Neural Networks and Learning from Data

Stefano Melacci
Department of Information Engineering and Mathematics
University of Siena

Disclaimer

The contents (and the style) of these slides are taken from Stefano Melacci's slides of the *Machine Learning & Deep Learning* course - Datum Academy (France)

They are used with the sole purpose of supporting Stefano Melacci's teaching activity.

They are not intended to be of public domain in any way, and they must not be published or shared out of the context in which they are presented by Stefano Melacci.

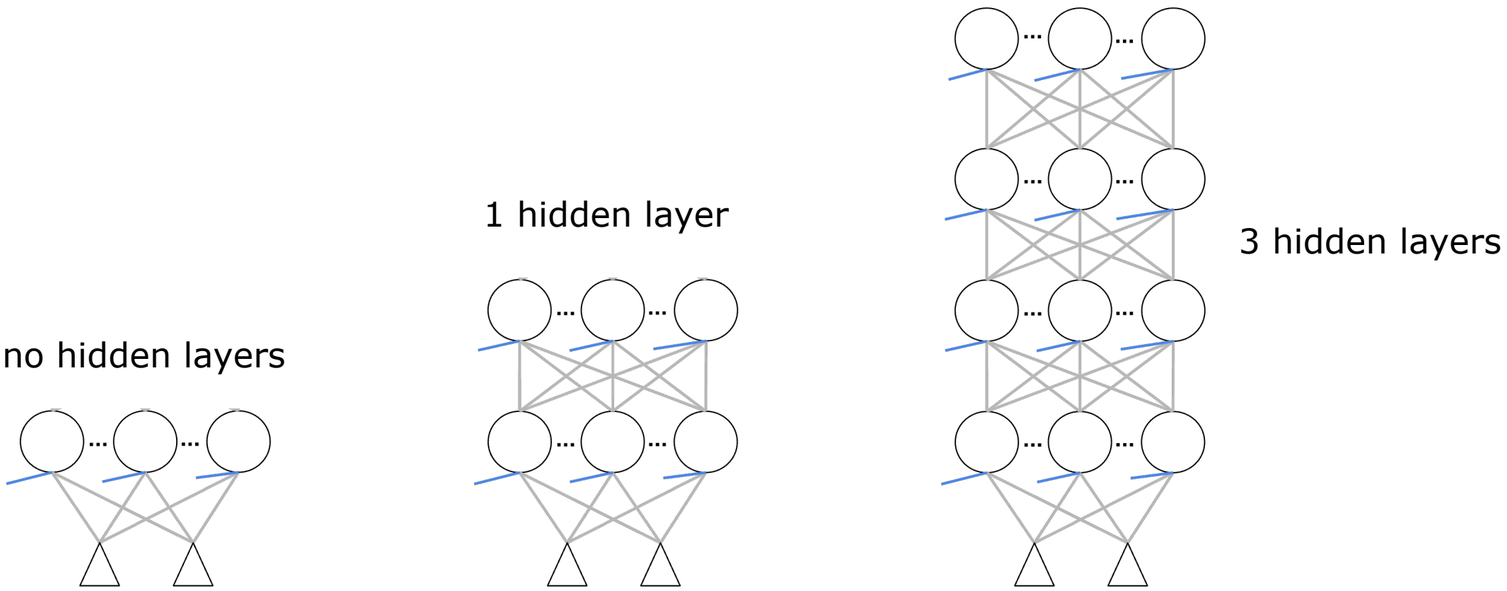
Warning: Course Website

- URL: https://elearning.unisi.it/course/view.php?id=11749
- Key: **NeuralNetworks2024Y**
- Write it down!

- After this lecture, the course webpage will be made accessible only using the above key
- Students will be automatically registered after having used the key to enter the website
- All the communications about changes in the course schedule, times, news, …, will be sent to the registered students only

Learning with Deep Architectures

- This lecture focuses on neural architectures with hidden neurons and on the Backpropagation algorithm (**neural networks**)
- The emphasis is on deep architectures, that are based on **many stacked hidden layers** to improve representational and computational capabilities



What is Machine Learning?

Stefano Melacci

What is Machine Learning?

Some passages from the Wikipedia page:

- *Machine learning is the subfield of computer science that gives computers the ability to learn without being explicitly programmed*
- *Machine learning explores the study and construction of algorithms that can learn from and make predictions on data*
- *Such algorithms overcome following strictly static program instructions by making data driven predictions or decisions, through building a model from sample inputs*

Process:

1. Learn from data ($\neq$ a static program!)
2. Build a model (or improve an already existing one)
3. Use the model to make predictions on data (not necessarily the same data that are mentioned above)

"Generalization"

What is Machine Learning?

A computer program is said to learn from **experience E** *with respect to some class of* **tasks** $\mathcal{T}$ *and* **performance measure P**, *if its performance at tasks in* $\mathcal{T}$, *as measured by P, improves with experience E.*

T. Mitchell, 1997

- Definition of a $\mathcal{T}$
- Representation of **E** (data)
- Definition of a criterion **P** that measures the quality of the performance

What about Artificial Intelligence?

AI vs. Machine Learning?

AI + Machine Learning?

AI = Machine Learning?

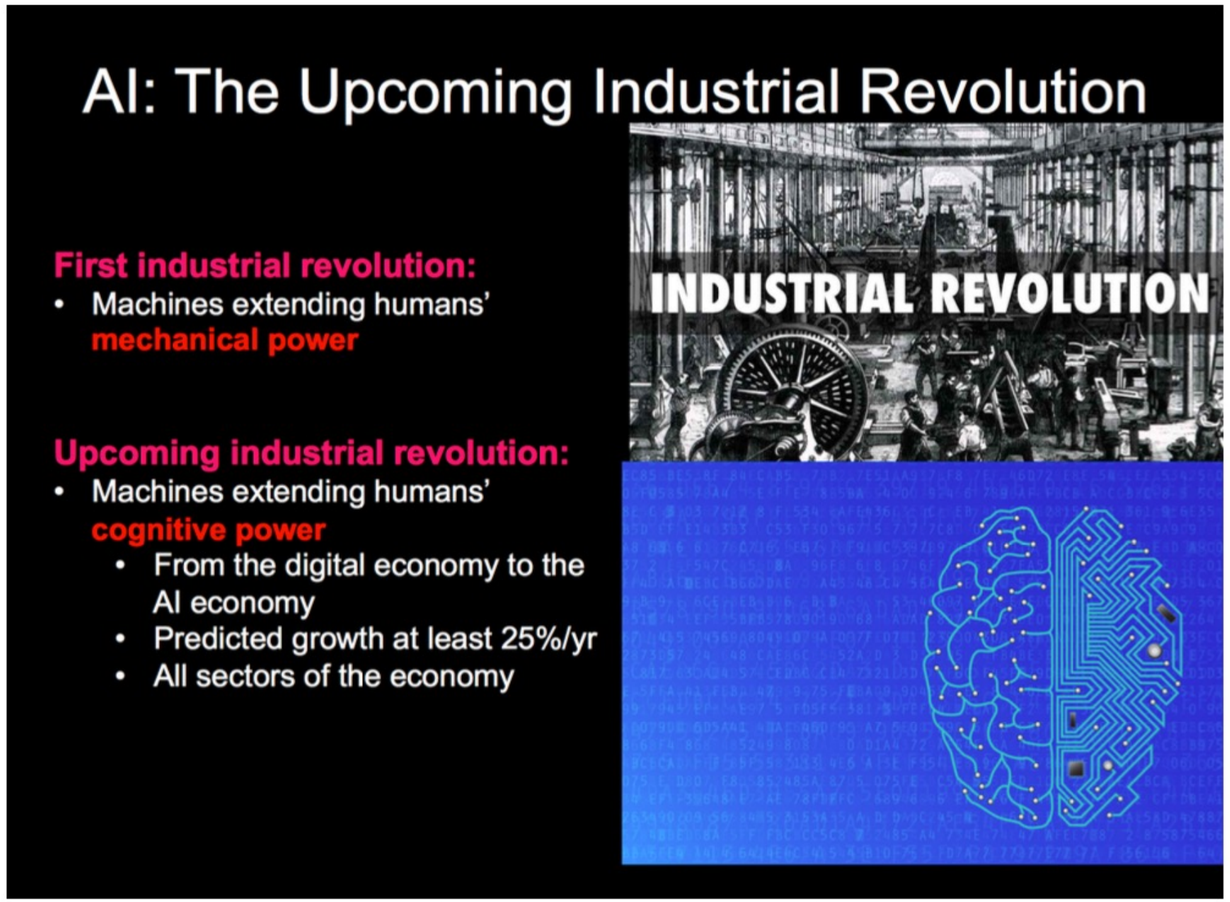
Artificial Intelligence (AI) *is a branch of Computer Science attempting to build machines capable of intelligent behaviour*

- Machine Learning is a (big) branch of AI
- AI is a broader concept

What is Going On?

- Machine Learning is everywhere….

What is Going On? (Cont.)



Taken from: Y. Bengio's talk at OSDC West, Santa Clara
http://www.iro.umontreal.ca/~bengioy/yoshua_en/talks.html

Learning Settings

Stefano Melacci

Example (Classification

Task: classify handwritten digits

- Learn $f(\boldsymbol{x})$ using the following T:

Training set

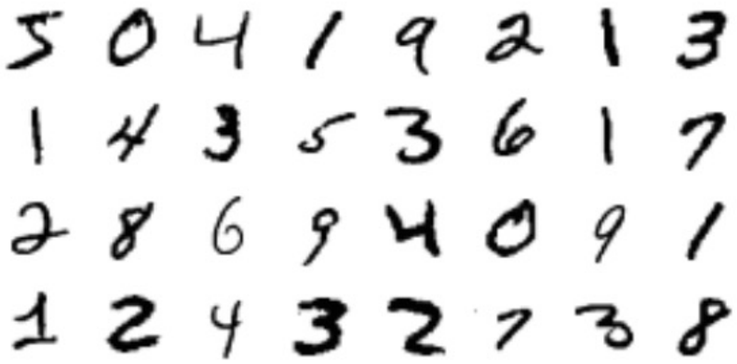
Taken from:
https://longhowlam.wordpress.com/2015/11/25/
a-little-h2o-deeplearning-experiment-on-the-mnist-data-set/

Example (Classification) – Cont.

Task: classify handwritten digits

- Use $f(\boldsymbol{x})$ to make predictions on new (unlabeled) data
- In other words, we want a system that is able to predict which **class** is associated to each of the following "new" data samples

Test set



Taken from:
http://luizgh.github.io/libraries/2015/12/08/getting-started-with-lasagne/

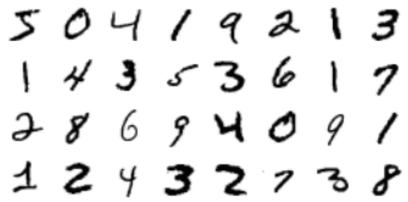
Machine Learning: Process

We are given a task $\mathcal{T}$, that we want to approach with ML

Before making choices on what ML algorithm could be used (and how), we have to:

- Collect task-related information
  - Task description
  - Goal of the task
  - Available background information (prior knowledge)
- **Get data!**
  - **Collect data**
  - **Collect the meta-information attached to the data**
    - It will be likely incomplete, noisy, poor
    - The background information can improve the meta-information...
    - ...and also the learning algorithm!

Machine Learning: Process (Previous Example)



Task $\mathcal{T}$: classify handwritten digits

- Some task-related information (example)
  - Desc: We are given images (fixed resolution) of handwritten digits, that belong to 10 different classes
  - Goal: Predict the class to which never-seen-before images belong
  - Classes are mutually exclusive (very important!)
- **Get data!**
  - **Collect data X**
  - **Collect class-labels Y (meta-info)** (and build T)

Classification

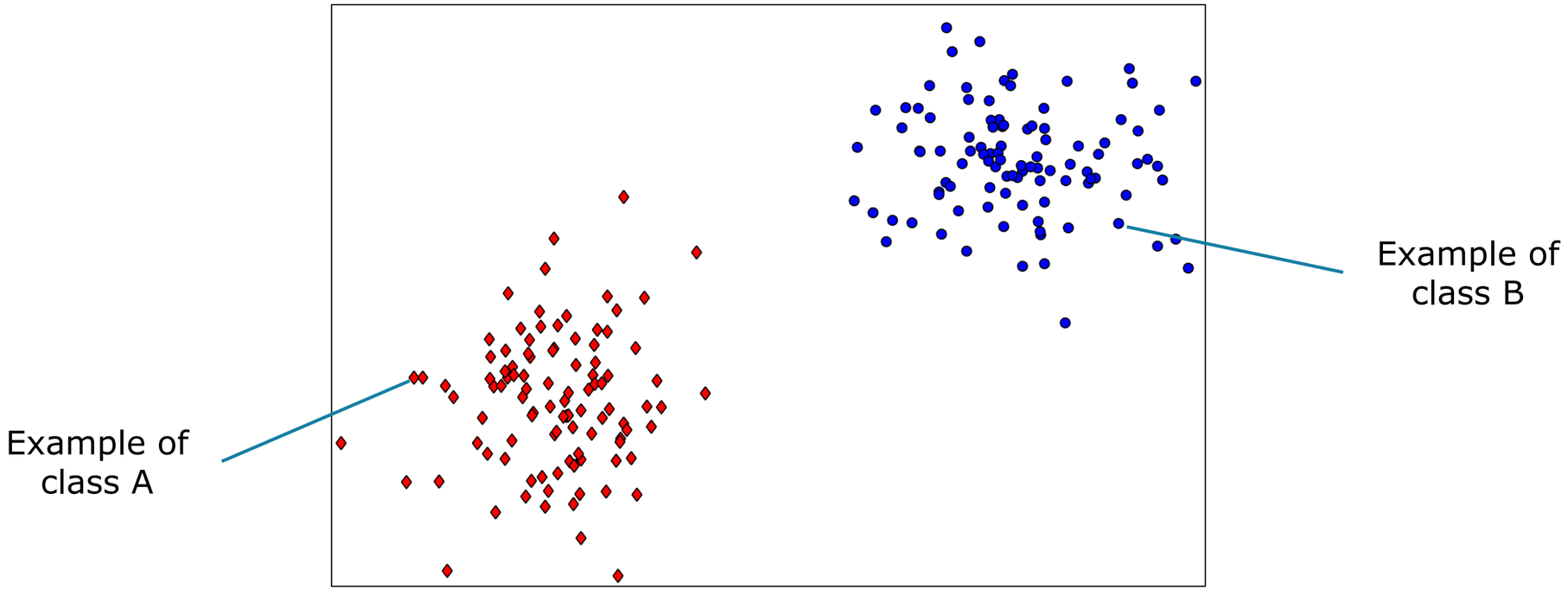
- We are given a set of examples of data belonging to **class A** and data belonging to **class B**
  - *Goal: learn a classifier that is able to distinguish elements of class A from the ones belonging to class B*
  - Generalize the learned concept, gain the capability of making predictions on new examples

- Examples
  - Pictures of cars (class A), pictures of people (class B)
  - Samples of your voice (class A) and of other people (class B)
  - Documents talking about math (class A), documents that are about geography (class B)
  - Data from sensors when your car is OK (class A), data when there are issues (class B)
  - …

- *Binary* classification (two classes) vs. *Multi-class* classification (more classes)

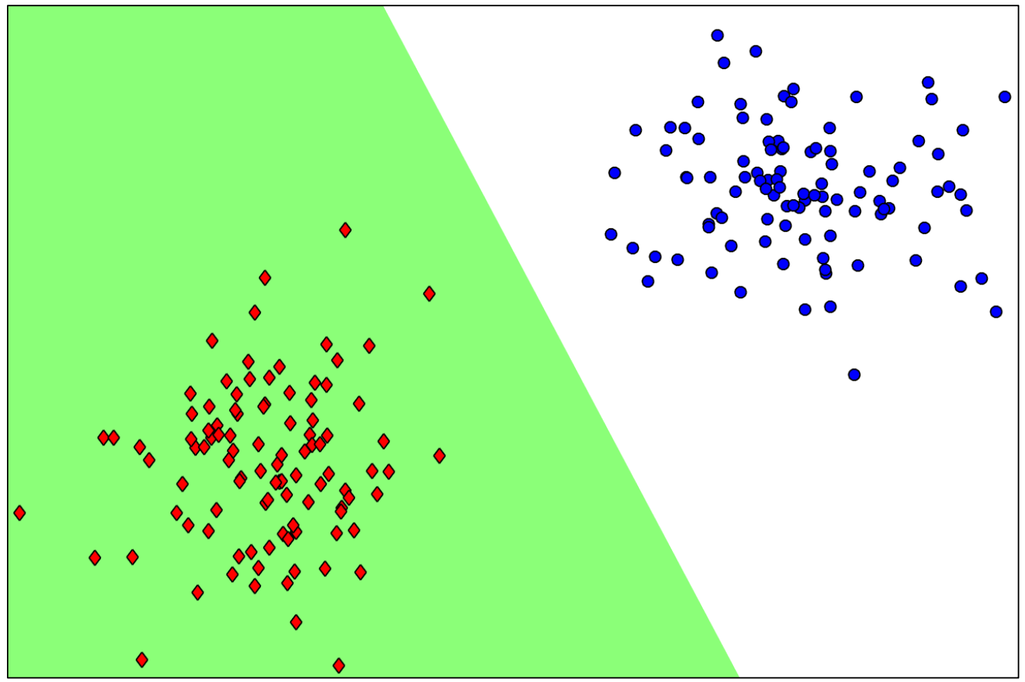
Classification (2)

- Suppose that red diamonds are 2D examples of data from class A, while blue circles are examples of data belonging to class B



Classification (3)

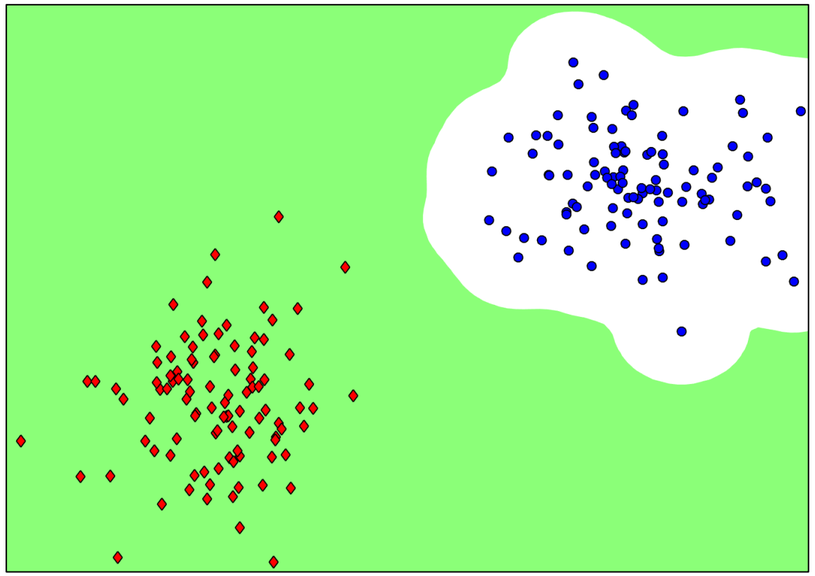
- Our classifier can be modeled by a **mathematical function**
- The classifier/function must learn to *output different values in the space regions associated to the two classes*
  - For example:
    - Green area: output value 1
    - White area: output value -1

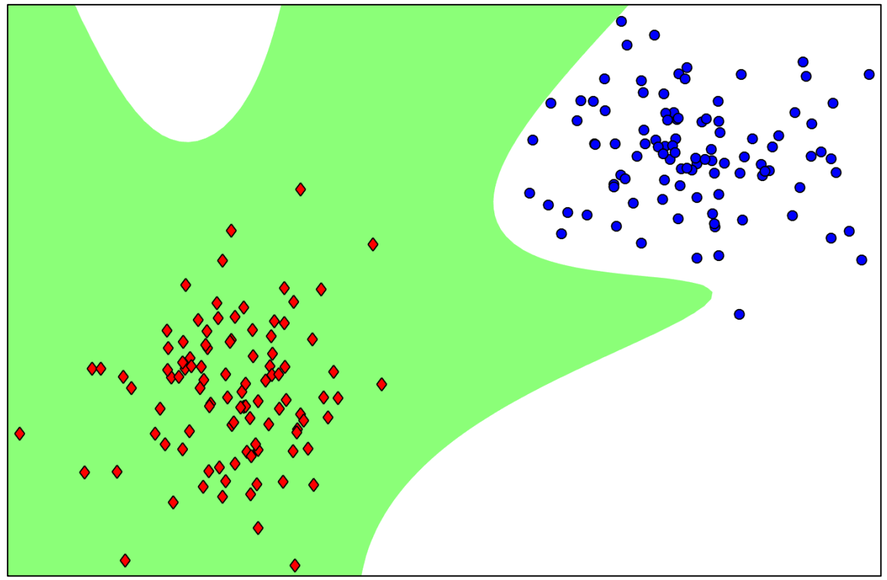


- If the function is defined in the whole input domain, than it can also be evaluated in new data points, thus predicting their classes

Classification (4)

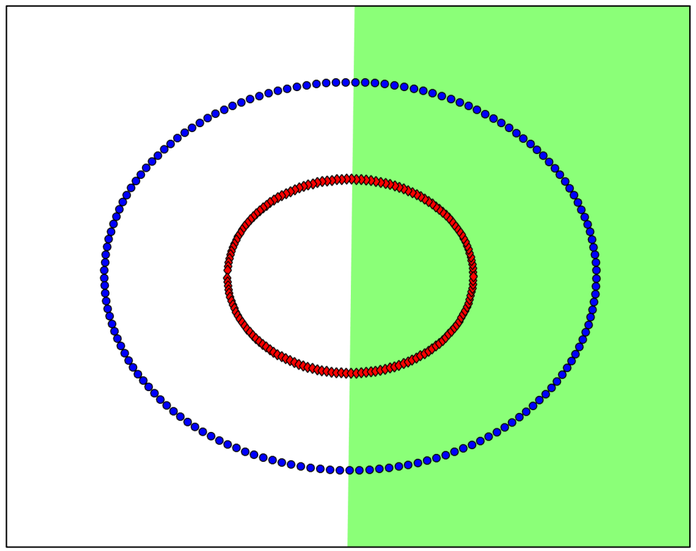
- Several classifiers/functions could be learnt from the same data
  - Depending on the class of functions on which we focus
  - Depending on further constraints on the learning problem



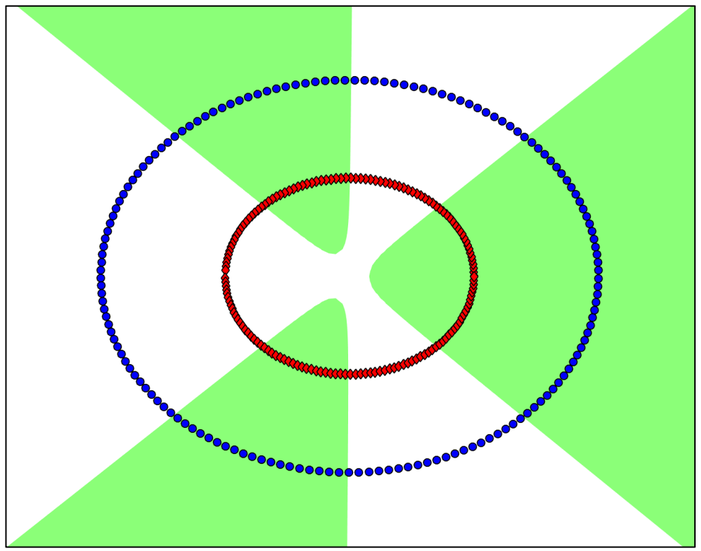


Classification (5)

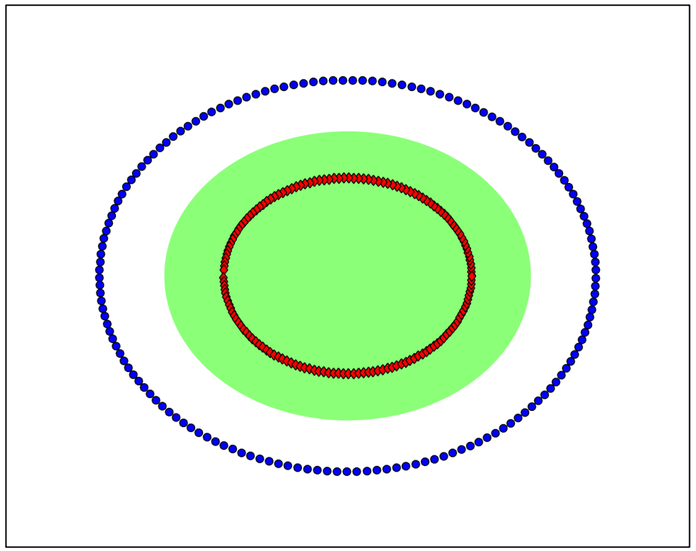
- If the class of functions that we select is “too simple”, we might not be able to learn a classifier that can clearly separate the two classes



Fail!



Fail!



Success!

Regression

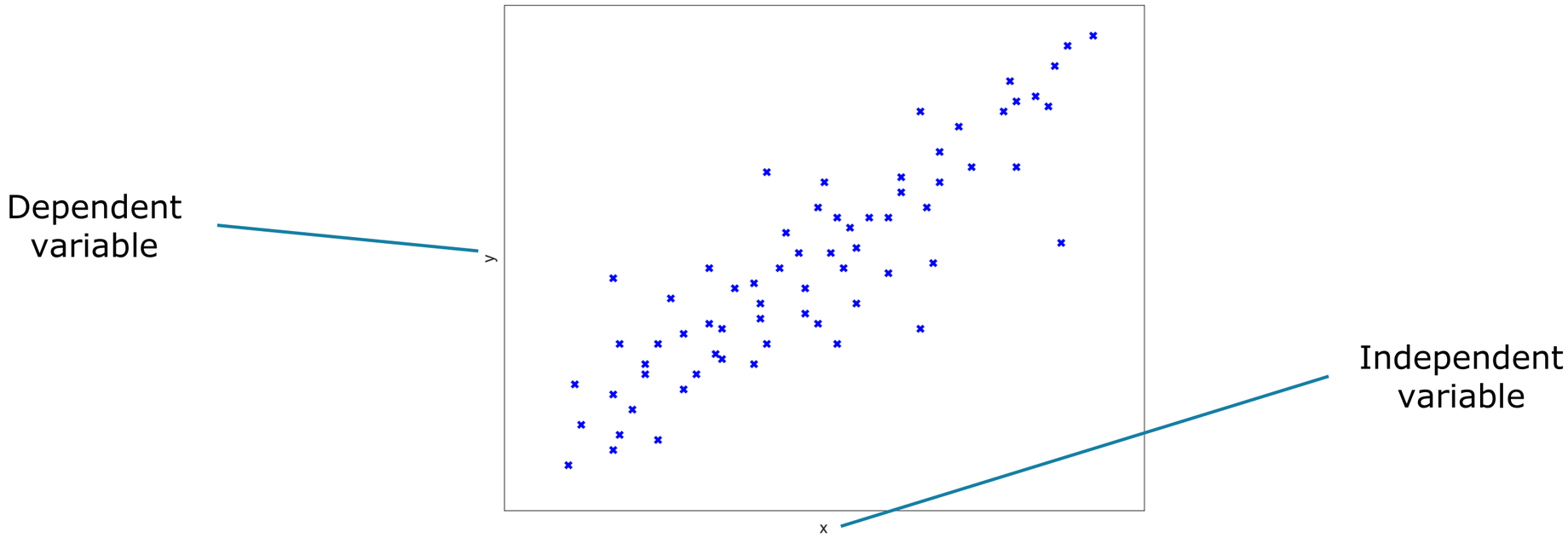
- We are given a set of pairs of measurements, each of them composed of: (values of **independent** variables, values of the **dependent** variables)
  - *Goal: learn a model that is able to predict the values of the dependent variables (tolerating some error) given the values of the independent ones*
  - Generalize the learned concept, gain the capability of making predictions on new examples

- Examples
  - Day of the week (independent), value of the temperature (dependent)
  - Distance covered with your car (independent), amount of gasoline (dependent)
  - Number of days before the flight (independent), price of the ticket (dependent)
  - …

- *Scalar* regression (single dependent variable) vs. *Vector-valued* regression (more dependent variables)

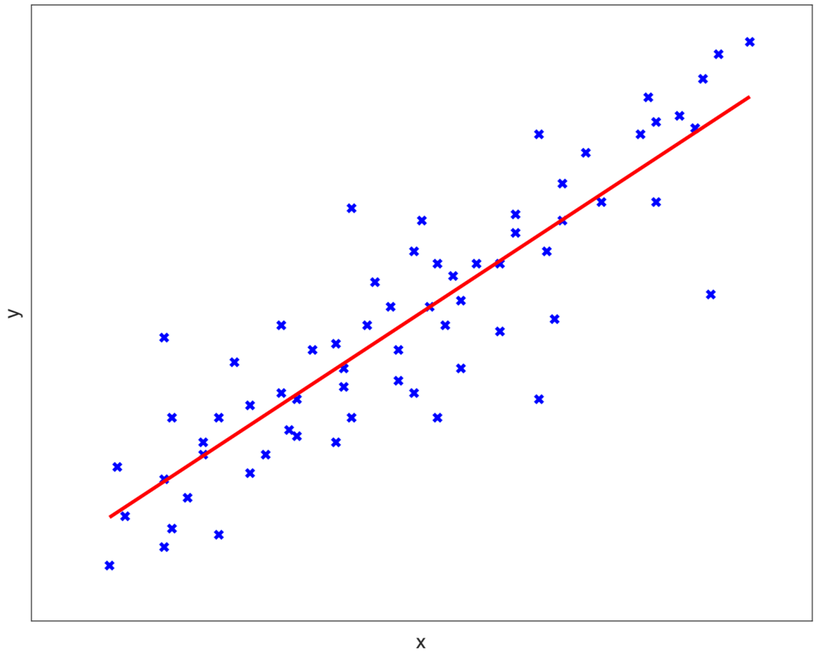
Regression (2)

- Suppose that each blue cross is the visualization of a pair of measurements, where x = value of the independent variable, y = value of the dependent variable



Regression (3)

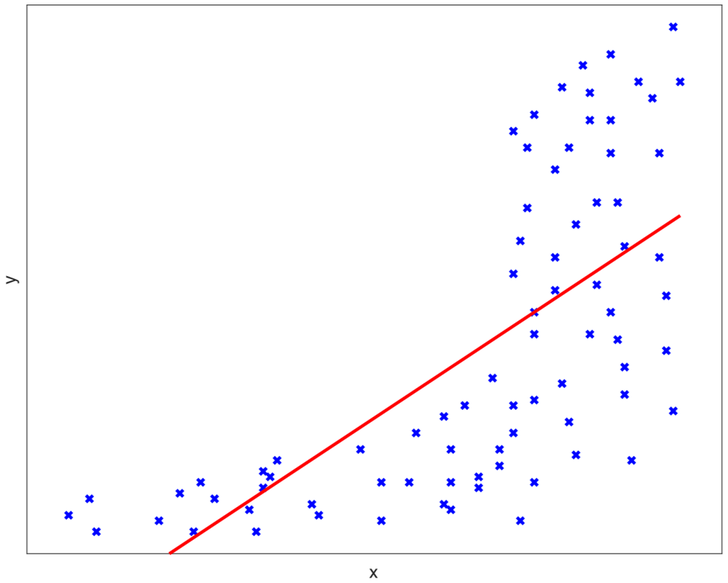
- The regression model is a **mathematical function**
- The regression model/function must learn to *output the value of y (+error) for each given x*
  - For example:
    - Linear regression: the output of the function is represented by the red line

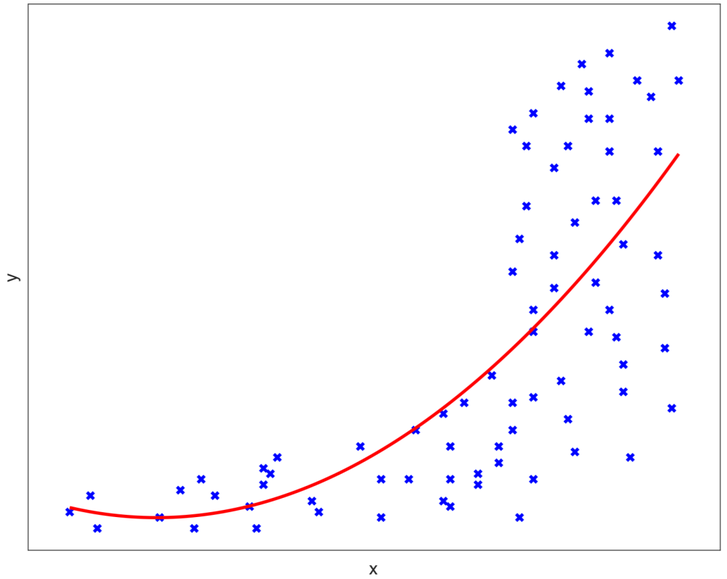


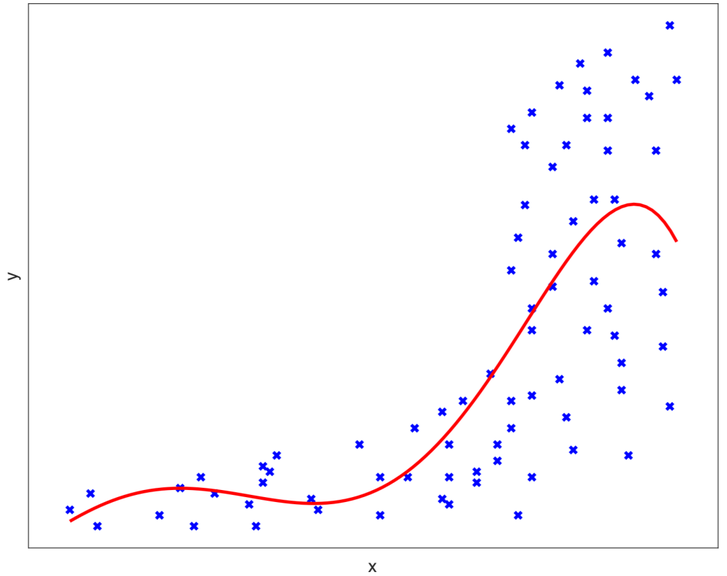
- If the function is defined in the whole input domain, than it can also be evaluated in new data points (independent var), thus predicting the value of the dependent variable

Regression (4)

- Several regression models/functions could be learnt from the same data
  - Depending on the class of functions on which we focus
  - Depending on further constraints on the learning problem
- If the class of functions that we select is “too simple”, we might not be able to learn a regression model that can reasonably approximate the trend of y







Classification, Regression: Learning Functions

- **Classification** and **regression** problems in Machine Learning introduce the same challenges, that is the one of *learning* a classifier/regression model from a set of examples
- In both the cases, we aim at *learning* a function *f(**x**)*, that is able to make a prediction on the provided input **x**

$$\mathbf{x} \in \mathbb{R}^m$$

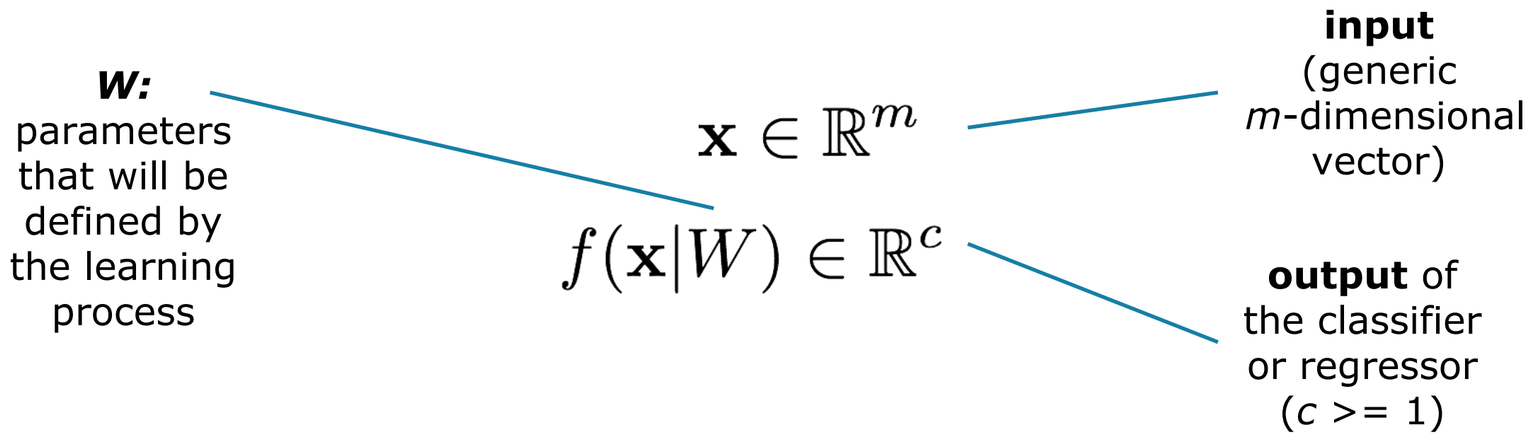
input (generic *m*-dimensional vector)

$$f(\mathbf{x}) \in \mathbb{R}^c$$

output of the classifier or regressor (*c* >= 1)

Functions & Learnable Parameters

- The function *f* can be defined in several ways
  - Such function will depend on a number of *parameters* whose value will be defined by the **learning** process ("learnable parameters")
  - Let's indicate them all with *W*



Class of Functions: Linear or Poly Functions

- For example, *f* could be defined as a *linear* function, and where the **learnable parameters** are the coefficients of the linear expansion

$$f(\mathbf{x}|W) = \mathbf{w}'\mathbf{x} \qquad W = \{\mathbf{w}\}$$

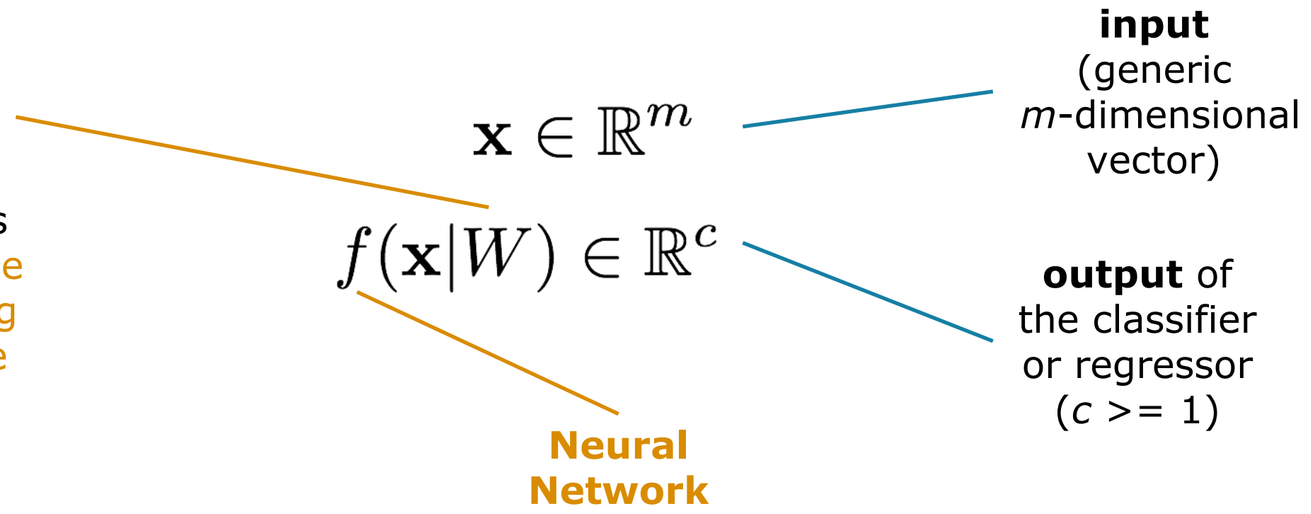
- However, we could also decide that *f* is a *polynomial* function of degree *n*

$$f(\mathbf{x}|W) = (\mathbf{w}'\mathbf{x} + 1)^n \qquad W = \{\mathbf{w}\}$$

  - The degree *n* is <u>not</u> a learnable parameter, it is a design choice (*hyperparameter*)
  - The coefficients ***w*** are the **learnable parameter**s of the model

Class of Functions: Neural Networks

- In the case of this module, the function *f* will be a Neural Network
- The parameters *W* that we have to learn are the so-called **weights** and **biases**
  - We will introduce them in the next slides
- *For the sake of simplicity, the explicit dependence on W will not be usually indicated in the expression of f*



Introduction to Neural Networks

Stefano Melacci

Neural Networks

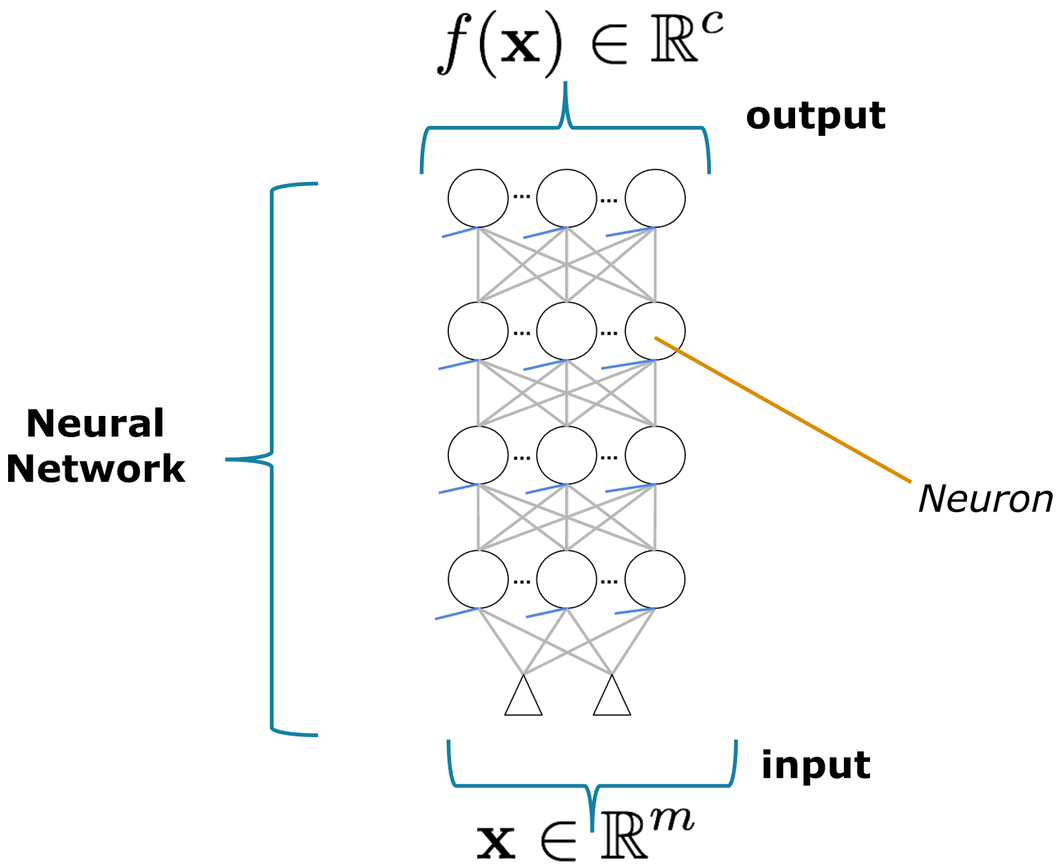
- **Artificial Neural Networks** (ANNs or, simply, NNs) are brain-inspired systems, that are based on a set of interconnected computational units, called *neurons*
  - They are "inspired" by the biological neural networks of our brains
  - This does not mean they are a reproduction of the mechanisms behind our brain

- ANNs are likely the most popular tools in the context of Machine Learning
  - **They can learn from examples**

- The early origins of ANNs can be traced back to 1943, when McCulloch & Pitts modeled a simple neuron using electrical circuits
  - Then, in the following decades, they were studied in the context of Artificial Intelligence, with the first applications to real word learning problems
    - Rosenblatt's perceptron (1958), Widrow & Hoff's models (1960), …

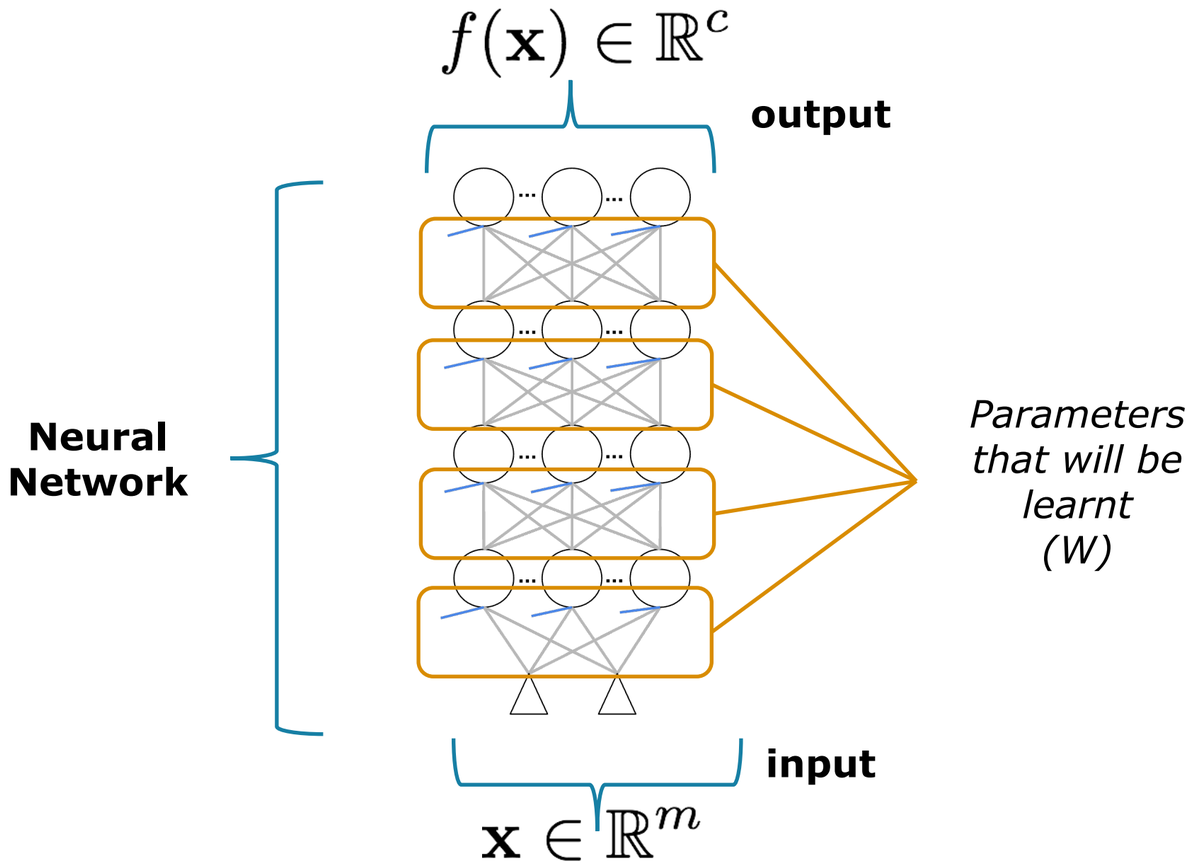
Neural Networks (2)

- Before going into further details, let's "visualize" an example of neural network in the context of what we described so far



Neural Networks (3)

- The structure of the network is <u>not</u> a learnable parameter
  - Number of neurons
  - Organization of the connections
  - …
- The values attached to each connection (**weights** and **biases**) are the **learnable** parameters



Neural Networks - History

- *Warren McCulloch*: University of Illinois, College of Medicine
- *Walter Pitts*: American mathematician, logician, interested in the field of computational neuroscience

- Their paper, "*A Logical Calculus of Ideas Immanent in Nervous Activity*" (1943) described the first model of a neural network (**neuron**)
  - They modeled logic functions with a mathematical model
  - It only allowed for binary inputs and outputs
  - Weights of the connections were fixed
  - From a very qualitative point of view, there was **no "learning" yet**, but their model was a strong basis of inspiration for the following work in this field

Neural Networks - History (2)

- *Frank Rosenblatt:* Cornell Aeronautical Laboratory (Buffalo, state of New York)

- Rosenblatt developed a model of a neuron (**Perceptron**) that was way more generic and advanced than the ones of McCulloch and Pitts
  - More importantly, he proposed an algorithm to **learn connection weights**

- He focused on the context of image recognition, and his model was implemented in hardware

- This result strongly increased the interest in the field of artificial neural networks, even if the model had several limits
  - Simple linear model

Neural Networks - History (3)

- *Marvin Minsky & Seymour Papert,* in 1969, wrote a book called *"Perceptrons: an introduction to computational geometry"* that, amongst other things, highlighted the limits of the perceptron model
  - It was not able to handle the XOR function!
  - Marvin Minsky: one of the most important leaders and innovators in AI

- Their book is frequently considered the reason why the research in the field of artificial neural network did not progress in the **seventies/early eighties**, and the interest toward neural models strongly decreased
  - AI Winter

Neural Networks - History (4)

- New results on "Multi-Layer" Perceptrons and their training procedure (Backpropagation) reactivated the interest
  - 1974: *Paul Werbos* wrote down the first ideas of backpropagation (first multilayered networks)
  - 1986: *Rumelhart, Hinton & Williams* popularized and made concrete the backpropagation algorithm

- …that went down again due to the popularity of Kernel Machines

- Early 2000: New results in Deep Learning lead to the nowadays progresses in the field of neural networks
  - *Yoshua Bengio,* "Learning deep architectures for AI", Foundations and trends in Machine Learning, 2(1):1–127, 2009

Neural Networks

Stefano Melacci

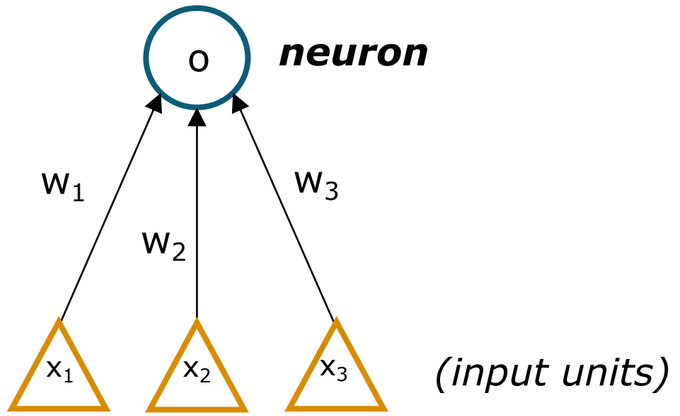
Neuron

- Each neuron of a NN performs a basic computation: the dot product between the input signal and the weights of the input-to-neuron connections
- This is commonly followed by the computation of the so-called **activation function**, that yields the output of the neuron
  - In the following example we have:

$\boldsymbol{x} = [x_1, x_2, x_3]'$	neuron *input*
w_j	connection *weight*
a	*activation* score
$\sigma(\ldots)$	activation *function*
o	neuron *output*

$$a = \sum_j w_j x_j$$

$$o = \sigma(a)$$



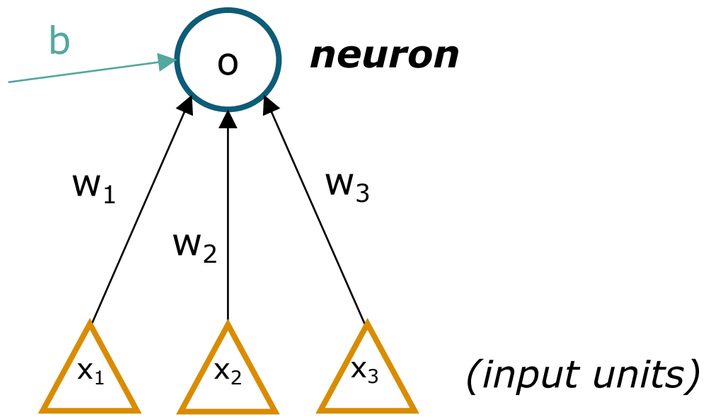
Neuron (2)

- A scalar **bias** is usually added to the activation score
  - It can be seen as a new connection with weight *b*, associated to an input unit that is constantly equal to *1*
    - *Sometimes the bias will not be explicitly shown in pictures or formulas, for simplicity*
  - The activation score can be more compactly written as the dot product between the **weight vector** and the **input vector**

$$a = \sum_j w_j x_j + b$$

$$a = \mathbf{x}'\mathbf{w} + b$$

$$o = \sigma(a)$$



Layer of Neurons

- Multiple neurons can be connected to the same input, thus generating the so-called **layer**

$\boldsymbol{x} = [x_1, x_2, x_3]'$	**layer** *input*
w_{ij}	connection *weight* from the j-th input unit to the i-th neuron of the layer
$\boldsymbol{b} = [b_1, b_2]'$	*biases* of the neurons belonging to the **layer**
$\boldsymbol{a} = [a_1, a_2]'$	*activation* scores of the **layer**
$\boldsymbol{o} = [o_1, o_2]'$	**layer** *output*

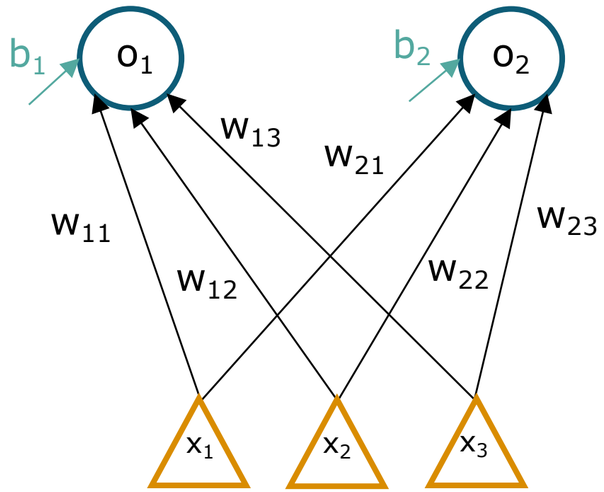
$$a_i = \sum_j w_{ij} x_j + b_i$$

$$\mathbf{a} = W\mathbf{x} + \mathbf{b}$$

each row of matrix W is the (transposed) weight vector of a neuron of the layer

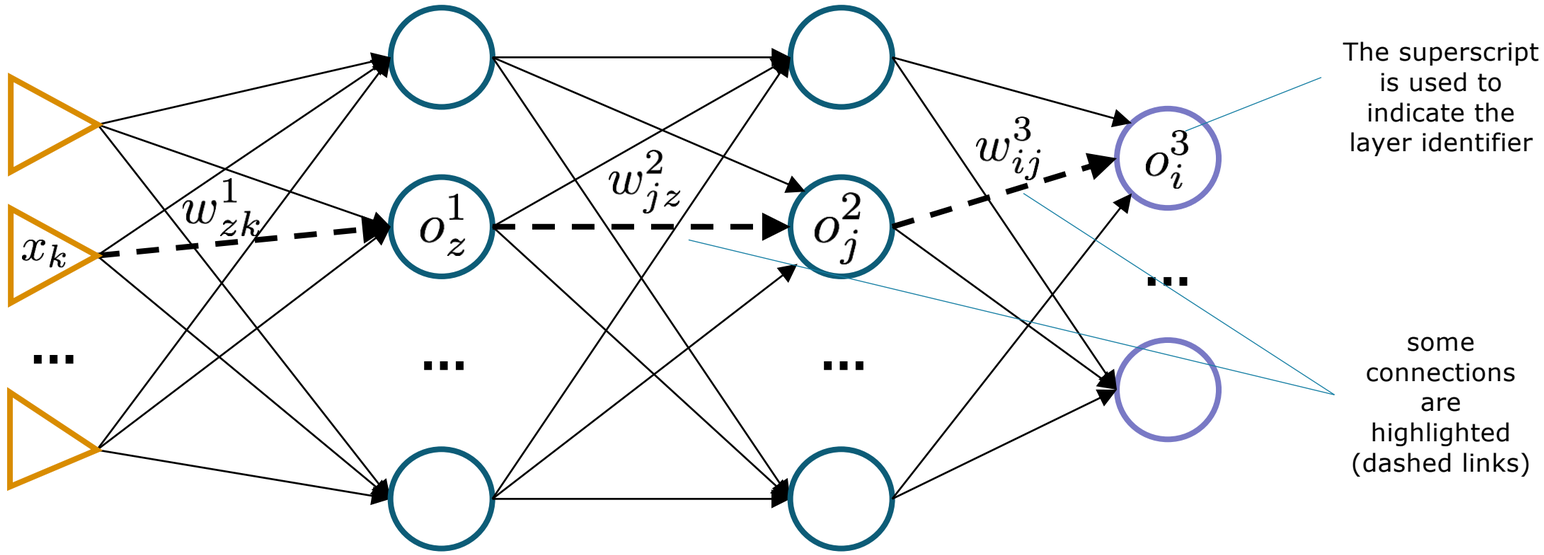
$$\mathbf{o} = \sigma(\mathbf{a})$$

the activation function operates element-wise on the component of its input vector



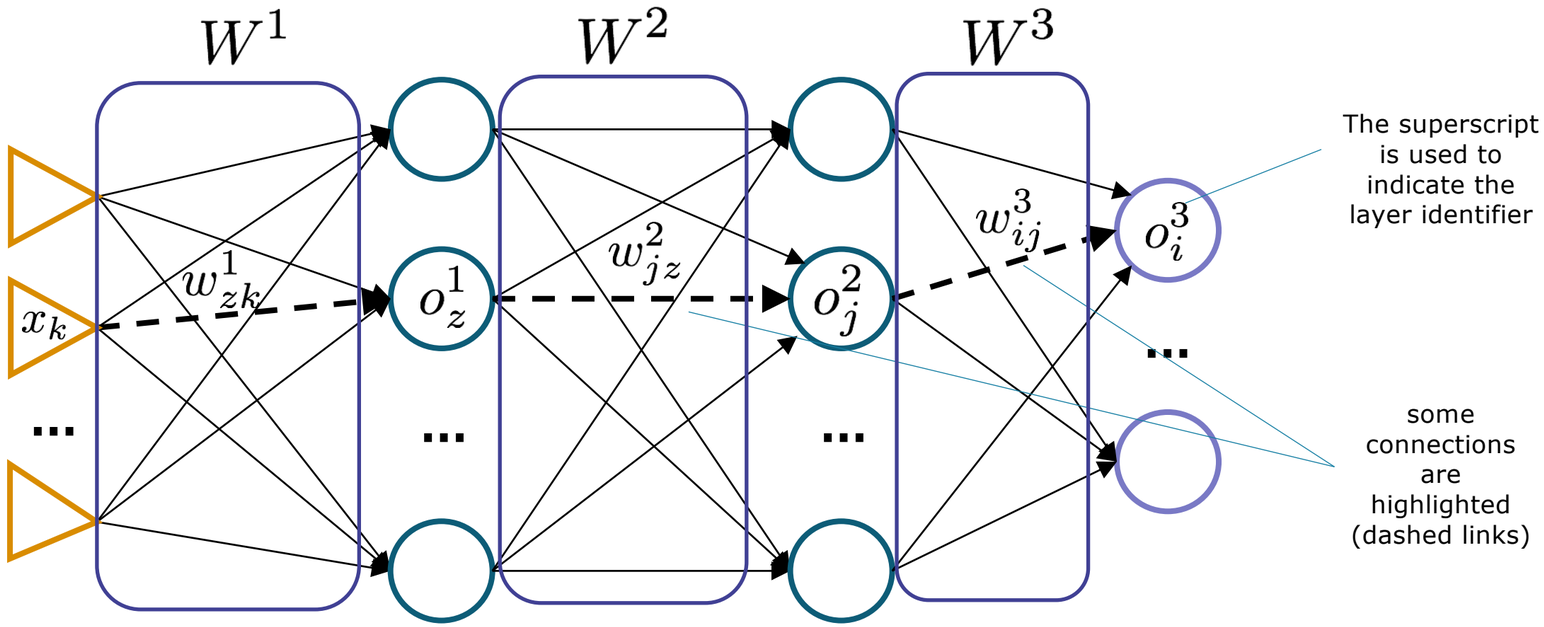
Multiple Layers

- We can stack layers, creating a **multi-layer feed-forward** network
  - *Feed-forward*: the input signal propagates up to the last layer, without performing any circles
  - Layers can have different **sizes** (i.e., different number of neurons)
  - The input units of the network are commonly represented with triangles



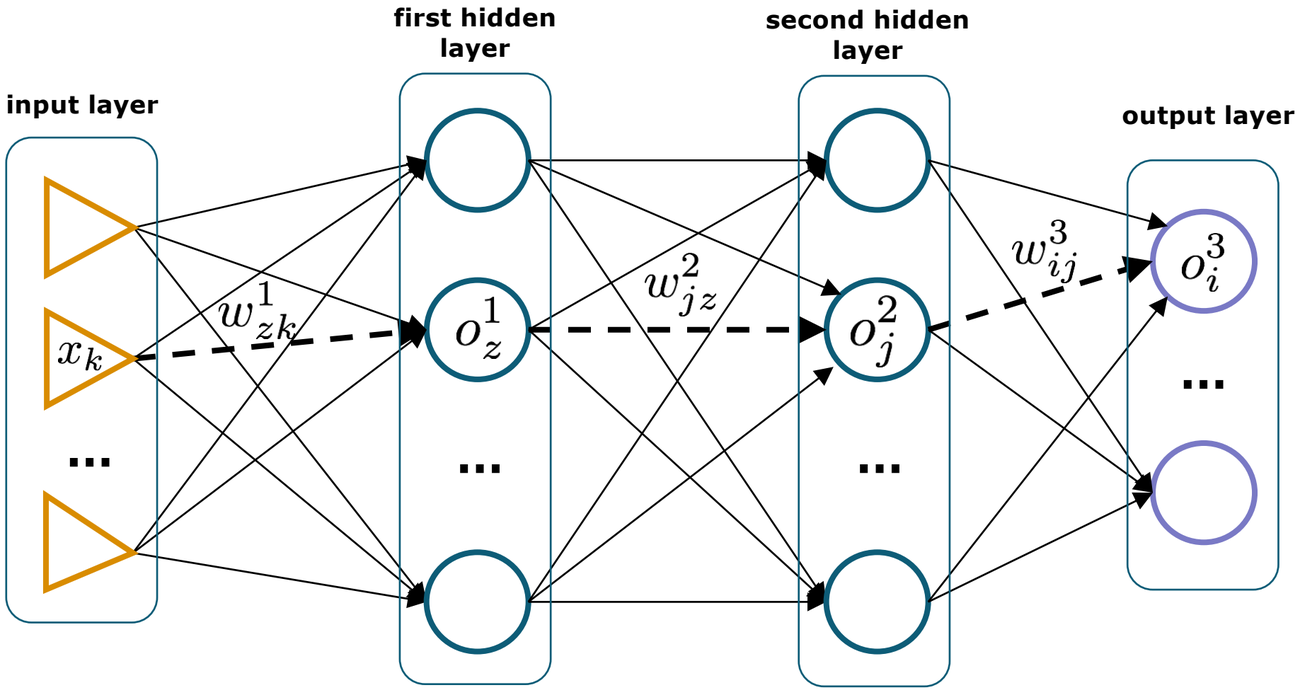
Multiple Layers (2)

- The weights connecting layer *h-1* to layer *h* are collected into matrix W^h
  - *Each layer has also a bias vector* $\boldsymbol{b}^h$ *(not shown - for simplicity)*



Multiple Layers (2)

- **Multi Layer Perceptron** (MLP)
- We distinguish among **input** layer, **hidden** layer(s), **output** layer



In the picture we have 3 layers

$$\ell = \text{number of layers}$$

$$\mathbf{o}^0 = \mathbf{x}$$

$$\mathbf{a}^h = W^h \mathbf{o}^{h-1} + \mathbf{b}^h, \quad h = 1, \ldots, \ell$$

$$\mathbf{o}^h = \sigma(\mathbf{a}^h), \quad h = 1, \ldots, \ell$$

Multiple Layers (3)

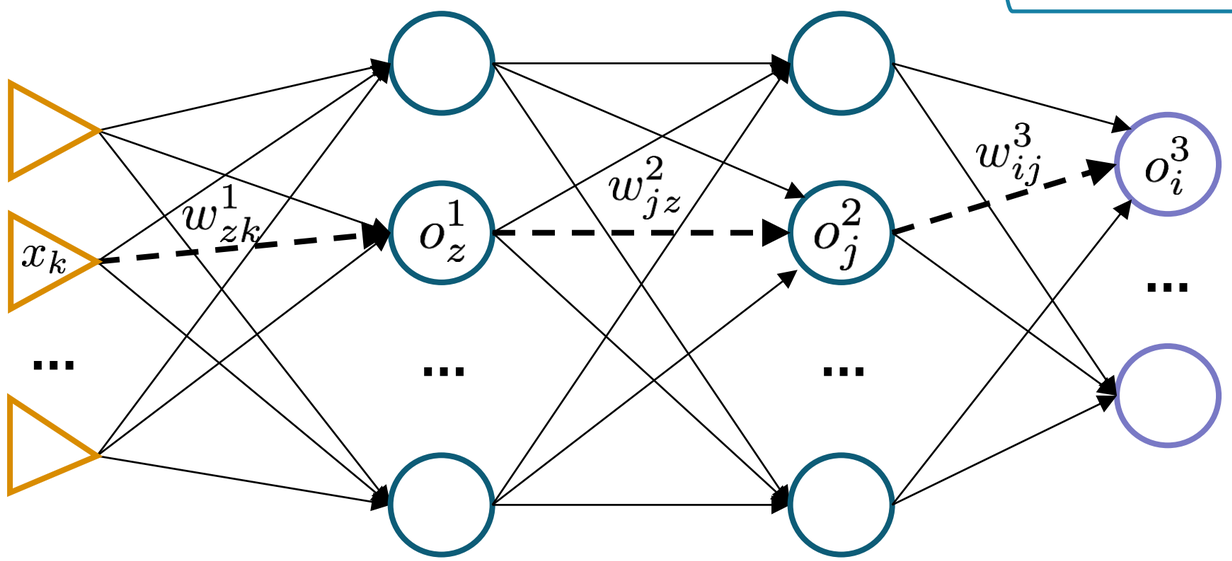
- Given an input ***x***, the MLP network computes the following function
  - *Discarding biases, for simplicity*

Output of the last layer (output of the network)

Output of the first layer

$$f(\mathbf{x}) = \sigma\left(W^3\sigma\left(W^2\sigma\left(W^1\mathbf{x}\right)\right)\right)$$

Output of the second layer



Multiple Layers (4)

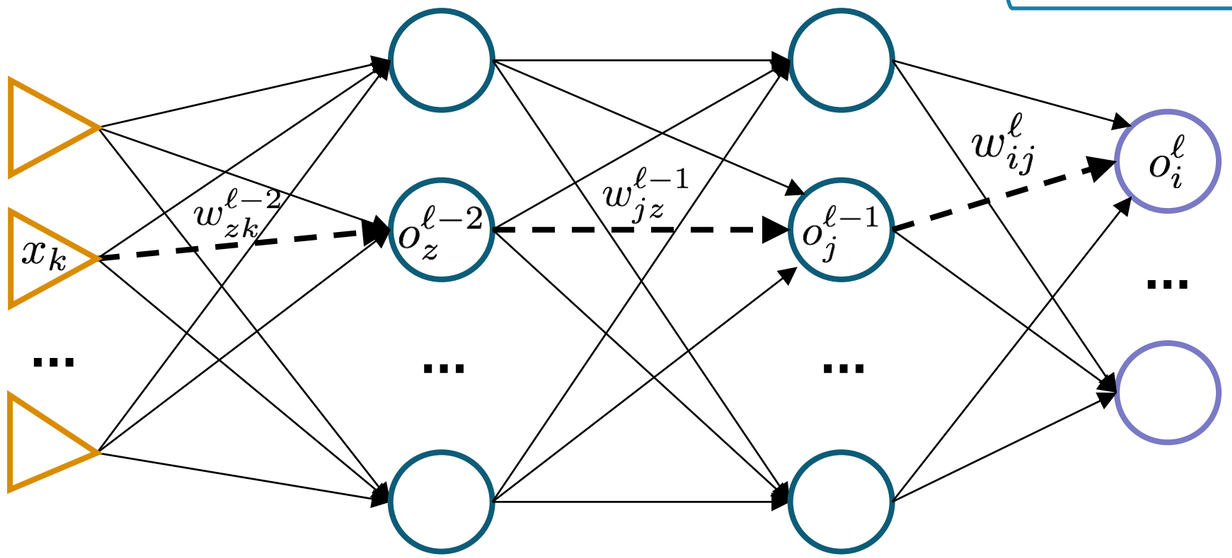
- We can equivalently write the formula (and draw the picture) using a notation in which the last layer index is explicit

Output of the last layer
(output of the network)

Output of the first layer

$$f(\mathbf{x}) = \sigma\left(W^{\ell}\sigma\left(W^{\ell-1}\sigma\left(W^{\ell-2}\mathbf{x}\right)\right)\right)$$

Output of the second layer

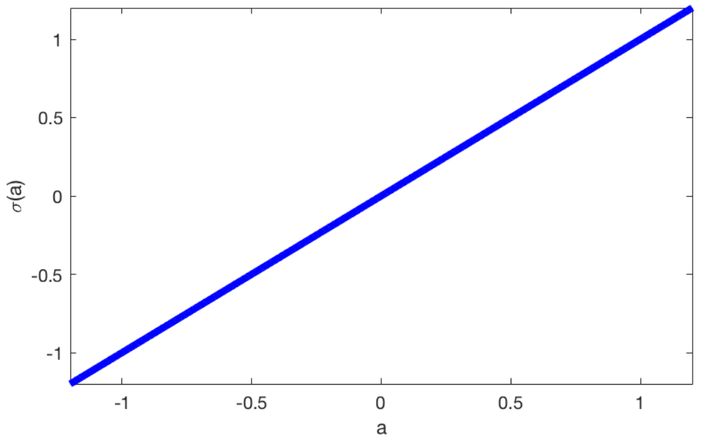


Activation Functions

- Most common activation functions
  - **Linear** (output layer)

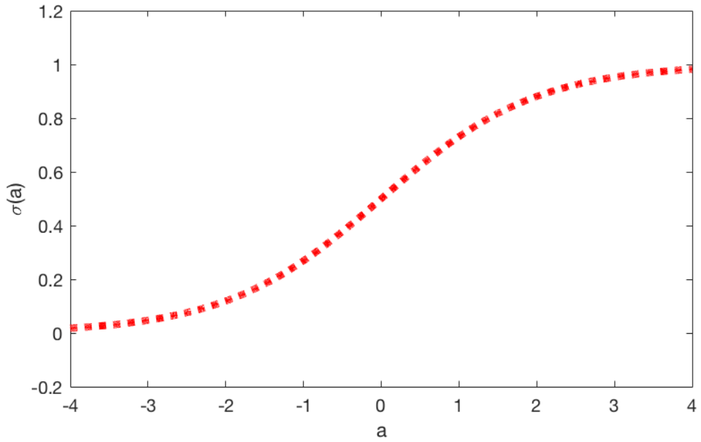
Stacking multiple linear layers we get a (single) linear layer

$$\sigma(a) = a$$



  - **Sigmoid**

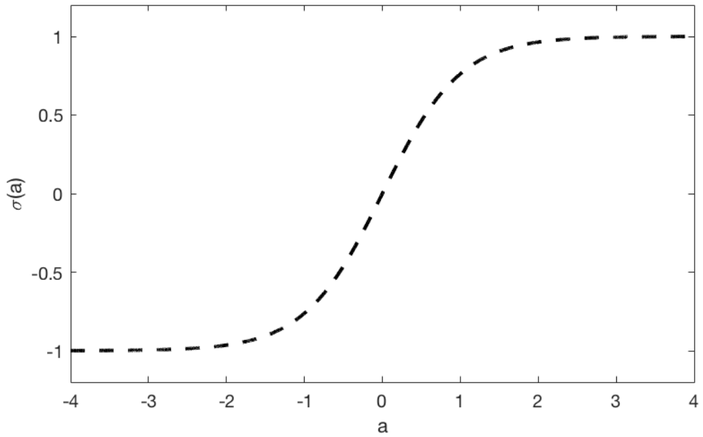
$$\sigma(a) = \frac{1}{1 + e^{-a}}$$



Activation Functions (2)

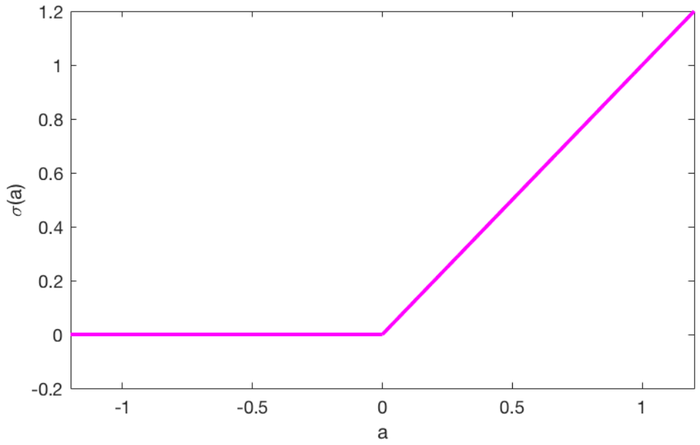
- Most common activation functions
  - **Hyperbolic Tangent** (*tanh*)

$$\sigma(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$



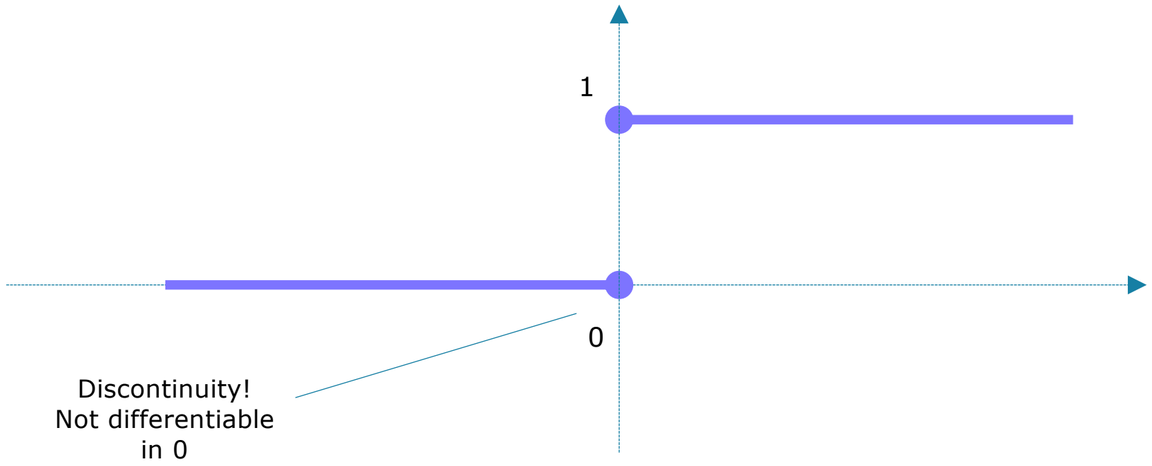
  - **Rectifier** (*ReLu*)

$$\sigma(a) = \max(0, a)$$



Activation Functions (3)

- Are these functions "differentiable"?
  - For reasons that will become clear later, in principle the activation function must be differentiable (it is a condition required by the network training procedure)
  - However, the derivative of the **ReLu** activation with respect to its input is:



This condition can be relaxed by assuming that the derivative in 0 is, for example, 0.

It is very unlikely to have an activation score that is exactly equal to 0.

Modern neural networks make a large use of the ReLu activation

Activation Functions (3)

- Another common activation function is the so-called **softmax**
  - It outputs values in [0,1]
  - Differently from the previously listed functions, the output of *softmax* is function of the activation scores of all the neurons of the layer

- When *softmax* is applied to all the neurons of the layer:
  - the sum of the layer outputs is 1
    - **Probabilistic normalization**
  - the *winning* neuron has output (close to) 1, the other neurons have output (close to) 0
    - Winning neuron: the neuron whose activation score is larger than the others

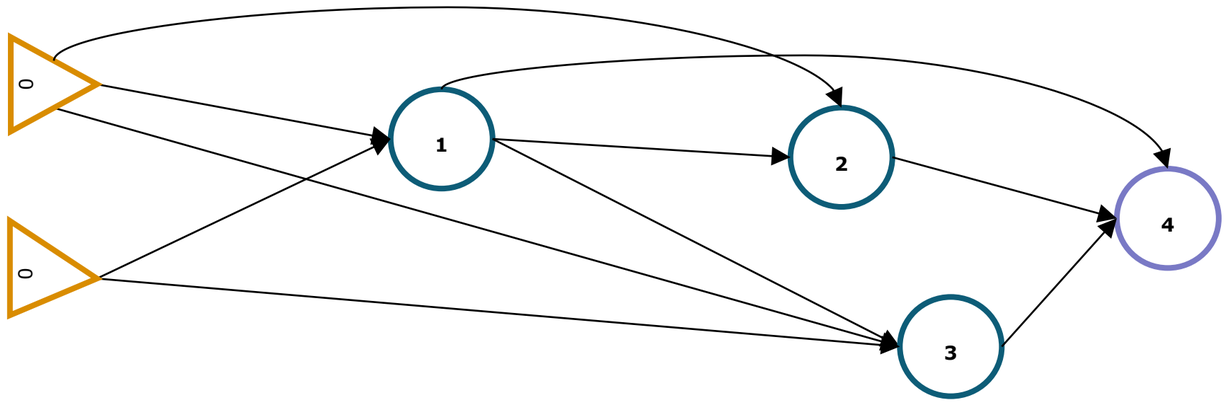
$$\sigma(\mathbf{a})_i = \mathtt{softmax}(\mathbf{a})_i = \frac{e^{a_i}}{\sum_j e^{a_j}}$$

the activation of the i-th neuron of the layer

all the activation scores of the layer are considered!

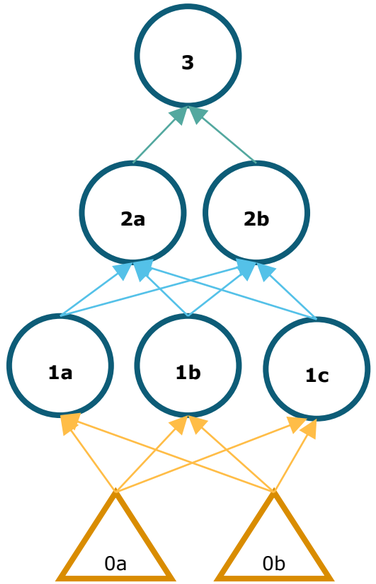
Directed Acyclic Graphs

- Instead of the classic MLP with dense connections from one layer to the following one, feed-forward Neural Networks can have different architectures
  - Sparse connections, connections from one layer to all the following layers, …
- More generally, all the Neural Networks for which we can define the ordering of the neurons, and every connection goes from one neuron to one of the following neurons
  - **Directed Acyclic Graph (DAG)**



Directed Acyclic Graphs (2)

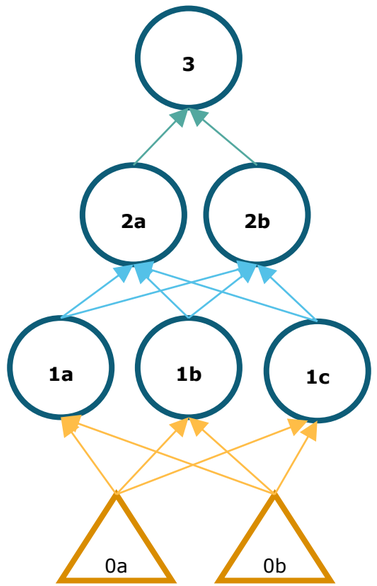
- Consider the case of an MLP with a two hidden layers (see below)
- The **adjacency matrix** of the graph associated to the network is shown on the right (where 1 = connected, 0 = no connections)
  - Square matrix
    - We assume that if cell *(i,j)* is 1 then we have a connection from *i* to *j*
  - Triangular matrix
    - Because edges are oriented
  - Zero-only diagonal
    - Because there are no self-loops
  - Sparse matrix
    - Connections from one layer to the following one are associated to a block of the matrix (see colors)



	0a	0b	1a	1b	1c	2a	2b	3
0a	0	0	1	1	1	0	0	0
0b	0	0	1	1	1	0	0	0
1a	0	0	0	0	0	1	1	0
1b	0	0	0	0	0	1	1	0
1c	0	0	0	0	0	1	1	0
2a	0	0	0	0	0	0	0	1
2b	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0

Directed Acyclic Graphs (3)

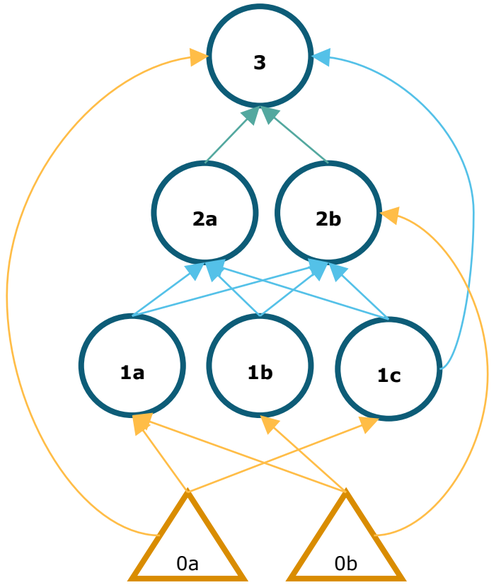
- Let's replace the 1's with the scalar values of the weights associated to the connections
- Each block is a weight matrix $(W^h)'$ where $'$ is the transpose operator



	0a	0b	1a	1b	1c	2a	2b	3
0a	0	0	$w_{1a,0a}$	$w_{1b,0a}$	$w_{1c,0a}$	0	0	0
0b	0	0	$w_{1a,0b}$	$w_{1b,0b}$	$w_{1c,0b}$	0	0	0
1a	0	0	0	0	0	$w_{2a,1a}$	$w_{2b,1a}$	0
1b	0	0	0	0	0	$w_{2a,1b}$	$w_{2b,1b}$	0
1c	0	0	0	0	0	$w_{2a,1c}$	$w_{2b,1c}$	0
2a	0	0	0	0	0	0	0	$w_{3,2a}$
2b	0	0	0	0	0	0	0	$w_{3,2b}$
3	0	0	0	0	0	0	0	0

Directed Acyclic Graphs (4)

- Consider the case of a more fancy network (see below)
- Again, the **adjacency matrix** of the graph associated to the network is shown on the right
  - Sparse matrix
    - The block-wise structure is now lost
    - Skip-layer connections



	0a	0b	1a	1b	1c	2a	2b	3
0a	0	0	1	0	1	0	0	1
0b	0	0	1	1	0	0	1	0
1a	0	0	0	0	0	1	1	0
1b	0	0	0	0	0	1	1	0
1c	0	0	0	0	0	1	1	1
2a	0	0	0	0	0	0	0	1
2b	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0

Learning with Neural Networks

Stefano Melacci

Learning from Data

- We consider the case of (fully) **supervised learning**
  - **c**-class **classification**
  - We have **c** neurons in the output layer, each of them associated to a class

- We are given a training set with *n* supervised pairs, where $\boldsymbol{x}_i$ is a training example and $\boldsymbol{y}_i$ is its target (supervision)

$$T = \{(\mathbf{x}_i, \mathbf{y}_i),\ i = 1, \ldots, n\}$$

  - The target is commonly represented as a binary 1-hot vector, where the position whose value is 1 is the class index
  - For example, in the case of 3 classes:

Class 1:	[1,0,0]
Class 2:	[0,1,0]
Class 3:	[0,0,1]

Learning from Data: Objective Function

- *Training the Neural Network consists in providing the training data to the network and optimizing its parameters (weights and biases) in order to minimize an objective function (**loss function**) that measures the mismatch between the network outputs and the supervisions (targets)*

- Loss functions (examples):
  - **Mean Squared Error (MSE)**

$$\texttt{obj} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left\|f(\mathbf{x}_i)-\mathbf{y}_i\right\|^2$$

This is the function computed by the neural network, and it depends on the values of the **weights** and **biases** of the net

  - **Cross-Entropy**

$$\texttt{obj} = -\frac{1}{n}\sum_{i=1}^{n}\mathbf{y}'_i\log(f(\mathbf{x}_i))$$

this loss makes sense only if we have a **probabilistic normalization** in the output layer (softmax)

Learning from Data: Objective Function (2)

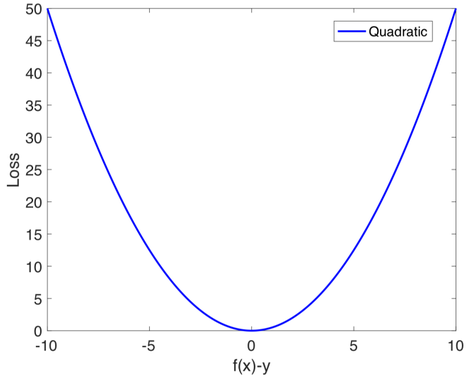
- **Mean Squared Error (MSE)**
  - Sum over the *n* training examples
    Divided by *n* (average)
  - No restrictions on the values of *f*
  - Always positive
  - It is zero $f(\mathbf{x}_i) = \mathbf{y}_i$ for all training examples

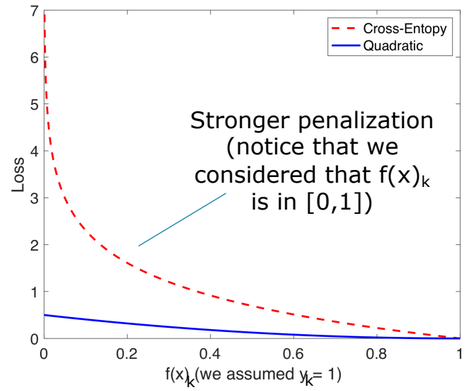
$$\texttt{obj} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left\|f(\mathbf{x}_i) - \mathbf{y}_i\right\|^2$$

- **Cross-Entropy**
  - Sum over the *n* training examples
    Divided by *n* (average)
  - Each component of the function *f* must be in [0,1]
    …and they must sum to 1
    and $\mathbf{y}_i$ *must* be 1-hot vector for all examples
  - Always positive
    The negative sign is needed: *log* is negative for values <1
  - It is zero when $f(\mathbf{x}_i)_{k(i)} = \mathbf{y}_{i,k(i)} = 1$ for all training examples
    The index *k(i)* is the one for which $\mathbf{y}_{i,k(i)} = 1$

$$\texttt{obj} = -\frac{1}{n}\sum_{i=1}^{n}\mathbf{y}_i'\log(f(\mathbf{x}_i))$$

For simplicity, we are considering a single training pair *(x,y)* in both the figures



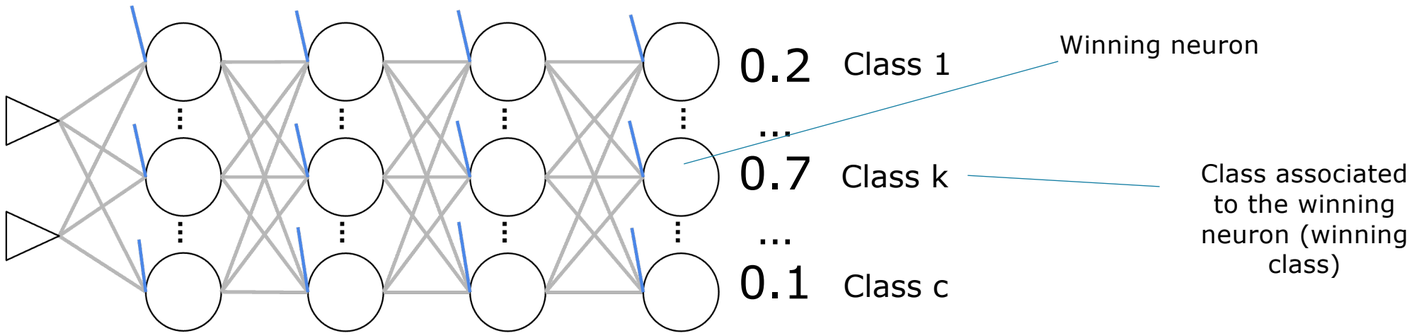


(in this figure we consider only the contribute of the *k*-th output neuron, the one for which y_k is 1)

Learning from Data: Decision Rule

- Once we are given a trained network and we make a prediction on a never-seen-before example, how do we decide what is the predicted class?
  - The network output *f(**x**)* is a vector of *c* scalar values (one for each output neuron)
  - **Winning class**: the class that corresponds to the neuron that yielded

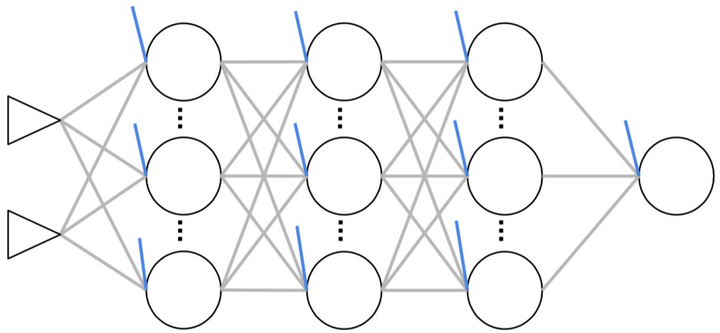
$$\max f(\mathbf{x})$$



Learning from Data: Decision Rule (2)

- Special case: **binary classification**
  - If we have $c = 2$, then we can either implement a network with $\boldsymbol{c}$ output neurons, or use a **single** output neuron
  - In the latter case, we compare the output score with a threshold (0.5, for example), and we predict the first class only if (otherwise we predict the second class)

$$f(\mathbf{x}) \geq \tau$$



0.63 Class 1/2

Both the classes are associated to the same output neuron (the winning class depends on the neuron output and the selected threshold)

If the decision threshold is set to 0.5, then the winning class is Class 1

Learning from Data: Decision Rule/Obj

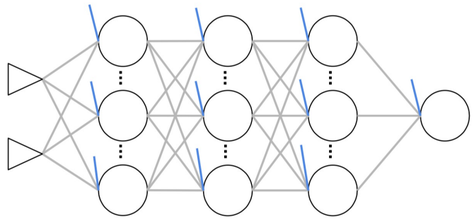
- Special case: **binary classification**
  - **Targets** y_i are scalar values in {0,1} where 1 is the target associated to the first class and 0 is the target associated to the second class)
  - In this case, we cannot use the previously defined cross-entropy loss, but the **binary cross-entropy** one

$$\texttt{obj} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log(f(\mathbf{x}_i)) + (1-y_i)\log(1-f(\mathbf{x}_i))\right)$$

this loss makes sense only if we have network output **in [0,1]**

Loss for examples of the first class (y_i = *1*), **when *f(x_i) = 1* we have no loss**

Loss for examples of the second class (y_i = *0*), **when *f(x_i) = 0* we have no loss**

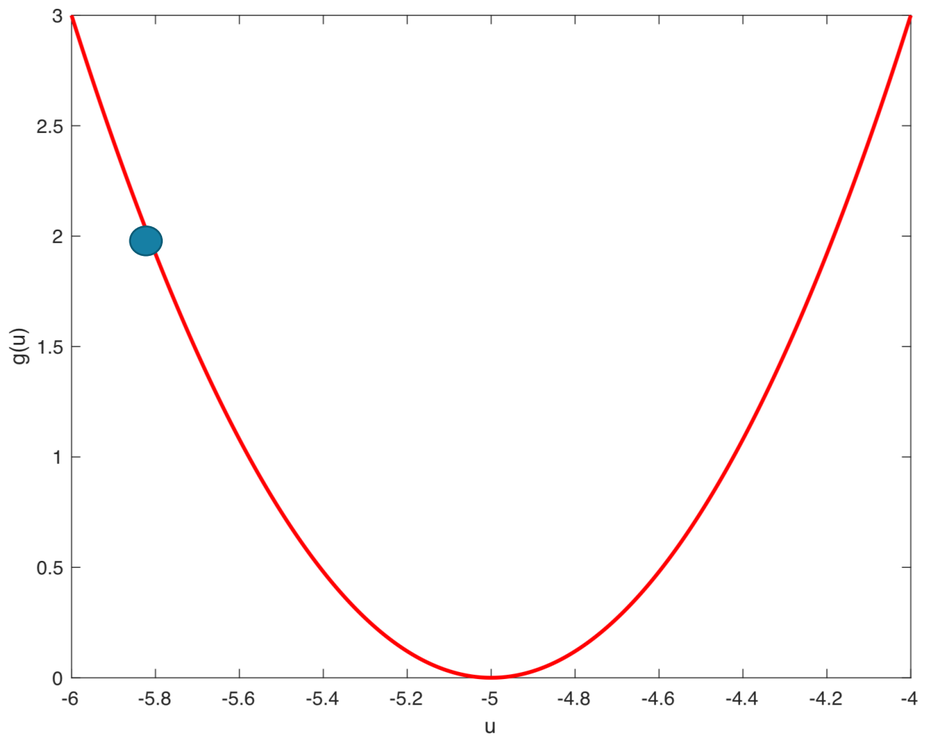


Learning from Data: Gradient-based Opt

- Neural networks are trained by means of gradient-based optimization
  - **Gradient descent**
- The basic principle behind gradient descent consists in computing the gradient of a function with respect to the parameters that we want to optimize
- The "*negative gradient*" is a descent direction, in the sense that if we update the parameters following such direction, the considered function decreases
- When we reach the minimum of the considered function, the gradient is zero
  - No need to descend anymore

Learning from Data: Gradient-based Opt (2)

- Example



$$g(u) = 3(u+5)^2$$

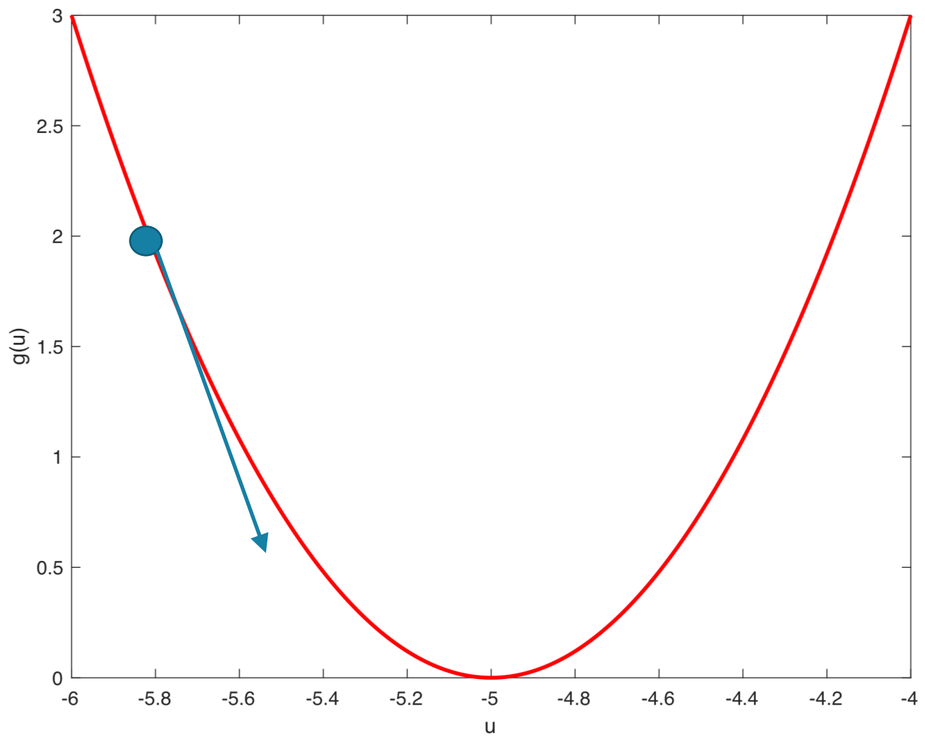
$$u = -5.8$$

$$\rho = 0.1$$

Step size

Learning from Data: Gradient-based Opt (3)

- Example



$$g(u) = 3(u+5)^2$$

$$u = -5.8$$

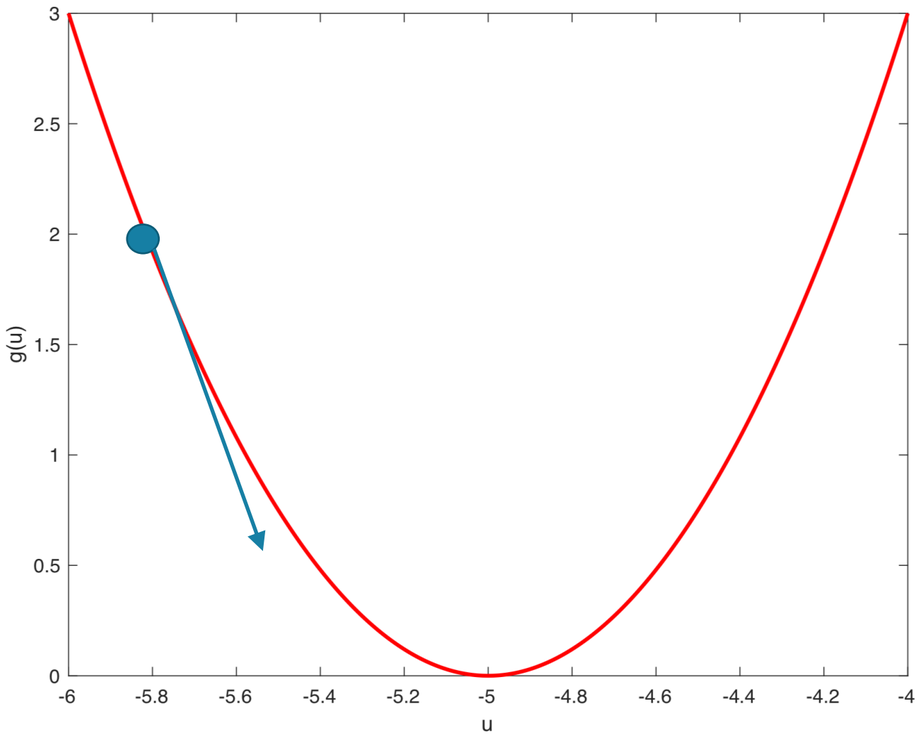
$$\rho = 0.1$$

$$\nabla g(u) = 6(u+5)$$

$$\nabla g(u)\big|_{u=-5.8} = 6(-5.8+5) = -4.8$$

Learning from Data: Gradient-based Opt (4)

- Example



$$g(u) = 3(u+5)^2$$

$$u = -5.8$$

$$\rho = 0.1$$

$$\nabla g(u) = 6(u+5)$$

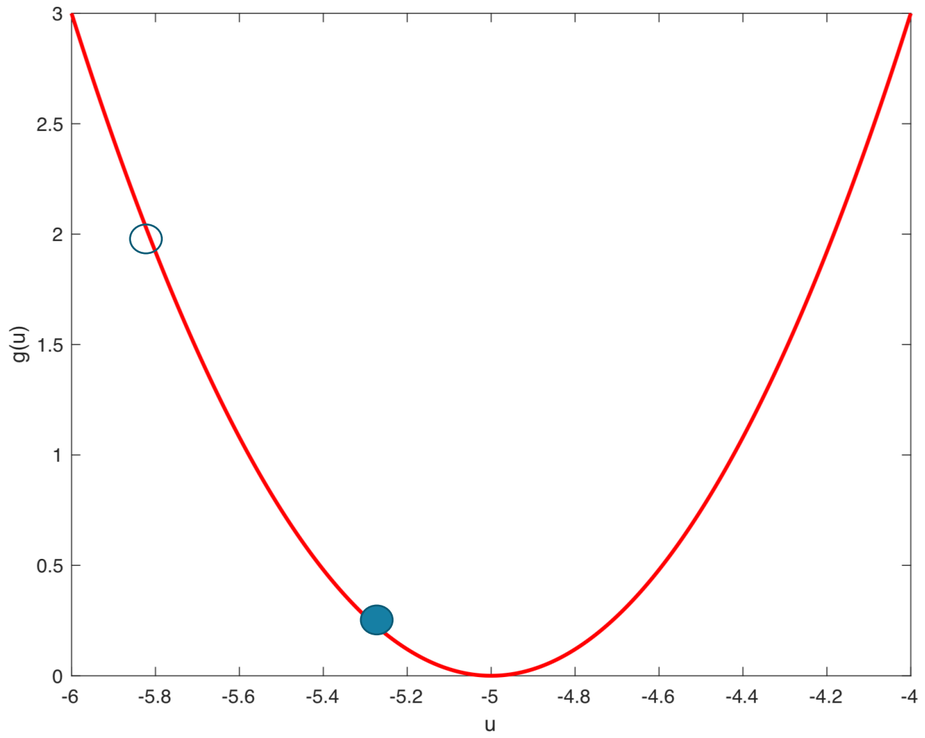
$$\nabla g(u)\big|_{u=-5.8} = 6(-5.8+5) = -4.8$$

$$u = u - \rho \nabla g(u)$$

Update rule: single step in the direction of the (negative) gradient

Learning from Data: Gradient-based Opt (5)

- Example



$$g(u) = 3(u+5)^2$$

$$u = -5.8$$

$$\rho = 0.1$$

$$\nabla g(u) = 6(u+5)$$

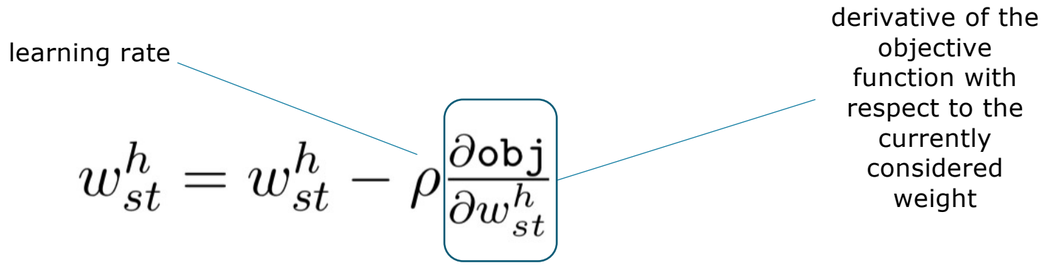
$$\nabla g(u)\big|_{u=-5.8} = 6(-5.8+5) = -4.8$$

$$u = u - \rho \nabla g(u)$$

$$= -5.8 + \rho 4.8 = -5.32$$

Learning from Data: Gradient Descent

- How can we minimize the objective (loss) function? **Gradient descent**
  1. Weights are initialized to (small) random values
  2. The gradient of the objective function with respect to the weights and biases is computed
  3. Each weight (and bias) is updated with the following rule



where ρ is a scalar value that determines the step-size of the update (**learning rate**)

- Several techniques to adaptively adjust its value (*Adam*)

Backpropagation

- **Backpropagation (BP)** *is an efficient way to compute the gradient of the objective function with respect to all the weights (and biases) of the network*
  - It is one the main motivations behind the popularity of neural networks in the late eighties and early nineties

- Its origins can be traced back to the work of *Paul Werbos* (1974, PhD Thesis - Harvard University), with the first ideas to apply the algorithm to multilayered networks
  - Related work, not in the context of neural networks, was already circulating in the scientific community even earlier
  - *Rumelhart, Hinton, Williams* (1986) popularized and made concrete the Backpropagation algorithm
    - Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Nature* (1986)

Backpropagation (2)

- Going back to gradient descent…

$$w_{st}^h = w_{st}^h - \rho \frac{\partial \texttt{obj}}{\partial w_{st}^h}$$

Backpropagation allows us to efficiently compute this term!

- Main ingredients of Backpropagation
  - **Chain rule**: if a variable *a* depends on another variable *k*, that, in turn, depends on *b*, then *a* depends on *b* as well, and we can compute the derivative of *a* with respect to *b* as follows:

$$\frac{d\texttt{a}}{d\texttt{b}} = \frac{d\texttt{a}}{dk} \cdot \frac{dk}{d\texttt{b}}$$

  - This concept is used to compute the derivatives of **composite functions**.
  - Recall that an MLP computes:

$$f(\mathbf{x}) = \sigma\left(W^\ell \sigma\left(W^{\ell-1} \sigma\left(W^{\ell-2}\mathbf{x}\right)\right)\right)$$

Backpropagation (3)

- Going back to gradient descent…

$$w_{st}^{h} = w_{st}^{h} - \rho \frac{\partial \texttt{obj}}{\partial w_{st}^{h}}$$

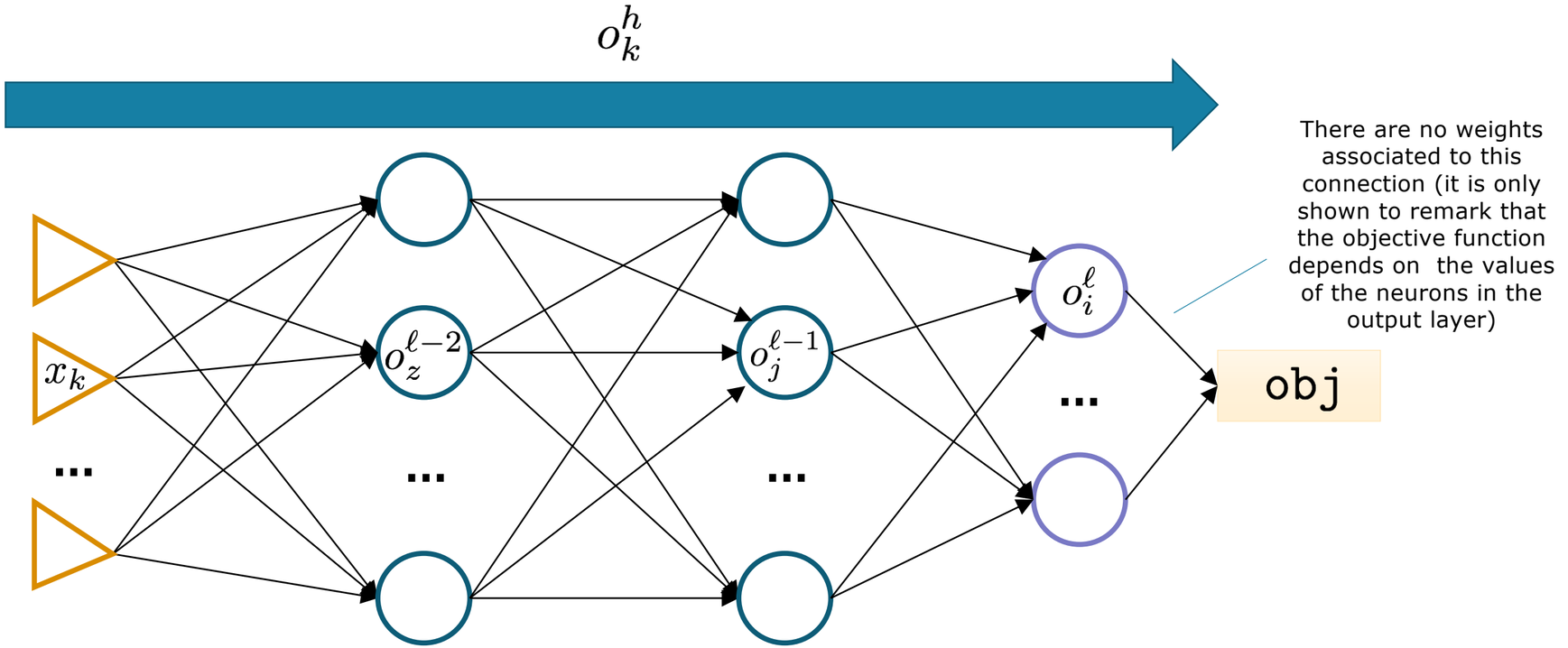
Backpropagation allows us to efficiently compute this term!

- Main ingredients of Backpropagation
  - Storing the **output** of each neuron
    - **Forward stage**
  - Computing derivatives **following the network structure**
    - **Backward stage**

  - Storing the result of **intermediate computations** that are needed in the process of differentiation, allowing the algorithm to avoid repeating those computations that were already done
    - The so-called **deltas**

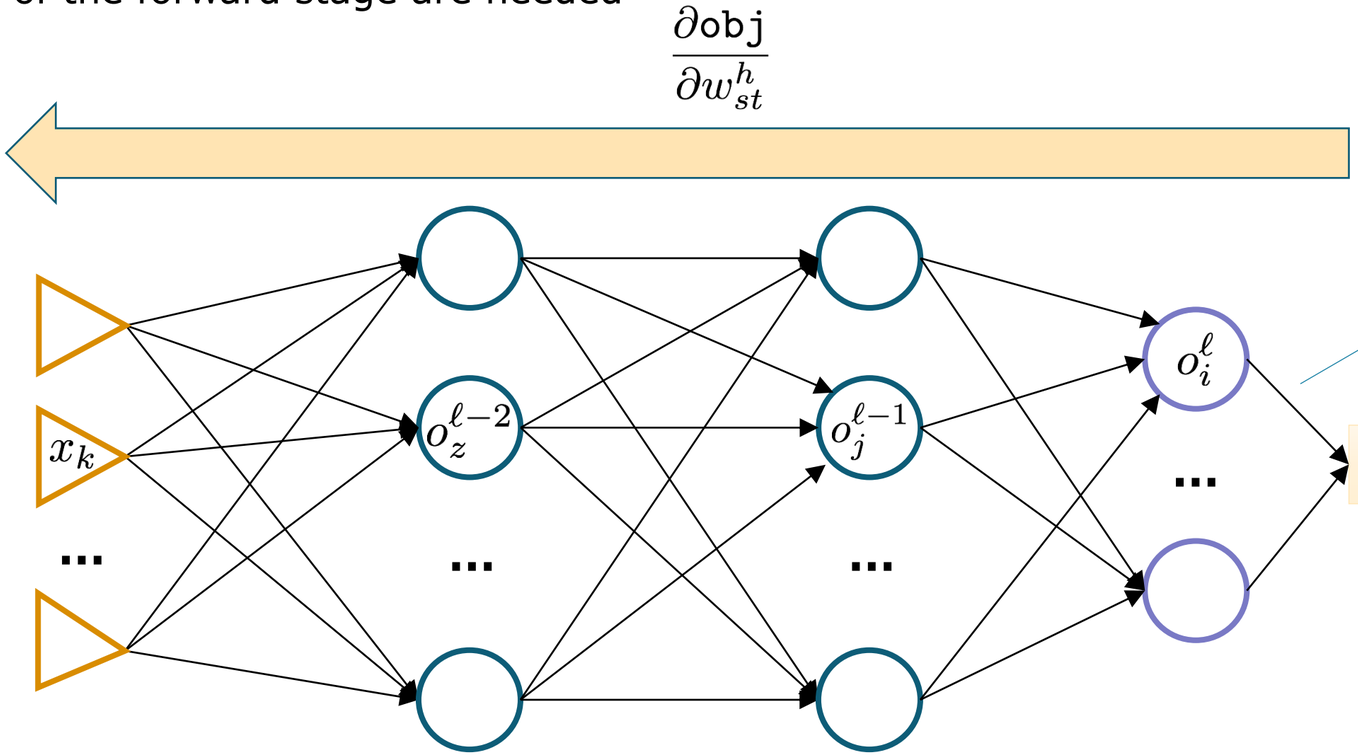
Backpropagation (4)

- **Forward stage**
  - *Let us consider the case of a **single training example***
  - Forward: compute the output of each neuron of the network (and store it)



Backpropagation (5)

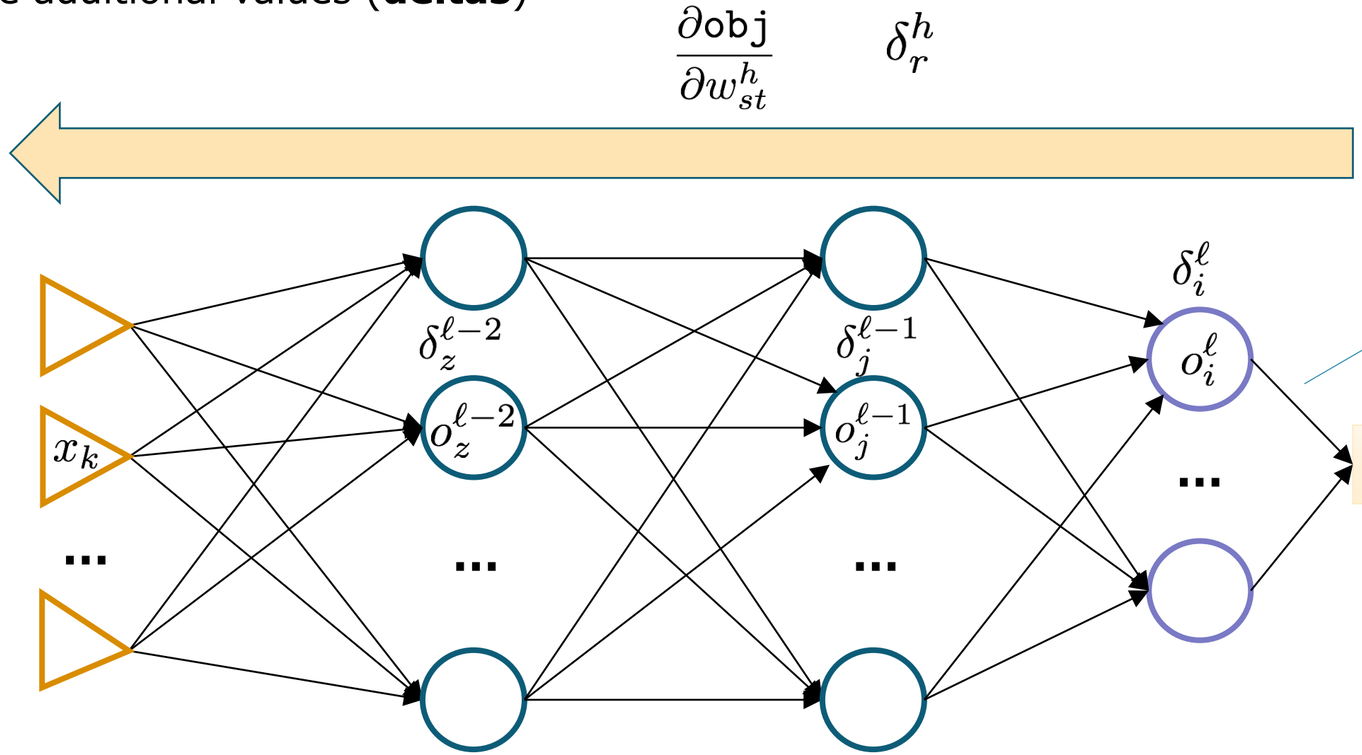
- **Backward stage**
  - Backward: compute derivatives (signal backprop)
    - The outputs of the forward stage are needed



There are no weights associated to this connection (it is only shown to remark that the objective function depends on the values of the neurons in the output layer)

Backpropagation (6)

- **Backward**
  - Compute derivatives (signal backprop)
    - We will store additional values (**deltas**)



There are no weights associated to this connection (it is only shown to remark that the objective function depends on the values of the neurons in the output layer)