Artificial Intelligence

7. Neural Networks

Florin Leon

"Gheorghe Asachi" Technical University of Iași
Faculty of Automatic Control and Computer Engineering

"Al. I. Cuza" University of Iași
Faculty of Computer Science

Neural Networks

1. Introduction
2. The Perceptron. Adaline
3. The Multilayer Perceptron
4. Deep Networks
5. Conclusions

Neural Networks

1. Introduction
2. The Perceptron. Adaline
3. The Multilayer Perceptron
4. Deep Networks
5. Conclusions

Classification

- A training set is given with a set of instances (also called vectors, records, objects)
- Instances have attributes
- Each instance has attributes with certain values
- Usually the last attribute is the class

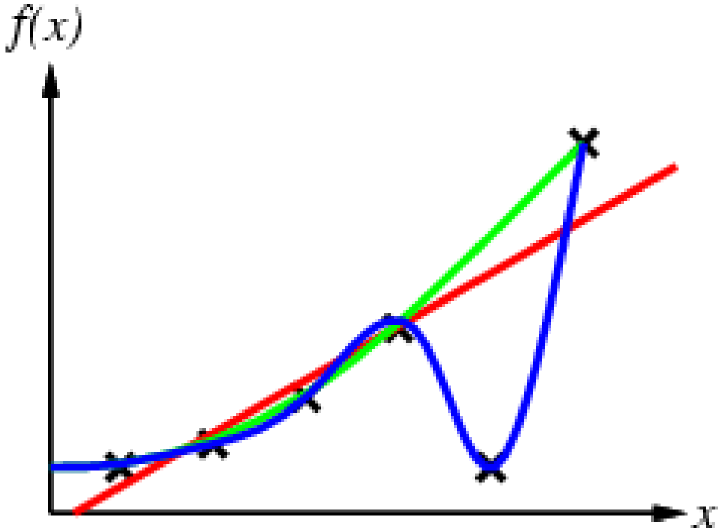
Attributes

Instances

Tid	Refund	Marital Status	Taxable Income	Cheat
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Regression

- It represents the approximation of a function
  - Any kind of function, not just the type $f : \mathbb{R}^n \rightarrow \mathbb{R}$
  - Only the output is continuous; some inputs may be discrete or non-numeric



Classification and regression

- The same basic idea: learning a relationship between inputs (vector **x**) and output (*y*) from data
- The only difference between classification and regression is the type of output: discrete and continuous, respectively
- Classification estimates a discrete output, the class
- Regression estimates a function *h* such that $h(\mathbf{x}) \approx y(\mathbf{x})$ with a certain precision
- For each training instance, the desired output value is given ⇒ supervised learning

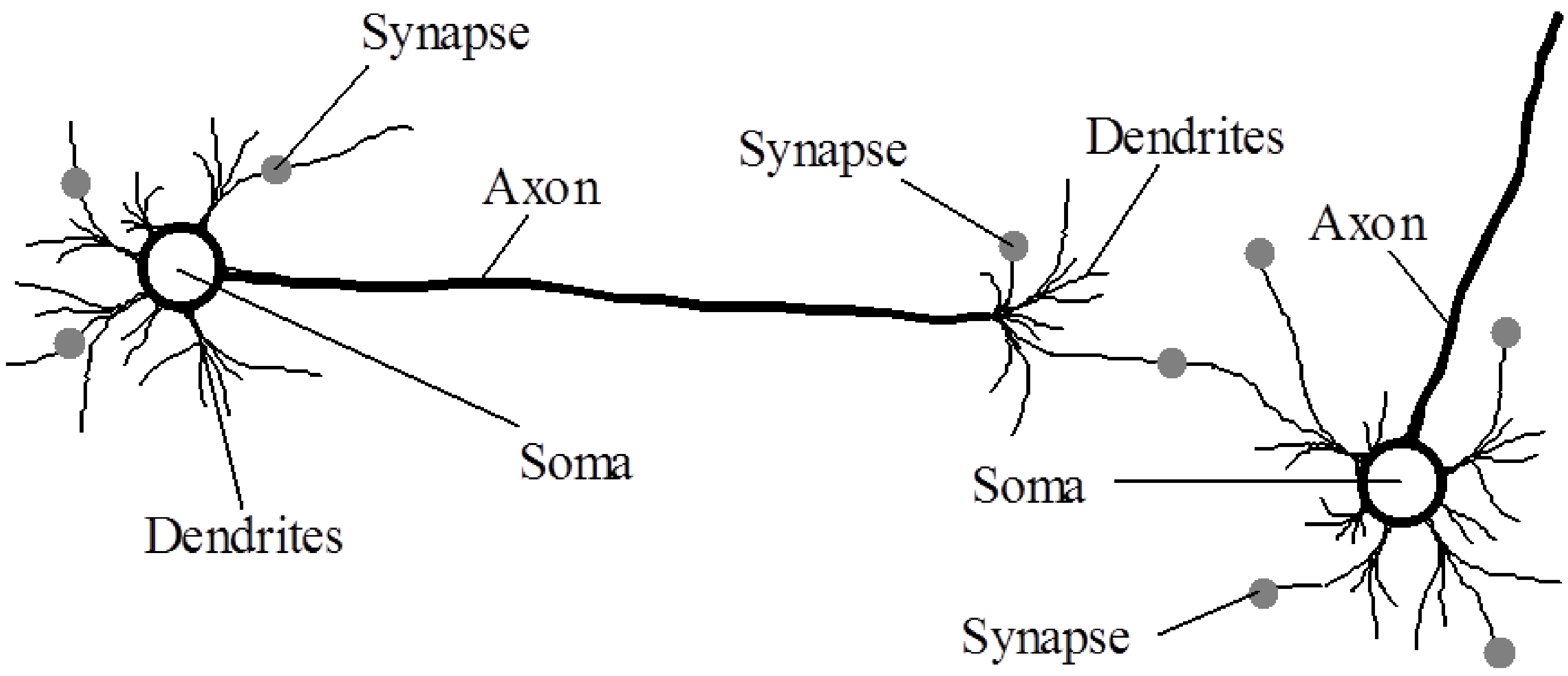
Neural networks

- The perceptron type neural networks, presented in this lecture, are generally used for regression and classification problems

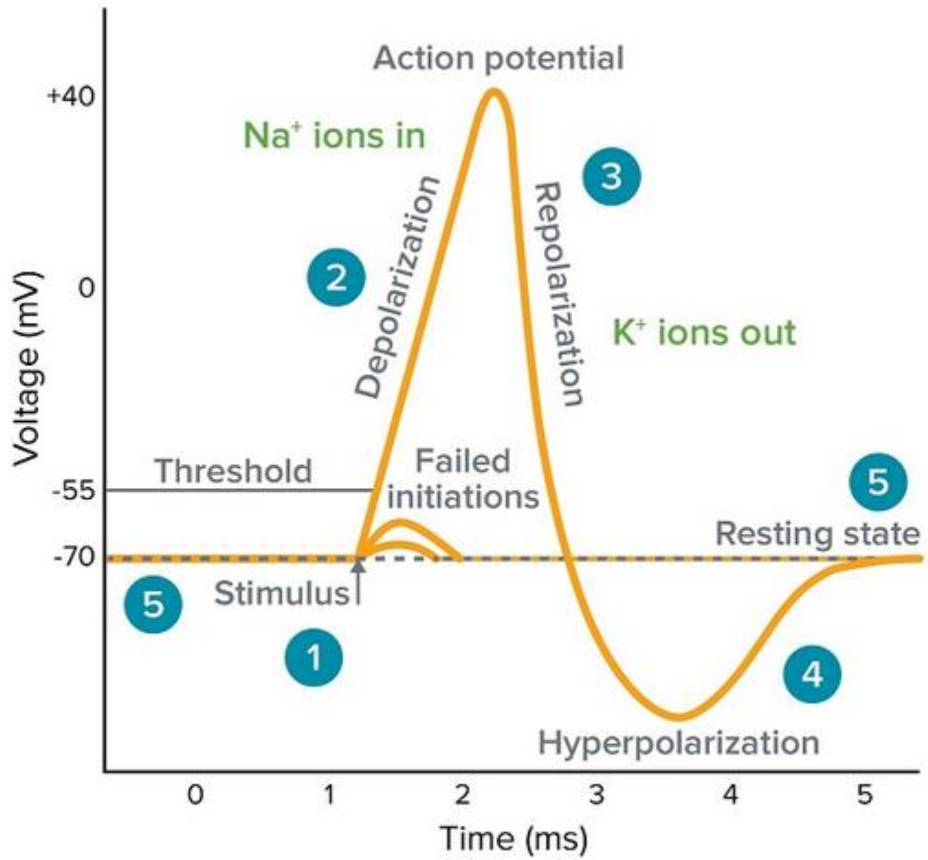
The biological model of neurons

- The way the brain of living things works is completely different from that of conventional numerical computers
- An artificial neural network is a simplified model of the biological brain
- The human brain has an average of 86 billion neurons and 150 trillion synapses
- The computations are parallel, complex and nonlinear
- Each neuron has a simple structure (only apparently...), but their interconnection provides the computing power

Interconnection of neurons



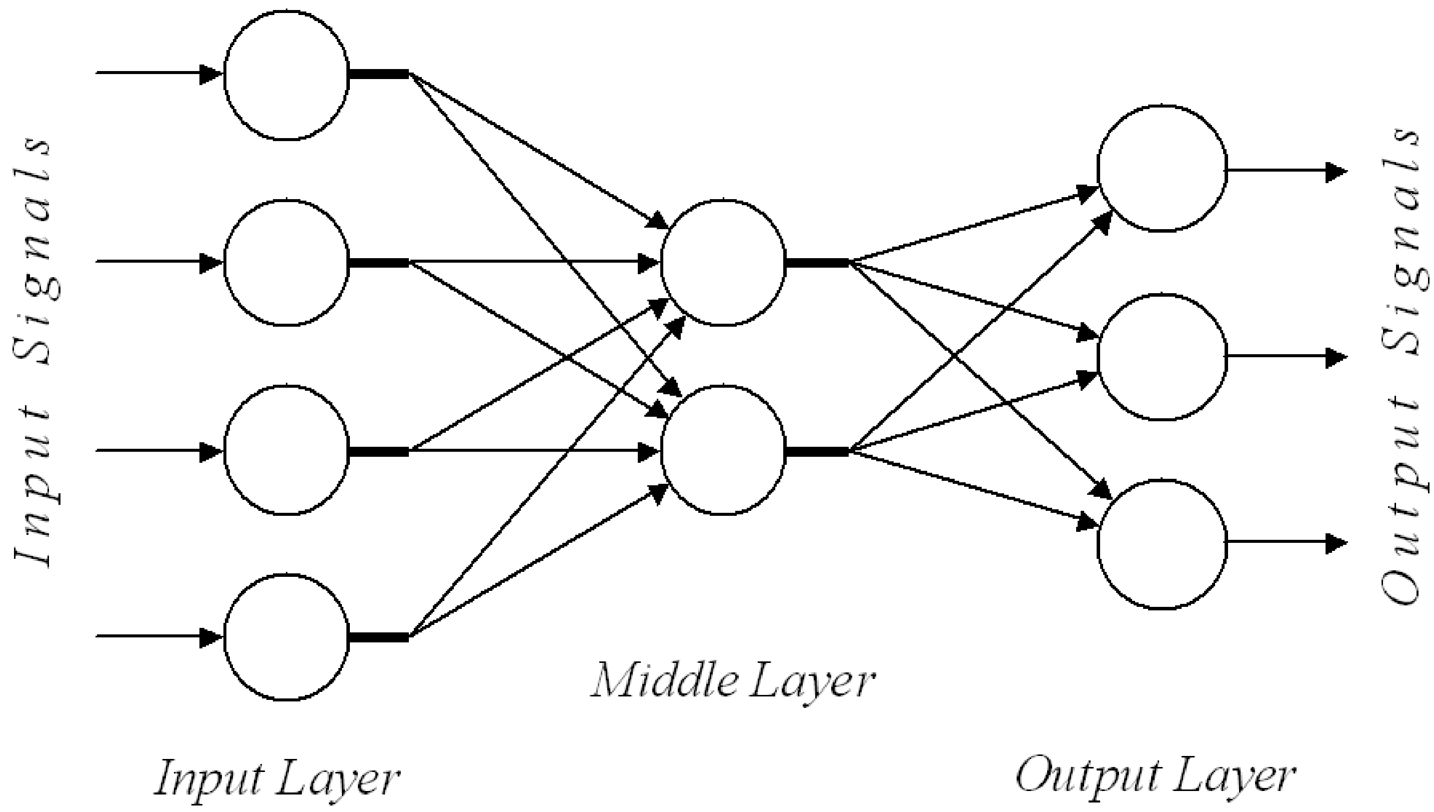
Typical neuronal impulse



The features of a biological neural network

- Information is stored and processed throughout the network: they are global, not local
- An essential feature is plasticity: the ability to adapt, to learn
- The possibility of learning led to the idea of modeling some (much) simplified neural networks using computers

Artificial neural networks



Analogies

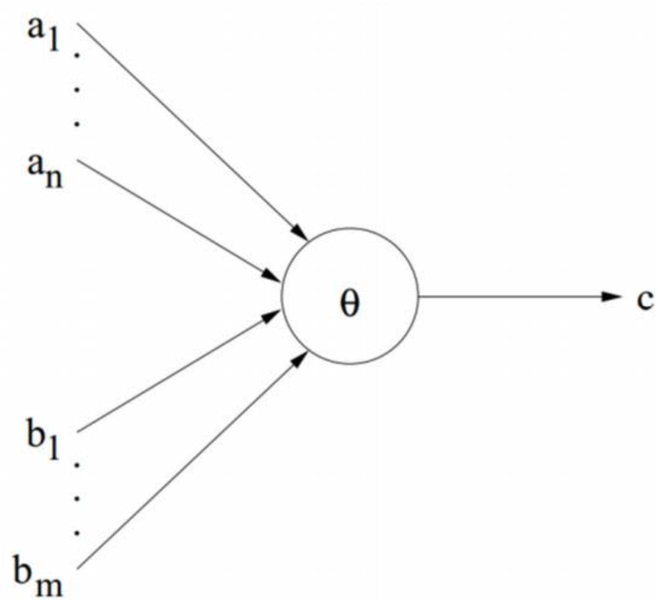
Biological NN	Artificial NN
Soma (cell body)	Neuron
Dendrites	Inputs
Axon	Output
Synapses	Weights

Neural Networks

1. Introduction
2. The Perceptron. Adaline
3. The Multilayer Perceptron
4. Deep Networks
5. Conclusions

The artificial neuron

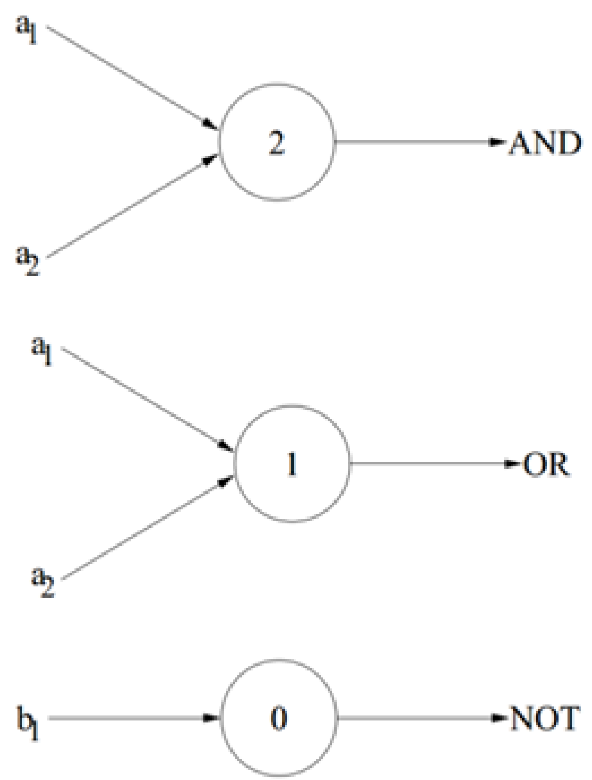
- The first mathematical model of a neuron was proposed by McCulloch and Pitts (1943)
- a_i are excitatory inputs, b_j are inhibitory inputs



$$c_{t+1} = \begin{cases} 1 & \text{If } \sum_{i=0}^{n} a_{i,t} \geq \theta \text{ and } b_{1,t} = \cdots = b_{m,t} = 0 \\ 0 & \text{Otherwise} \end{cases}$$

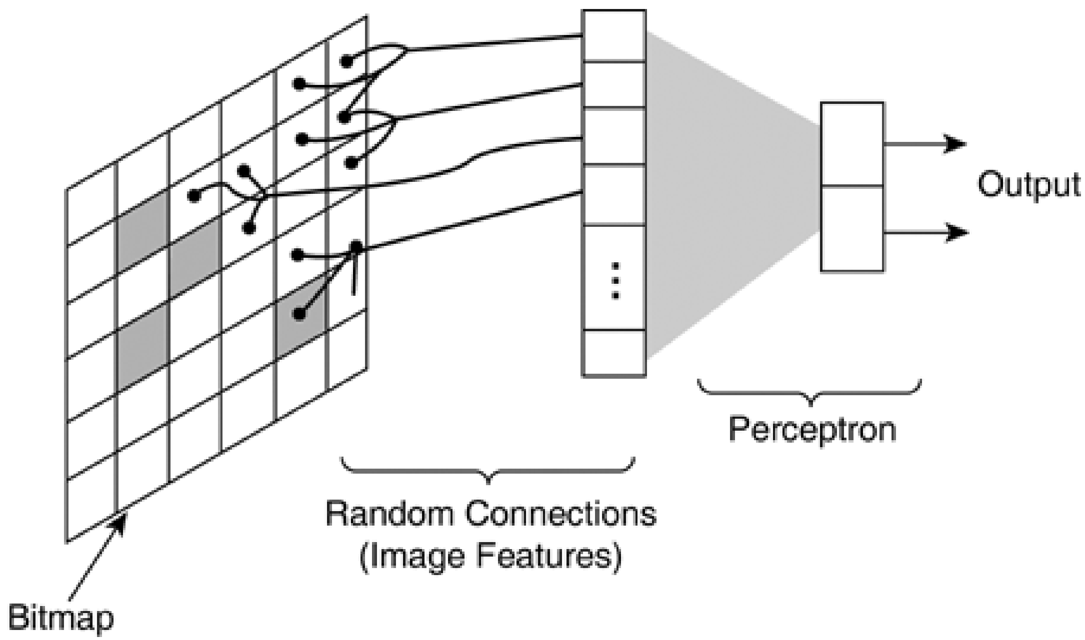
The McCulloch-Pitts neuron

- It cannot learn; the parameters are determined analytically



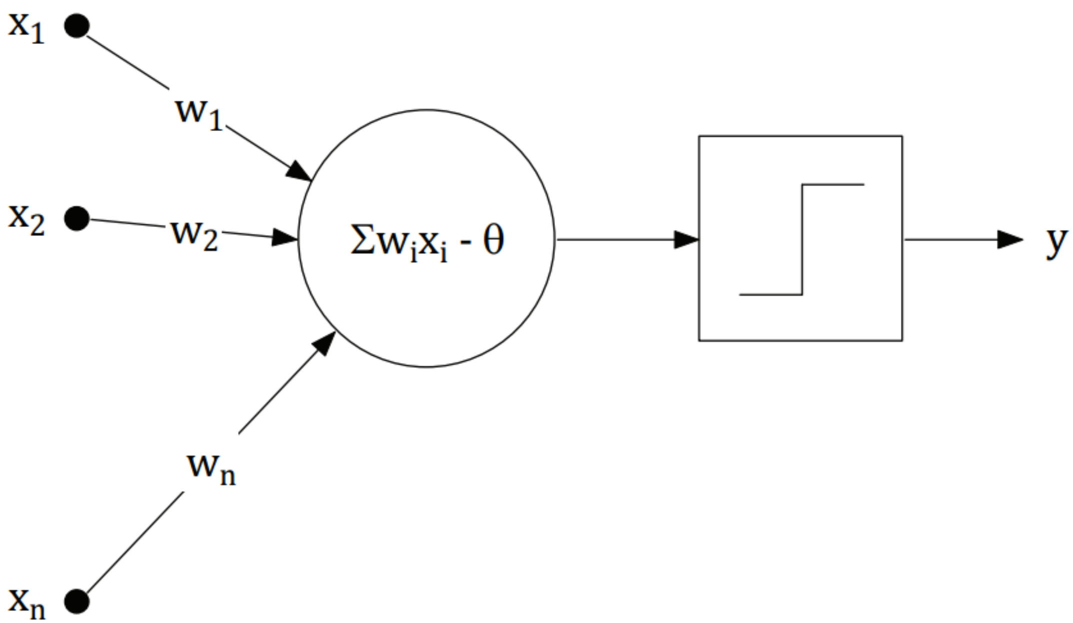
The original perceptron

- Rosenblatt (1958), trying to solve the problem of learning, proposed a model of neuron called perceptron, by analogy with the human visual system



The standard perceptron

- The input signals are summed, and the neuron generates a signal only if the sum exceeds the threshold



$$y = F\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

The perceptron

- The perceptron has adjustable synaptic weights and a sign or step activation function

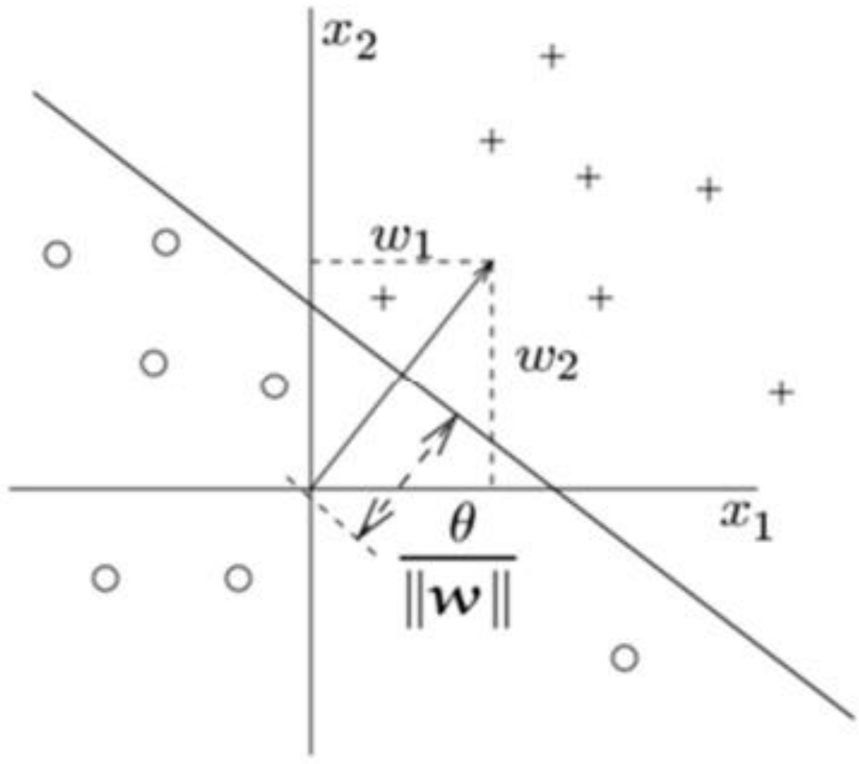
$$Y^{sign} = \begin{cases} +1, \text{if } X \geq 0 \\ -1, \text{if } X < 0 \end{cases} \qquad Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{if } X < 0 \end{cases}$$

Learning

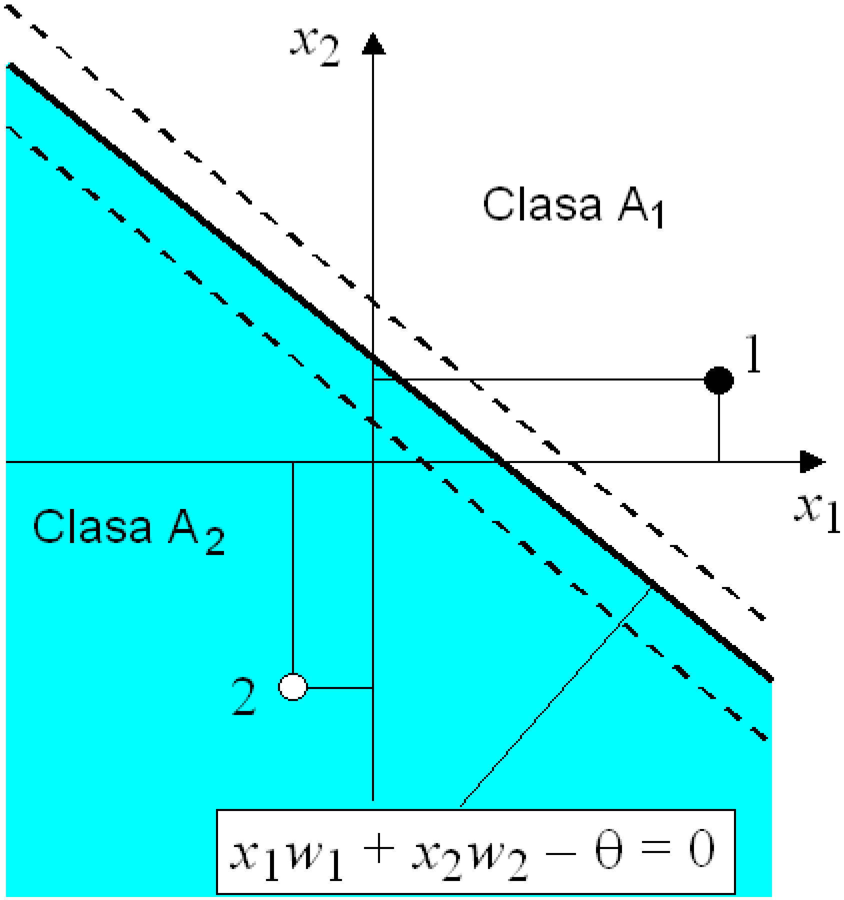
- The purpose of the perceptron is to classify the inputs x_1, x_2, ..., x_n with two classes A_1 and A_2
- The n-dimensional input space is divided into two by a hyperplane defined by the equation

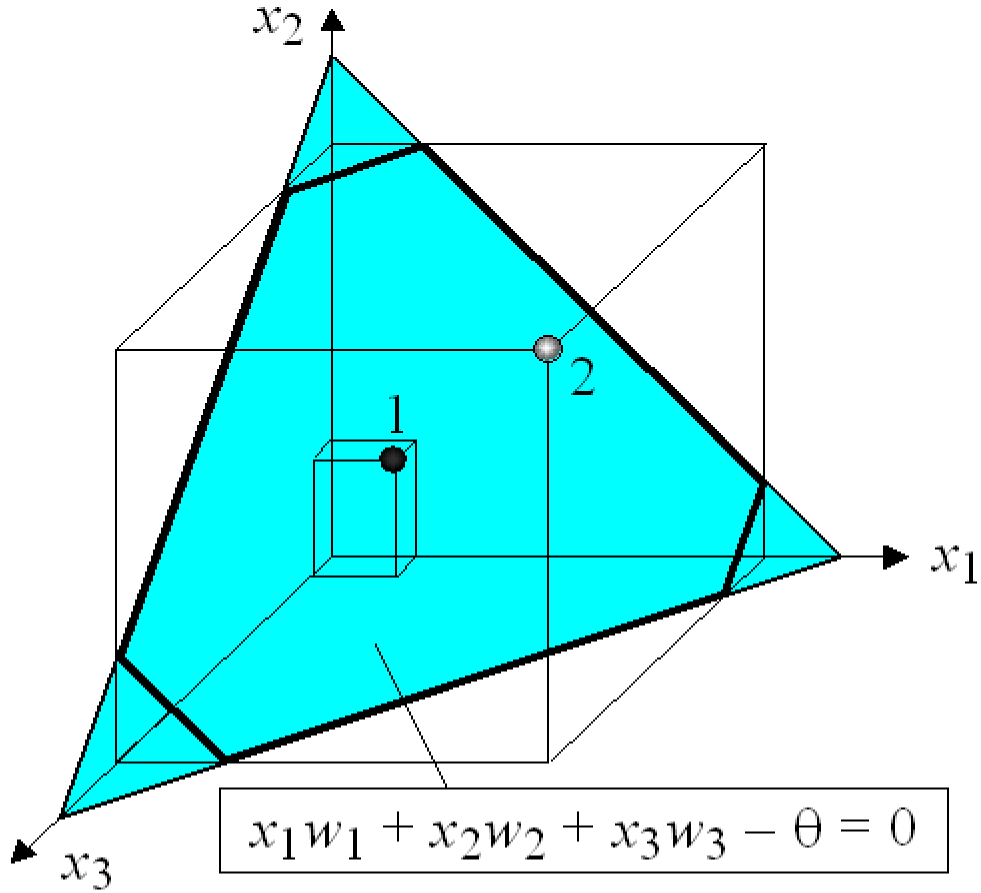
$$\boxed{\sum_{i=1}^{n} x_i w_i - \theta = 0}$$

Geometric interpretation

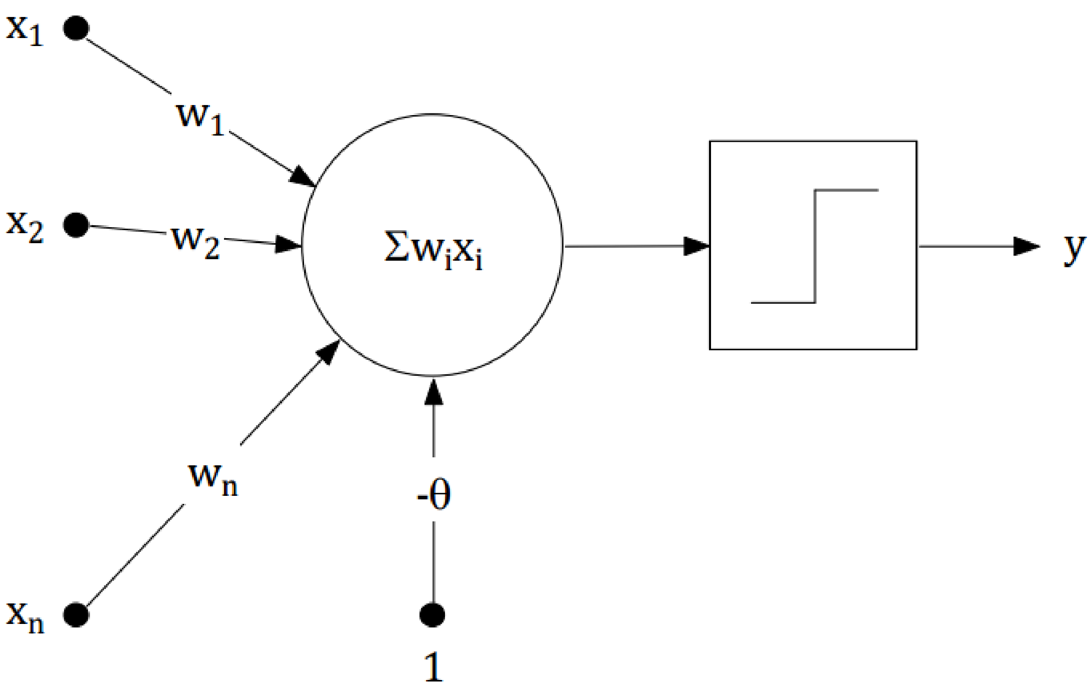


Examples: 2 and 3 inputs





The threshold as a weight



$$y = F\left(\sum_{i=1}^{n+1} w_i x_i\right)$$

net input

Learning

- Learning takes place by successively adjusting weights to reduce the difference between actual and desired outputs for all training data
- If, for a training vector, the actual output is y and the desired output is y_d, then the error is: $e = y_d - y$
- If the error is positive, we need to increase the perceptron output y
- If the error is negative, we need to decrease the output y

Learning

- If $y = 0$ and $y_d = 1 \Rightarrow e = 1$
- $x \cdot w = 0$
- $x \cdot w_d = 1$
- If $x > 0$, w should increase
- If $x < 0$, w should decrease

- Then: $\Delta w = \alpha \cdot x = \alpha \cdot x \cdot e$
- α is the learning rate

Learning

- If $y = 1$ and $y_d = 0 \Rightarrow e = -1$
- $x \cdot w = 1$
- $x \cdot w_d = 0$
- If $x > 0$, w should decrease
- If $x < 0$, w should increase

- Then: $\Delta w = \alpha \cdot (-x) = \alpha \cdot x \cdot e$

Learning

- The perceptron learning rule:

  $\Delta w = \alpha \cdot x \cdot e$

- Using the step activation function

The perceptron training algorithm

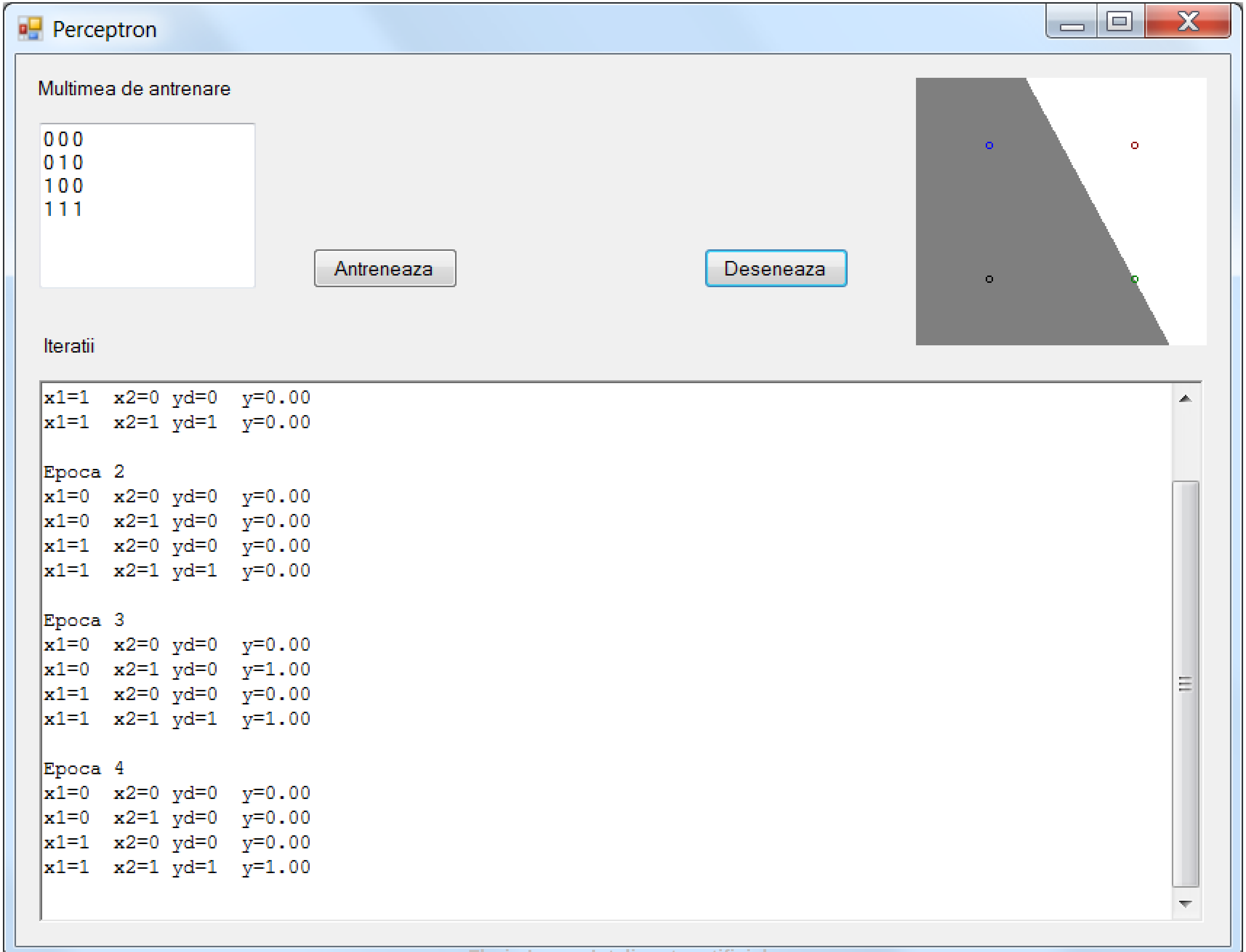
all weights w_i are initialized with 0 or random values in the [–0.5, 0.5] interval
the learning rate alpha is initialized with a value in the (0, 1] interval, e.g. 0.1
the maximum number of epochs P is initialized, e.g. 100
p = 0 *// the number of current epoch*
errors = *true // a flag indicating the existence of training errors*

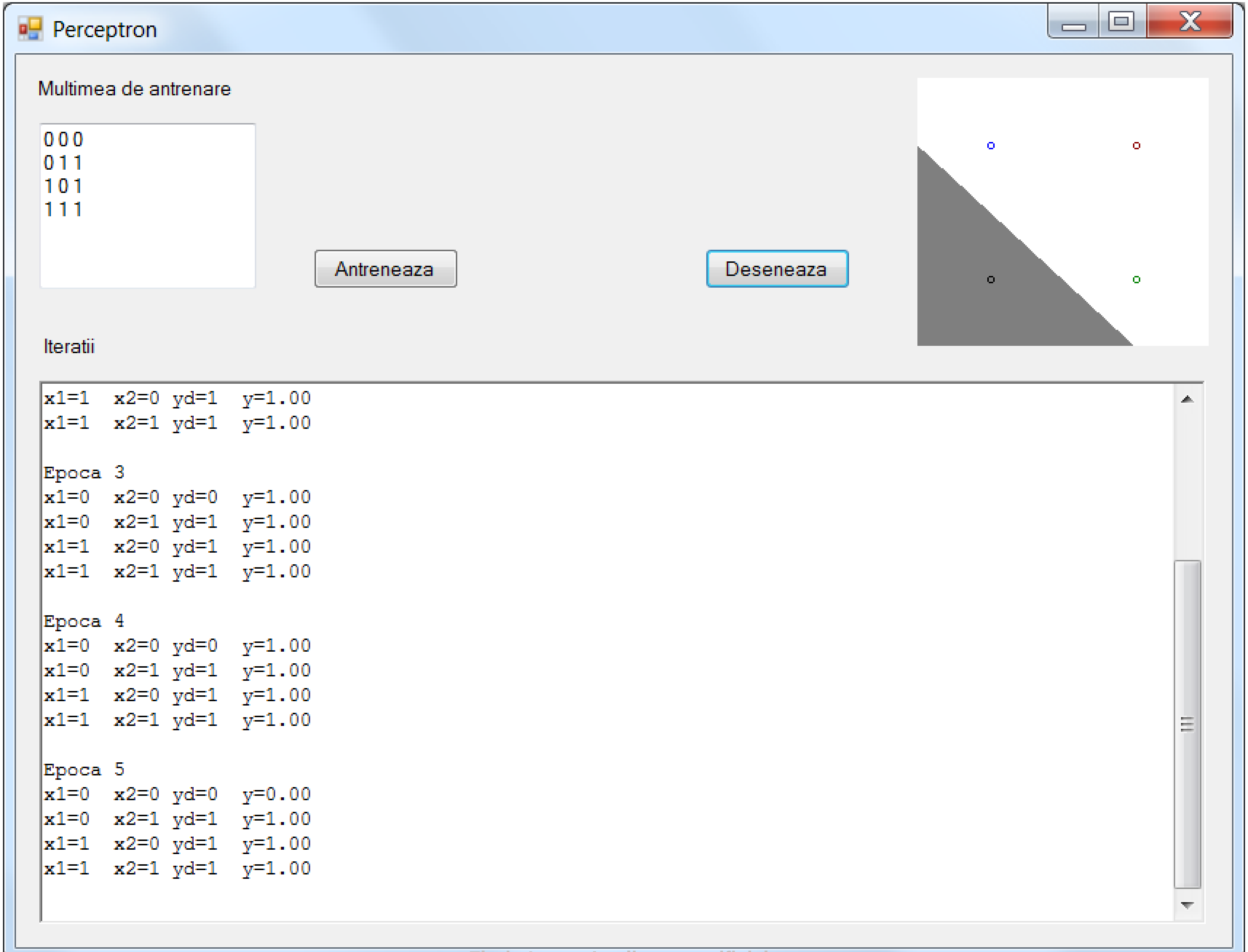
```
while p < P and errors == true
{
    errors = false
    for each training vector x_i with i = 1..N
    {
        y_i = F(sum(x_ij * w_j)) with j = 1..n+1
        if (y_i != y_di)
        {
            e = y_di – y_i
            errors = true
            for each input j = 1..n+1
                w_j = w_j + alpha * x_ij * e
        }
    }
    p = p + 1
}
```

Training example: the AND logic function

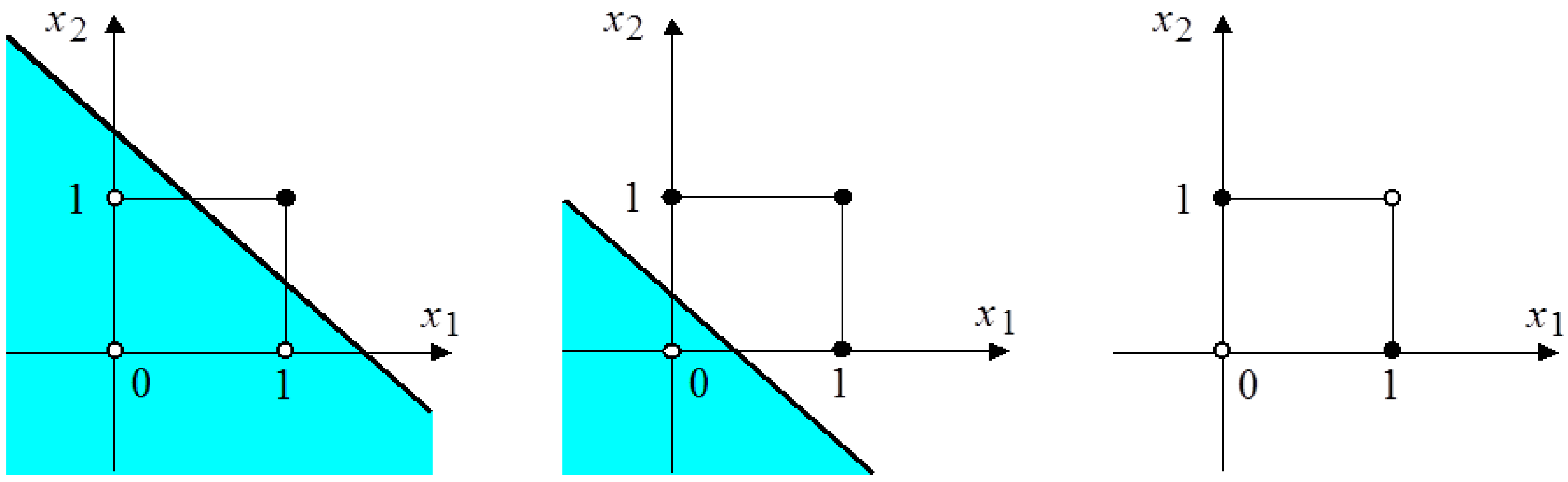
Epoch	Inputs x_1	Inputs x_2	Desired output Y_d	Initial weights w_1	Initial weights w_2	Actual output Y	Error e	Final weights w_1	Final weights w_2
1	0	0	0	0.3	–0.1	0	0	0.3	–0.1
	0	1	0	0.3	–0.1	0	0	0.3	–0.1
	1	0	0	0.3	–0.1	1	–1	0.2	–0.1
	1	1	1	0.2	–0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	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: $\alpha = 0.1$


Perceptron
Multimea de antrenare
0 0 0
0 1 0
1 0 0
1 1 1
Antreneaza
Deseneaza
Iteratii
x1=1 x2=0 yd=0 y=0.00
x1=1 x2=1 yd=1 y=0.00
Epoca 2
x1=0 x2=0 yd=0 y=0.00
x1=0 x2=1 yd=0 y=0.00
x1=1 x2=0 yd=0 y=0.00
x1=1 x2=1 yd=1 y=0.00
Epoca 3
x1=0 x2=0 yd=0 y=0.00
x1=0 x2=1 yd=0 y=1.00
x1=1 x2=0 yd=0 y=0.00
x1=1 x2=1 yd=1 y=1.00
Epoca 4
x1=0 x2=0 yd=0 y=0.00
x1=0 x2=1 yd=0 y=0.00
x1=1 x2=0 yd=0 y=0.00
x1=1 x2=1 yd=1 y=1.00


Perceptron
Multimea de antrenare
0 0 0
0 1 1
1 0 1
1 1 1
Antreneaza
Deseneaza
Iteratii
x1=1 x2=0 yd=1 y=1.00
x1=1 x2=1 yd=1 y=1.00
Epoca 3
x1=0 x2=0 yd=0 y=1.00
x1=0 x2=1 yd=1 y=1.00
x1=1 x2=0 yd=1 y=1.00
x1=1 x2=1 yd=1 y=1.00
Epoca 4
x1=0 x2=0 yd=0 y=1.00
x1=0 x2=1 yd=1 y=1.00
x1=1 x2=0 yd=1 y=1.00
x1=1 x2=1 yd=1 y=1.00
Epoca 5
x1=0 x2=0 yd=0 y=0.00
x1=0 x2=1 yd=1 y=1.00
x1=1 x2=0 yd=1 y=1.00
x1=1 x2=1 yd=1 y=1.00

Graphic representation



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

- The perceptron can only represent linearly separable functions

Discussion

- The perceptron can represent complex linearly separable functions, for example, the majority function
  - A decision tree would need 2^n nodes
- It can learn everything it can represent, but it cannot represent many functions
  - It cannot represent non-linearly separable functions, e.g. XOR

Adaline

- Linear activation function

$$y = \sum_{i=1}^{n+1} w_i x_i$$

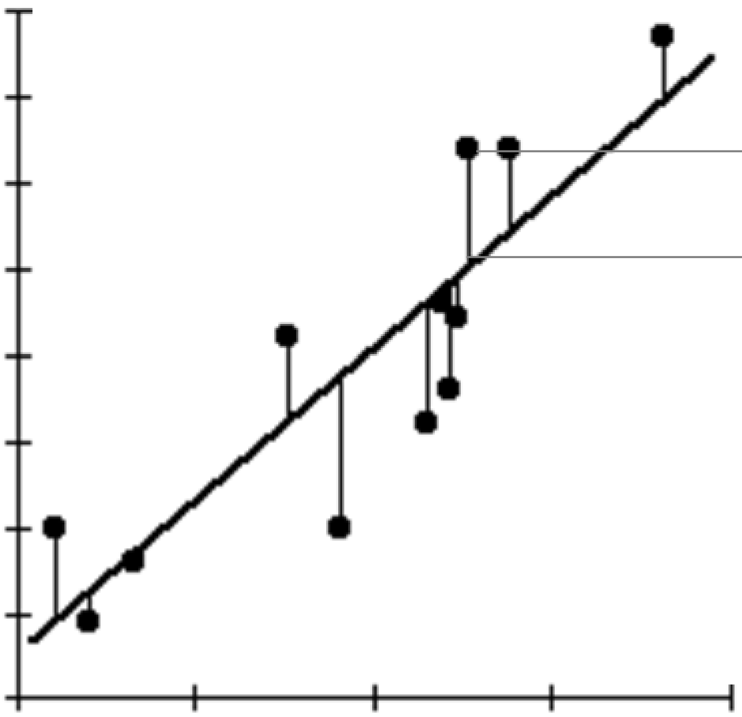
- The error of a training vector

$$E_i = \frac{1}{2}\left(y_{di} - y_i\right)^2$$

- The error on the training set

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i = \frac{1}{2N}\sum_{i=1}^{N}\left(y_{di} - y_i\right)^2$$

Mean square error (MSE)



$$y_{di} - y_i$$

$$E_i = \frac{1}{2}\left(y_{di} - y_i\right)^2$$

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i = \frac{1}{2N}\sum_{i=1}^{N}\left(y_{di} - y_i\right)^2$$

The training rule

$$\Delta_i w_j = -\alpha \frac{\partial E_i}{\partial w_j}$$

$$\frac{\partial E_i}{\partial w_j} = \frac{\partial E_i}{\partial y_i} \frac{\partial y_i}{\partial w_j}$$

$$\frac{\partial E_i}{\partial y_i} = -\left(y_{di} - y_i\right)$$

$$\frac{\partial y_i}{\partial w_j} = \frac{\partial \left(\sum_k x_{ik} w_k \right)}{\partial w_j} = \frac{\partial x_{ij} w_j}{\partial w_j} = x_{ij}$$

$$\Delta w = \alpha \cdot x \cdot e \quad \longleftarrow$$ α is the learning rate

Delta rule

- In the general case:

(derivable) activation function

$$\frac{\partial E_i}{\partial w_j} = -x_{ij}\left(y_{di} - y_i\right) f'(y_i)$$

- The weight updates:

$$\Delta_i w_j = \alpha \cdot x_{ij} \cdot e_i \cdot f'(y_i)$$

The delta rule for the linear activation function

$$\Delta_i w_j = \alpha \cdot x_{ij} \cdot e_i \cdot f'(y_i) \qquad f'(y_i) = 1$$

$$\boxed{w_j(p+1) = w_j(p) + \alpha \cdot x_{ij}(p) \cdot e_i(p)}$$

p = the step after which the updates are made
(the training vector or epoch)

- It is the same expression as in the perceptron learning rule
- But in the previous case the step activation function was used, which is not derivable
- The delta rule can only be applied for derivable activation functions

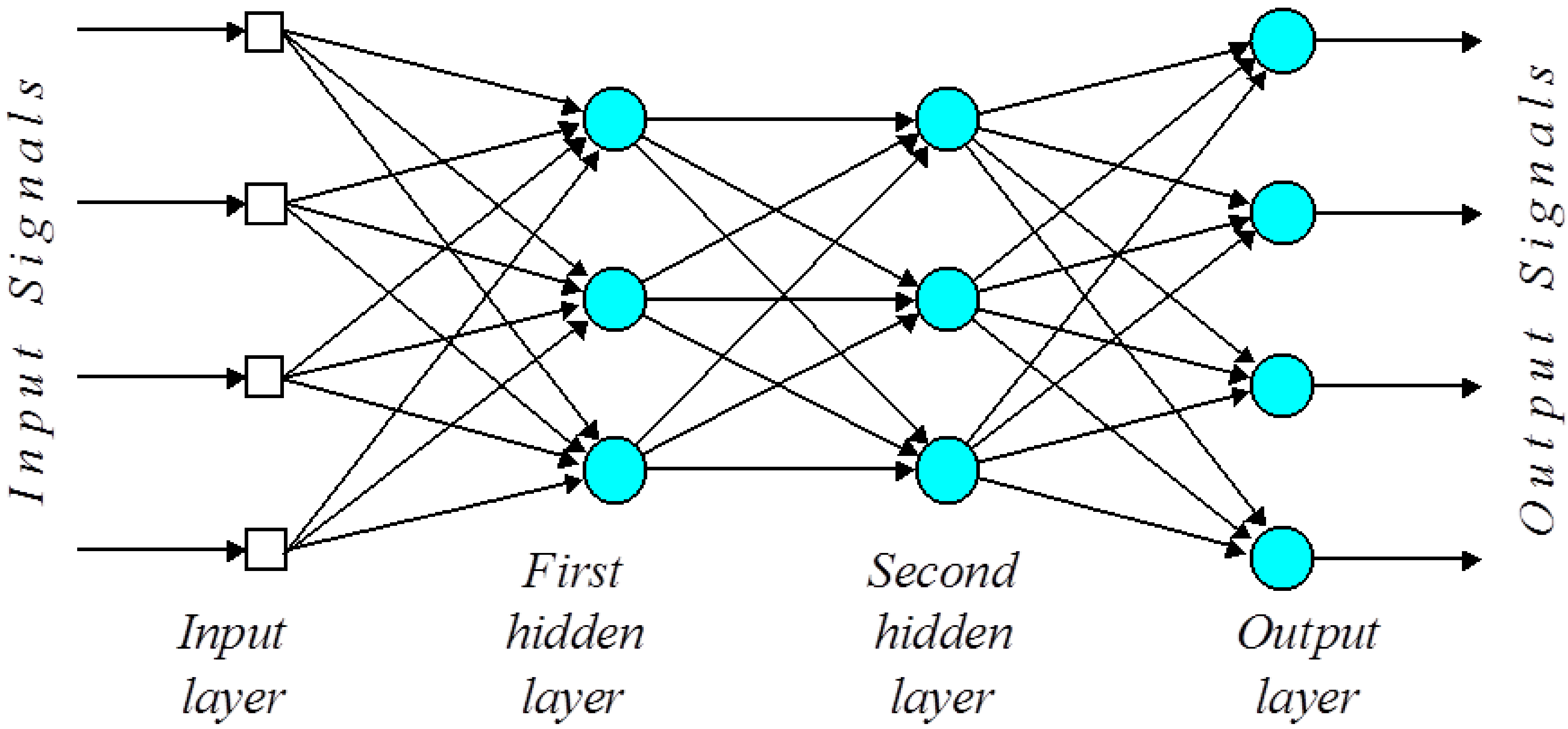
Neural Networks

1. Introduction
2. The Perceptron. Adaline
3. The Multilayer Perceptron
4. Deep Networks
5. Conclusions

The multilayer perceptron

- The multilayer perceptron is a feed-forward neural network with one or more hidden layers
  - An input layer
  - One or more hidden / intermediate layers
  - An output layer
- The computations are performed only in neurons from the hidden layers and from the output layer
- The input signals are propagated forward successively through the layers of the network

A multilayer perceptron with two hidden layers



The “hidden” layers

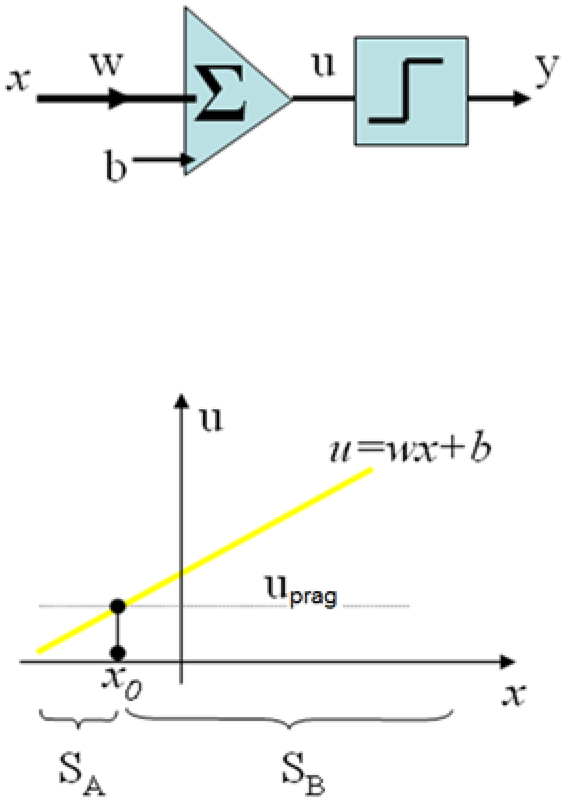
- A hidden layer “hides” its desired output. Knowing the input-output mapping of the network (“black box”), it is not obvious what the desired output of a neuron in a hidden layer should be
- Classical neural networks have 1 or 2 hidden layers. Each layer can contain, for example, 10 – 50 – 100 neurons

Universal approximation property

- A neural network with a single hidden layer, possibly with an infinite number of neurons, can approximate any continuous real function

- However, an extra layer can greatly reduce the number of neurons needed in the hidden layers

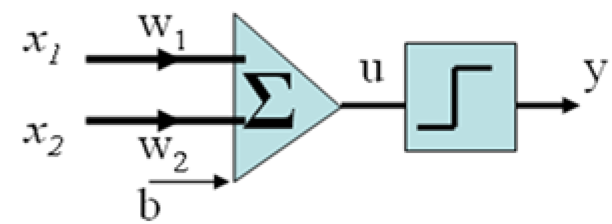
Perceptron with 1 input

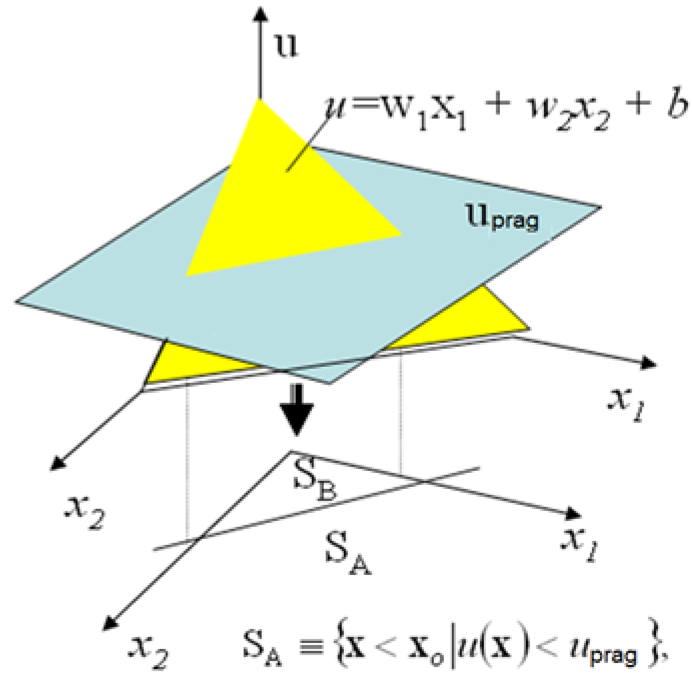


$$S_A \equiv \{x < x_o \mid u(x) < u_{\text{prag}}\},$$

$$S_B \equiv \{x \geq x_o \mid u(x) \geq u_{\text{prag}}\},$$

Perceptron with 2 inputs

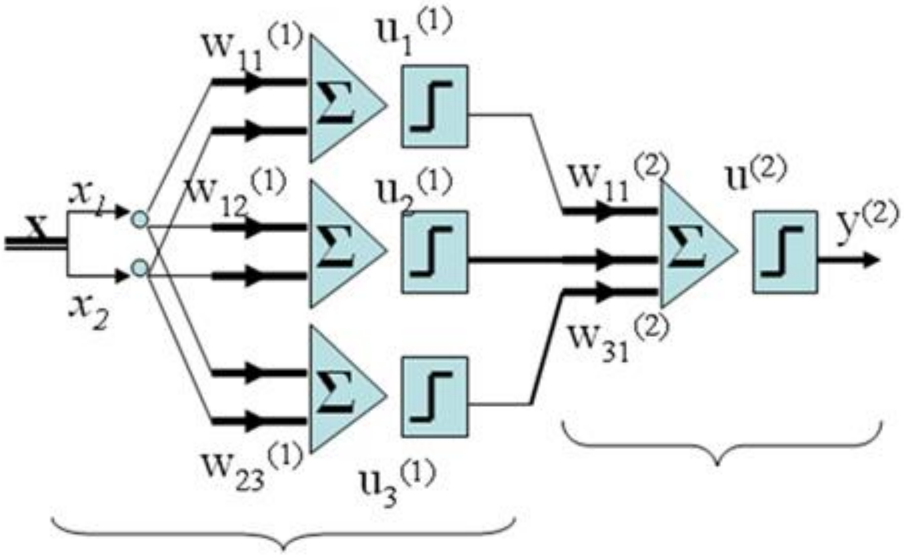


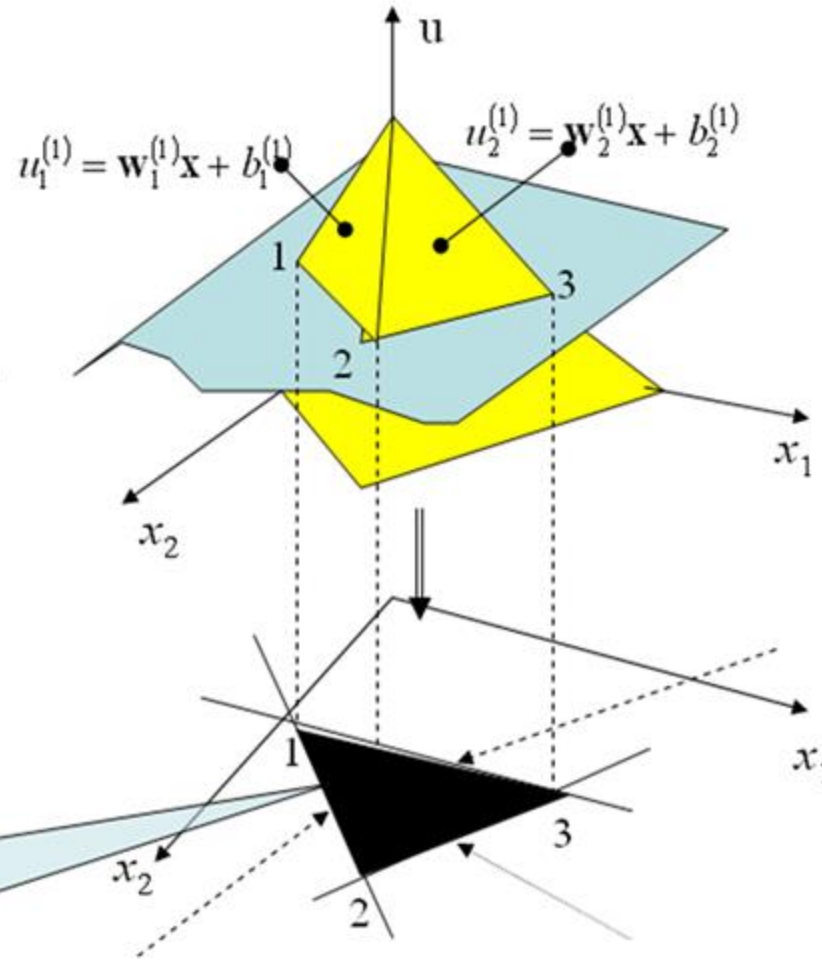


$$S_A \equiv \{\mathbf{x} < \mathbf{x}_o \mid u(\mathbf{x}) < u_{\text{prag}}\},$$

$$S_B \equiv \{\mathbf{x} \geq \mathbf{x}_o \mid u(\mathbf{x}) \geq u_{\text{prag}}\}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

Multilayer perceptron with 1 hidden layer

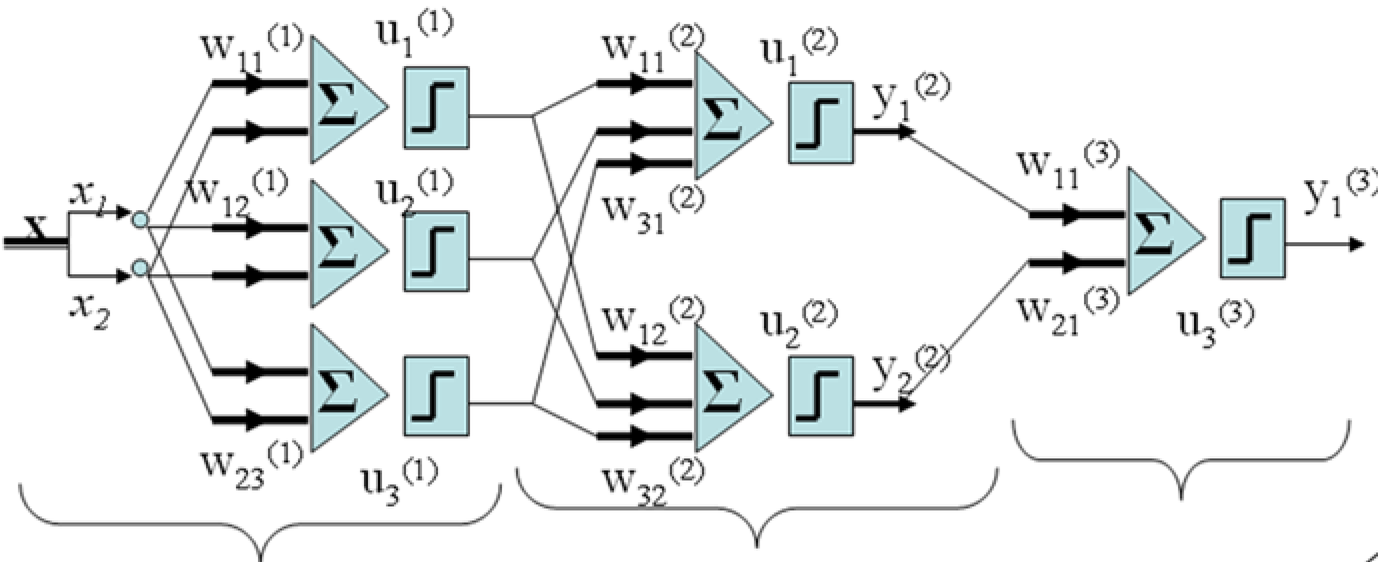


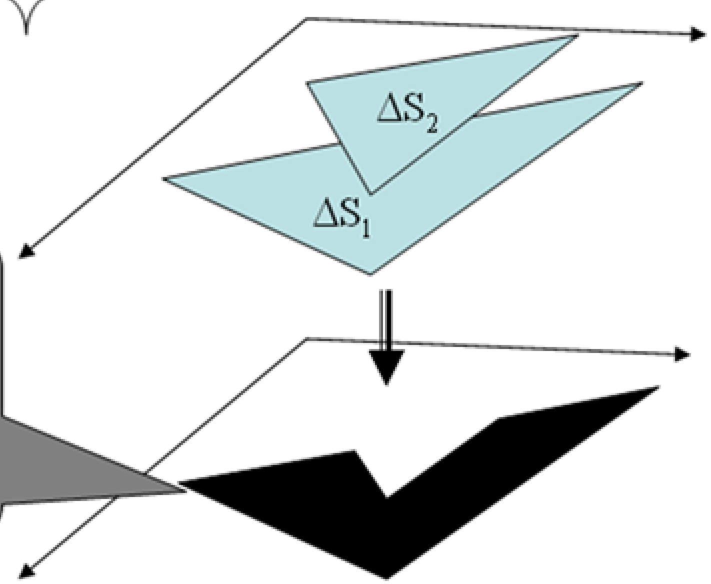


Convex decision boundary

$$\Delta S_A = S_1 \cap S_2 \cap \overline{S}_3$$

Multilayer perceptron with 2 hidden layers





Arbitrary decision boundary

$$Z = \Delta S_1 \cup \Delta \overline{S}_2$$

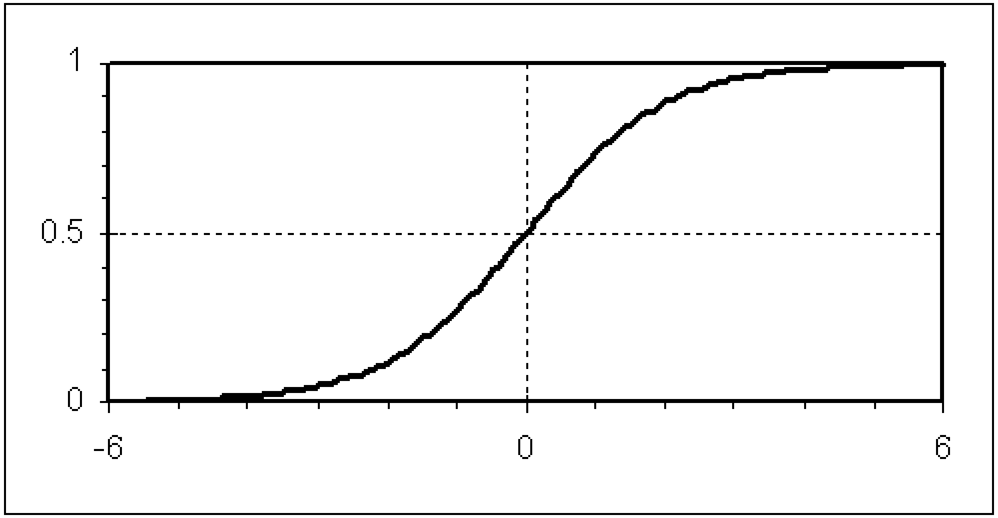
$$\Delta S_1 \equiv \{ \mathbf{x}_i \mid u_1^{(2)} \geq u_{\text{prag}} \}$$

$$\Delta S_2 \equiv \{ \mathbf{x}_i \mid u_2^{(2)} \geq u_{\text{prag}} \}$$

Activation functions

- The most commonly used functions are the unipolar sigmoid (or logistic function):

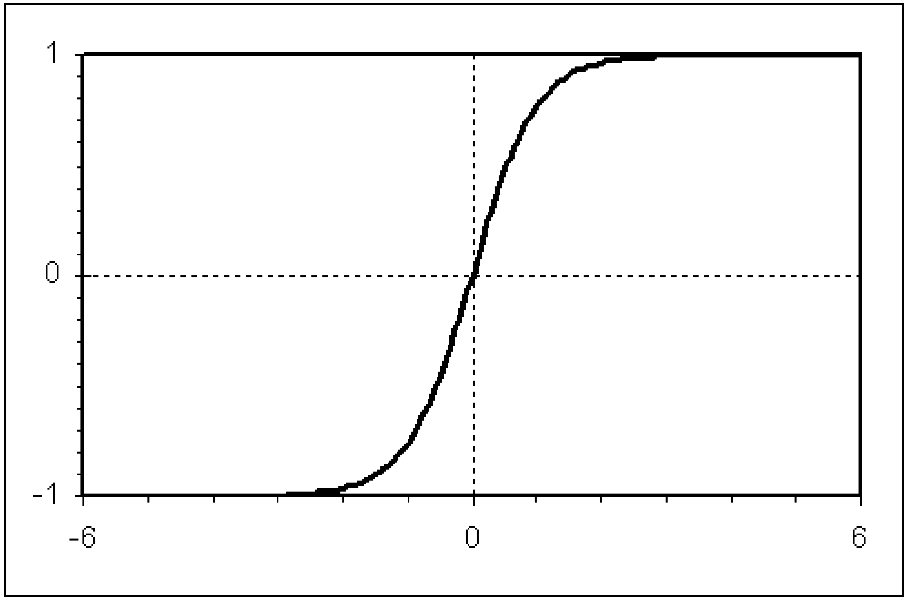
$$f(x) = \frac{1}{1 + e^{-x}}$$



Activation functions

- and especially the bipolar sigmoid (the hyperbolic tangent):

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$



Discussion

- A single-layer perceptron has the same limitations even if it uses a nonlinear activation function
- A multilayer perceptron with linear activation functions is equivalent to a single-layer perceptron
- A linear combination of linear functions is also a linear function. For example:
  - $f(x) = 2x + 2$
  - $g(y) = 3y - 3$
  - $g(f(x)) = 3(2x + 2) - 3 = 6x + 3$

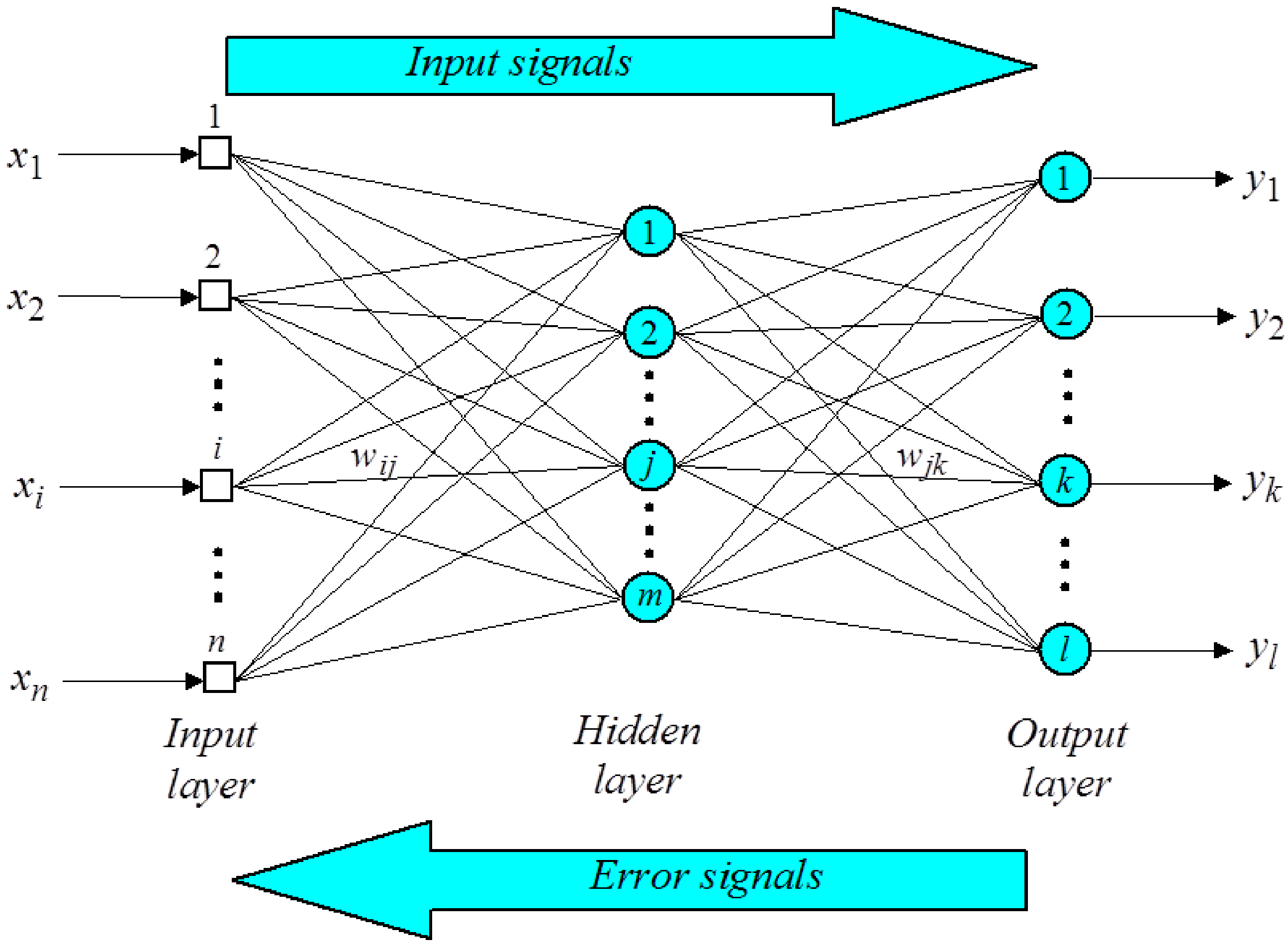
Learning

- A multilayer network learns in a similar way as the perceptron
- The network receives the input vectors and computes the output vectors
- If there is an error (a difference between the desired output and the actual output), the weights are adjusted to reduce the error

The backpropagation algorithm

- Authors:
  - Bryson & Ho, 1969
  - Rumelhart, Hinton & Williams, 1986
- The algorithm has two phases:
  - The network receives the input vector and propagates the signal forward, layer by layer, until the output is generated
  - The error signal is propagated back, from the output layer to the input layer, adjusting the weights of the network

The phases of the backpropagation algorithm



Step 1. Initialization

- Set all the weights and thresholds of the network to small, but non-zero random values
- In general, they can be in the [-0.1, 0.1] interval
- A heuristic recommends values from the interval $(-2.4 / F_i , 2.4 / F_i)$, where F_i is the number of inputs of neuron *i* (fan-in)
- The initialization of the weights is done independently for each neuron
- In the following, we will use the following indices: *i* for the input layer, *j* for the hidden layer and *k* for the output layer

Step 2. Activation

- Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ with desired outputs $y^d{}_1(p), y^d{}_2(p), \ldots, y^d{}_o(p)$
- 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 a sigmoid activation function is used

Step 2. Activation

- Calculate the actual outputs of the neurons in the output layer:

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

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

Step 3a. Weight updating

- Calculate the error gradient for the neurons in the output layer, with the delta rule

see next slide

$$\boxed{\delta_k(p) = \overbrace{y_k(p) \cdot [1 - y_k(p)]}\cdot e_k(p)}$$

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

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

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

The derivative of the activation function

For unipolar sigmoid

$$f'(x) = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = f(x)\cdot\left(1-f(x)\right).$$

For bipolar sigmoid

$$f'(x) = \frac{2a\cdot e^{-a\cdot x}}{\left(1+e^{-a\cdot x}\right)^2} = \frac{a}{2}\cdot\left(1-f(x)\right)\cdot\left(1+f(x)\right).$$

Assuming a unipolar sigmoid:

$$\delta_k(p) = y_k(p)\cdot\left(1-y_k(p)\right)\cdot e_k(p).$$

Step 3b. Weight updating

- 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

- Increment *p*
- The presentation of all training vectors (instances) to the network represents and epoch
- The training of the network continues until the mean square error reaches an acceptable threshold or until a predefined maximum number of training epochs is reached

$$E = \frac{1}{V} \sum_{p=1}^{V} \sum_{k=1}^{O} \left(y_k^d(p) - y_k(p) \right)^2$$

V is the number of training vectors
O is the number of outputs

Justification

- The gradient descent formulas are based on the idea of minimizing the network error by adjusting the weights
- But $\partial E / \partial w$ cannot be calculated directly, so the chain rule applies:

$$\frac{\partial E}{\partial w} = \underbrace{\frac{\partial E}{\partial y}}_{e} \cdot \underbrace{\frac{\partial y}{\partial net}}_{f'} \cdot \underbrace{\frac{\partial net}{\partial w}}_{x}$$

Justification

$$\frac{\partial E}{\partial w} = \underbrace{\frac{\partial E}{\partial y}}_{e} \cdot \underbrace{\frac{\partial y}{\partial net}}_{f'} \cdot \underbrace{\frac{\partial net}{\partial w}}_{x}$$

- The differentials on the right side can be calculated, and finally the weights are adjusted:

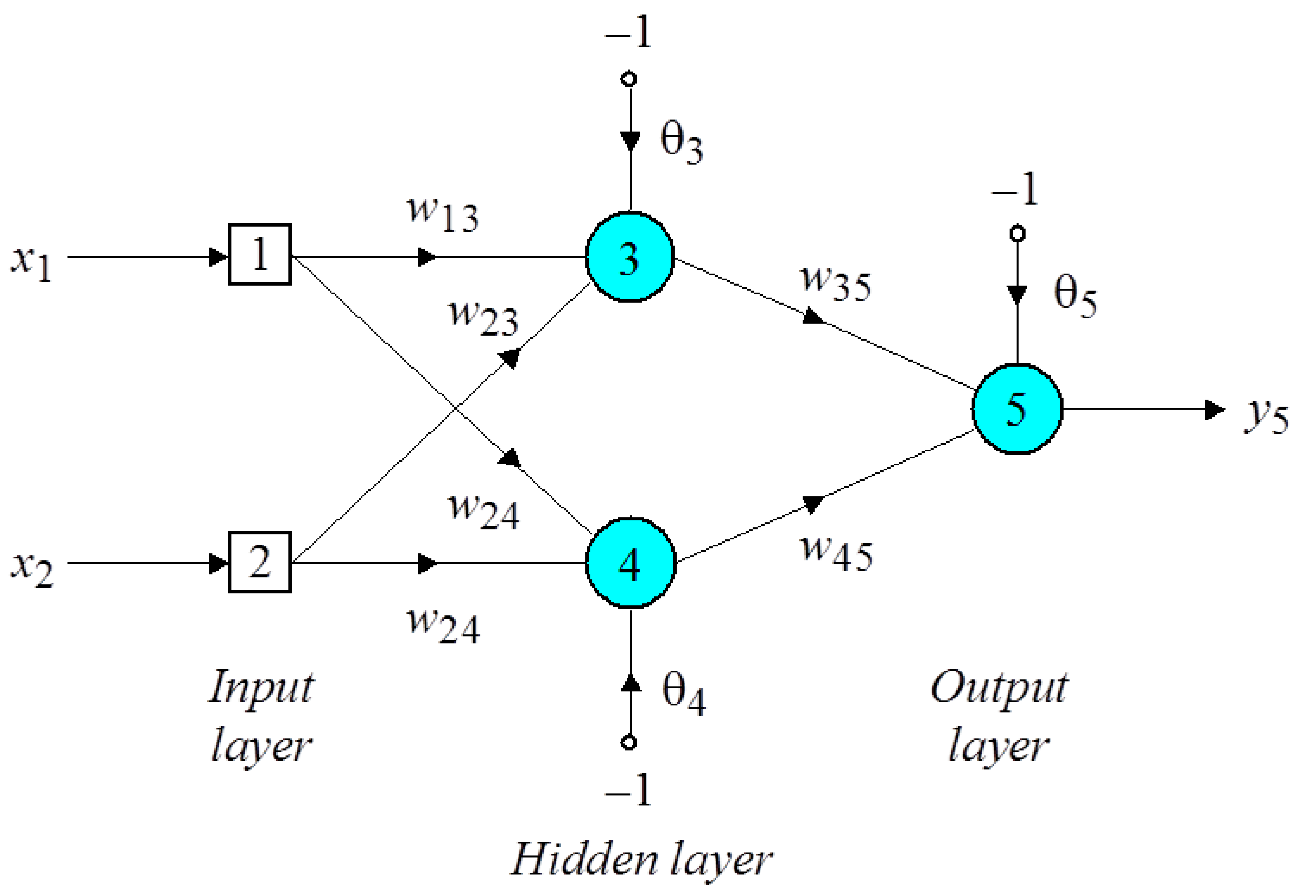
$$\Delta w_{jk} = \alpha \cdot y_j \cdot [y_k(1 - y_k)] \cdot e_k$$

$$\Delta w_{ij} = \alpha \cdot x_i \cdot [y_j(1 - y_j)] \cdot \sum_k \delta_k w_{jk}$$

Networks with more hidden layers

- The adjustment of the weights in the output layer is done as in step 3a
- Adjusting the weights of each hidden layer is done as in step 3b
- If the network has many hidden layers, the gradients are small and the training is very slow or does not converge at all
- Therefore, recent deep learning architectures use other training methods

Example: a network with one hidden layer for approximating the XOR binary function



Initialization

- The threshold applied to a neuron in a hidden or output layer is equivalent to another connection with the weight equal to θ, connected to a fixed input equal to –1
- The weights and the thresholds are initialized randomly, for example:
  - $w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$
  - $w_{35} = 1.2$, $w_{45} = 1.1$
  - $\theta_3 = 0.8$, $\theta_4 = 0.1$, $\theta_5 = 0.3$

- 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:

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

- The following error is obtained:

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

- We propagate the error 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 α is 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:

$$\delta_3 = y_3(1-y_3)\cdot\delta_5\cdot w_{35} = 0.5250\cdot(1-0.5250)\cdot(-0.1274)\cdot(-1.2) = 0.0381$$
$$\delta_4 = y_4(1-y_4)\cdot\delta_5\cdot w_{45} = 0.8808\cdot(1-0.8808)\cdot(-0.1274)\cdot 1.1 = -0.0147$$

- We then determine the weight/threshold 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 thresholds:

$$
\begin{aligned}
w_{13} &= w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038 \\
w_{14} &= w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985 \\
w_{23} &= w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038 \\
w_{24} &= w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985 \\
w_{35} &= w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067 \\
w_{45} &= w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888 \\
\theta_{3} &= \theta_{3} + \Delta\theta_{3} = 0.8 - 0.0038 = 0.7962 \\
\theta_{4} &= \theta_{4} + \Delta\theta_{4} = -0.1 + 0.0015 = -0.0985 \\
\theta_{5} &= \theta_{5} + \Delta\theta_{5} = 0.3 + 0.0127 = 0.3127
\end{aligned}
$$

- The training process is repeated until the mean square error is less than an acceptable value, e.g. here, 0.001

The final results

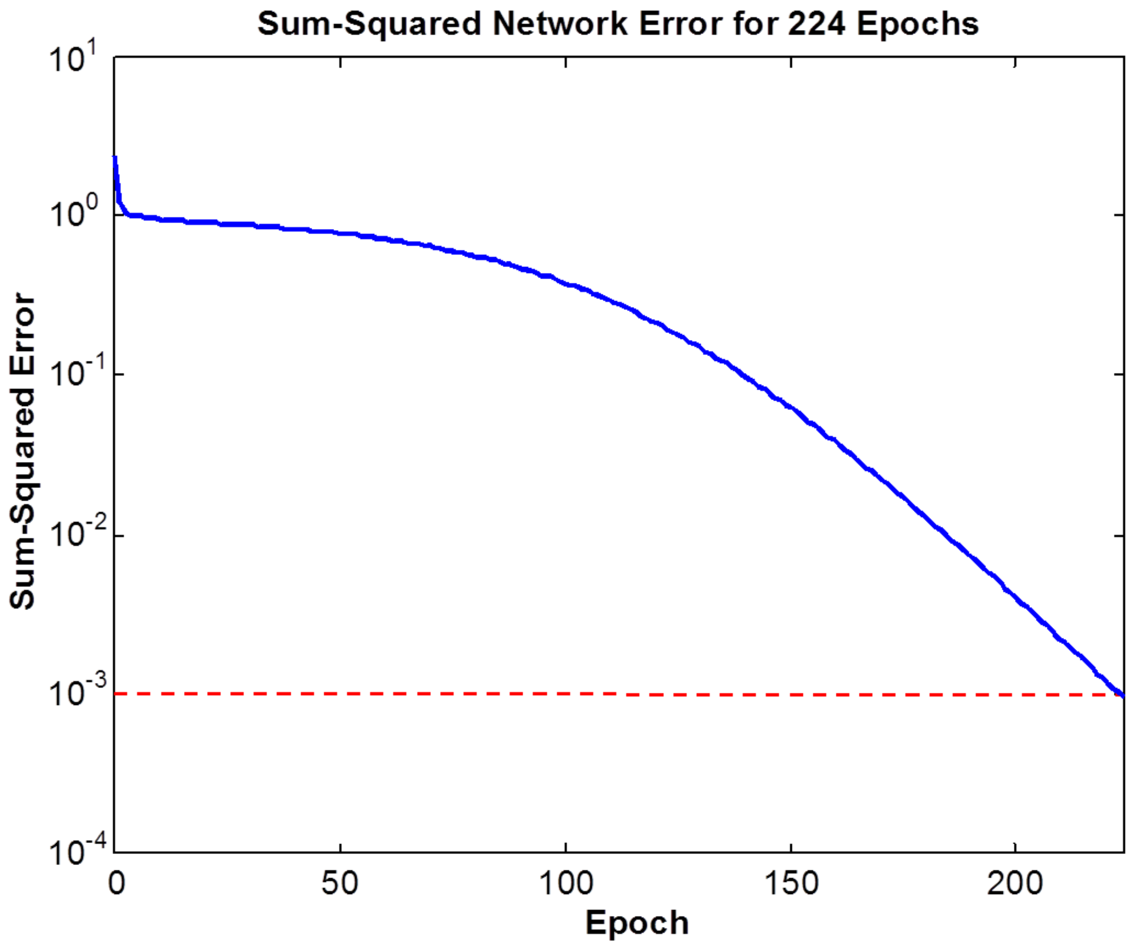
Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	–0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	–0.0175	

$$E = \frac{1}{V}\sum_{p=1}^{V}\sum_{k=1}^{O}\left(y_k^d(p) - y_k(p)\right)^2$$

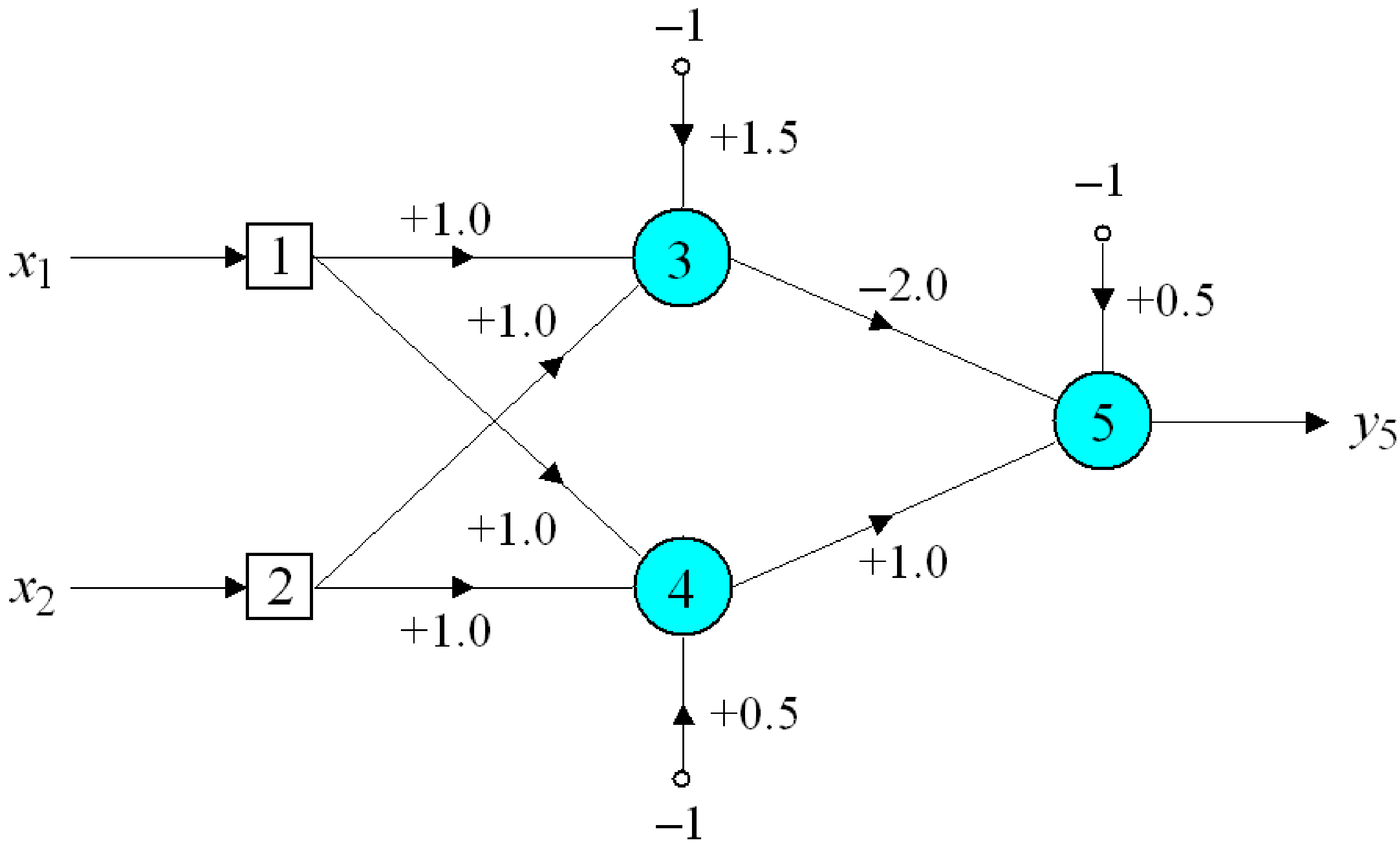
V is the number of training vectors (here 4)

O is the number of outputs (here 1)

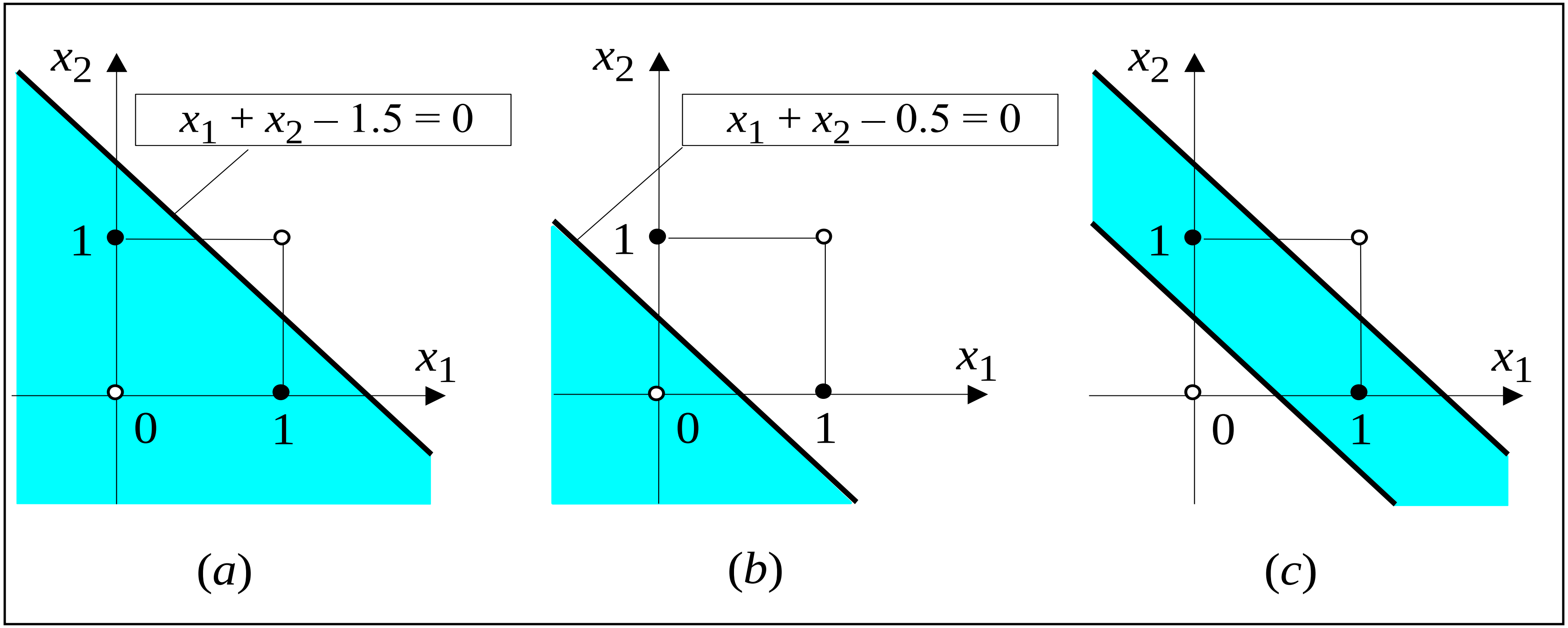
Learning curve for XOR



The final parameters of the network



Decision boundaries



(a) Decision boundary constructed by hidden neuron 3

(b) Decision boundary constructed by hidden neuron 3

(c) Decision boundaries constructed by the complete network: neuron 5 combines the boundaries

Types of training

- Incremental learning (online learning)
  - The weights are updated after processing each training vector
  - As in the previous example
- Batch learning
  - After processing a training vector, the error gradients accumulate in the weight corrections Δw
  - The weights are updated only once at the end of an epoch, after presenting all the training vectors: $w \leftarrow w + \Delta w$
  - Advantage: the results of the training no longer depend on the order in which the training vectors are presented

Methods to accelerate learning

- Sometimes networks learn faster when using the bipolar sigmoid function (hyperbolic tangent) instead of the unipolar sigmoid

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

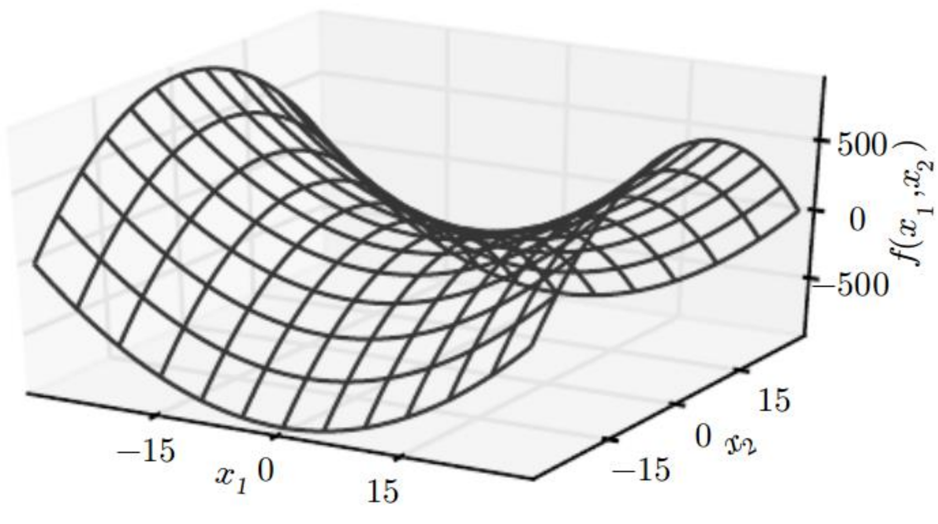
Momentum

- Generalized delta rule:

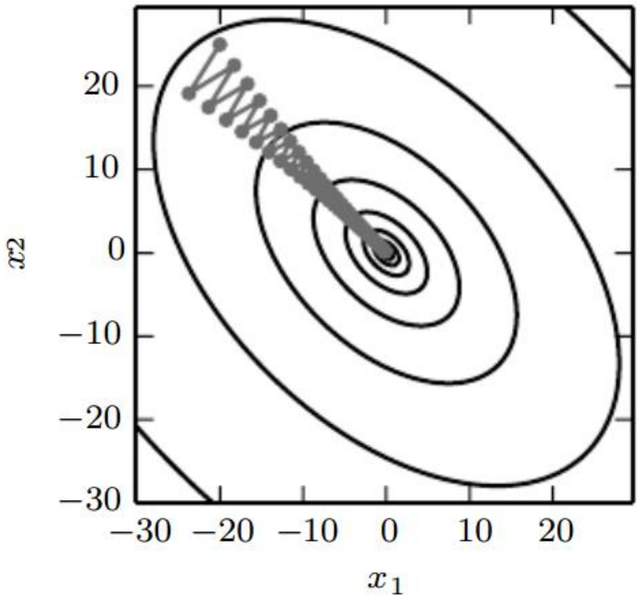
$$\Delta w_{jk}(p) = \beta \cdot \Delta w_{jk}(p-1) + \alpha \cdot y_j(p) \cdot \delta_k(p)$$

- The term β is a positive number ($0 \leq \beta < 1$) called momentum constant
- Typically, β is around 0.95

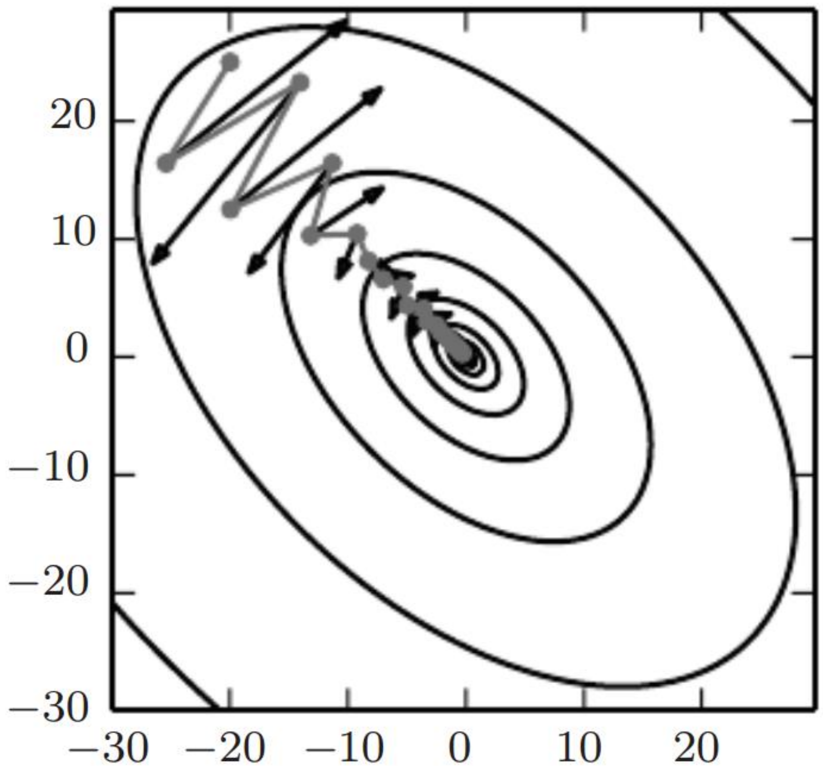
Example: Gradient Descent



$$f(\boldsymbol{x}) = x_1^2 - x_2^2$$

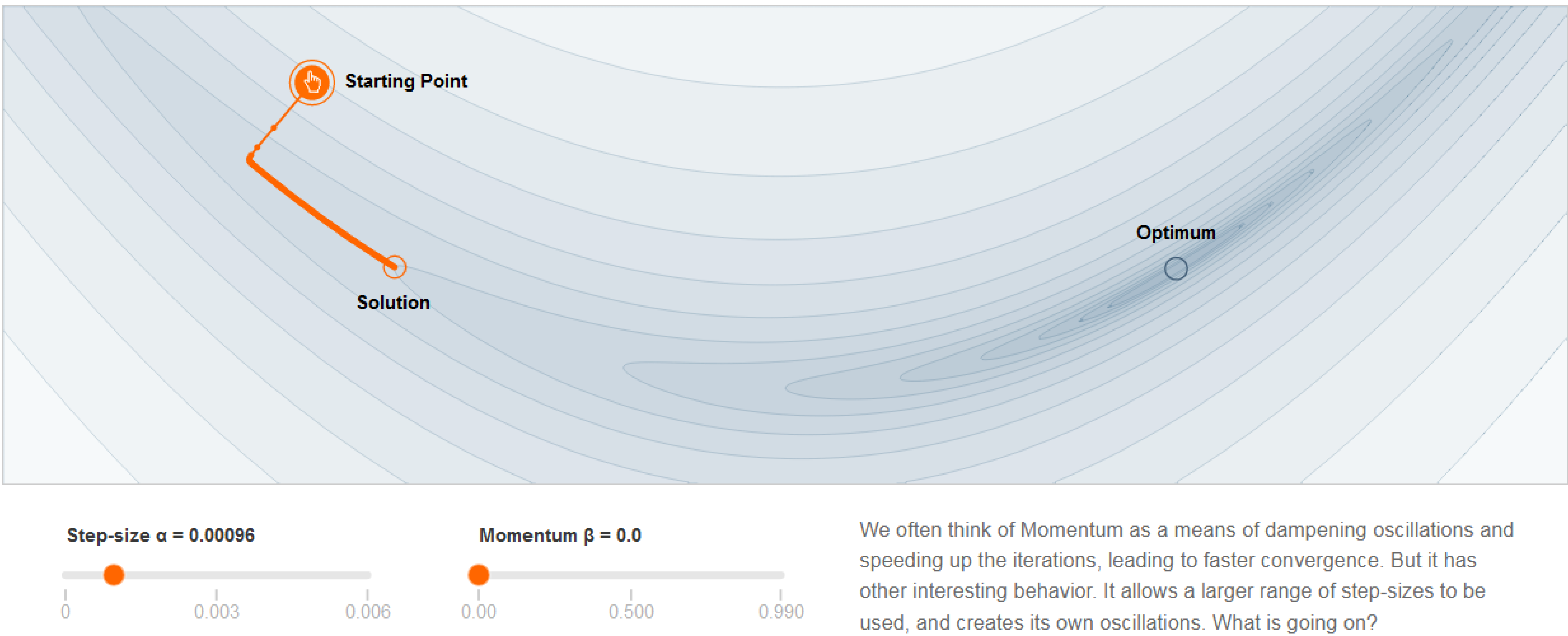


Example: GD with momentum



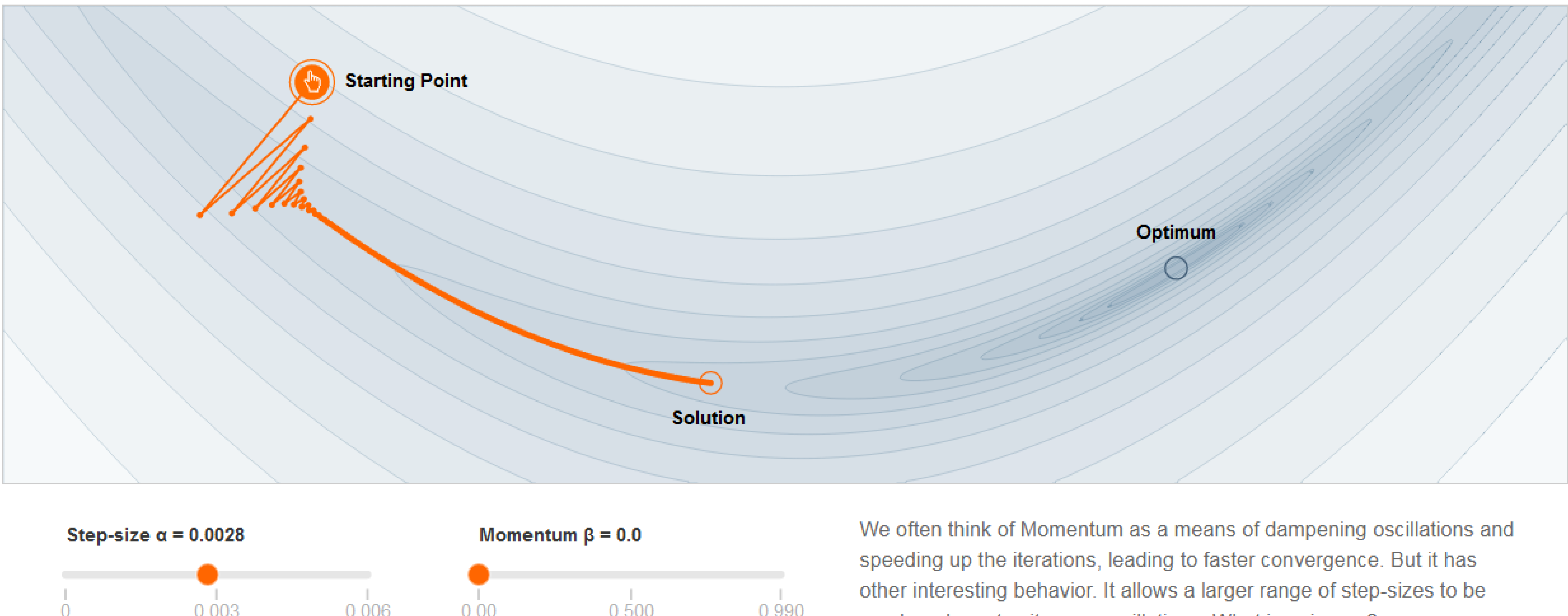
in grey: GD+M
in black: the step GD would do

Learning rate and momentum



Why Momentum Really Works, https://distill.pub/2017/momentum/

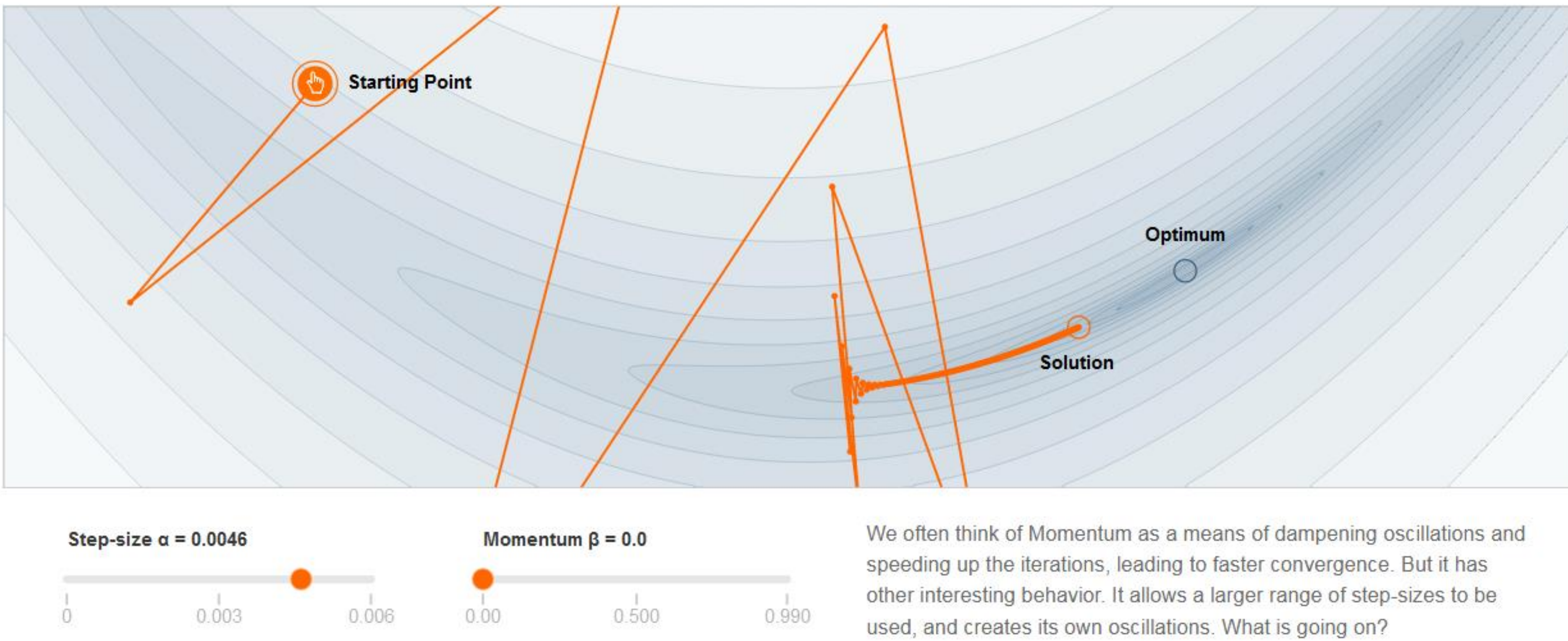
Learning rate and momentum



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

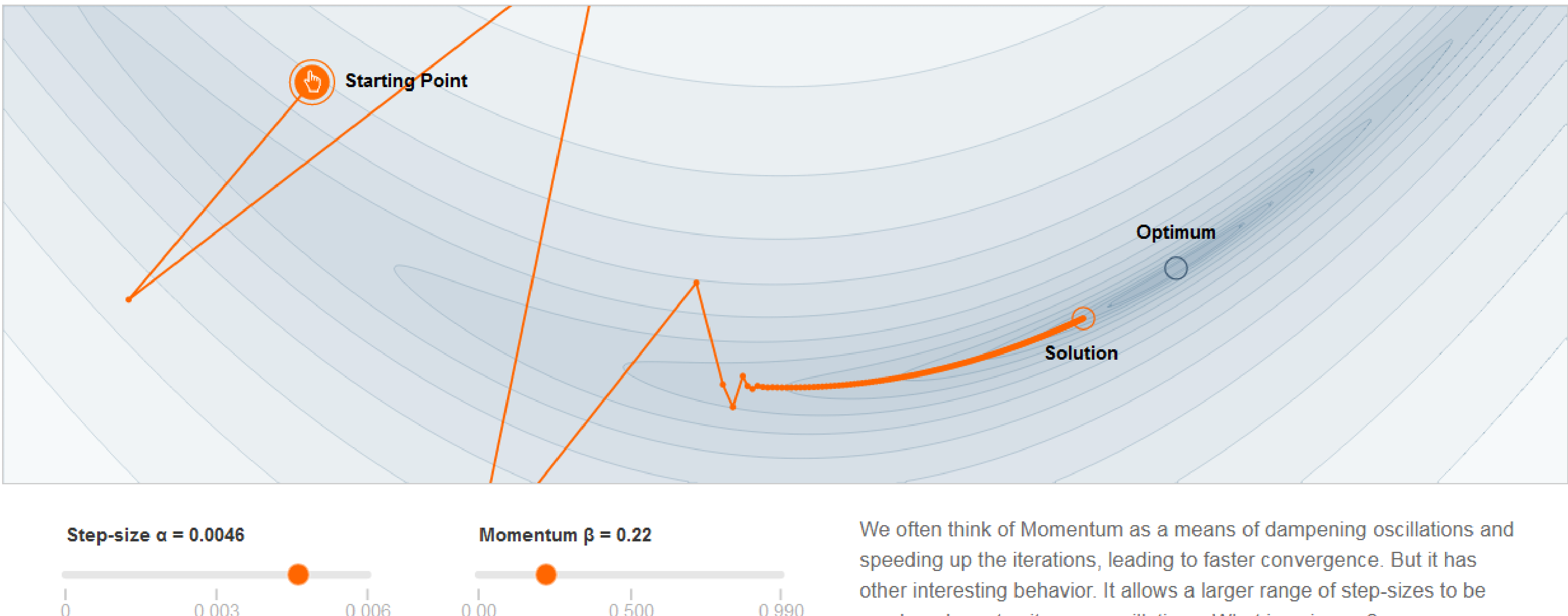
Why Momentum Really Works, https://distill.pub/2017/momentum/

Learning rate and momentum



Why Momentum Really Works, https://distill.pub/2017/momentum/

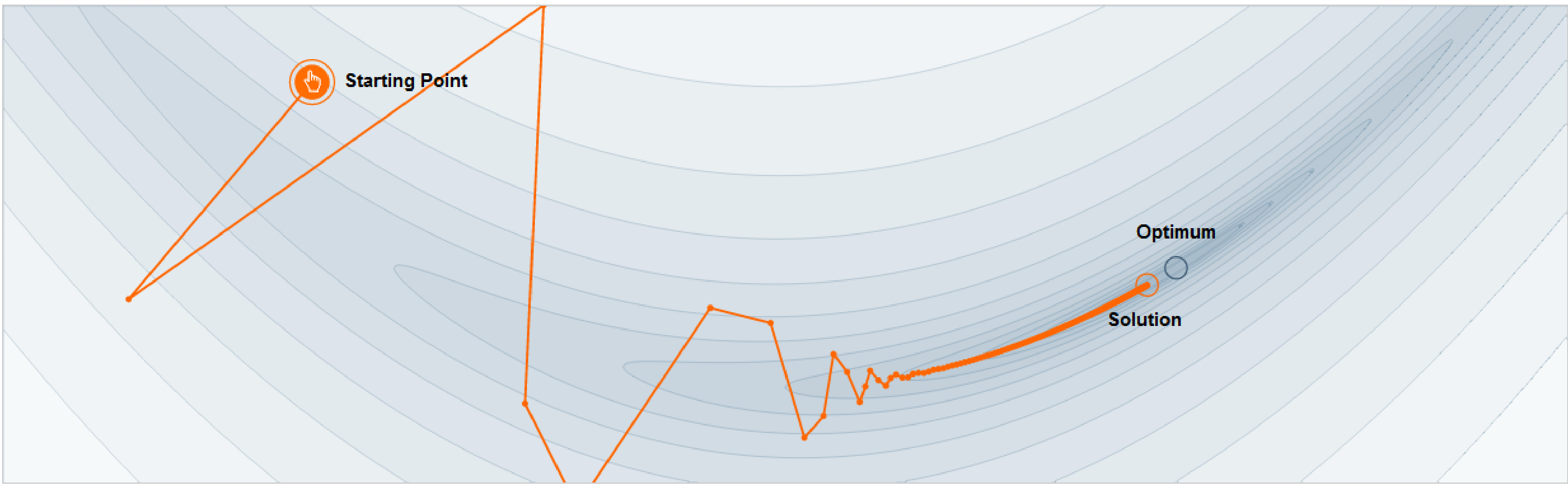
Learning rate and momentum



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, https://distill.pub/2017/momentum/

Learning rate and momentum



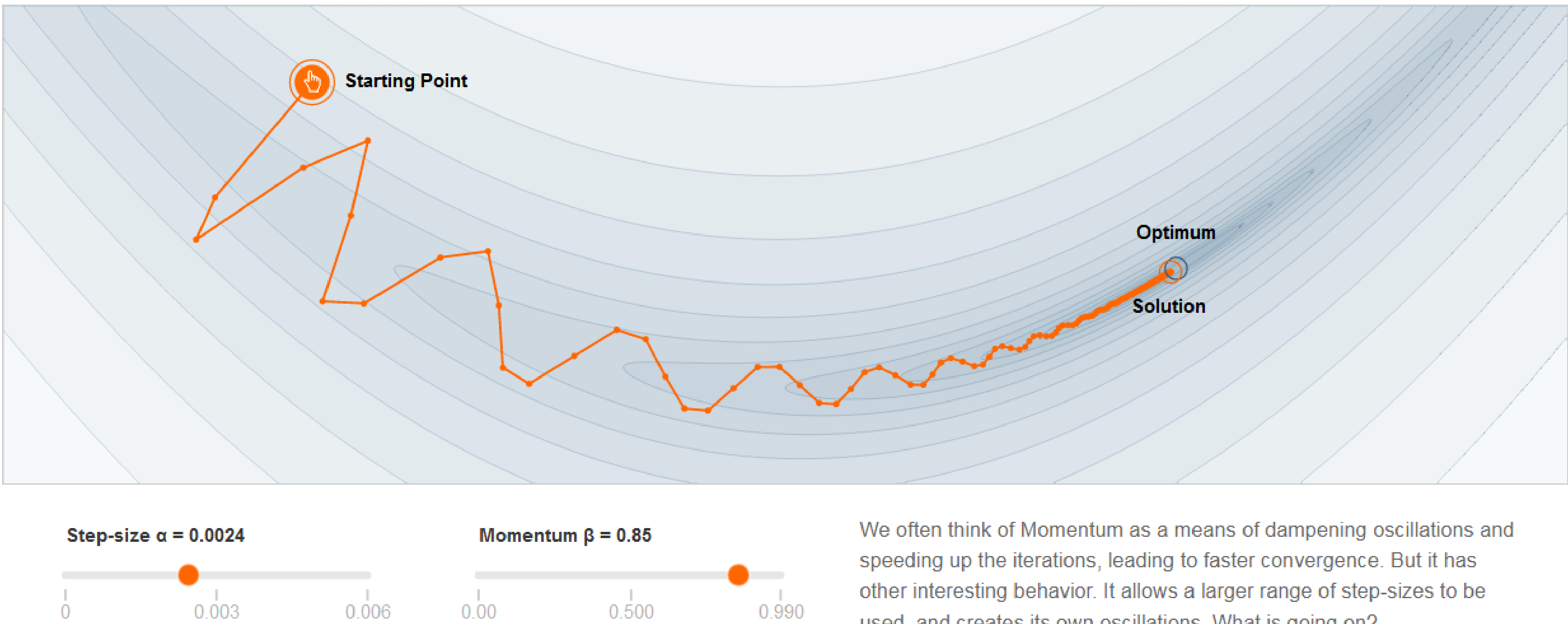
Step-size α = 0.0046 (0 – 0.003 – 0.006)

Momentum β = 0.55 (0.00 – 0.500 – 0.990)

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, https://distill.pub/2017/momentum/

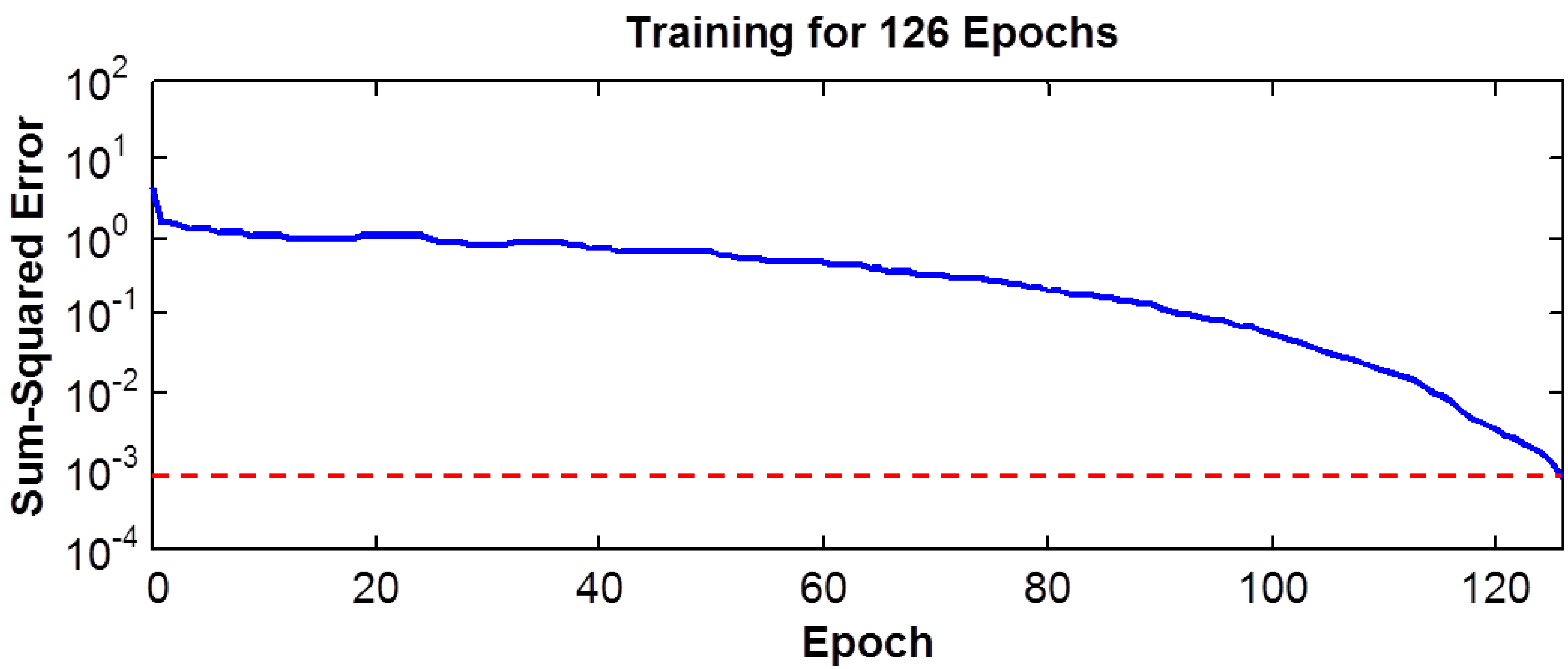
Learning rate and momentum



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Why Momentum Really Works, https://distill.pub/2017/momentum/

Learning with momentum for XOR



Adaptive learning rate

- In order to accelerate convergence and avoid instability, two heuristics can be applied:
  - If ΔE has the same algebraic sign for several consequent epochs, then the learning rate α should be increased
  - If the algebraic sign of ΔE alternates for several consequent epochs, then the learning rate α should be decreased

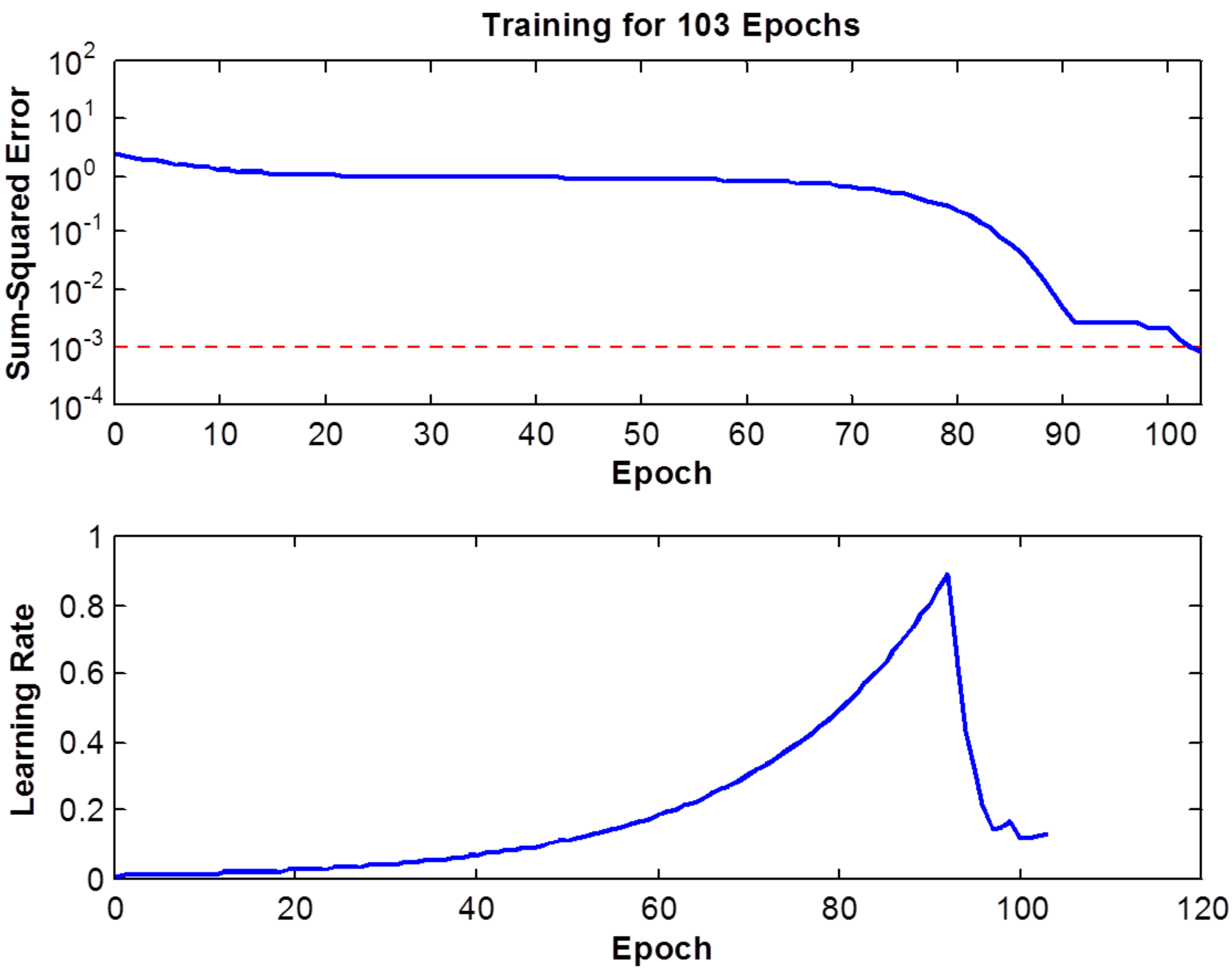
Adaptive learning rate

- Let E_p be the sum of the quadratic errors in the present epoch
- If $E_p > r \cdot E_{p\text{-}1}$, then $\alpha \leftarrow \alpha \cdot d$
- If $E_p < E_{p\text{-}1}$, then $\alpha \leftarrow \alpha \cdot u$
- Then the new weights are computed

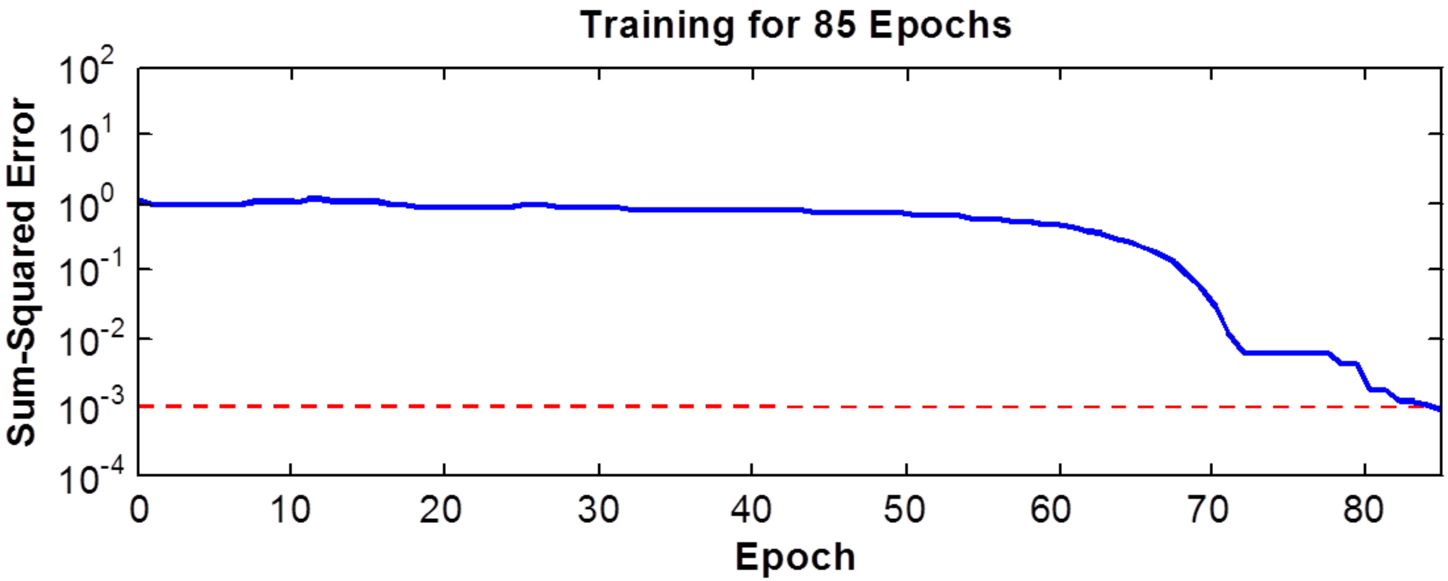
- Typical values: $r = 1.04$, $d = 0.7$, $u = 1.05$

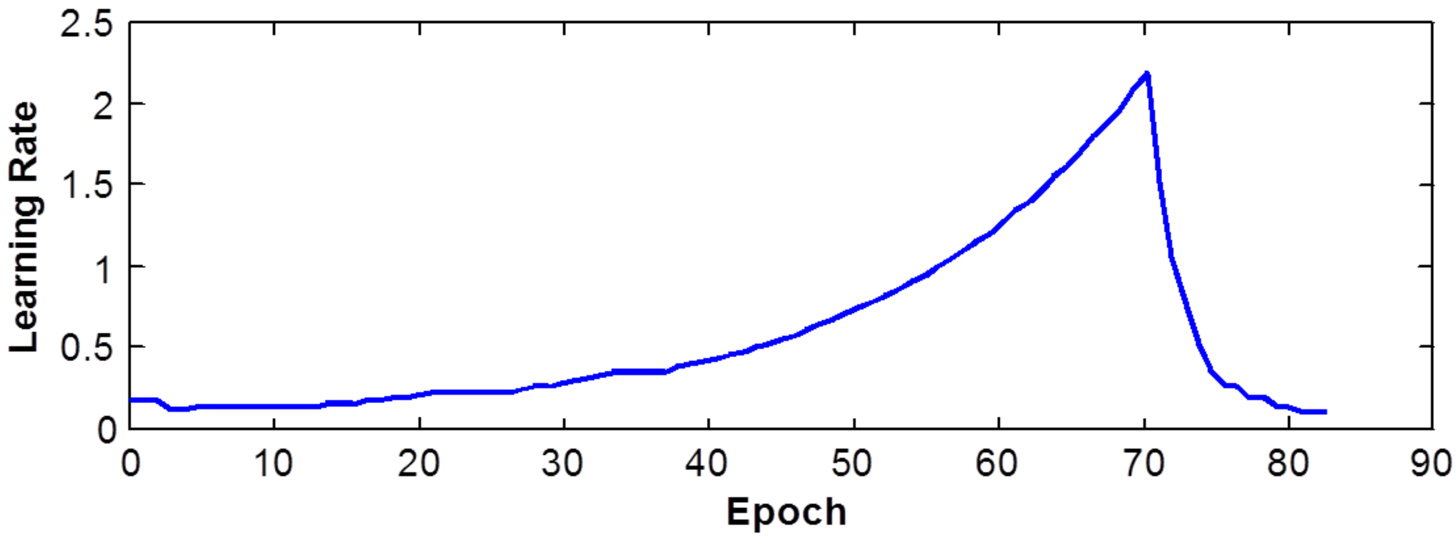
ratio down up

Learning with adaptive rate



Learning with momentum and adaptive rate





Practical issues

- In order not to saturate sigmoid functions, which would decrease the convergence speed, the inputs and outputs are usually scaled in the range [0.1, 0.9], or [-0.9, 0.9], depending on the type of sigmoid used
- If the network is used for regression and not for classification, the activation function of the output neurons can be linear or semi-linear (limited to 0 and 1, or -1 and 1, respectively) instead of sigmoid
- Establishing the number of layers and especially the number of neurons in each layer is relatively difficult and may require numerous attempts to achieve the desired performance

Heuristic: Kudrycki's rule

- $H = 3 \cdot O$ or
- $H_1 = 3 \cdot H_2$

- where:
  - O is the number of outputs
  - H is the number of neurons in the hidden layer (for a single hidden layer)
  - H_1 is the number of neurons in the first hidden layer and H_2 is number of neurons in the second hidden layer (for 2 hidden layers)

Neural Networks

1. Introduction
2. The Perceptron. Adaline
3. The Multilayer Perceptron
4. Deep Networks
5. Conclusions

Classic networks and deep networks

- Classic networks
  - 1-2 layers
  - Sigmoid activation functions
  - Cost functions based on the MSE
  - Training algorithms: Backpropagation, RProp, Levenberg-Marquardt etc.
- Deep networks
  - More layers
  - Simpler activation functions: ReLU
  - Cost functions based on the MLE
  - Training algorithms: SGD, RMSProp, Adam, etc.
  - Other methods of weight initialization, regularization, pre-training
- Apart from the number of layers, the differences are not strict!

Unstable gradients

- The first layers of a classical multilayer perceptron with a larger number of hidden layers do not train well
- When the weights are small or the sigmoid activation functions are saturated (small gradients), the weight corrections Δw are small and the training is very slow
- Also, the last layers, those close to the output, can learn the problem “quite well” and so the error signal sent to the first layers becomes even smaller
- Other training methods are needed to take advantage of the first layers

ReLU activation function

- *ReLU (Rectified Linear Unit)*

$$f(x) = \max(0, x)$$

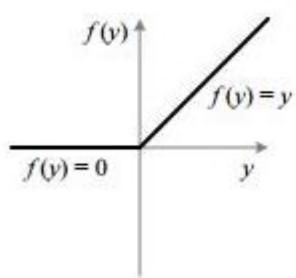
- *Leaky ReLU*

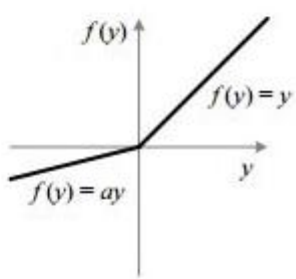
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

- *Parametric ReLU (PReLU)*

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

a can be learned





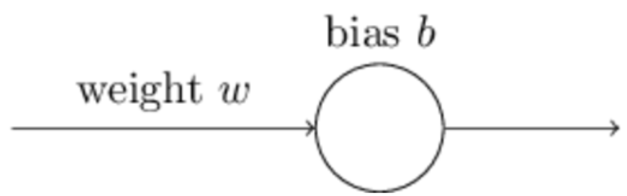
ReLU activation function

- ReLU is the activation function recommended in many cases for deep networks
- It is nonlinear
- It is easy to optimize by differential methods
  - Even if the gradient is 0 on the negative side, in practice it works well
  - It has been observed that learning is about 6 times faster than with sigmoid functions

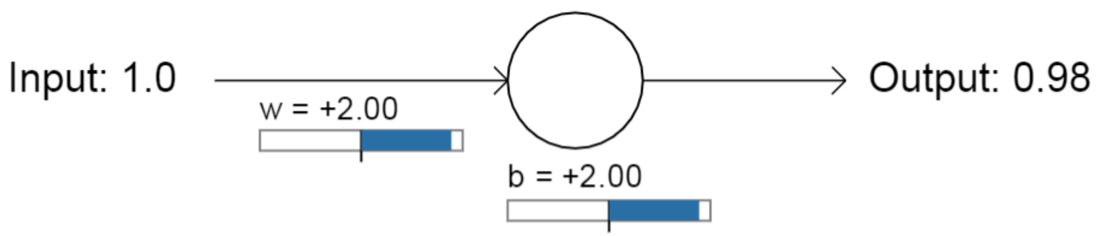
$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$

The cost function

- Example: a neuron that receives input 1 and should output 0

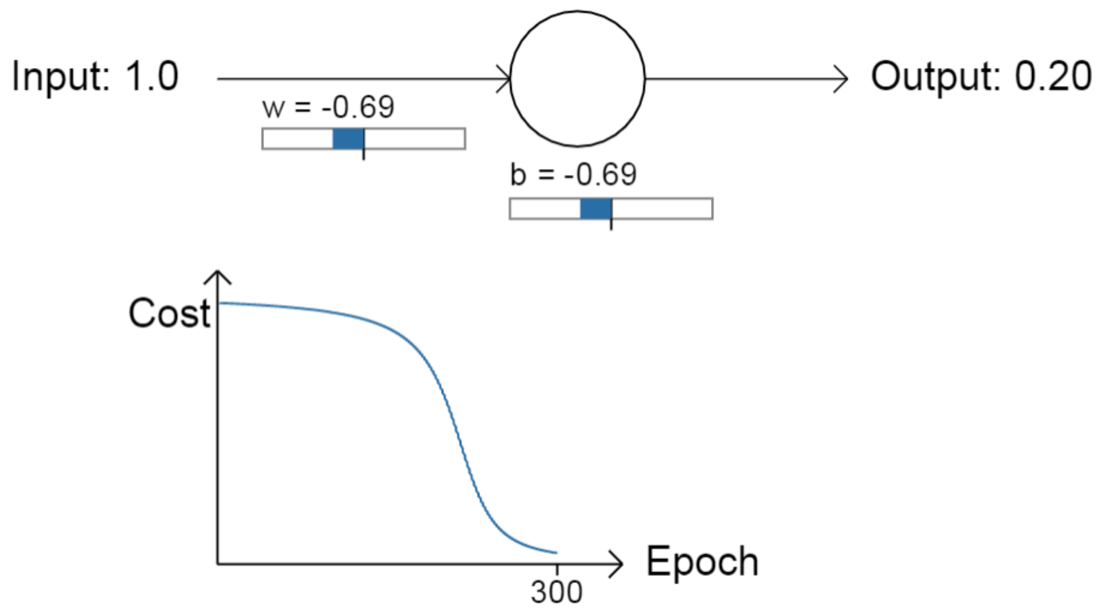


- The initial state



Learning

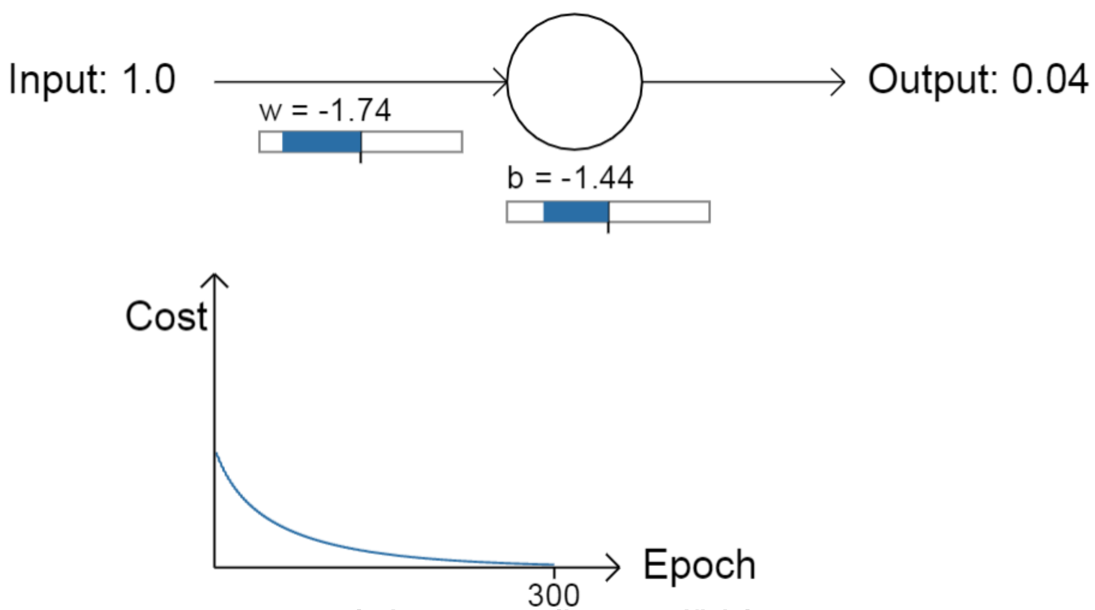
- With the MSE cost function



Learning

- With the *cross-entropy* cost function

$$w_j = w_j + \alpha \sum_{i=1}^{n} x_{ij} \left(y_i - p_i \right)$$



Stochastic Gradient Descent

- For current problems, the number of training instances can be very large
- SGD works similar to gradient descent, but does not take into account all instances when calculating gradients, but only a minibatch
- At each iteration, other instances are randomly chosen
- The gradients are just an approximation of the real gradients
- The time complexity can be controlled by the size of the minibatch

Other training algorithms

- RMSProp (Root Mean Square Propagation)
- Adam (Adaptive Moment)
- RMSProp with Nesterov momentum
- AdaDelta
- AdaGrad
- AdaMax
- Nadam (Nesterov-Adam)

Conclusions

- Rosenblatt's perceptron can learn linearly separable functions
- The multilayer perceptron can approximate nonlinearly separable functions, with complex decision boundaries
- The most used training algorithm for the multilayer perceptron is the backpropagation algorithm
- Deep networks, beside a greater number of layers, may have different activation functions and training algorithms than traditional networks, in addition to other optimizations