

Artificial Intelligence

Neural Networks

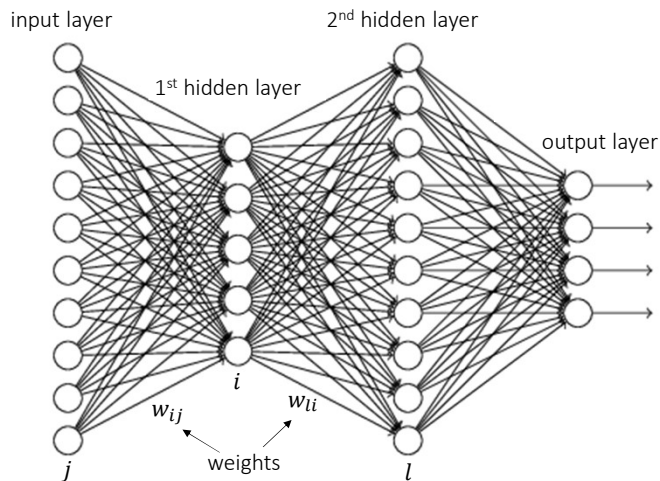


Slides by Carlos Grilo & Rolando Miragaia

(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

3

w_{ij} - the weight of the connection between unit j and unit i . By tradition (but not only because of that), the indexes appear in reverse order.

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

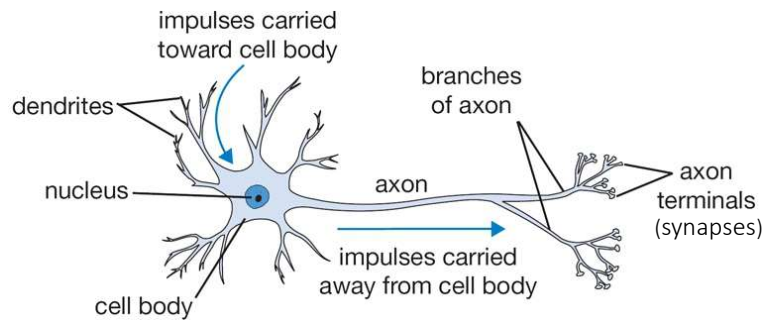
Dataset: set of **examples** used to train the neural network

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

The brain and the biological neuron

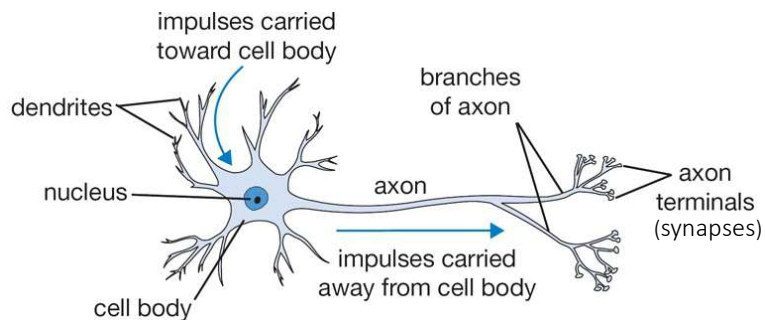
The brain

- 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**



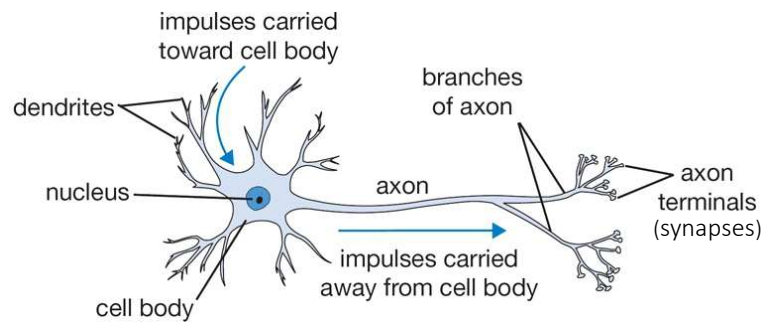
Dendrites connect to other neurons' **axons** through junctions called **synapses**

We can view **dendrites** 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

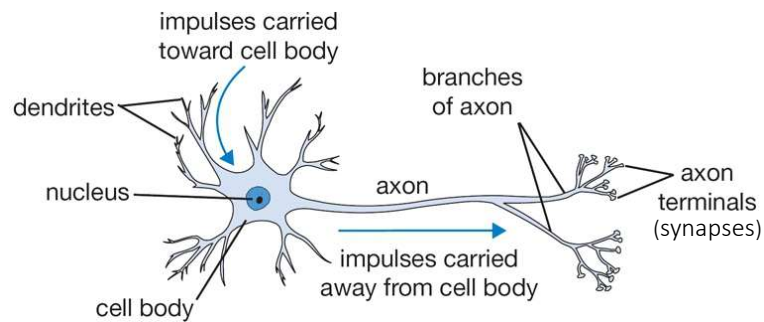
- Signals propagate between neurons through a complicated electrochemical reaction which leads synapses to produce chemical substances that enter through dendrites



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

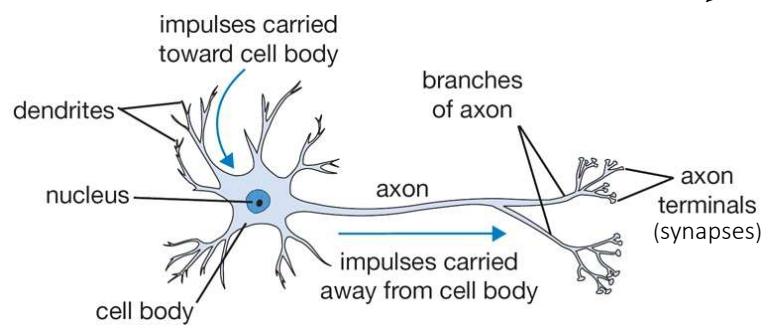
The biological neuron

- 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



The biological neuron

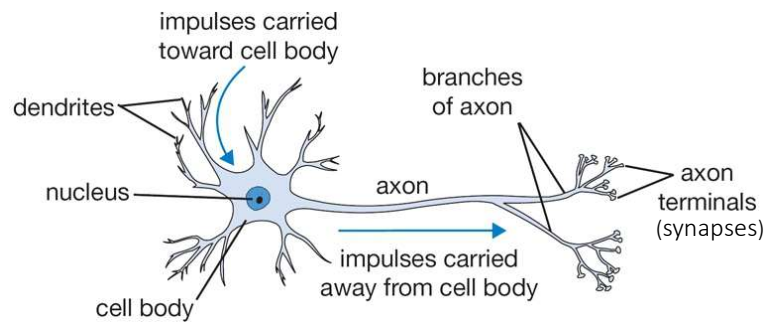
- It has been observed that often used **connections become stronger** and that **neurons sometimes form new connections** with other neurons



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



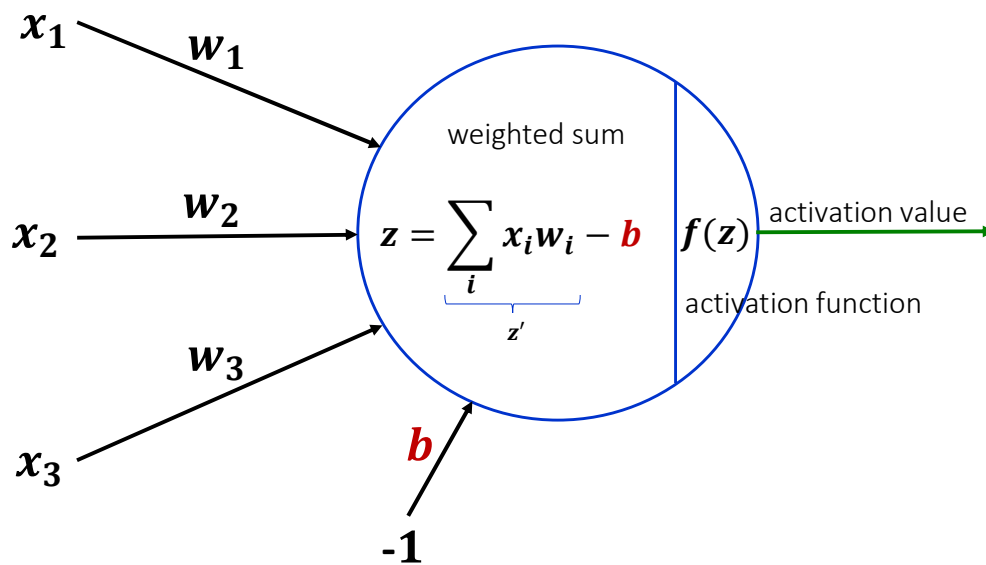
Computation is strongly
parallel and asynchronous

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:
 - First, a **weighted sum of the inputs** is computed
 - 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

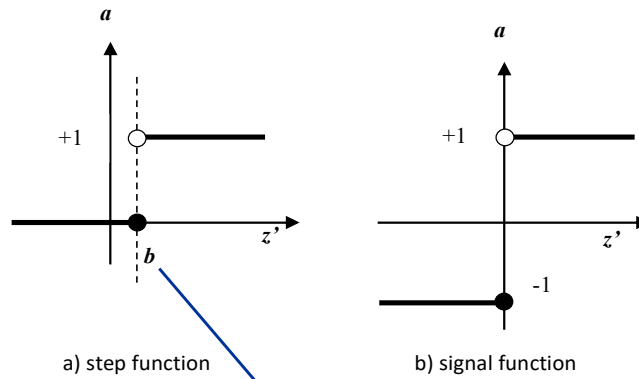
The artificial neuron



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



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

16

These are the functions that were used in the first invented neural networks and the ones we will first work with.

You may notice that both functions have the form of a step. Why do you think it is like this?

Neuron example

- Let us consider a neuron with two inputs and with the [step](#) activation function. So,

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

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

- 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

17

Ahead, we will see an alternative way of visualizing the bias effect.

Vector notation

- It is common to use vectors notation to represent the sets of inputs and weights
- Using this notation, we can say

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

Example:
 $[0, 1] \cdot [2, 3] + 4 = 7$
 $[x_1, x_2] \cdot [w_1, w_2] + b$

where x represents the inputs vector and w the weights vector

Dot product

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

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

- Example:

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

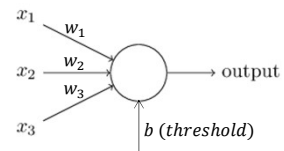
Neurons, weights, thresholds and decisions

- 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

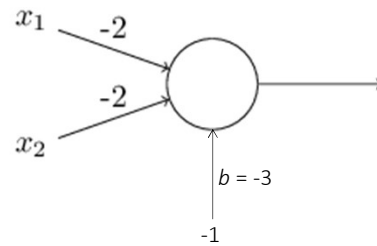
21

Dropping the bias (threshold) means you're more willing to go to the concert.

Building a decision model like this largely consists in defining the values of the weights, including the bias.

Another example

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



Assume:

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

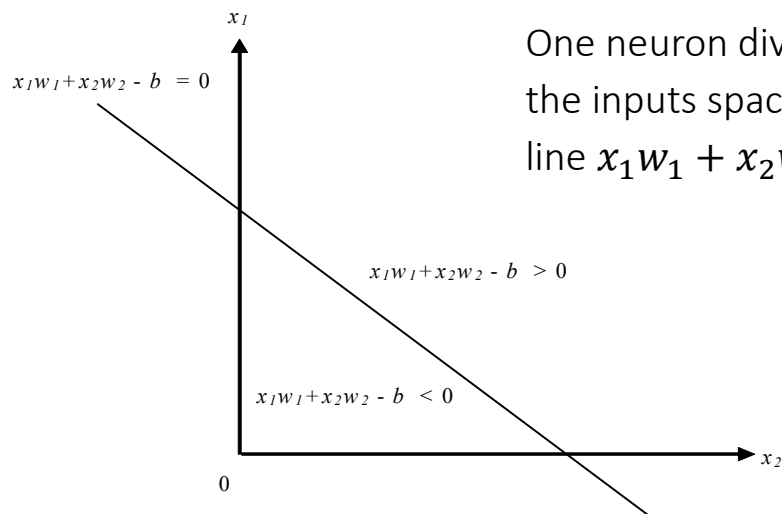
x_1	x_2	$x_1 \wedge x_2$	$\overline{x_1 \wedge x_2}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

- Exercise: confirm this

22

This slide's purpose is to show that neurons can also simulate (mathematical) functions (not just mundane decision rules as in the previous slide, although, in fact, a decision is still taken!...)

Limitations of a single neuron



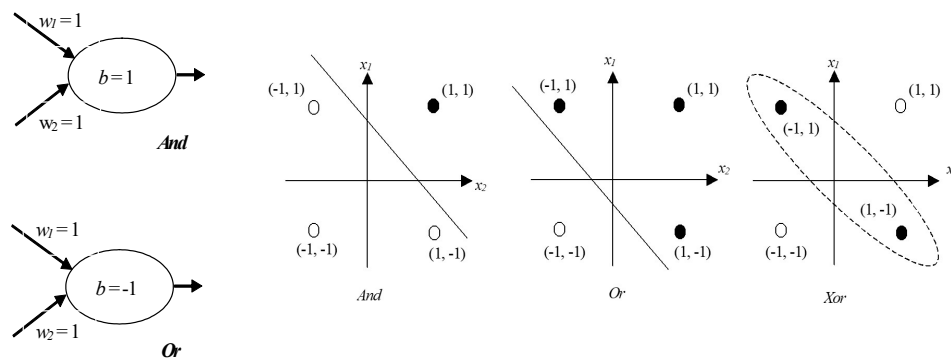
One neuron divides in two parts
the inputs space with the straight
line $x_1 w_1 + x_2 w_2 - b = 0$

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 logic *Or*, or a *NAND*, but it is impossible to represent a function as simple as the logic *Xor*:



25

How do we solve this limitation? Using multilayer networks (later...)

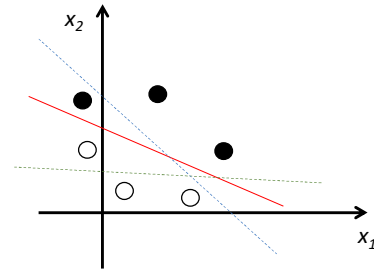
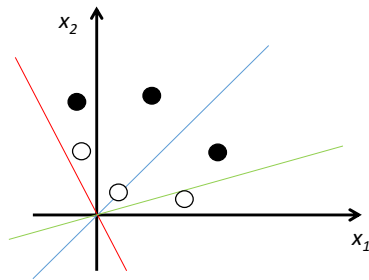
More on the effect of b

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 $x_1 w_1 + x_2 w_2 = 0$.

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

Learning rules

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

28

Until now we have seen how an artificial neuron works. Now we will be talking about how to train an artificial neuron so that it works as desired.

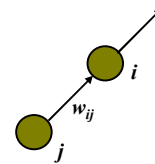
Hebb's rule

- **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

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

or

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



- γ 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

29

We remember that, in the most part of the neural networks literature, w_{ij} is interpreted as the weight of the connection that goes from unit j to unit i .

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 $t = T(x)$

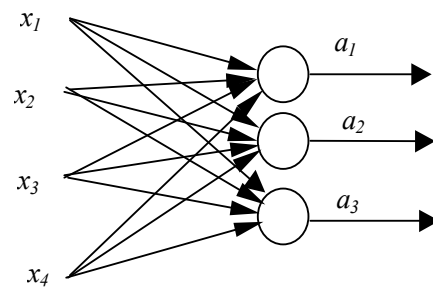
$$T: \{-1, 1\}^n \rightarrow \{-1, 1\}^m$$

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

30

feed-forward networks -> we will talk about them later

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

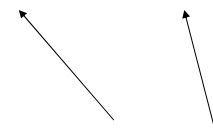
1. Initialize the weights randomly
2. **Do**
3. **For** each training example x from the training set **do**
4. Compute the output a of the network for input x
5. If $a \neq T(x)$ (the perceptron answers incorrectly), modify the connections weights (including b) as follows
$$w_i(\text{new}) = w_i(\text{current}) + \gamma(T(x) - a) x_i$$
where $0 < \gamma < 1$ represents the learning rate
6. **Until** $a == T(x)$ for all training examples

32

Usually, weights are initialized in the $[-1, 1]$ interval.

Perceptron training

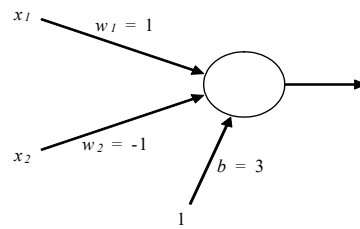
Error = desired output – current output

$$w_i(\text{new}) = w_i(\text{current}) + \gamma(T(x) - a) x_i$$


- 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

- 1st presentation of the training set

w_1 w_2 b
 current weights: [1, -1, 3]

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

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

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

$$w_1(\text{new}) = w_1(\text{current}) + \gamma \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$w_2(\text{new}) = w_2(\text{current}) + \gamma \times (t - a) \times x = -1 + 1 \times (-2) \times (-1) = 1$$

$$b(\text{new}) = b(\text{current}) + \gamma \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

We now present vector [-1, 1]:

current weights: [3, 1, 1]

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

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

Perceptron training - example

- 1st presentation of the training set

current weights: [3, 1, 1]

Presentation of [1, -1]:

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

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

$$w_1(\text{new}) = w_1(\text{current}) + \gamma \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

$$w_2(\text{new}) = w_2(\text{current}) + \gamma \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$b(\text{new}) = b(\text{current}) + \gamma \times (t - a) \times x = 1 + 1 \times (-2) \times 1 = -1$$

Presentation of [1, 1]:

current weights: [1, 3, -1]

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

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

- 2nd presentation of the training set

current weights: [1, 3, -1]

Presentation of [-1, -1]:

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

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

Presentation of [-1, 1]:

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

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

$$w_1(\text{new}) = w_1(\text{current}) + \gamma \times (t - a) \times x = 1 + 1 \times (-2) \times (-1) = 3$$

$$w_2(\text{new}) = w_2(\text{current}) + \gamma \times (t - a) \times x = 3 + 1 \times (-2) \times 1 = 1$$

$$b(\text{new}) = b(\text{current}) + \gamma \times (t - a) \times x = -1 + 1 \times (-2) \times 1 = -3$$

Perceptron training - example

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

Presentation of [1, -1]:

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

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

Presentation of [1, 1]:

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

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]:

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

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

Presentation of [-1, 1]:

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

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

Perceptron training - example

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

Presentation of [1, -1]:

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

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

Presentation of [1, 1]:

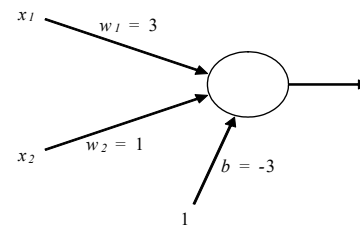
$$a = \text{signal}(3 \times 1 + 1 \times 1 + -3 \times 1) = \text{signal}(3 + 1 - 3) = \text{signal}(1) = 1$$

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

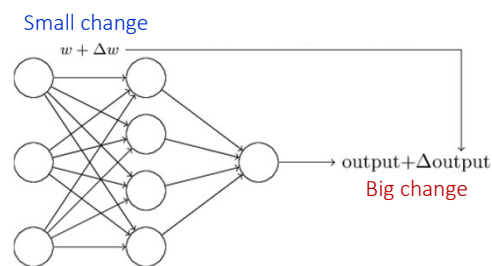
- Therefore, we can stop the training process. The resulting perceptron is:



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

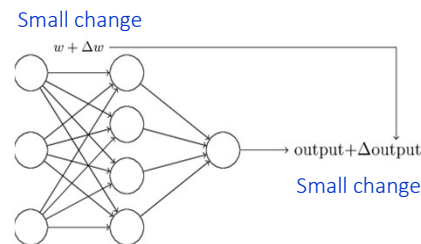
Discontinuous activation functions

- 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

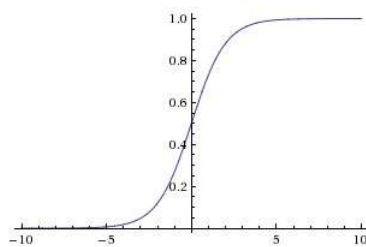
- This turns more difficult the training process of the network
- We would expect that a **small change in the weights and bias** would lead to a **small change in the output**



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

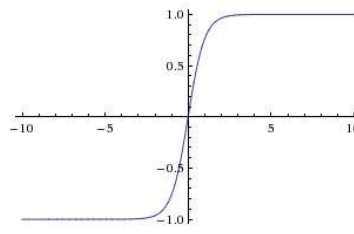
- We can overcome this problem by introducing **continuous activation functions**

Common (continuous) activation functions



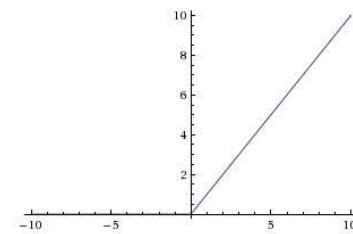
Sigmoid

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



Tanh

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



ReLU

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

46

The ReLU function is not differentiable but it allows the effect described in the previous slides. It is often used in hidden layer units in multilayer networks.

Why do we need to use non-linear functions as activations functions, in particular, step like ones? Besides the answer you have given for the step and signal functions (because we want to simulate decision processes), another reason is that the world is full of nonlinear phenomena and we want neural networks to be able to simulate those phenomena also.

Ask me about the softmax function, which is not in the slide 😊

Discontinuous activation functions

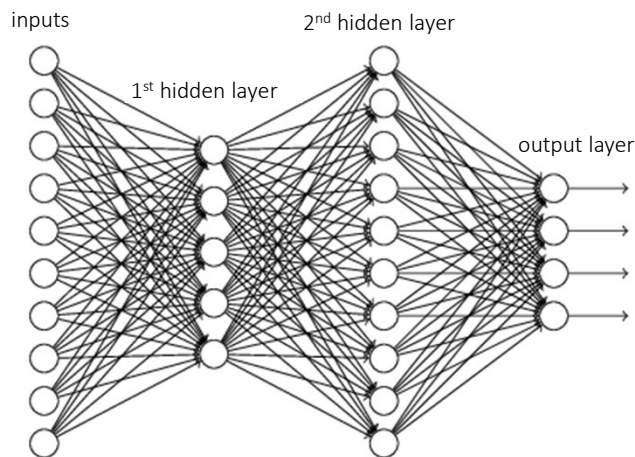
- In false/true (binary) problems, how do we interpret the result of activation functions like the sigmoid function, for example?
- If the function outputs a number equal or larger than 0.5, it is interpreted as a 1, else, it is interpreted as 0

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

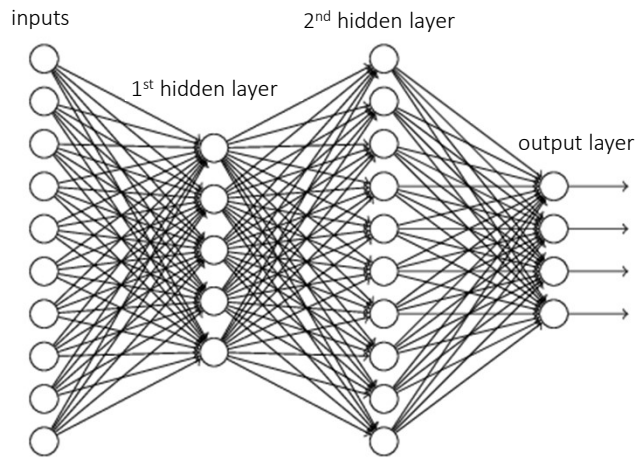
They are also called **fully connected** or **dense** neural networks

50

“Fully connected” or “dense” because there are connection between all the units of one layer and the units of the next layer.

Note: The “MLP” designation, often used nowadays, can be misleading because, in fact, the units are not perceptrons (which use the signal or de step activation functions). Usually, in these networks the sigmoid, tanh, ReLU, softmax and other continuous functions are used.

Feed-forward networks



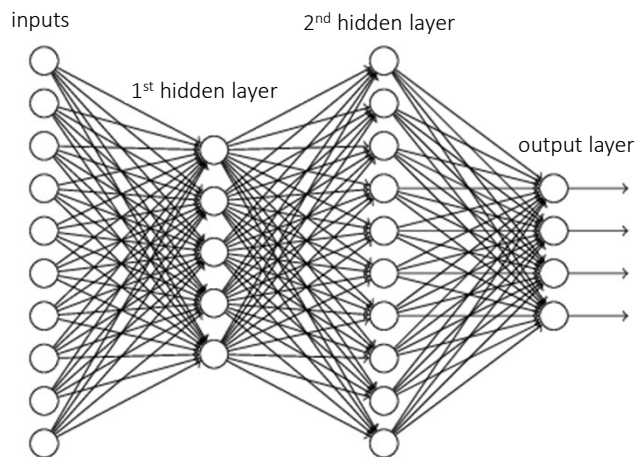
In these networks, there are multiple hidden layers and each layer is densely connected to its previous layer

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

51

Note: Some authors consider that inputs layer as a layer of the network, while others don't.

Feed-forward networks



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

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

Feed-forward networks

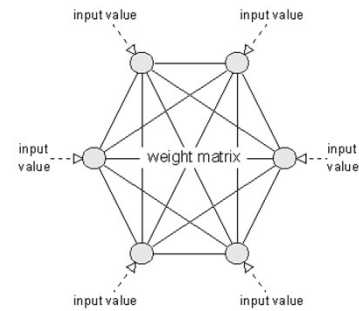
- A feed-forward network may not have hidden layers
- This simplifies the learning problem, but it implies, as already seen, strong limitations on what can be represented by the network
- With **one** sufficiently large hidden layer, it is possible to represent any continuous function
- With **two**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

53

“As already seen” because that is the case we have seen with perceptron networks, where we have only one layer. In such cases, each unit works independently from the others, hence, we have the same limitations that one single neuron has. In fact, we have a set of neurons instead of a network...

Hopfield networks

- Hopfield networks are an example of recurrent networks that have been extensively studied
- All the network's units are both input and output neurons and the connections are bidirectional
- The signal function is used as activation function



Hopfield networks

- This type of network works as an associative memory
- For example, if the training set consists in a set of photos and if a small part of one of these photos is latter presented to the network, the activation levels of the network should reproduce the photo to which the fragment presented belongs
- It is able to memorize $0.138n$ examples, where n is the number of neurons

How to put neural networks to work?

- 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 the 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

57

In some networks other things may be changed besides the weights, as for example, the number of neurons (not common)

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

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?

61

Second question answer: while there are some heuristics for doing it, this is usually done, fundamentally, through trial and error.

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

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

64

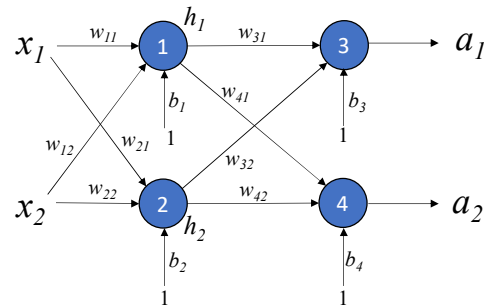
In classification problems, typically a vector **One-Hot Encoding**

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

Only slides for supervised learning. Unsupervised and reinforcement learning are explained in the class.

Multilayer neural networks



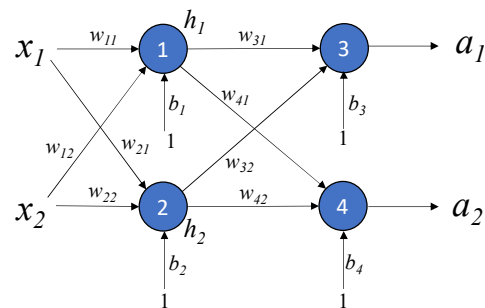
As we have seen before, these networks are often called **Multi-Layer Perceptron networks (MLP)**

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

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

Students should be aware that different layers may have a different number of units

1st 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

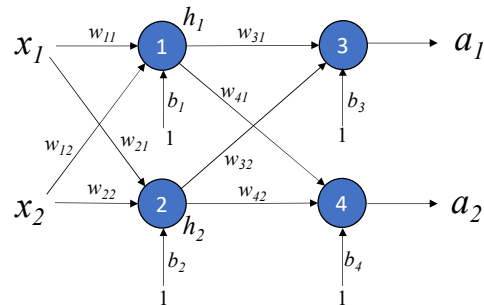
$z_i \rightarrow i^{\text{th}}$ unit weighted sum

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

67

Here, we can see why “the weight of the connection from unit j to unit i is represented as w_{ij} ”: often, matrices are represented as w_{lc} , where l is the line and c is the column. That is, the weights for one unit are in a line that multiplies by the inputs array.

2nd layer output computation



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

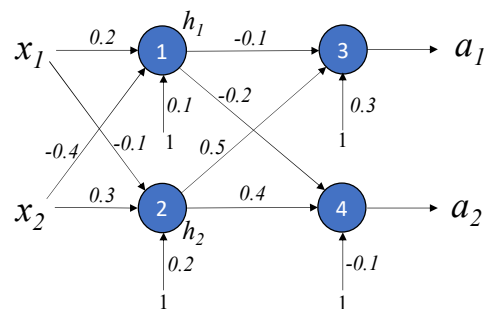
$f(z) \rightarrow$ activation function

$z_i \rightarrow$ ith unit weighted sum

$a_i \rightarrow$ ith output unit activation value

Exercise

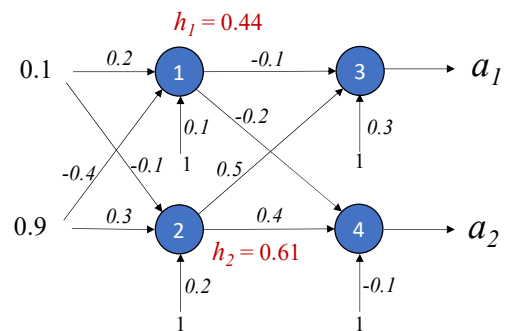
- Compute the output of the network below when the input sample is $x = [x_1, x_2] = [0.1, 0.9]$. Assume that the sigmoid function is used



Exercise - solution

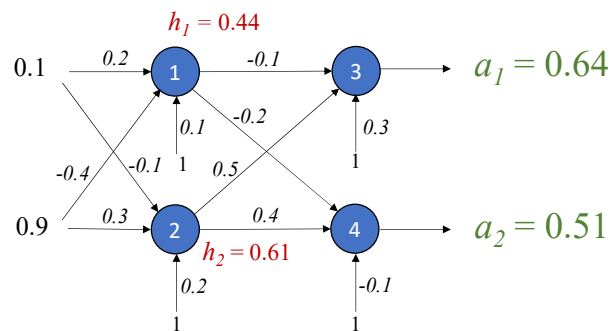
▪ $h_1 = f(0.1 \times 0.2 + 0.9 \times (-0.4) + 0.1) = f(-0.24) = \frac{1}{1+e^{-(-0.24)}} = 0.44$

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



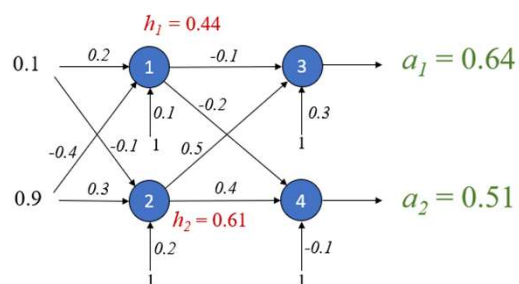
Exercise - solution

- $a_1 = f(0.44 \times (-0.1) + 0.61 \times 0.5 + 0.3) = f(0.561) = \frac{1}{1+e^{-0.561}} = 0.64$
- $a_2 = f(0.44 \times (-0.2) + 0.61 \times 0.4 - 0.1) = f(0.056) = \frac{1}{1+e^{-0.056}} = 0.51$



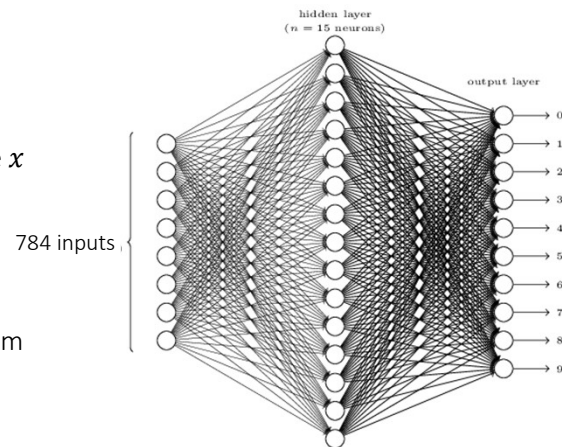
Exercise – solution using vector notation

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



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



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**

74

Math revision: How to compute the difference between two vectors?

Math revision: How to compute the norm $\|v\|$ of a vector v ?

Try to understand the formula for a neural network with two output units, for example, and compute the value of L , given vectors y and a .

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

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

- This is a function of \mathbf{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 $L(\mathbf{w})$ ($L(\mathbf{w}) \approx 0$)

76

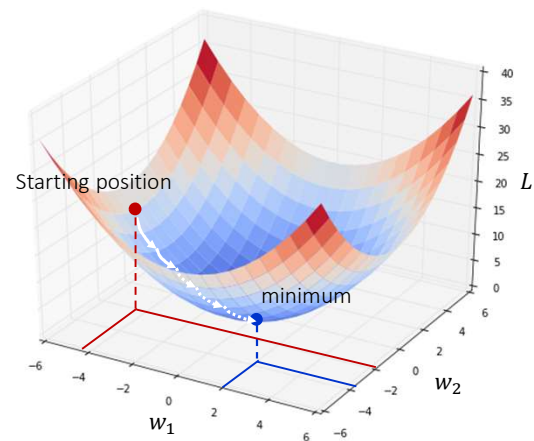
Regarding the first sentence, usually, the $L(\mathbf{w})$ is not expressed as a function of the weights to keep it simple (imagine if we had a lot of weights, which is what usually happens, actually!)

Note: After a well succeeded training process, the value of the function should be almost 0 because $y(x) - a$ are almost 0 for all x .

This formula corresponds to a procedure where we first compute the output for all the samples of the dataset and, then, we compute the loss based on the desired and actual outputs for all those samples. We will later see other possibilities (we don't need to do it for ALL samples always!)

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

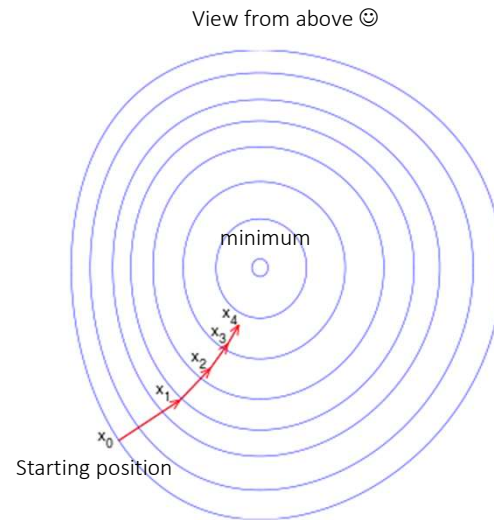


77

In a function as the one depicted in the slide, it would be easy to compute the minimum analitically. However, usually, the loss function is far more complicated (with several vales, ridges and local mínima), which turns impossible to compute the global minimum that way. Besides, it usually has many many parameters, not just two!

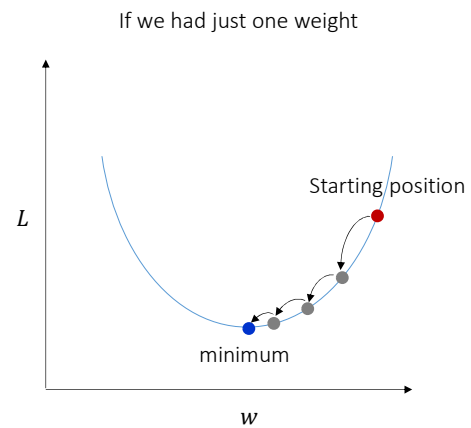
Gradient descent

- 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

- 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

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

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

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

- That is,

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

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

80

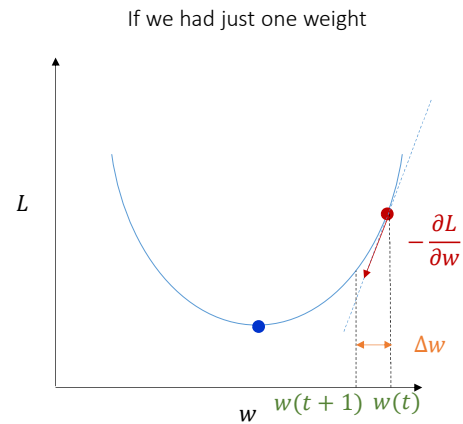
VL is the vector of partial derivatives of L

Gradient descent

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

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

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 $w(t+1) > w(t)$



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

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)

83

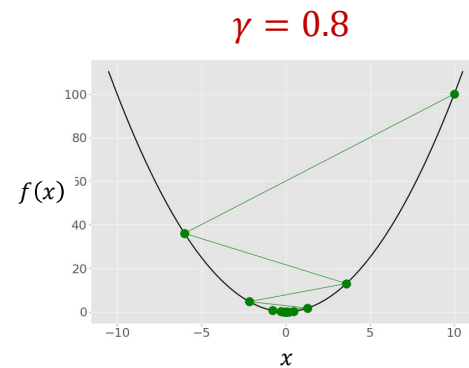
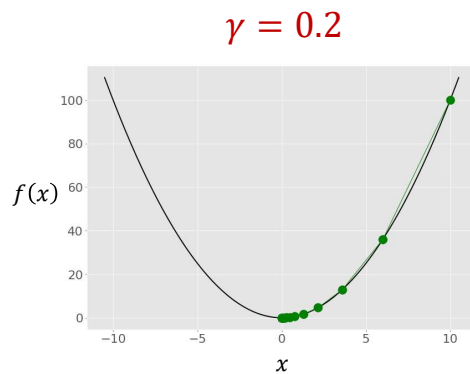
In practice, gradient descent is applied to non-convex functions as well (and even non-differentiable functions, ReLU, for example).

It may happen that the algorithm takes more time to converge with larger learning rates because it can oscillate around the minimum (see next slide).

“The rule doesn't always work - several things can go wrong and prevent gradient descent from finding the global minimum of L , a point we'll return to explore in later chapters. But, in practice gradient descent often works extremely well, and in neural networks we'll find that it's a powerful way of minimizing the cost function, and so helping the net learn.”

Learning rate

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



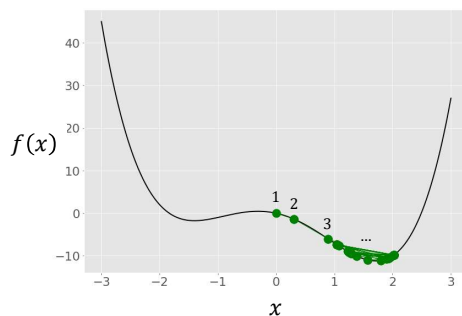
84

This slide shows that a larger learning rate doesn't lead necessarily to a faster convergence.

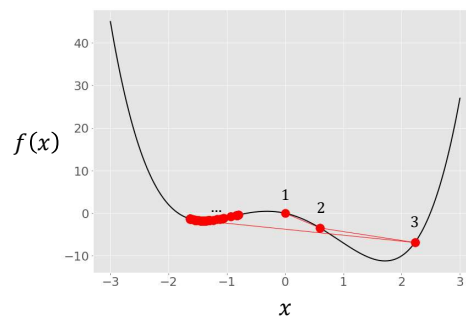
Non-convex functions

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

$\gamma = 0.1$



$\gamma = 0.2$



85

This slide shows that 1) we may not find the minimum when the function is non-convex and 2) a larger learning rate isn't necessarily better than a smaller one (in this case, it leads the algorithm to converge to a local minimum instead of the global one).

But, you should be aware that there might be situations where we need larger learning rates to escape from local minima. Only through experimentation we can "discover" the best value. It is common that algorithms start with a larger value in the initial phase of the training process and then go smaller during the process.

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

86

We can do this because “the derivative of the sum is equal to the sum of the derivatives”.

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**

87

The choice of the batch size value is up to us.

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`

88

n -> number of samples of the dataset

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

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

Updating the last layer weights

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

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

or

$$w_{ij} = w_{ij} + \gamma \delta_i a_j$$

$$\text{where } \delta_i = (y_i - a_i) f'(z_i)$$

δ_i are usually called **delta errors**

90

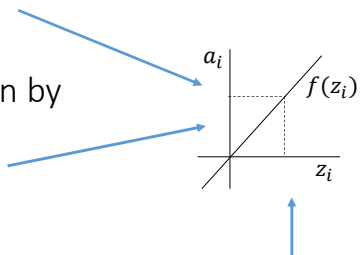
In order to confirm that $-(y_i - a_i) f'(z_i) = \frac{\partial L}{\partial z_i}$, it is enough to derive the expressions in the previous slides in a different order. This is left as an exercise.

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

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

91

This is the simpler example we can have.

Curiously, this is the expression used to update weights in the perceptron algorithm!...

Later, we will compute the expression for the sigmoid activation function.

Backpropagation

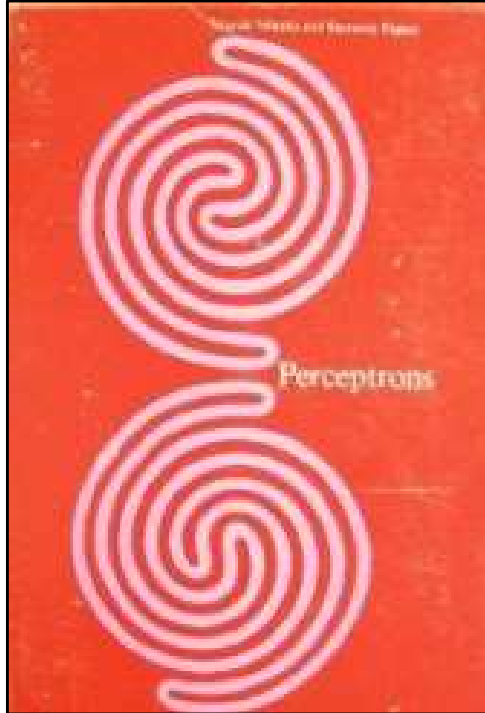
- So far, we have the expression to update the weights when the network has just one layer
- What about networks with more than one layer?
- Whereas the error $y_i - a_i$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the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

92

In order to understand last sentence, we need to be aware that for output units, we are able to compute $y_i - a_i$, that is, we know what is the error in the unit whose weights we want to update ($w_{ij} = w_{ij} + \gamma \delta_i a_j$, where $\delta_i = (y_i - a_i) f'(z_i)$)

This is not the case for hidden layer units.

So, the question is: how can we compute the error for hidden layer units?



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

93

This is one of the most well known books in the AI history and one that had a strong impact (search for AI Winter).

Wikipedia (AI Winter):

“... However, one type of connectionist work continued: the study of [perceptrons](#), invented by Frank Rosenblatt, who kept the field alive with his salesmanship and the sheer force of his personality.^[11] He optimistically predicted that the perceptron “may eventually be able to learn, make decisions, and translate languages”.^[12] Mainstream research into perceptrons came to an abrupt end in 1969, when [Marvin Minsky](#) and [Seymour Papert](#) published the book [Perceptrons](#), which was perceived as outlining the limits of what perceptrons could do. Connectionist approaches were abandoned for the next decade or so. While important work, such as [Paul Werbos](#)’ discovery of [backpropagation](#), continued in a limited way, major funding for connectionist projects was difficult to find in the 1970s and early 1980s.^[13] The “winter” of connectionist research came to an end in the middle 1980s, when the work of [John Hopfield](#), [David Rumelhart](#) and others revived large scale interest in neural networks.^[14] Rosenblatt did not live to see this, however, as he died in a boating accident shortly after *Perceptrons* was published.”

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:

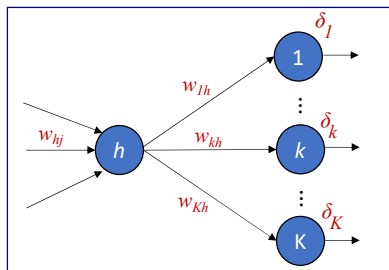
$$\delta_k = (y_k - a_k) f'(z_k)$$

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

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

96

From the first sentence, we take that the expression for updating weight w_{jh} should consider the errors of all the units of the next layer.

From the second sentence, we take that each one of these errors δ_k should be somehow combined with the weight of the connection that connects unit h to unit k .

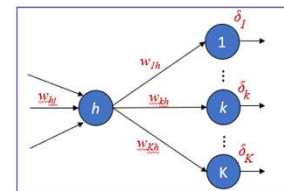
Backpropagation – hidden layer units

- 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_h 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

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

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



97

This is one of the most important formulas of the AI history because it is this formula that allows multilayer networks to learn.

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**

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

For each hidden unit h **do**

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

 Update each weight of the network

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

Backpropagation

Stochastic
Gradient (SGD)
Descent

98

“The term backpropagation strictly refers only to the algorithm for computing the gradient, not how the gradient is used; however, the term is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by 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**

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

For each hidden unit h **do**

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

Update each weight of the network

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

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

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

99

We divide by m because we are using MSE as loss function (average of the losses obtained with several samples). In the derivation of the L gradient, $1/m$ never appears. In the derivation that we have done before, this is not evident because it was done for one sample only (because it is simpler and because it is usually explained like that) and, hence, we never divide the gradient expression by the number of samples (m) in each batch.

In fact, it is indifferent to divide or not, because m is just a constant. If we opted for not dividing by m , we would have to use a smaller learning rate in order to have a similar effect, just that.

Backpropagation – the algorithm

- Let us consider the [sigmoid](#) activation function

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

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

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

Backpropagation – the algorithm

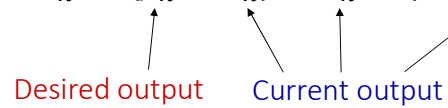
- So, the delta for an output unit k is given by

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

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

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

Desired output Current output



The diagram shows two labels at the bottom: 'Desired output' in red and 'Current output' in blue. Arrows point from these labels to the terms in the equation $\delta_k = (y_k - a_k) \times a_k \times (1 - a_k)$ above. Specifically, an arrow points from 'Desired output' to y_k , another from 'Current output' to a_k , and a third from 'Current output' to $(1 - a_k)$.

Backpropagation – the algorithm

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

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

Current output of unit h (points to a_h)
 Weight of the connection to destination unit k (points to $w_{k,h}$)
 Error for the destination unit k (points to δ_k)
 Forward connections (points to the summation symbol \sum_k)

Backpropagation – the algorithm

- And the new weights:

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

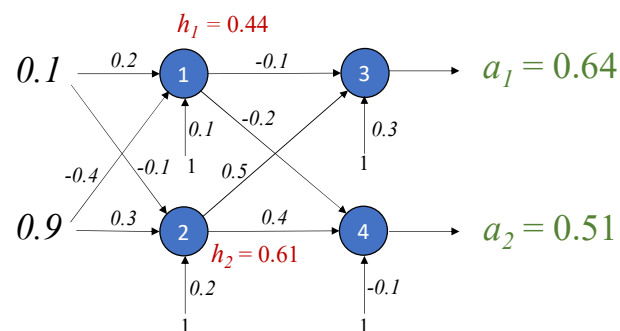
Diagram illustrating the weight update formula $w_{i,j} = w_{i,j} + \gamma \delta_i a_j$ with annotations:

- Current value** (orange text) points to the first $w_{i,j}$ term.
- Error for the destination unit** (red text) points to the δ_i term.
- Output value for the origin unit** (blue text) points to the a_j term.

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

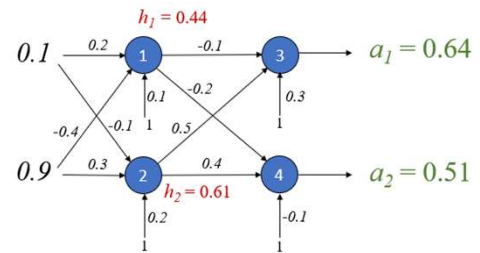
Delta errors:

$$\delta_{a_1} = (1 - 0.64) \times 0.64 \times (1 - 0.64) = \mathbf{0.083}$$

$$\delta_{a_2} = (0 - 0.51) \times 0.51 \times (1 - 0.51) = \mathbf{-0.127}$$

$$\delta_{h_1} = 0.44 \times (1 - 0.44) \times (-0.1 \times 0.083 + (-0.2) \times (-0.127)) = \mathbf{0.004}$$

$$\delta_{h_2} = 0.61 \times (1 - 0.61) \times (0.5 \times 0.083 + 0.4 \times (-0.127)) = \mathbf{-0.002}$$



Exercise - resolution

$$w_{h1x1} = 0.2 + 0.75 \times 0.004 \times 0.1 = 0.2003$$

$$w_{h1x2} = -0.4 + 0.75 \times 0.004 \times 0.9 = -0.3973$$

$$w_{h2x1} = -0.1 + 0.75 \times (-0.002) \times 0.1 = -0.10015$$

$$w_{h2x2} = 0.3 + 0.75 \times (-0.002) \times 0.9 = 0.29865$$

$$w_{a1h1} = -0.1 + 0.75 \times 0.083 \times 0.44 = -0.07261$$

$$w_{a1h2} = 0.5 + 0.75 \times 0.083 \times 0.61 = 0.5379$$

$$w_{a2h1} = -0.2 + 0.75 \times (-0.127) \times 0.44 = -0.24191$$

$$w_{a2h2} = 0.4 + 0.75 \times (-0.127) \times 0.61 = 0.34189$$

$$b_1 = 0.1 + 0.75 \times 0.004 \times 1 = 0.103$$

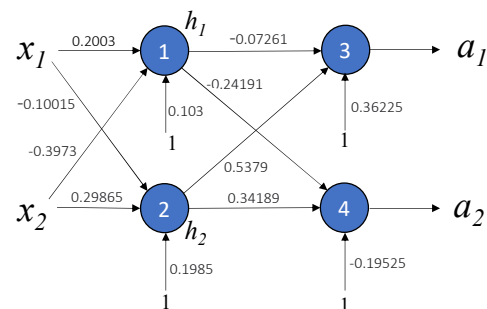
$$b_2 = 0.2 + 0.75 \times (-0.002) \times 1 = 0.1985$$

$$b_3 = 0.3 + 0.75 \times 0.083 \times 1 = 0.36225$$

$$b_4 = -0.1 + 0.75 \times (-0.127) \times 1 = -0.19525$$

Let us define $\gamma = 0.75$

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)

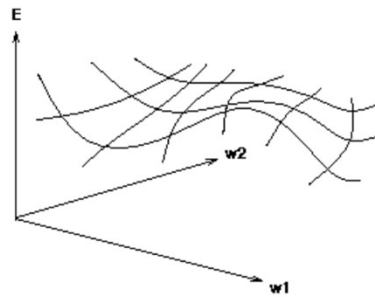
Gradient descent/backpropagation problems

■ 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

- Voice/image recognition (faces detection/identification, sentiment analysis)
- Translation
- Industrial control
- Financial analysis
- Medicine
- Robotics
- Pattern classification
- Non-linear control
- Etc. etc.

Neural networks applications

- Historical example: **NETTALK** (1987) was a system able to convert written English to intelligible speech (search for “sejnowski nettalk” in Youtube)

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