Lecture 11

Artificial neural networks: Supervised learning

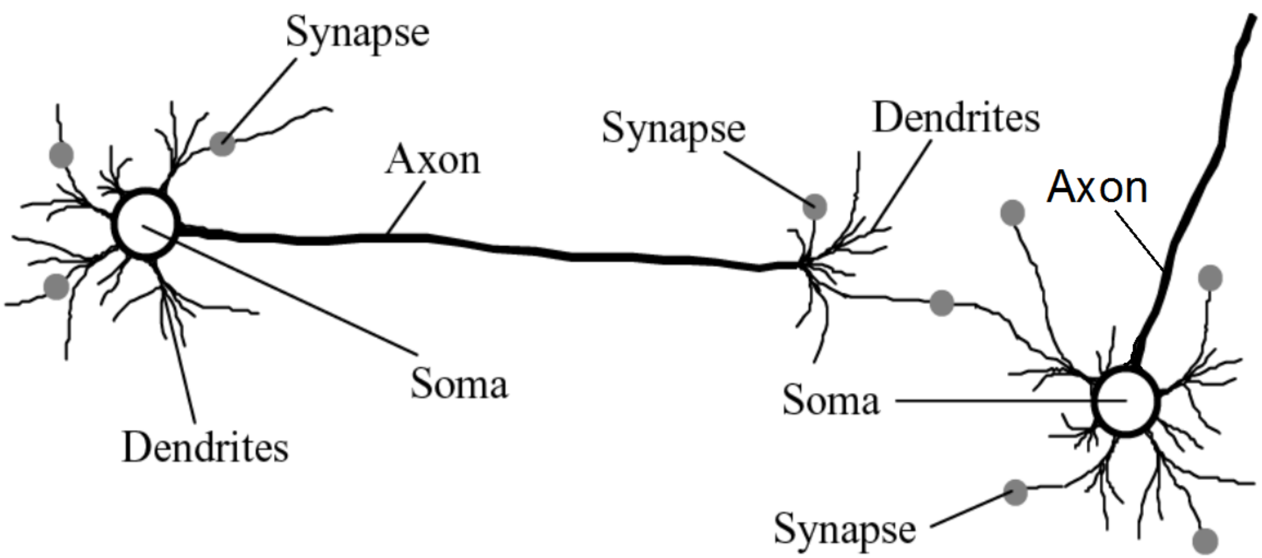
- Introduction, or how the brain works
- The neuron as a simple computing element
- The perceptron and its learning algorithm
- Multilayer neural networks
- Accelerated learning in multilayer neural networks

Introduction, or how the brain works

Machine learning involves adaptive mechanisms that enable computers to **learn from experience, learn by example and learn by analogy**. Learning capabilities can improve the performance of an intelligent system over time. The most popular approaches to machine learning is **artificial neural networks**. This lecture is dedicated to neural networks.

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

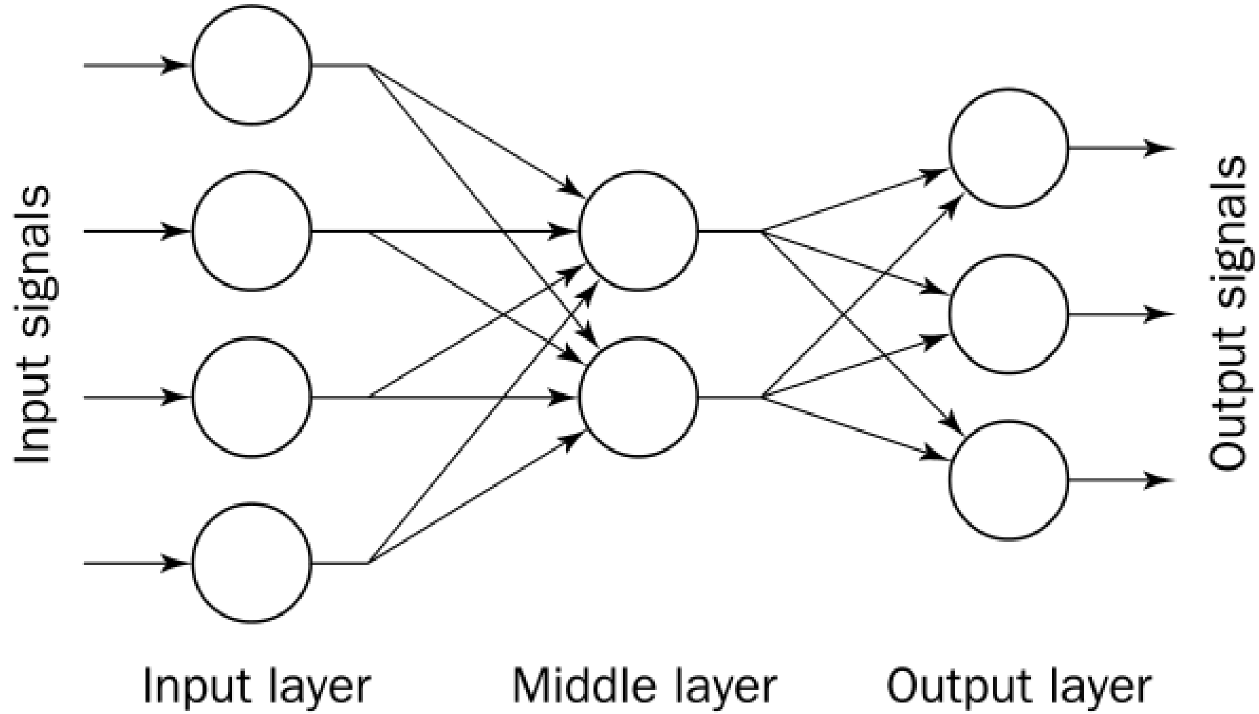
- Each neuron has a very simple structure, but an army of such elements constitutes a wonderful processing power.
- A neuron consists of a cell body, **soma**, a number of fibers called **dendrites**, and a single long fiber called the **axon**.



- Our brain can be considered as a highly complex, non-linear and parallel information-processing system.
- Information is stored and processed in a neural network simultaneously throughout the whole network, rather than at specific locations. In other words, in neural networks, both data and its processing are **global** rather than local.
- Learning is a fundamental and essential characteristic of biological neural networks. Bio neural networks' cap in learning led to attempts to emulate a biological neural network in a computer.

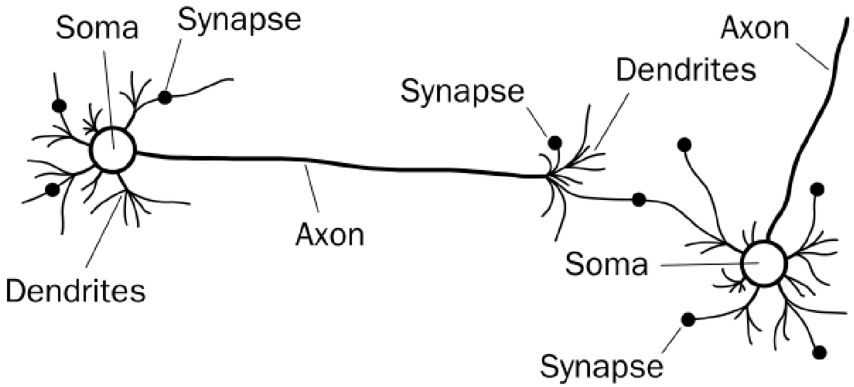
- An artificial neural network consists of a number of very simple processors, also called **neurons**, which are **analogous** to the biological neurons in the brain.
- The neurons are connected by weighted links passing signals from one neuron to another.
- The output signal is transmitted through the neuron's outgoing connection. The outgoing connection splits into a number of branches that transmit the same signal. The outgoing branches terminate at the incoming connections of other neurons in the network.

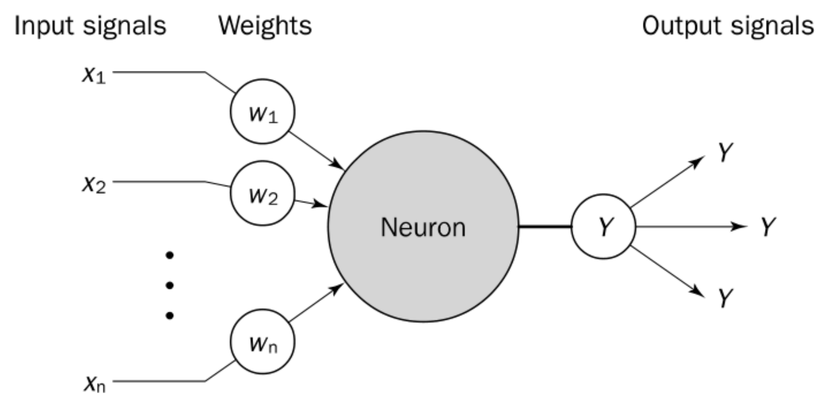
Architecture of a typical artificial neural network



Analogy between biological and artificial neural networks

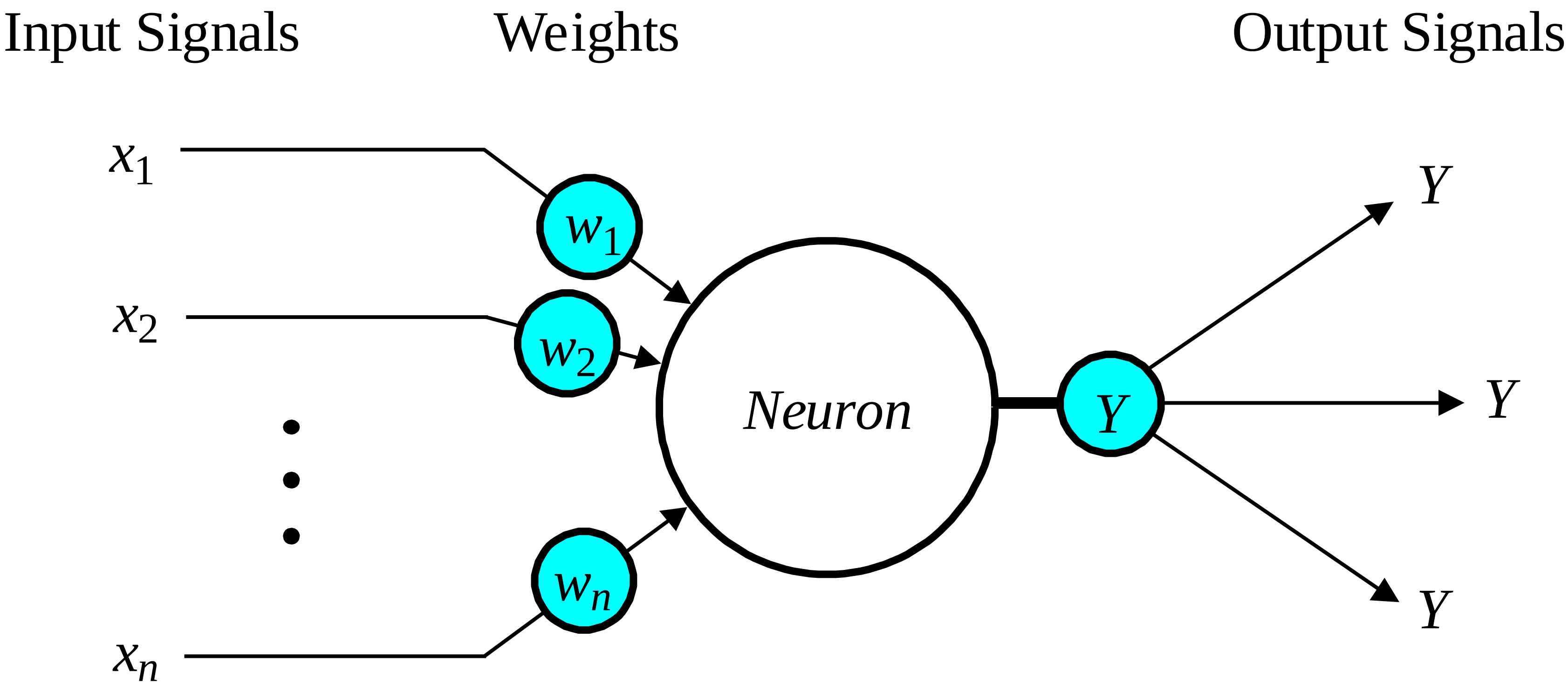
Biological neural network	Artificial neural network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight





The neuron as a simple computing element

Diagram of a neuron

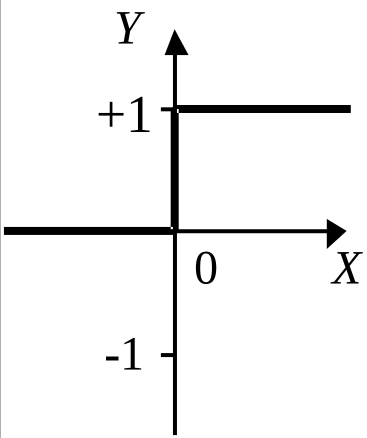
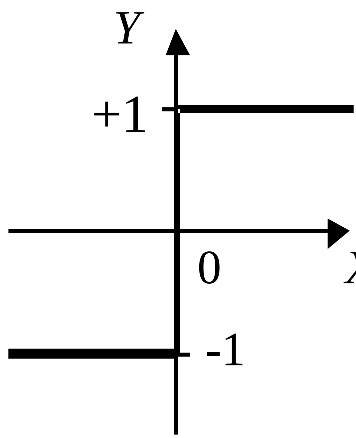
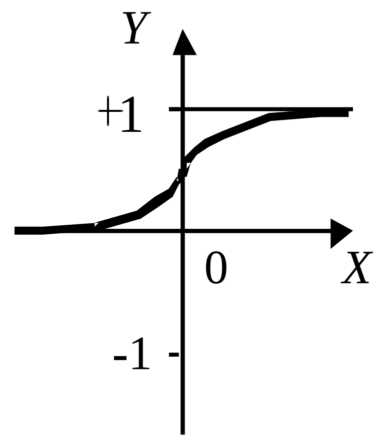
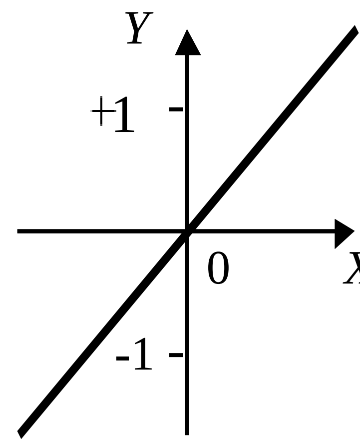


- The neuron computes the **weighted sum** of the input signals and compares the result with a **threshold value, θ**. If the net input is less than the threshold, the neuron output is –1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains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

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

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

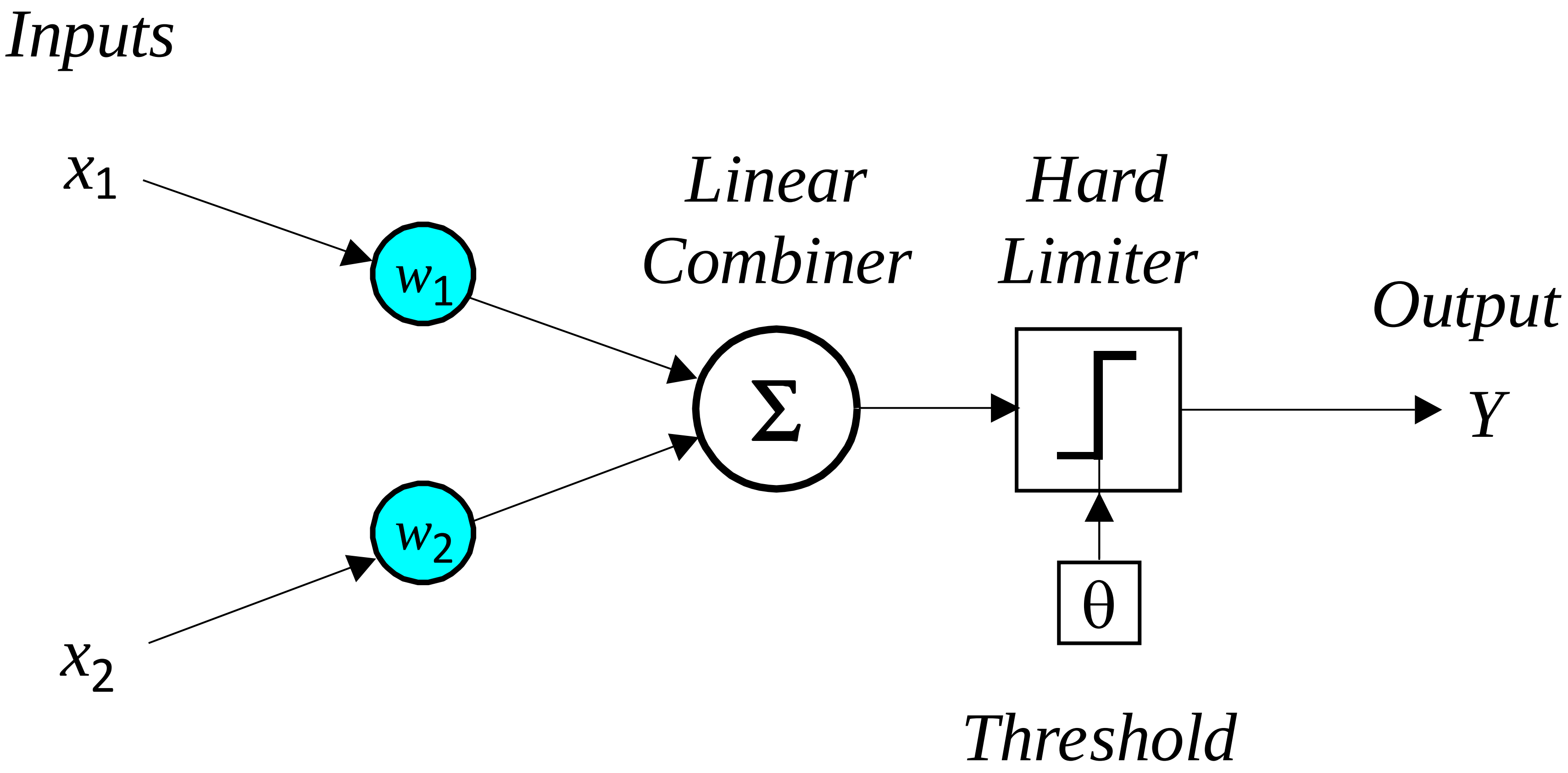
Activation functions of a neuron

Step function	*Sign function*	*Sigmoid function*	*Linear function*
			
$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{ if } X < 0 \end{cases}$	$Y^{sign} = \begin{cases} +1, \text{ if } X \geq 0 \\ -1, \text{ if } X < 0 \end{cases}$	$Y^{sigmoid} = \dfrac{1}{1+e^{-X}}$	$Y^{linear} = X$

Can a single neuron learn a task?

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

Single-layer two-input perceptron



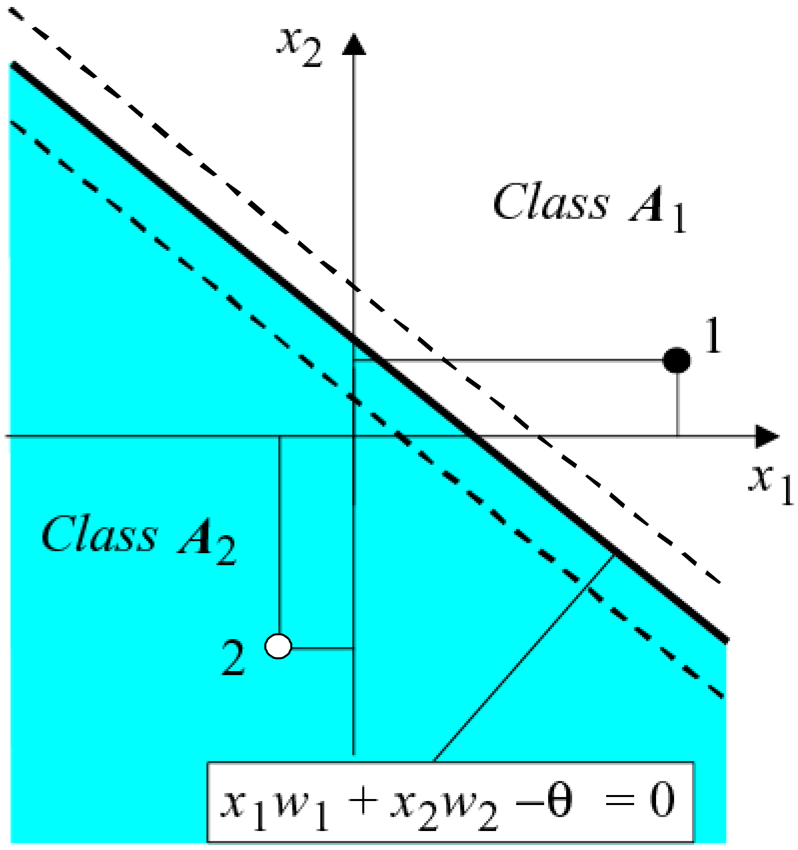
The Perceptron

- The operation of Rosenblatt's perceptron is based on the **McCulloch and Pitts neuron model**. The model consists of a linear combiner followed by a hard limiter.
- The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to +1 if its input is positive and –1 if it is negative.

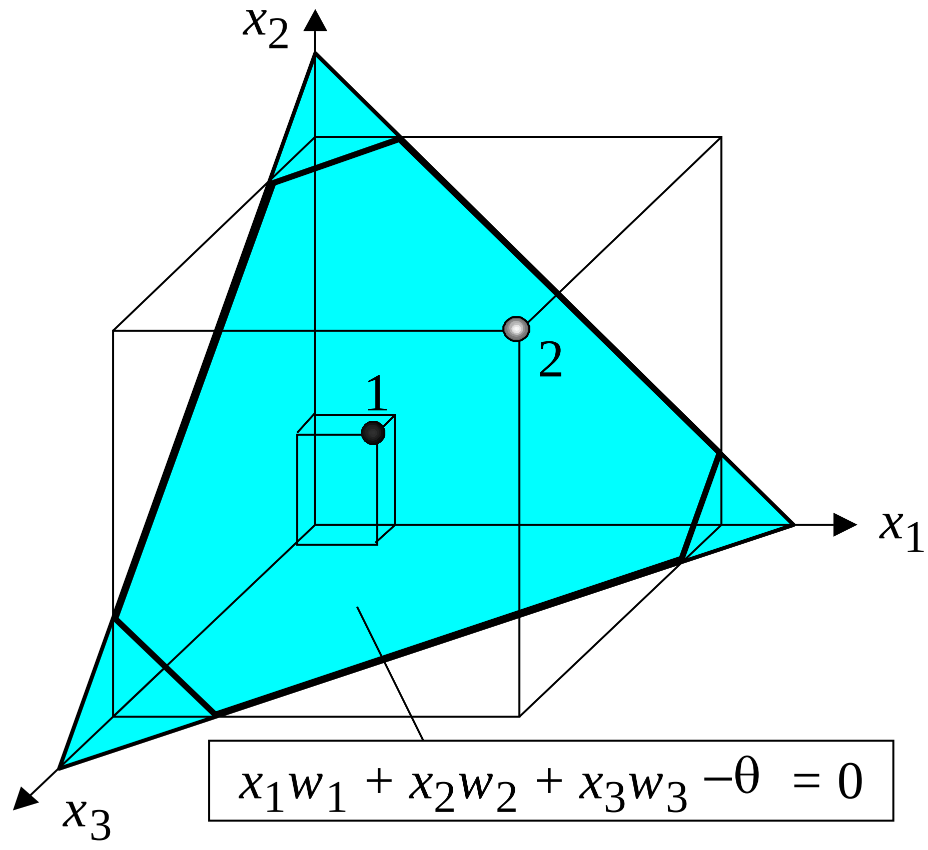
- The **aim of the perceptron** is to classify inputs, $x_1, x_2, \ldots, x_n$, into one of two classes, say $\mathbf{A}_1$ and $\mathbf{A}_2$.
- In the case of an elementary perceptron, the n-dimensional space is divided by a hyperplane into two decision regions. The hyperplane is defined by the **linearly separable function**:

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

Linear separability in the perceptrons



(*a*) Two-input perceptron.



(*b*) Three-input perceptron.

How does the perceptron learn its classification tasks?

That is, if the perceptron has a group of historical data about the inputs and the output classes, how the perceptron embeds the knowledge from historical data into the perceptron itself?

This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range [–0.5, 0.5], and then updated to obtain the output consistent with the training examples.

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

$$e(p) = Y_d(p) - Y(p) \qquad \text{where } p = 1, 2, 3, \ldots$$

Iteration p here refers to the *p*th training example presented to the perceptron.

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

The perceptron learning rule

$$w_i(p+1) = w_i(p) + \quad x_i(p) \cdot e(p)$$

where p = 1, 2, 3, . . . ; α is the **learning rate**, a **positive** constant less than unity.

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

Perceptron's training algorithm

Step 1: Initialization

Set initial weights w_1, w_2,…, w_n and threshold θ to random numbers in the range [–0.5, 0.5].

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

Perceptron's training algorithm (continued)

Step 2: Activation

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

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)\, w_i(p) - \theta\right]$$

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

Perceptron's training algorithm (continued)

Step 3: Weight training

Update the weights of the perceptron

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

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

The weight correction is computed by the **delta rule**:

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

Step 4: Iteration

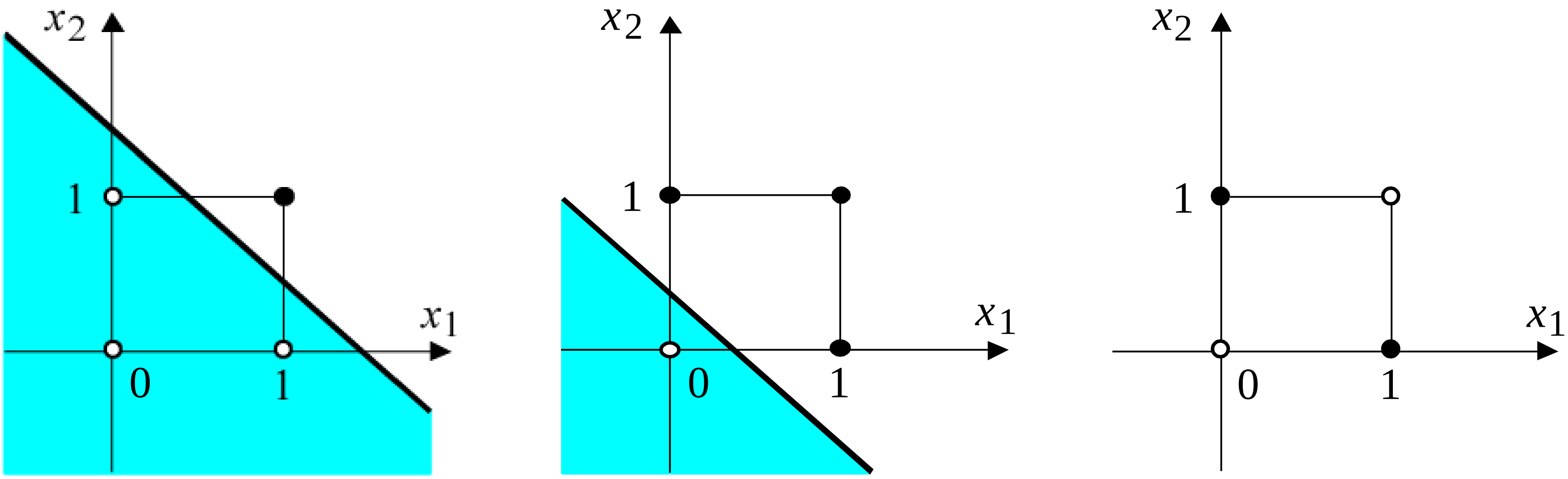
Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Example of perceptron learning: the logical operation AND

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

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

Two-dimensional plots of basic logical operations



(a) AND ($x_1 \cap x_2$) (b) OR ($x_1 \cup x_2$) (c) Exclusive-OR ($x_1 \oplus x_2$)

A perceptron can learn the operations AND and OR, **but not Exclusive-OR**.

Perceptron Review

Input output mapping

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)\, w_i(p) - \theta\right]$$

Learning rule

$$w_i(p+1) = w_i(p) + \alpha \times x_i(p) \times e(p), \qquad e(p) = Y_d(p) - Y(p)$$

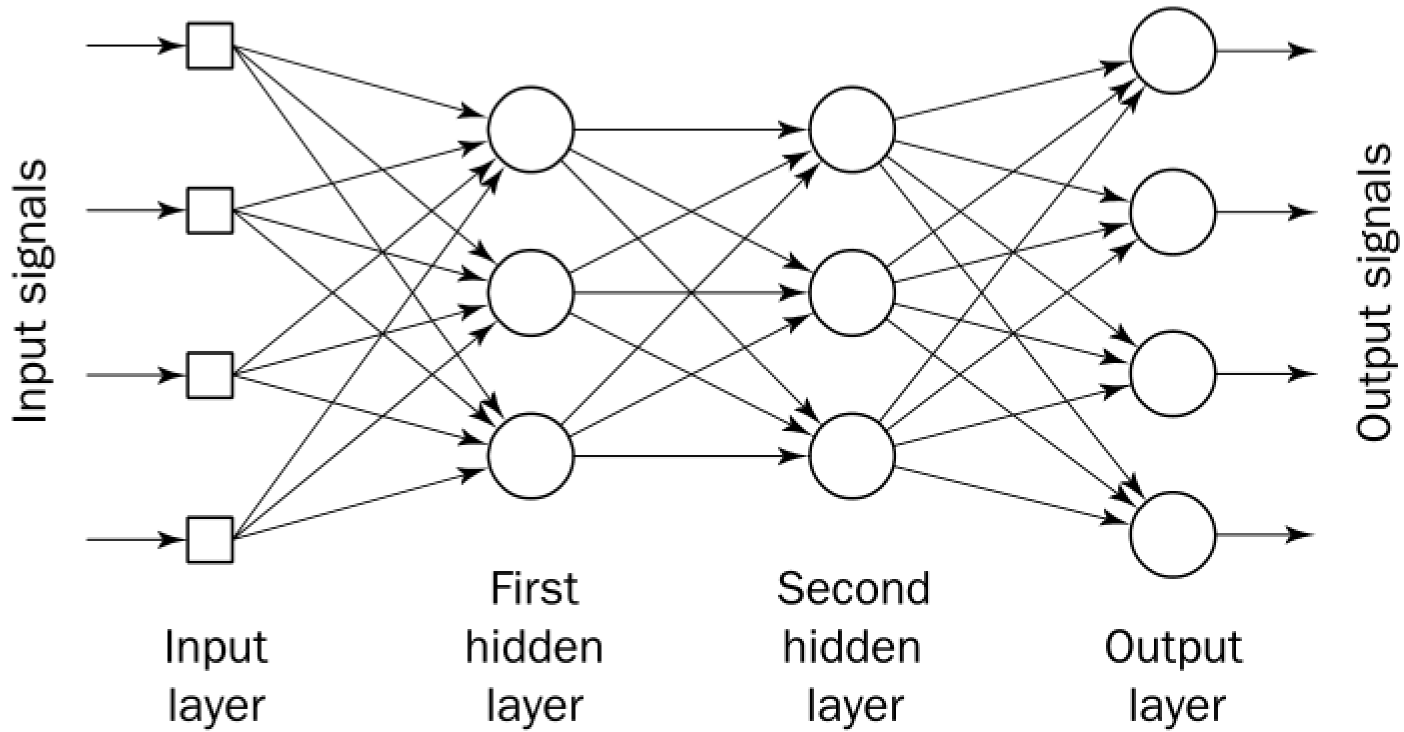
A perceptron can learn the operations *AND* and *OR*, **but not *Exclusive-OR*.** In other word, the perceptron can learn only linearly separable functions.

To cop with nonlinearly separable functions, we need multilayer neural networks

Multilayer neural networks

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

Multilayer perceptron with two hidden layers



What does the middle layer hide?

- A hidden layer "hides" its desired output. Neurons in the hidden layer cannot be observed through the input/output behavior of the network. There is no obvious way to know what the desired output of the hidden layer should be.

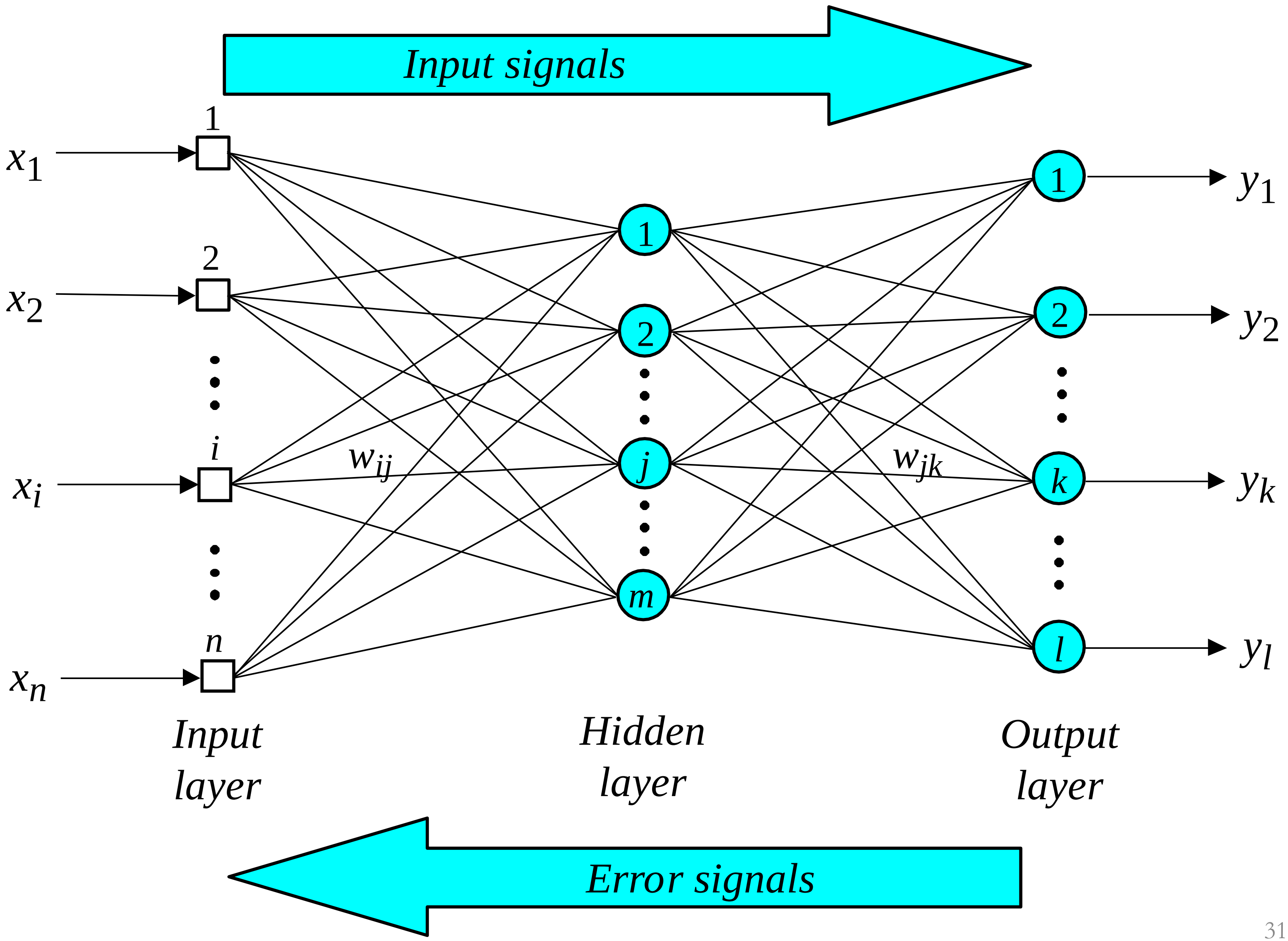
- Commercial ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Experimental neural networks may have five or even six layers, including three or four hidden layers, and utilize millions of neurons.

Back-propagation neural network

- Learning in a multilayer network proceeds the same way as for a perceptron.
- A training set of input patterns is presented to the network.
- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are **adjusted to reduce this error.**

- In a back-propagation neural network, the learning algorithm has two phases.
- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

Three-layer back-propagation neural network



The back-propagation training algorithm

<u>Step 1</u>: Initialization

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

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

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: Activation

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

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

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \cdot w_{ij}(p) - \theta_j\right]$$

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

Step 2 : Activation (continued)

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

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \cdot w_{jk}(p) - \theta_k\right]$$

where m is the number of inputs of neuron k in the output layer.

Step 3: Weight training

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

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

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

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

Calculate the weight corrections:

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

Update the weights at the output neurons:

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

<u>Step 3</u>: Weight training (continued)

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

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

Calculate the weight corrections:

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

Update the weights at the hidden neurons:

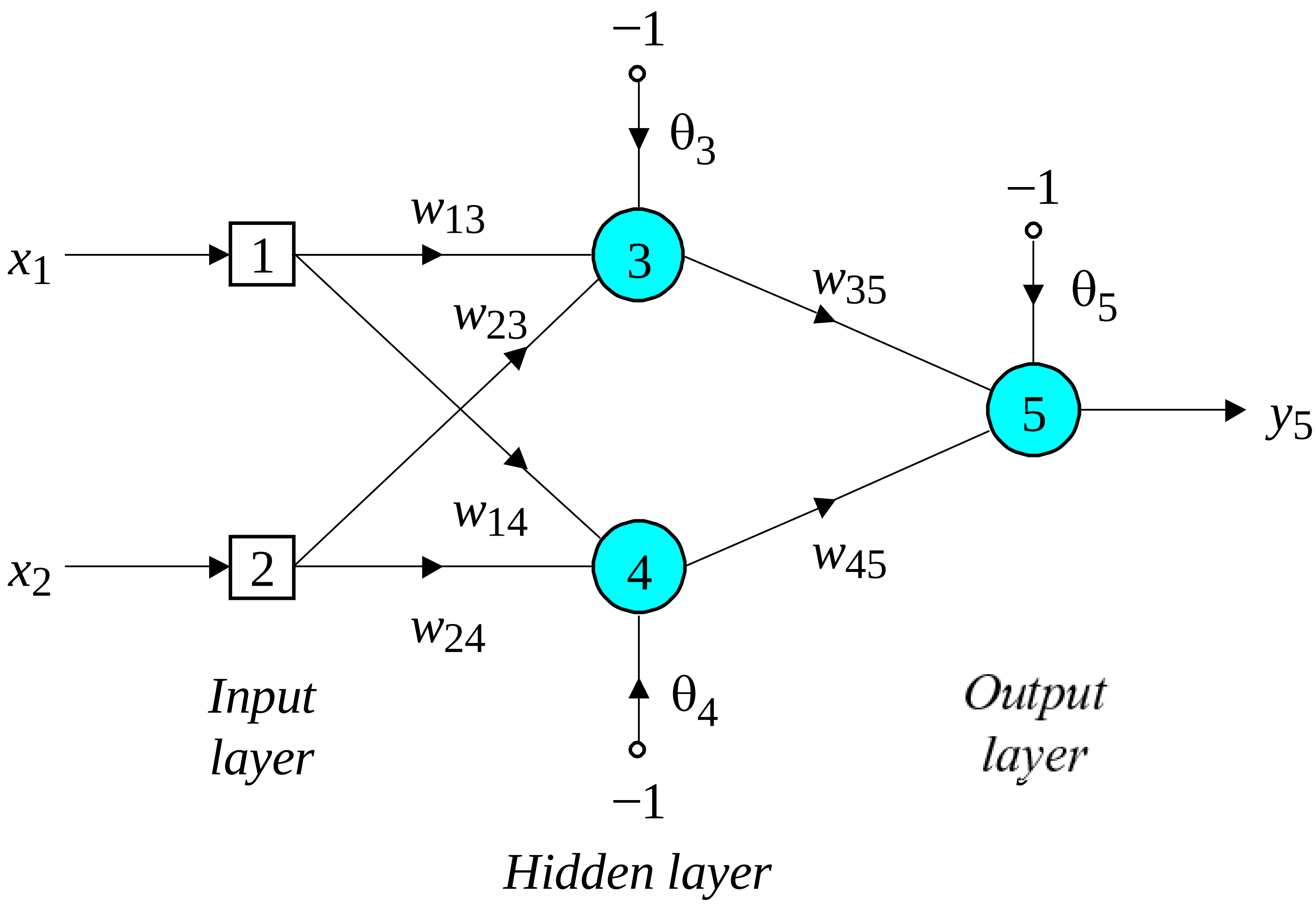
$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4: Iteration

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

As an example, we may consider the three-layer back-propagation network. Suppose that the network is required to perform logical operation *Exclusive*-*OR*. Recall that a single-layer perceptron could not do this operation. Now we will apply the three-layer net.

Three-layer network for solving the Exclusive-OR operation



- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.
- The initial weights and threshold levels are set randomly as follows:

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

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

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

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

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

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

- Thus, the following error is obtained:

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

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, *e*, from the output layer backward to the input layer.

- First, we calculate the error gradient for neuron 5 in the output layer:

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

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

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$
$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$

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

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

$$\delta_4 = y_4(1-y_4)\cdot\delta_5\cdot w_{45} = 0.8808\cdot(1-0.8808)\cdot(-0.127\,4)\cdot 1.1 = -0.0147$$

- We then determine the weight corrections:

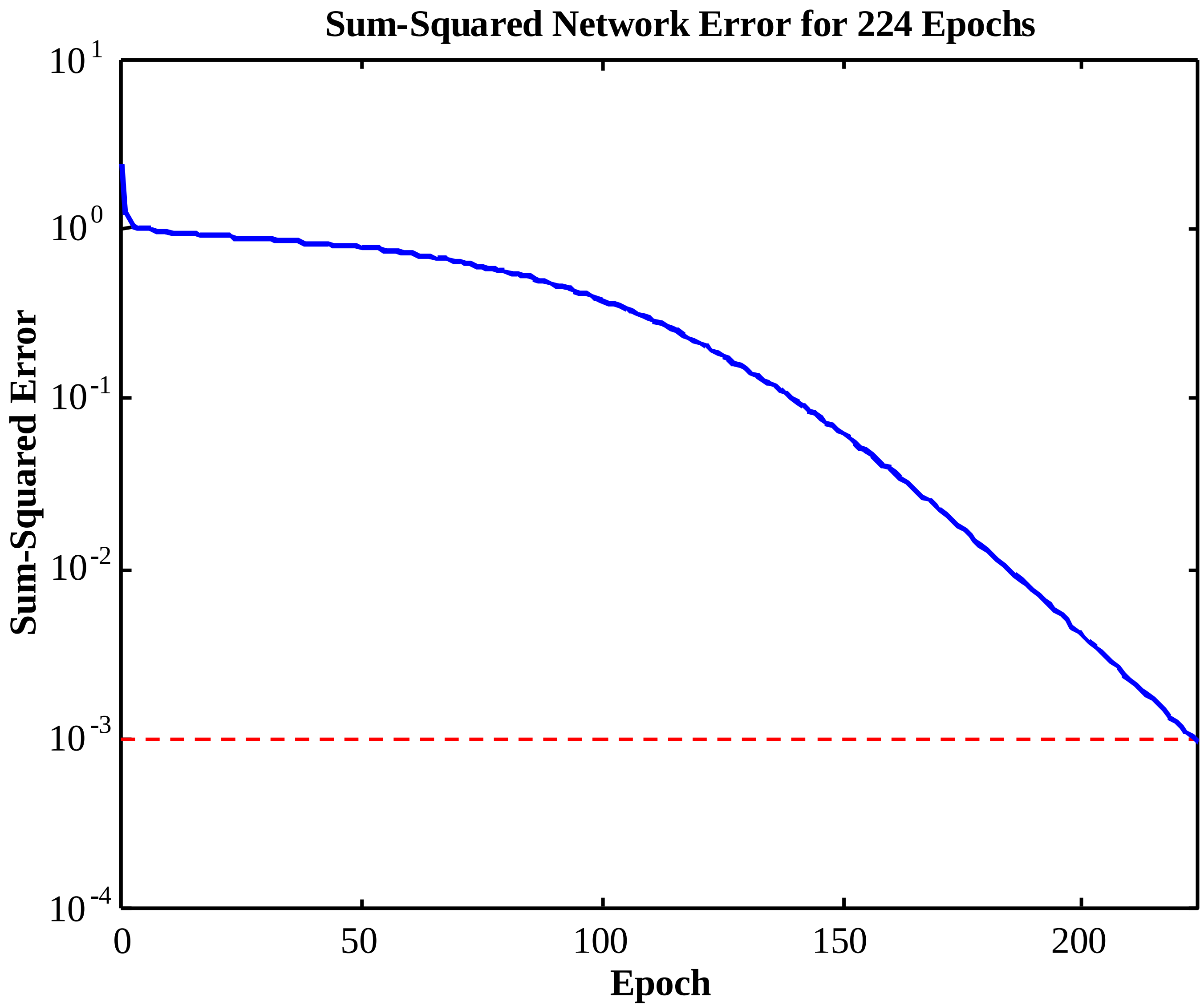
$$\Delta w_{13} = \alpha\cdot x_1\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta w_{23} = \alpha\cdot x_2\cdot\delta_3 = 0.1\cdot 1\cdot 0.0381 = 0.0038$$
$$\Delta\theta_3 = \alpha\cdot(-1)\cdot\delta_3 = 0.1\cdot(-1)\cdot 0.0381 = -0.0038$$
$$\Delta w_{14} = \alpha\cdot x_1\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta w_{24} = \alpha\cdot x_2\cdot\delta_4 = 0.1\cdot 1\cdot(-0.0147) = -0.0015$$
$$\Delta\theta_4 = \alpha\cdot(-1)\cdot\delta_4 = 0.1\cdot(-1)\cdot(-0.0147) = 0.0015$$

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$
$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$
$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$
$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$
$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$
$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$
$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$
$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$
$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$

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

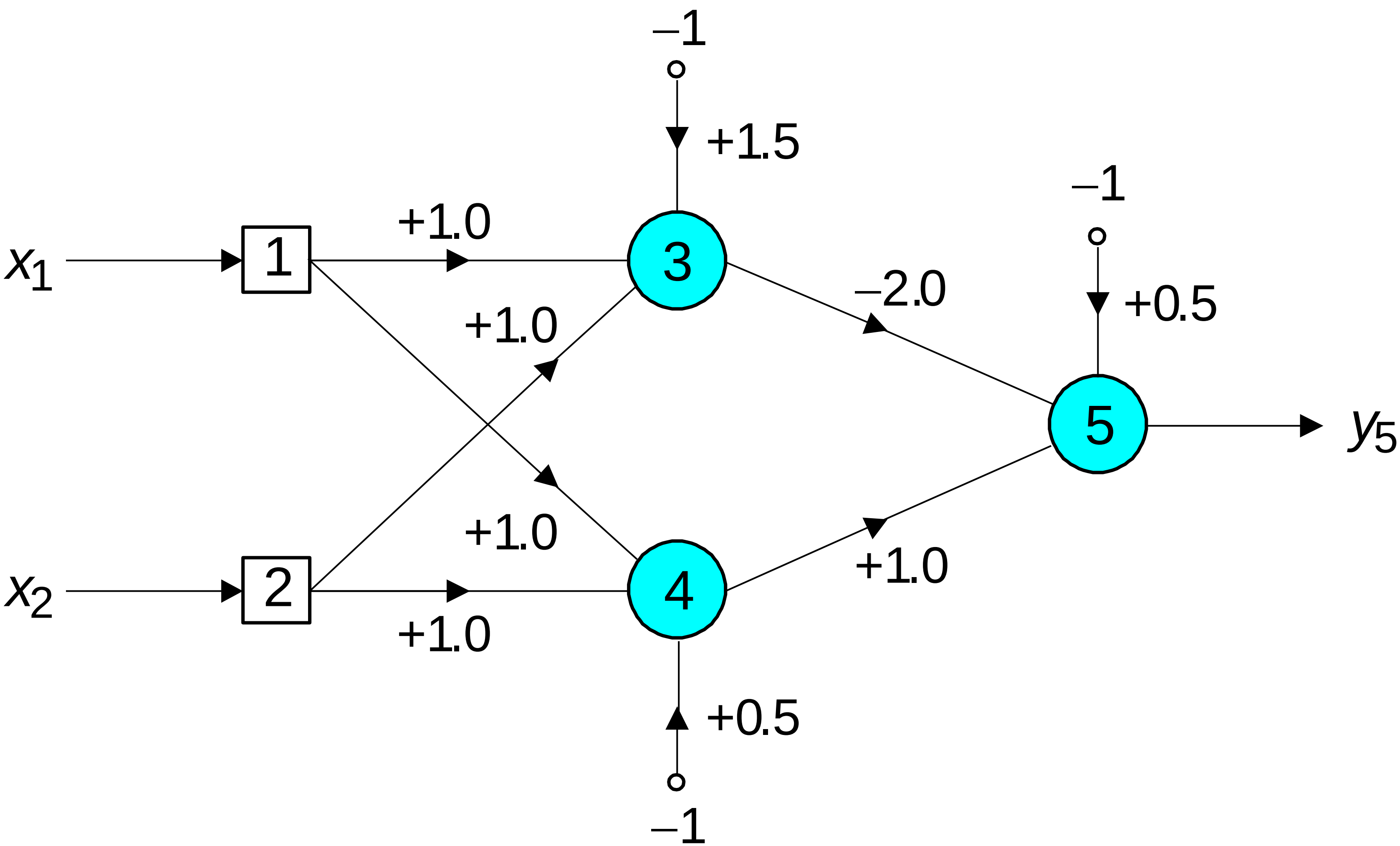
Learning curve for operation *Exclusive-OR*



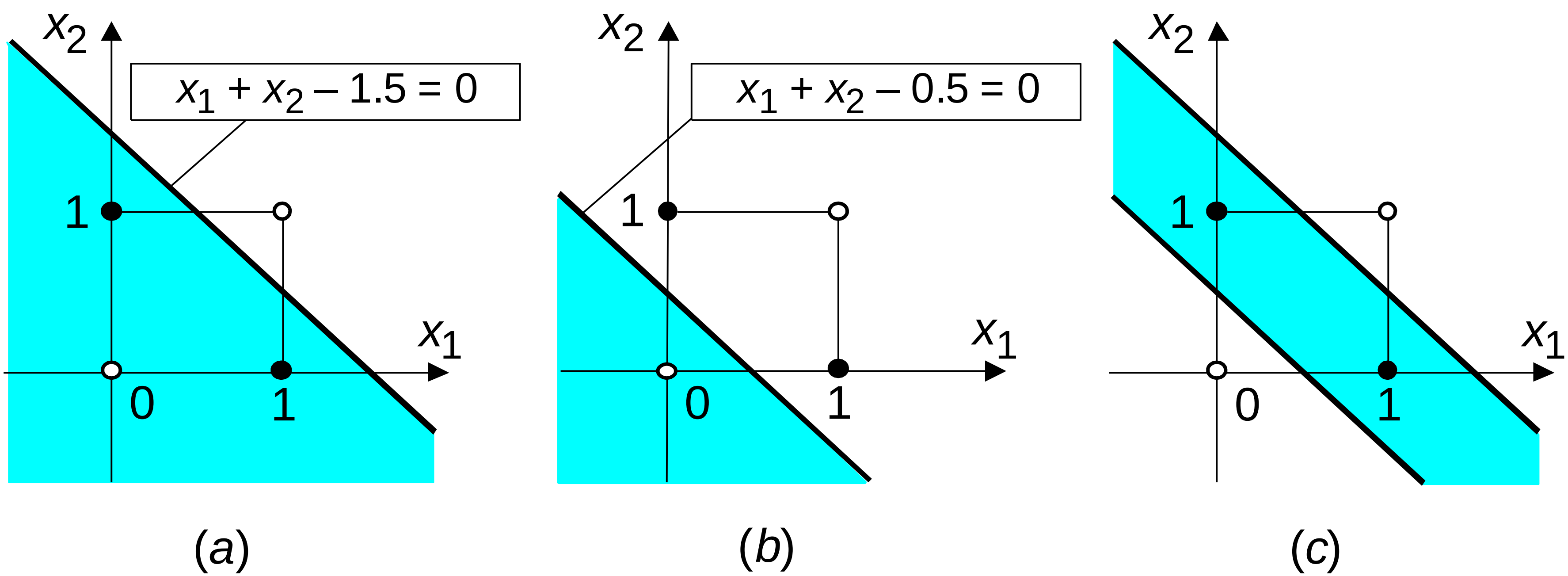
Final results of three-layer network learning

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

Network represented by McCulloch-Pitts model for solving the *Exclusive-OR* operation



Decision boundaries



(a) Decision boundary constructed by hidden neuron 3;

(*b*) Decision boundary constructed by hidden neuron 4;

(*c*) Decision boundaries constructed by the complete three-layer network

Accelerated learning in multilayer neural networks

- A multilayer network learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent**:

$$Y^{tanh} = \frac{2a}{1 + e^{-bX}} - a$$

where a and b are constants.

Suitable values for a and b are:

$a = 1.716$ and $b = 0.667$

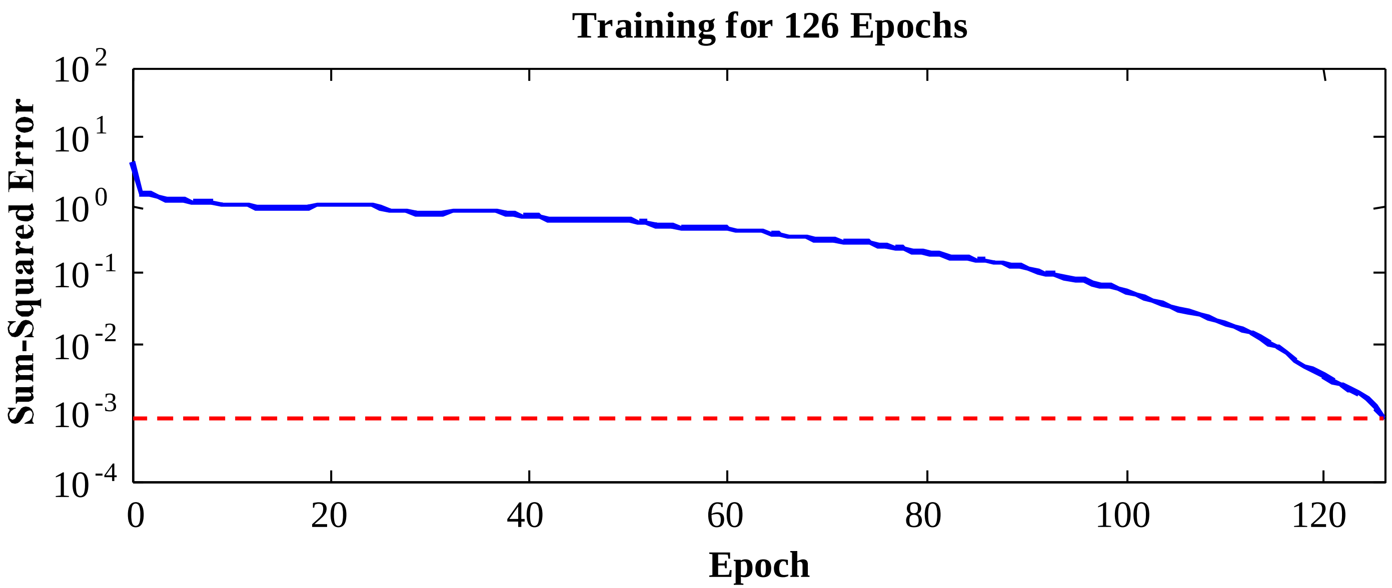
- We also can accelerate training by including a **momentum term** in the delta rule:

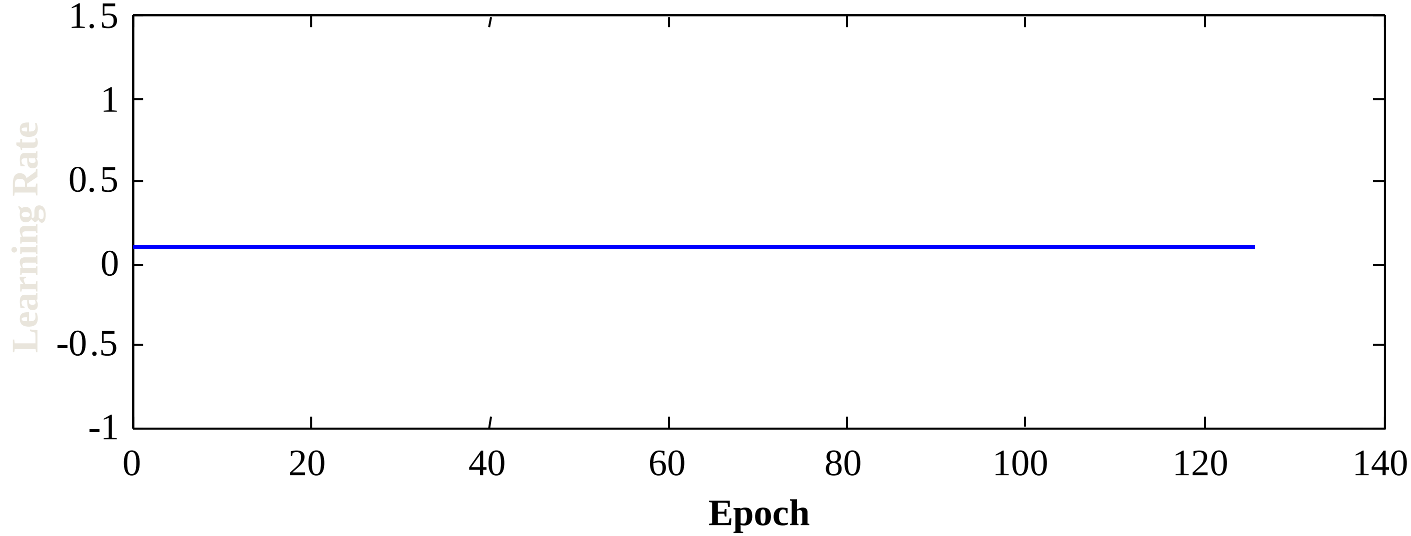
$$\Delta w_{jk}(p) = \beta \cdot \Delta w_{jk}(p-1) + \alpha \cdot y_j(p) \cdot \delta_k(p)$$

where β is a positive number ($0 \leq \beta < 1$) called the **momentum constant**. Typically, the momentum constant is set to 0.95.

This equation is called the **generalized delta rule**.

Learning with momentum for operation *Exclusive-OR*





Learning with adaptive learning rate

To accelerate the convergence and yet avoid the danger of instability, we can apply two heuristics:

Heuristic 1

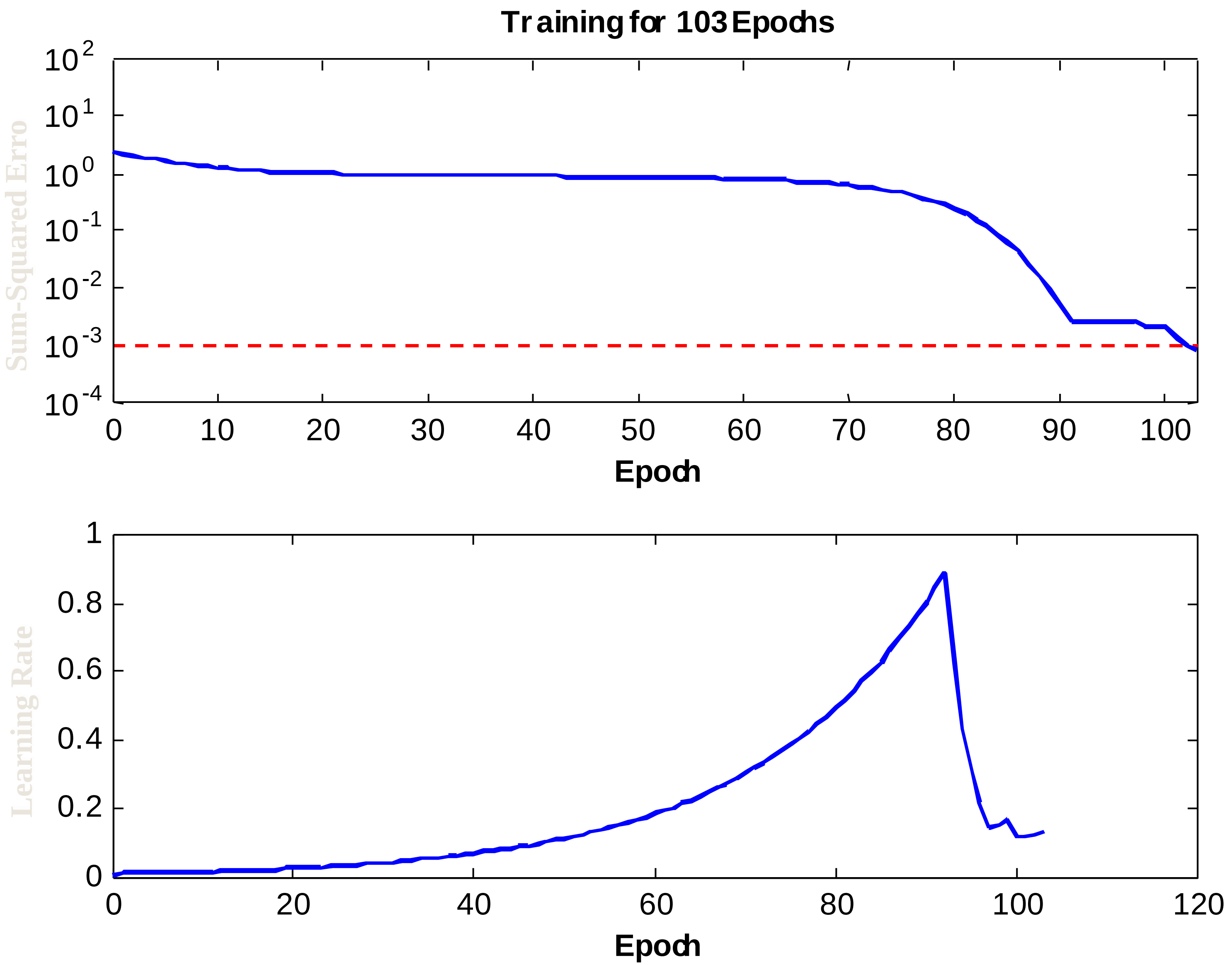
If the change of the sum of squared errors has the same algebraic sign for several consequent epochs, then the learning rate parameter, α, should be increased.

Heuristic 2

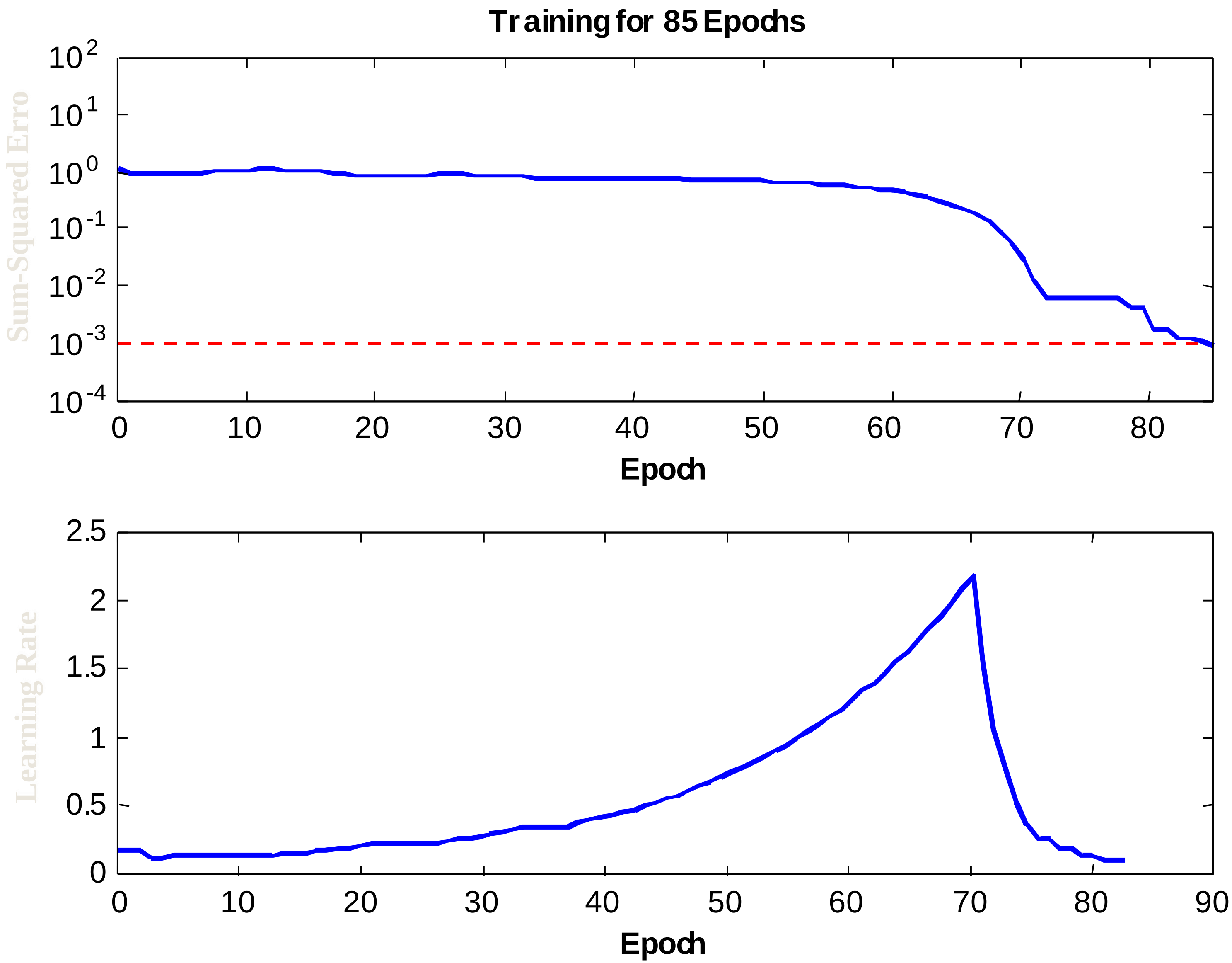
If the algebraic sign of the change of the sum of squared errors alternates for several consequent epochs, then the learning rate parameter, α, should be decreased.

- Adapting the learning rate requires some changes in the back-propagation algorithm.
- If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated.
- If the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Learning with adaptive learning rate



Learning with momentum and adaptive learning rate



Want to know more?

- Free e-book about neural networks
- http://neuralnetworksanddeeplearning.com/index.html