Artificial Intelligence

Neural Networks

POLITÉCNICO DE LEIRIA

Slides by Carlos Grilo & Rolando Miragaia

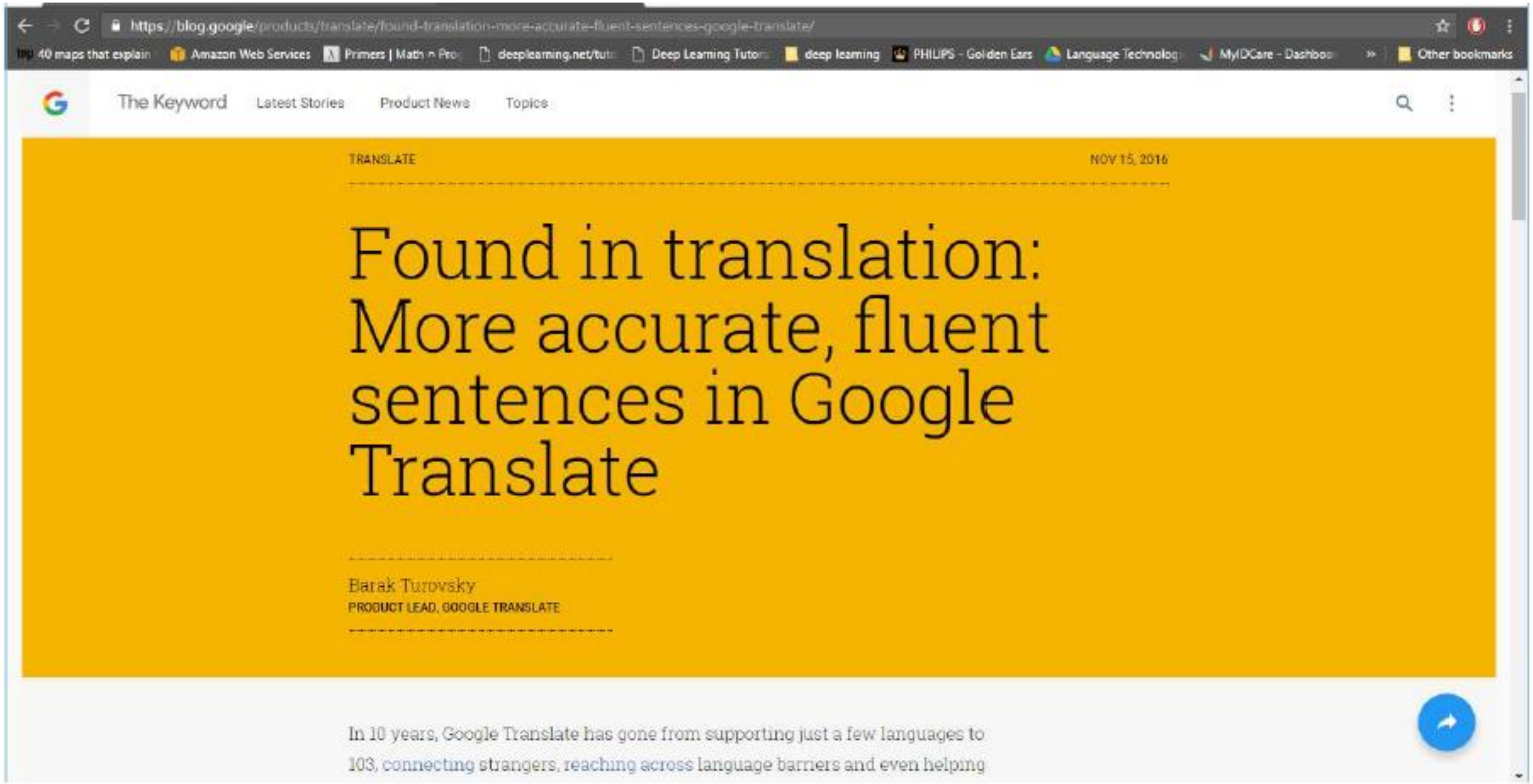
Neural networks are taking over

- Neural networks have become one of the most thrust areas recently in various pattern recognition, prediction, and analysis areas

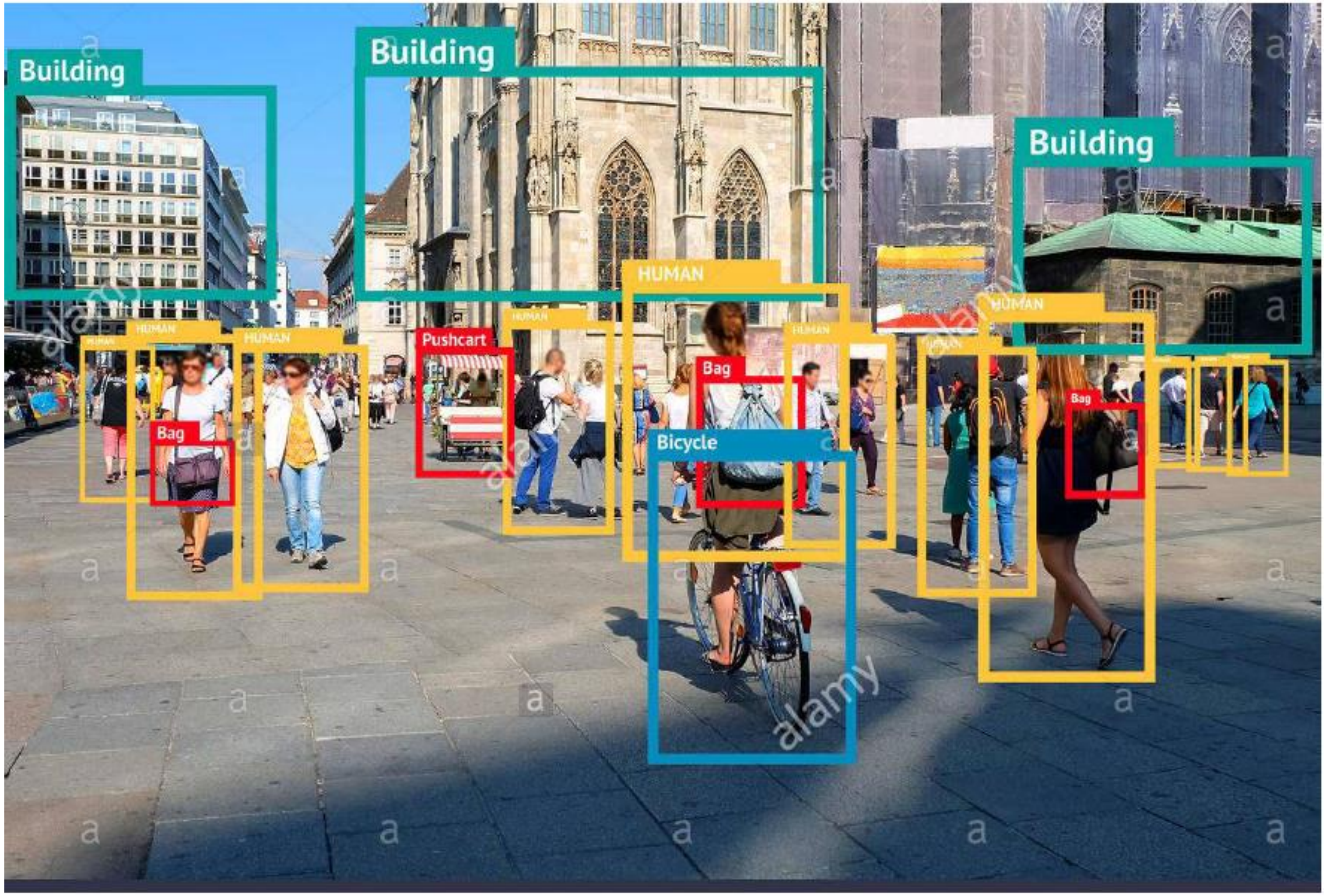
- In many problems, they have established the state of the art, often exceeding previous benchmarks by large margins

Breakthrough successes with neural networks

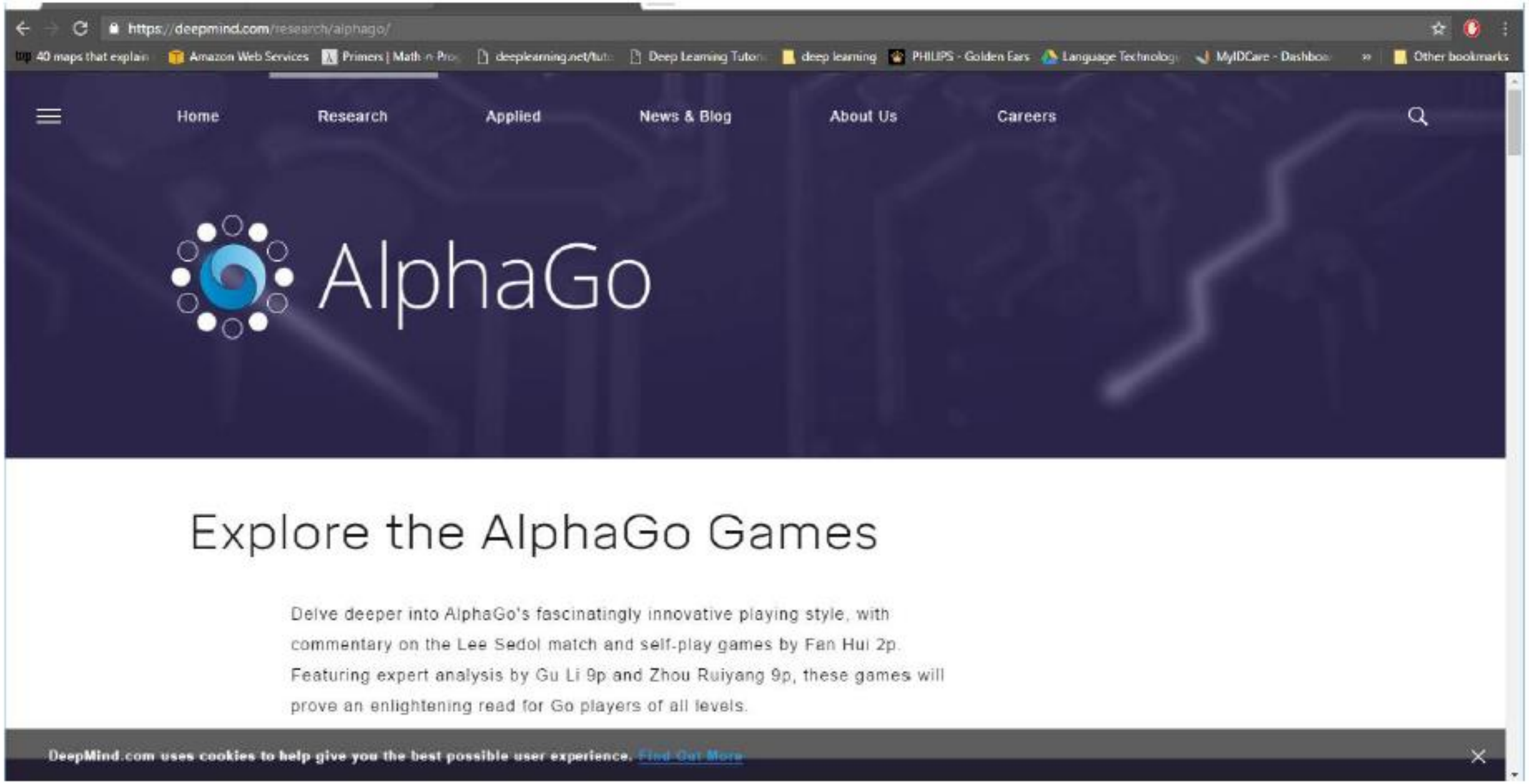
Breakthrough successes with neural networks



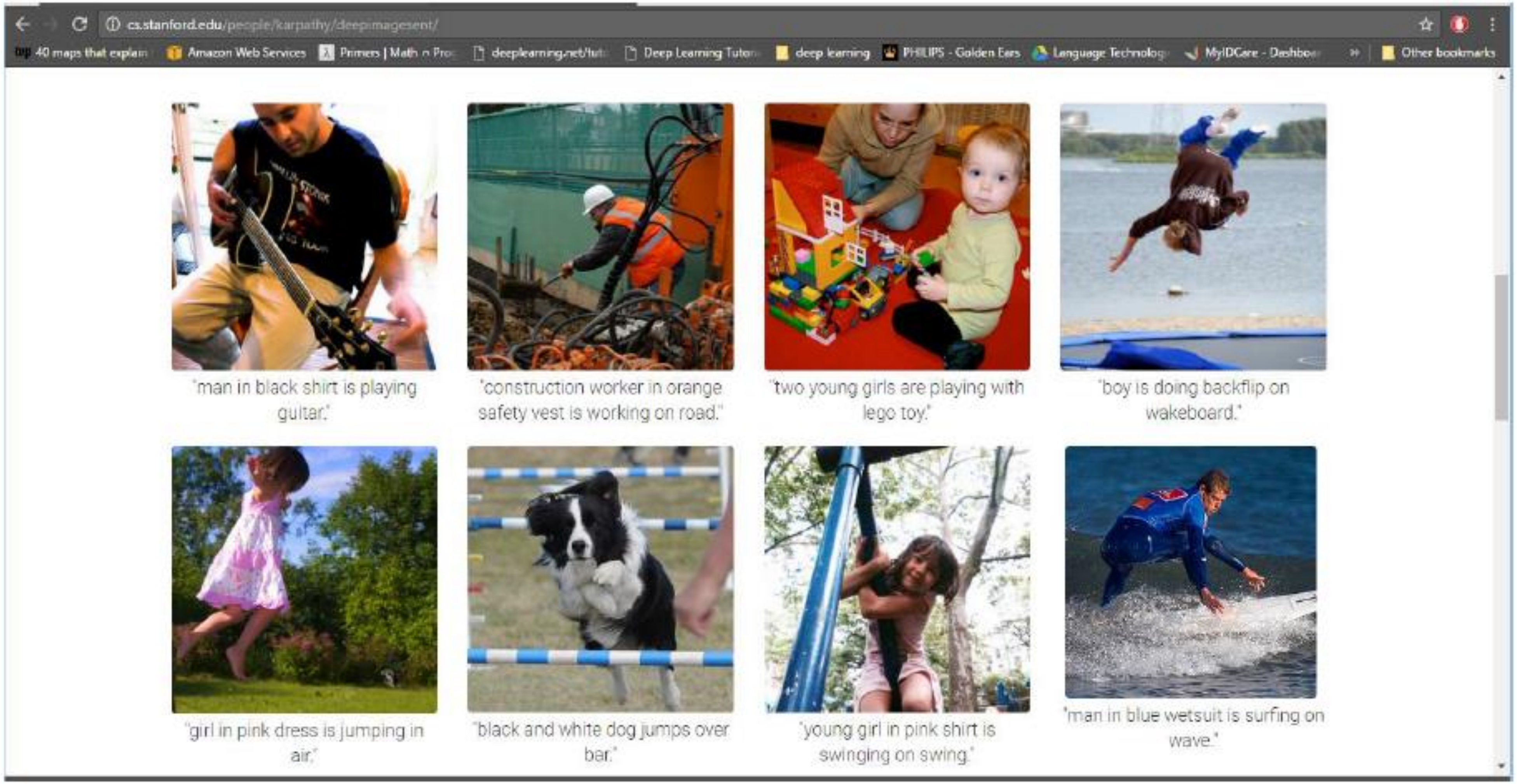
Breakthrough successes with neural networks



Breakthrough successes with neural networks



Breakthrough successes with neural networks



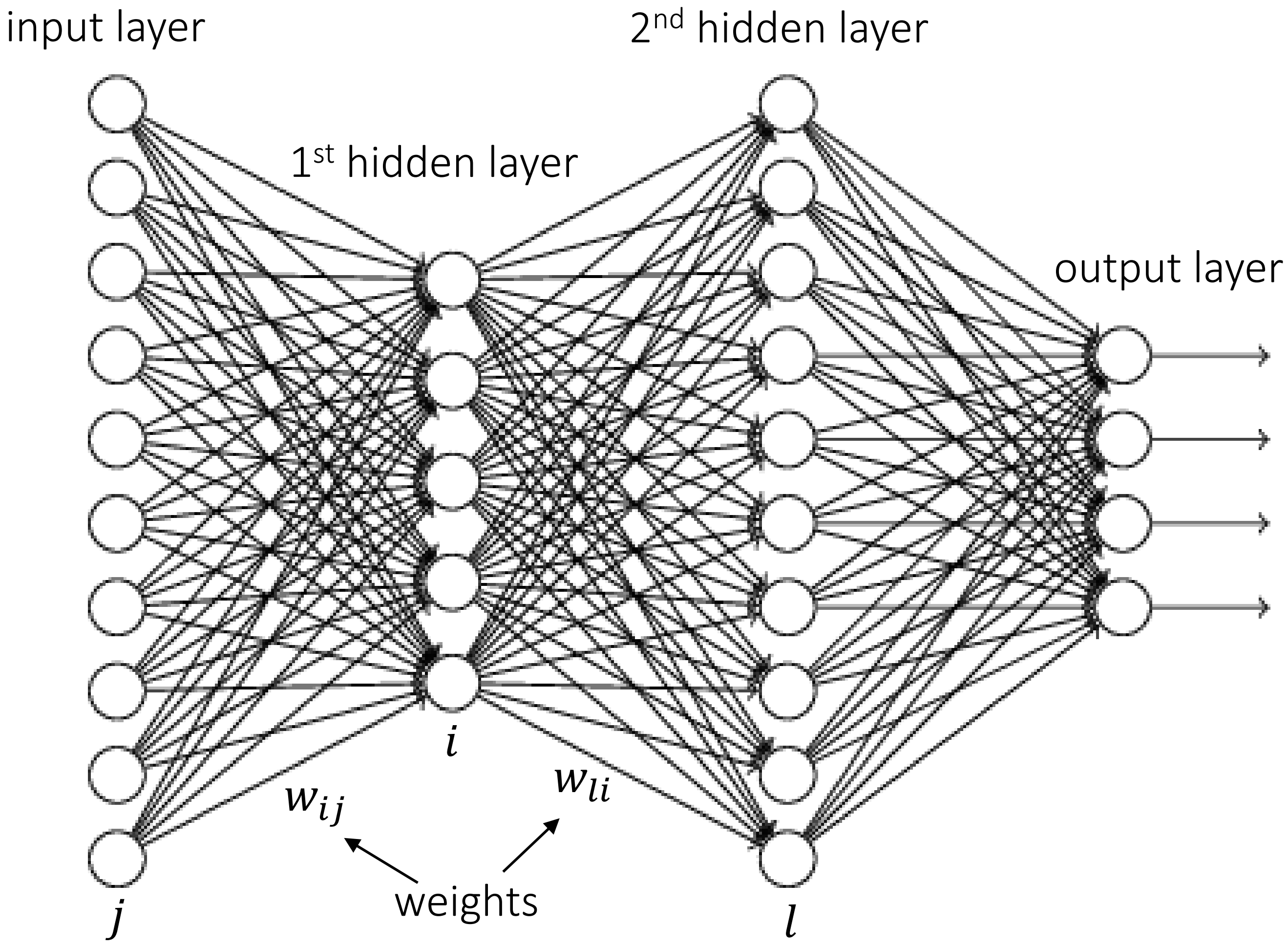
(Artificial) Neural networks

- They are information processing systems inspired on our understanding of biologic neural networks

- A NN consists in an interconnected set of simple processing units called neurons whose functionality is vaguely similar to that of a biological neuron

- Neurons communicate between them by sending signals through a large number of connections and information processing occurs as if it occurred simultaneously in several neurons

Neural networks



- Each neuron has a set of input connections and a set of output connections
- There are some neurons connected to the “outside world” (some for input, others for output)
- Each connection has a weight associated to it. Weights are the main means of information storage of a network
- NNs learn by modifying their weights

Learning by examples

- Neural networks learn through examples (inductive learning):
  - They use a large number of examples of what should be learned
  - Using these examples, they build a model (function) of those examples
  - The more examples and the more representative those examples are, the better the model

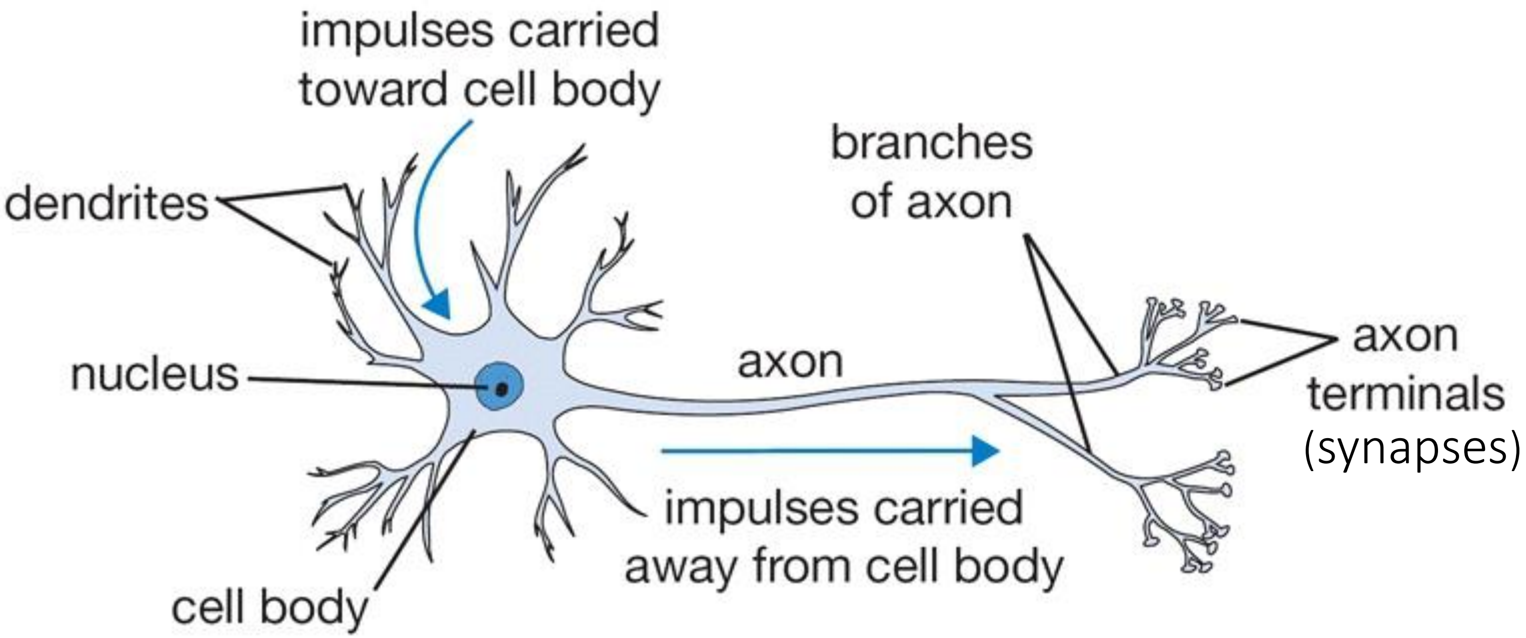
Dataset: set of examples used to train the neural network

Sample: designation often used to name an example of the dataset

The brain and the biological neuron

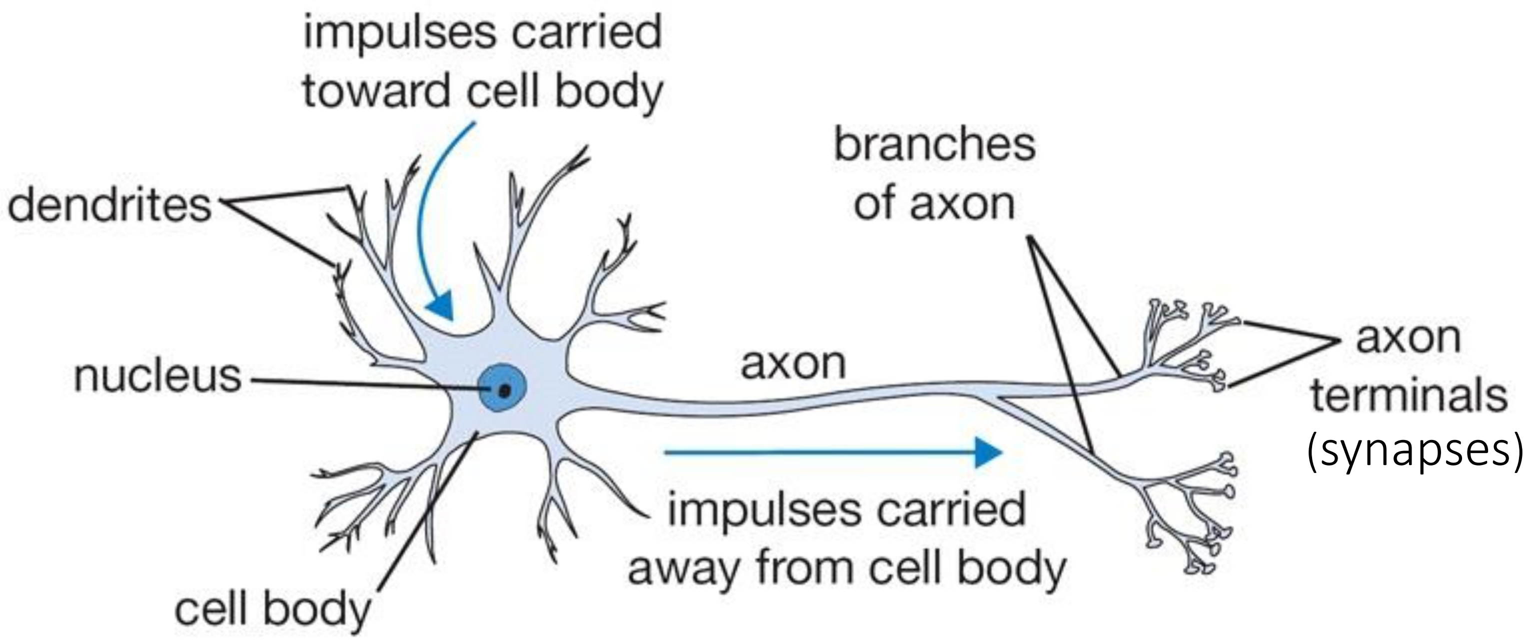
The brain

- Neurons are the fundamental unit of the nervous system tissue



The biological neuron

- Each neuron is composed of a cellular body with several branches called dendrites and an isolated longer branch, called axon



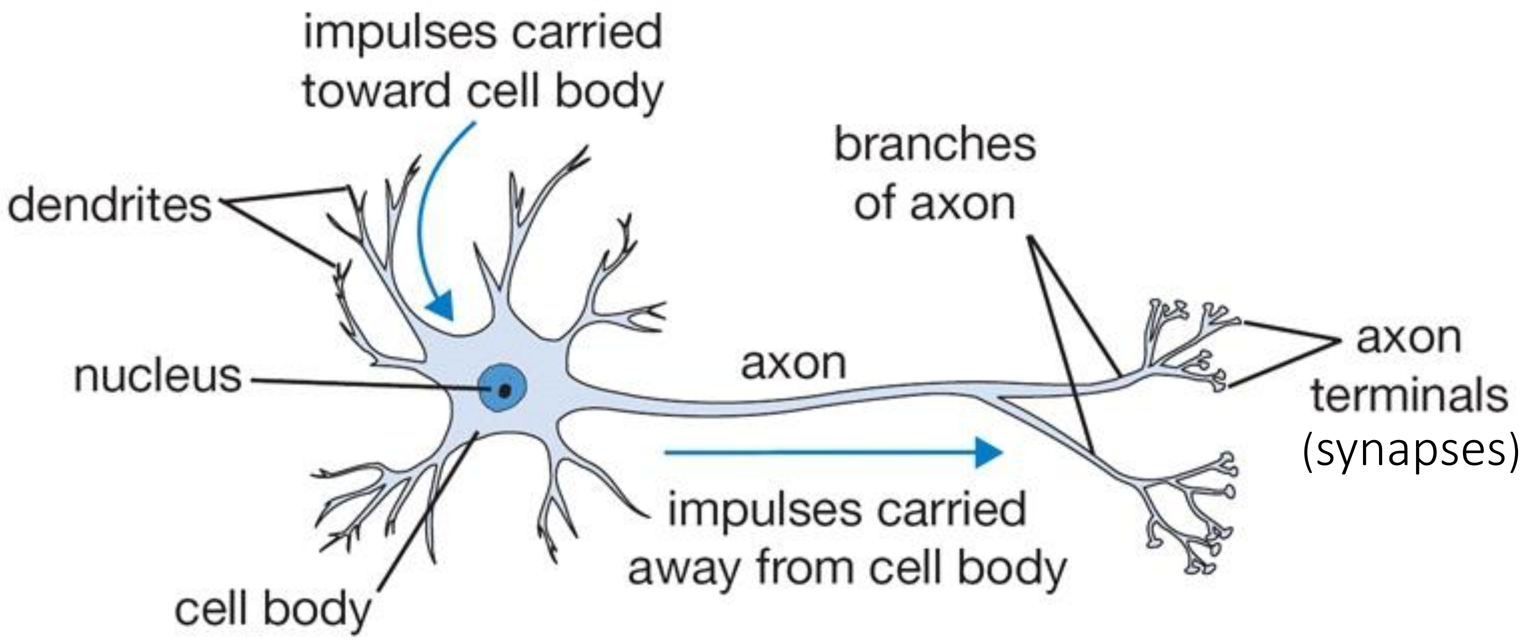
Dendrites connect to other neurons' axons through junctions called synapses

We can view dentrites as the input branches and the axon as the output branch, which ramifies in several branches

A neuron can be connected to hundreds of thousands of other neurons

The biological neuron

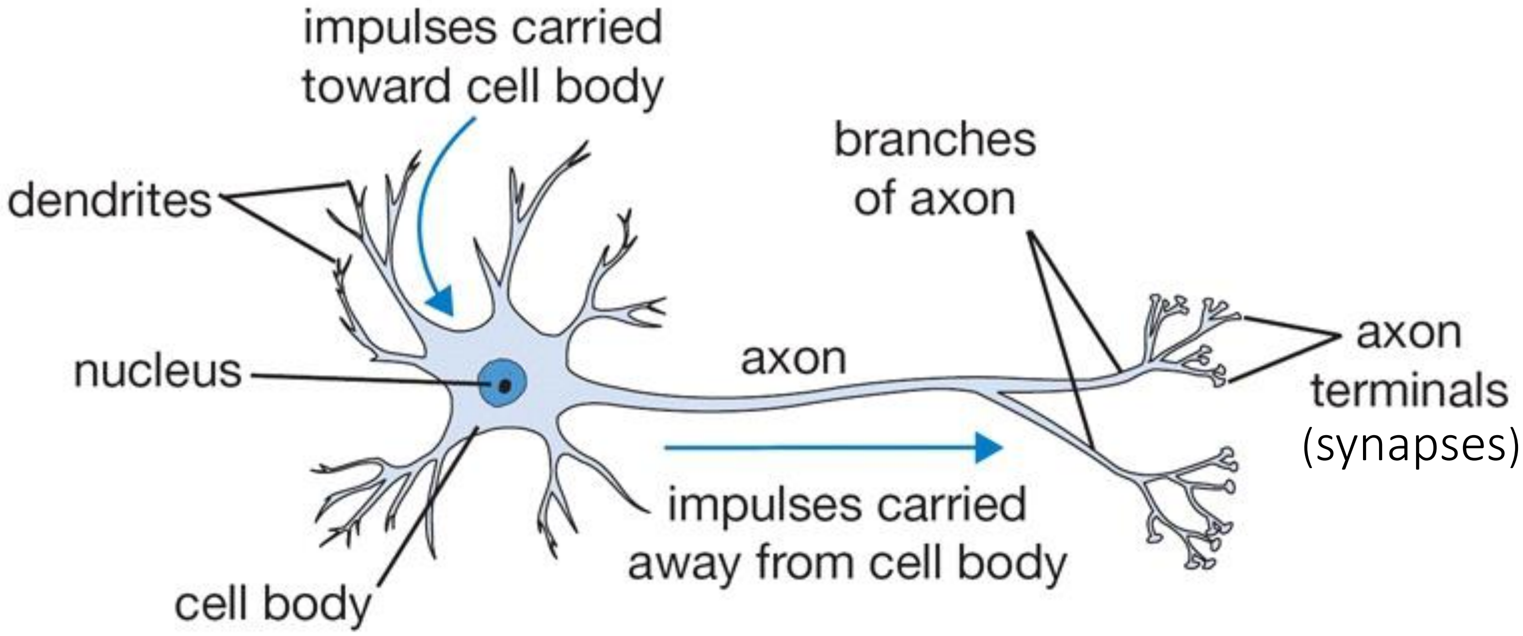
- Signals propagate between neurons through a complicated electrochemical reaction which leads synapses to produce chemical substances that enter through dendrites



This can raise or diminish the electrical potential of the cellular body

The biological neuron

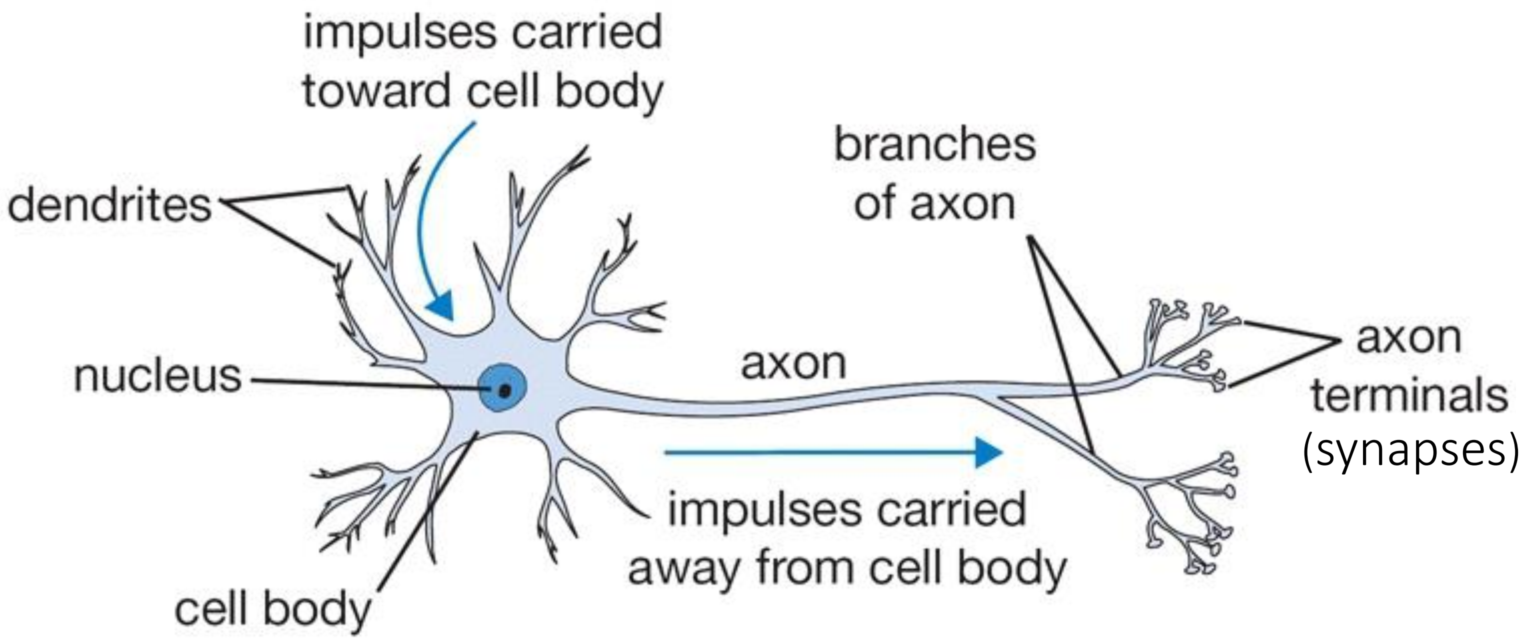
- If the electrical potential overpasses some limit, an electrical impulse is sent to the axon, which spreads by its ramifications, thus transmitting electrical signals to other neurons



(synapses)

The biological neuron

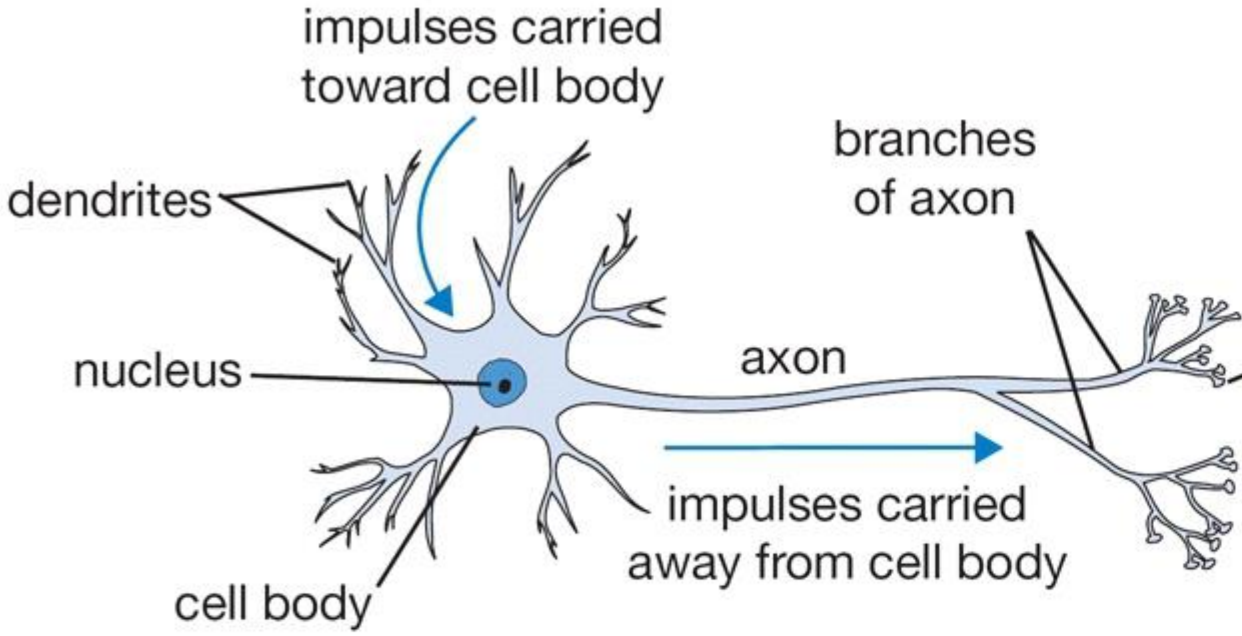
- It has been observed that often used connections become stronger and that neurons sometimes form new connections with other neurons



It is thought that these mechanisms lead to learning

The brain

- An average brain has something on the order of 100 billion neurons. Each neuron is connected to up to 10,000 other neurons, which means that the number of synapses is between 100 trillion and 1,000 trillion



(synapses)

Computation is strongly parallel and asynchronous

The artificial neuron

The artificial neuron

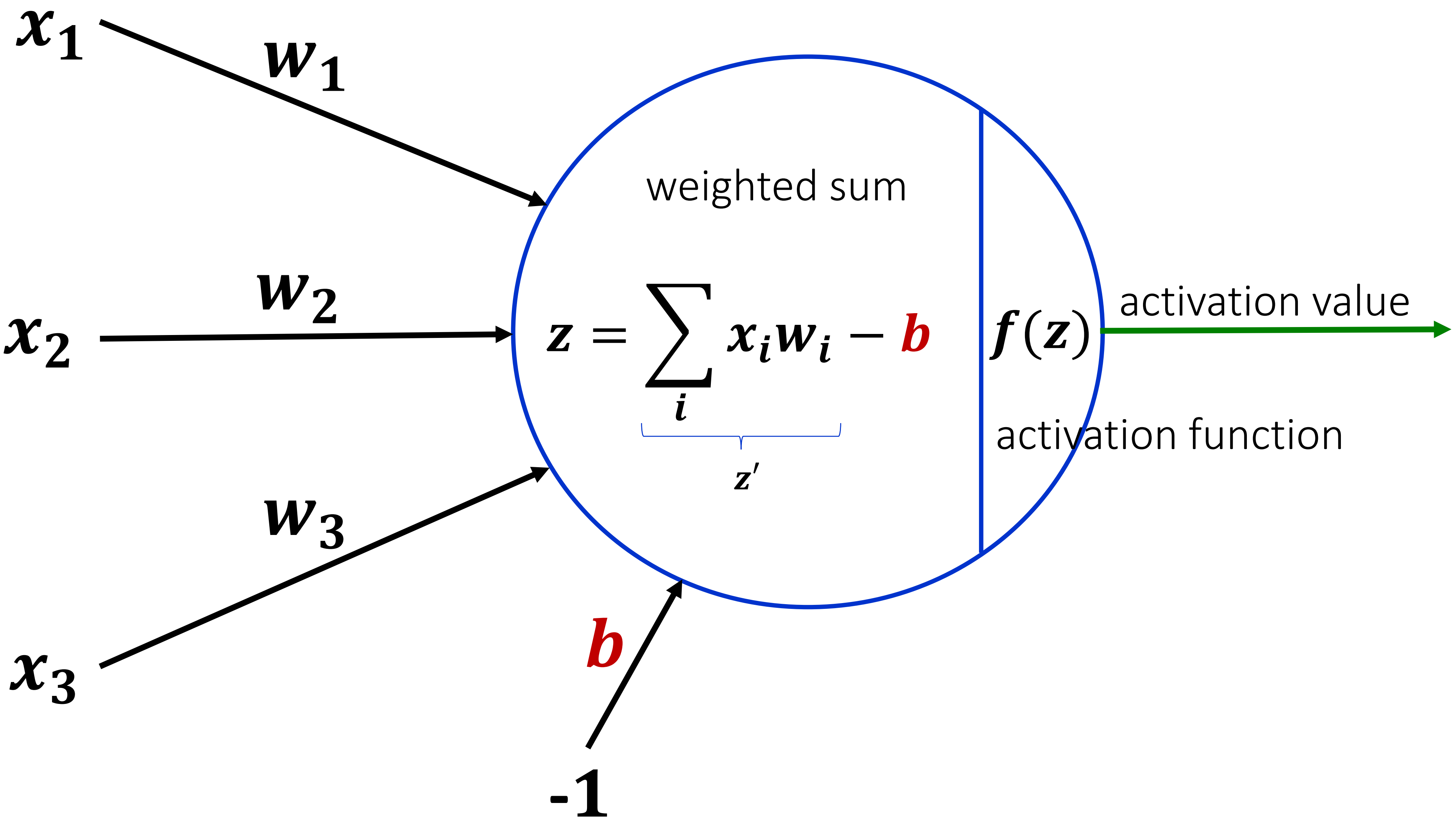
- The computation in each neuron is simple: it receives signals from the input connections and computes a new activation value which is sent through its output connections

- The computation of this value is done in two phases:

  1. First, a weighted sum of the inputs is computed

  2. Then, the neuron’s activation value is computed using a so called activation function which has as input the value computed in the first phase

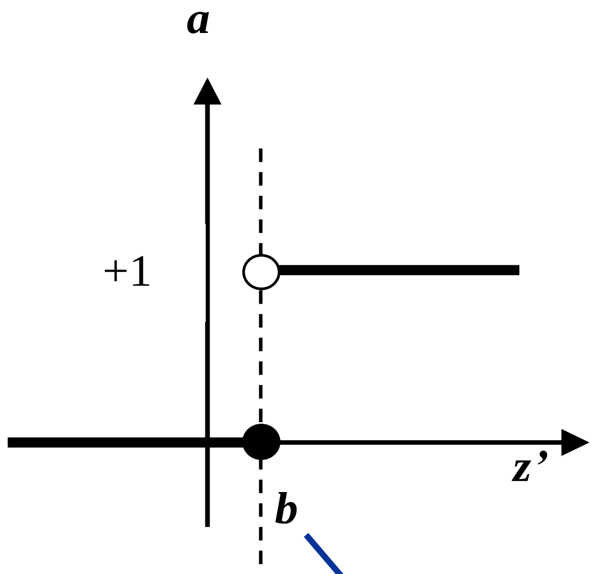
The artificial neuron



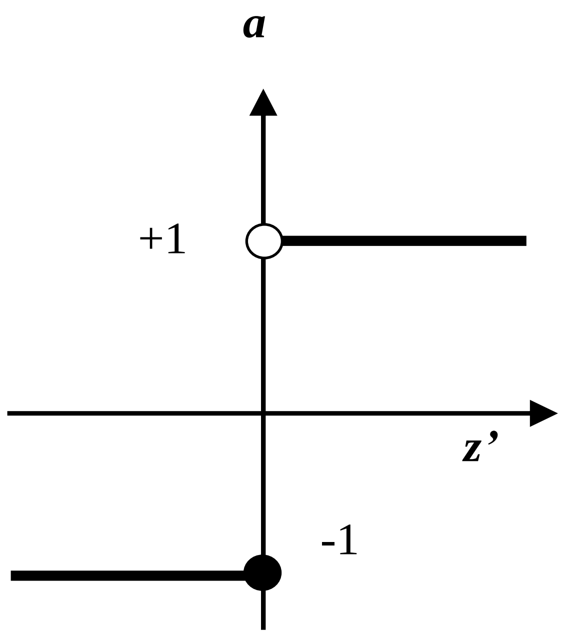
The artificial neuron

- Each connection has a weight associated
- Weights express the importance of the respective inputs to the output
- There is an especial input with value -1, with a weight called *offset* or *bias* represented as *b*
- The role of *b*'s value will be illustrated in the next slides

Activation functions



a) step function



b) signal function

The bias helps controlling the value at which the activation function will trigger

Neuron example

- Let us consider a neuron with two inputs and with the step activation function. So,

$$a = \begin{cases} 1, x_1 \times w_1 + x_2 \times w_2 \; - b \; > 0 \\ 0, \qquad otherwise \end{cases}$$

$$a = \begin{cases} 1, x_1 \times w_1 + x_2 \times w_2 \; > b \\ 0, \qquad otherwise \end{cases}$$

- This means that the neuron's activation value (output) will be 1 if the weighted sum of the inputs is larger than *b* and 0, otherwise

- *Therefore,* *b* represents the minimum value that the weighted sum of the inputs (x_1 and x_2) must have so that the neuron's activation value can be 1

- *b* can be used with activation functions, others than the step function

Vector notation

- It is common to use vectors notation to represent the sets of inputs and weights

- Using this notation, we can say

$$a = \begin{cases} 1, x \cdot w + b \ > 0 \\ 0, otherwise \end{cases}$$

Example:
[0, 1] · *[2, 3]* + *4* = *7*
[x_1, x_2] · [w_1, w_2] + b

where x represents the inputs vector and w the weights vector

Dot product

- The dot product of two vectors a = $[a_1, a_2, \ldots, a_n]$ and b = $[b_1, b_2, \ldots, b_n]$ is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

- Example:

$$\begin{aligned}[1, 3, -5] \cdot [4, -2, -1] &= (1 \times 4) + (3 \times -2) + (-5 \times -1) \\ &= 5 - 6 + 5 \\ &= 3\end{aligned}$$

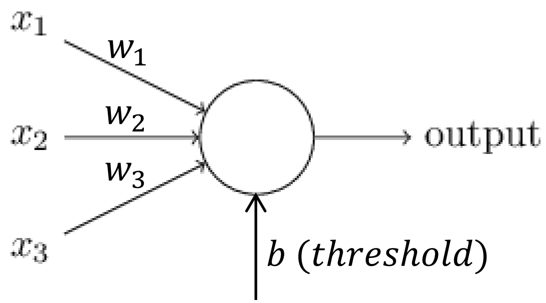
Neurons, weights, thresholds and decisions

- We can think about neurons as devices that make decisions by weighing up the inputs

- If the weighted sum on the inputs is larger than some threshold, one decision is taken, if not, another decision is taken (often, the opposite decision)

Example

- Example: to go or not to go to a concert (output)
  - Good weather (x_1)?
  - Girlfriend/boyfriend wants to go (x_2)?
  - Public transport (x_3)?



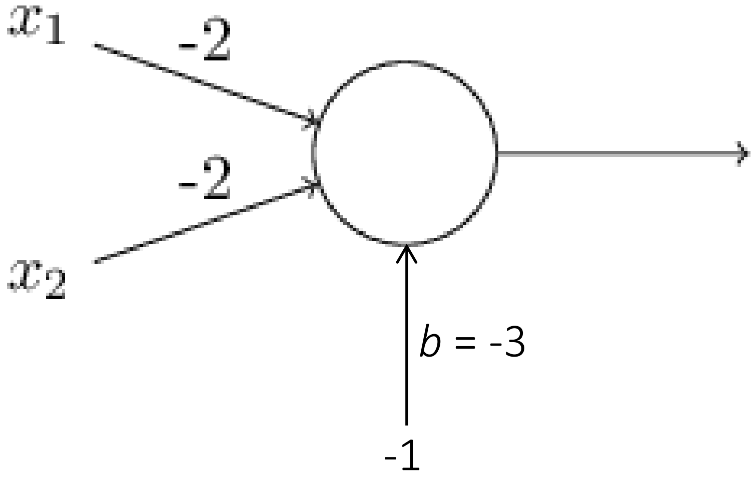
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq b \\ 1 & \text{if } \sum_j w_j x_j > b \end{cases}$$

Weights express the importance of each criterium (input) on our decision

- By varying the weights and the bias (threshold), we get different models of decision-making

Another example

- The following neuron implements a NAND gate ($\overline{x_1 \land x_2}$):



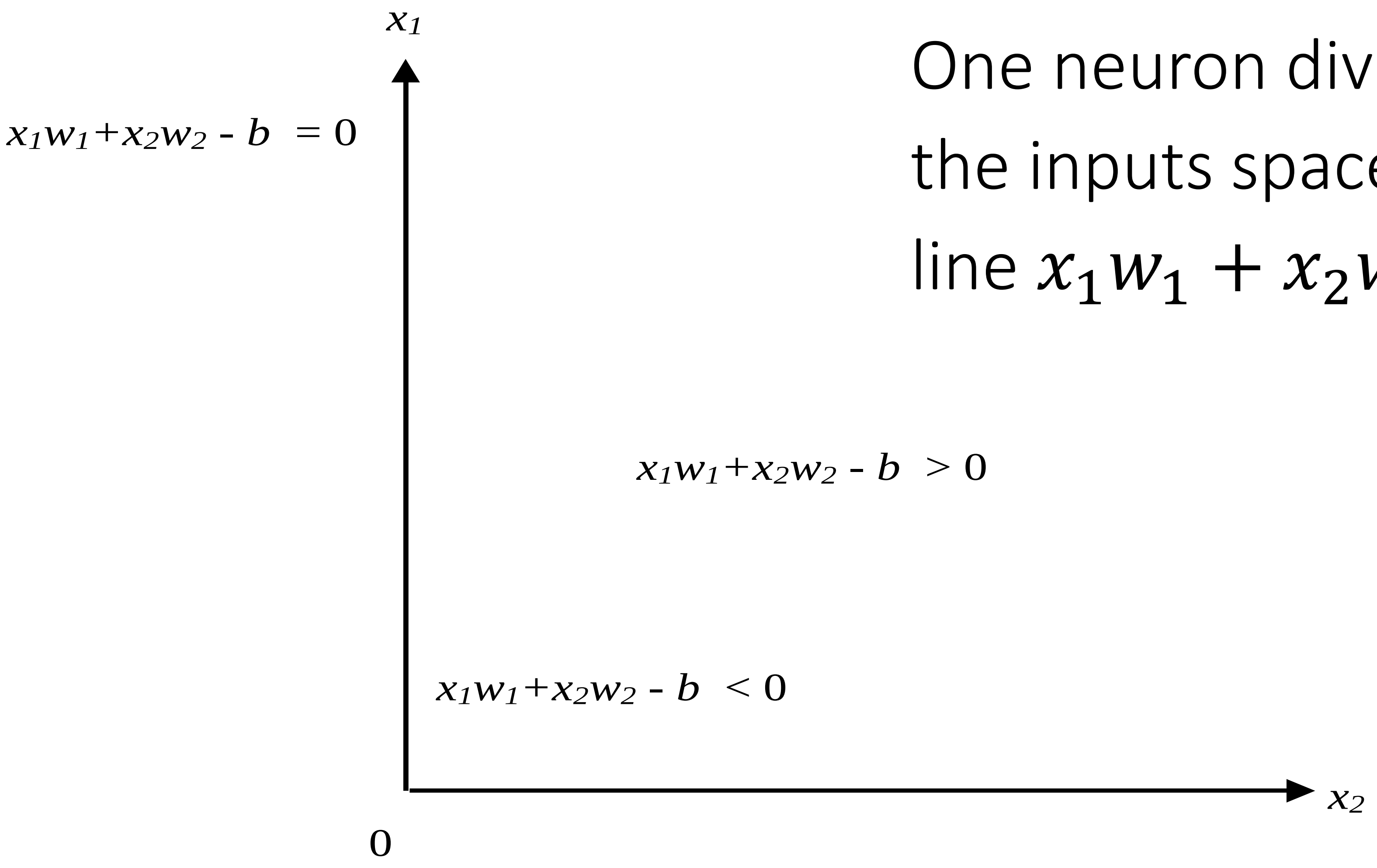
Assume:
- 0 (F) and 1 (T) inputs
- Step activation function

x_1	x_2	$x_1 \land x_2$	$\overline{x_1 \land x_2}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

- Exercise: confirm this

Limitations of a single neuron

One neuron divides in two parts the inputs space with the straight line $x_1w_1 + x_2w_2 - b = 0$



Limitations of a single neuron

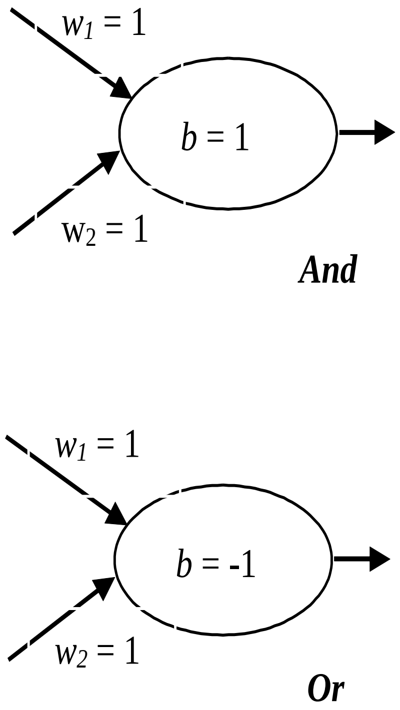
- In two dimensions (two inputs) the division is done by a straight line and, in three dimensions, by a plane

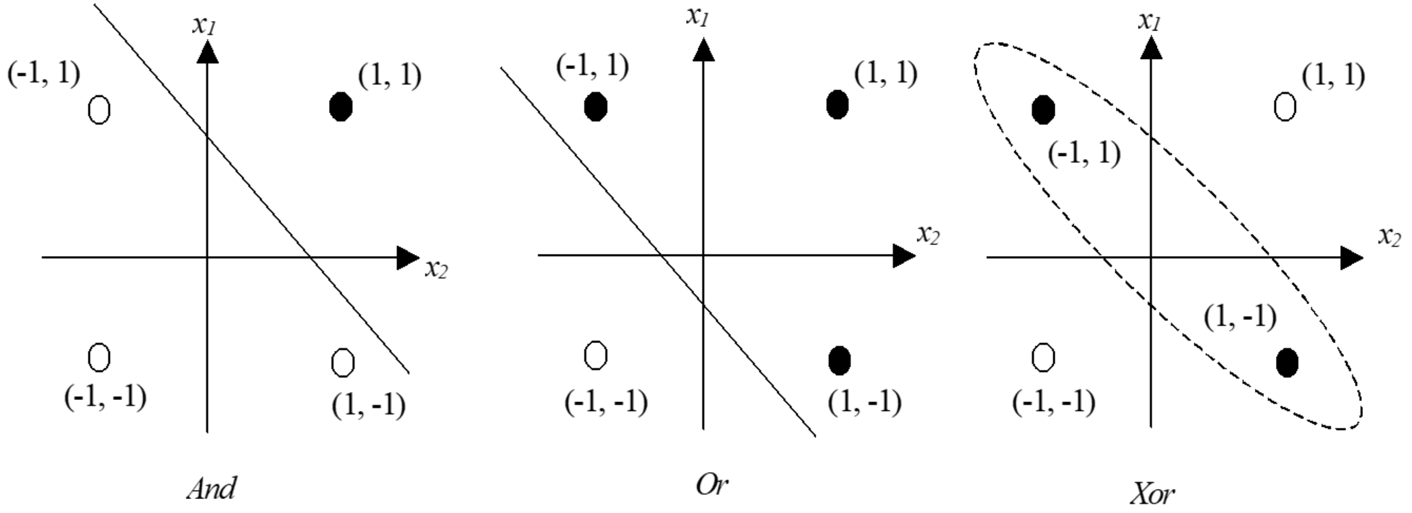
- In larger dimensional spaces, we say that the division is done by a hyperplane

- This implies that a single neuron is only able to represent linearly separable functions, which constitutes a serious limitation on the number of functions that can be represented

Limitations of a single neuron

- For example, it is very easy to build a network with just one element that behaves as a logic *And* a, a logic *Or*, or a NAND, but it is impossible to represent a function as simple as the logic *Xor*:





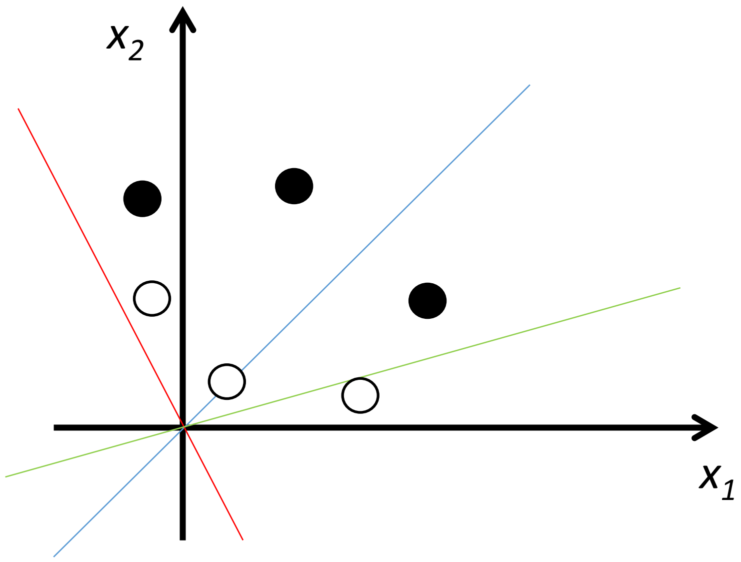
More on the effect of *b*

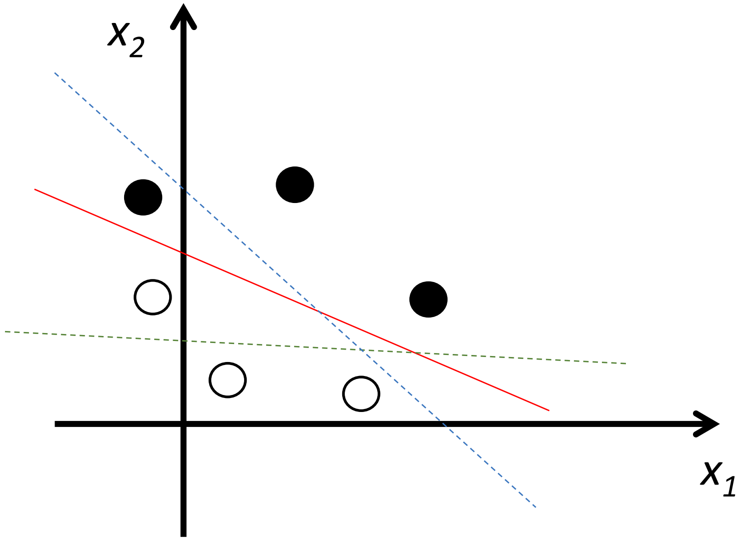
Let us now consider that we have two inputs x_1 and x_2 and that we use the step function

If $b = 0$ (if we don't use b), the straight line that divides the input space in two is equal to $x_1w_1+x_2w_2 = 0$.

This means that all straight lines will pass on (0, 0) regardless the values of w_1 and w_2

If b is used, and it is allowed to change during the training process, the algorithm can set a value to b that better optimizes the separation of inputs that should be classified as 0 and inputs that should be classified as 1



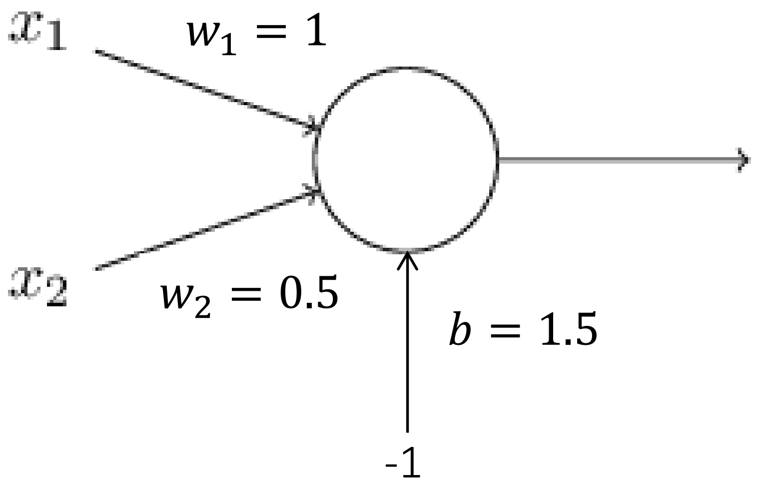


Bias connection input value

- The input value for the bias connection can be any value
- So far, we have used -1 because it is easier to understand the bias's role if we use this value
- However, the most commonly used value is 1
- Therefore, in the remaining slides we will assume bias connections have 1 as input

Exercise

- Consider the neuron below, which uses the step function as activation function



- Compute the output of the neuron for inputs samples [0, 0], [0, 1], [1, 0], [1, 1] and [0.4, 0.6]
- Also compute the output considering that the signal function is used; use also 1 as the bias inputs instead of 1

Learning rules

Virtually, all learning rules can be considered as variants of the Hebb's rule suggested by Hebb in [Organization of Behaviour, 1949]

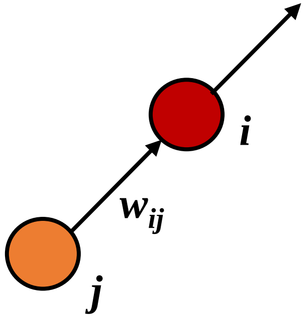
Hebb's rule

- Idea: if two units *i* and *j* are simultaneously active, then the connection between them should be strengthened
- If unit *i* receives a value from unit *j*, then Hebb's rule prescribes the modification of w_{ij} according to

$$\Delta w_{ij} = \alpha a_i a_j$$

or

$$w_{ij}(t+1) = w_{ij}(t) + \alpha a_i a_j$$



- α is a positive proportionality constant that represents the learning rate; a_i and a_j represent the output values of units *i* and *j*, respectively

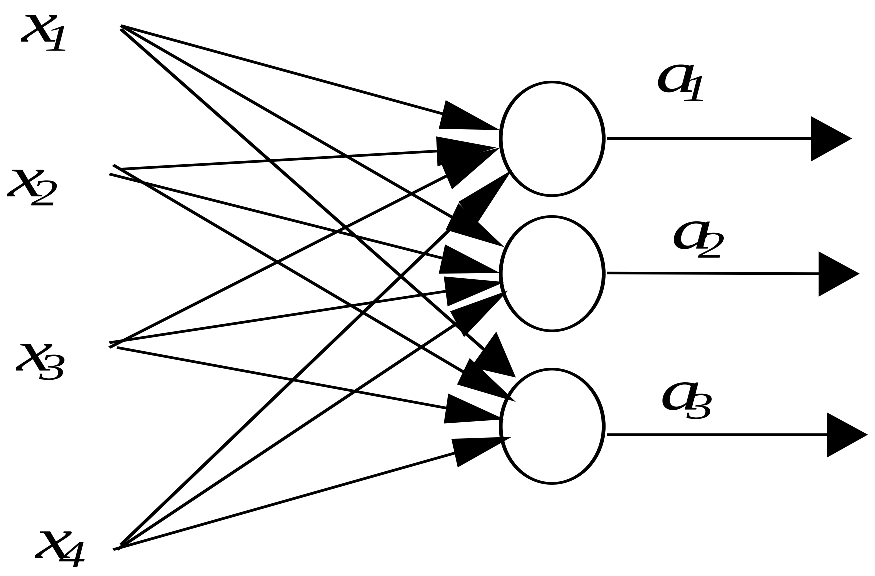
Perceptron (Frank Rosenblatt, 1957)

- We will now study how the train is done in one layer networks using, as example a perceptron network
- A perceptron network is one-layer feed-forward network
- Each neuron (perceptron), as seen before, computes the inputs' weighted sum and outputs a value of 1 if the sum is larger than $-b$ and -1, otherwise (the signal activation function is, therefore, used)
- The goal of a perceptron is the learning of some transformation T, using examples, each composed by the entry x and by the corresponding desired output $t = T(x)$

$$T:\{-1, 1\}^n \dashrightarrow \{-1, 1\}^m$$

This is the original formulation but $T:\{0, 1\}^n \dashrightarrow \{0, 1\}^m$ can be used as well (in fact, many sources describe perceptrons like this)

Scheme of a perceptron network



- Note that the units are independent from each other
- This means that we can limit our study to just one perceptron

Perceptron training

1. Initialize the weights randomly
2. **Do**
3. **For** each training example *x* from the training set **do**
4. Compute the output *a* of the network for input *x*
5. If $a \neq T(x)$ (the perceptron answers incorrectly), modify the connections weights (including *b*) as follows

$$w_i(new) = w_i(current) + \alpha(T(x) - a)\, x_i$$

where $0 < \alpha < 1$ represents the learning rate

6. **Until** $a == T(x)$ for all training examples

Perceptron training

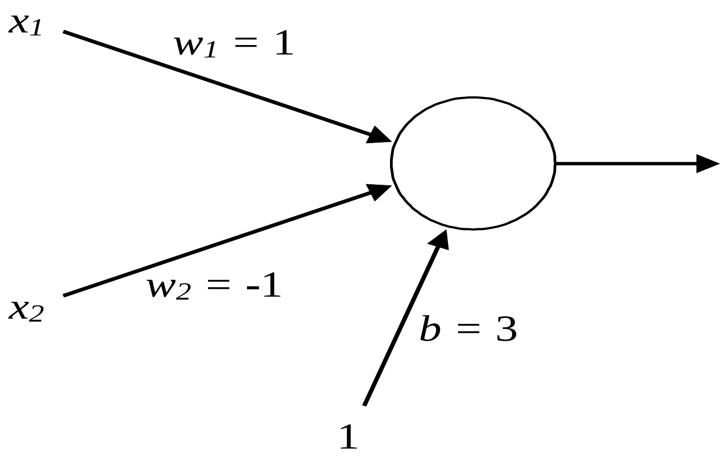
Error = desired output – current output

$$w_i(new) = w_i(current) + \alpha(T(x) - a)\, x_i$$

- This procedure is similar to the Hebb's rule
- The difference is that, instead of just the activation value of the perceptron, it uses the difference between the desired activation value, *T(x)*, and the current activation value, *a*

Perceptron training - example

- We want that the perceptron learns the logic And function using a learning rate of 1
- Let us consider that the perceptron is initialized in the following way



- The training examples consist of vectors [x_1, x_2] with values [-1, -1], [-1, 1], [1, -1] and [1, 1] (corresponding to the combinations that the And function inputs can have) associated to the corresponding desired output: -1, -1, -1, 1, respectively

Perceptron training - example

$w_1 \quad w_2 \quad b$

- 1^{st} presentation of the training set

current weights: [1, -1, 3]

Let us start by vector [-1, -1]. The perceptron's output is:

$$a = signal(1 \times (-1) + (-1) \times (-1) + 3 \times 1) = signal(-1 + 1 + 3) = signal(3) = 1$$

which **doesn't** correspond to the output of -1 *And* -1. Therefore, we need to update the weights:

$$w_1(new) = w_1(current) + \alpha \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$w_2(new) = w_2(current) + \alpha \times (t - a) \times x = -1 + 1 \times (-2) \times (-1) = 1$$

$$b(new) = b(current) + \alpha \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

We now present vector [-1, 1]:

current weights: [3, 1, 1]

$$a = signal(3 \times (-1) + 1 \times 1 + 1 \times 1) = signal(-3 + 1 + 1) = signal(-1) = -1$$

which corresponds to the output of -1 *And* 1. Therefore, nothing is done

Perceptron training - example

- 1^{st} presentation of the training set — current weights: [3, 1, 1]

Presentation of **[1, -1]**:

$$a = signal(3 \times 1 + 1 \times (-1) + 1 \times 1) = signal(3 - 1 + 1) = signal(3) = 1$$

which **doesn't** correspond to the output of 1 *And* -1. Therefore, we need to update the weights:

$$w_1(new) = w_1(current) + \alpha \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

$$w_2(new) = w_2(current) + \alpha \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$b(new) = b(current) + \alpha \times (t - a) \times x = 1 + 1 \times (-2) \times 1 = -1$$

Presentation of **[1, 1]**: — current weights: [1, 3, -1]

$$a = signal(1 \times 1 + 3 \times 1 + (-1) \times 1) = signal(1 + 3 - 1) = signal(3) = 1$$

which corresponds to the output of 1 *And* 1. Therefore, nothing is done

Perceptron training - example

- We need to do a 2nd presentation of the training set because the perceptron didn't answered correctly to at least one example during the 1st presentation

Perceptron training - example

- 2^{nd} presentation of the training set current weights: [1, 3, -1]

Presentation of **[-1, -1]**:

$$a = signal(1 \times (-1) + 3 \times (-1) + (-1) \times 1) = signal(-1 - 3 - 1) = signal(-5) = -1$$

which corresponds to the output of -1 *And* -1. Therefore, nothing is done

Presentation of **[-1, 1]**:

$$a = signal(1 \times (-1) + 3 \times 1 + (-1) \times 1) = signal(-1 + 3 - 1) = \mathrm{signal}(1) = 1$$

which **doesn't** correspond to the output of -1 *And* 1. Therefore, we need to update the weights:

$$w_1(new) = w_1(current) + \alpha \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$w_2(new) = w_2(current) + \alpha \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

$$b(new) = b(current) + \alpha \times (t - a) \times x = -1 + 1 \times (-2) \times 1 = -3$$

Perceptron training - example

- 2nd presentation of the training set current weights: [3, 1, -3]

Presentation of **[1, -1]**:

$$a = signal(3 \times 1 + 1 \times (-1) + (-3) \times 1) = signal(3 - 1 - 3\ ) = \mathrm{signal}(-1) = -1$$

which corresponds to the output of 1 *And* -1. Therefore, nothing is done

Presentation of **[1, 1]**:

$$a = signal(3 \times 1 + 1 \times 1 + (-3) \times 1) = signal(3 + 1 - 3\ ) = \mathrm{signal}(1) = 1$$

which corresponds to the output of 1 *And* 1. Therefore, nothing is done

Perceptron training - example

- We need to do a 3rd presentation of the training set because the perceptron didn't answered correctly to at least one example during the 2nd presentation

Perceptron training - example

- 3rd presentation of the training set current weights: [3, 1, -3]

Presentation of **[-1, -1]**:

$$a = signal(3 \times (-1) + 1 \times (-1) + -3 \times 1) = signal(-3 - 1 - 3) = signal(-7) = -1$$

which corresponds to the output of -1 *And* -1. Therefore, nothing is done

Presentation of **[-1, 1]**:

$$a = signal(3 \times (-1) + 1 \times 1 + -3 \times 1) = signal(-3 + 1 - 3) = signal(-5) = -1$$

which corresponds to the output of -1 *And* 1. Therefore, nothing is done

Perceptron training - example

- 3^{rd} presentation of the training set current weights: [3, 1, -3]

Presentation of **[1, -1]**:

$$a = signal(3 \times 1 + 1 \times (-1) + -3 \times 1) = signal(3 - 1 - 3) = signal(-1) = -1$$

which corresponds to the output of 1 *And* -1. Therefore, nothing is done

Presentation of **[1, 1]**:

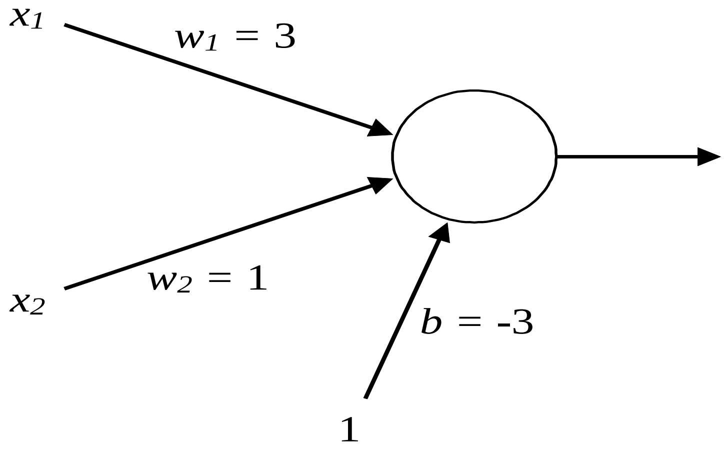
$$a = signal(3 \times 1 + 1 \times 1 + -3 \times 1) = signal(3 + 1 - 3) = signal(1) = 1$$

which corresponds to the output of 1 *And* 1. Therefore, nothing is done

Perceptron training - example

- In the 3rd presentation of the training set, the perceptron answered correctly to all examples

- Therefore, we can stop the training process. The resulting perceptron is:

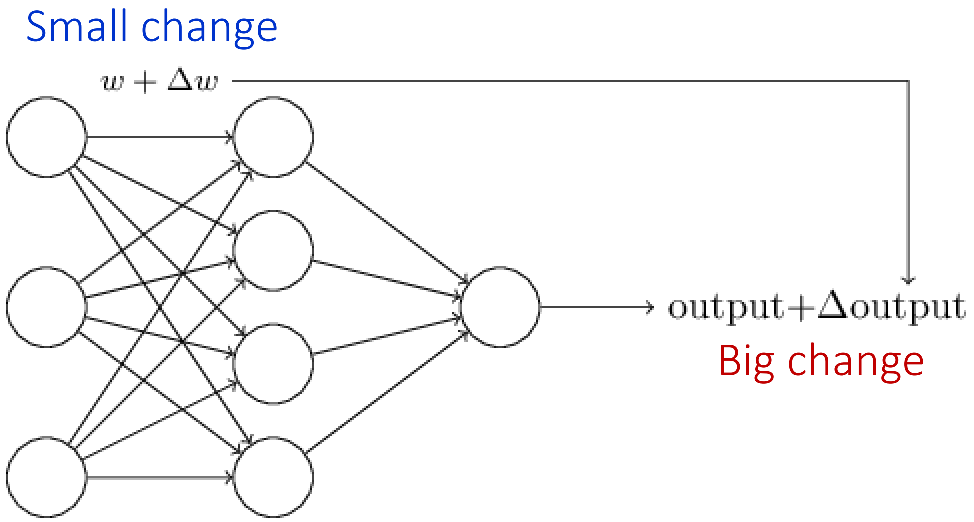


- Note: in the 3rd presentation of the training set, we could have stopped the training process after the presentation of example [-1, 1]. Why?

Discontinuous activation functions

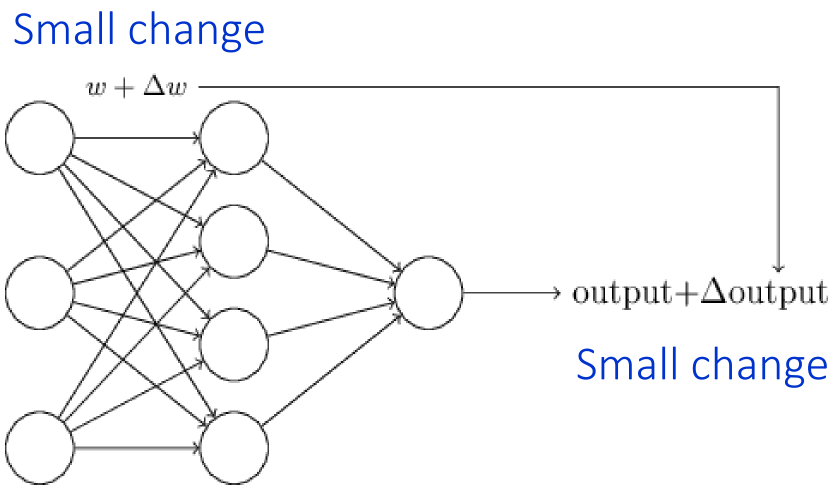
- Until now, we have seen examples mainly with the step and signal activation functions

- In NNs with this type of units, a small change in any weight or bias can sometimes cause the output of the network to completely flip (e.g. 0 to 1)



Continuous activation functions

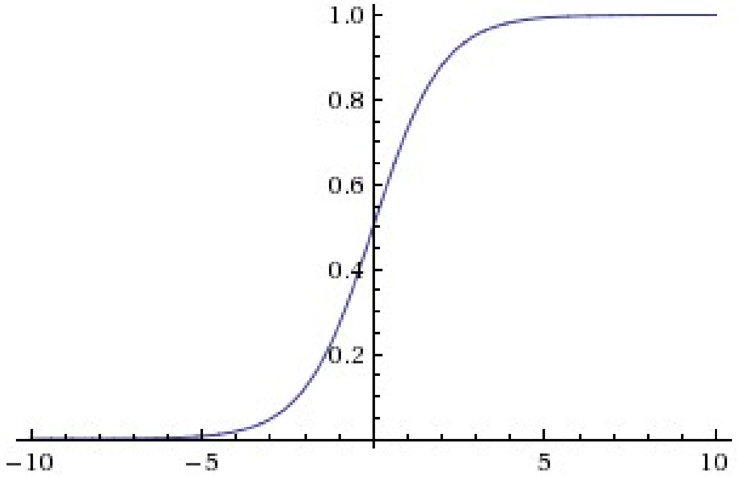
- This turns more difficult the training process of the network
- We would expect that a small change in the weights and bias would lead to a small change in the output



That would allow us to gradually modify the weights and biases so that the network gets progressively closer to the desired behavior

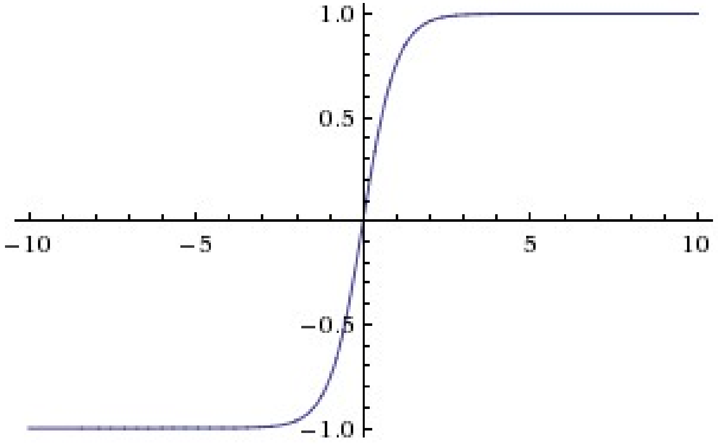
- We can overcome this problem by introducing continuous activation functions

Common (continuous) activation functions



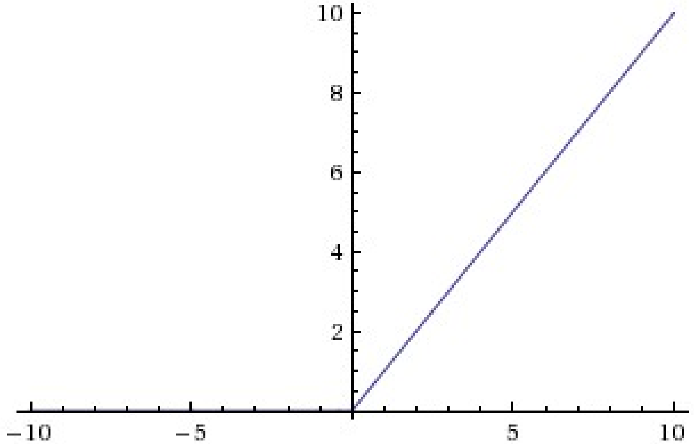
Sigmoid

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



Tanh

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



ReLU

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

Continuous activation functions

- In false/true (binary) problems, how do we interpret the result of activation functions like the sigmoid function, for example?

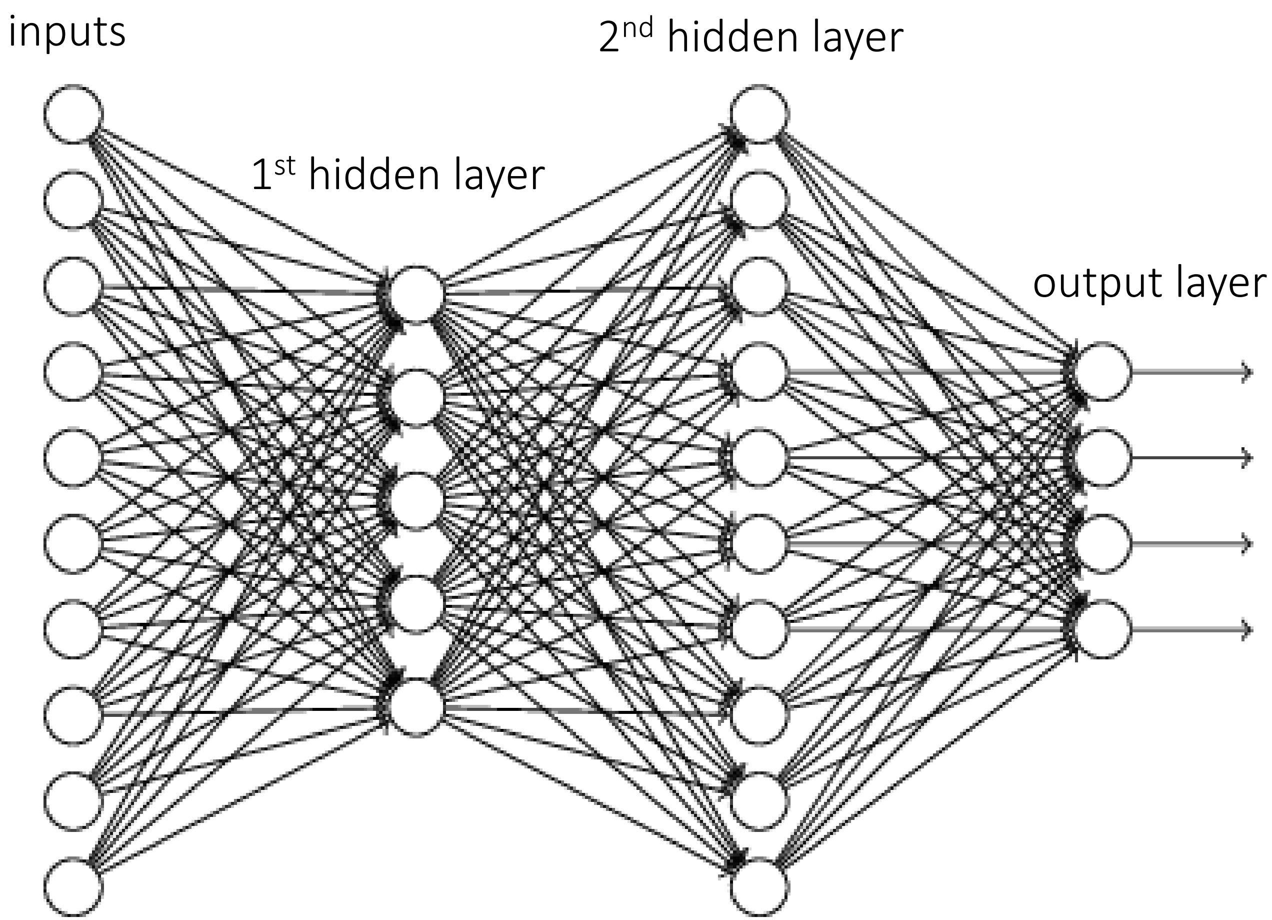
➢ If the function outputs a number equal or larger than 0.5, it is interpreted as a 1, else, it is interpreted as 0

Neural networks

Neural networks topologies

- There are different types of networks topologies, each one with its own characteristics
- The main distinction is between:
  - Feed-forward networks: where the connections are unidirectional and there are no feedback loops
  - Recurrent networks: where there are feedback loops (for example, the output neurons can be connected to the input ones). These networks can form arbitrary topologies

Feed-forward networks

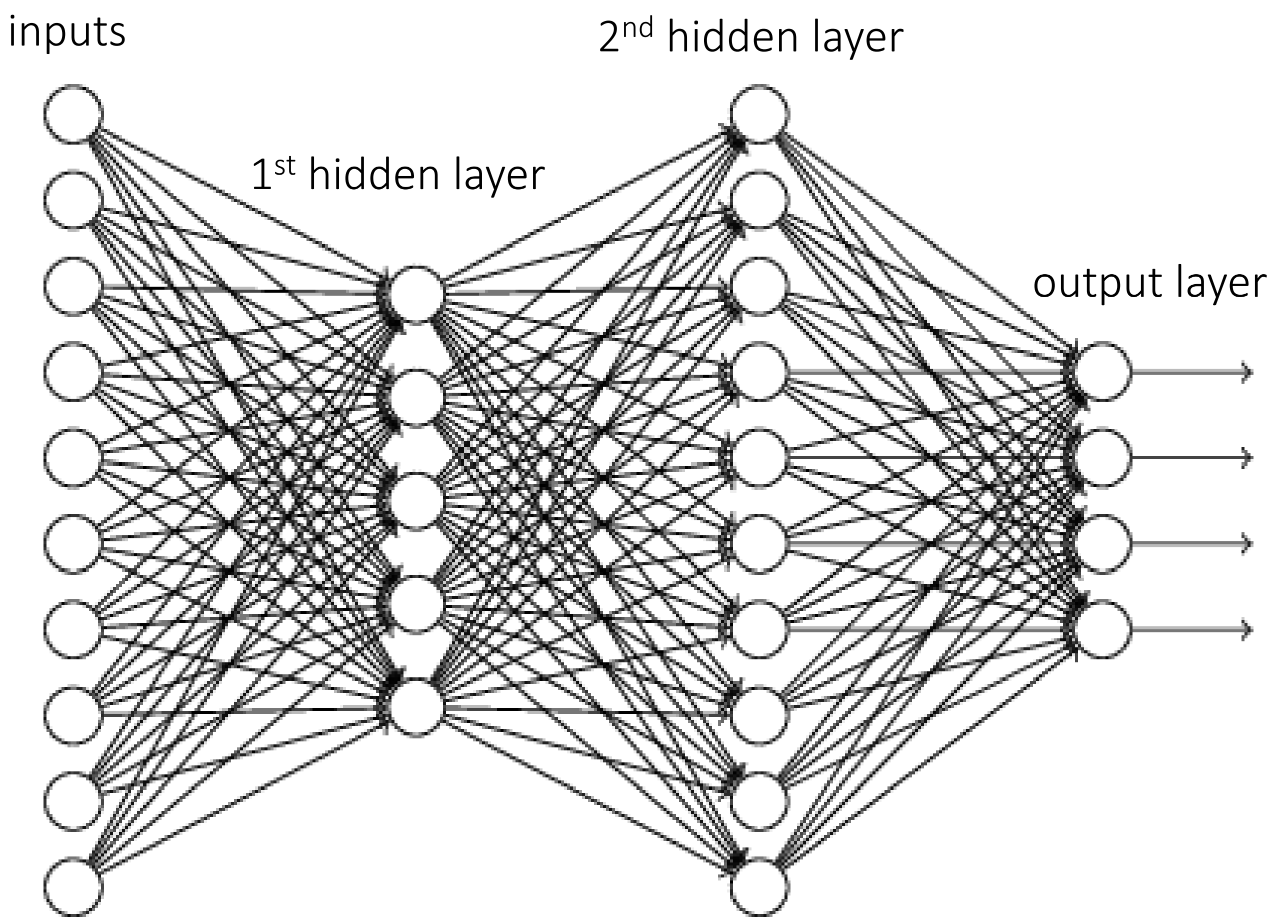


The most common FFN is the Multi-Layer Perceptron (MLP) network

In this type of networks, it is common to organize the neurons in layers

They are also called fully connected or dense neural networks

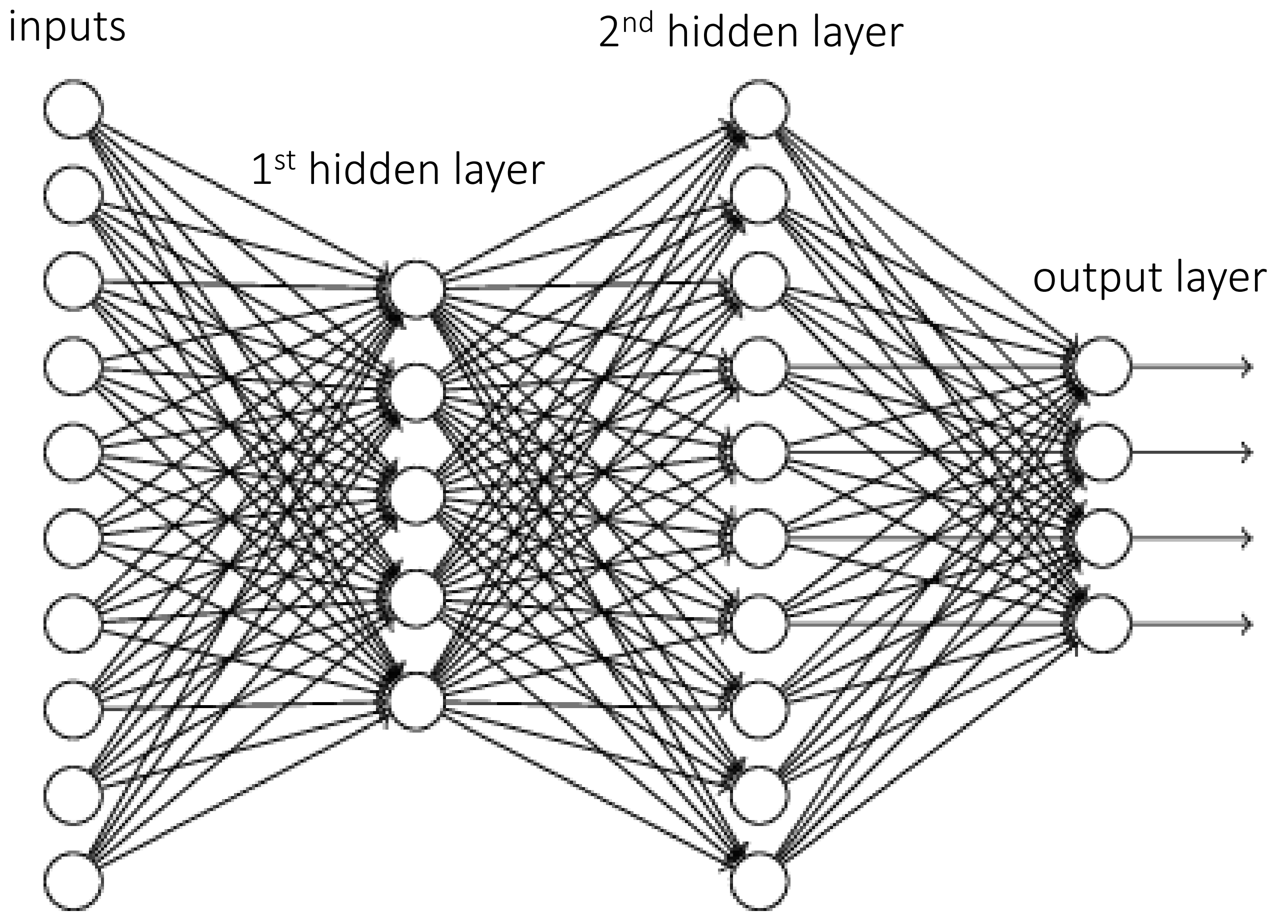
Feed-forward networks



In these networks, there are multiple hidden layers and each layer is densely connected to its previous layer

There are no connections between neurons of the same layer neither with neurons of preceding layers

Feed-forward networks



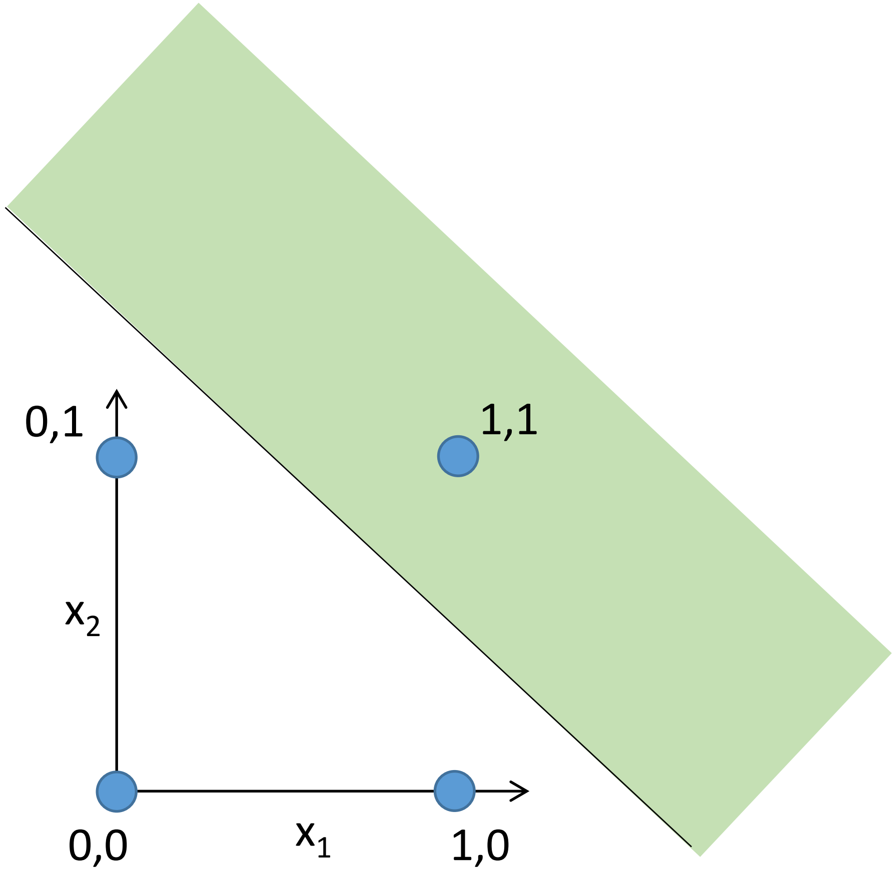
In each layer, processing happens as if it occurred in parallel

It it is fairly simple to make the computations because there are no loops

Feed-forward networks

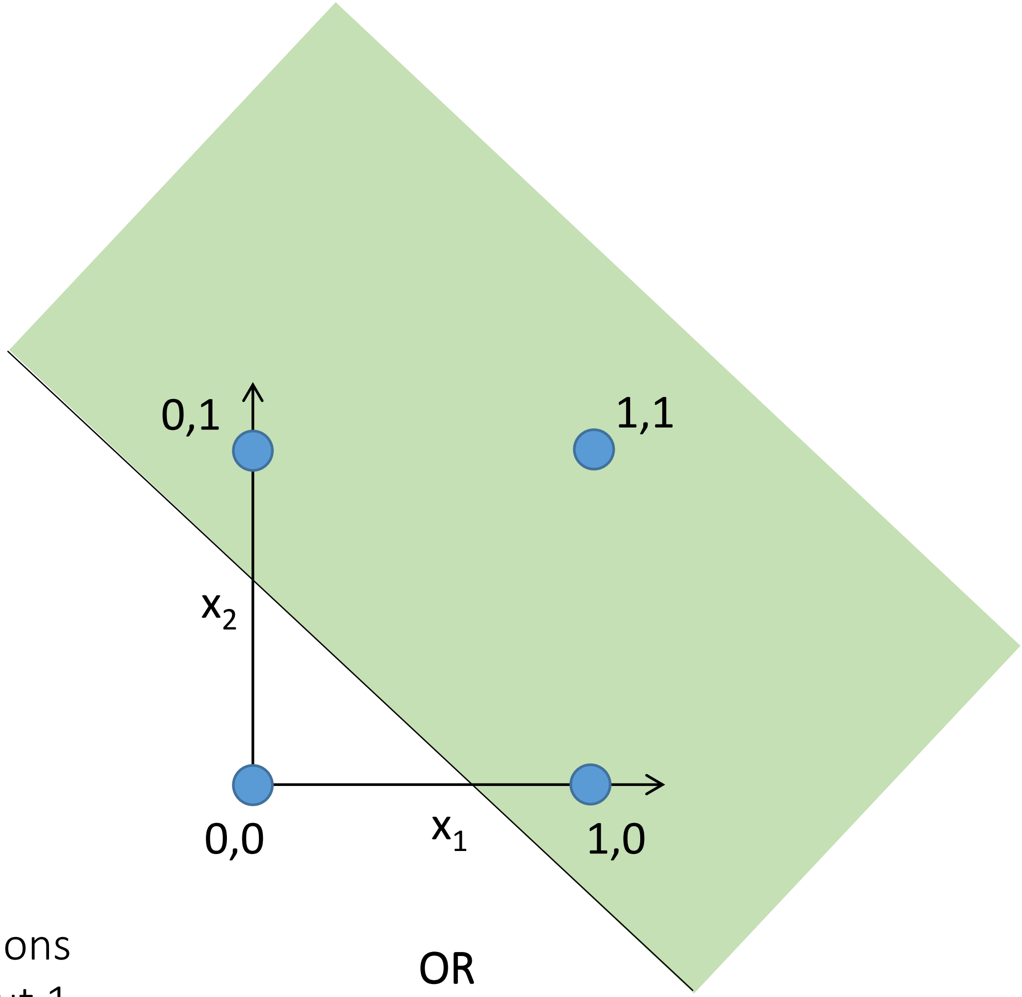
- A feed-forward network may not have hidden layers
- This simplifies the learning problem, but it implies, as already seen, strong limitations on what can be represented by the network
- With one sufficiently large hidden layer, it is possible to represent any continuous function
- With two hidden layers, discontinuous functions can be represented

AND and OR



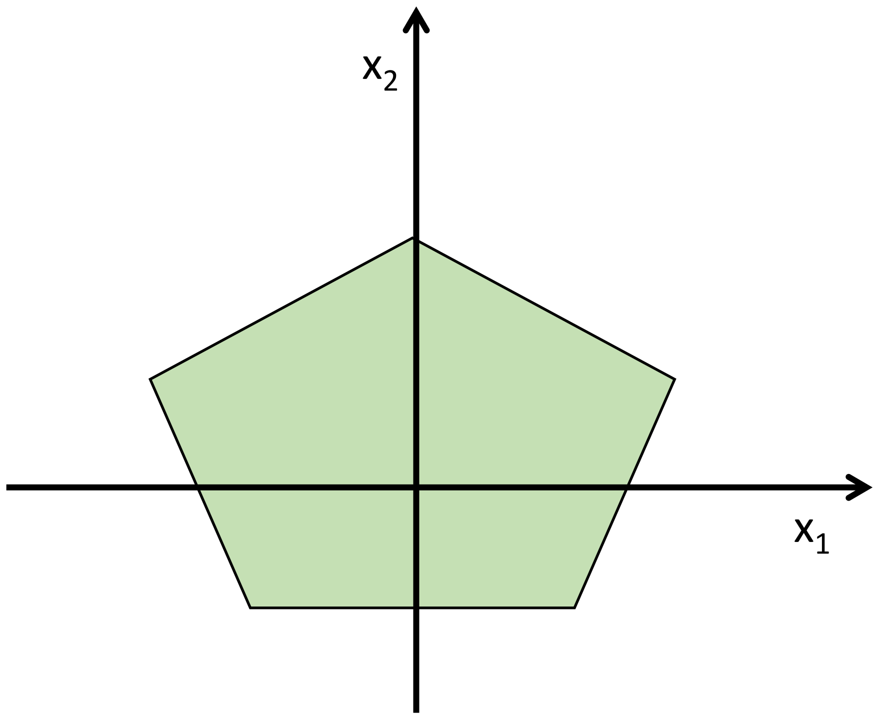
AND

Green regions
have output 1



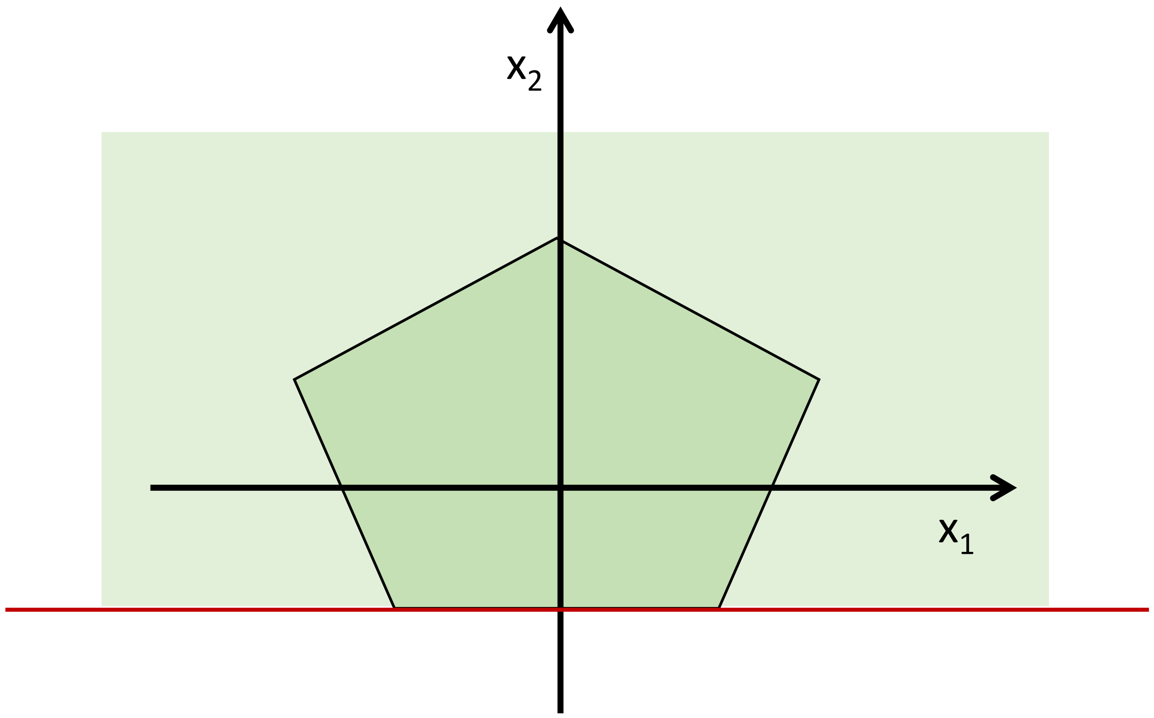
OR

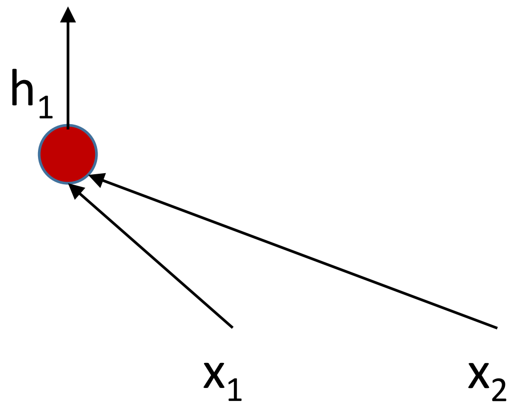
Composing convex polygons



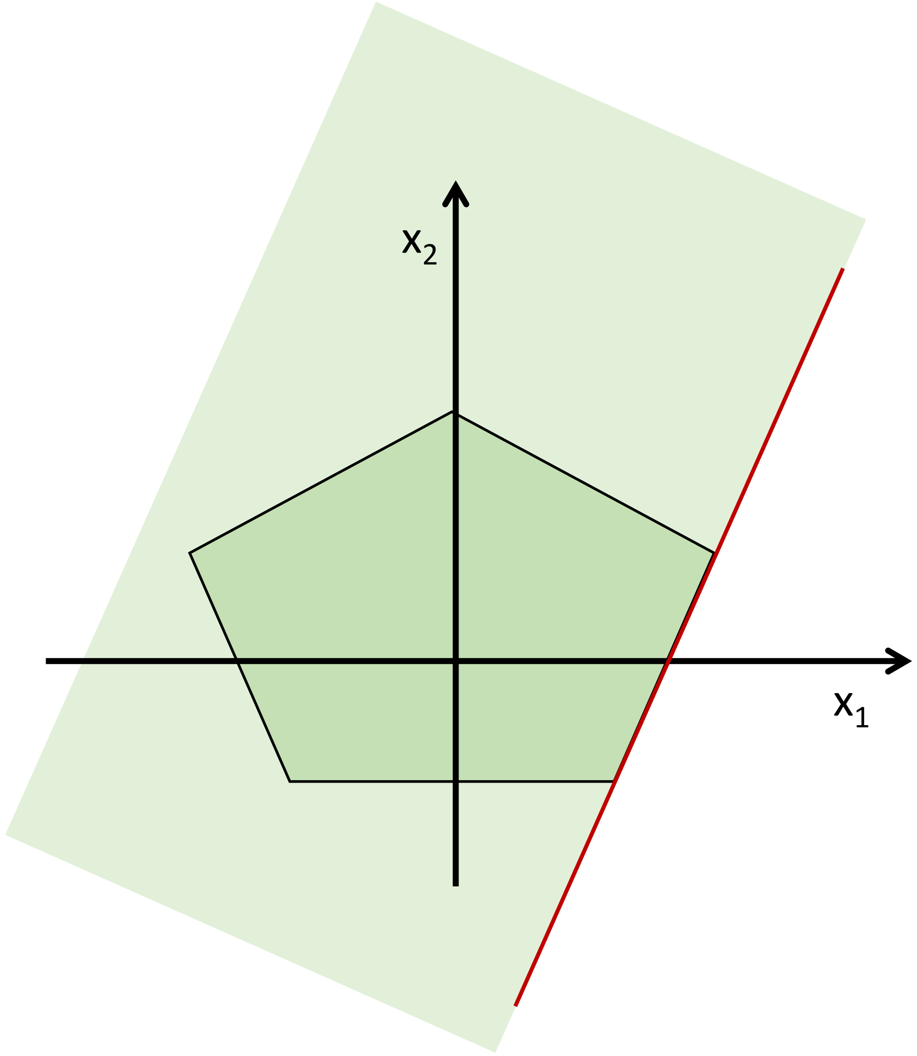
Let us build a neural network with a single output that fires 1 if the input is in the coloured area

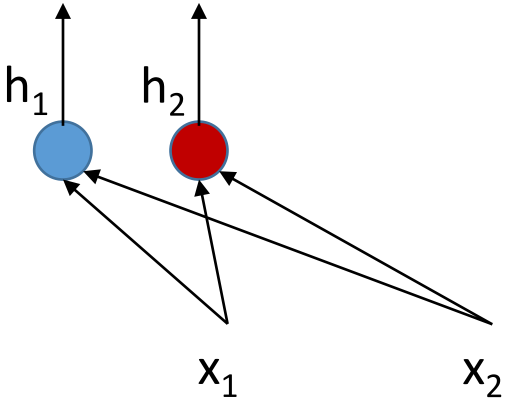
Composing convex polygons



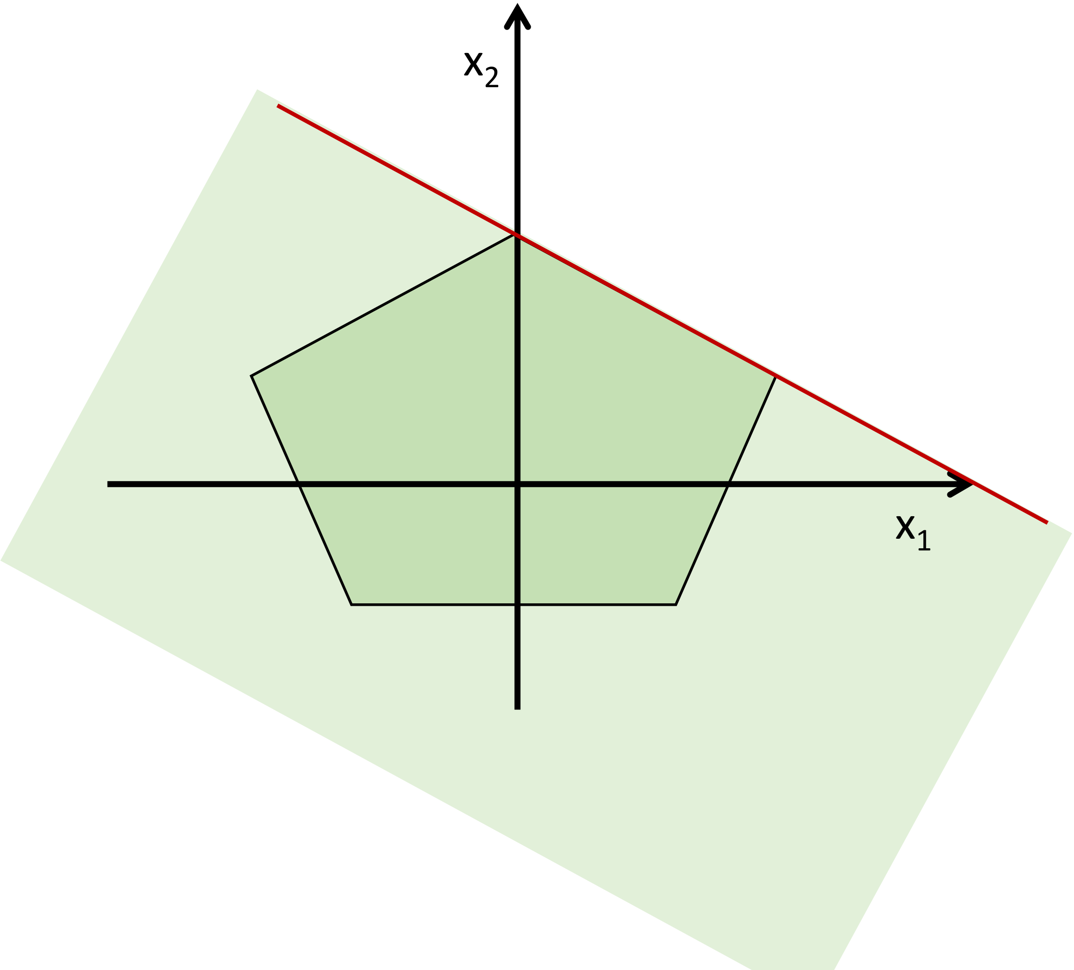


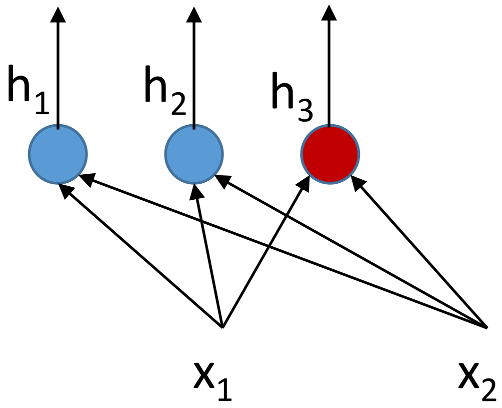
Composing convex polygons



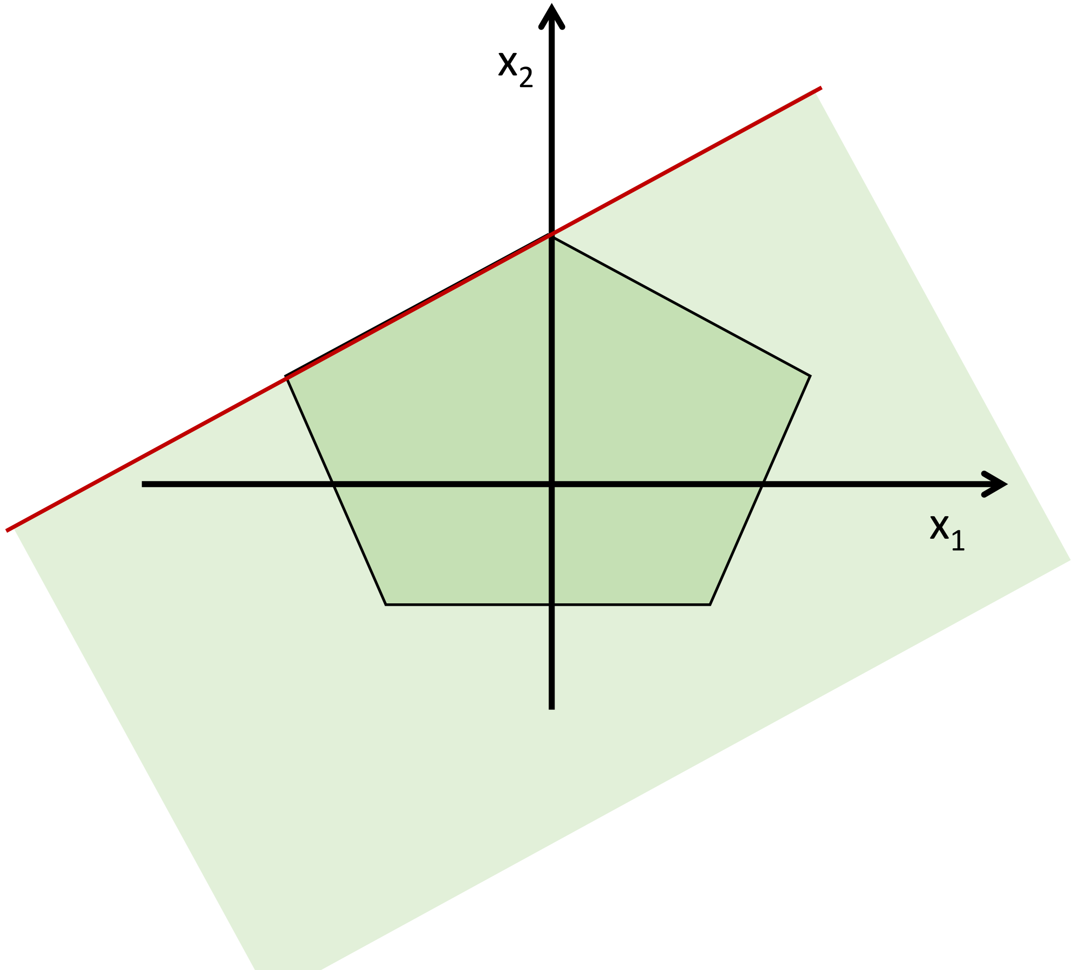


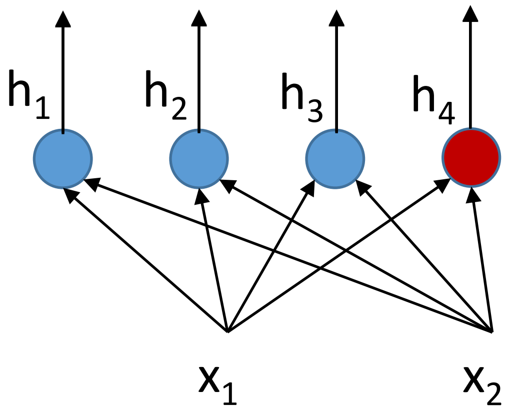
Composing convex polygons



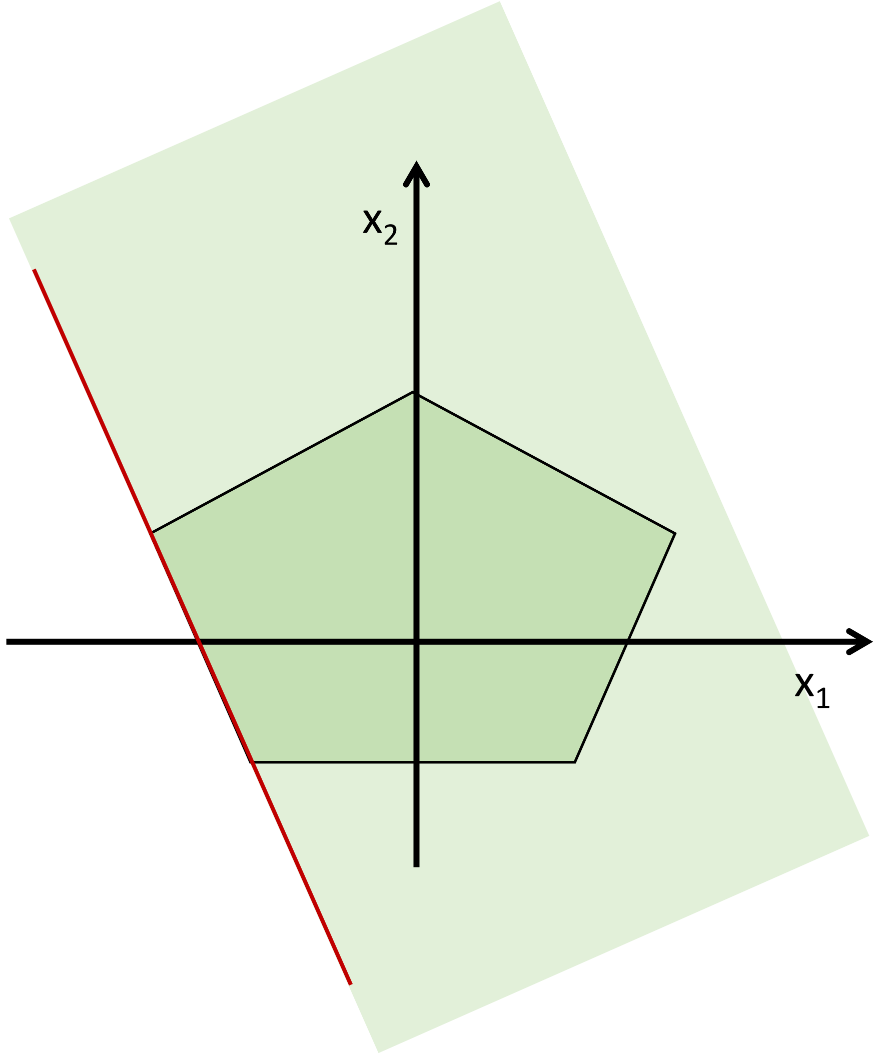


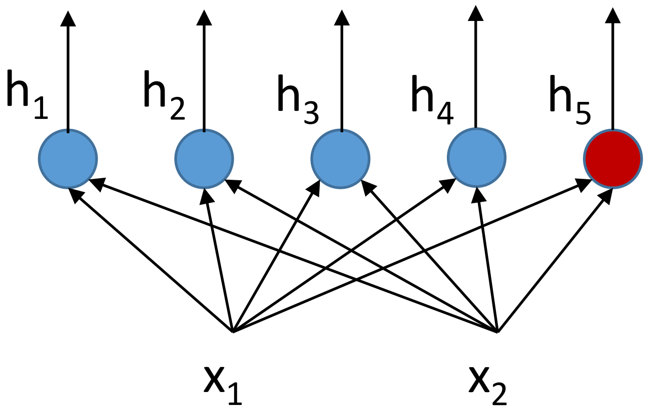
Composing convex polygons



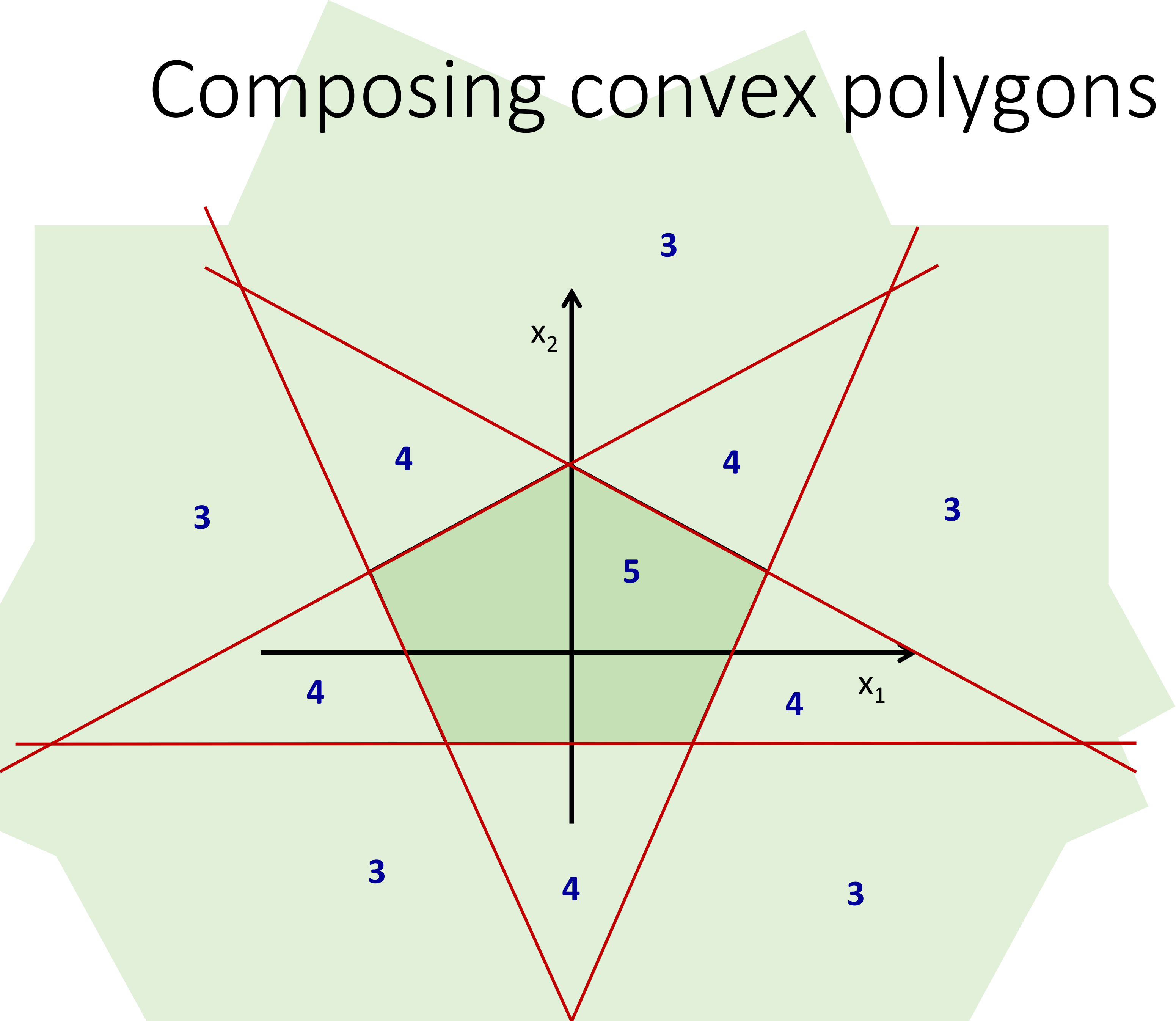


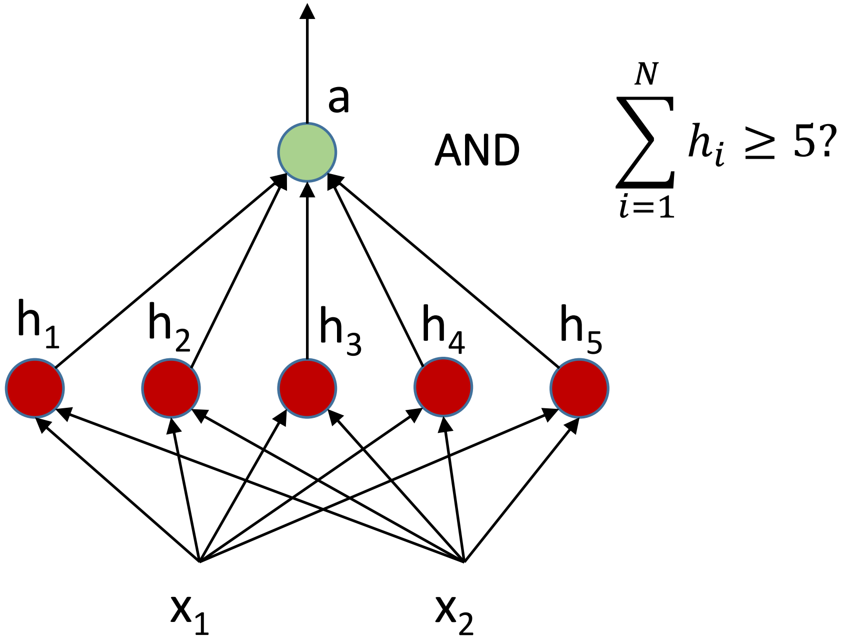
Composing convex polygons



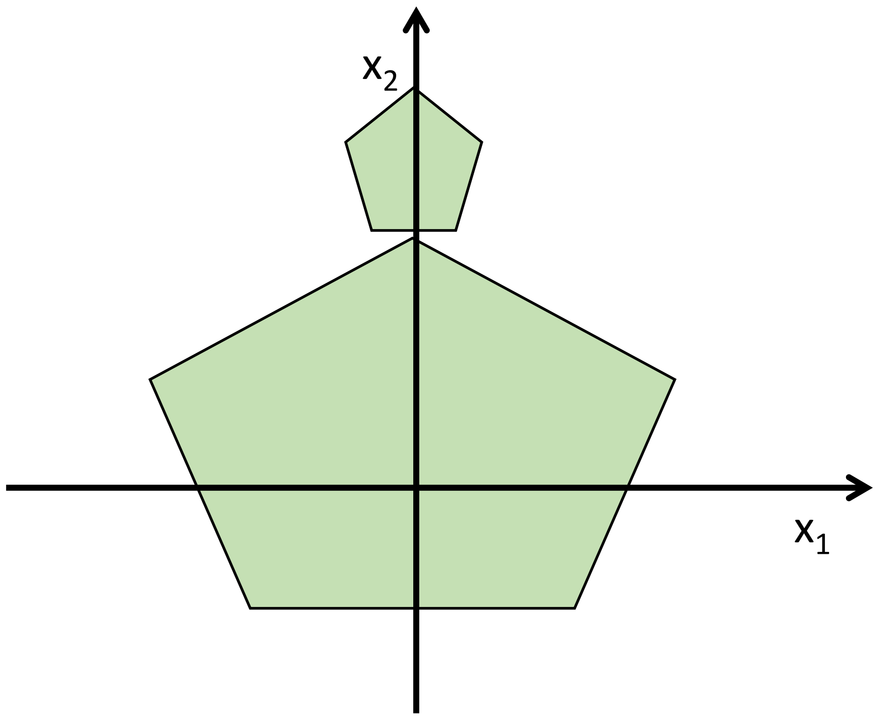


Composing convex polygons

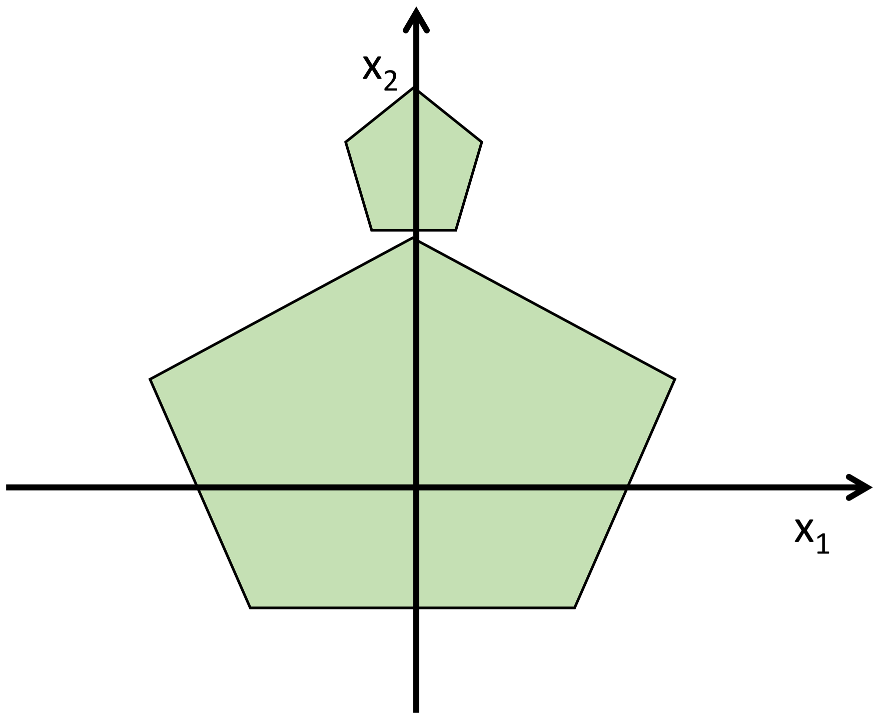


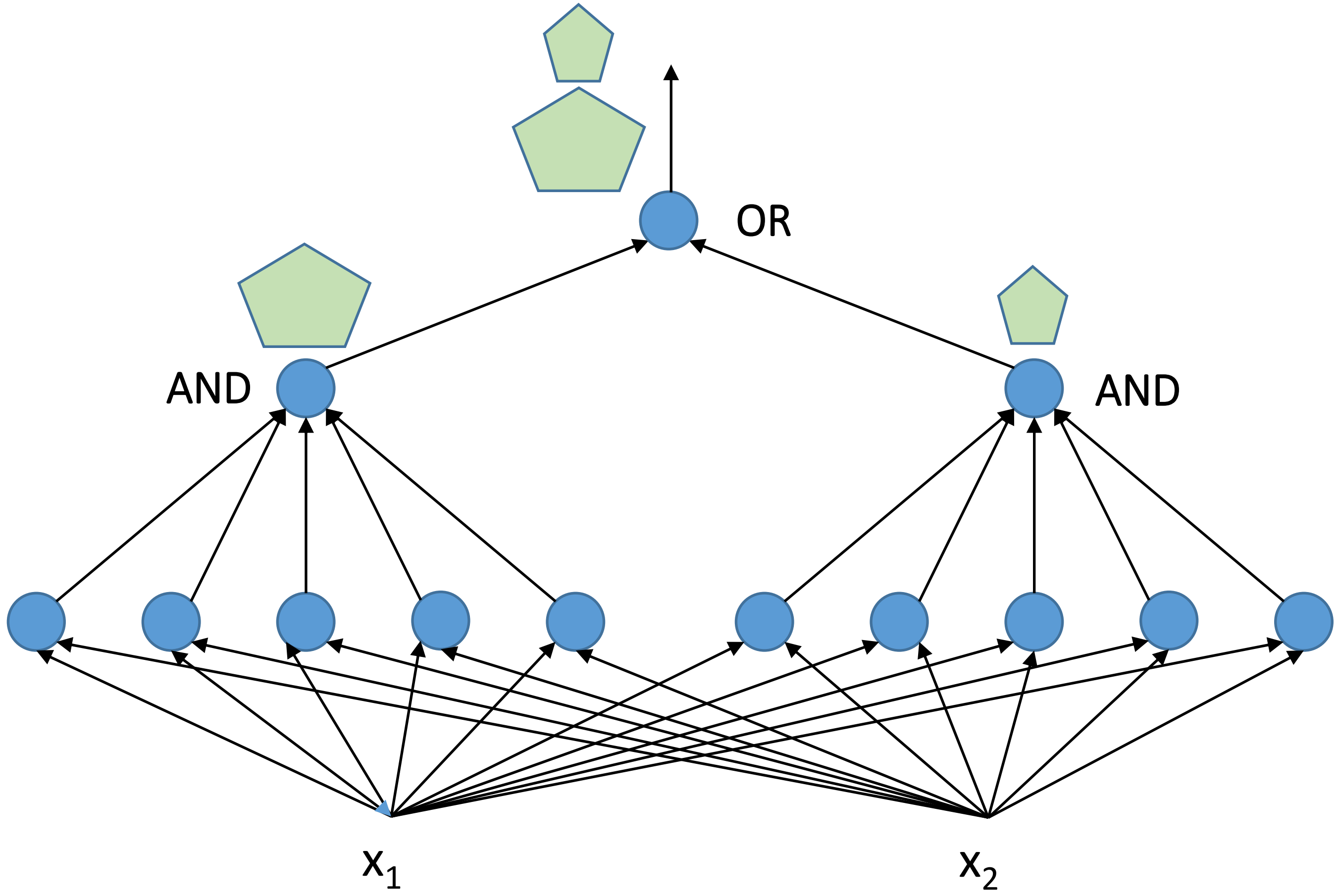


Composing N convex polygons



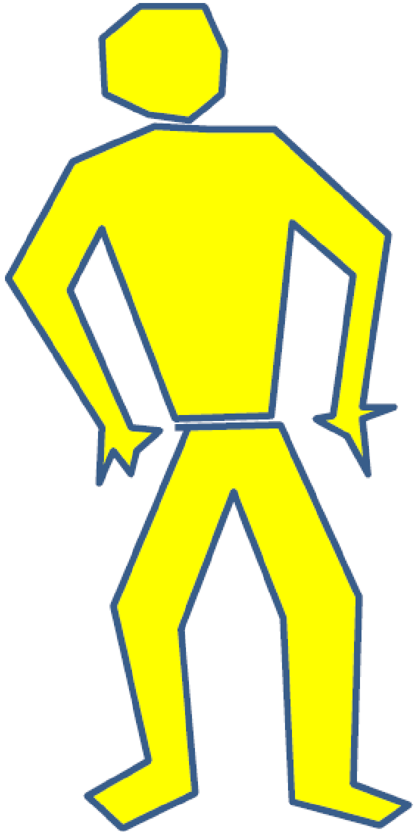
Composing N convex polygons

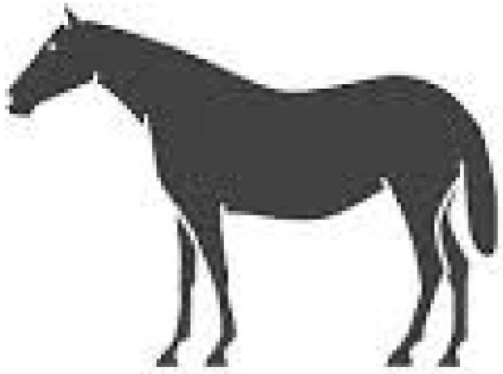




We need three layers

Complex shapes



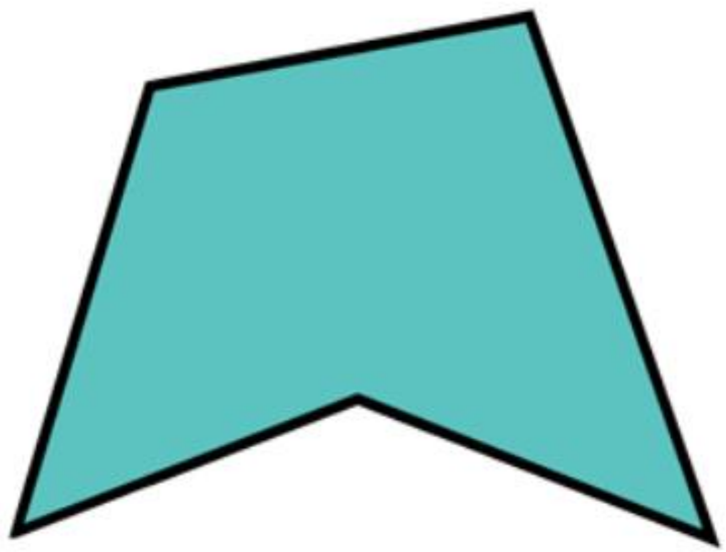


Neural networks can model very complex shapes

Exercise

- Setup a neural network able to model de XOR logic function

- Setup a neural network to model a polygon like the one below



How to put neural networks to work?

- What we have seen in the last previous slides can be done only in simple cases (e.g., two inputs, polygons)

- So, how to setup a neural network so that it works as desired?

➢ Given a dataset, there are learning algorithms which can automatically tune the weights and biases of a neural network

Using neural networks

- In order to use a neural network, we have to decide:
  - What network architecture will be used (how many neurons and how will they be connected)
  - The type of neurons that will be used
- Then, initiate the weights and start the training phase, during which the weights and biases are modified until the network works as desired

What is the best architecture?

- One of the problems posed to who wants to build a neural network is the choice of its structure
- If the network is too small, it may not be able to represent/model the desired function
- If it is too big, it can be able to represent exactly the training examples but be unable to generalize beyond those examples (this is called the *overfitting* problem)
- The truth is that there is no method for choosing what is exactly the best structure for a neural network

What is the best architecture?

Picking the right network architecture is more an art than a science; and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect.

Deep learning with Python, Francois Chollet, Manning Publications, 2018

Exercise

- Suppose that we want to build one neural network able to recognize handwritten digits like the ones below. You may assume that all images are 28x28 pixels

- How many inputs and outputs should the neural network have? What activation function would you use in the output layer?

- What about the number of hidden layers and the number of units each layer should have?

Neural networks training

- A neural network must be configured so that a set of inputs produce the desired output

- There are basically two methods to set the network connections' weights:
    - Using a priori knowledge (rarely/never used)
    - Training the network using some learning method

Hyperparameters and parameters

- Hyperparameters are
  - the parameters that determine the network architecture: number of layers, number of neurons in each layer, activation functions used
  - the parameters of the learning process: optimizer, loss function, learning rate, batch size, number of epochs, etc.
- Parameters are the parameters tuned during the training process, typically, the weights and biases

Neural networks training

- Types of learning:
  - Supervised
  - Unsupervised
  - Reinforcement learning
- These learning paradigms involve the actualization of weights of the connections between neurons, according to some learning rule

Supervised learning

- In supervised learning, the samples of the dataset used to train the neural network are composed of the input value(s), as well as the corresponding desired output values (often called labels)

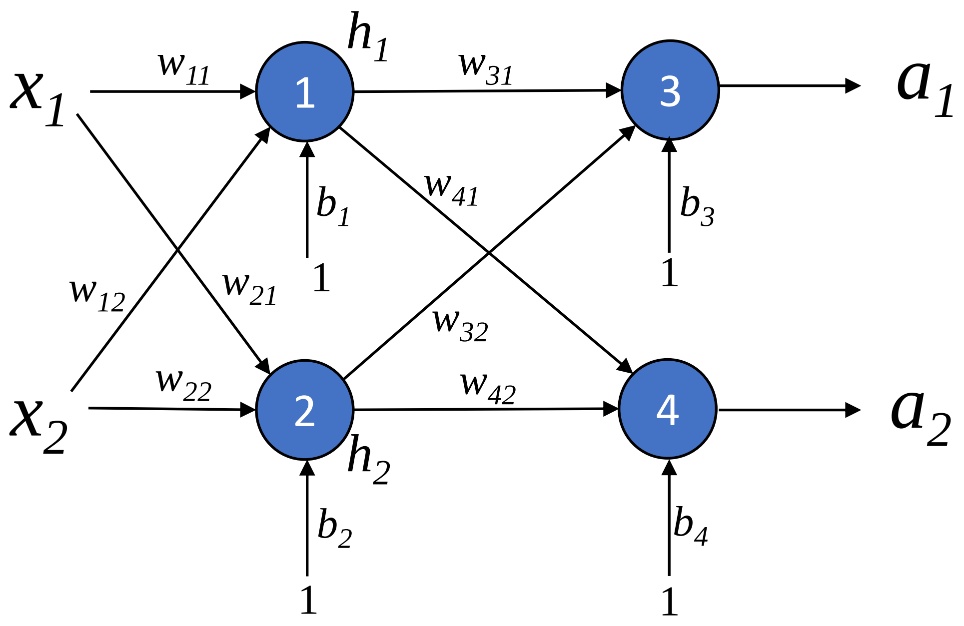
- For example, in the handwritten digits recognition problem, a sample corresponding to digit 6 would be composed by the set x of the image pixel values, as well as by the desired output

$$y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$$

Supervised learning

- The training algorithm uses the desired output to adjust the network weights and biases of the neural network in order to reduce the difference between the desired and actual output

Multilayer neural networks

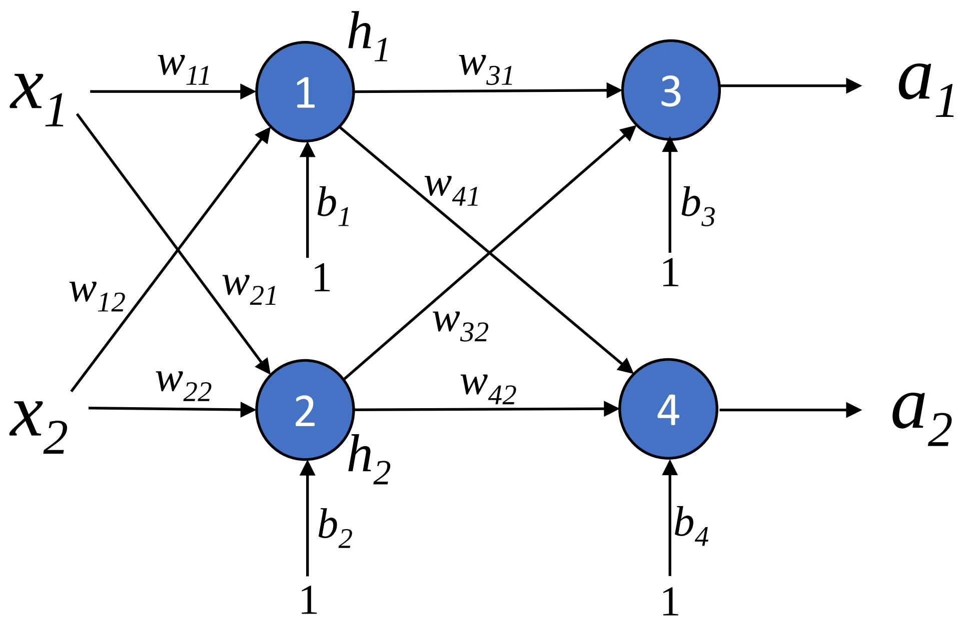


Example with 2 inputs, 1 hidden layer with 2 units and output layer with 2 units

As we have seen before, these networks are often called Multi-Layer Perceptron networks (MLP)

By tradition, the weight of the connection from unit *j* to unit *i* is represented as w_{ij}

1^{st} layer output computation



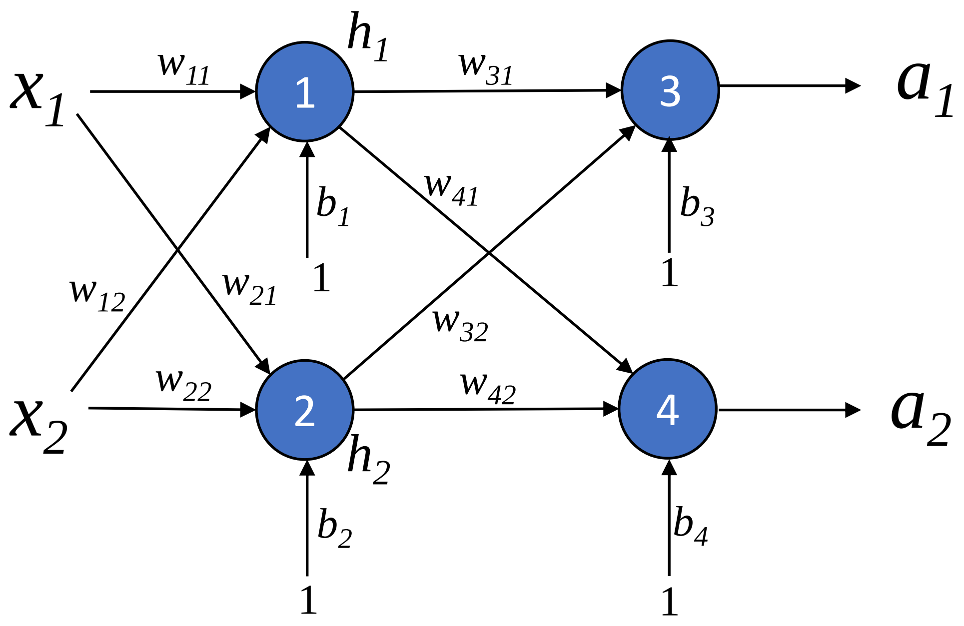
$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \xrightarrow{f(z)} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

$f(z) \rightarrow$ activation function

$z_i \rightarrow$ i^{th} unit weighted sum

$h_i \rightarrow$ i^{th} hidden unit activation value

2nd layer output computation



$$\begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} z_3 \\ z_4 \end{bmatrix} \xrightarrow{f(z)} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

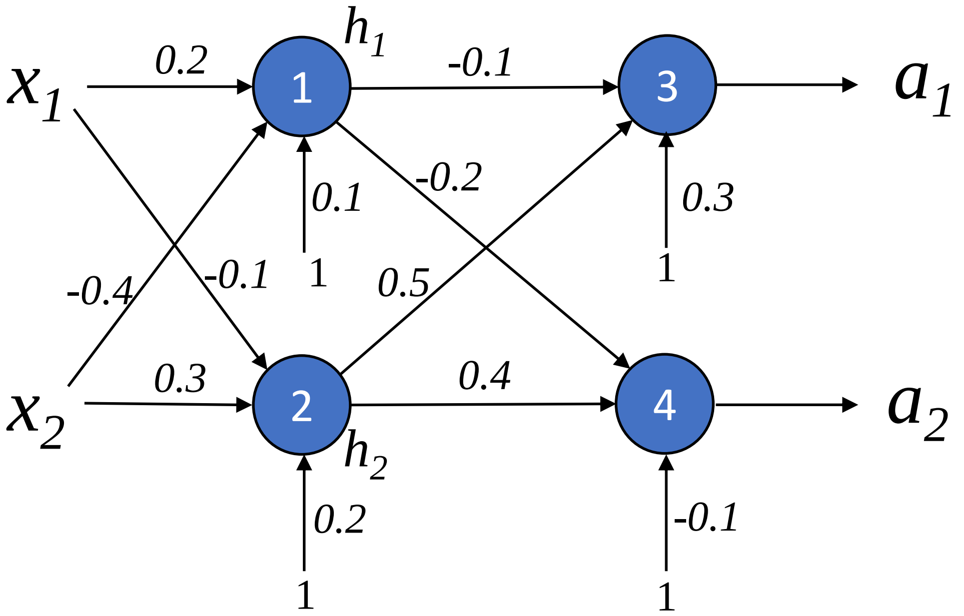
$f(z) \rightarrow$ activation function

$z_i \rightarrow$ i^{th} unit weighted sum

$a_i \rightarrow$ i^{th} output unit activation value

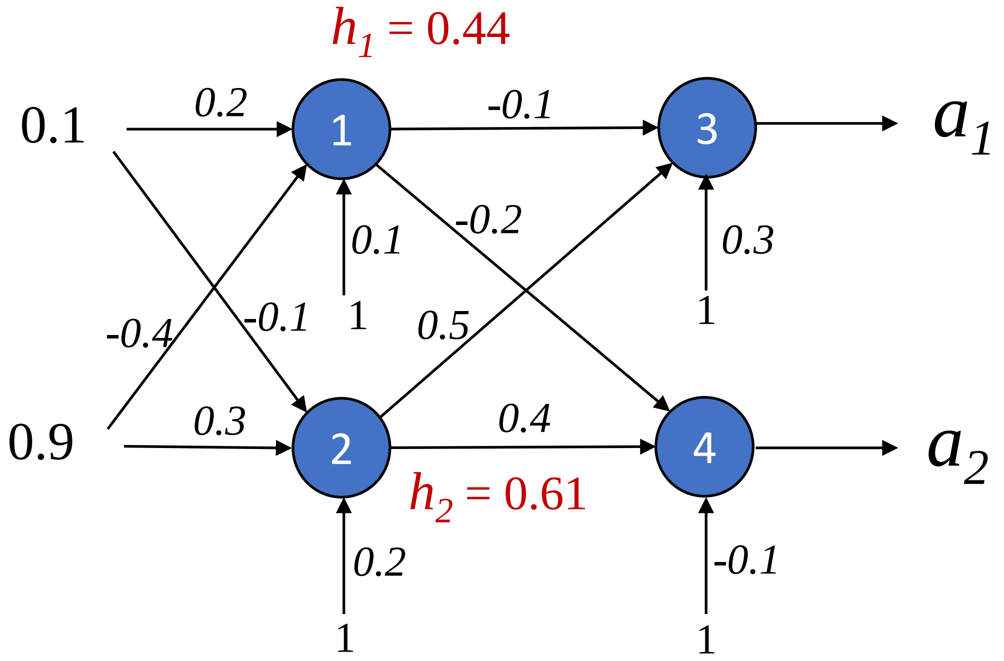
Exercise

- Compute the output of the network below when the input sample is $[x_1, x_2] = [0.1, 0.9]$. Assume that the sigmoid function is used



Exercise - solution

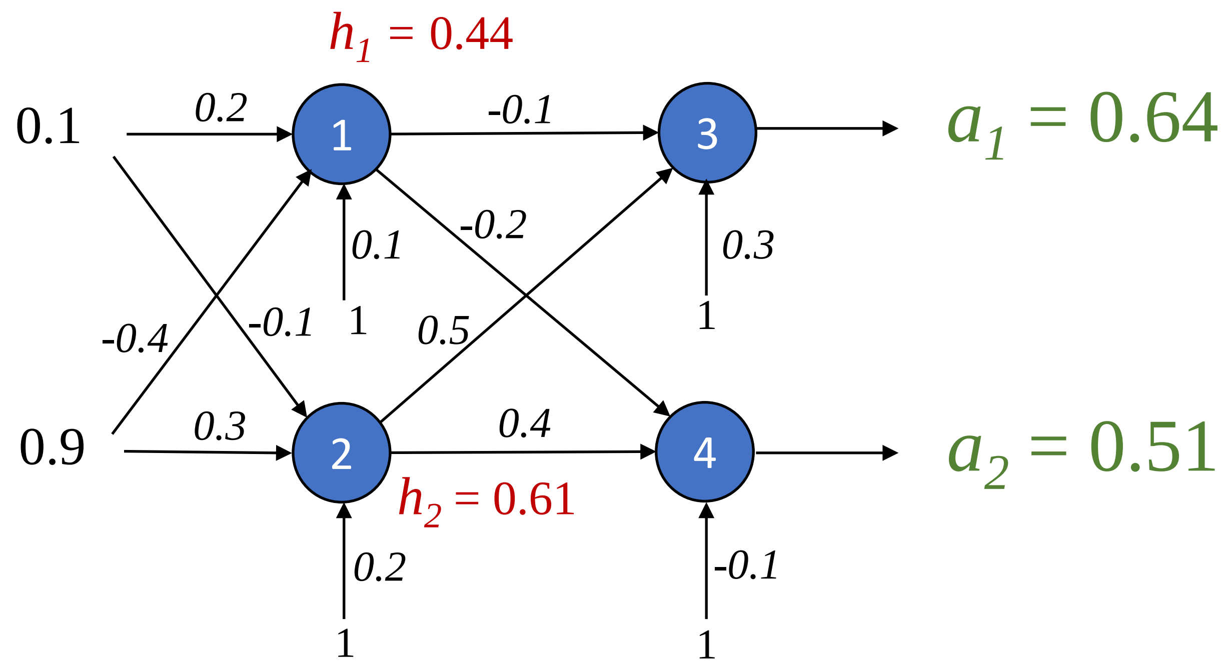
- $h_1 = f(0.1 \times 0.2 + 0.9 \times (-0.4) + 0.1) = f(-0.24) = \frac{1}{1+e^{-(-0.24)}} = 0.44$
- $h_2 = f(0.1 \times (-0.1) + 0.9 \times 0.3 + 0.2) = f(0.46) = \frac{1}{1+e^{-0.46}} = 0.61$



Exercise - solution

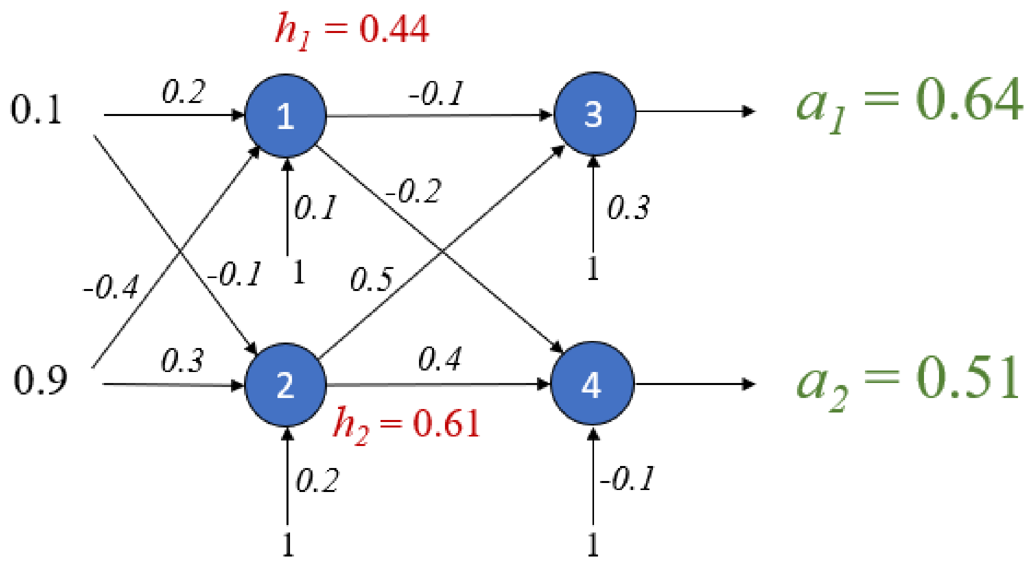
- $a_1 = f(0.44 \times (-0.1) + 0.61 \times 0.5 + 0.3) = f(0.561) = \frac{1}{1+e^{-0.561}} = 0.64$

- $a_2 = f(0.44 \times (-0.2) + 0.61 \times 0.4 - 0.1) = f(0.056) = \frac{1}{1+e^{-0.056}} = 0.51$



Exercise – solution using vector notation

- $\begin{bmatrix} 0.2 & -0.4 \\ -0.1 & 0.3 \end{bmatrix} \begin{bmatrix} 0.1 \\ 0.9 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} -0.24 \\ 0.46 \end{bmatrix} \xrightarrow{f(z)} \begin{bmatrix} 0.44 \\ 0.61 \end{bmatrix} \begin{matrix} h_1 \\ h_2 \end{matrix}$ *1st layer*
- $\begin{bmatrix} -0.1 & 0.5 \\ -0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 0.44 \\ 0.61 \end{bmatrix} + \begin{bmatrix} 0.3 \\ -0.1 \end{bmatrix} = \begin{bmatrix} 0.561 \\ 0.056 \end{bmatrix} \xrightarrow{f(z)} \begin{bmatrix} 0.64 \\ 0.51 \end{bmatrix} \begin{matrix} a_1 \\ a_2 \end{matrix}$ *2nd layer*



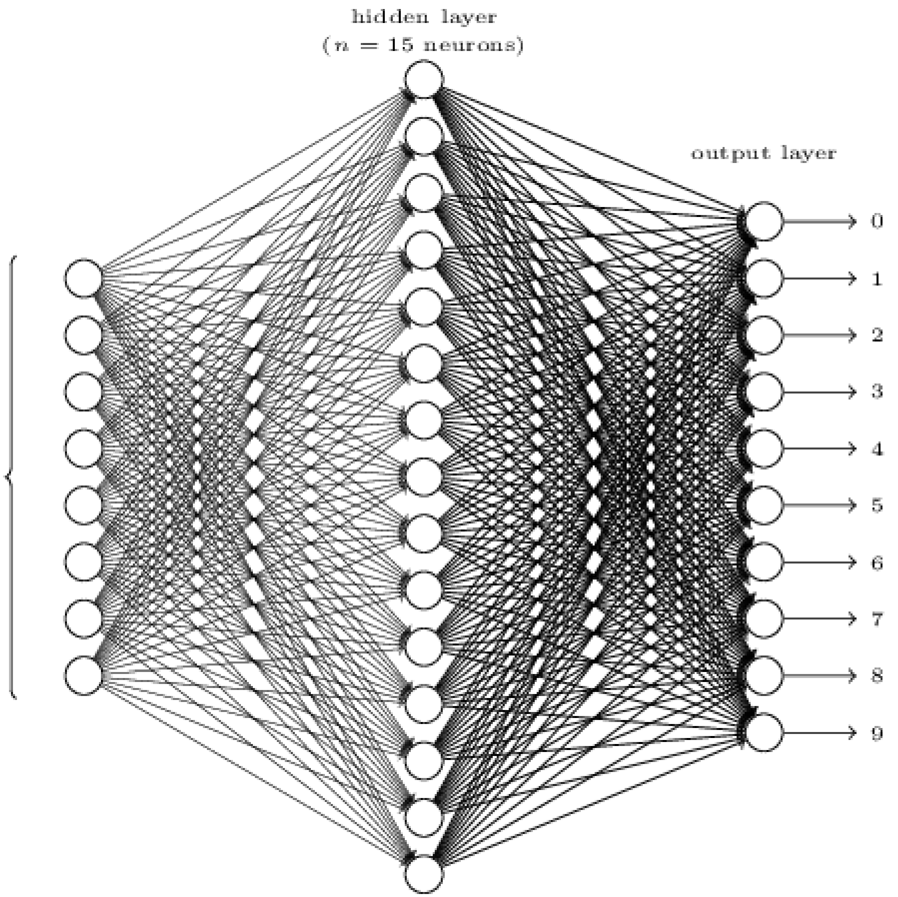
Handwritten digits recognition

- Consider the following neural network architecture for the handwritten digits recognition problem

- The desired output for an image corresponding to digit 6 would be $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$, where x represents the network input sample (vector with the image pixels)

- We need an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training samples x

784 inputs



Loss function

- To quantify how well the network behaves, we can use the following function

$$L(w) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

This function is called the Mean Squared Error (MSE)

- Where:
  - w represents all the network's weights and biases (after all, biases are weights!)
  - n represents the number of training samples
  - a is the output vector for sample x
  - $y(x)$ is the desired output vector for sample x

Other functions can be used. The function used to assess the performance of a neural network is called the loss function. Other used names are error, cost, or objective function

MSE computation example

- If we have two training samples x_a and x_b for which
  - $y(x_a) = [1, 0]$ and $a_a = [0.8, 0.2]$
  - $y(x_b) = [1, 1]$ and $a_b = [0.7, 0.9]$
- Then, $L(w) = \frac{1}{2n} \sum_x \| y(x) - a \|^2 = \frac{1}{2\times 2} \left(\left\| \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \right\|^2 + \left\| \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.7 \\ 0.9 \end{bmatrix} \right\|^2 \right) =$

$$= \frac{1}{4} \left(\left\| \begin{bmatrix} 1 - 0.8 \\ 0 - 0.2 \end{bmatrix} \right\|^2 + \left\| \begin{bmatrix} 1 - 0.7 \\ 1 - 0.9 \end{bmatrix} \right\|^2 \right) =$$

$$= \frac{1}{4} \left(\sqrt{(1-0.8)^2 + (0-0.2)^2}^{\,2} + \sqrt{(1-0.7)^2 + (1-0.9)^2}^{\,2} \right) =$$

$$= \frac{1}{4} [(1-0.8)^2 + (0-0.2)^2 + (1-0.7)^2 + (1-0.9)^2] = \dots$$

Loss function

$$L(w) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

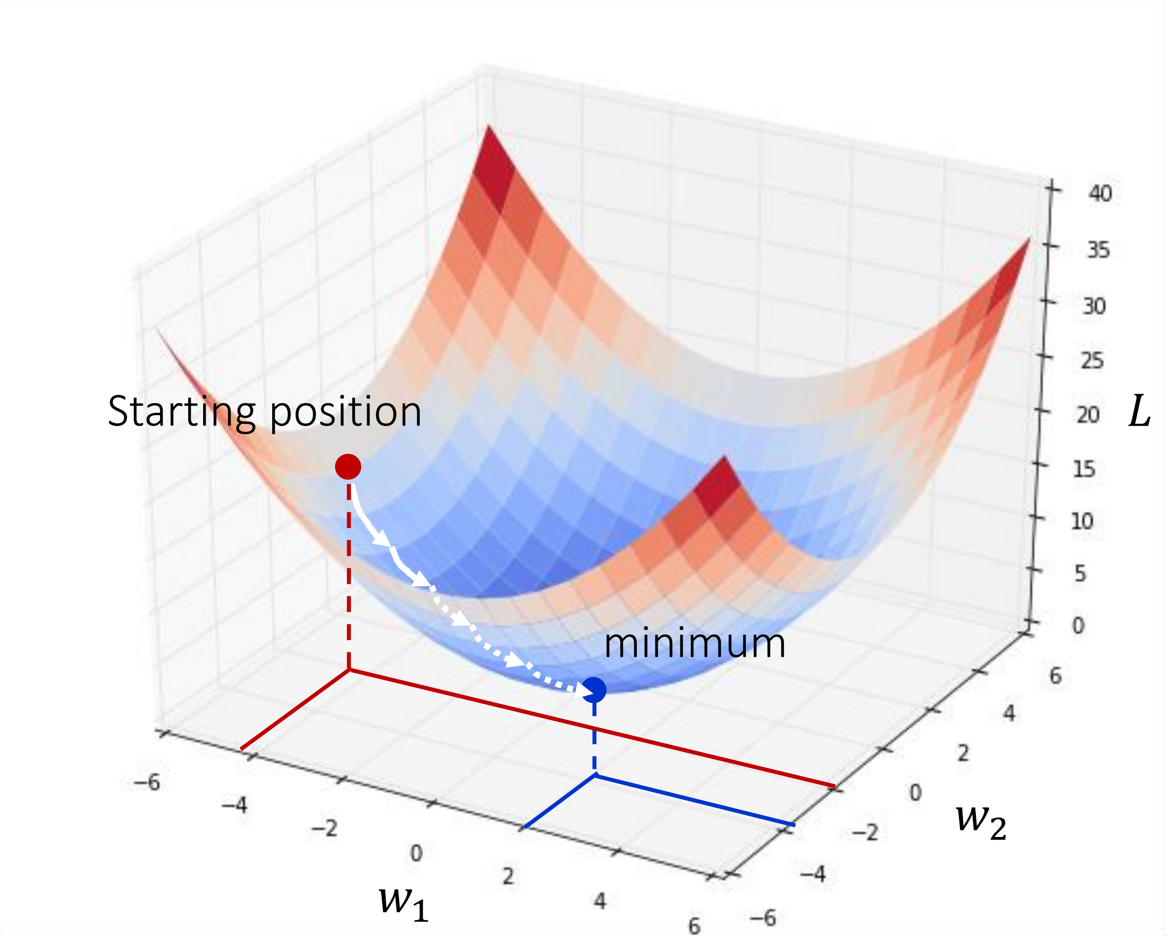
- This is a function of w because, in fact, the value of a depends on the value of the weights and biases
- This function also turns obvious that the performance of the network depends on the values of the weights and biases

➢ We need an algorithm that allows us to find the combination of weights and biases that minimize the value of $\boldsymbol{L(w)}$ $(\boldsymbol{L(\mathbf{w}) \approx 0})$

Gradient descent

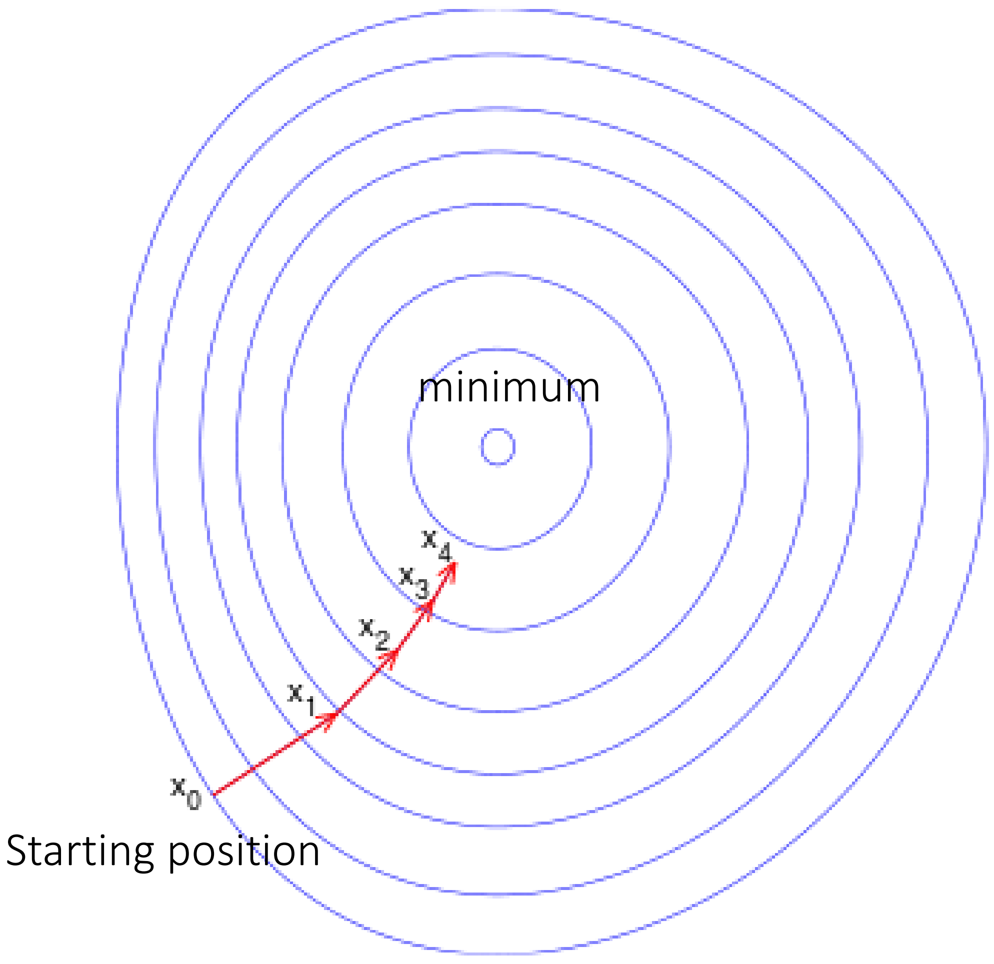
- Gradient descent is an algorithm that gradually changes a vector of parameters in order to minimize some function

- In our case, we want to gradually change the vector of weights and biases in order to minimize the loss function



Gradient descent

View from above ☺

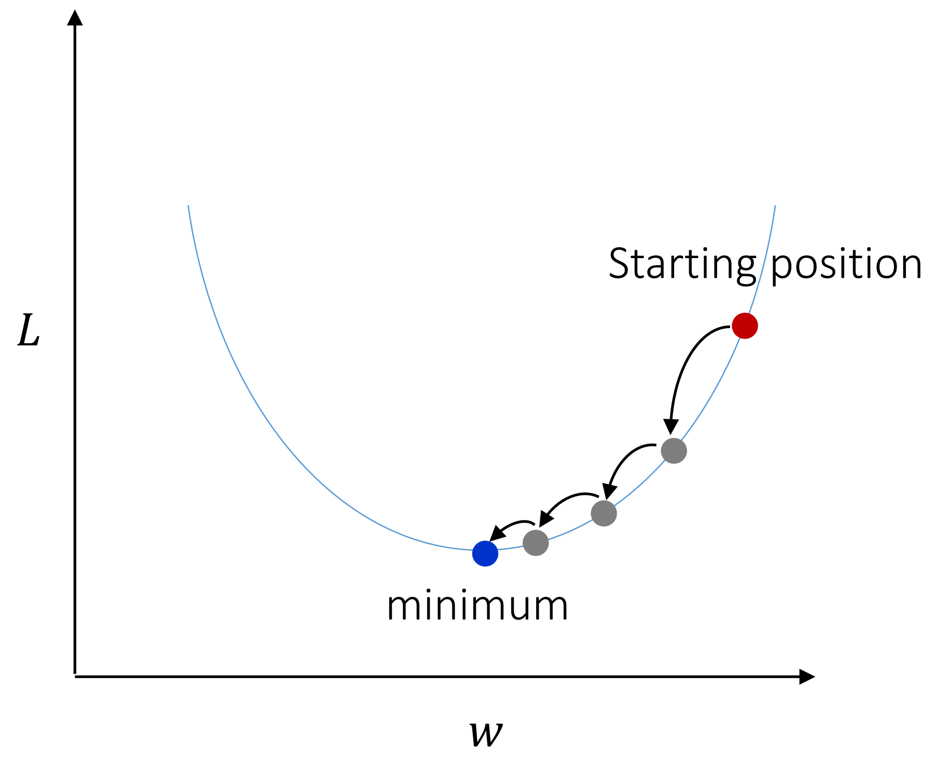


- Gradient descent is an algorithm that gradually changes a vector of parameters in order to minimize some function
- In our case, we want to gradually change the vector of weights and biases in order to minimize the loss function

Gradient descent

- Gradient descent is an algorithm that gradually changes a vector of parameters in order to minimize some function
- In our case, we want to gradually change the vector of weights and biases in order to minimize the loss function

If we had just one weight



Gradients and partial derivatives

revision

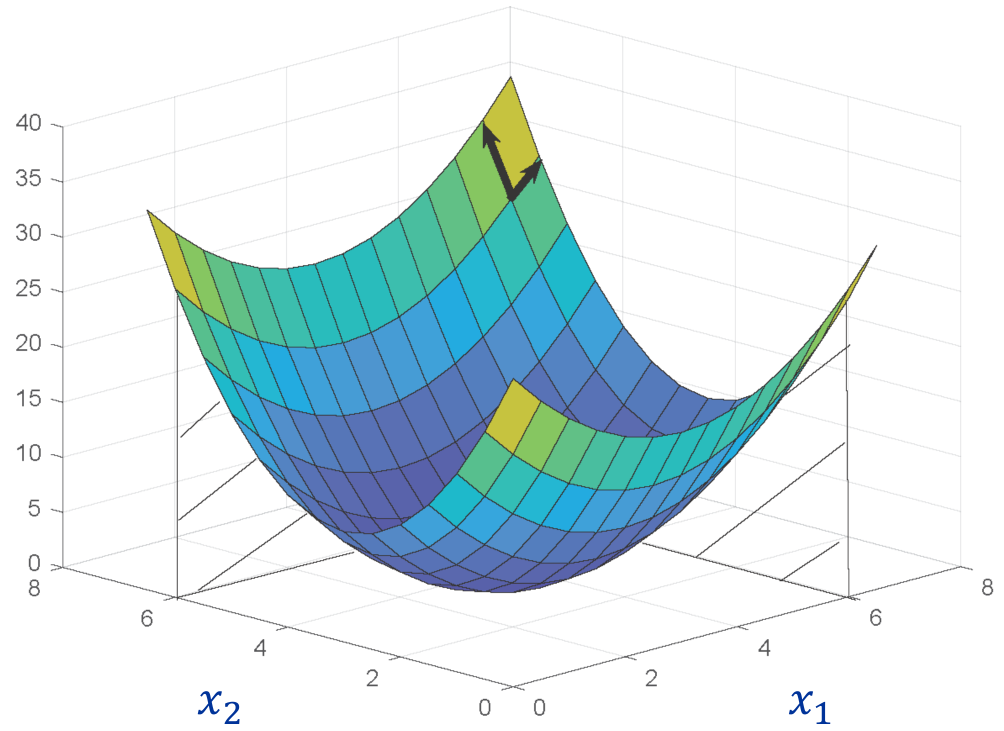
Gradients and partial derivatives

- Suppose we have function

$$y(x_1, x_2) = (x_1 - 3)^2 + 2(x_2 - 3)^2 + 1$$

depicted in the image

- Consider point $(x_1, x_2) = (6, 6)$
- The two arrows show the rate of change of y with respect to x_1 and x_2



Gradients and partial derivatives

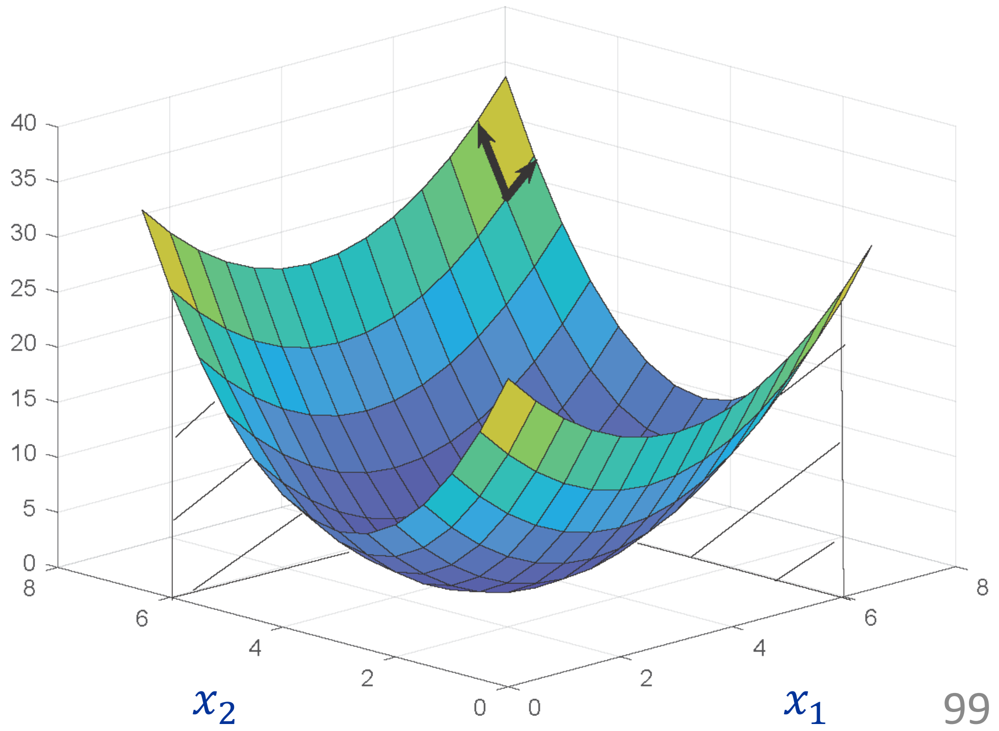
- If we are on the surface and traveling in the x_1 direction, we will climb at the rate given by the partial derivative $\frac{\partial y}{\partial x_1}$

- Partial derivative of y regarding x_1:

$$\frac{\partial y}{\partial x_1} = 2(x_1 - 3)$$

- For $x_1 = 6$, we have $\frac{\partial y}{\partial x_1} = 2(6 - 3) = 6$

- This slope is indicated by the short arrow



Gradients and partial derivatives

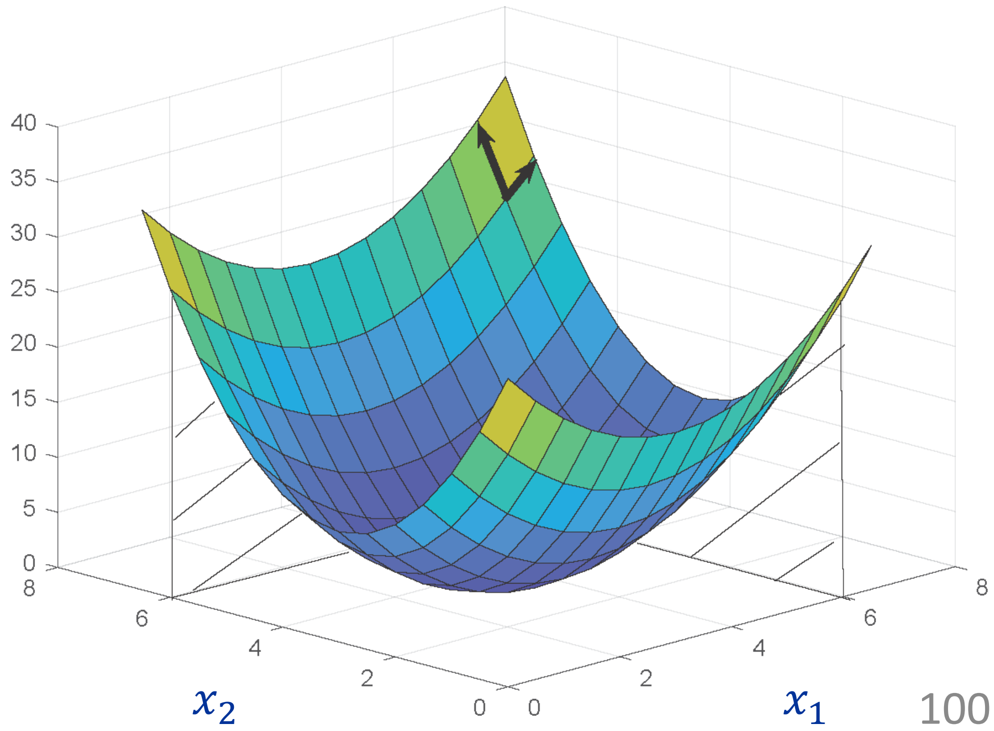
- If we are on the surface and traveling in the x_2 direction, we will climb at the rate given by the partial derivative $\frac{\partial y}{\partial x_2}$

- Partial derivative of y regarding x_2:

$$\frac{\partial y}{\partial x_2} = 4(x_2 - 3)$$

- For $x_2 = 6$, we have $\frac{\partial y}{\partial x_2} = 4(6 - 3) = 12$

- This slope is indicated by the longer arrow



Gradients and partial derivatives

- The gradient at some point is the vector of partial derivatives at that point

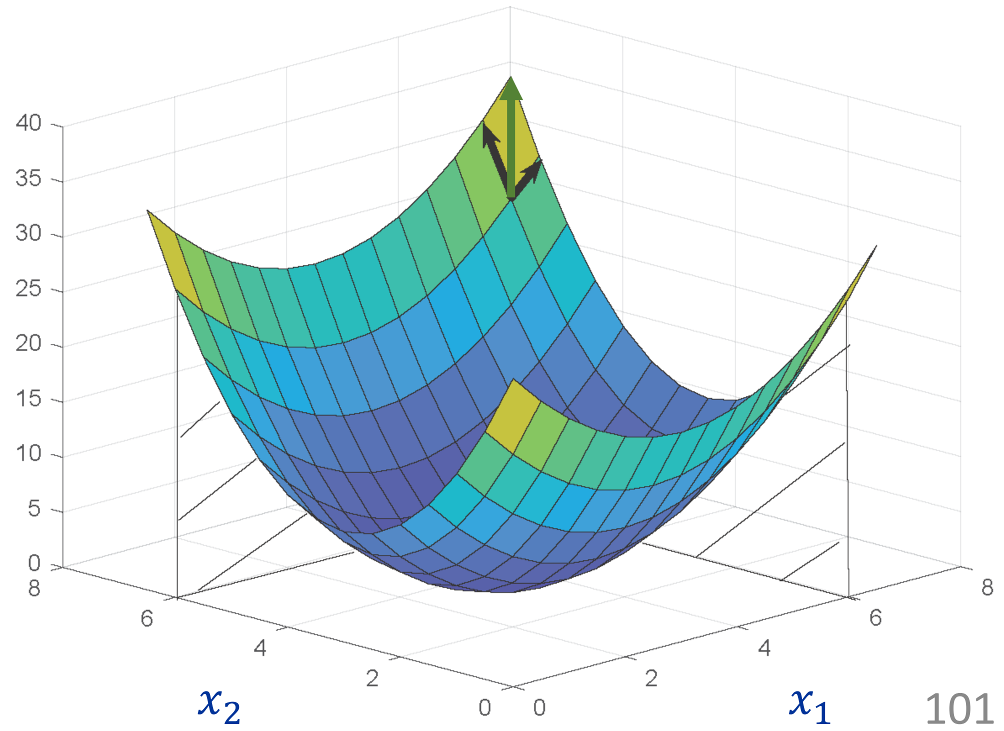
- So, the gradient at point

$$(x_1, x_2) = (6, 6)$$

is

$$\nabla y = [6, 12]$$

- It is represented by the green arrow



Gradients and partial derivatives

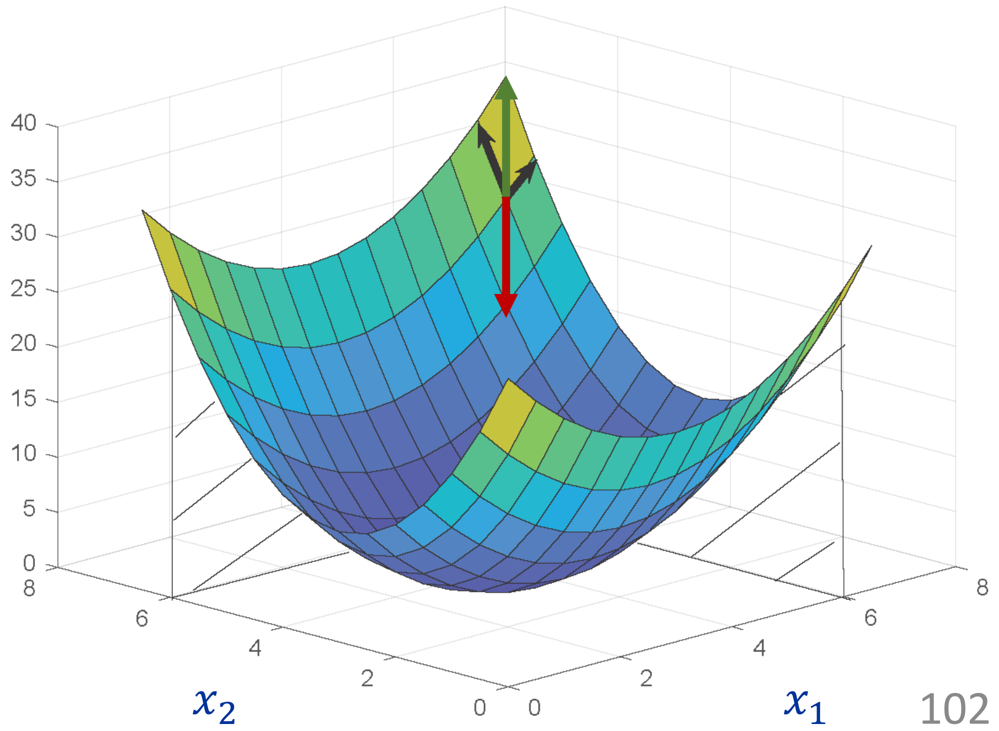
- The negative gradient at point

$$(x_1, x_2) = (6, 6)$$

is represented by the red arrow

...which corresponds to vector

$$-\nabla y = [-6, -12]$$



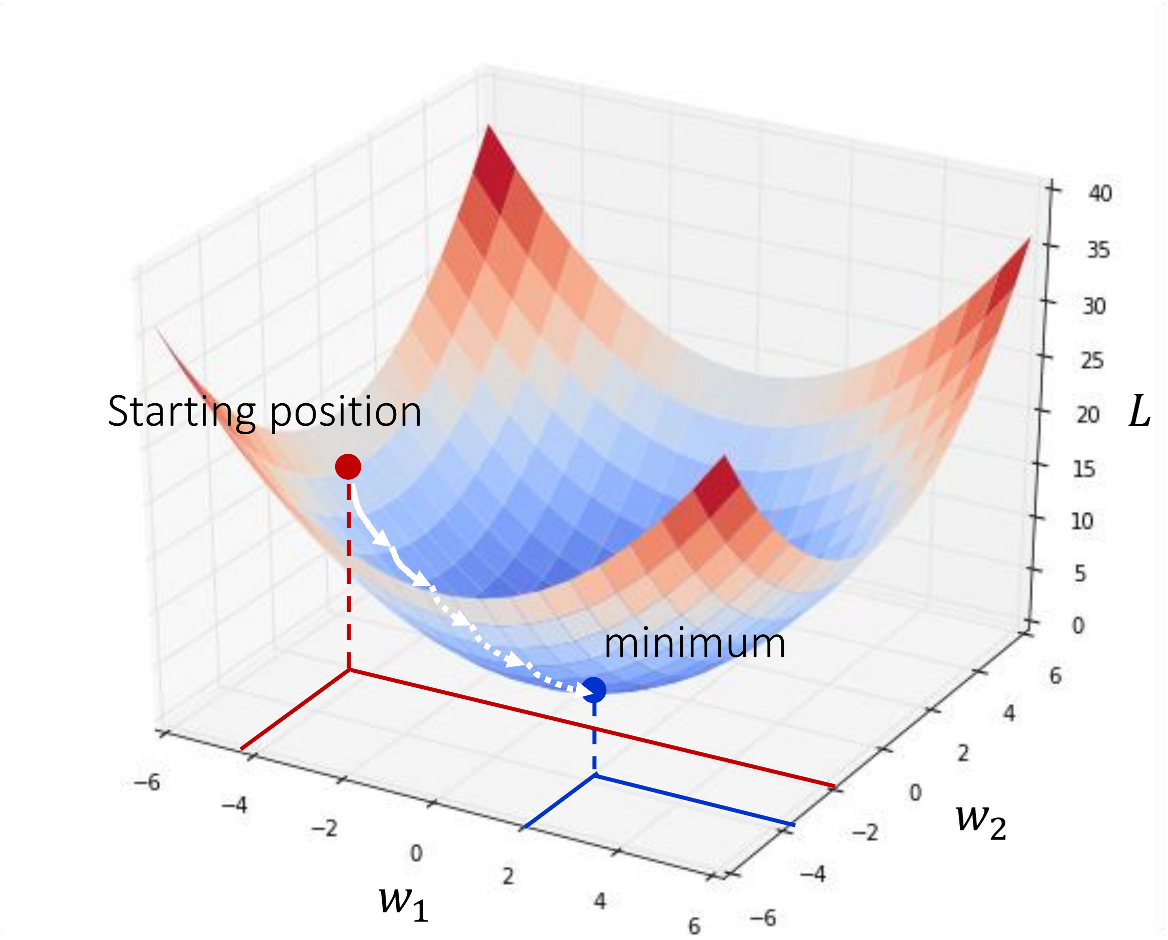
Gradients and partial derivatives revision

the end

Gradient descent

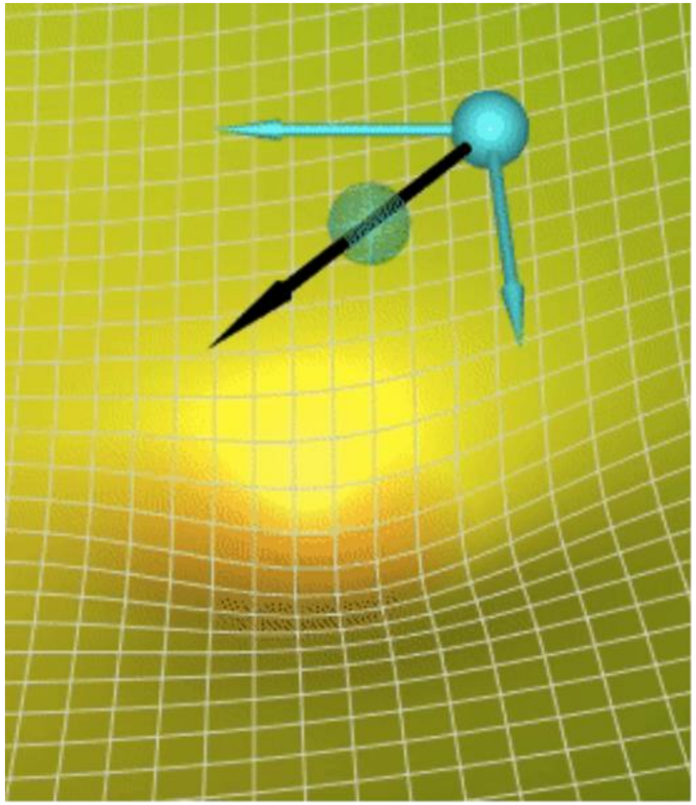
- Gradient descent follows the idea that the opposite direction of the gradient points to where the lower area is

- So, it iteratively takes steps in the opposite directions of the gradients

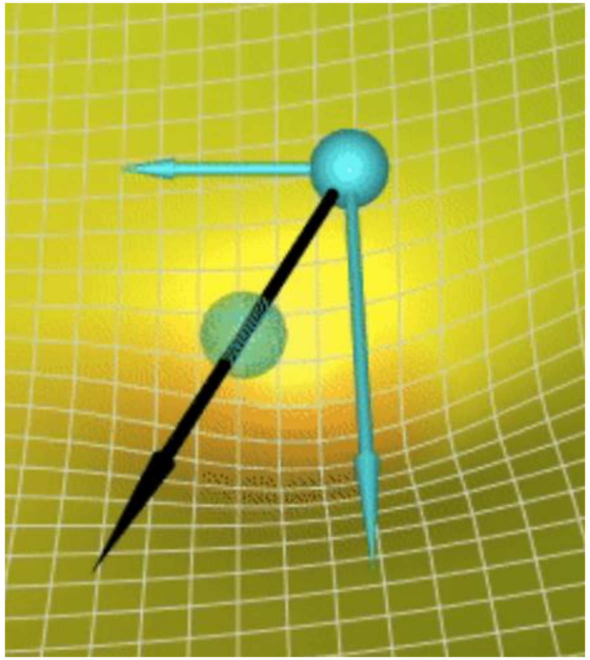


Gradient descent

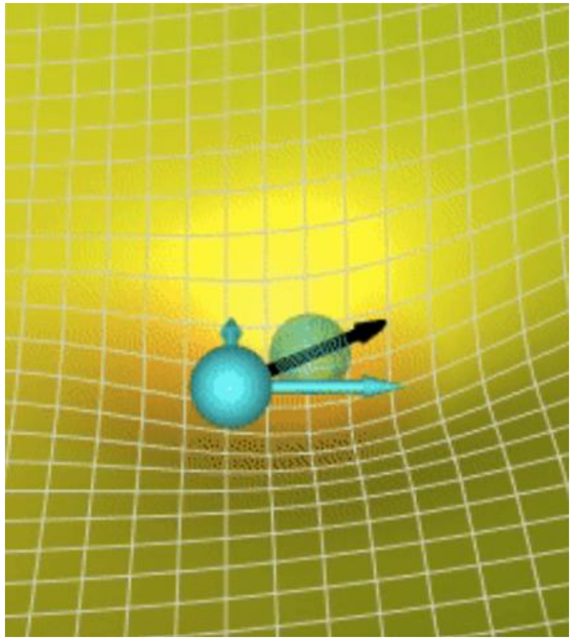
Iteration 1



Iteration 3



Iteration 5



Gradient descent

- It consists in doing changes to the weights/biases in the direction of the negative gradient of the loss function L

$$\Delta w = -\alpha \nabla L$$

α is a positive proportionality constant called the learning rate that is chosen by us

- That is,

$$\Delta w_{ij} = -\alpha \frac{\partial L}{\partial w_{ij}} \equiv w_{ij}(t+1) = w_{ij}(t) - \alpha \frac{\partial L}{\partial w_{ij}}$$

Putting it another way: it consists in changing each weight in an amount proportional to the partial negative derivative of the error function along that weight axis

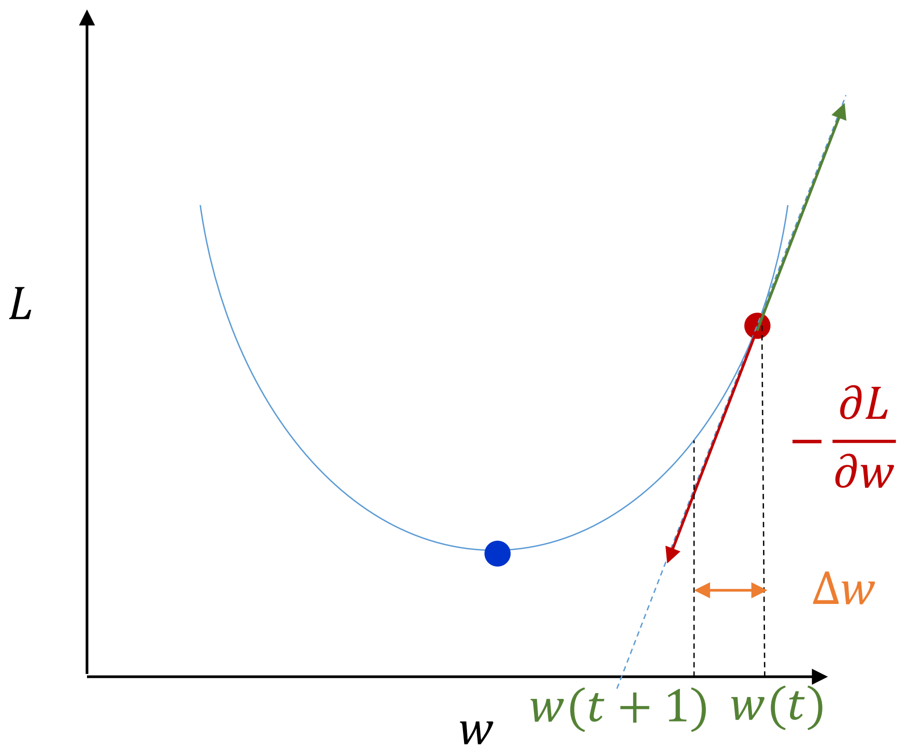
Gradient descent

$$w(t+1) = w(t) - \alpha \frac{\partial L}{\partial w}$$

$$\Delta w = -\alpha \frac{\partial L}{\partial w}$$

Note: If the value of w were in the left side of the curve, $\frac{\partial L}{\partial w}$ would be negative, hence, - $\frac{\partial L}{\partial w}$ would be positive, leading to $w(t+1) > w(t)$

If we had just one weight



Gradient descent algorithm

Initialize the neural network weights/biases

While stop condition not satisfied **do**

  Compute the output of the network for all samples

  Compute the gradient of the loss function ∇L

  Update the weights/biases using $\Delta w_{ij} = -\alpha \frac{\partial L}{\partial w_{ij}}$

Stop condition:

- maximum number of iterations reached
- step size is too small
- both

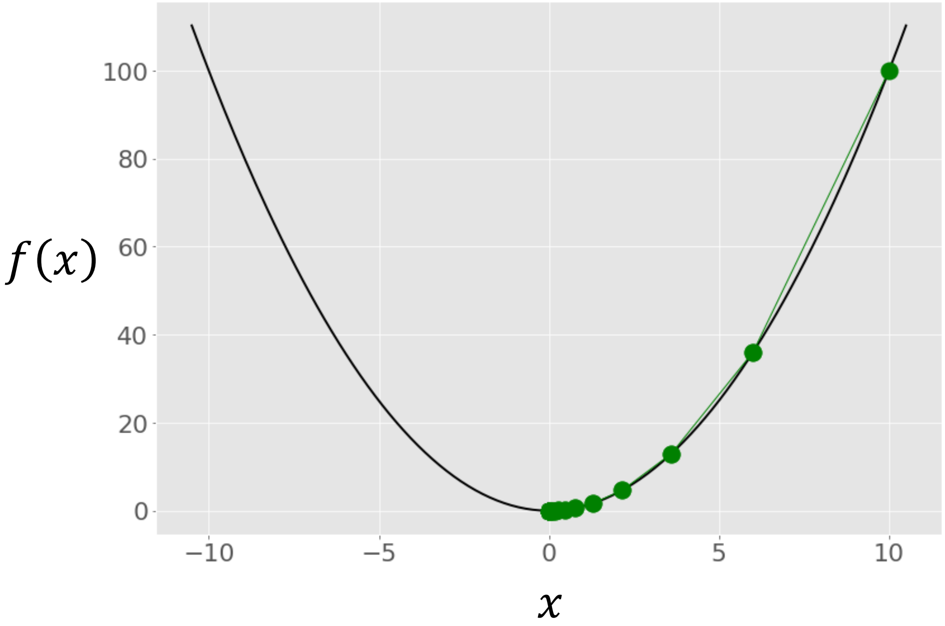
Gradient descent

- The function to be minimized must be differentiable and convex
- The learning rate has a strong influence on the performance of the algorithm:
  - The smaller the learning rate, the longer the algorithm takes to converge (is it?)
  - The algorithm may not converge (easily) to the global minimum if the learning rate is too large (it may even diverge from the minimum)

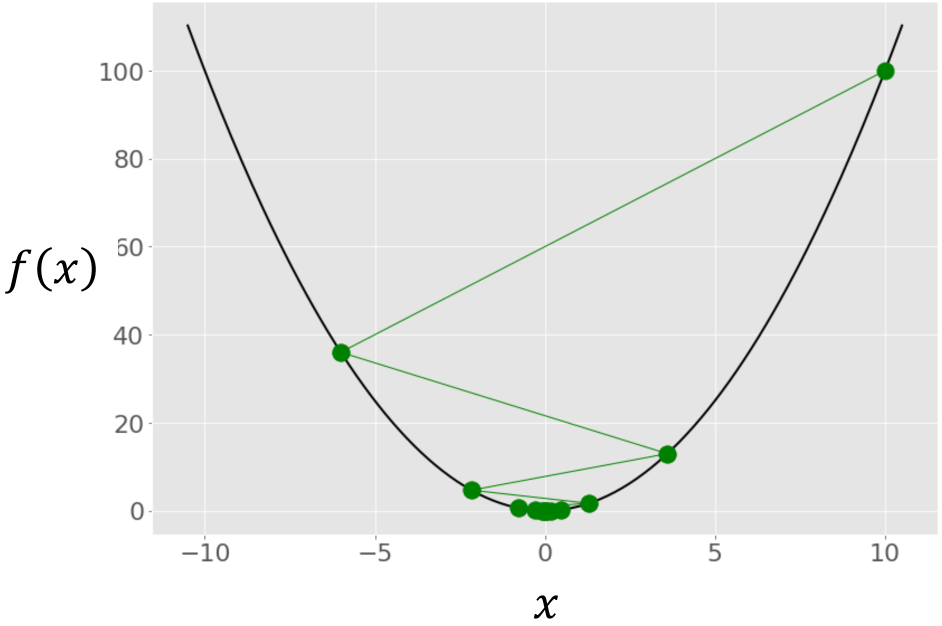
Learning rate

- Let us apply gradient descent to function $f(x) = x^2$

$\alpha = 0.2$



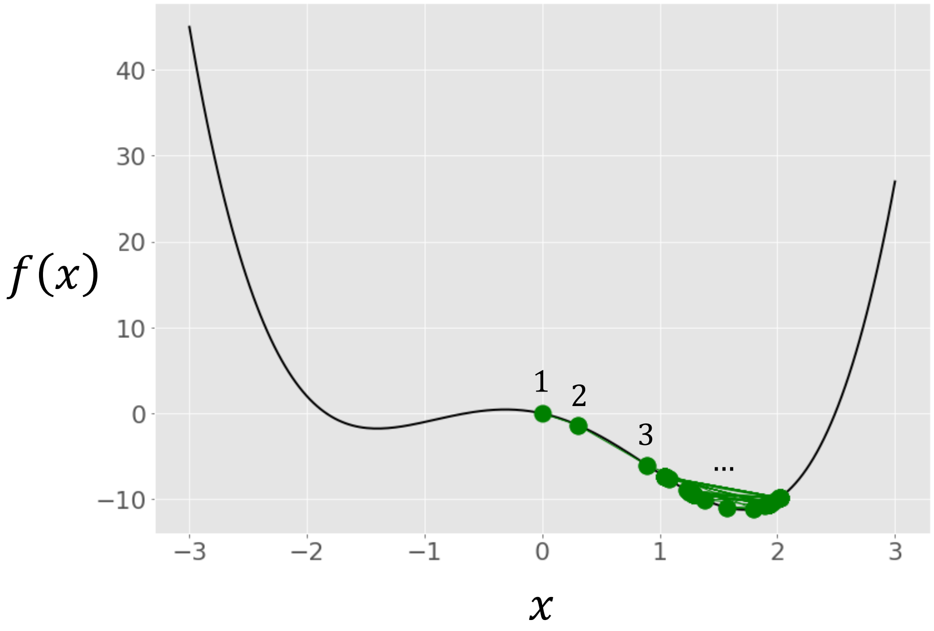
$\alpha = 0.8$



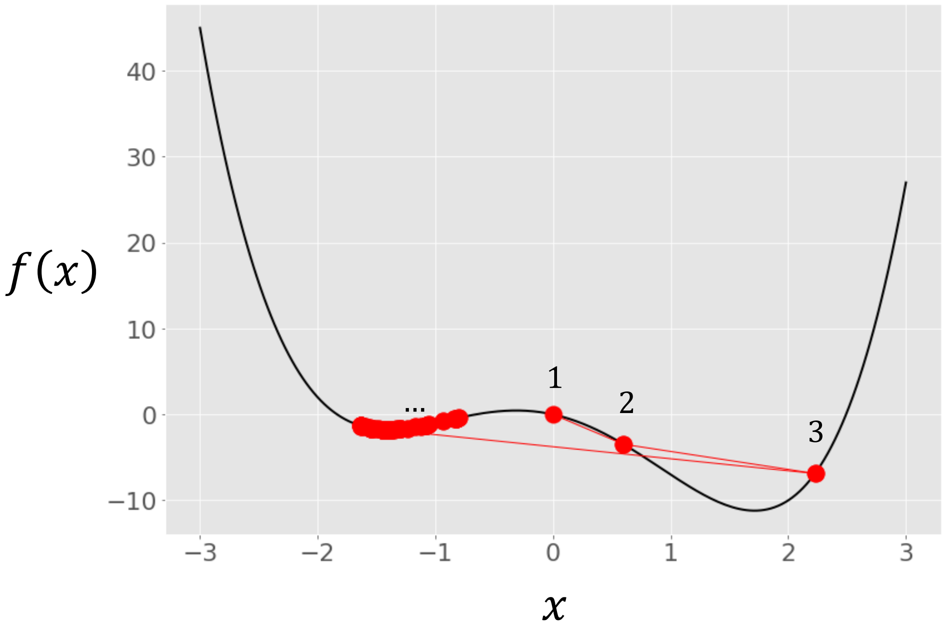
Non-convex functions

- Let us apply gradient descent to function $f(x) = x^4 + 5x^2 - 3x$

$\alpha = 0.1$



$\alpha = 0.2$



Gradient descent is heavy

- ...to compute the gradient ∇L, we need to compute the gradients ∇L_x separately for each training sample x and, then, average them:

$$\nabla L = \nabla\left(\frac{1}{n}\sum_{x} L_x\right) = \frac{1}{n}\sum_{x} \nabla L_x$$

➢ **This is computationally very heavy**

Stochastic gradient descent

- Instead of computing ∇L for all n samples, this is done for a small set of randomly chosen training samples
- This way, we can quickly get a good estimate of the true gradient ∇L
- Each set of samples is called a batch, and the size of this set is called the batch size

Stochastic gradient descent

- Stochastic gradient descent is the basis for most of the learning techniques used with neural networks

- Extremes:
  - If batch_size = n, we have the classical gradient descent
  - If batch_size = 1, we have the so-called online learning

Stochastic gradient descent algorithm

Initialize the neural network weights/biases

While stop condition not satisfied **do**

  Compute the output of the network for all samples

  Repeat

    Compute the gradient of the error function for batch_size randomly chosen samples (**selection without reposition**)

    Change the weights/biases using $\Delta w_{ij} = -\alpha \frac{\partial L}{\partial w_{ij}}$

  Until all samples have been used (this is called an epoch)

Computing gradients

- Until now, we didn't discuss how to compute the gradient of the loss function ∇L, or the weights update expression...
- In the next slides, we will do it, for one neural network of just one layer
- We will do it for just one sample, also
- We will derive the weights update general expression and then the expression for when MSE is used

Weights update expression

$$\Delta w_{ij} = -\alpha \frac{\partial L}{\partial w_{ij}}$$

$$w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}}$$

$$w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial w_{ij}}$$

$$\boxed{w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial a_i} f'(z_i) a_j}$$

General expression

$$L = \frac{1}{2} \sum_i (y_i - a_i)^2$$ quadratic error (MSE) for just one sample

$$a_i = f(z_i)$$ a_i -> output of unit i; f -> activation function

$$z_i = \sum_i w_{ij} a_j + b_i$$ z_i -> weighted sum for unit i; -> a_i input j of unit i

---

$$w_{ij} = w_{ij} - \alpha(-(y_i - a_i))\, f'(z_i) a_j$$

$$\boxed{w_{ij} = w_{ij} + \alpha (y_i - a_i) f'(z_i) a_j}$$

Expression when we use the MSE loss function

Weights update expression using MSE

- For reasons that will become clear ahead, we will rewrite

$$w_{ij} = w_{ij} + \alpha (y_i - a_i)\, f'(z_i)\; a_j$$

as

$$\boxed{\begin{gathered} w_{ij} = w_{ij} + \alpha \delta_i a_j \\ \text{where } \delta_i = (y_i - a_i)\, f'(z_i) \end{gathered}}$$

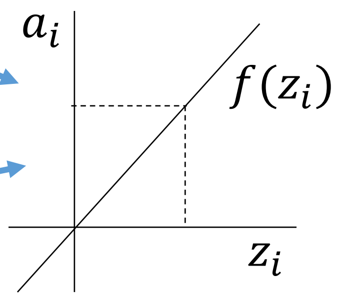
δ_i are usually called delta errors

Updating the last layer weights, example

- Let's assume that our units are purely linear, that is, the activation value is simply equal to the inputs weighted sum

- For a network of this type, the output of unit *j* is given by

$$a_i = f(z_i) = z_i = \sum_j w_{ij} a_j + b$$



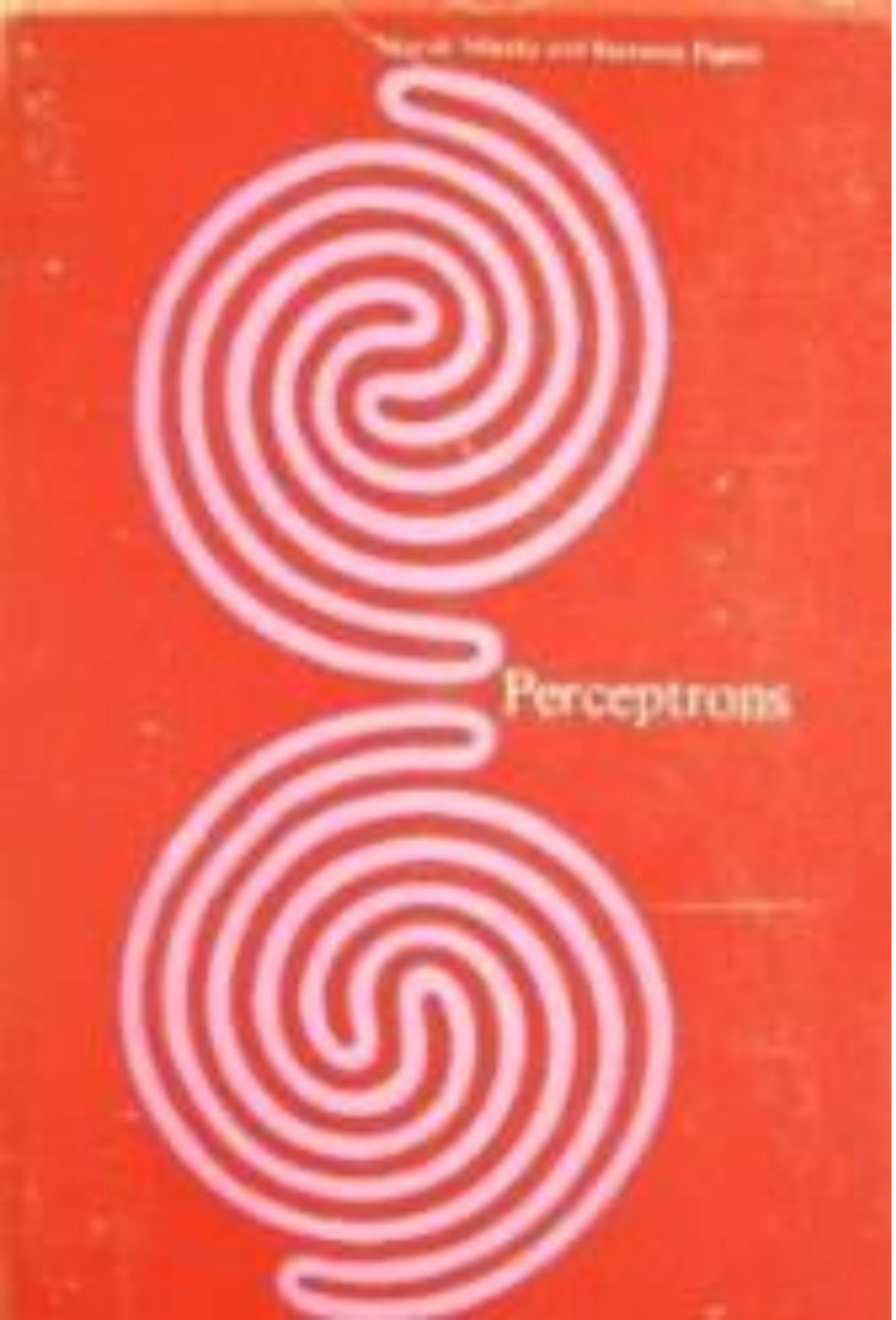
Identity activation function

- Since $f'(z_i) = 1$, we have

$$w_{ij} = w_{ij} + \alpha(y_i - a_i) f'(z_i) a_j = w_{ij} + \alpha(y_i - a_i) a_j$$

Backpropagation

- So far, we have the expression to update the weights when the network has just one layer
- What about networks with more than one layer?
- Whereas the error $y_i - a_i$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have



Backpropagation

- Minsky and Papert have shown in 1969 that a feed-forward network with two layers could solve many of the restrictions found until then, but presented no solution for the problem of adjusting the weights of the hidden layers

Backpropagation

- In 1986 Rumelhart, Hinton e Williams presented a solution for this problem (in fact, this solution was independently discovered before by other authors)
- The central idea of this solution is that the errors for the hidden layers units are computed backpropagating the errors of the output layer units

Backpropagation – output layer units

- For the output layer units, the expressions are the ones we have derived before for one layer networks:

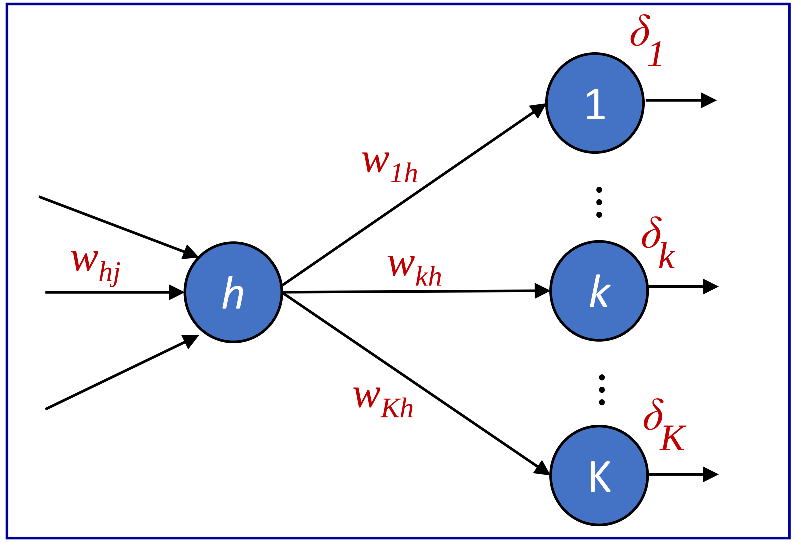
$$\delta_k = (y_k - a_k)\, f'(z_k)$$

$$w_{kh} = w_{kh} + \alpha \delta_k a_h$$

Note: h and k represent, respectively, the indexes of the last hidden layer and output layer units

Backpropagation – hidden layer units

- The idea is that hidden unit h is "responsible" for some fraction of the error δ_k in each of the next units to which it connects



Also, the "responsability" of unit h on the error of the next layer units is proportional to the weight of the connection between them

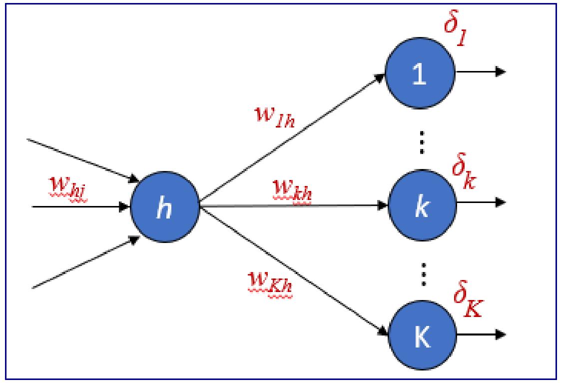
Backpropagation – hidden layer units

- Thus, the δ_k values are combined according to the strength of the connection between the hidden node and the output unit and are propagated back to provide the δ_k values for the hidden layer as follows

Note: *j* represents the index of one of the units of the hidden layer preceding the layer to which *h* belongs

$$w_{hj} = w_{hj} + \alpha \delta_h a_j$$

$$\delta_h = f'(z_h) \sum_k w_{kh} \delta_k$$



SGD algorithm with Backpropagation

for batch_size = 1

Initialize the neural network weights/biases

While stop condition not satisfied **do**

For each training sample **do**

Compute the output for the training sample

For each *output* unit k **do**

$$\delta_k = -\frac{\partial L_k}{\partial a_k} f'(z_k)$$

When we use MSE

$$\delta_k = (y_k - a_k) f'(z_k)$$

For each *hidden* unit h **do**

$$\delta_h = f'(z_h) \sum_k w_{kh} \delta_k$$

Backpropagation

Update each weight of the network

$$w_{i,j} = w_{i,j} + \alpha \delta_i a_j$$

Stochastic Gradient Descent (SGD)

SGD algorithm with Backpropagation

for batch_size = *m*

Initialize the neural network weights/biases

While stop condition not satisfied **do**

Repeat

Randomly select a $batch$ of $batch\_size = m$ samples (selection without reposition)

Compute the output of the network for all samples in the $batch$

For each training sample x in the $batch$ **do**

For each *output* unit k **do**

$$\delta_k = -\frac{\partial L_k}{\partial a_k} f'(z_k) \longrightarrow \delta_k = (y_k - a_k) f'(z_k)$$

When we use MSE

For each *hidden* unit h **do**

$$\delta_h = f'(z_h) \sum_k w_{kh} \delta_k$$

Update each weight of the network

$$w_{i,j} = w_{i,j} + \frac{\alpha}{m} \sum_x \delta_i^x a_j^x$$

In fact, we can opt for not dividing by m. Why?

Until all samples have been used (this is called an **epoch**)

Backpropagation – the algorithm

- Let us consider the sigmoid activation function

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

whose derivative is $$f'(z) = \frac{1}{1+e^{-z}} \times (1\text{-} \frac{1}{1+e^{-z}})$$

that is $$f'(z) = f(z) \times (1 - f(z))$$

Backpropagation – the algorithm

- So, the delta for an output unit *k* is given by

$$\delta_k = (y_k - a_k) \times f'(z_k)$$

$$\delta_k = (y_k - a_k) \times f(z_k) \times (1 - f(z_k))$$

$$\delta_k = (y_k - a_k) \times a_k \times (1 - a_k)$$

Desired output Current output

POLITÉCNICO DE LEIRIA

Backpropagation – the algorithm

- And, for the hidden layers units *h*, the error is given by:

Weight of the connection to destination unit *k*

Current output of unit *h*

Error for the destination unit *k*

$$\delta_h = f'(z_h)\sum_k w_{k,h}\delta_k = a_h \times (1 - a_h) \times \underbrace{\sum_k w_{k,h}\delta_k}_{\text{Forward connections}}$$

Backpropagation – the algorithm

- And the new weights:

Error for the destination unit

Current value

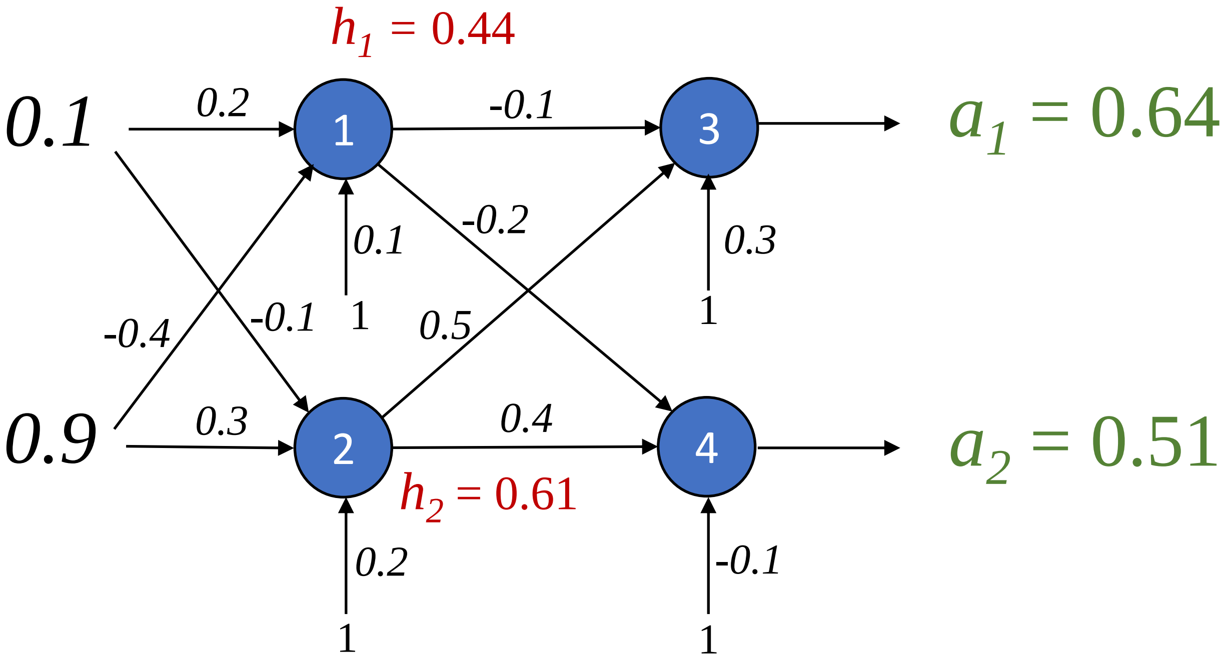
$$w_{i,j} = w_{i,j} + \alpha \delta_i a_j$$

Output value for the origin unit

Exercise

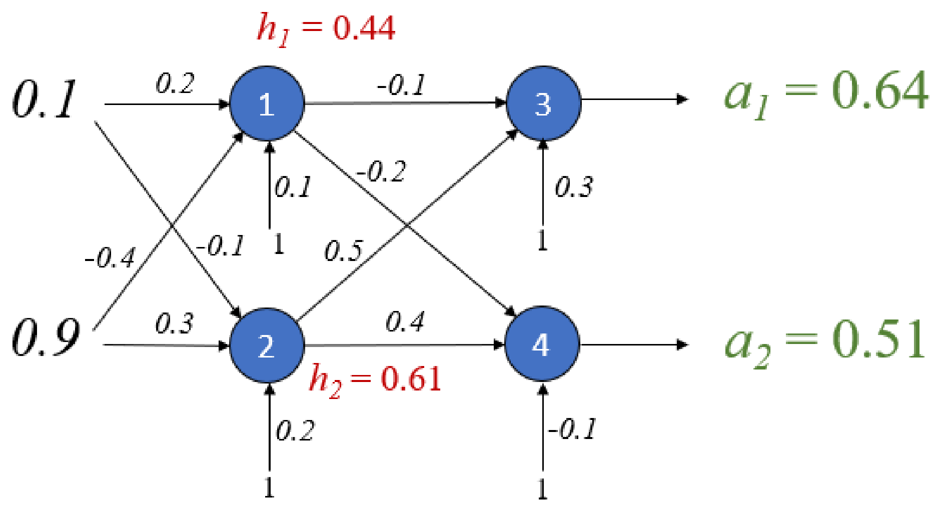
- Do you remember the neural network below? We have computed the network output for the input sample $[x_1, x_2] = [0.1, 0.9]$. We now want to update the weights values using the Backpropagation algorithm assuming that the desired output is $[y_1, y_2] = [1, 0]$

Remember:
Sigmoid activation function



Exercise - resolution

Delta errors:



δ_{a_1} = (1 – 0.64) × 0.64 × (1 – 0.64) = **0.083**

δ_{a_2} = (0 – 0.51) × 0.51 × (1 – 0.51) = **-0.127**

δ_{h_1} = 0.44 × (1 – 0.44) × (-0.1 × 0.083 + (-0.2) × (-0.127)) = **0.004**

δ_{h_2} = 0.61 × (1 – 0.61) × (0.5 × 0.083 + 0.4 × (-0.127)) = **-0.002**

Exercise - resolution

w_{h1x1} = 0.2 + 0.75 × 0.004 × 0.1 = 0.2003

w_{h1x2} = -0.4 + 0.75 × 0.004 × 0.9 = -0.3973

w_{h2x1} = -0.1 + 0.75 × (-0.002) × 0.1 = -0.10015

w_{h2x2} = 0.3 + 0.75 × (-0.002) × 0.9 = 0.29865

w_{a1h1} = -0.1 + 0.75 × 0.083 × 0.44 = -0.07261

w_{a1h2} = 0.5 + 0.75 × 0.083 × 0.61 = 0.5379

w_{a2h1} = -0.2 + 0.75 × (-0.127) × 0.44 = -0.24191

w_{a2h2} = 0.4 + 0.75 × (-0.127) × 0.61 = 0.34189

b_1 = 0.1 + 0.75 × 0.004 × 1 = 0.103

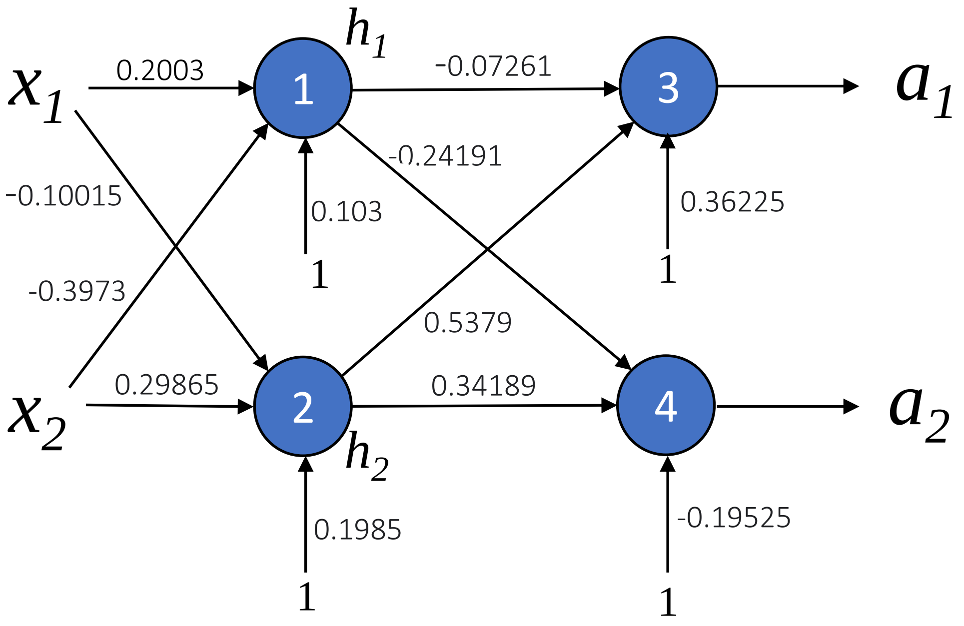
b_2 = 0.2 + 0.75 × (-0.002) × 1 = 0.1985

b_3 = 0.3 + 0.75 × 0.083 × 1 = 0.36225

b_4 = -0.1 + 0.75 × (-0.127) × 1 = -0.19525

Let us define $\alpha = 0.75$

Updated network



Gradient descent/backpropagation problems

- Long training periods: it can be due to an inadequate learning rate:

  - If we use a learning rate too small, the algorithm may take too long to converge

  - If we use a learning rate too high, the weights can oscillate, making it difficult the convergence to good solutions (weights close to the minimum error value)

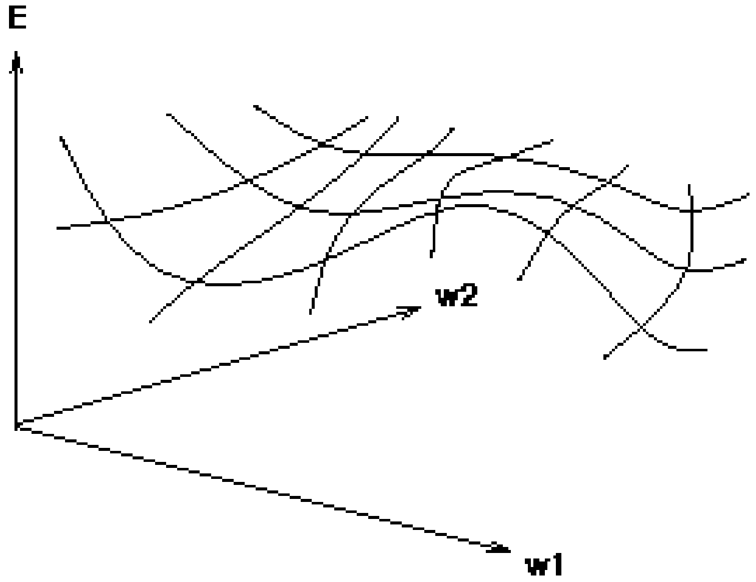
Gradient descent/backpropagation problems

- Units saturation:

  - As the training process advances, the weights can reach very high values
  - This can lead to very high weighted inputs sums (modulus) and, due to the usage of the sigmoid activation function, the units will have activation values very close to 0 or very close to 1
  - In these circumstances, the variation of weights, which is proportional to $a \times (1 - a)$, will be close to zero and the network doesn't learn
  - A way of trying to avoid this problem is to normalize the entries in the [0, 1] interval

Backpropagation problems

- Local minima: the error surface of a complex network has plenty o hills and valleys, and the network can be stuck in a local minimum, even if there is a nearby global minimum



Training neural networks – the training set

- Usually, it is not possible to train a neural network with all the examples of the universe of interest because it is often too large or infinite (e.g., the inputs/outputs are real numbers) or simply because we just have access to a subset of the universe

- In order to train the network, a subset of all the universe of possibilities is used. This subset, called the training set, is hoped to be representative of the function that the neural network is supposed to learn

Training neural networks – the test set

- After the training process, the neural network is tested with the so called test set (with no common examples with the training set)
- This is done in order to verify if the network is able to generalize beyond the examples used in the training process
- If the neural network has a good performance with the test set, the training process is finished
- If not, the training process must be repeated after something has been changed (the training set, the network architecture, the learning rate, etc.)

Neural networks applications

- Speech/image analysis (faces detection/identification, sentiment analysis)
- Natural Language processing (e.g. translation)
- Industrial control
- Financial analysis (e.g. stock market prediction)
- Medicine
- Robotics
- Pattern classification
- Non-linear control
- Etc. etc.

Final notes

- Neural networks are specially suitable for situations where inputs and outputs have continuous values

- They are quite resistant to noise

- They work as a “black box”: they do not provide any explanation for the output. This turns difficult the usage of knowledge to set up the network as, for example, setting the network’s architecture