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)

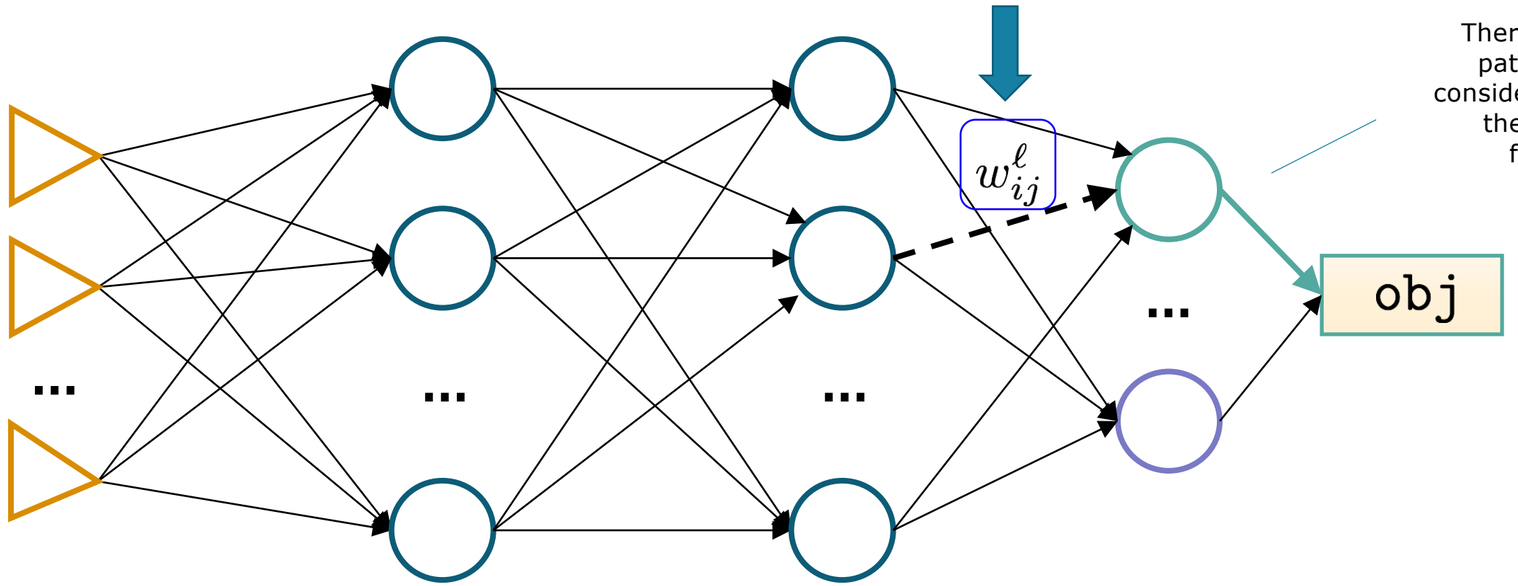
They are used with the sole purpose of supporting Stefano Melacci's teaching activity.

They are not intended to be of public domain in any way, and they must not be published or shared out of the context in which they are presented by Stefano Melacci.

Details of Backpropagation

Stefano Melacci

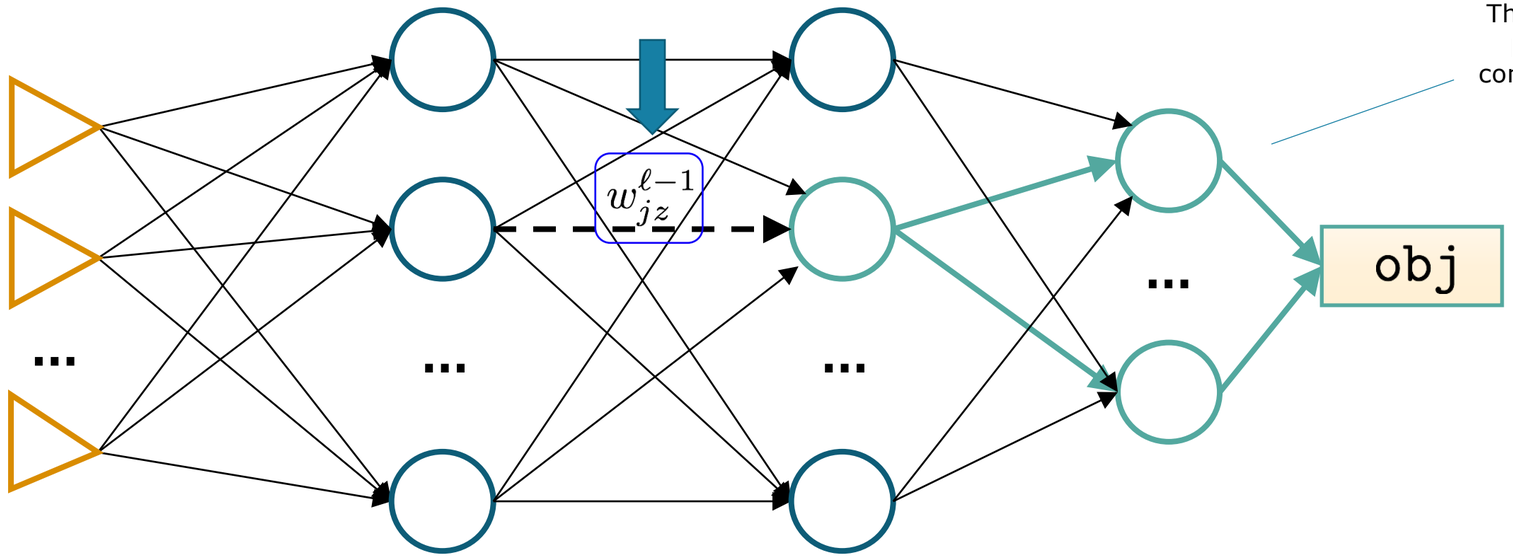
Backpropagation: Details



- In order to better understand Backpropagation, let us focus on *which portions of the network are affected by a variation of a certain weight*
  - **CASE 1: last layer weight** (or, equivalently, a weight of a network with no hidden layers)
    - The greenish elements depends on the value of the considered weight (output neurons, objective function)

Copyright Stefano Melacci (University of Siena)

Backpropagation: Details (2)

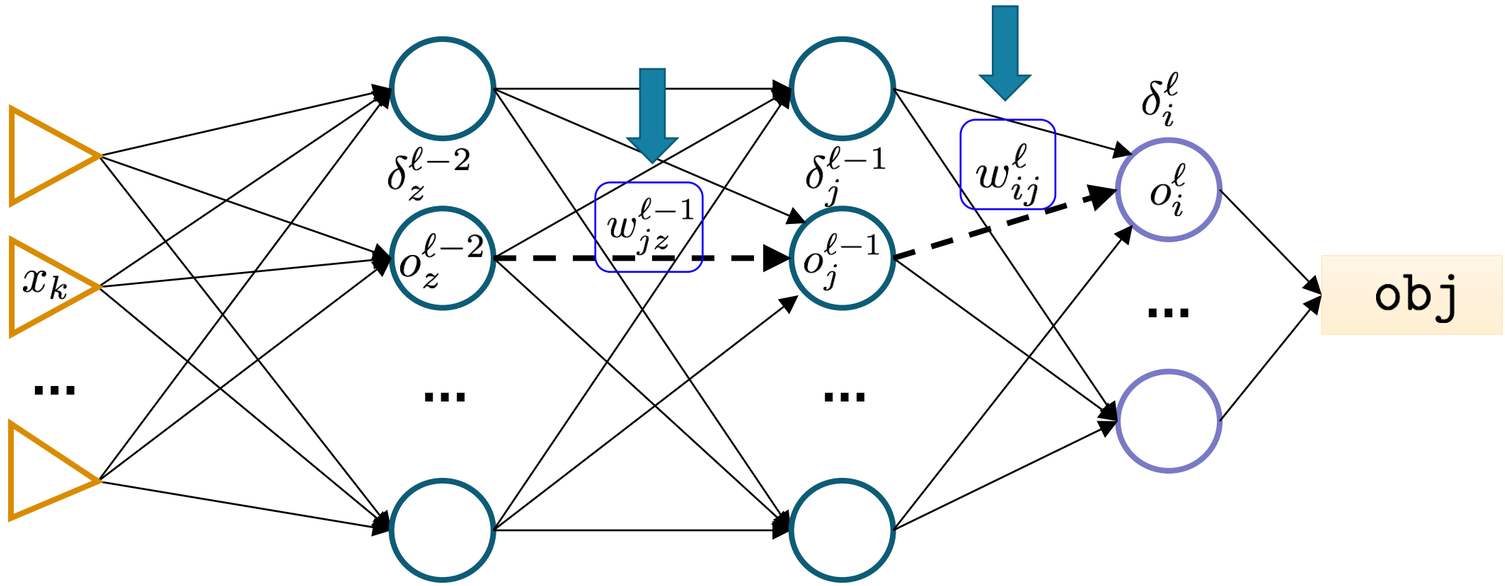


There are **multiple paths** from the considered weight to the objective function!

- **CASE 2: hidden layer weight**
  - The greenish elements depends on the value of the considered weight
  - Multiple paths from the considered weight to the objective function
  - Backpropagation provides the tools for efficiently computing the derivative of the objective function also with respect to the considered weight

Copyright Stefano Melacci (University of Siena)

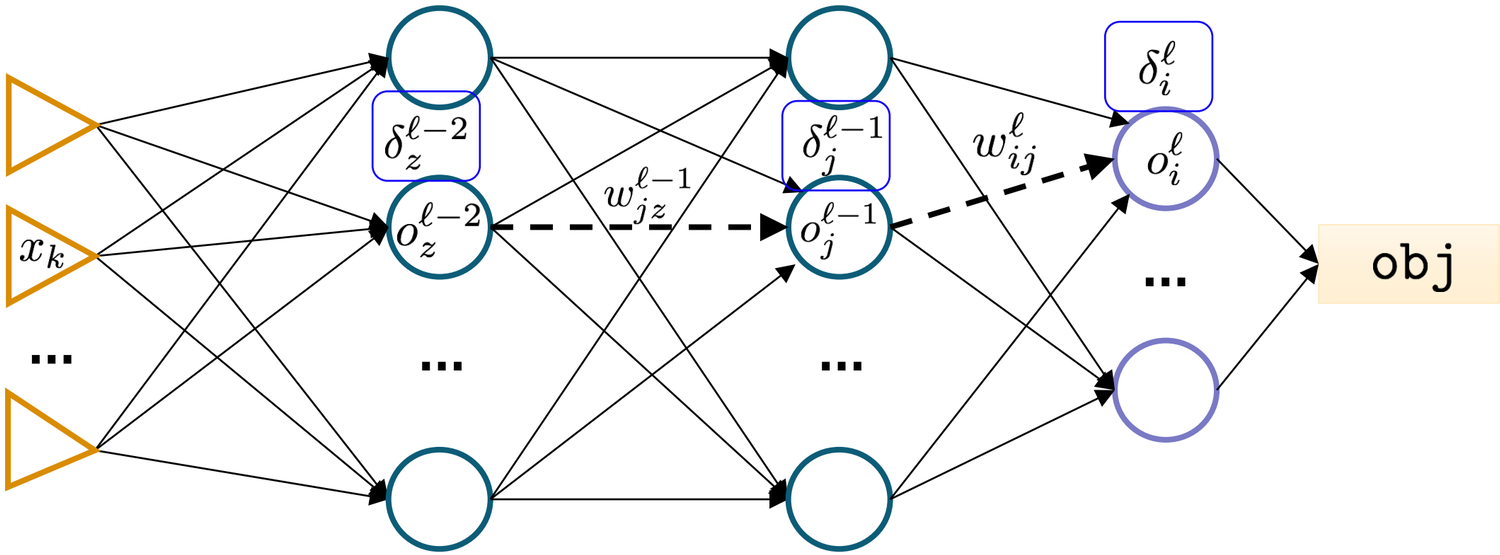
Backpropagation: Details (3)



- As a result of the previous considerations, in order to devise the gradient computation scheme of Backpropagation, we will distinguish between the case of weights connecting the output layer and the case of the other weights of the network

Copyright Stefano Melacci (University of Siena)

Backpropagation: Details (4)



- **Delta** variables are defined as the *derivatives of the objective function with respect to the neuron activation*

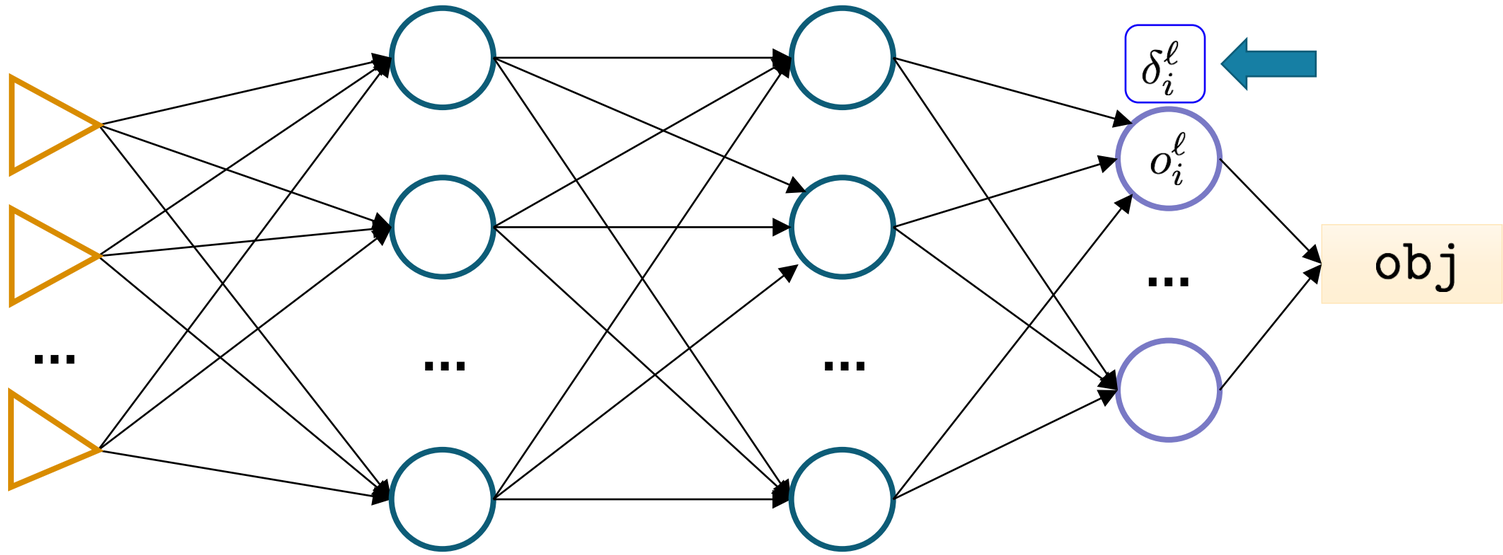
$$\delta_r^h = \frac{\partial \texttt{obj}}{\partial a_r^h}$$

Copyright Stefano Melacci (University of Siena)

Last Layer Deltas

Output of neuron "z" in layer "h"

$$o_z^h = \sigma(a_z^h) = \sigma\left(\sum_s w_{zs}^h o_s^{h-1}\right)$$



- In the case of deltas belonging to the output layer, we have (recall that the neuron output depends on the activation score)

$$\delta_i^\ell = \frac{\partial \texttt{obj}}{\partial a_i^\ell} = \frac{\partial \texttt{obj}}{\partial o_i^\ell}\frac{\partial o_i^\ell}{\partial a_i^\ell}$$

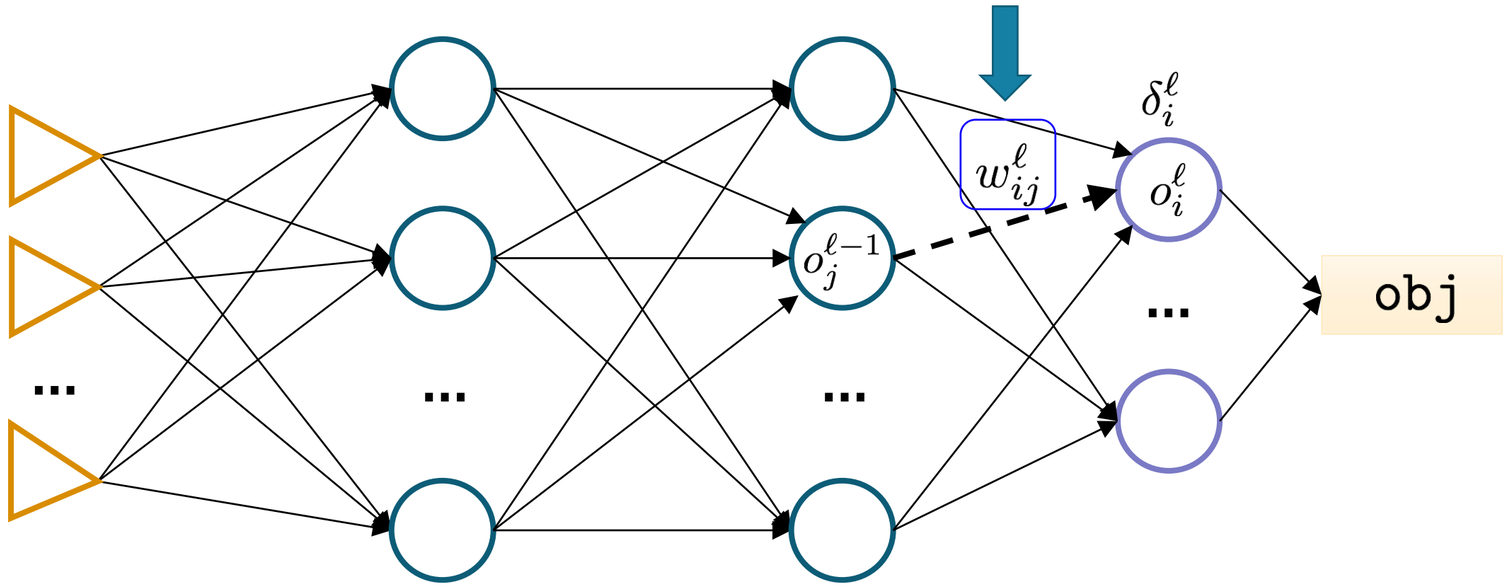
derivative of the activation function with respect to its argument

Copyright Stefano Melacci (University of Siena)

Gradient w.r.t. Last Layer Weights

Output of neuron "z" in layer "h"

$$o_z^h = \sigma(a_z^h) = \sigma\left(\sum_s w_{zs}^h o_s^{h-1}\right)$$



- In order to compute the derivative of the objective function with respect to the weight that is indicated in the picture, we have

Delta of the destination neuron, output of the source neuron

$$\frac{\partial \text{obj}}{\partial w_{ij}^\ell} = \frac{\partial \text{obj}}{\partial a_i^\ell}\frac{\partial a_i^\ell}{\partial w_{ij}^\ell} = \delta_i^\ell \frac{\partial a_i^\ell}{\partial w_{ij}^\ell} = \delta_i^\ell o_j^{\ell-1}$$

This is what we need to update the value of the considered weight!
(gradient descent)

Copyright Stefano Melacci (University of Siena)

Last Layer Deltas (example)

- For example, let's consider the case of an objective function based on the MSE, and, as we already did so far, a single training example

$$\texttt{obj} = \frac{1}{2}\|f(\mathbf{x}) - \mathbf{y})\|^2$$

  - If we only focus on the output of the *i*-th neuron of the last layer it becomes:

$$\texttt{obj} = \frac{1}{2}\left(f(\mathbf{x})_i - y_i\right)^2 = \frac{1}{2}\left(o_i^\ell - y_i\right)^2 \qquad o_i^\ell = \sigma\left(a_i^\ell\right)$$

- Then, we have

$$\delta_i^\ell = \frac{\partial \texttt{obj}}{\partial a_i^\ell} = \frac{\partial \texttt{obj}}{\partial o_i^\ell}\frac{\partial o_i^\ell}{\partial a_i^\ell} \quad \Rightarrow \quad \delta_i^\ell = \left(o_i^\ell - y_i\right) \cdot \sigma'(a_i^\ell)$$

(previous slides)

Copyright Stefano Melacci (University of Siena)

Gradient w.r.t. Last Layer Weights (example)

- Still in the previously considered case of the MSE, we can compute the derivative with respect to a weight connecting the last layer as follows

$$\delta_i^\ell = \left(o_i^\ell - y_i\right) \cdot \sigma'\left(a_i^\ell\right)$$

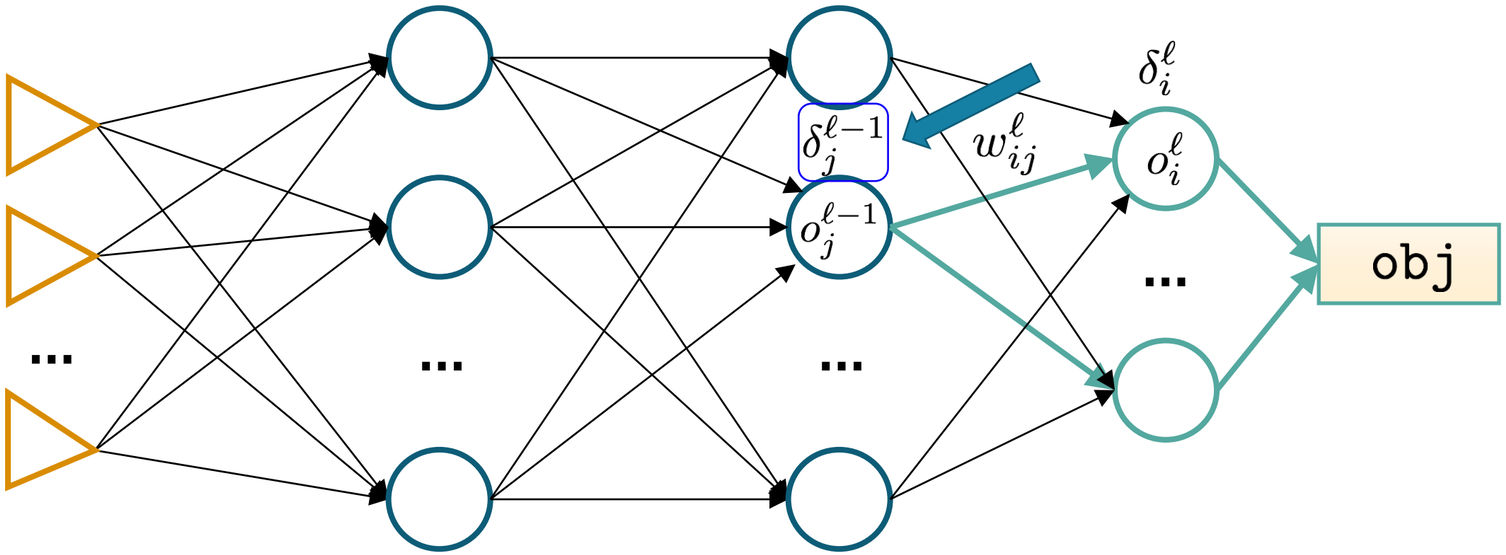
(previous example)

$$\frac{\partial \texttt{obj}}{\partial w_{ij}^\ell} = \delta_i^\ell o_j^{\ell-1} \quad \Rightarrow \quad \left(o_i^\ell - y_i\right) \cdot \sigma'\left(a_i\right) \cdot o_j^{\ell-1}$$

(previous slides)

Copyright Stefano Melacci (University of Siena)

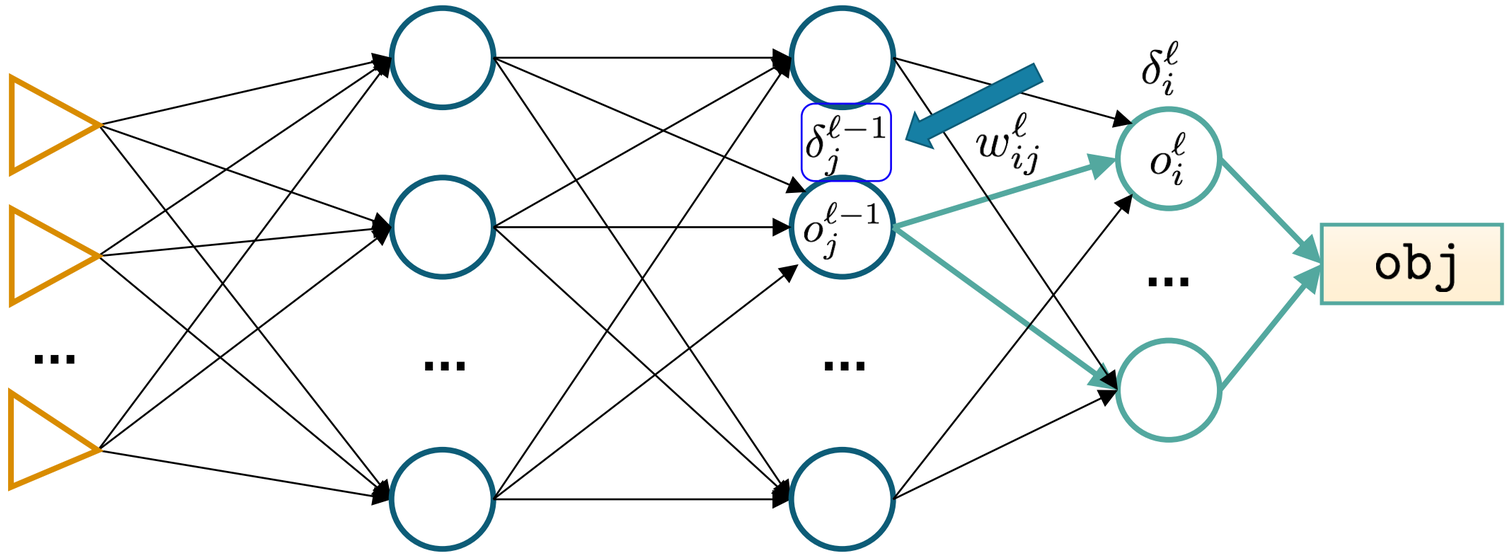
Hidden Layer Deltas



- In the case of deltas belonging to a neuron of the last hidden layer…
  1. All the outputs of the neurons of the last layer depends on the output of the considered neuron
  2. Recall that the output of each neuron is function of its activation
  - As a result, the activation of each neuron in the output layer depends on the activation of the considered neuron (each green paths in the picture depicts a dependency)

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (2)



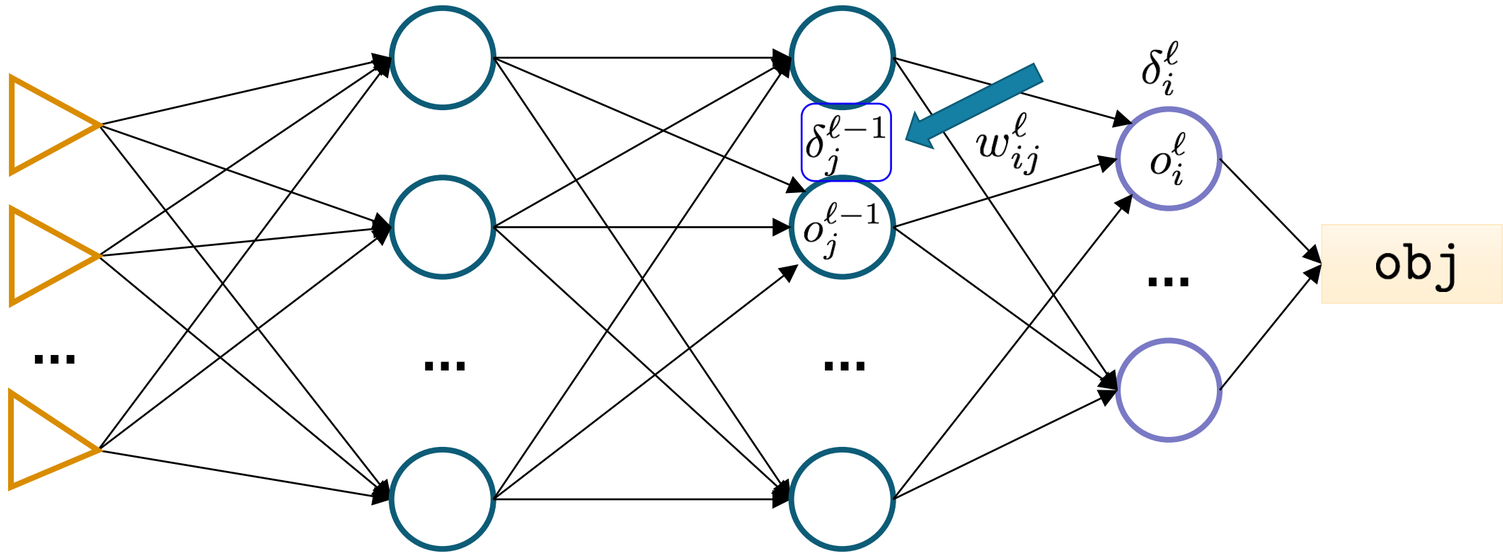
- We can apply the chain rule on the definition of delta

$$\delta_j^{\ell-1} = \frac{\partial \texttt{obj}}{\partial a_j^{\ell-1}} = \sum_i \frac{\partial \texttt{obj}}{\partial a_i^\ell} \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}}$$

We simply applied the chain rule over each of the green paths, and then we summed up the result

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (3a)

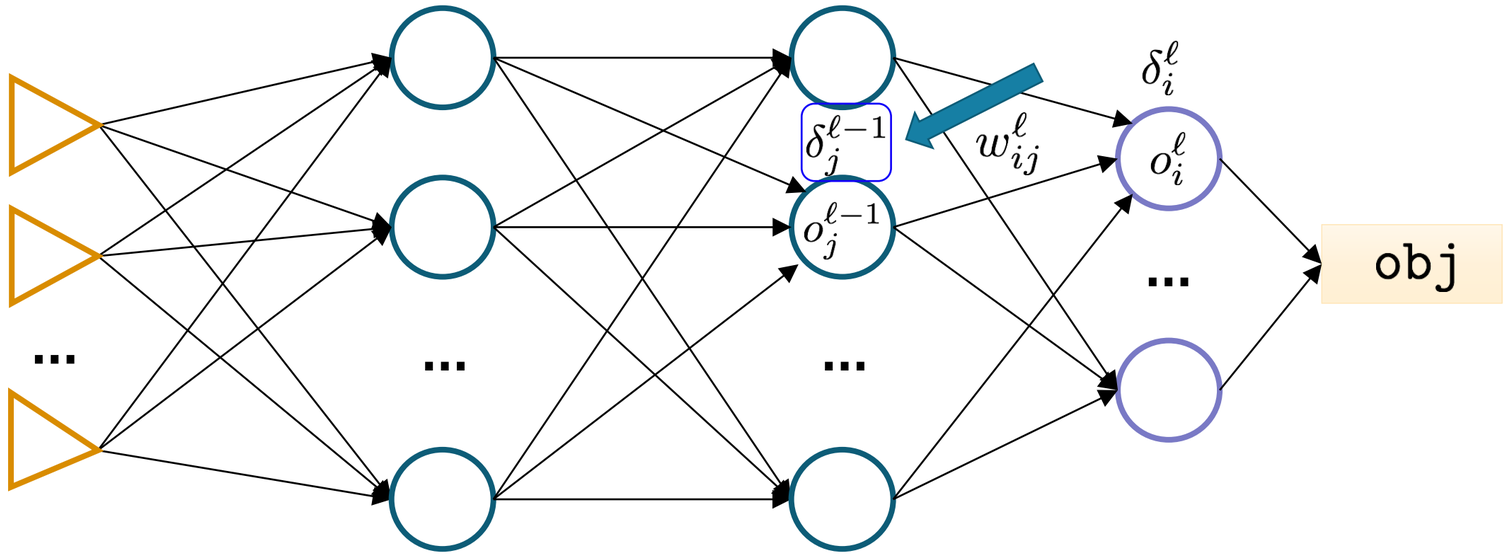


$$\delta_j^{\ell-1} = \frac{\partial \texttt{obj}}{\partial a_j^{\ell-1}} = \sum_i \frac{\partial \texttt{obj}}{\partial a_i^\ell} \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}}$$

Previous slide

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (3b)



Definition of delta

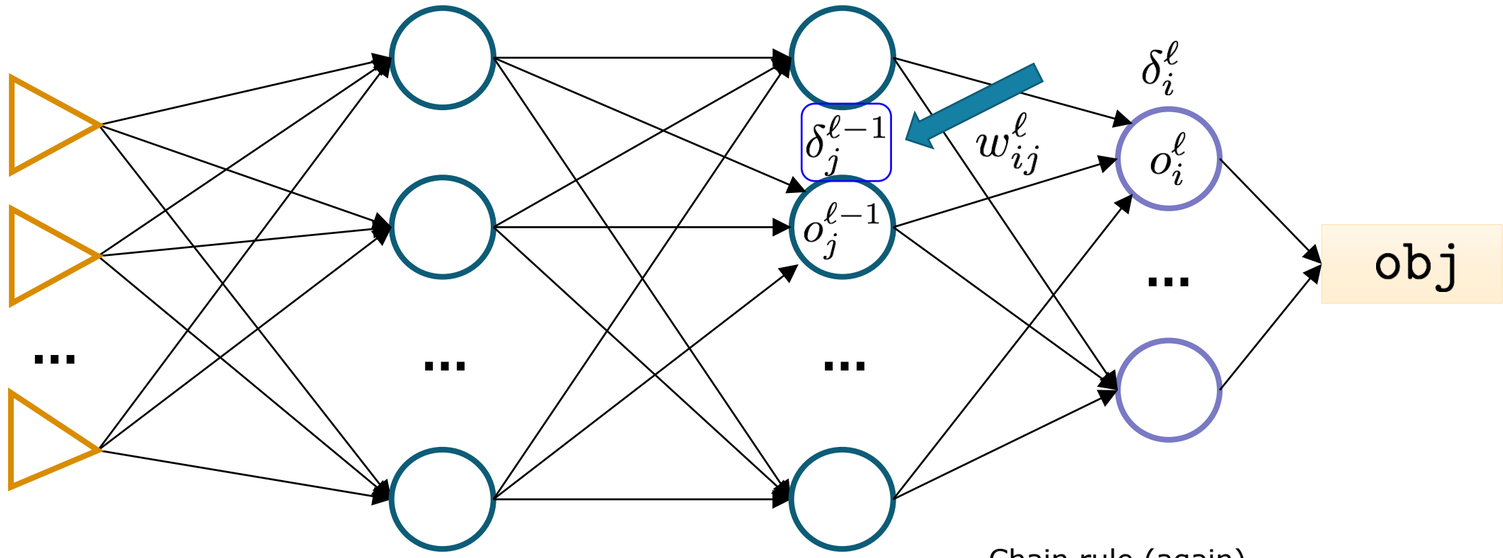
$$\delta_j^{\ell-1} = \frac{\partial \texttt{obj}}{\partial a_j^{\ell-1}} = \sum_i \frac{\partial \texttt{obj}}{\partial a_i^\ell} \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}} = \sum_i \delta_i^\ell \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}}$$

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (3c)

Output of neuron "z" in layer "h"

$$o_z^h = \sigma(a_z^h) = \sigma\left(\sum_s w_{zs}^h o_s^{h-1}\right)$$



Chain rule (again)

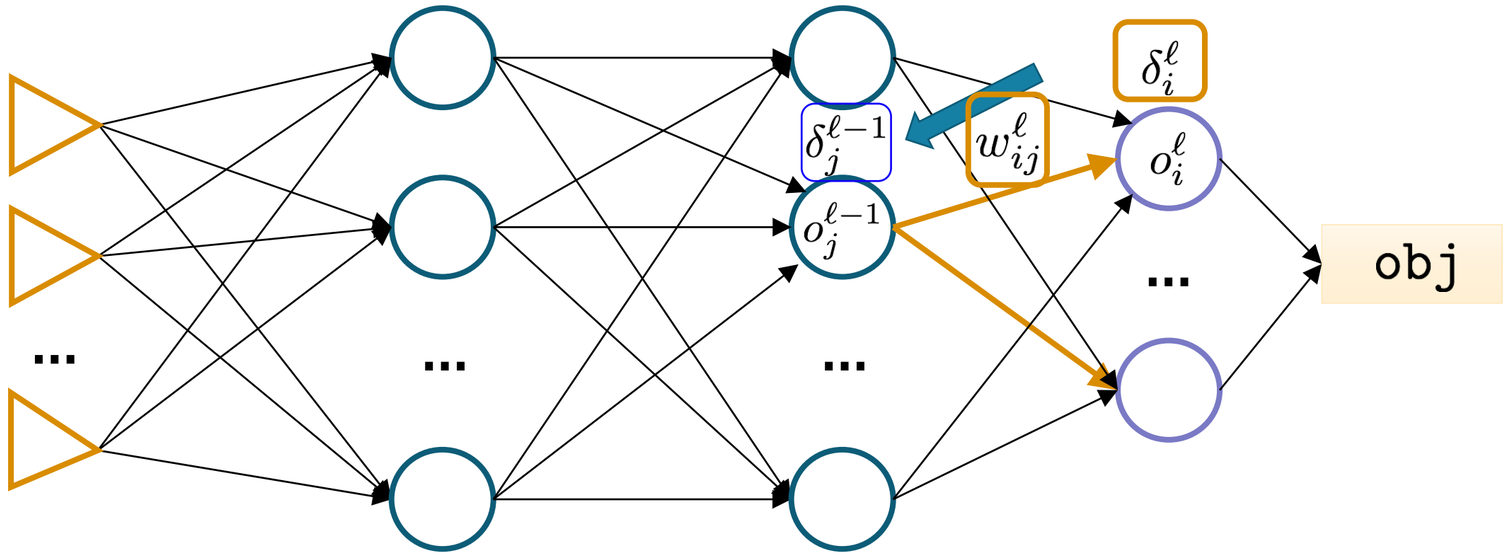
$$\delta_j^{\ell-1} = \frac{\partial \texttt{obj}}{\partial a_j^{\ell-1}} = \sum_i \frac{\partial \texttt{obj}}{\partial a_i^\ell}\frac{\partial a_i^\ell}{\partial a_j^{\ell-1}} = \sum_i \delta_i^\ell \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}} = \sum_i \delta_i^\ell \frac{\partial a_i^\ell}{\partial o_j^{\ell-1}}\frac{\partial o_j^{\ell-1}}{\partial a_j^{\ell-1}}$$

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (4)

Output of neuron "z" in layer "h"

$$o_z^h = \sigma(a_z^h) = \sigma\left(\sum_s w_{zs}^h o_s^{h-1}\right)$$



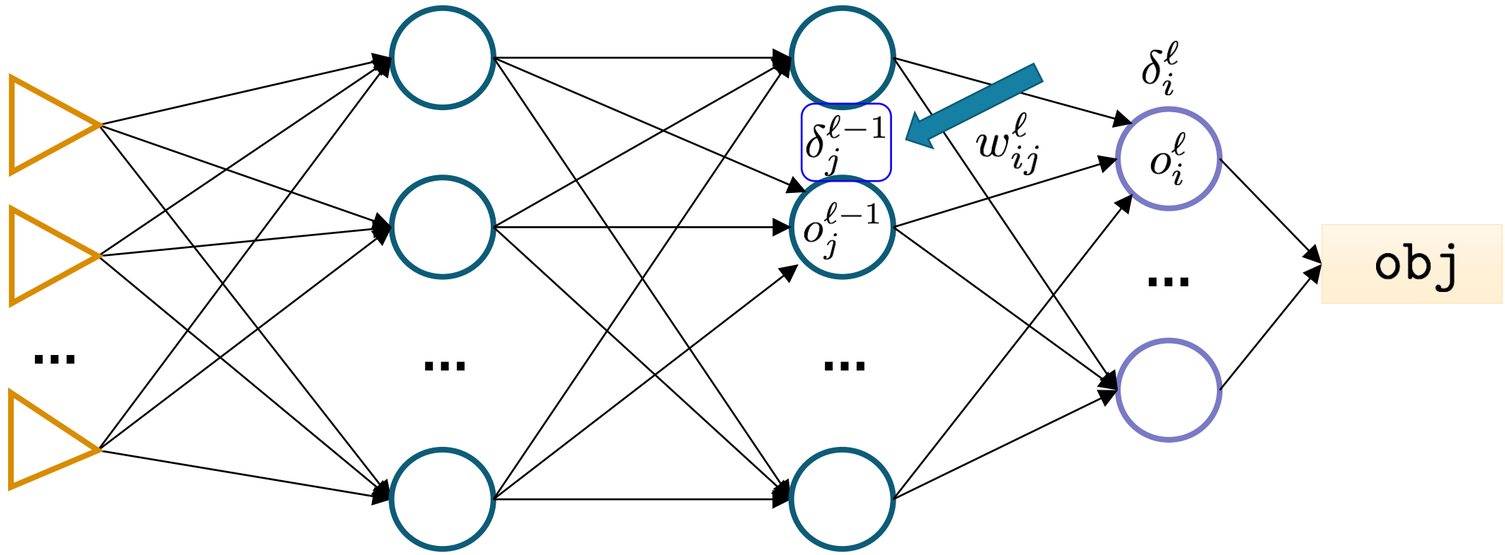
This sum involves all the deltas of the layer above, and all the connections going from the currently considered neuron to the neurons of the layer above

$$\delta_j^{\ell-1} = \frac{\partial \texttt{obj}}{\partial a_j^{\ell-1}} = \sum_i \frac{\partial \texttt{obj}}{\partial a_i^\ell}\frac{\partial a_i^\ell}{\partial a_j^{\ell-1}} = \sum_i \delta_i^\ell \frac{\partial a_i^\ell}{\partial a_j^{\ell-1}} = \sum_i \delta_i^\ell \frac{\partial a_i^\ell}{\partial o_j^{\ell-1}}\frac{\partial o_j^{\ell-1}}{\partial a_j^{\ell-1}} = \frac{\partial o_j^{\ell-1}}{\partial a_j^{\ell-1}}\sum_i \delta_i^\ell w_{ij}^\ell$$

derivative of the activation function with respect to its argument

Copyright Stefano Melacci (University of Siena)

Hidden Layer Deltas (5)



- We can rewrite the equation in **general way**, leading to a *rule to update deltas of a layer using the deltas of the layer above*

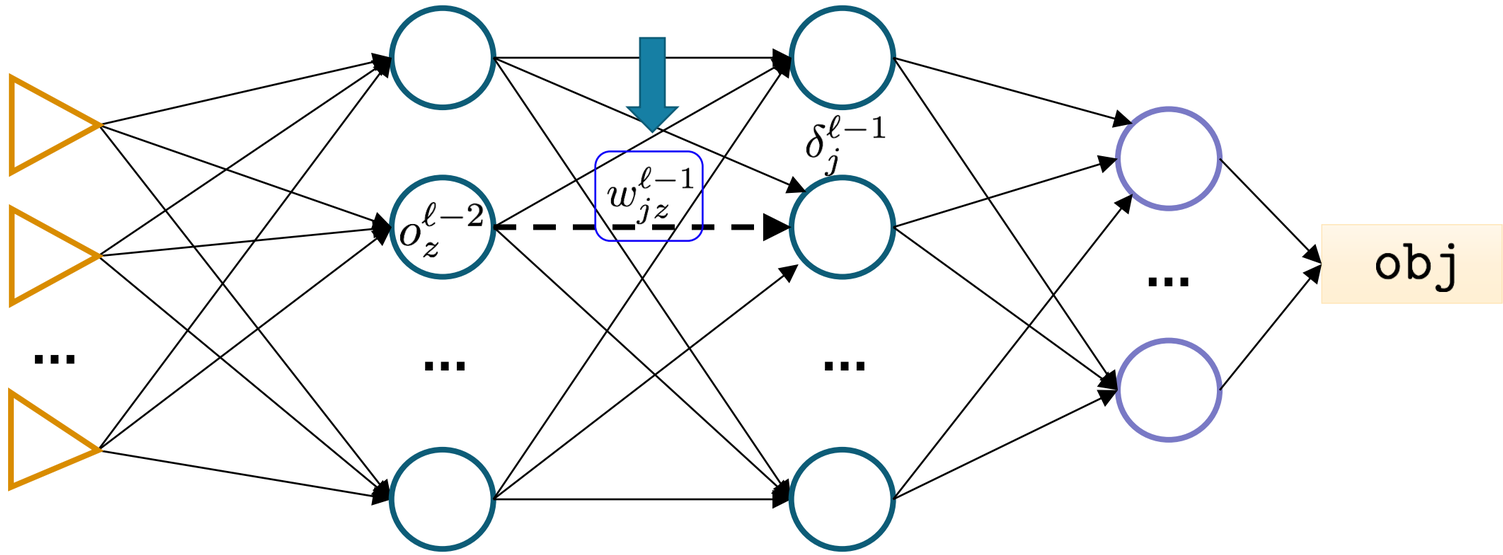
$$\underbrace{\delta_j^{\ell-1} = \frac{\partial o_j^{\ell-1}}{\partial a_j^{\ell-1}} \sum_i \delta_i^\ell w_{ij}^\ell}_{\text{Previous slide}} \Rightarrow \delta_j^{h-1} = \frac{\partial o_j^{h-1}}{\partial a_j^{h-1}} \sum_i \delta_i^h w_{ij}^h, \quad \forall h = 2, \ldots, \ell$$

Copyright Stefano Melacci (University of Siena)

Gradient w.r.t. Hidden Layer Weights

Output of neuron "z" in layer "h"

$$o_z^h = \sigma(a_z^h) = \sigma\left(\sum_s w_{zs}^h o_s^{h-1}\right)$$



- In order to compute the derivative of the objective function with respect to the weight that is indicated in the picture, we have
  - Same formula that we applied in the case of the output layer!

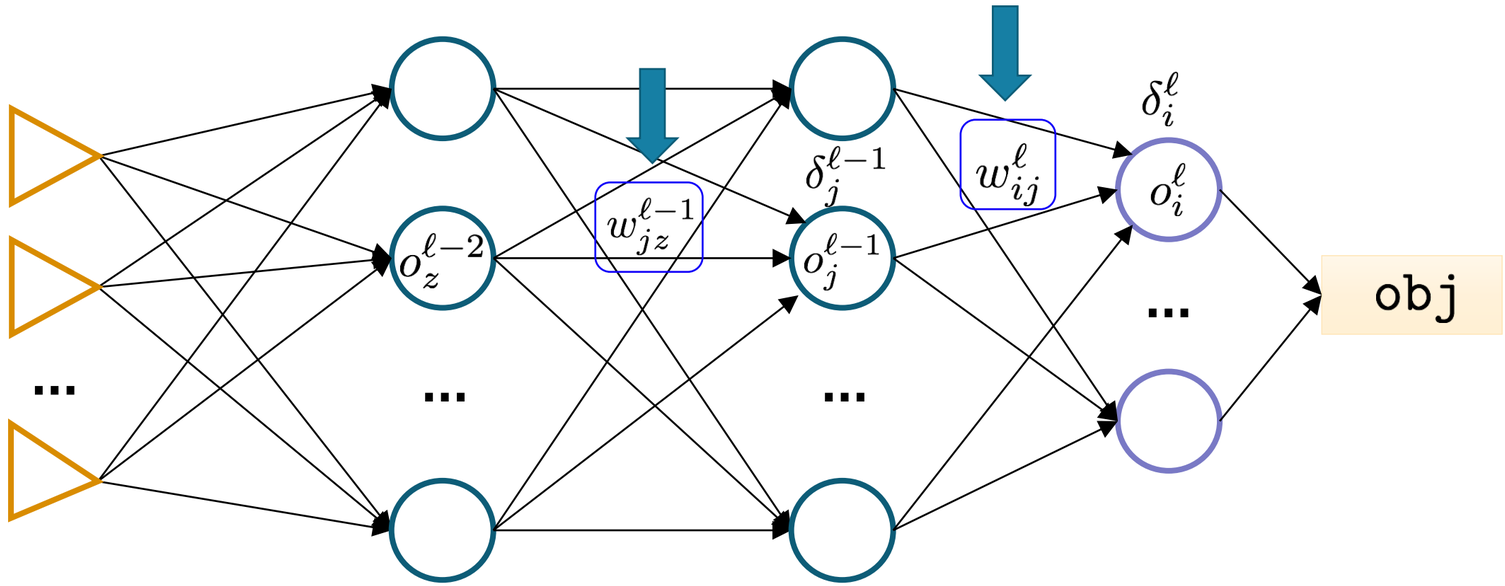
Delta of the destination neuron, output of the source neuron

$$\frac{\partial \text{obj}}{\partial w_{jz}^{\ell-1}} = \frac{\partial \text{obj}}{\partial a_j^{\ell-1}} \frac{\partial a_j^{\ell-1}}{\partial w_{jz}^{\ell-1}} = \delta_j^{\ell-1} \frac{\partial a_j^{\ell-1}}{\partial w_{jz}^{\ell-1}} = \delta_j^{\ell-1} o_z^{\ell-2}$$

This is what we need to update the value of the considered weight!
(gradient descent)

Copyright Stefano Melacci (University of Siena)

Gradient w.r.t. Hidden Layer Weights (2)



- Again, this is general, and it holds for all the weights of the network

$$\underbrace{\frac{\partial \texttt{obj}}{\partial w_{jz}^{\ell-1}} = \delta_j^{\ell-1} o_z^{\ell-2}}_{\text{Previous slide}} \quad \Rightarrow \quad \frac{\partial \texttt{obj}}{\partial w_{jz}^{h}} = \delta_j^{h} o_z^{h-1}$$

Copyright Stefano Melacci (University of Siena)

Summary

- *Deltas* of the **output** layer:

$$\delta_i^\ell = \frac{\partial \texttt{obj}}{\partial a_i^\ell} = \frac{\partial \texttt{obj}}{\partial o_i^\ell} \frac{\partial o_i^\ell}{\partial a_i^\ell}$$

- *Deltas* of the **hidden** layers:

$$\delta_j^{h-1} = \frac{\partial o_j^{h-1}}{\partial a_j^{h-1}} \sum_i \delta_i^h w_{ij}^h, \quad \forall h = 2, \ldots, \ell$$

- **Derivative with respect to a weight of the network**

$$\frac{\partial \texttt{obj}}{\partial w_{jz}^h} = \delta_j^h o_z^{h-1}$$

This is derivative with respect to the contribute of a **single example** to the loss function. We have to compute this quantity for all the training examples and **sum them up** in order to get the final gradient (the loss is a sum over the training examples)

Copyright Stefano Melacci (University of Siena)

Learning from Data

Stefano Melacci

Back to: Learning from Data

- 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)

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

- BackProp (BP)-based optimization, **batch mode**:
  1. *Provide all the training data to the net*
  2. *Compute gradients (**BP**) – sum them up*
  3. *Update weights and biases*

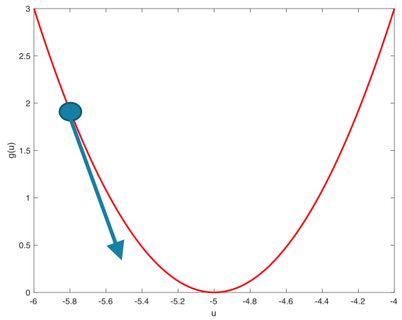
  Training **EPOCH**
  4. **Check convergence** (gradient close to zero, objective function close to zero, …)
  5. If not converged, go to 1

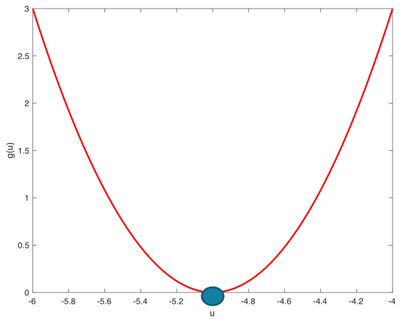
Copyright Stefano Melacci (University of Siena)

Back to: Gradient-based Optimization

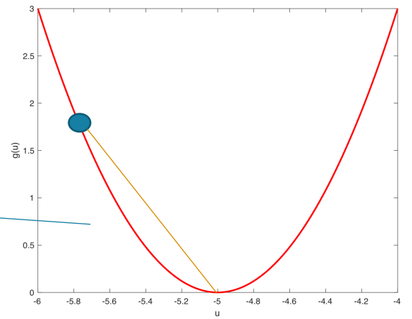
- A well known issue with gradient descent is that the choice of the step-size is crucial to ensure a fast convergence toward a minimizer of the objective function
  - Let's consider two cases: the ideal one (*top row*), the common zig-zag effect (*bottom row*)

(a)

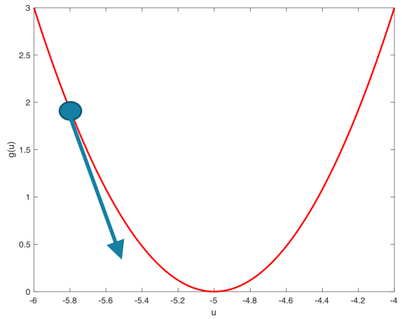


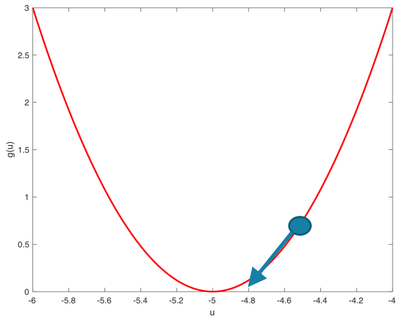


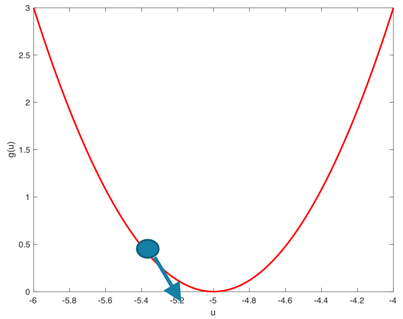
Ideal step size (not easily determinable in non-linear optimization)

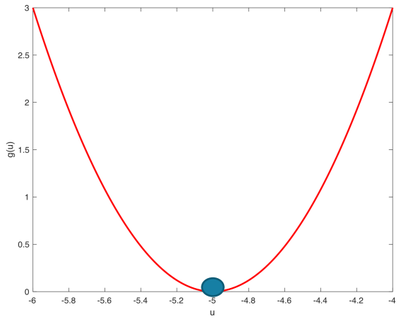


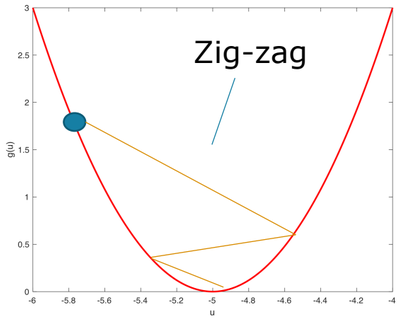
(b)









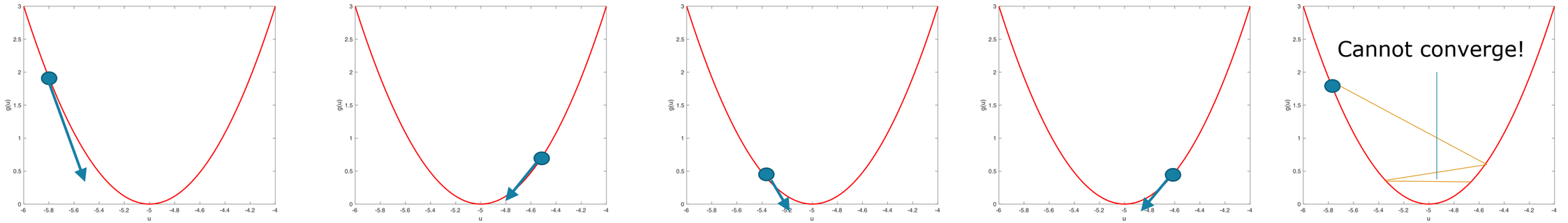


Copyright Stefano Melacci (University of Siena)

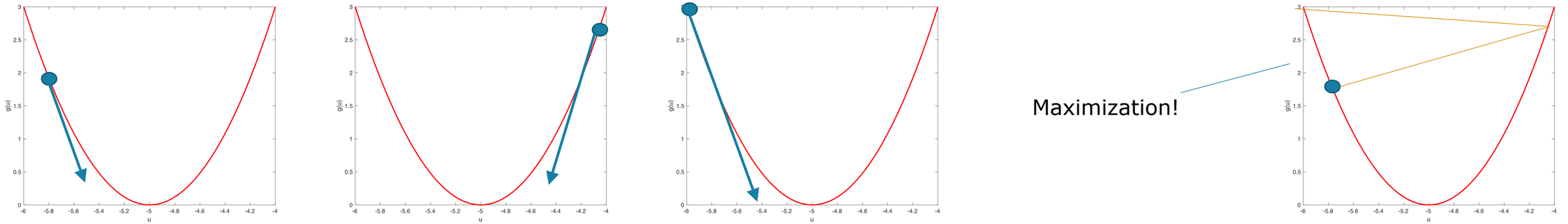
Back to: Gradient-based Optimization (2)

- Fixed step size? What is the good one?
  - Too small: very slow convergence
  - Not appropriate: issues in convergence (oscillations around the minimizer – *top row*)
  - Too large: it could end up in maximizing the function! (numerical issues – *bottom row*)

(a)



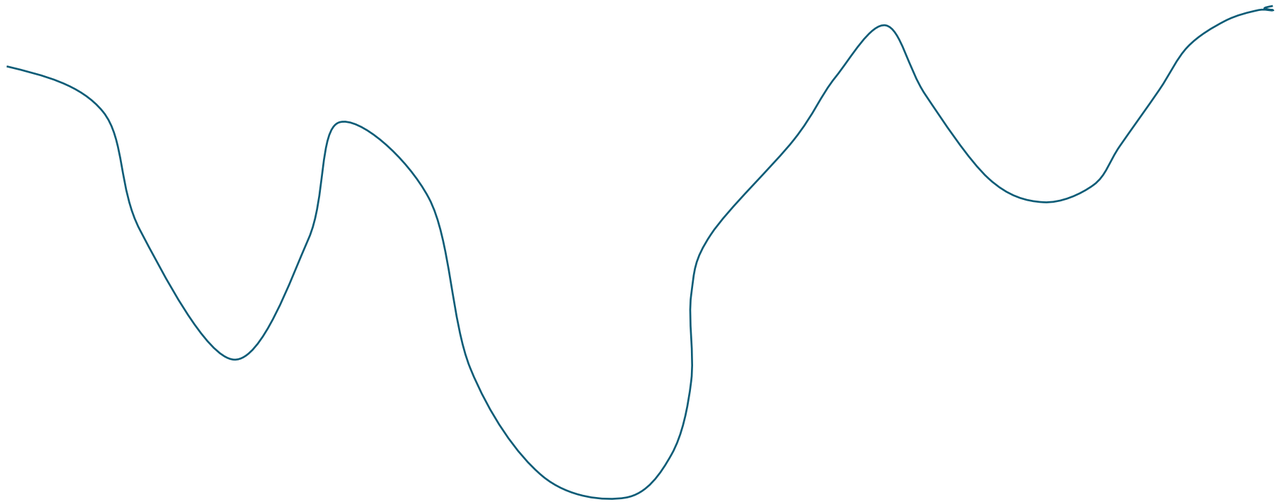
(b)



Copyright Stefano Melacci (University of Siena)

Gradient-based Opt & Neural Networks

- In the case of Neural Networks, the situation is way more complicated than what has been sketched in the pictures of the previous slide
- The learning objective is *not convex* with respect to the network parameters (weights and biases)
- Selecting a good step size is **problem dependent** and not straightforward



Copyright Stefano Melacci (University of Siena)

Adaptive Solutions

- The step size (also called *learning rate)* can be automatically adapted to improve the converge speed
- More generally, the update term in the following equation could be the result of an adaptive process

$$\mathbf{w} = \mathbf{w} - \rho \nabla \mathtt{obj}$$
$$= \mathbf{w} + \Delta \mathbf{w}$$

Update term

- Several techniques
  1. Basic Adaptive Learning Rate
  2. Resilient Backpropagation (RPROP)
  3. Momentum Term
  4. Adaptive Moment Estimation (Adam)
  5. …

Copyright Stefano Melacci (University of Siena)

Adaptive Solutions (2)

- **Basic Adaptive Learning Rate**
  - Initialize the step size
  1. Update the weights

$$\mathbf{w} = \mathbf{w} - \rho \nabla \mathtt{obj}$$

  2. If, after having update the weights, the objective function decreases, then increase the learning rate (otherwise, reduce the learning rate)
  3. The learning rate reduction is usually stronger than the increment
  4. Keep the learning rate whitin a selected interval, go to 1

$$\rho \in [0.0001, 100]$$

$$\rho_0 = 0.001$$

$$\mathtt{inc} = 2.0$$

$$\mathtt{dec} = 0.1$$

$$\rho_{t+1} = \mathtt{inc} \cdot \rho_t \quad \text{if obj went down}$$

$$\rho_{t+1} = \mathtt{dec} \cdot \rho_t \quad \text{if obj went up}$$

$$\text{ensure that} \quad \rho_{t+1} \in [0.0001, 100]$$

Copyright Stefano Melacci (University of Siena)

Adaptive Solutions (3)

- **RPROP**
  - The update term, for each weight, is automatically determined in function of the *sign of the gradient*, while the real value of the gradient *is not used at all*

Update term

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

  - In detail:
    - Initialize the weight update terms to given values, compute the gradient, perform a first update
    1. Compute the gradient
    2. If the gradient with respect to a certain weight kept the **same sign**, increase the update term
    3. If the gradient with respect to a certain weight **switched sign**, then decrease the update term (we crossed the minimum in the dimension associated to the considered weight)
    4. Force the update term to have the sign of the negative gradient
    5. Update weights and go to 1

  - The way the update term is incremented/reduced is analogous to what we described in the previous slide for the step size

Copyright Stefano Melacci (University of Siena)

Adaptive Solutions (4)

- **Momentum Method**
  - The update term is a linear combination of the gradient and the previous update term

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

Update term

$$\Delta\mathbf{w} = -\rho\nabla\mathtt{obj} + \beta\Delta\mathbf{w}$$

New element introduced by the momentum method

  - It basically smooths the changes of the update term between consecutive iterations, making the learning process more stable, preventing oscillations
  - It depends on a new parameter (β)

Copyright Stefano Melacci (University of Siena)

Adaptive Solutions (5)

- **Adam**
  - The update term, for each weight, is progressively updated in function of the gradient and of the norm of the gradient

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

Update term

$$\Delta w_{ij} = -\rho \frac{m_{ij}}{\sqrt{v_{ij}} + \epsilon}$$

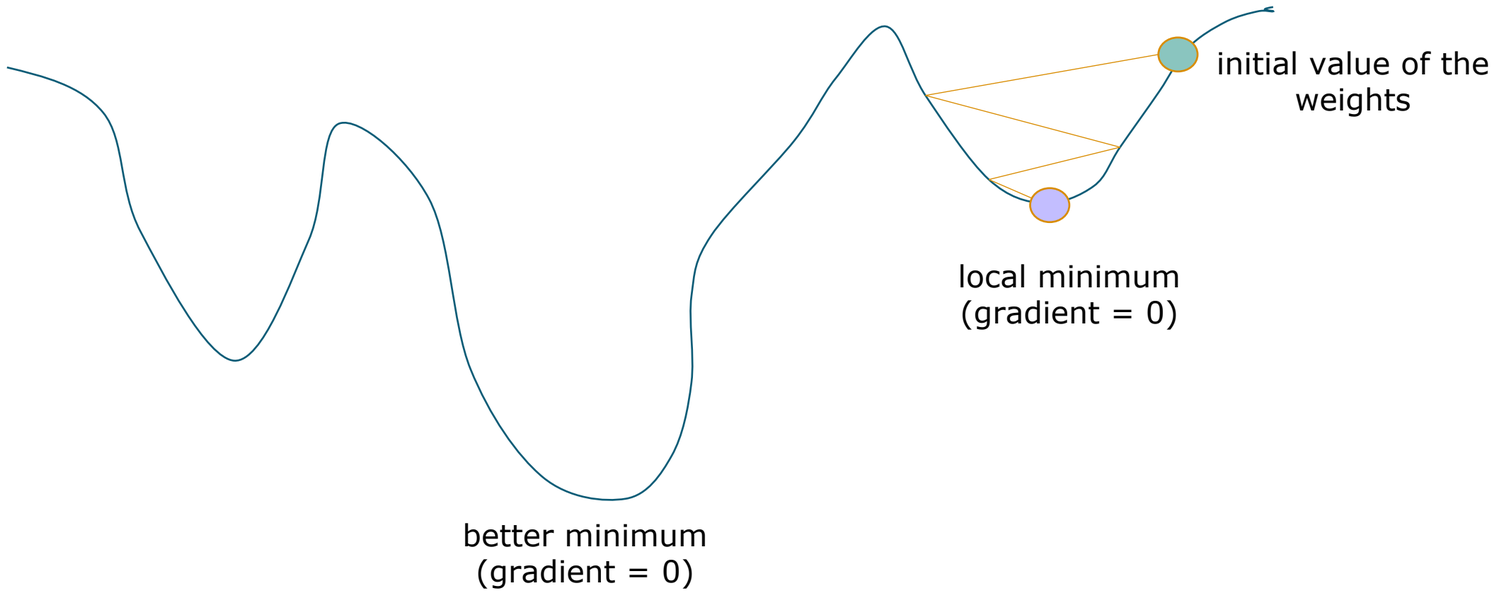
Avoids divisions by zero

  - m_{ij} depends on the gradient
  - v_{ij} depends on the squared norm of the gradient
  - Both the terms are updated by averaging them with their previous values
    - From a certain perspective, this is related to what happens in the Momentum Method
  - The averaging operations depends on two new parameters (given) β_1 and β_2

Copyright Stefano Melacci (University of Siena)

Local Minima

- As we already discussed, the learning objective is *not convex* with respect to the network parameters (weights and biases)
  - **Local minima**
  - More frequently: flat regions, **saddle points**



Some "tricks" to help in preventing critical issues:

- Multiple (random) initializations **(restarts)**
- **Adaptive** learning rate
- ***Stochastic*** *optimization*

Copyright Stefano Melacci (University of Siena)

Stochastic Gradient Descent

- In most of nowadays neural network optimization tools, it is extremely frequent to follow a stochastic strategy in the gradient descent procedure
  - **Stochastic Gradient**

- It consists in computing the gradient for a **single training example**, and update the network weights and biases using such gradient
  - If the learning rate is too large, the contribute of each example in updating the values of the weights could be too strong, leading to instabilities
  - If the learning rate is small, then we get a relatively stable behavior

- *This approach helps the learning algorithm to escape from local minima and find (possibly) better solutions*
  - Differently, in batch-mode learning:
    - The gradient is the sum of the gradients of all the training examples
    - "Averaging" effect in the gradient computation

Copyright Stefano Melacci (University of Siena)

Stochastic Gradient Descent and Mini-Batches

- 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)

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

Mini-batch sizes usually go from 10 to 1000 examples (roughly)

- BP-based optimization, **stochastic (mini-batch) mode**:
  1. ***Randomize*** *training data*
  2. *Split it into small, disjoint, subsets (mini-batch)*
     - *i. Provide a new mini-batch to the net*
     - *ii. Compute gradients (**BP**) – sum them up*
     - *iii. Update weights and biases*
     - *iv. Did we consider all the mini-batches? If not go to i.*
  3. **Check convergence**
  4. If not converged, go to 1

Training **EPOCH** (several updates!)

Copyright Stefano Melacci (University of Siena)

Learning is not a Bare Optimization Procedure!

- Our goal **is not only** to push the objective toward the smallest possible value!
- We want to evaluate how the model **generalizes**, i.e., how good it is at making prediction on out-of-sample data
  - **Early stopping** by means of validation data *V* (*not used to compute gradients*)
    - We stop the training procedure when the *accuracy* on validation data does not improve anymore
    - Equivalently, we stop when the *error rate* on validation data does not reduces anymore

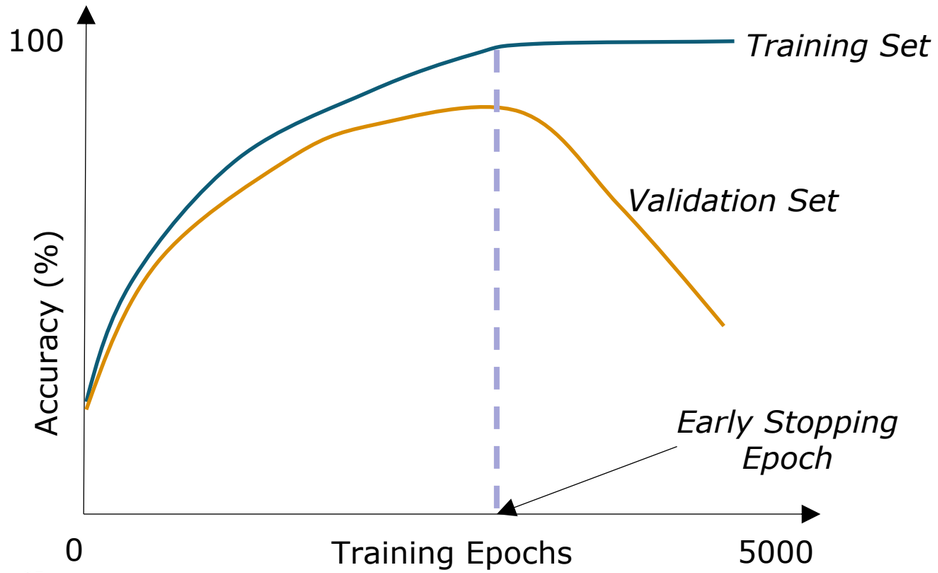
Training set (what we use in BP)

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

Validation set

$$V = \{(\mathbf{x}_i, \mathbf{y}_i),\ i = 1, \ldots, m\}$$

$$T \cap V = \emptyset$$



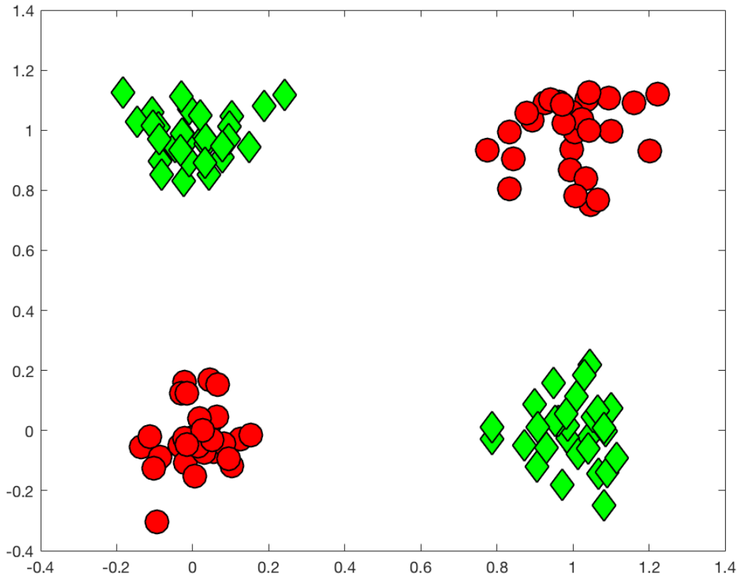
Copyright Stefano Melacci (University of Siena)

Deep Architectures

Stefano Melacci

Learning and Hidden Layers

- A classical problem: **XOR-like data**
  - Green: data belonging to class 1
  - Red: data belonging to class 2
  - Goal: train a Neural Network to (correctly) classify the data samples
    - Output layer activation: linear

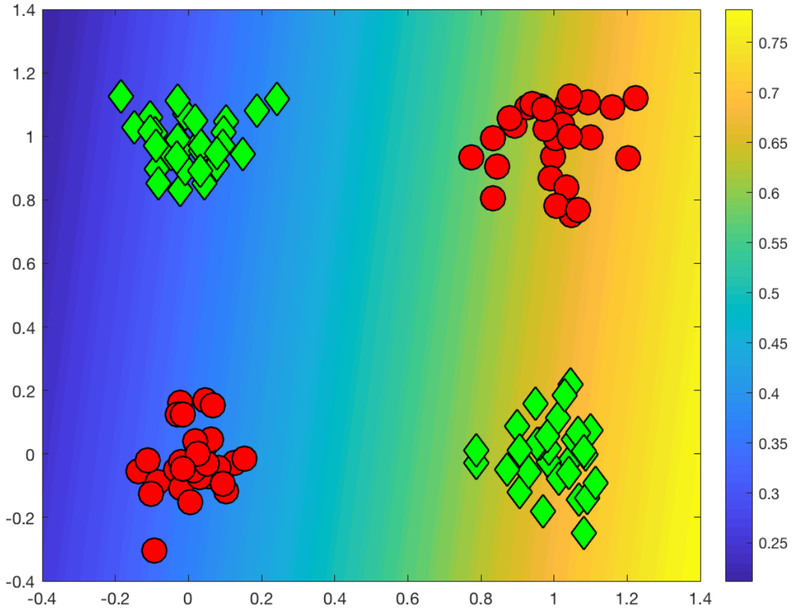


Classes are NOT linearly separable!

Copyright Stefano Melacci (University of Siena)

Learning and Hidden Layers (2)

- Neural Network without any hidden layers (input → output)
  - This network is **not able** to correctly classify the data samples!
  - In the following picture, **yellow**ish regions are predicted as belonging to *class 1*, while **blue**ish areas are about *class 2*

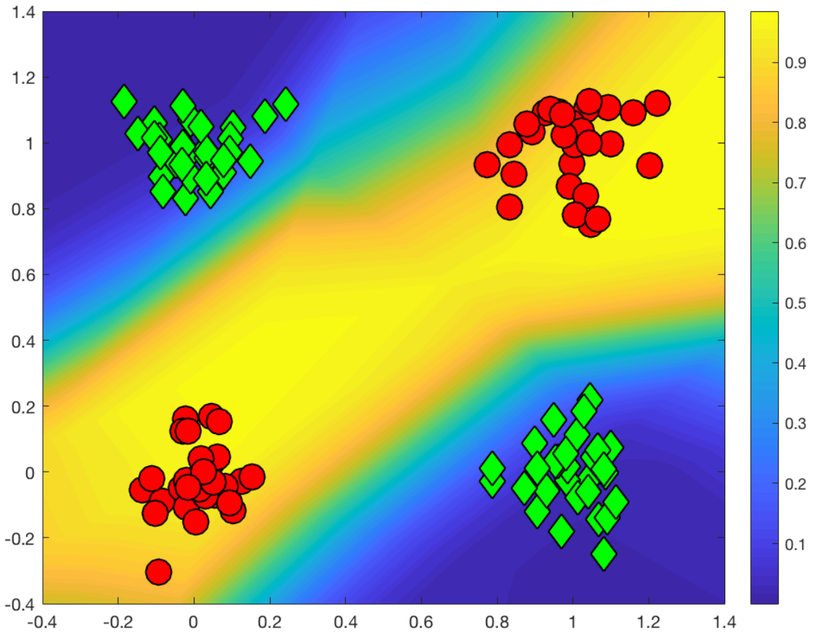


A neural network without hidden layers is basically a linear classifier!

Copyright Stefano Melacci (University of Siena)

Learning and Hidden Layers (3)

- Neural Network with a single hidden layer (input → hidden → output)
  - This network is **able** to correctly classify the data samples!
  - The hidden layer **computes a representation of the data that is linearly separable**



A neural network with hidden layers is a non-linear classifier!

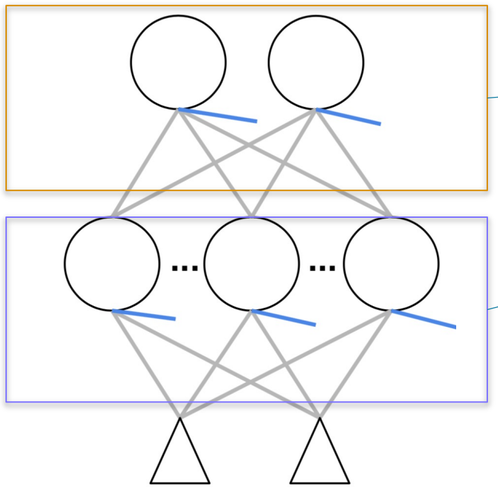
Copyright Stefano Melacci (University of Siena)

Shallow Architectures

- "Shallow" vs. "Deep" architectures
  - Sometimes a questionable distinction
- **Shallow**
  - Project the data into a "more appropriate space"
    - **Up-to one (non-linear) projection**
    - Better representation of the data
  - Classify the data using a linear classifier
    - Eventually adding a non-linear activation on top of it

Simple classifier

Non-linear projection

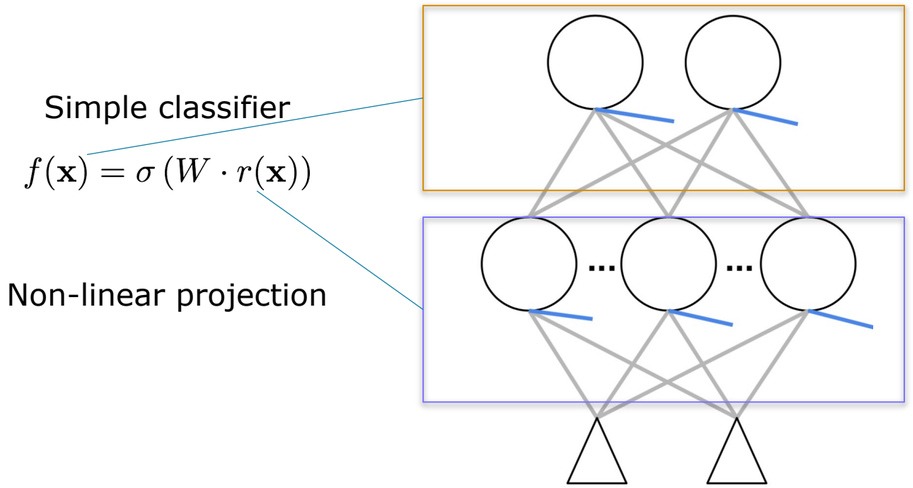


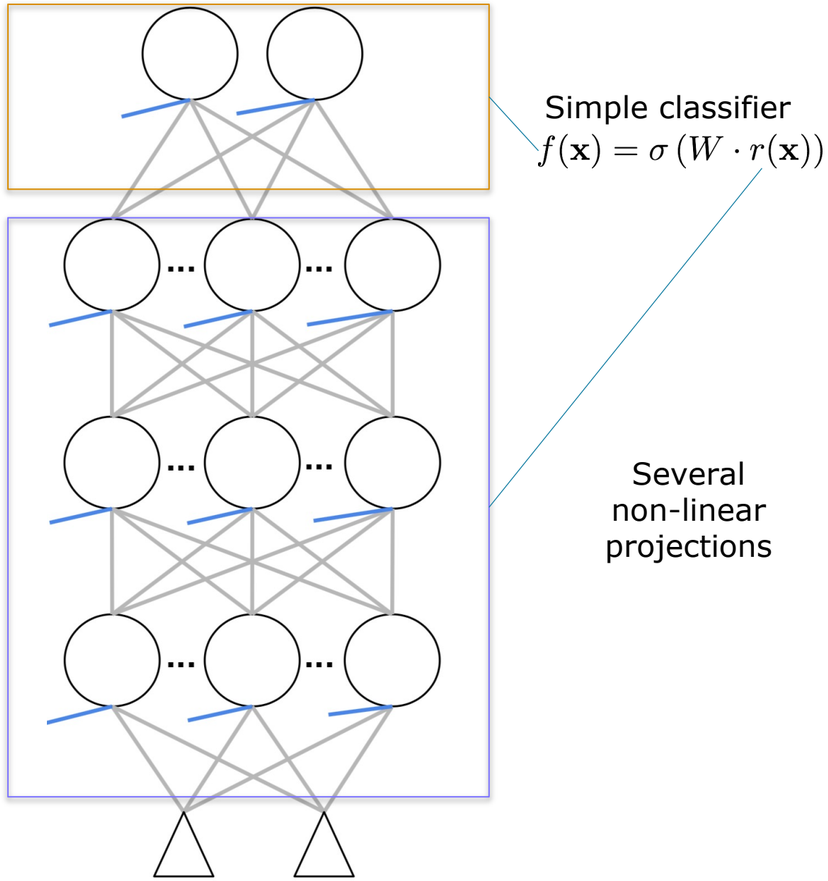
$$f(\mathbf{x}) = \sigma\left(W \cdot r(\mathbf{x})\right)$$

Copyright Stefano Melacci (University of Siena)

Deep Architectures

- "Shallow" vs. "Deep" architectures
  - Sometimes a questionable distinction
- **Deep**
  - Project the data into a "more appropriate space"
    - **More-than one (non-linear) projections**
    - Even better representation of the data
  - Classify the data using a linear classifier
    - Eventually adding a non-linear activation on top of it

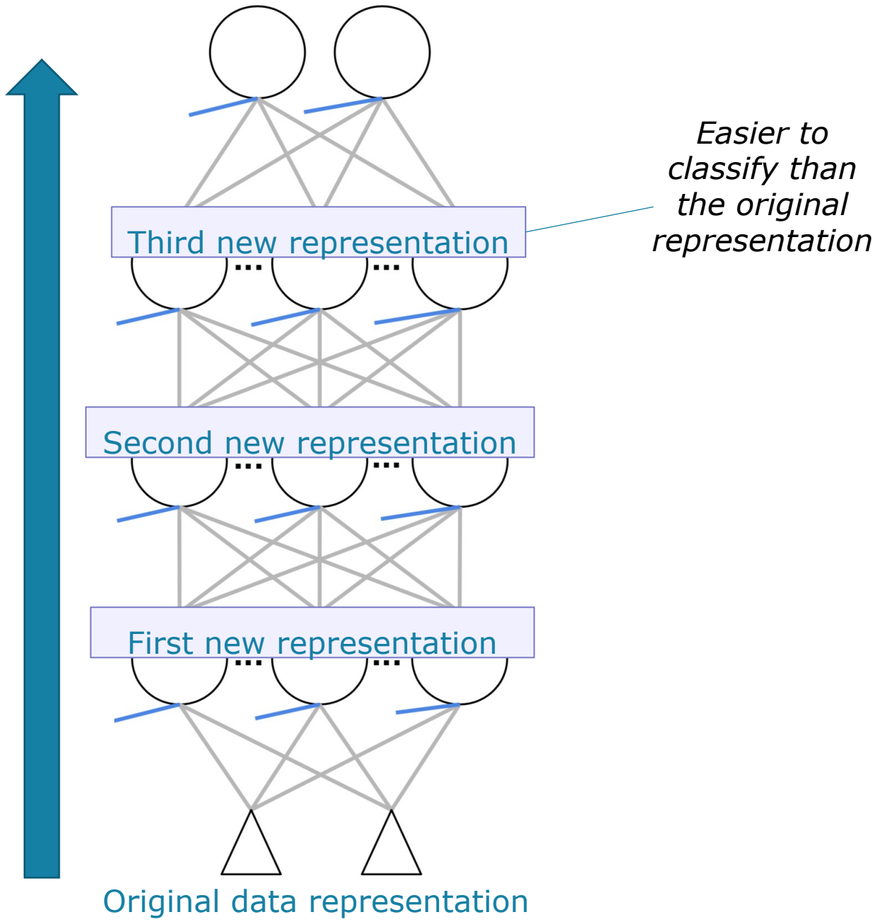




Copyright Stefano Melacci (University of Siena)

Why Deep Architectures?

- Shallow architectures do not emphasize any compositional properties
- Deep architectures **do emphasize compositionality**
  - Each layer input is the outcome of composing the projections computed by the layers below
  - If we imagine that each hidden neuron is a feature extractor, a feature of layer *h* contributes to the computation of all the features of layer *h+1*
  - The same holds from features of layer *h+1* to the ones of *h+2*, and so on
- **Hierarchical representation of the data**
  - Higher layers ➔ **higher level of abstraction** in the generated data representation



Copyright Stefano Melacci (University of Siena)

Why Deep Architectures? (2)

- **Universal Approximation Theorem**
  - *Neural networks with a **single hidden layer** with a **finite number of neurons** can approximate any continuous function (under some assumptions)*

    Cybenko, "Approximations by superpositions of sigmoidal functions", *Mathematics of Control, Signals, and Systems*, 2(4), 303–314, **1989**
    Hornik, "Approximation Capabilities of Multilayer Feedforward Networks", *Neural Networks*, 4(2), 251–257, **1991**

- However, we do not know how many neurons we need to build the right representation
  - It could be huge!
- Deep Networks: compositionality gives a large gain in **representational power**!
  - We can more efficiently use the available resources to get better representations than single layer networks

Copyright Stefano Melacci (University of Siena)

On the Popularity of Deep Networks

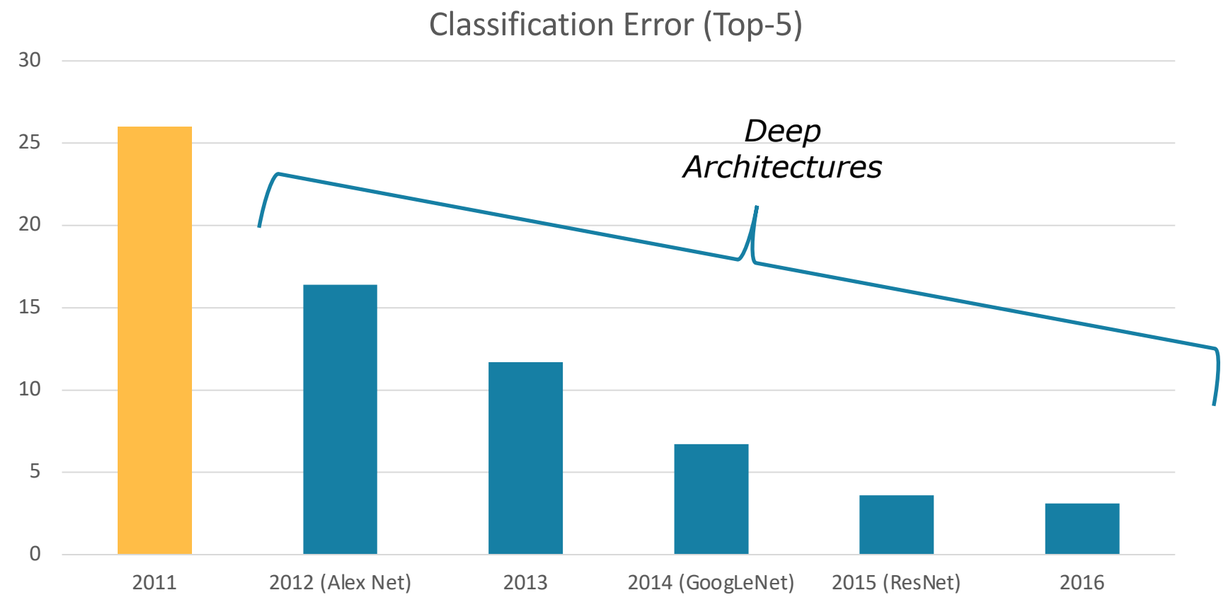
- In the late 1990/early 2000, shallow architectures were the most popular ones
  - Mostly kernel machines, that, from certain point of views, can be considered shallow models
  - Neural networks with a single hidden layer
- Moreover, neural networks were considered hard to optimize (local minima), less performant than kernel machines
  - Neural network popularity went up and down in their long history

- Since (roughly) **2010**, deep architectures were applied to **several different tasks**, leading to *significant improvements* in the overall performance
  - Computer vision
  - Speech recognition
  - Machine translation
  - …

Copyright Stefano Melacci (University of Siena)

On the Popularity of Deep Networks (2)

- The case of Computer Vision (Image Classification)
  - **ImageNet**
    - Classify an input image among 1000 classes
    - Millions of examples



Copyright Stefano Melacci (University of Siena)

On the Popularity of Deep Networks (3)

- Some of the key ingredients that played a crucial role in making learning on deep architectures (**Deep Learning**) popular are
  1. Emphasizing the importance of compositionality and distributed representations
  2. Large computing power (Graphics Processing Units – GPUs)
  3. Large-scale datasets (mostly fully supervised!)
  4. Tricks and improvements in the neural network optimization process
     - Tricks to avoid issues in gradient computations (we will discuss them in the next slides)

- Key people:
  1. Yoshua Bengio (Montreal)
  2. Geoffrey Hinton (Toronto)
  3. Yann LeCun (New York)
     - **Turing Award 2018**

Copyright Stefano Melacci (University of Siena)

Huge Datasets with Fully Supervised Data

- **Dream (example of the ideal case)**
  - ImageNet: an image database with millions of pictures annotated with their category

- **Reality (most of the small/medium companies)**
  - "We have no supervised data, we are going to create them, we can prepare a small number of annotations"

Copyright Stefano Melacci (University of Siena)

Computational Power

- **Dream (example of the ideal case)**
  - Cluster of GPUs, advanced contracts with power suppliers

- **Reality (small companies)**
  - "We have no GPUs, you can setup the model and make it run in this machine"

Alternatives:
Cloud-computing, Deep Learning services hosted on the servers of some major companies,
…

What about privacy?

Copyright Stefano Melacci (University of Siena)

Transfer Learning

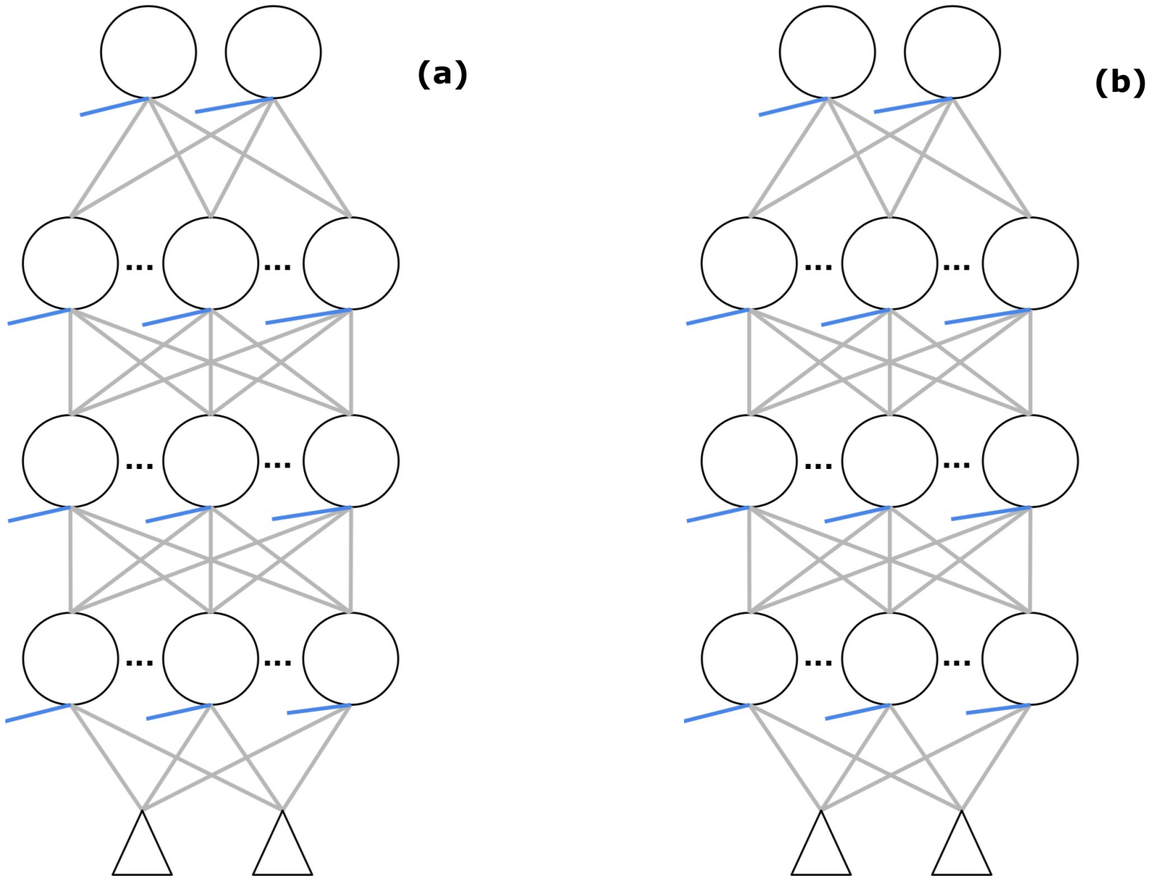
- Some learning tasks are related, or they share several aspects
  - Classification of images of cats and **dogs** and classification of images of **dogs** and birds
  - Classification of *handwritten digits* and classification of *handwritten letters*
  - Classification of *voices* and classification of *music*
  - Classification of *documents (topics)* and classification of *conversation logs (topics)*
  - ...
- *A model that leads good results in a certain task could be used to transfer knowledge to a related task*

- In the context of Deep Neural Networks, the **hidden representations** that have been learned in a certain task could be a useful starting point to initialize a network that we are going to train in a reated task
  - The hidden representation might have captured some properties that are common in the two tasks

Copyright Stefano Melacci (University of Siena)

Transfer Learning (2)

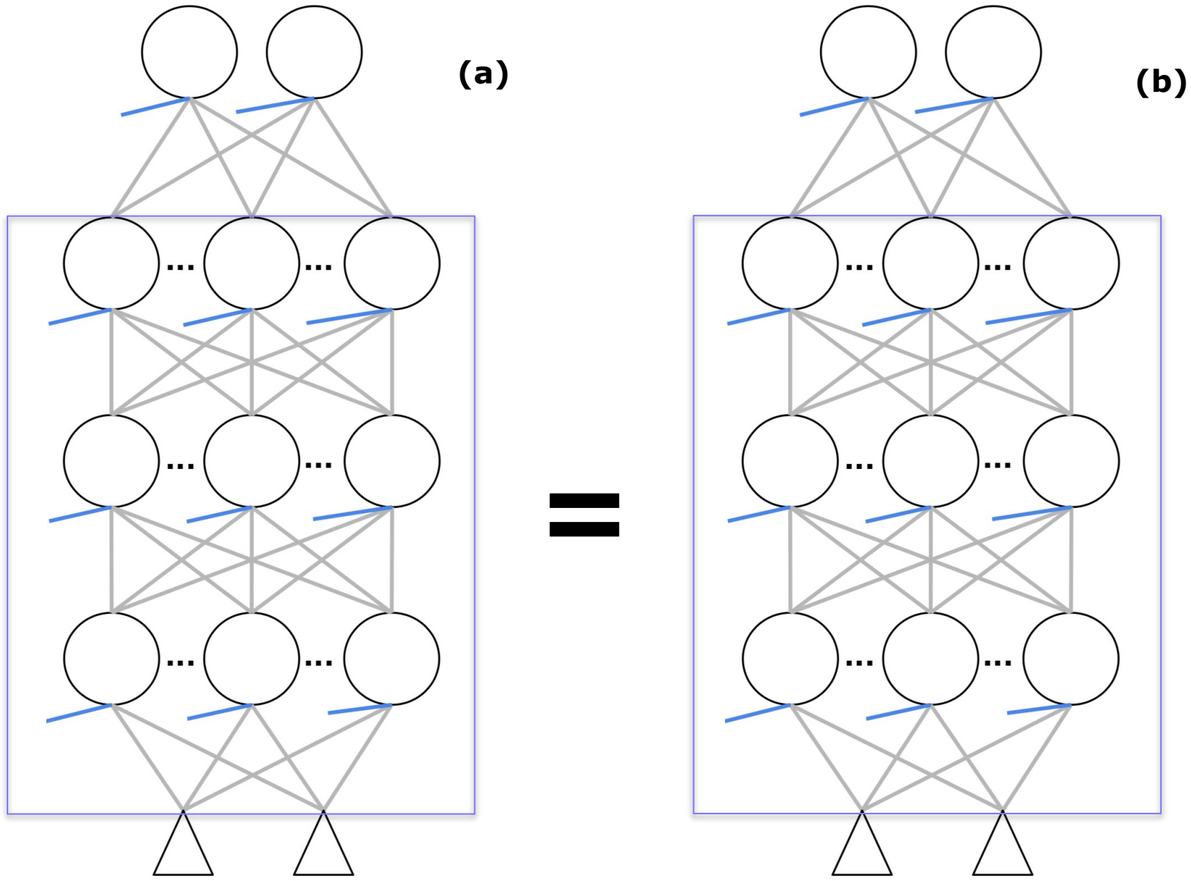
- Suppose that (a) is a network trained on millions of examples and that is providing great results in generic image classification
- Suppose that (b) is a network with the same architecture, that we are going to train in another image classification task involving classes that were not considered in (a)
- Suppose that we have a relatively small number of examples to train (b)



Copyright Stefano Melacci (University of Siena)

Transfer Learning (3)

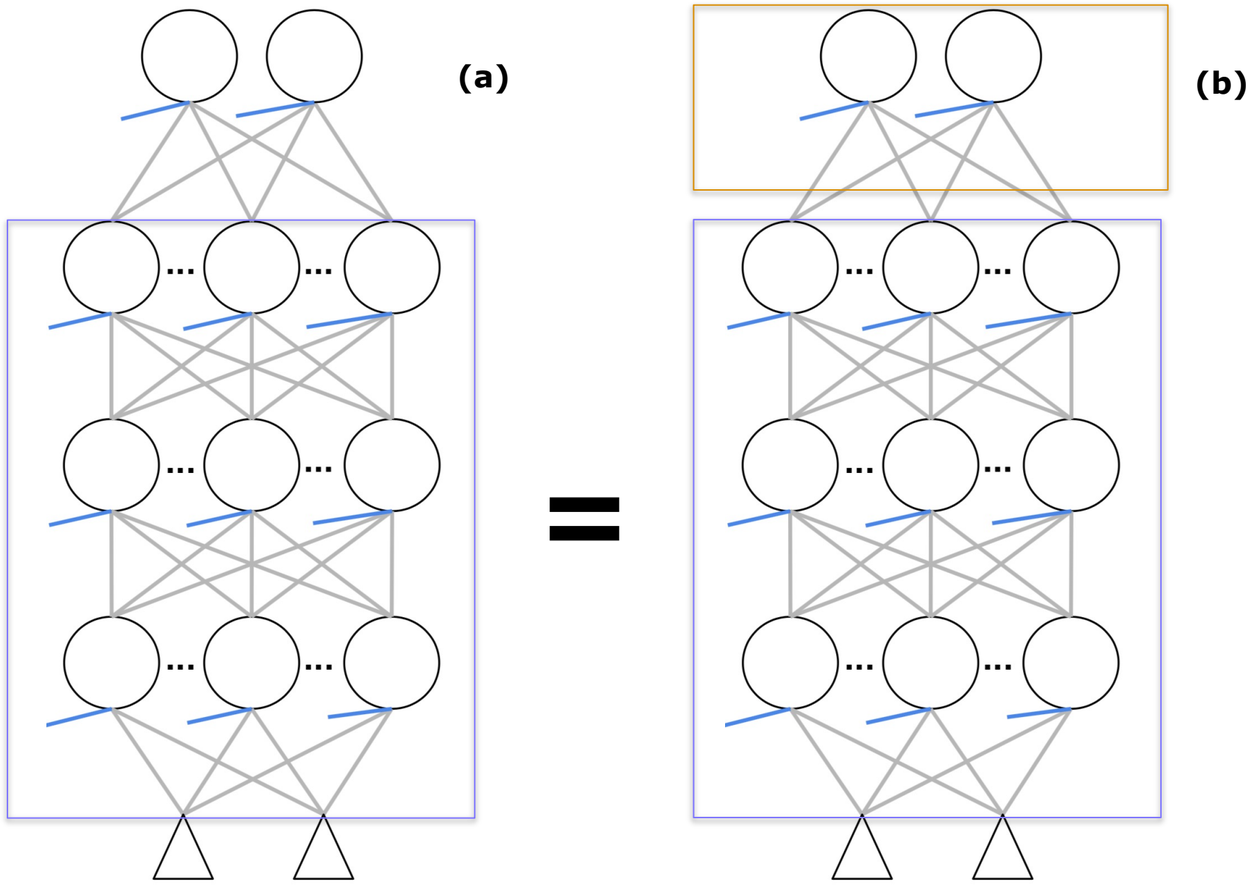
- We can initialize (b) with the values of the weights belonging to the portion of (a) that is highlighted in the picture



Copyright Stefano Melacci (University of Siena)

Transfer Learning (4)

- Then, we can simply train the last layer of (b) with the few available training examples…
  - We can also consider architecture with multiple layers in the orange box
- … or we can train the whole network (b), allowing the transferred portion of network to be fine-tuned in the new task
- *This approach is extremely common in nowadays Computer Vision research*



Copyright Stefano Melacci (University of Siena)