Neural Networks

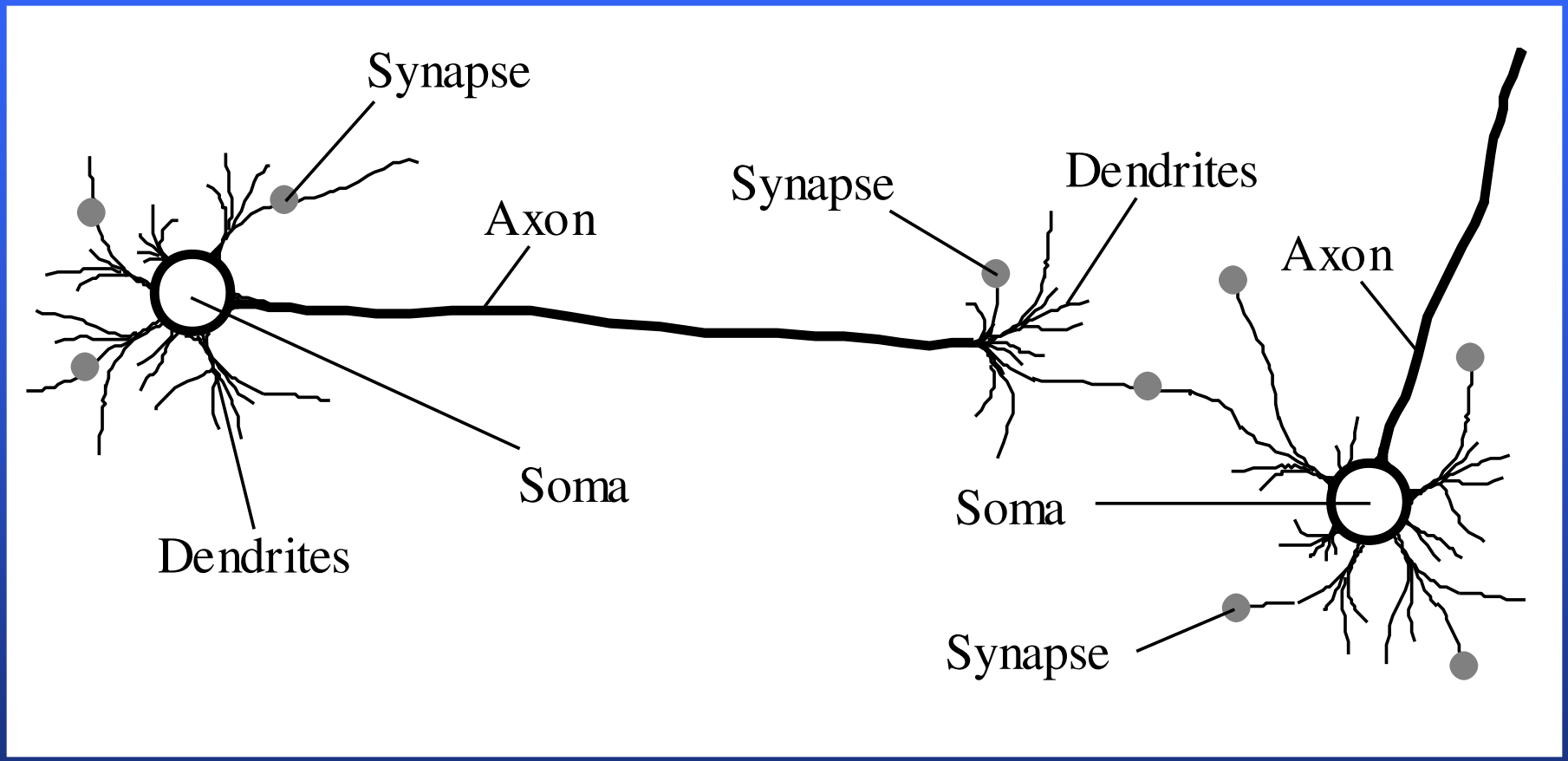
Neural Networks and the Brain

- A **neural network** is a model of reasoning inspired by 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.

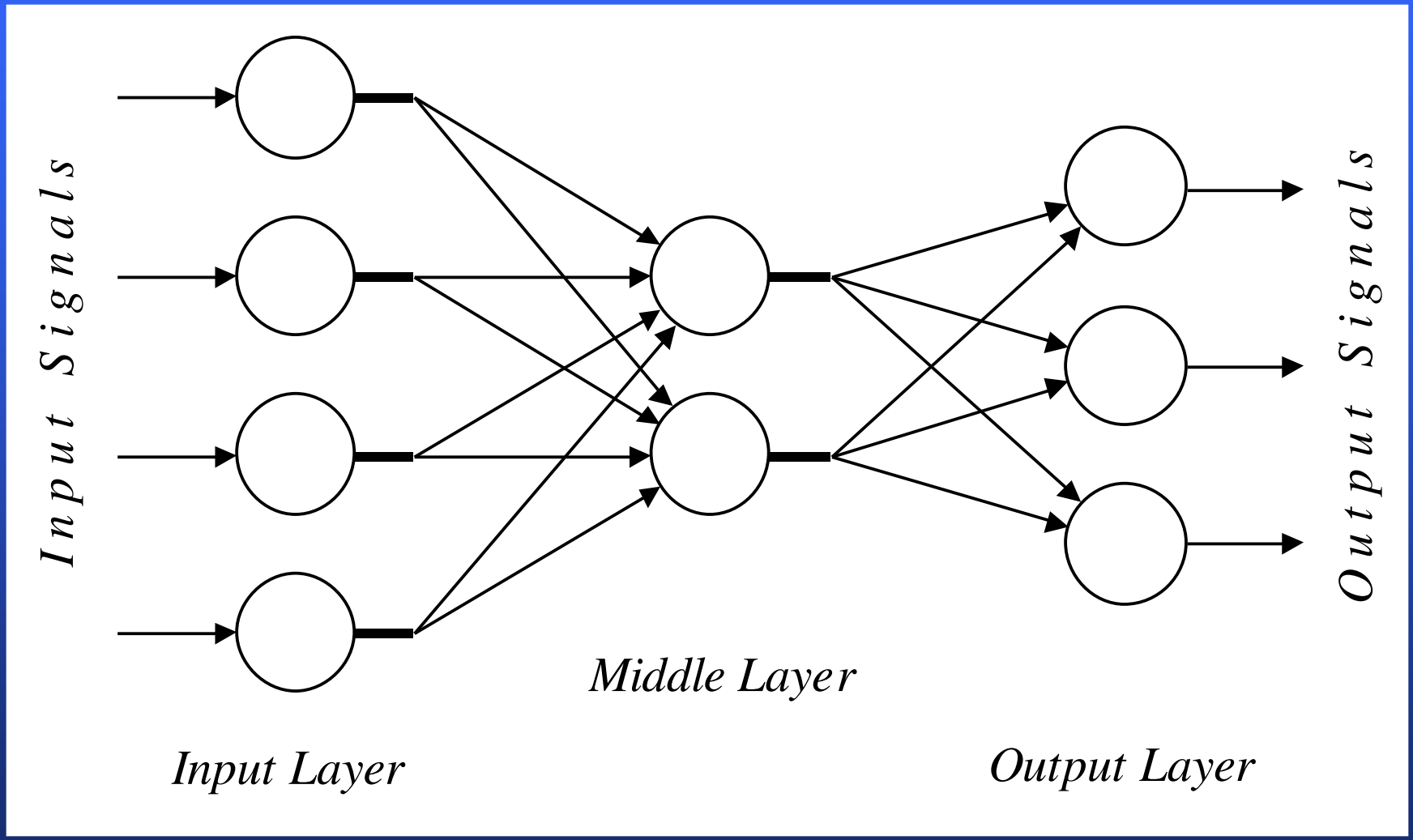
Neural Networks and the Brain

- 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 tremendous processing power.
- A neuron consists of a cell body, soma, a number of fibers called dendrites, and a single long fiber called the axon.

Biological neural network



Architecture of a typical artificial neural network

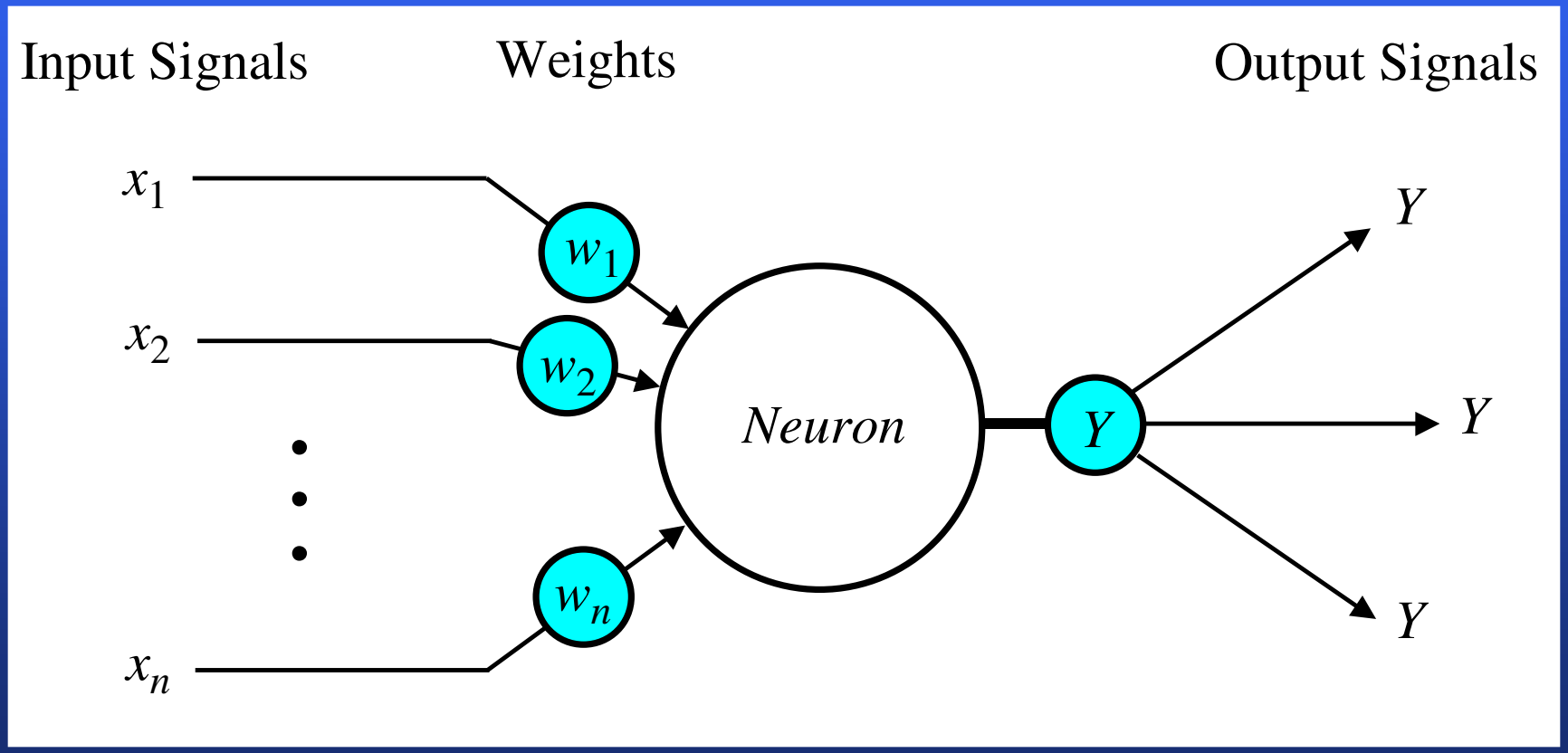


Analogy between biological and artificial neural networks

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



A Simple Activation Function – Sign Function

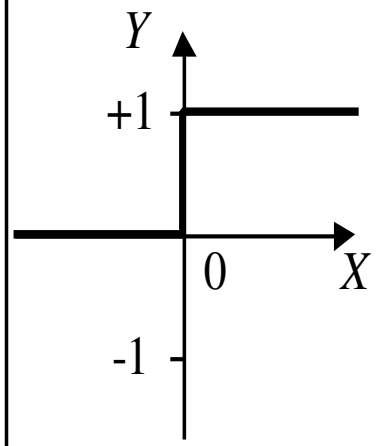
- The neuron computes the weighted sum of the input signals and compares the result with a **threshold value**, θ.
- If the net input is less than the threshold, the neuron output is –1.
- if the net input is greater than or equal to the threshold, the neuron becomes activated and its output is +1.
- 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**. (McCulloch and Pitts 1943)

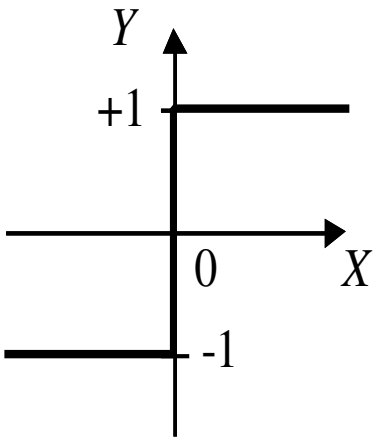
4 Common Activation functions of a neuron

Step function



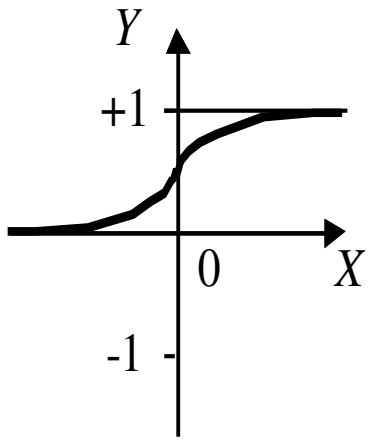
$$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{if } X < 0 \end{cases}$$

Sign function



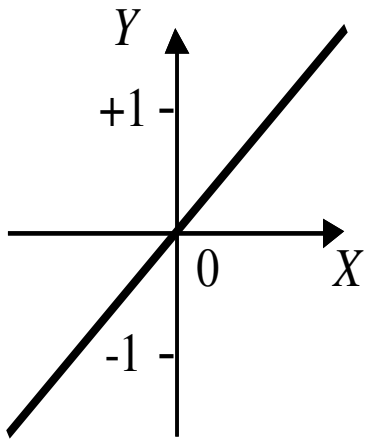
$$Y^{sign} = \begin{cases} +1, \text{if } X \geq 0 \\ -1, \text{if } X < 0 \end{cases}$$

Sigmoid function



$$Y^{sigmoid} = \frac{1}{1+e^{-X}}$$

Linear function



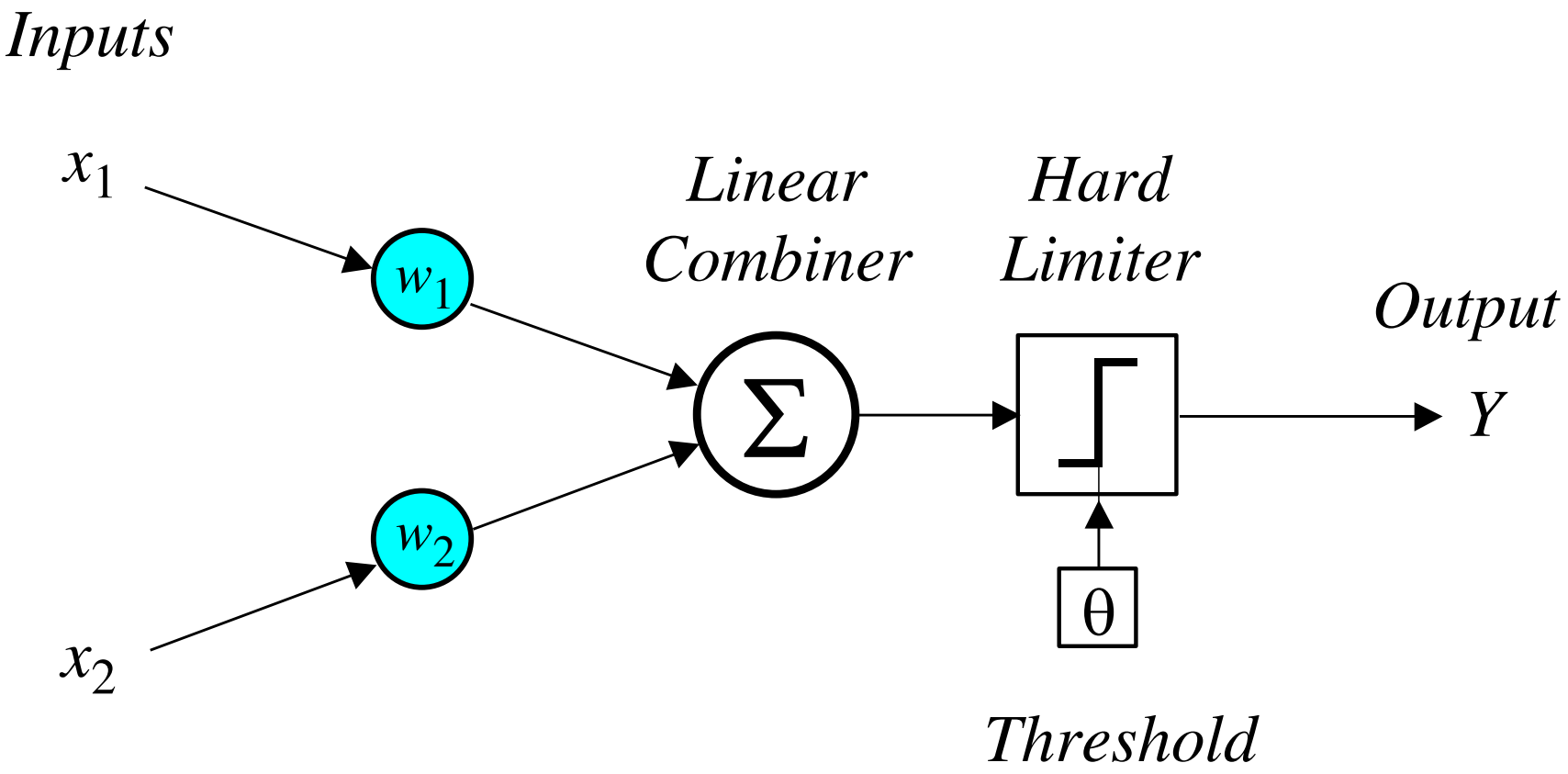
$$Y^{linear} = X$$

Most Common?

Can a single neuron learn a task?

- Start off with earliest/ simplest
  - 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

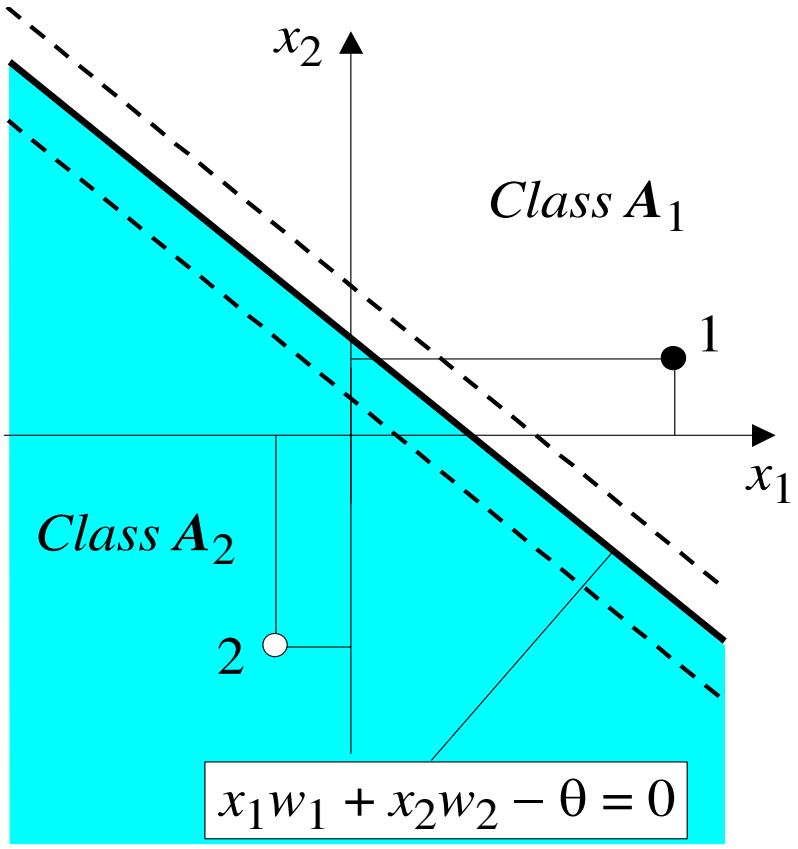
- The operation of Rosenblatt's perceptron is based on the **McCulloch and Pitts neuron model**. The model consists of a linear combiner followed by a hard limiter.
- The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to +1 if its input is positive and –1 if 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 A_1 and 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**:

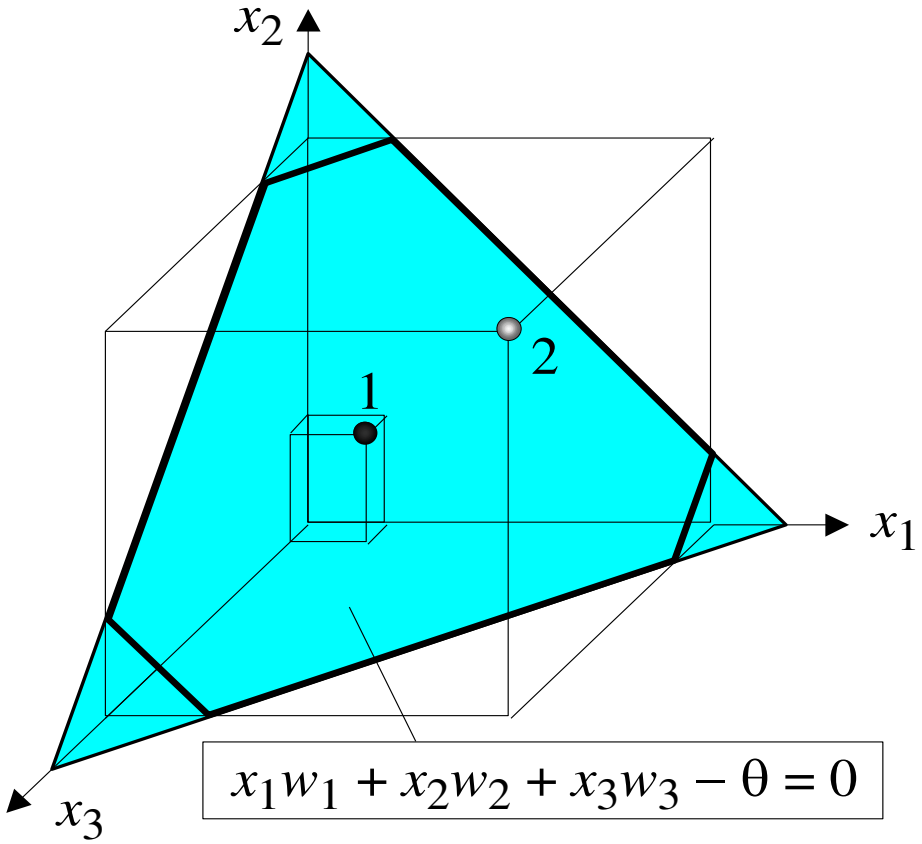
$$\sum_{i=1}^{n} x_i w_i - \theta = 0$$

- See next slide

Linear separability in the perceptrons



(*a*) Two-input perceptron.



(*b*) Three-input perceptron.

Changing θ shifts the boundary

How does the perceptron learn its classification tasks?

- making small adjustments in the weights
  - to reduce the difference between the actual and desired outputs of the perceptron.
  - Learns weights such that output is consistent with the training examples.
- The initial weights are randomly assigned,
  - usually in the range [–0.5, 0.5],

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

$$e(p) = Y_d(p) - Y(p)$$

where $p = 1, 2, 3, \ldots$

Iteration p here refers to the pth 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

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

where p is iteration # = 1, 2, 3, . . .

- α is the **learning rate**, a positive constant less than unity (1).
- Intuition:
  - Weight at next iteration is based on an adjustment from the current weight
  - Adjustment amount is influenced by the amount of the error, the size of the input, and the learning rate
  - Learning rate is a free parameter that must be "***tuned***"
- 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: Initialisation

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

(during training, 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 for weight i 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

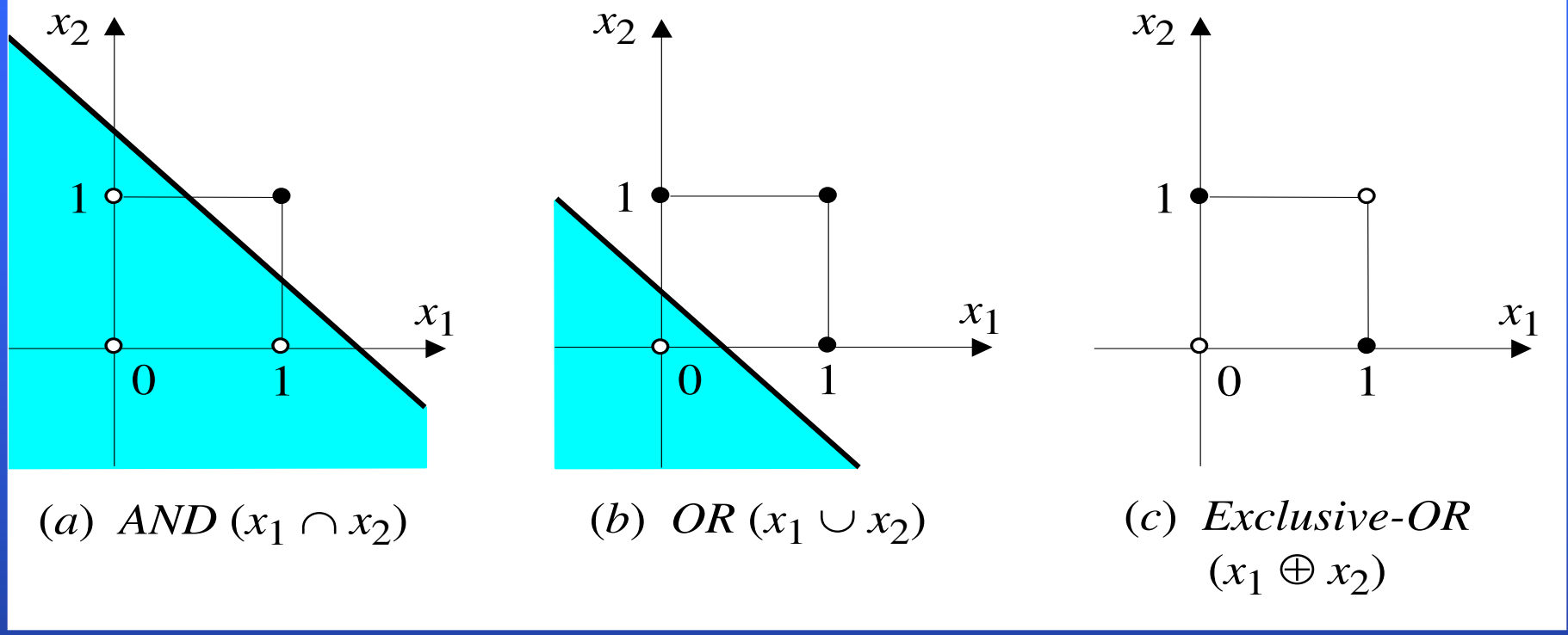
Increase iteration p by one, go back to *Step 2* and repeat the process until convergence.

Example of perceptron learning: the logical operation *AND*

Epoch	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	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

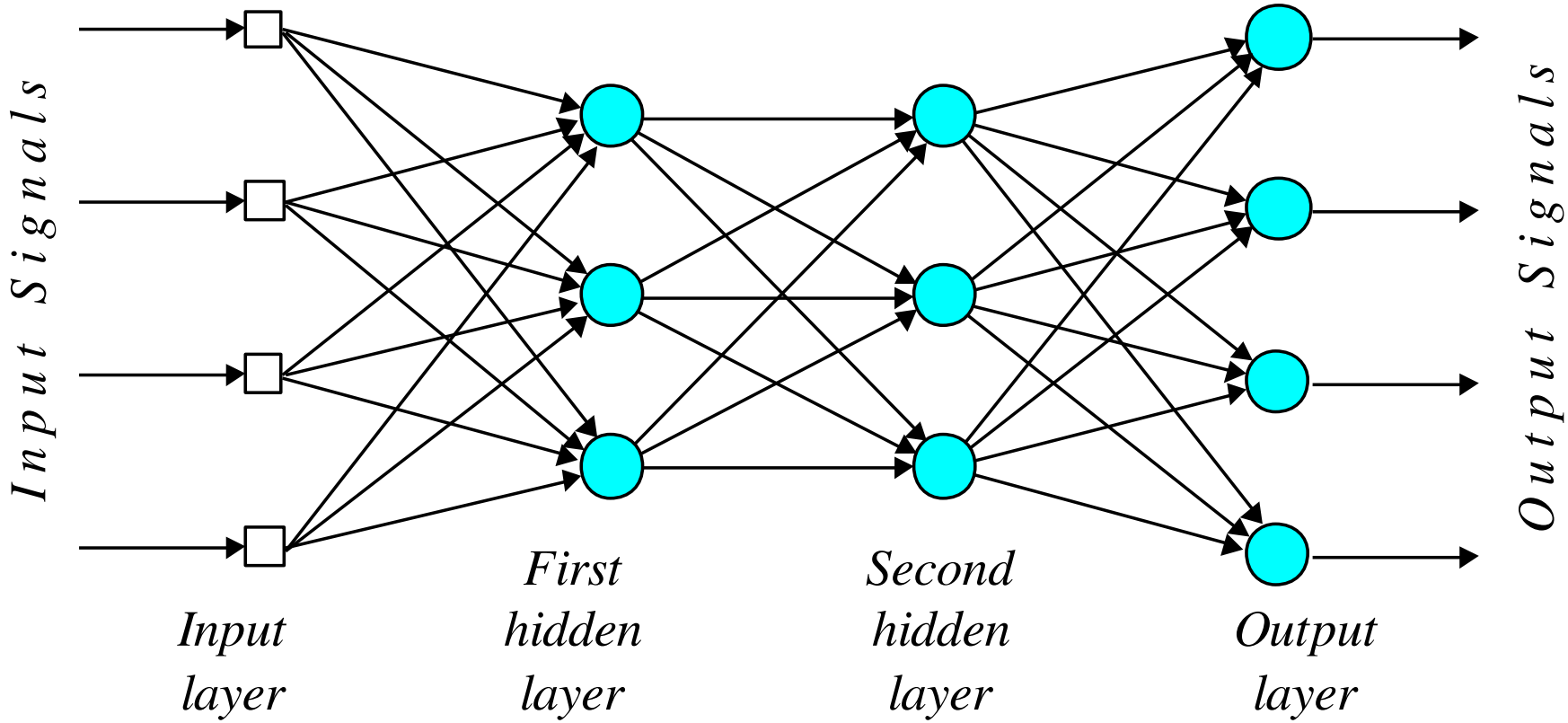


- A perceptron can learn the operations *AND* and *OR*, but not *Exclusive-OR*.
- *Exclusive-OR* is NOT linearly separable
- This limitation stalled neural network research for more than a decade

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



Hidden Layer

- Detects features in the inputs – hidden patterns
- With one hidden layer, can represent any continuous function of the inputs
- With two hidden layers even discontinuous functions can be represented

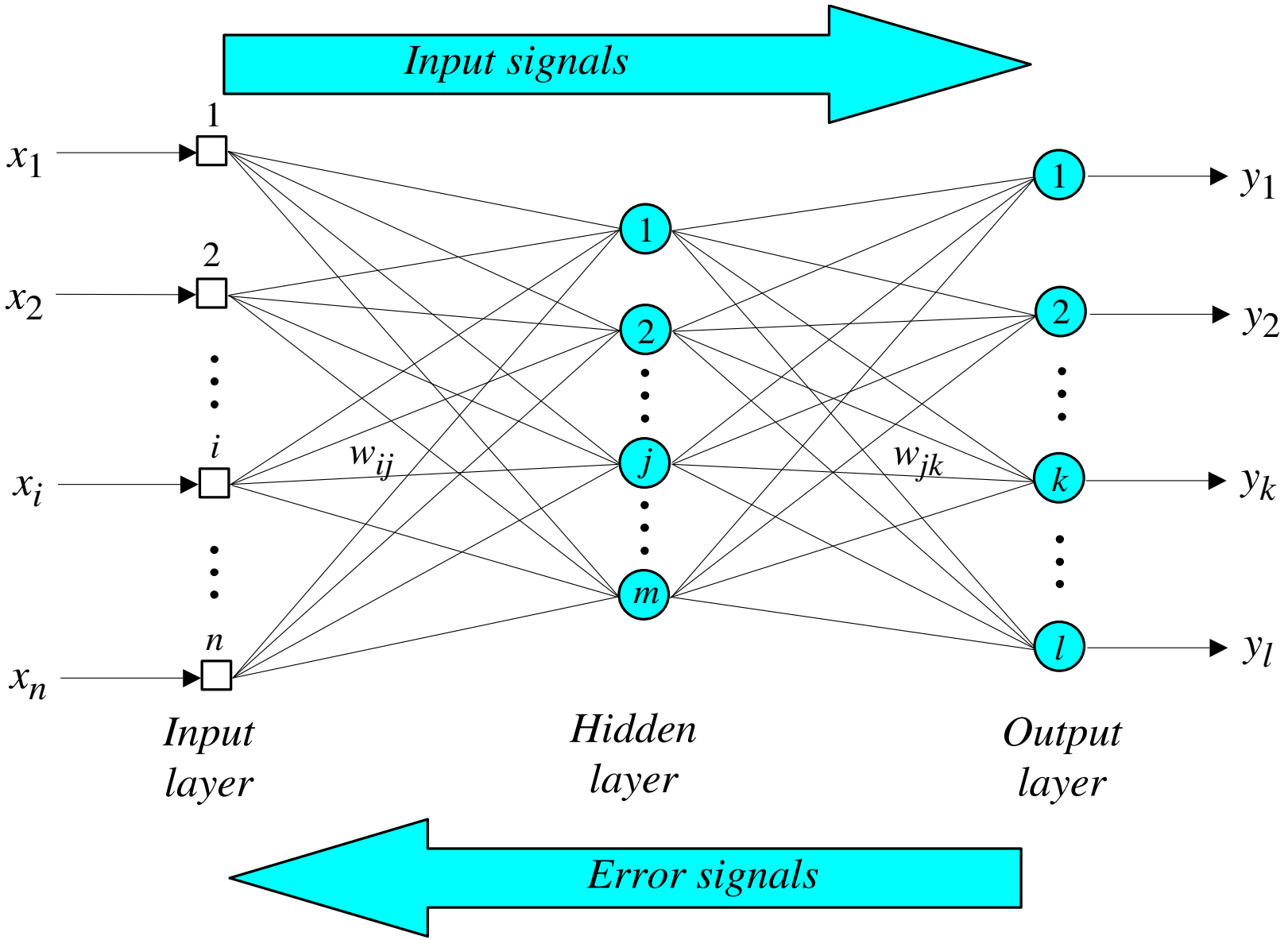
Back-propagation neural network

- Most popular of 100+ ANN learning algorithms
- 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.
- The difference is in the number of weights and architecture …

Back-propagation neural network

- In a back-propagation neural network, the learning algorithm has two phases:
  - 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.
    - Activation function generally sigmoid
  - 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.
- See next slide for picture …

Three-layer back-propagation neural network



The back-propagation training algorithm

Step 1: Initialisation

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 $e_k(p) = y_{d,k}(p) - y_k(p)$ (error at output unit k)

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)$$ (weight change for j to k link)

Update the weights at the output neurons:

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

Step 3: 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:

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

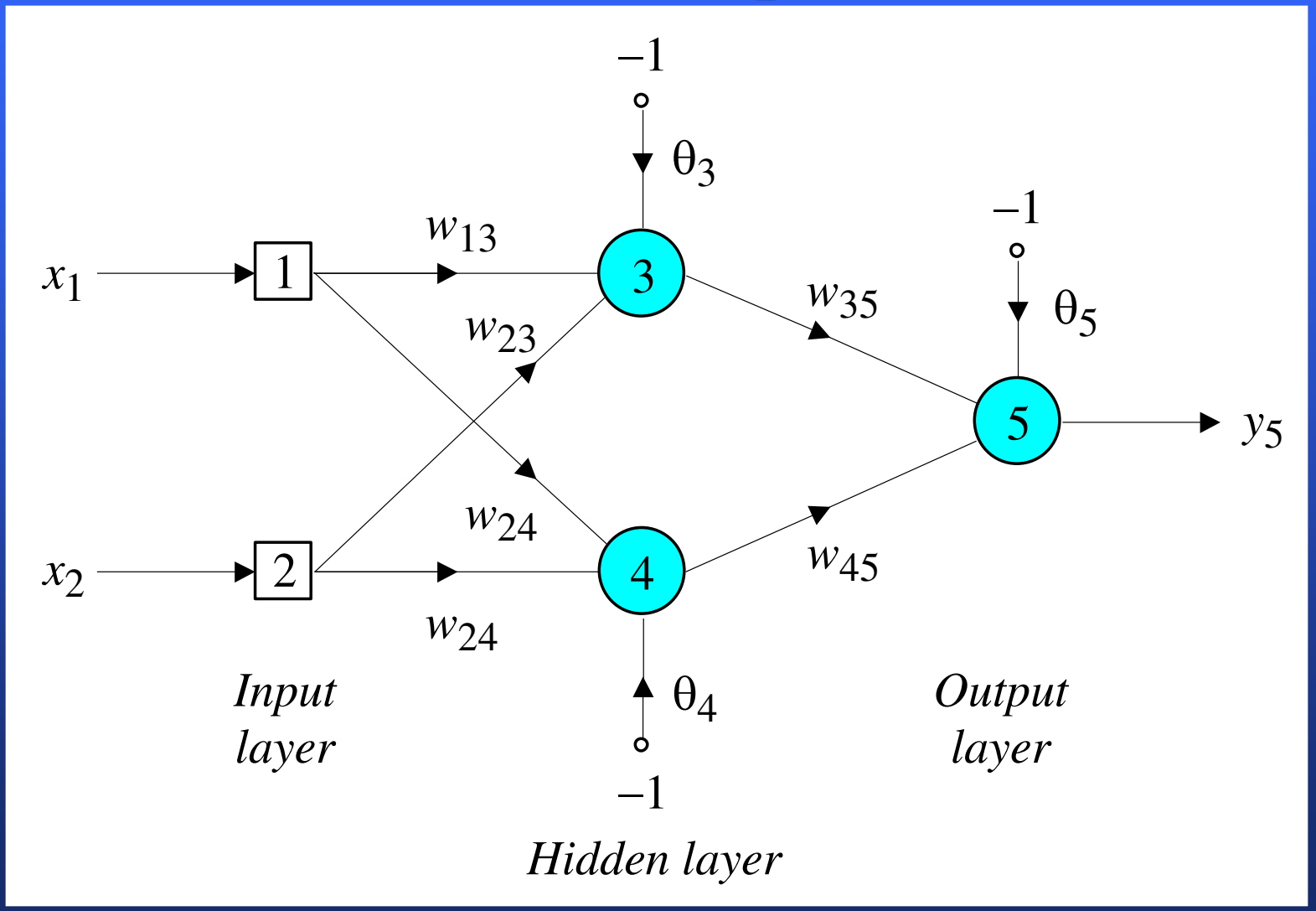
Step 4: Iteration

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

Example

- 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 back-propagation network
- See BackPropLearningXor.xls

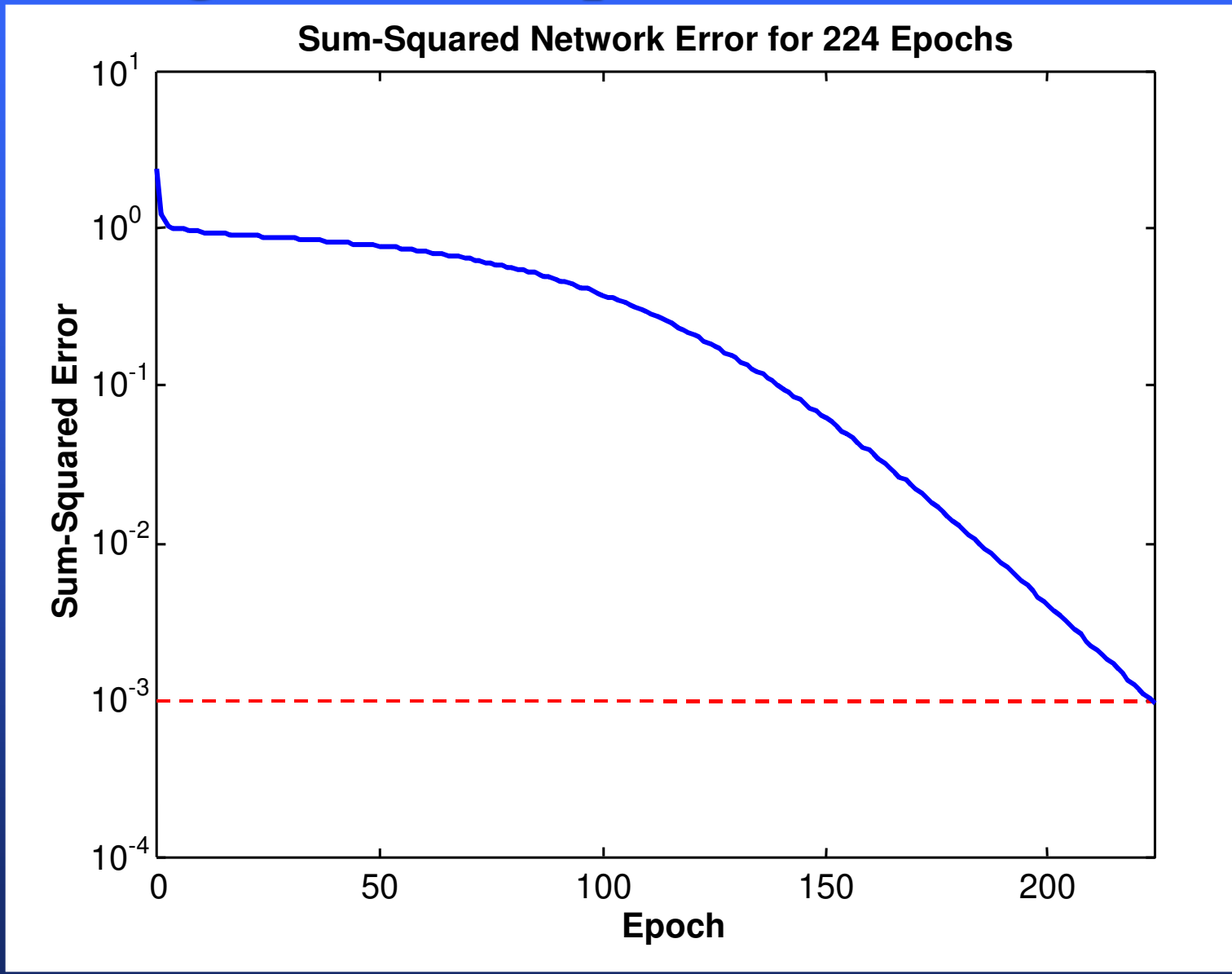
Three-layer network for solving the Exclusive-OR operation



Example (con)

- 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 $\theta_5 = 0.3$.

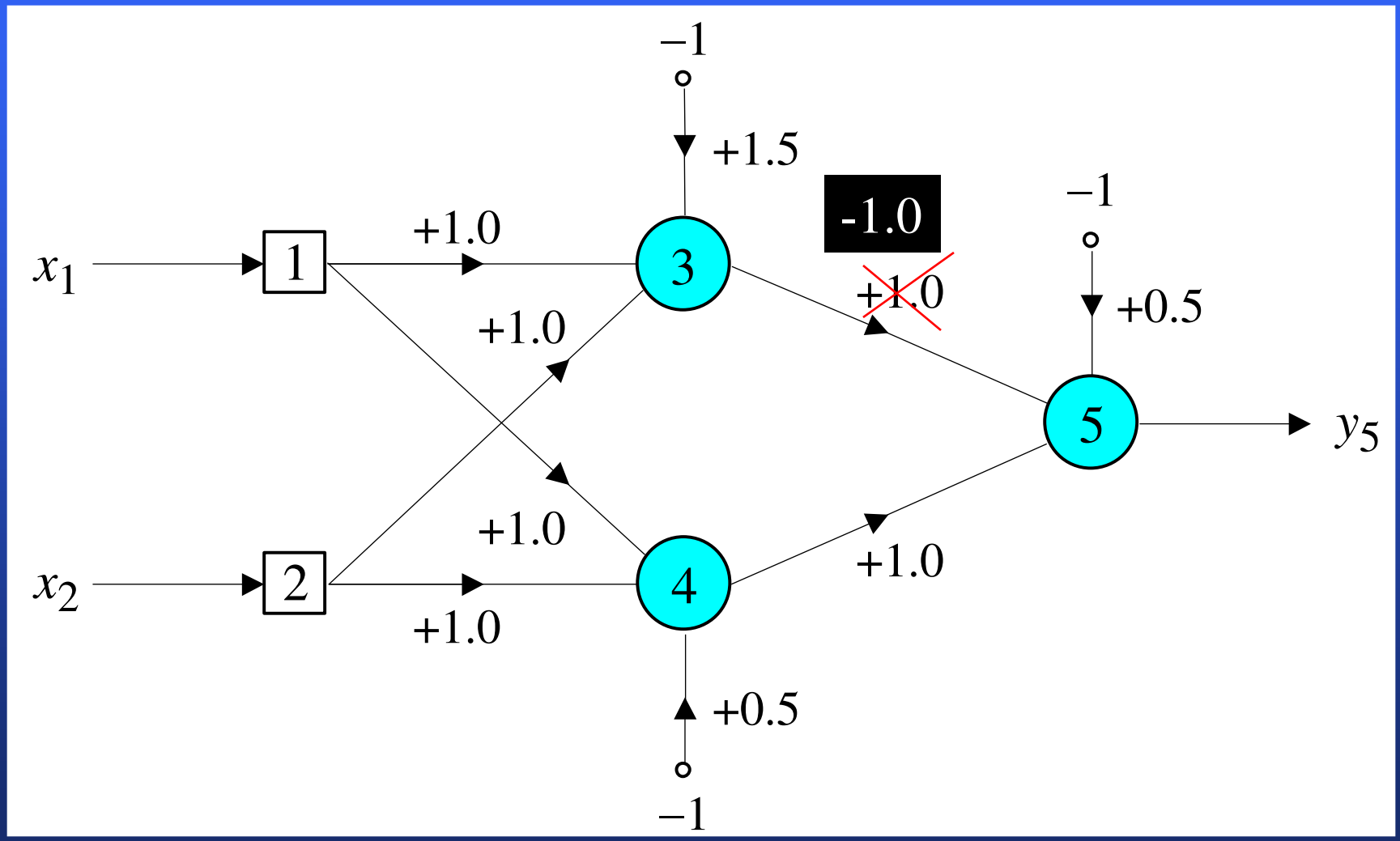
Learning curve for operation *Exclusive-OR*



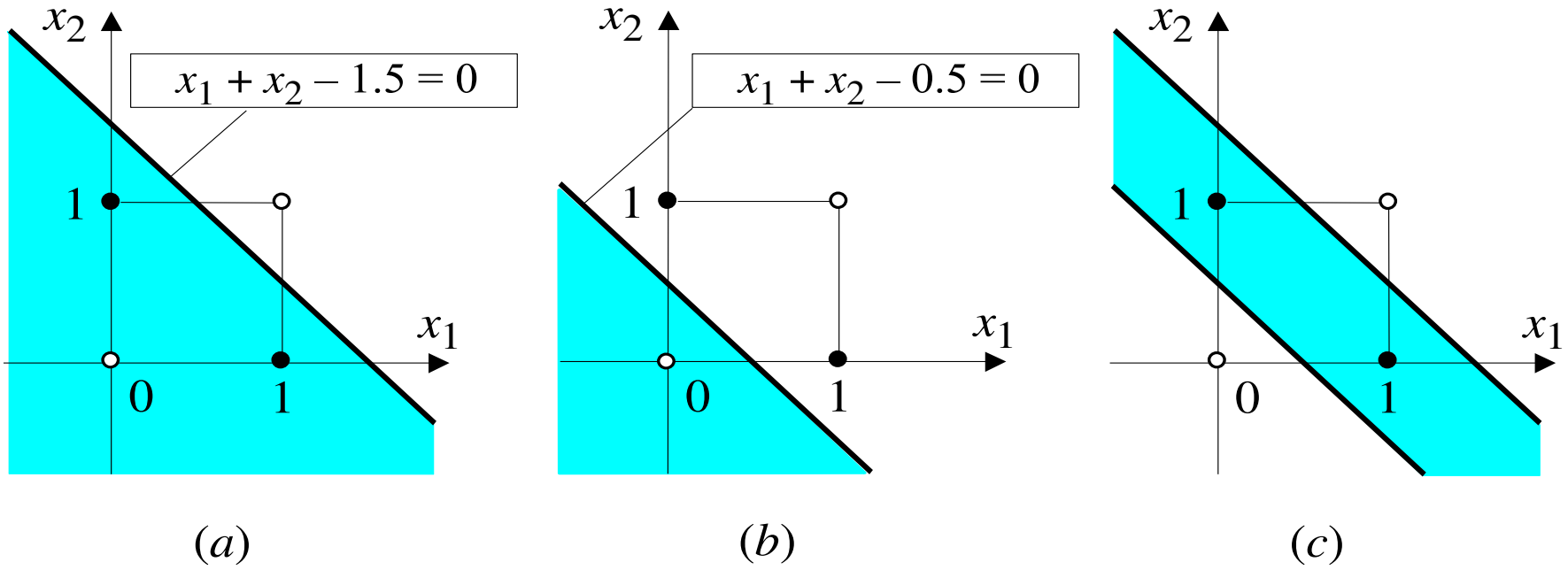
Final results of three-layer network learning

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

Neural Nets in Weka

- Xor – with default hidden layer
- Xor – with two hidden nodes
- Basketball Class
- Broadway Stratified – default
- Broadway Stratified – 10 hidden nodes

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 $b = 0.667$

Accelerated learning in multilayer neural networks

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

Basic version

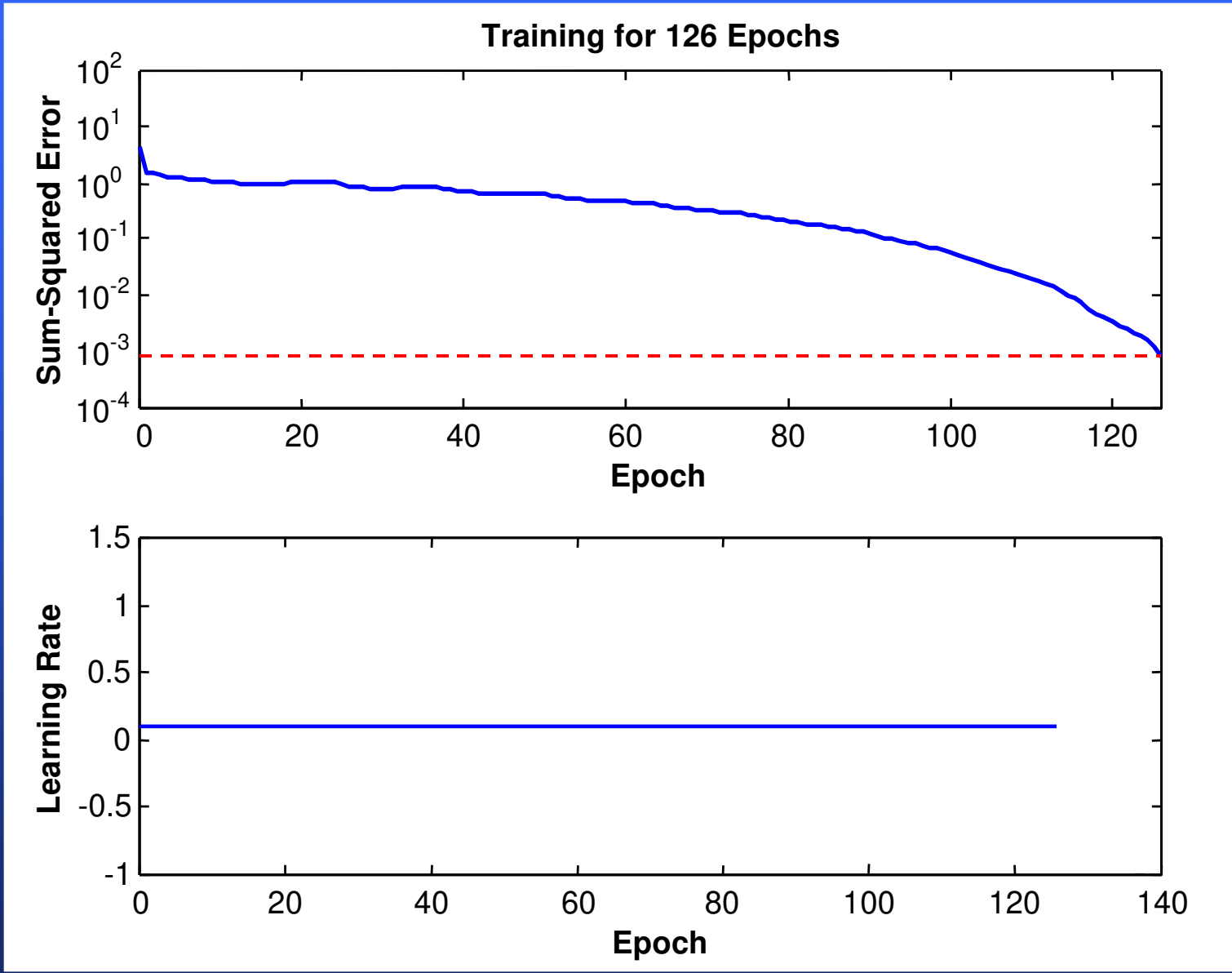
$$\Delta w_{jk}(p) = \beta \cdot \Delta w_{jk}(p-1) + \alpha \cdot y_j(p) \cdot \delta_k(p)$$

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

- This iteration's change in weight is influenced by last iteration's change in weight !!!

- This equation is called the **generalised 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.

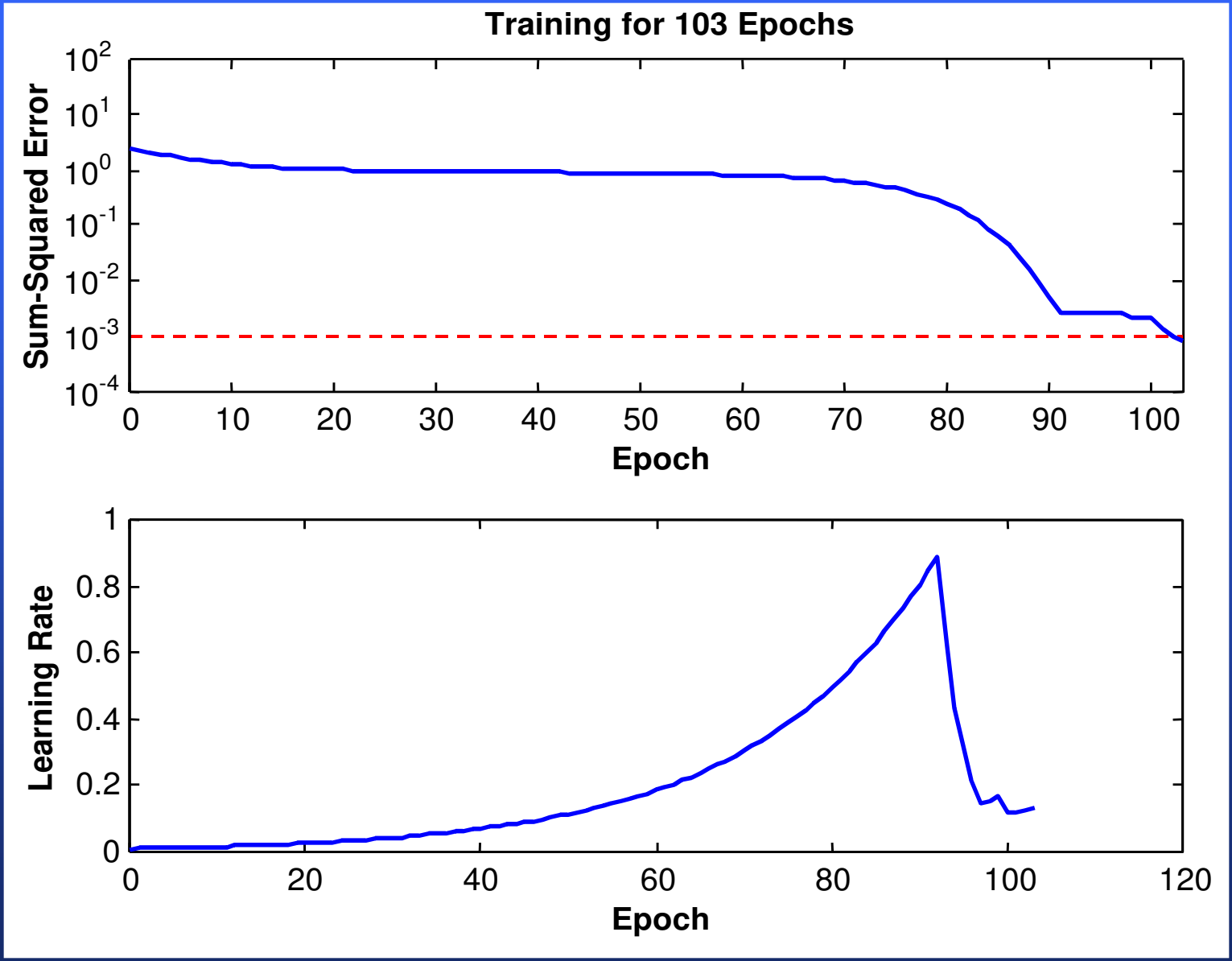
Heuristic 2

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.

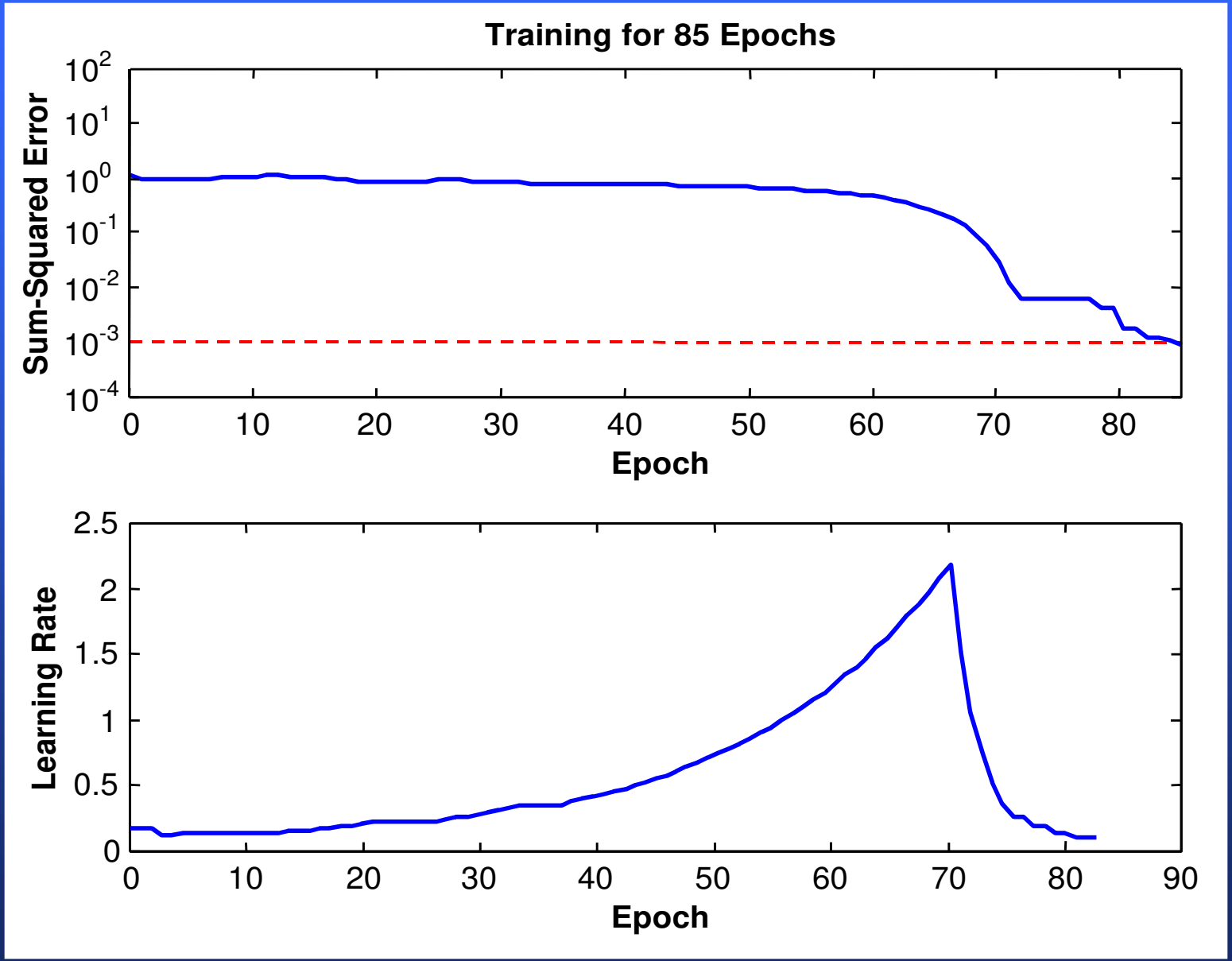
Learning with adaptive learning rate (con)

- 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



End Neural Networks