VI ESCOLA AVANÇADA DE BIG DATA ANALYSIS

MULTI-LAYER PERCEPTRONS

RICARDO CERRI

Departament of Computer Science
Universidade Federal de São Carlos

Multilayer Perceptron

- Overcomes Perceptron's practical limitations
  - The model of each neuron includes a nonlinear and differentiable activation function
  - Contains one or more hidden layers between the input layer and the output layer
  - The network has a high degree of connectivity

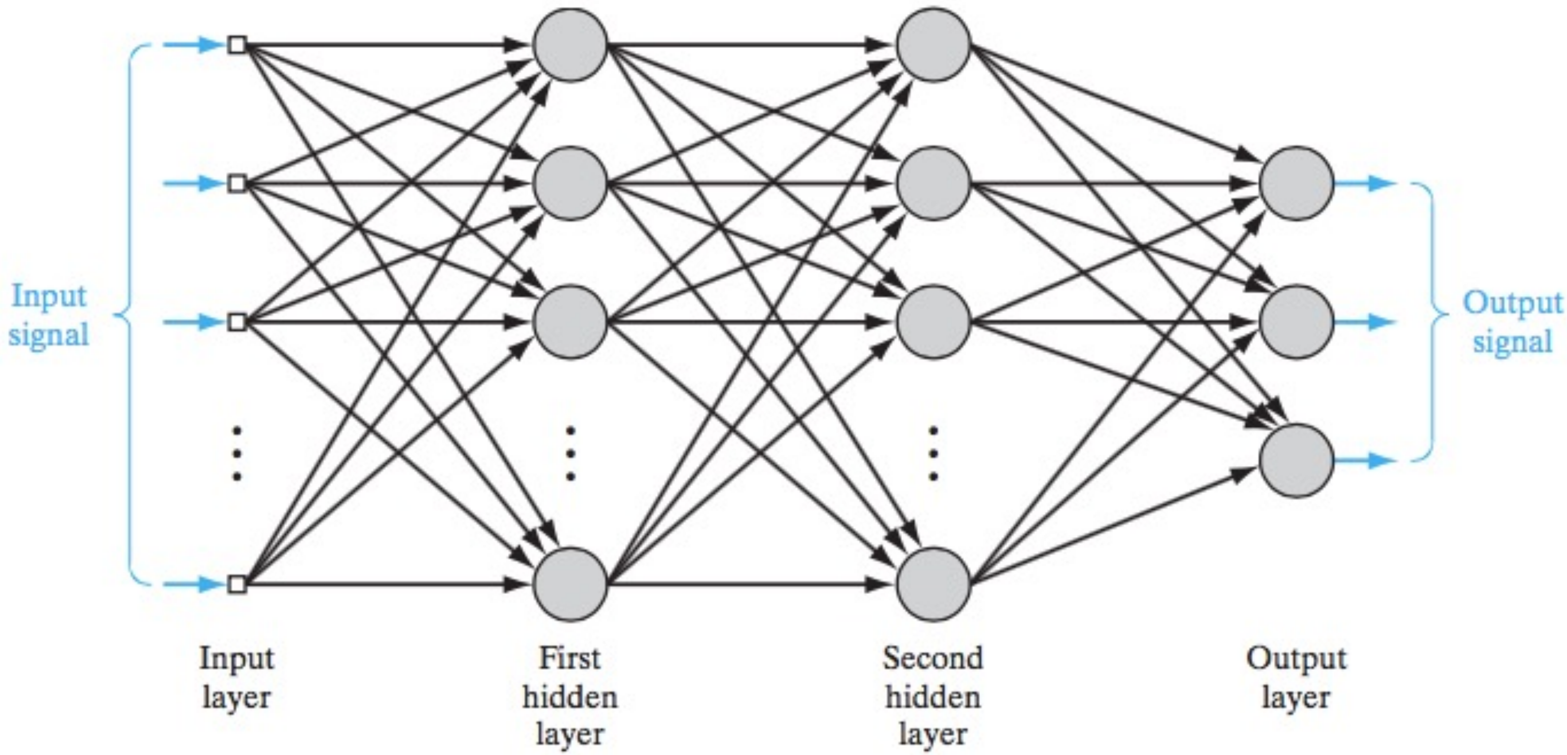
Multilayer Perceptron

- How to learn? Back-propagation
  - **Forward phase:** fixed weights, and the signal is propagated through the network, layer by layer, until the exit
  - Changes only occur in the activation potentials and in the outputs of the neurons in the network

Multilayer Perceptron

- How to learn? Back-propagation
  - **Backward phase:** an error signal is produced by comparing the desired output with the obtained output
  - The error is propagated back through the network, layer by layer
  - Adjustments are made to the synaptic weights of the network

Multilayer Perceptron



Multilayer Perceptron

- Function Signals and Error Signals

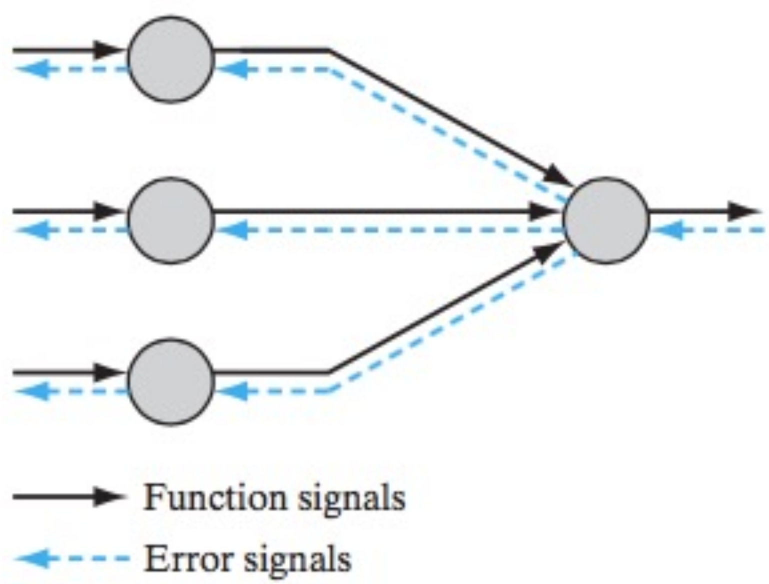


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back propagation of error signals.

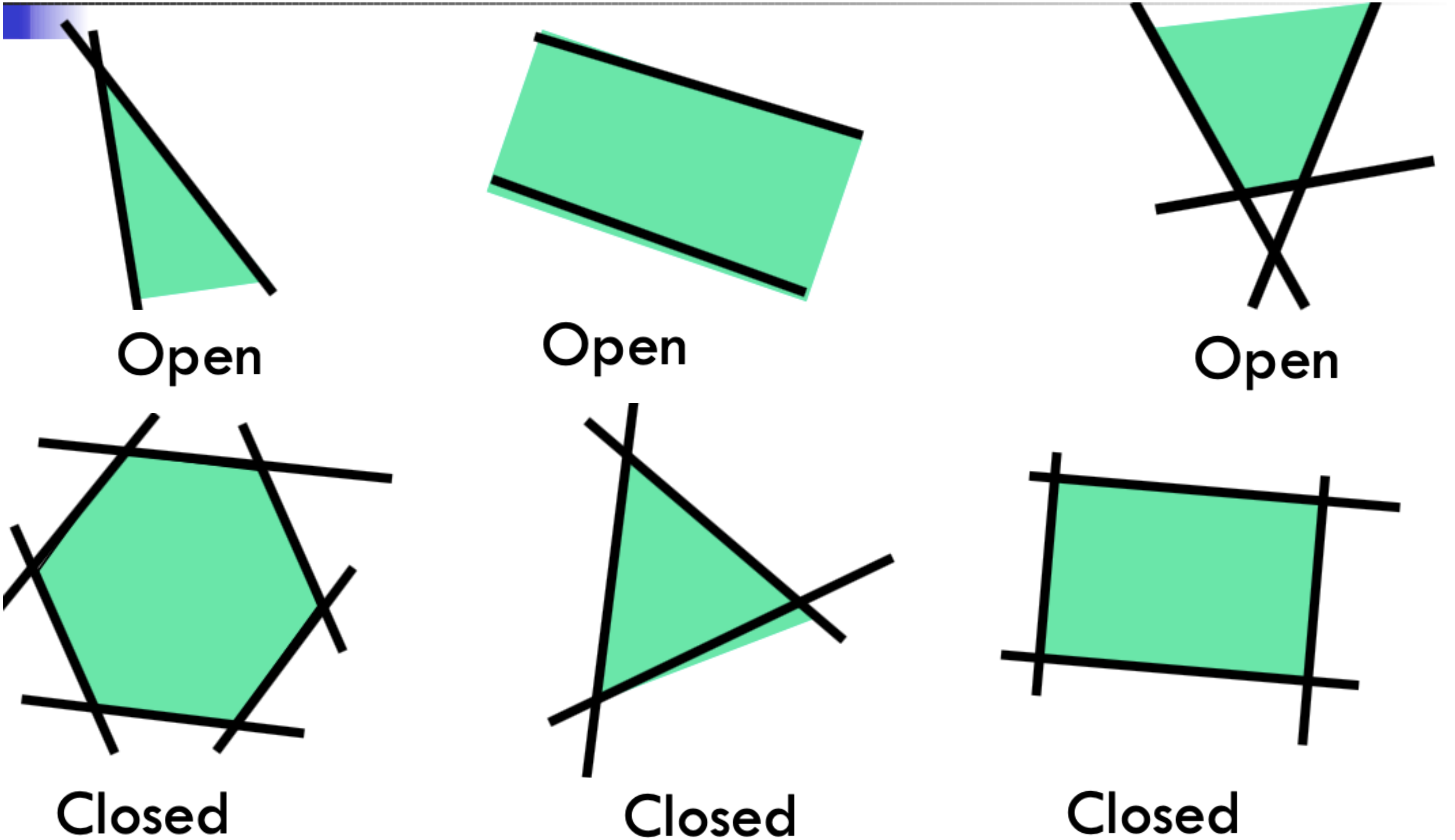
Multilayer Perceptron

- Function of hidden neurons
  - They act as attribute detectors. As learning progresses, these neurons begin to discover the attributes that characterize the training data
  - This is done through the nonlinear transformation of the input data into a new space called the feature space
  - In this new space, classes (for example in a classification problem) can be more easily separated from each other than in the original input space

Multilayer Perceptron

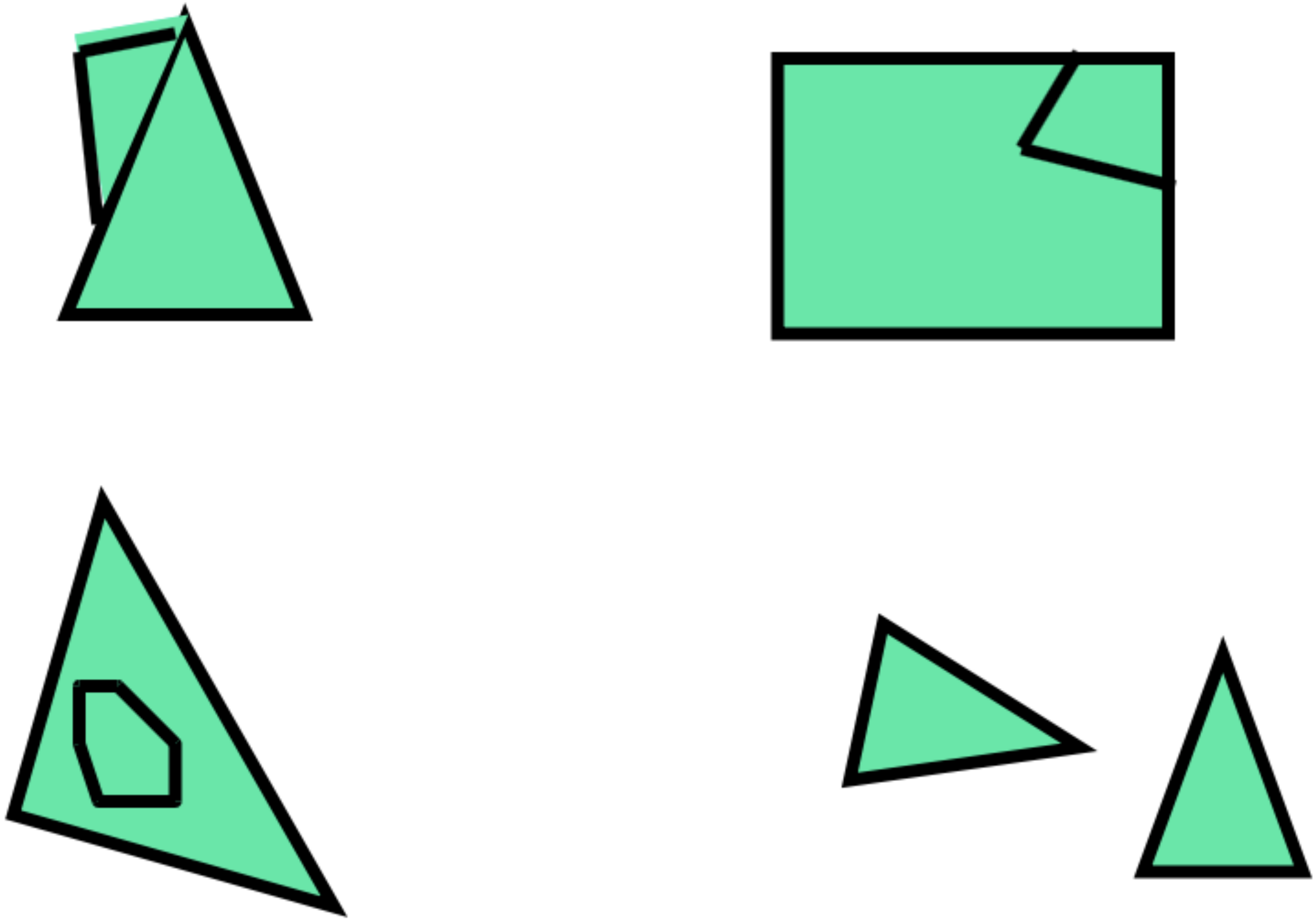
- Intermediate layers
  - First layer: straight lines in the decision space
  - Second layer: combines the lines of the previous layer to form convex regions
  - Third layer: combines convex figures producing abstract shapes

Multi-layer Perceptrons – Convex regions

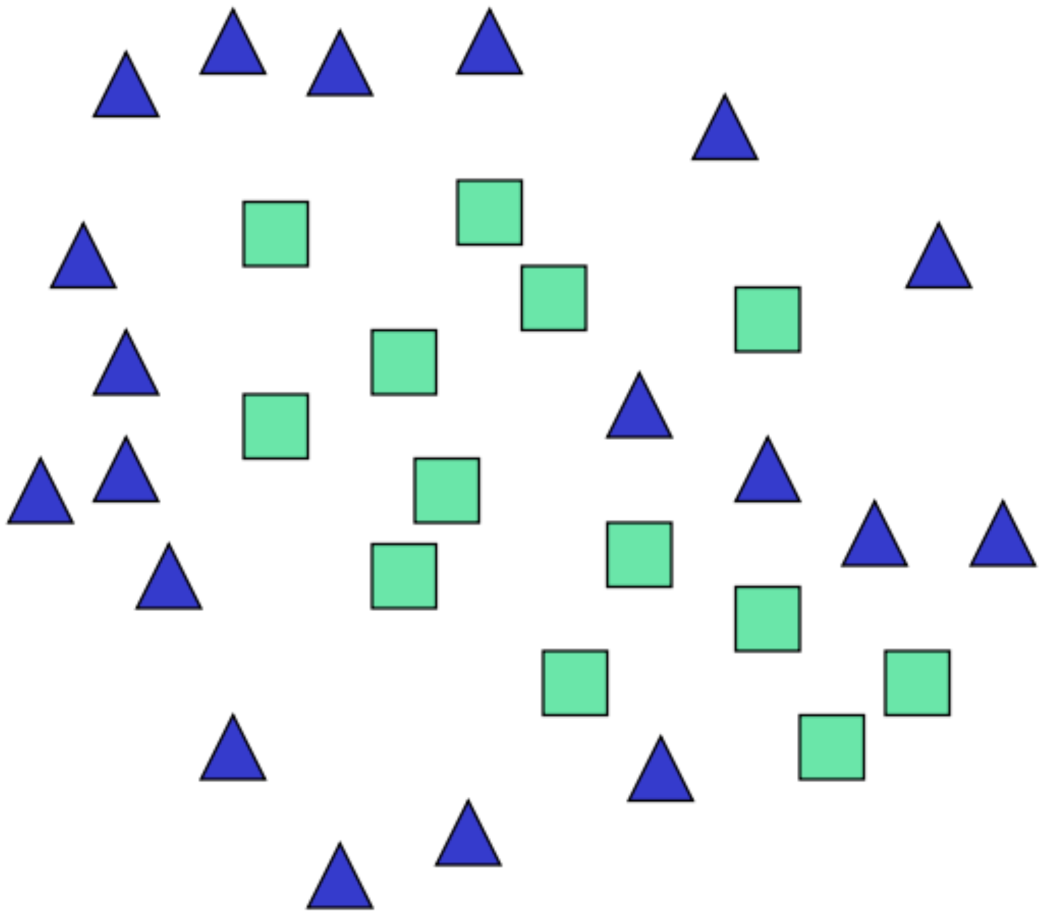


Multi-layer Perceptrons

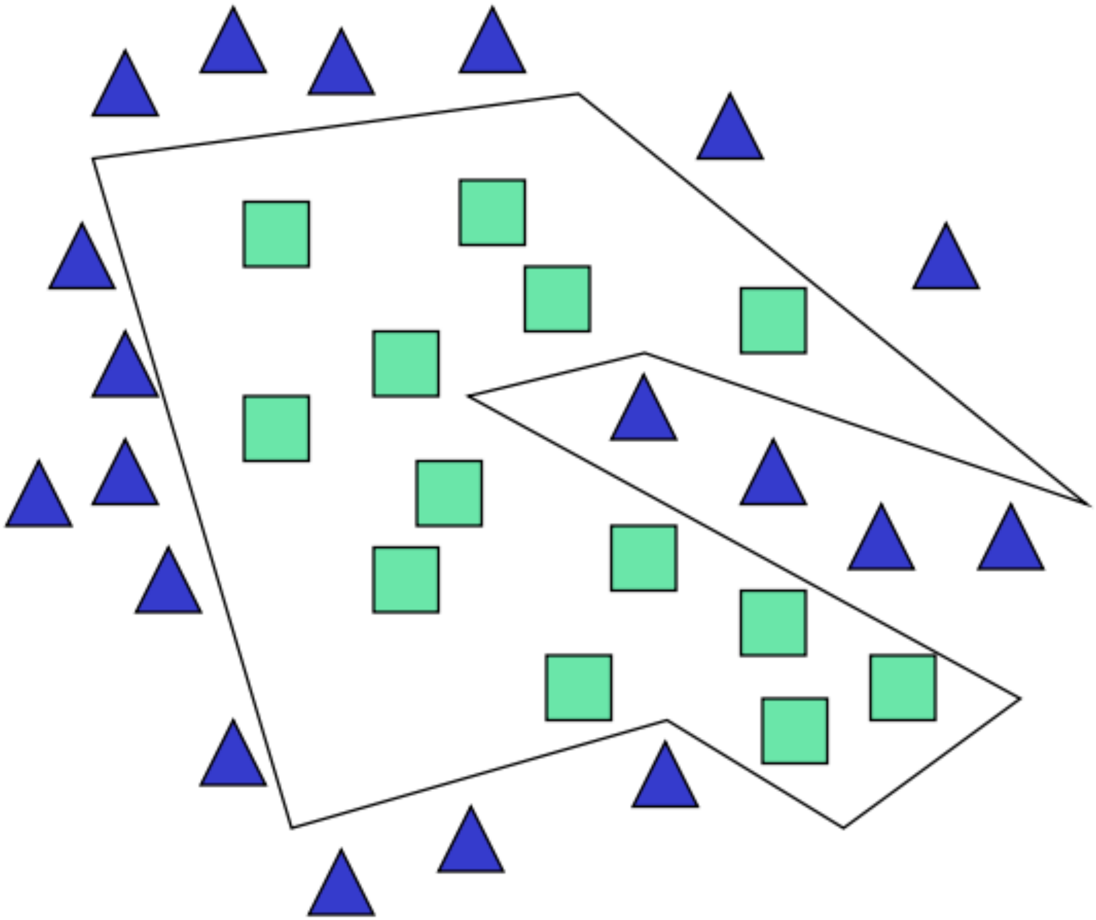
Combinations of convex regions



Multilayer Perceptron



Multilayer Perceptron



Multilayer Perceptron

- Consider a Multilayer Perceptron.

- Consider $\tau = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N}$ a training instance. Being $y_j(n)$ the signal produced in the output of neuron j in the output layer, stimulated by $\mathbf{x}(n)$ applied in the input layer

- The error signal produced at the output of neuron j is given by $e_j(n) = d_j(n) - y_j(n)$

Multilayer Perceptron

- The error signal produced at the output of neuron j is given by $e_j(n) = d_j(n) - y_j(n)$, where $d_j(n)$ is the j–ih element of the vector of desired responses $\mathbf{d}(n)$

- The instantaneous error of neuron j is given by

$$\varepsilon_j(n) = \frac{1}{2} e_j^2(n)$$

Multilayer Perceptron

- Adding the errors of all neurons in the output layer, the total error of the entire network is given by

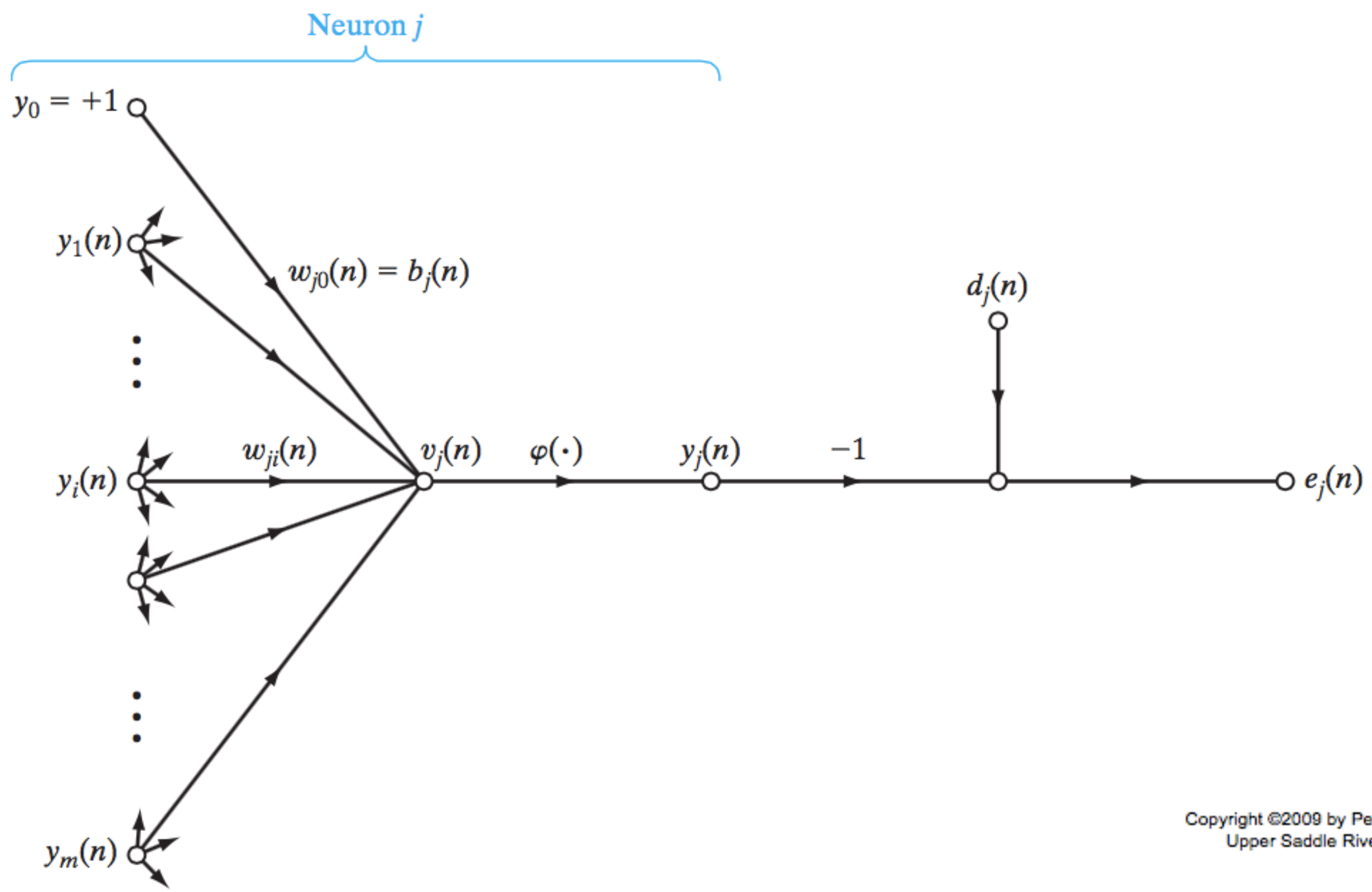
$$\varepsilon(n) = \sum_{j \in C} \varepsilon_j(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n)$$

- C is the set of all output neurons. In a training set with N examples, the average error over all examples (empirical risk) is given by

$$\varepsilon_{av}(N) = \frac{1}{N} \sum_{n=1}^{N} \varepsilon(n) = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n)$$

The Back-propagation Algorithm

- Neuron j being fed by a set of function signals



The Back-propagation Algorithm

- The induced local field (activation potential) $v_j(n)$ produced at the input of the associated activation function is given by

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

  - m : number of inputs (excluding the bias)
  - w_{j0} : weight applied to the fixed input $y_0=+1$ (bias)

The Back-propagation Algorithm

- The signal $y_j(n)$ in the output of neuron j at iteration n is :

$$y_j(n) = \varphi_j(v_j(n))$$

- The algorithm applies a correction $\Delta w_{ji}(n)$ in the synaptic weight $w_{ji}(n)$, proportional to the partial derivative:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

The Back-propagation Algorithm

- The partial derivative $\partial\varepsilon(n)/\partial w_{ji}(n)$ determines the search direction for w_{ji} in the weight space

- Differentiating the equation below on both sides with respect to $e_j(n)$:

$$\varepsilon(n) = \sum_{j \in C} e_j^2(n) \rightarrow \frac{\partial\varepsilon(n)}{\partial e_j(n)} = e_j(n)$$

The Back-propagation Algorithm

- The partial derivative $\partial \varepsilon(n) / \partial w_{ji}(n)$ determines the search direction for w_{ji} in the weight space

- Differentiating the equation below on both sides with respect to $y_j(n)$:

$$e_j(n) = d_j(n) - y_j(n) \rightarrow \frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

The Back-propagation Algorithm

- The partial derivative $\partial \varepsilon(n) / \partial w_{ji}(n)$ determines the search direction for w_{ji} in the weight space

- Differentiating the equation below on both sides with respect to $v_j(n)$:

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

The Back-propagation Algorithm

- The partial derivative $\partial \varepsilon(n) / \partial w_{ji}(n)$ determines the search direction for w_{ji} in the weight space

- Differentiating the equation below on both sides with respect to $w_{ji}(n)$:

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \rightarrow \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

The Back-propagation Algorithm

- The partial derivative $\partial \varepsilon(n) / \partial w_{ji}(n)$ determines the search direction for w_{ji} in the weight space

- Substituting the equations obtained in the chain rule equation, we obtain

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

The Back-propagation Algorithm

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

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)}$$

  - η: learning rate of the algorithm
  - The negative sign refers to the gradiente descent in the weight space

- Since $\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

The Back-propagation Algorithm

- The local gradient $\delta_j(n)$ is defined by

$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= e_j(n)\varphi_j'(v_j(n))$$

- The local gradient defines the necessary change in the weights. It is given by the product of the error and the derivative of activation function in the neuron

The Back-propagation Algorithm

- The error signal of the output neuron is the key factor in calculating the weight adjustment

- Thus, two cases for calculating the error can be identified

  - The neuron is located in the last layer (output)

  - The neuron is located in a hidden layer

The Back-propagation Algorithm

- Neuron j in na output layer

  - In this case, the neuron is directly associated with the desired output. Thus, the error is calculated directly :

$$e_j(n) = d_j(n) - y_j(n)$$

  - After calculated the error, the local gradient is calculated directly:

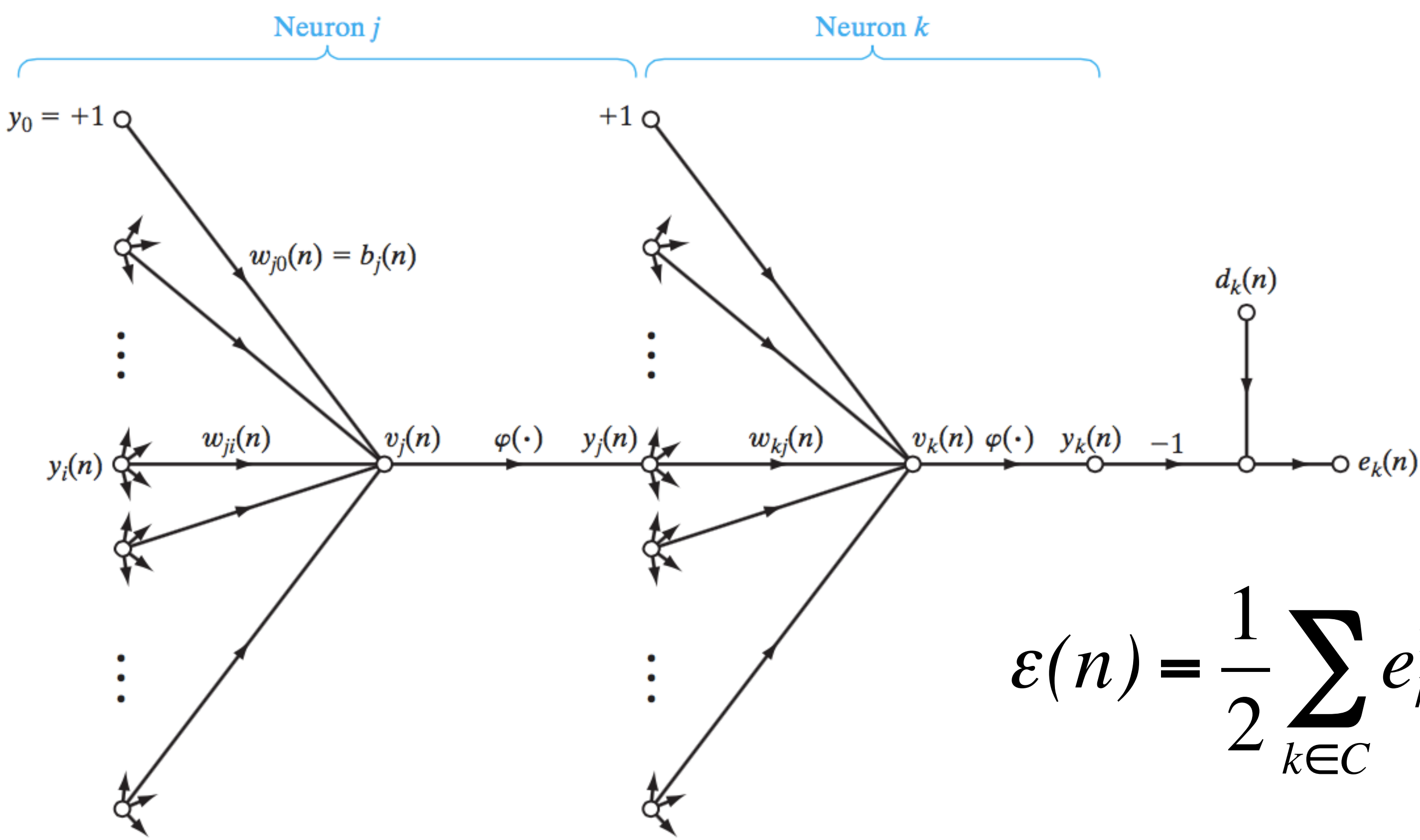
$$\delta_j(n) = e_j(n)\varphi'_j(v_j(n))$$

The Back-propagation Algorithm

- Neuron j is a hidden neuron
  - In this case, there is no specific desired output associated with the neuron.
  - The error signal must be recursively calculated, in terms of the error signals of all neurons which to neuron j is directly connected
  - At this point the development of back-propagation becomes more complicated

The Back-propagation Algorithm

- Neuron j is a hidden neuron

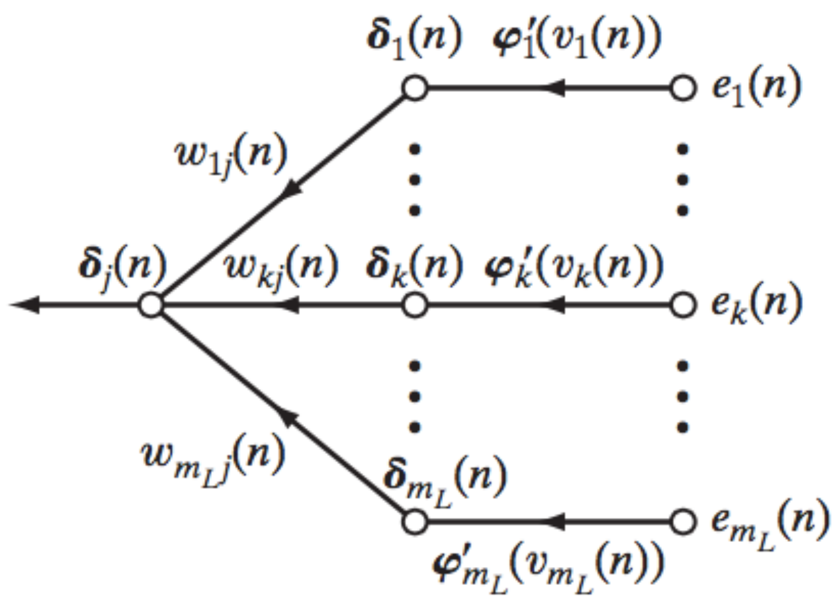


$$\varepsilon(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$$

The Back-propagation Algorithm

- Neuron j is a hidden neuron
  - Graphical representantion of

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



The Back-propagation Algorithm

- **Summarizing:** the correction applied to the weights connecting a neuron i to a neuron j is given by the dealt rule

$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{Learning} \\ \text{rate} \\ \eta \end{pmatrix} \times \begin{pmatrix} \text{Local} \\ \text{gradient} \\ \partial_j(n) \end{pmatrix} \times \begin{pmatrix} \text{Input signal} \\ \text{Neuron } j \\ y_i(n) \end{pmatrix}$$

- **Output:** $\delta_j(n)$ is the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both associated to neuron j

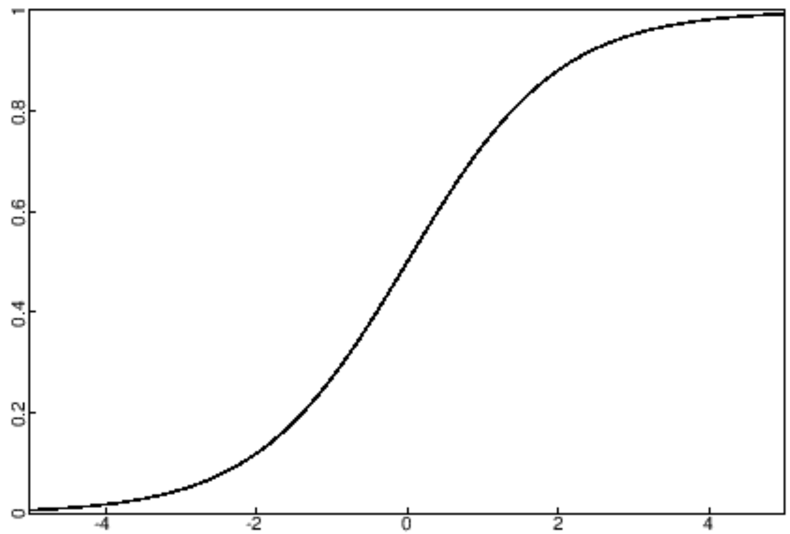
- **Hidden:** $\delta_j(n)$ is the product of the derivative associated to $\varphi_j'(v_j(n))$ and the weighted sum of the δs calculated for the neurons of the next layer, or the output layer, connected to neuron j

Activation Functions

- To calculate δ for each neuron, we need to know the derivative of the activation function $\varphi(\cdot)$ associated with the neuron
- For the derivative to exist, $\varphi(\cdot)$ must be continuous
- The, **differentiability** is the only requirement the function has to satisfy
- A common continuously differentiable nonlinear function: **sigmoidal**

Activation Functions

- **Logistic Function:** $\varphi_j(v_j(n)) = \dfrac{1}{1 + exp(-av_j(n))}, \qquad a > 0$
  - Output signal amplitude : $0 \leq y_j \leq 1$



Activation Functions

- **Logistic Function:** $\varphi_j(v_j(n)) = \dfrac{1}{1+exp(-av_j(n))}, \qquad a > 0$
  - Output signal amplitude : $0 \le y_j \le 1$
  - The derivative of the function with respect to $v_j(n)$ gives:

$$\varphi_j'(v_j(n)) = \frac{a\ \ exp(-av_j(n))}{\left[1+exp(-av_j(n))\right]^2}$$

  - With $y_j(n) = \varphi_j(v_j(n))$, we can rewrite the derivative:

$$\varphi_j'(v_j(n)) = ay_j(n)\left[1-y_j(n)\right]$$

Activation Functions

$$\varphi_j'(v_j(n)) = a y_j(n)\left[1 - y_j(n)\right]$$

- For a neuron j of the output layer, $y_j(n) = o_j(n)$. The local gradiente of neuron j is given by:

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= a\left[d_j(n) - o_j(n)\right]o_j(n)\left[1 - o_j(n)\right]\end{aligned}$$

Activation Functions

$$\varphi_j'(v_j(n)) = ay_j(n)\left[1 - y_j(n)\right]$$

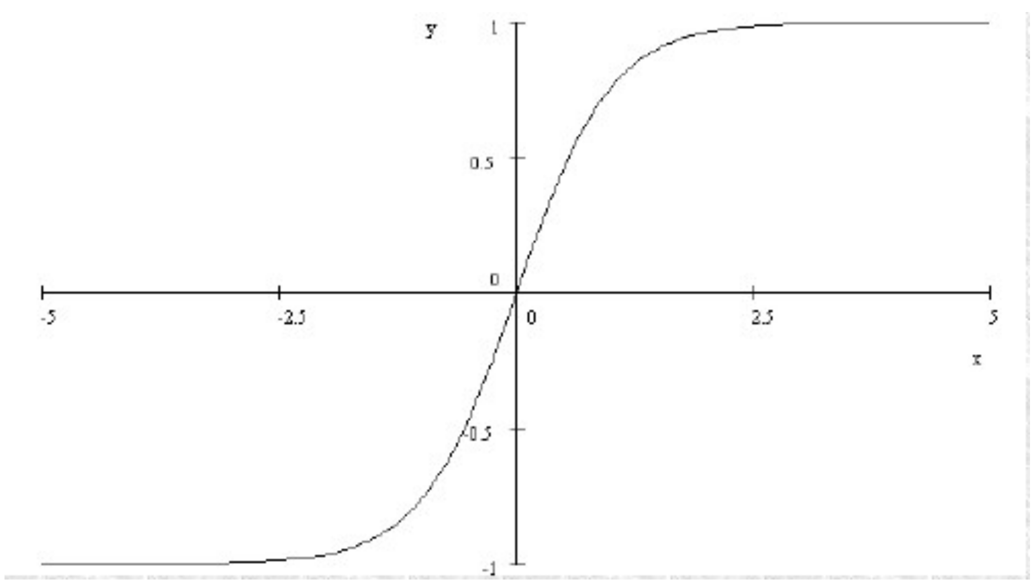
- From a neuron j of a hidden layer, the local gradiente of neuron j is given by:

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

$$= ay_j(n)\left[1 - y_j(n)\right]\sum_k \delta_k(n)w_{kj}(n)$$

Activation Functions

- **Hyperbolic tangent:** $\varphi_j(v_j(n)) = a\tanh(bv_j(n))$
  - *a* and *b* are positive constants
  - Output signal amplitude: $-a \le y_j \le a$



Activation Functions

- **Hyperbolic tangent:** $\varphi_j(v_j(n)) = a\,tanh(bv_j(n))$
  - *a* and *b* are positive constants
  - Output signal amplitude: $-1 \leq y_j \leq 1$

  - The derivative of the function with respect to $v_j(n)$ gives:

$$\varphi_j'(v_j(n)) = ab\ sech^2(bv_j(n))$$
$$= ab\left(1 - tanh^2(bv_j(n))\right)$$
$$= \frac{b}{a}\left[a - y_j(n)\right]\left[a + y_j(n)\right]$$

Activation Functions

$$\varphi_j'(v_j(n)) = \frac{b}{a}\left[a - y_j(n)\right]\left[a + y_j(n)\right]$$

- For a neuton j of the output layer, the local gradiente of neuron j is given by:

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

$$= \frac{b}{a}\left[d_j(n) - o_j(n)\right]\left[a - o_j(n)\right]\left[a + o_j(n)\right]$$

Activation Functions

$$\varphi_j'(v_j(n)) = \frac{b}{a}\left[a - y_j(n)\right]\left[a + y_j(n)\right]$$

- For a neuron j of a hidden layer, the local gradiente of neuron j is given by:

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

$$= \frac{b}{a}\left[a - y_j(n)\right]\left[a + y_j(n)\right]\sum_k \delta_k(n) w_{kj}(n)$$

Learning Rate

- The lower the learning rate, the smaller the change in the synaptic weights of the network and the smoother the trajectory in the search space will be.
  - Learning will be slower

- Increasing the learning rate, we have a faster learning, with major changes in synaptic weights
  - The network may become unstable

Learning Rate

- How to increase the learning rate without losing stability?
  - Momentum constant α : positive number

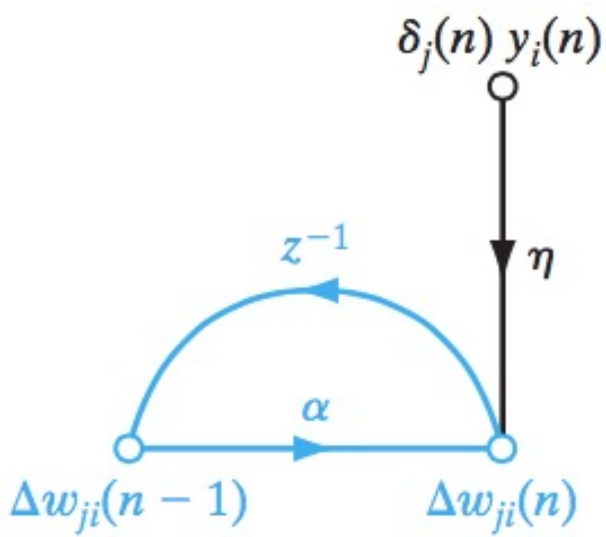
$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

  - Controls adjustment $\Delta w_{ji}(n)$

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

$$-\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \delta_j(n)y_i(n)$$



Learning Rate

- When the partial derivative $\partial\varepsilon(n)/\partial w_{ji}(n)$ has the same sign in consecutive iterations, $\Delta w_{ji}(n)$ grows, and $w_{ji}(n)$ is adjusted by a large amount. Thus the momentum constant accelerates the algorithm in regions of constant descent on the error surface.

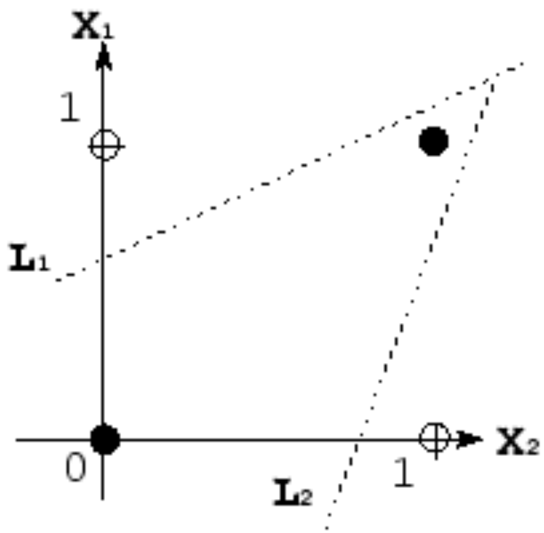
- If the partial derivative $\partial\varepsilon(n)/\partial w_{ji}(n)$ has opposite signs in consecutive iterations, $\Delta w_{ji}(n)$ shrinks, and $w_{ji}(n)$ is adjusted in small quantities. Thus, the momentum constant has a stabilizing efect in directions in which the signal oscillates.

The XOR Problem

- Rosenblatt's Perceptron cannot classify nonlinearly separable patterns

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

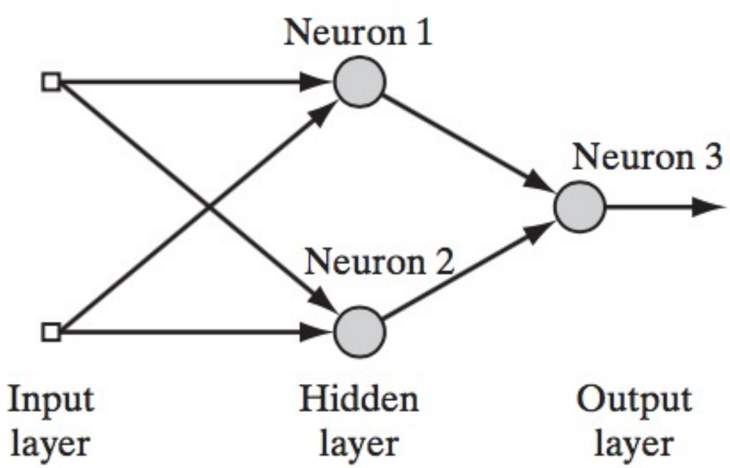
$$Y = X_1 \oplus X_2$$

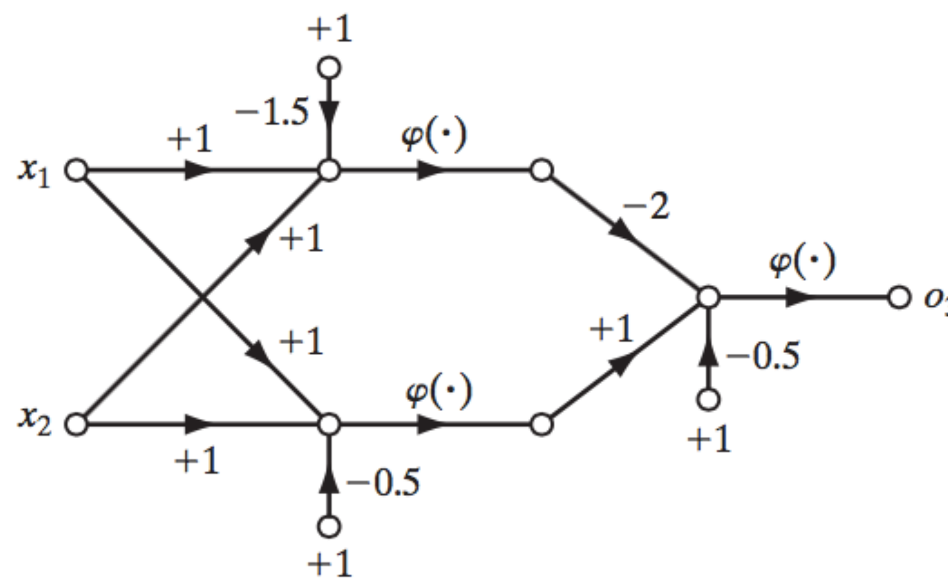


  - A single Perceptron can trace only one hyperplane

The XOR Problem

- We can solve the problem using a hidden layer with two neurons





$$w_{11} = w_{12} = +1 \qquad b_1 = -\frac{3}{2}$$

$$w_{21} = w_{22} = +1 \qquad b_2 = -\frac{1}{2}$$

$$w_{31} = -2 \qquad w_{32} = +1 \qquad b_3 = -\frac{1}{2}$$

The XOR Problem

- We can solve the problem using a hidden layer with two neurons

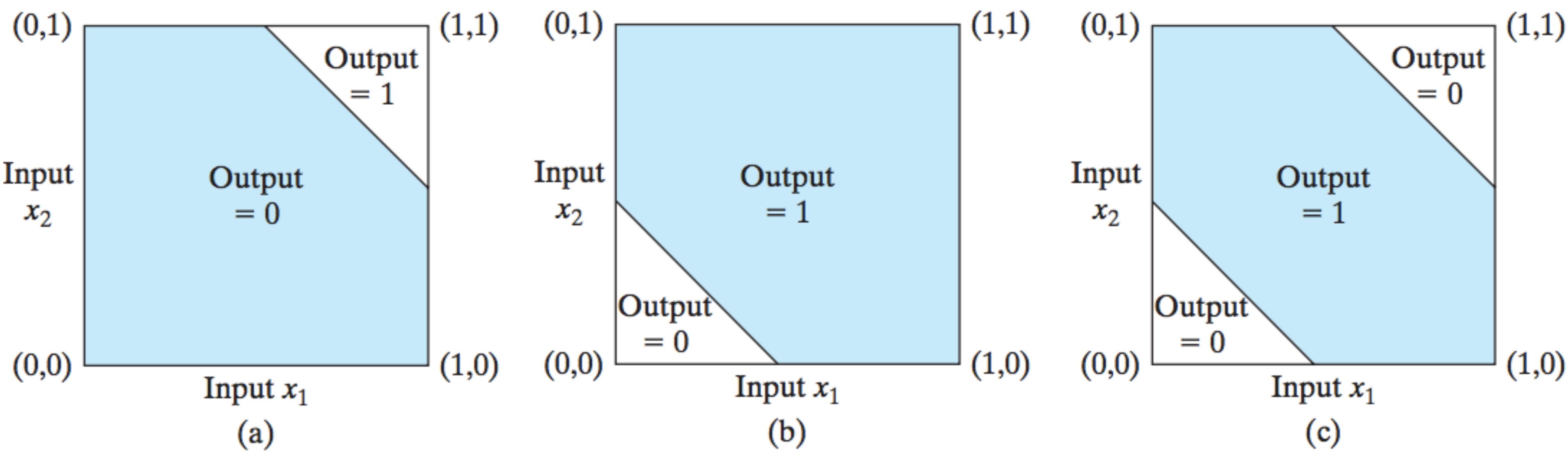


FIGURE 4.9 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8. (b) Decision boundary constructed by hidden neuron 2 of the network. (c) Decision boundaries constructed by the complete network.

Thank you!

BIOMaL
BIOINFORMATICS AND MACHINE LEARNING GROUP

Departamento de Computação

ufscar

cerri@ufscar.br - www.biomal.ufscar.br