synaptic weights are drawn from a uniform distribution with zero mean and the same variance σ_w^2
- It is then fairly easy to show (see Haykin, pp. 183-184) that:
- For the activation function $\varphi(v) = 1.716 \tanh(0.667v)$ discussed earlier, we should choose $\sigma_w^2 = m^{-1}$.
- Here m is the number of synaptic connections of a neuron.

Generalization

- In back-propagation learning, as many training examples as possible are typically used.
- It is hoped that the network so designed generalizes well.
- A network *generalizes* well when its input-output mapping is (almost) correct also for test data.
- The test data is not used in creating or training the network.
- Assumption: test data comes from the same population (distribution) as the training data.
- Training of a neural network may be viewed as a curve fitting (nonlinear mapping) problem.
- The network can simply be considered as a nonlinear input-output mapping.

Good and Poor Generalization

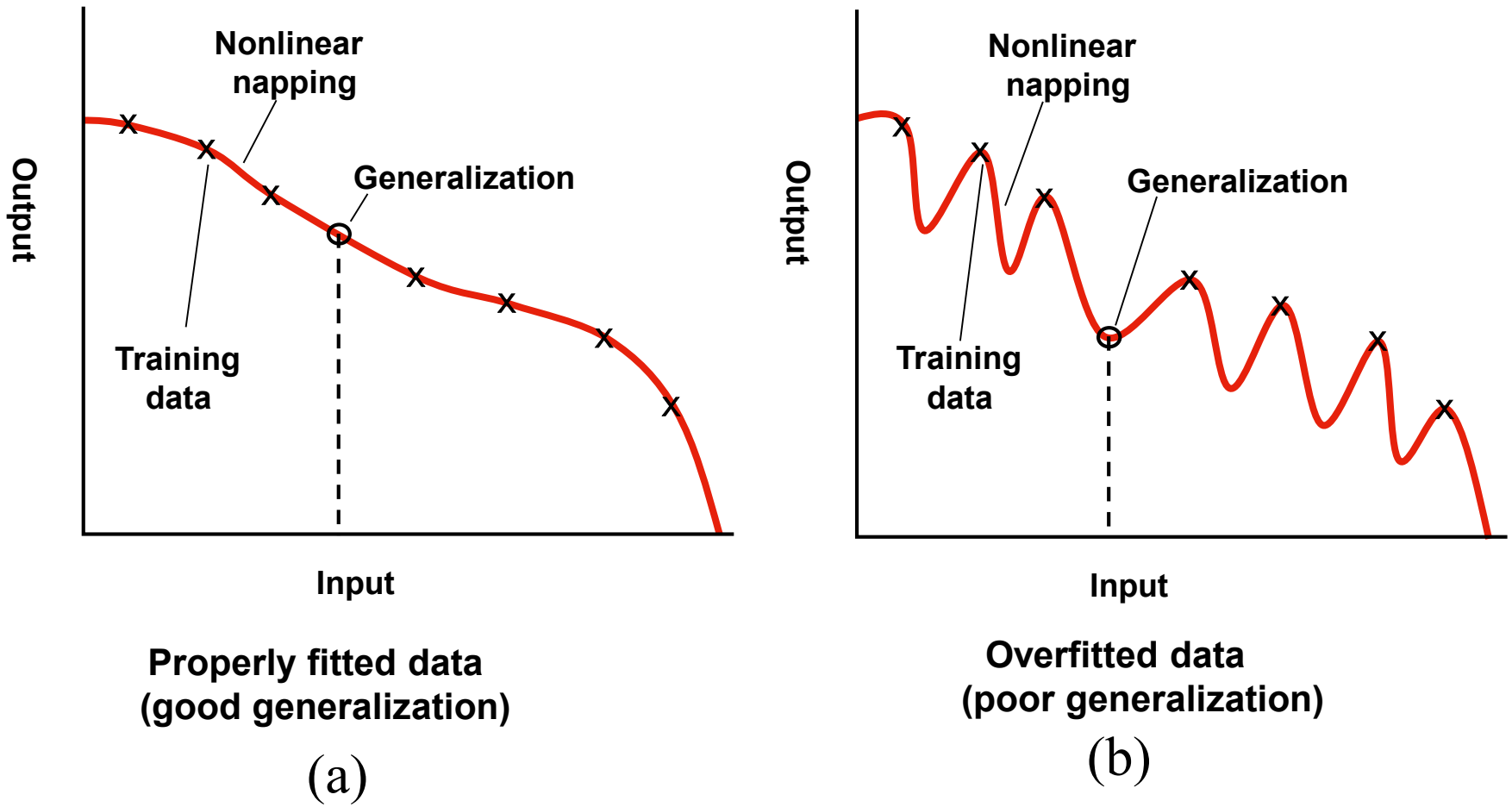


Figure 4-19

Generalization -2

- Generalization can be studied in terms of the nonlinear interpolation ability of the network.
- MLP networks with continuous activation functions perform useful interpolation because they have continuous outputs.
- Figure 4.19a shows an example of good generalization for a data vector not used in training.
- Using too many training examples may lead to a poor generalization ability.
- This is called *overfitting* or *overtraining*.
- The network then learns even unwanted noise present in the training data.
- More generally, it learns "features" which are present in the training set but actually not in the underlying function to be modeled.

Generalization -3

- Basic reason for overfitting: there are more hidden neurons than necessary in the network.
- Similar phenomena appear in other modeling problems if the chosen model is too complicated, containing too many free parameters.
- For example least-squares fitting, autoregressive modeling, etc.
- *Occam's razor* principle in model selection: select the simplest model which describes the data adequately.
- In the neural network area, this implies choosing the smoothest function that approximates the input-output mapping for a given error criterion.
- Such a choice generally demands the fewest computational resources.

Sufficient Training Data

- Three factors affect generalization:
  1. The size and representativeness of the training set.
  2. The architecture of the neural network.
  3. The physical complexity of the problem at hand.
- Only the first two factors can be controlled.
- The issue of generalization may be viewed from two different perpectives:
  - *The architecture of the network is fixed.* Determine the size of the training set for a good generalization.
  - *The size of the training set is fixed.* Determine the best architecture of network for achieving a good generalization.

Sufficient Training Data -2

- Here we focus on the first viewpoint, hoping that the fixed architecture matches the complexity of the problem.
- Distribution-free, worst-case formulas are available for estimating the size of sufficient training set for a good generalization performance.
- *A practical condition for a good generalization:*
- The size N of the training set must satisfy the condition

$$N = O\left(\frac{W}{\varepsilon}\right)$$

- Here W is the total number of free parameters (weights and biases) in the network.
- ε denotes the fraction of classification errors permitted on test data (as in pattern classification).

Sufficient Training Data -3

- $O(.)$ is the order of quantity enclosed within it.
- Example: If an error of 10% is permitted, the number of training examples should be about 10 times the number of free parameters.

Approximation of Functions

- A MLP network trained with back-propagation is a practical tool for performing a *general nonlinear input-output mapping*.
- Let m_0 be the number of input nodes (neurons), and $M = m_L$ the number of output nodes.
- The input-output mapping of the MLP network is from m_0-dimensional input space to M-dimensional output space.
- If the activation function is infinitely continuously differentiable, the mapping is also.
- A fundamental question: *What is the minimum number of hidden layers in a MLP network providing an approximate realization of any continuous mapping?*

Universal Approximation Theorem

- The *universal approximation theorem* for a nonlinear input-output mapping provides the answer.
- The theorem is presented in Haykin's book, pp. 208-209.
- Its essential contents are as follows:
- *Let $\varphi(.)$ be a nonconstant, bounded, and monotonically increasing continuous function.*
- *Let I_{m_0} denote the m_0-dimensional unit hypercube $[0,1]^{m_0}$, and $C(I_{m_0})$ the space of continuous functions on I_{m_0}.*
- *For any given function $f \in C(I_{m_0})$ and $\varepsilon > 0$, there exist an integer M and sets of real constants α_i, b_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ so that:*

Universal Approximation Theorem -2

- *The function*

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi \left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$

is an approximate realization of the function $f(.)$.

- *That is,*

$$| F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0}) | < \varepsilon$$

for all $x_1, x_2, \ldots, x_{m_0}$ *that lie in the input space.*

- The universal approximation theorem is directly applicable to multilayer preceptrons.

Universal Approximation Theorem -3

- The logistic function $\varphi(v) = 1/[1 + \exp(-v)]$ is a nonconstant, bounded, and monotonically increasing function.
- Furthermore, Eq. (4.86) represent the output of a MLP network described as follows:
  1. The network has m_0 input nodes with inputs $x_1, x_2, \ldots, x_{m_0}$, and a single hidden layer consisting of m_1 neurons.
  2. Hidden neuron i has synaptic weights $w_{i1}, \ldots, w_{im_0}$, and bias b_i.
  3. The network output is a linear combination of the outputs of the hidden neurons, with $\alpha_1, \ldots, \alpha_{m_1}$ defining the weights of the output layer.

Universal Approximation Theorem -4

- The universal approximation theorem is an *existence* theorem.

- In effect, the theorem states that *a MLP network with a single hidden layer is sufficient for uniform approximation with accuracy* ε.

- However, the theorem does not say that a single hidden layer is optimal with respect to:
  - learning time
  - ease of implementation
  - generalization ability (most important property).

Practical Consideration

- The universal approximation theorem is important from a theoretical viewpoint.
- It gives a rigorous mathematical foundation for using multilayer perceptrons in approximating nonlinear mappings.
- However, the theorem is not constructive.
- It does not actually tell how to specify a MLP network with the stated approximation properties.
- Some assumptions made in the theorem are unrealistic in most practical applications:
  - The continuous function to be approximated is given.
  - A hidden layer of unlimited size is available.
- A problem with MLP's using a single hidden layer: the hidden neurons tend to interact globally.

Practical Consideration -2

- In complex situations, improving the approximation at one point typically worsens it at some other point.
- With two hidden layers, the approximation (nonlinear mapping) process becomes more manageable.
- One can proceed as follows:
  1. *Local features* are extracted in the first hidden layer.
     - The input space is divided into regions by some neurons.
     - Other neurons in the first hidden layer learn the local features characterizing these regions.
  2. *Global features* corresponding to each region are extracted in the second hidden layer.
     - Neurons in this layer combine the outputs of the neurons describing certain region.
- This procedure somehow corresponds to piecewise polynomial (spline) approximation in curve fitting.

Cross-Validation

- It is hoped that a MLP network trained with back-propagation learns enough from the past to generalize to the future.
- How the network parameterization should be chosen for a specific data set?
- This is a model selection problem: choose the best one of a set of candidate model structures (parameterizations).
- A useful statistical technique for model selection: *cross-validation*.
- The available data set is first randomly partitioned into a training set and a test set.
- The training set is further partitioned into two disjoint subsets:
  - *Estimation subset*, used to select the model.
  - *Validation subset*, used to test or validate the model.

Cross-Validation -2

- The motivation is to validate the model on a data set different from the one used for parameter estimation.
- In this way, the training set can be used to assess the performance of various model.
- The "best" of the candidate models is then chosen.
- This procedure ensures that a model which might in the worst case end up with overfitting the validation subset is not chosen.
- The use of cross-validation is appealing when one should design a large network with good generalization ability.
- For MLP networks, cross-validation can be used to determine:

  - the optimal number of hidden neurons.
  - when it is best to stop training.

Early-Stopping Rule

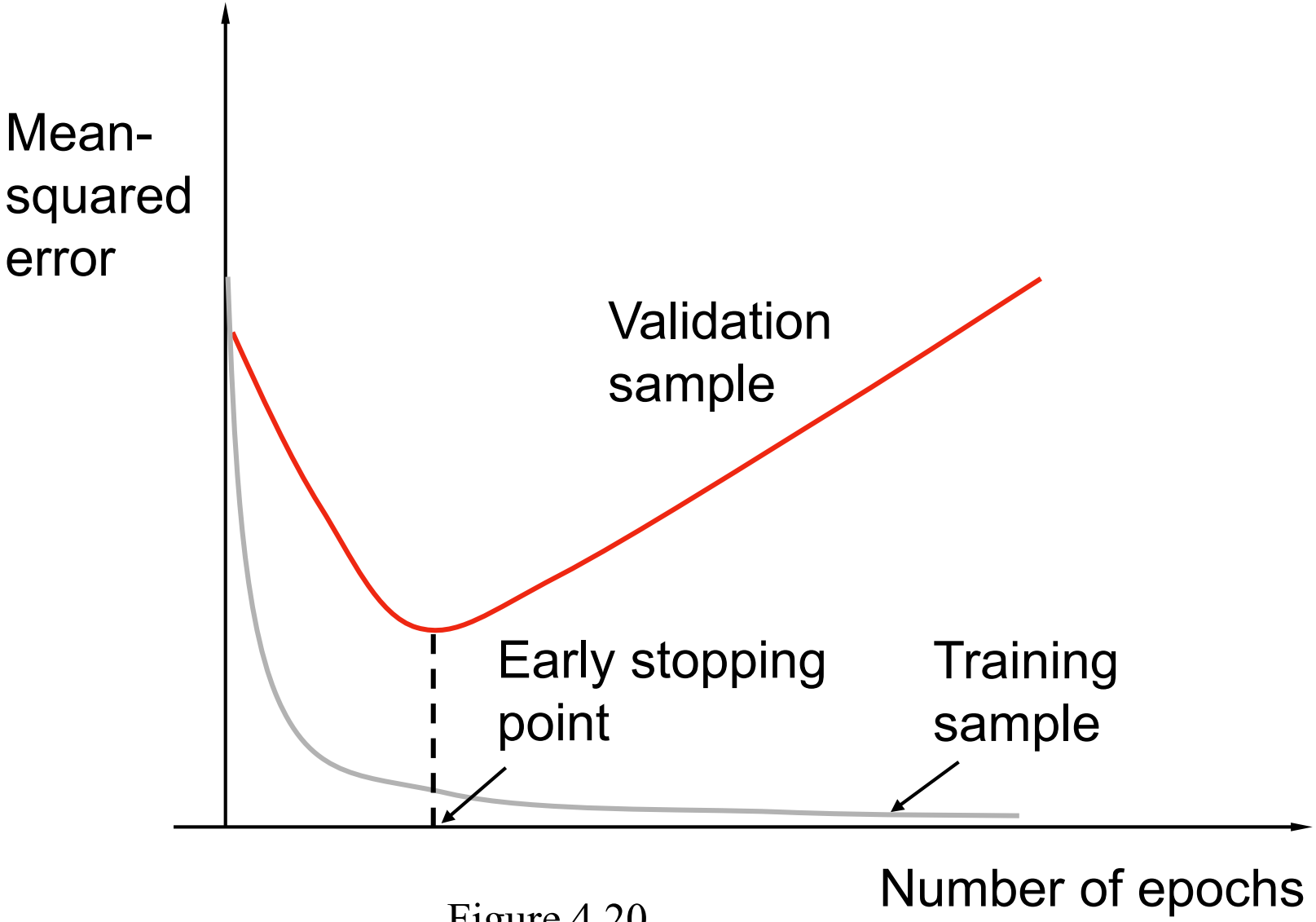
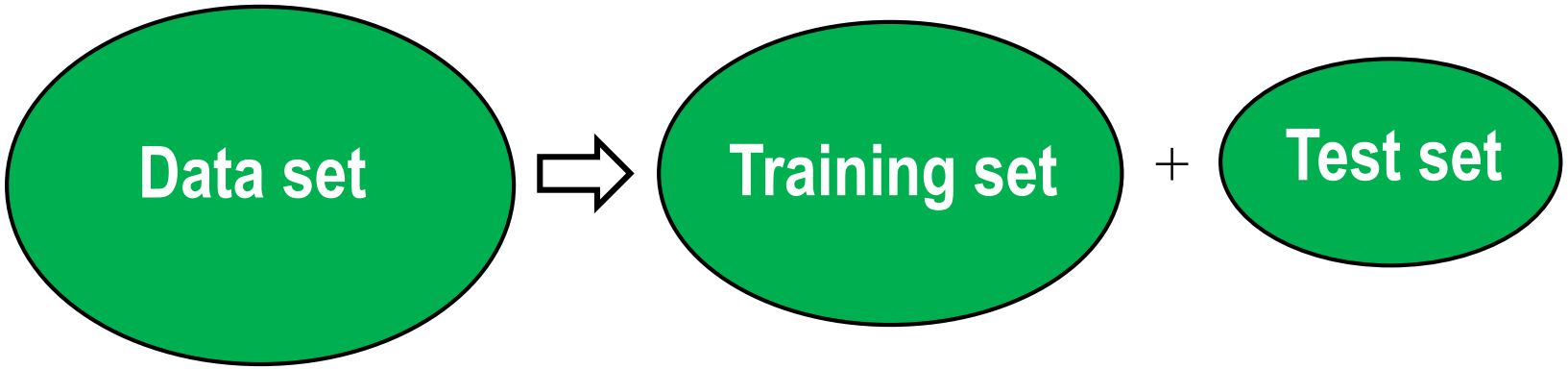


Figure 4.20

Cross-Validation

- Partition of data set



$$T_{All} = T_{Train} \bigcup T_{Test} \; ; \quad T_{Train} \bigcap T_{Test} = \Phi$$

- The training set is further partitioned into two disjoint subsets:
  - Estimation subset, used to select the model
  - Validation subset, used to test or validate the model

Multifold Cross-Validation: Hold-out Method

- Divide data set of N examples into K subsets, 1<K<=N
- The model is trained on all the subsets except for one
- The validation error is measured by testing it on the subset left out
- This procedure is repeated for a total of K trials, each time using a different subset for validation
- The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment
- Leave-one-out method: K=N

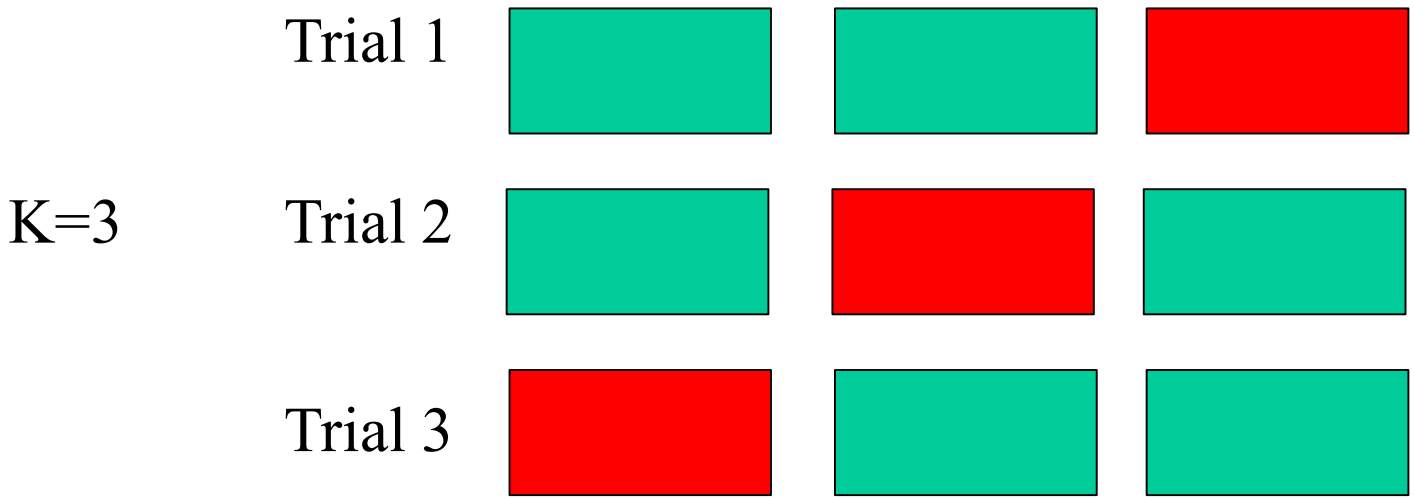



Illustration of the hold-out method