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They are not intended to be of public domain in any way, and they must not be published or shared out of the context in which they are presented by Stefano Melacci.

Learning in Deep Architectures

Stefano Melacci

Gradient-related Issues

- The gradient with respect to the weights of the network is directly proportional to the delta terms, that are updated with the following rule (**BackProp**)

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

- Let us consider a sigmoidal activation function, for which $\frac{\partial o_j^{\ell-1}}{\partial a_j^{\ell-1}} \in (0, 0.25]$
  - Weight are generally randomly initialized, usually getting values smaller than 1 (absolute value). As a result, the value of the gradient will be progressively downscaled (**vanishing gradients**)
  - On the other hand, gradient could also **explode** when weights are very large
    - (or when using activation functions with large derivatives)

Gradient-related Issues (2)

- Some work-arounds
  - **Rectifiers** (*ReLu* activations)
    - Constant derivative in each linear region (equal to 1, or to 0)
    - It reduces the vanishing gradient effect (when we consider the linear region with derivative 1)

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

  - **Gradient Clipping**
    - When the norm of the gradient is too large (above a certain threshold) it is clipped to a given value
    - It helps with exploding gradients

$$\|\nabla \texttt{obj}\| \leq \tau$$

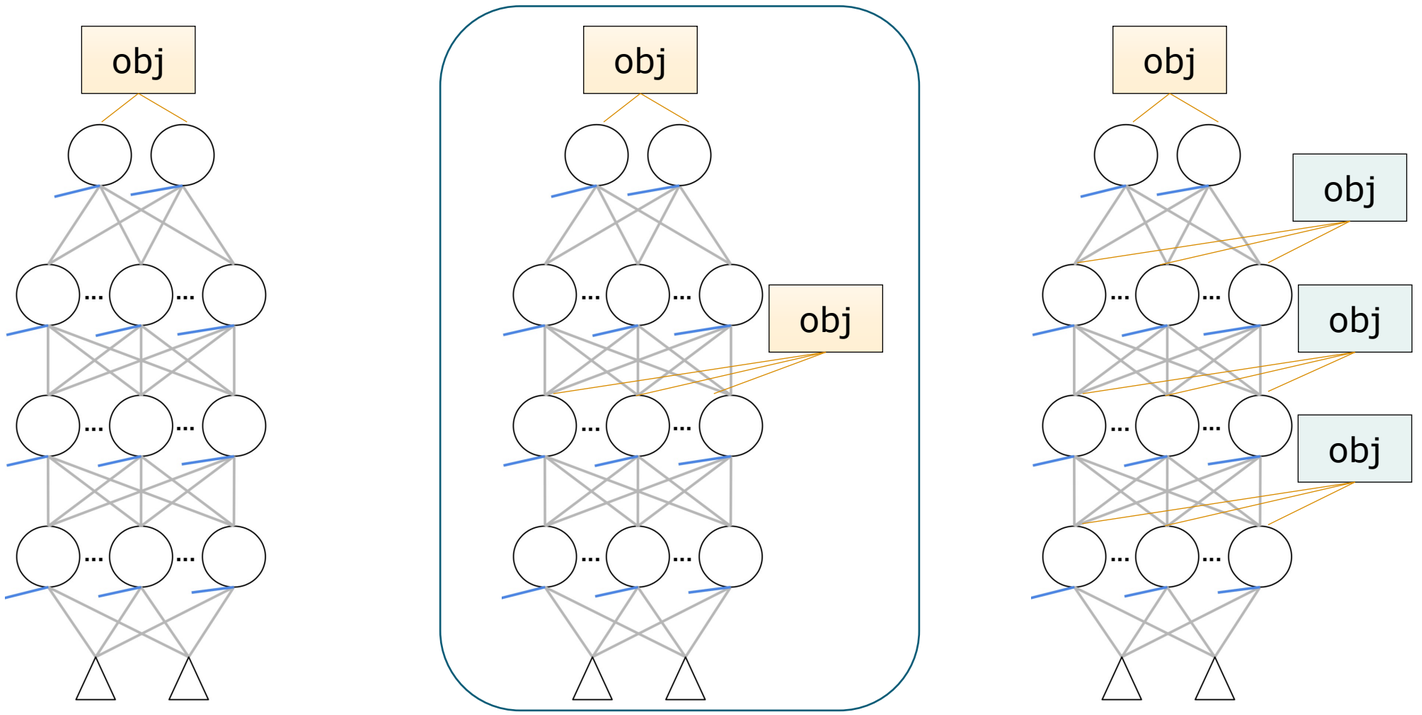
    - Simplified clipping: coordinate-wise clipping (warning, the direction of the gradient is lost!)

$$\frac{\partial \texttt{obj}}{\partial w_{ij}^{\ell}} = \max\left(\min\left(\frac{\partial \texttt{obj}}{\partial w_{ij}^{\ell}}, \tau\right), -\tau\right)$$

  - Accurate choice of the *loss function*, of the weight *initialization* routine, …

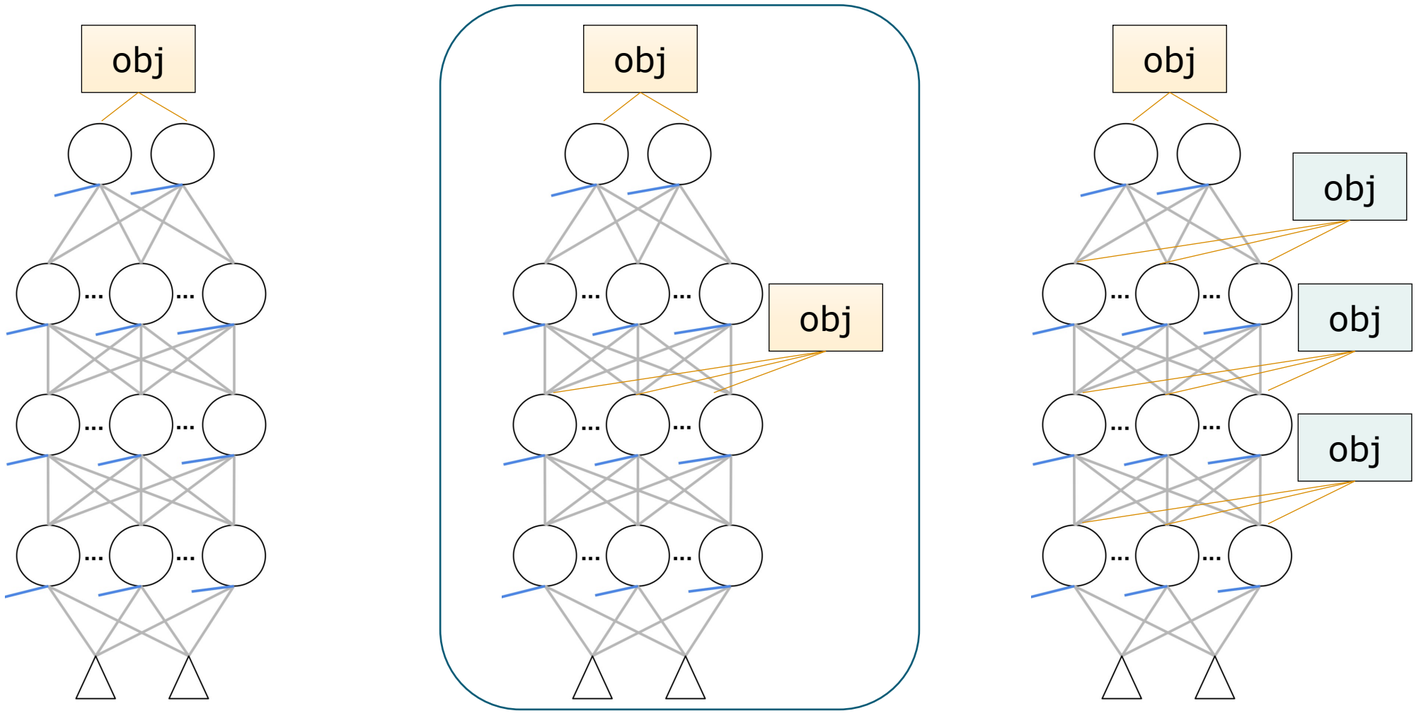
Gradient-related Issues (3)

- Some work-arounds
  - **Introduce objective functions that are closer to the lower layers**
    - It reduces the vanishing gradient effect
    - *Which objective functions?*



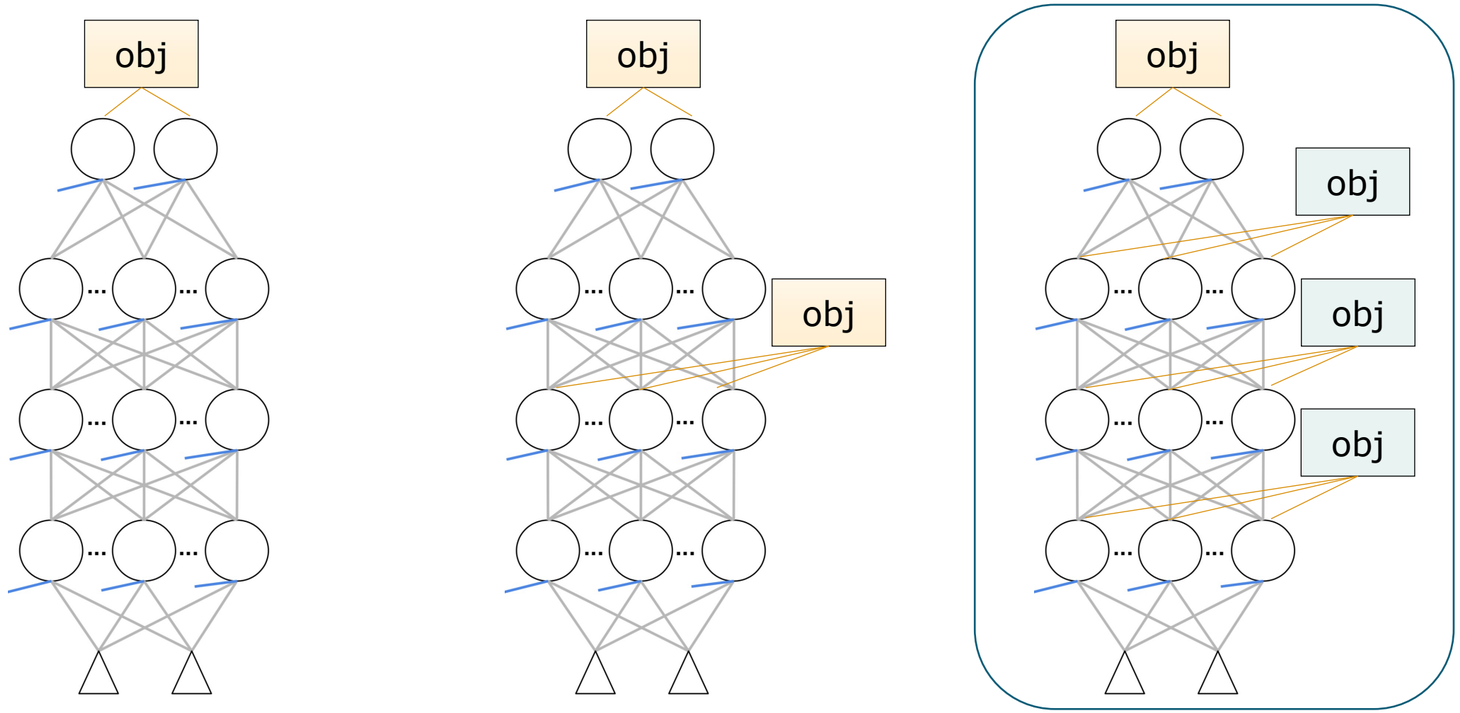
Gradient-related Issues (4)

- Some work-arounds
  - **Introduce objective functions that are closer to the lower layers**
    - Sometimes, enforcing a *downscaled* instance of the same classification loss the we use on top the network can *sometimes* help in speeding-up the development of discriminative representations



Gradient-related Issues (5)

- Some work-arounds
  - **Introduce objective functions that are closer to the lower layers**
    - Another popular solution, is to use *unsupervised learning* to train the network in a **layer-wise** fashion



Layer-wise Training

- **A more generic definition of training set**
  - 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)
  - The training set *might* also include *other* examples without any targets (blank)

$$T = \{(\mathbf{x}_i, \mathbf{y}_i)\,,\ i = 1, \ldots, n\} \cup \{(\mathbf{x}_j, \texttt{blank})\,,\ j = 1, \ldots, m\}$$

  - There is some precious information in the way the data is *distributed* (ignoring the targets), that can be used to pre-train the network in a layer-wise fashion
  - Some regularities in the data can be discovered in an unsupervised way

- **Layer-wise training**
  - Train the first layer using an *unsupervised* objective, then *freeze* it
  - Train the next layer using, an *unsupervised* objective, then *freeze* it, and so on…
  - Finally, **fine-tune** the whole network using the classic *supervised* loss
    - Exploiting the supervised portion of the training data

Layer-wise Training: Stacked Autoencoders

- **Autoencoder**: neural architecture that learns to reconstruct the input signal (under some conditions)
  - As a basic example, consider a neural network with
    - Single hidden layer
    - Number of output units equal to the number of dimensions of the input
    - Linear output activation

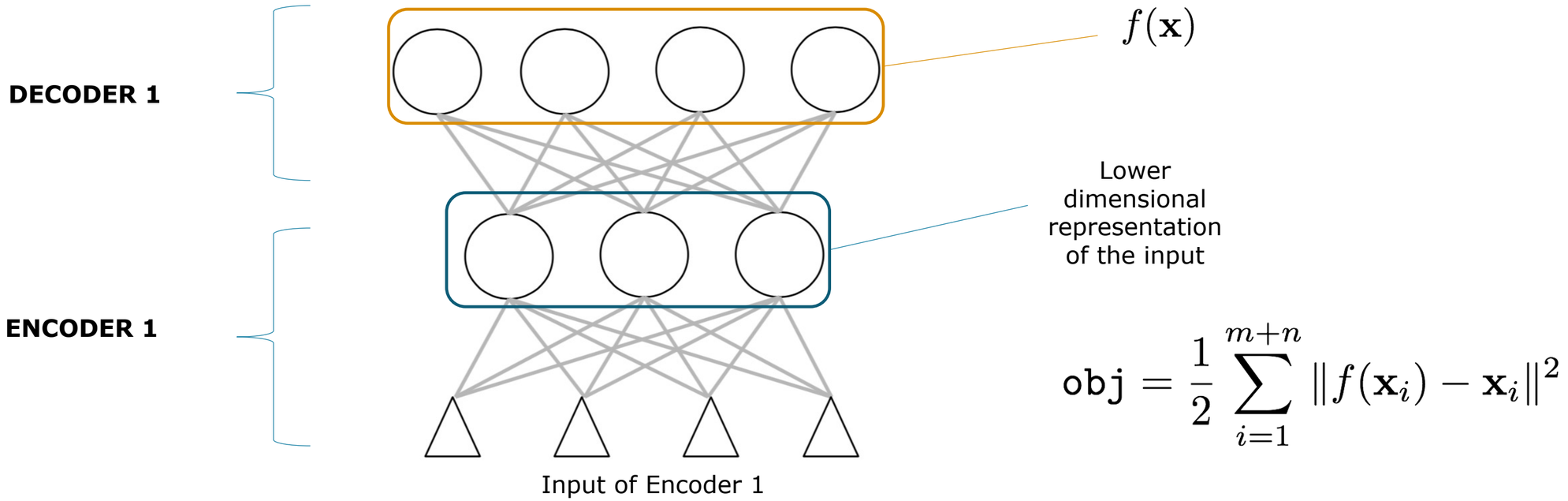
  - **Reconstruction loss**

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

(the sum is intended to go over all the available training points (being them supervised or not))

Layer-wise Training: Stacked Autoencoders (2)

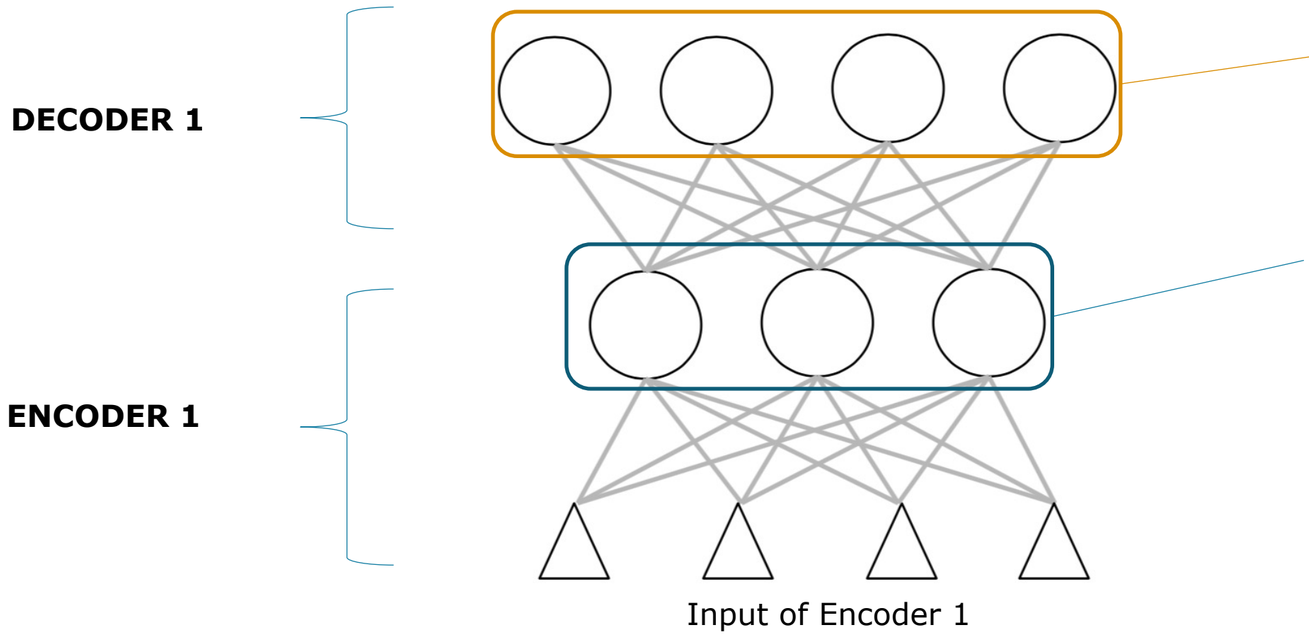
- Consider the case of autoencoders that projects the input onto a lower-dimensional space, before re-constructing it
  - There exists different types of autoencoders, with different architectural constraints



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

Layer-wise Training: Stacked Autoencoders (3)

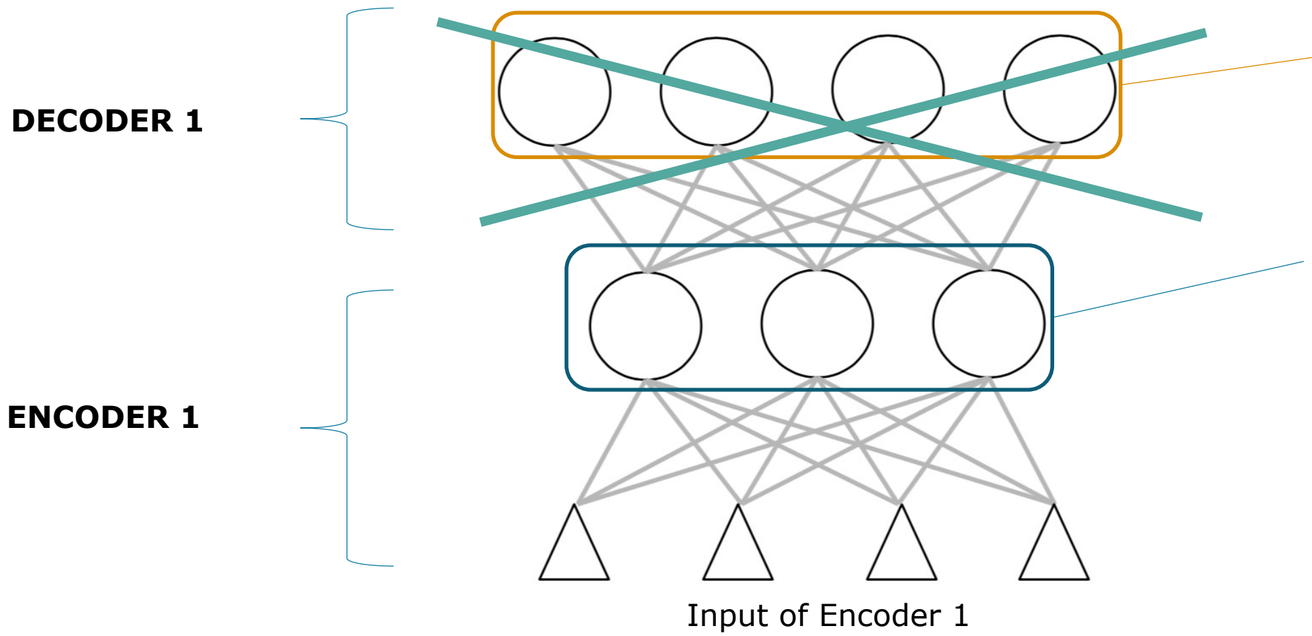
- The autoencoder is enforced to represent the data into a lower dimensional representation, thus discarding not-useful information
  - Similar examples are likely to be projected onto similar lower-dimensional representations



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

Layer-wise Training: Stacked Autoencoders (4)

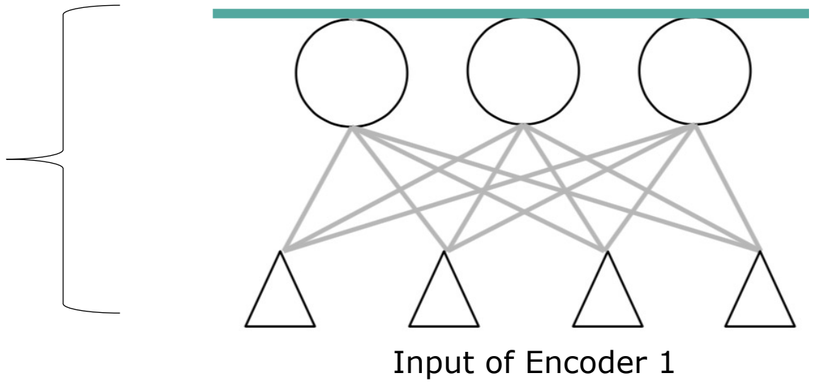
- Once the autoencoder has been trained, we can *remove the decoder*, and learn a new autoencoder, progressively building (layer-wise) a deep architecture



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

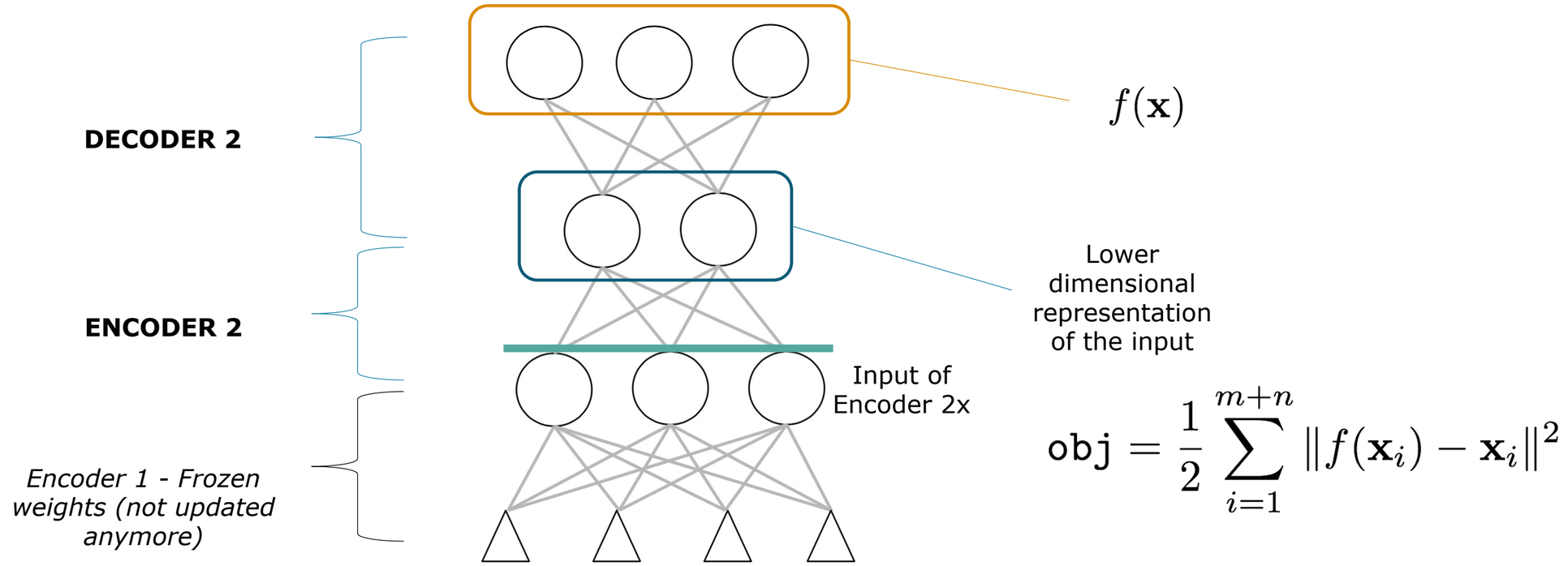
Layer-wise Training: Stacked Autoencoders (4)

- The weights of the encoder are kept **frozen** (that means the will not get updated by the following operations)



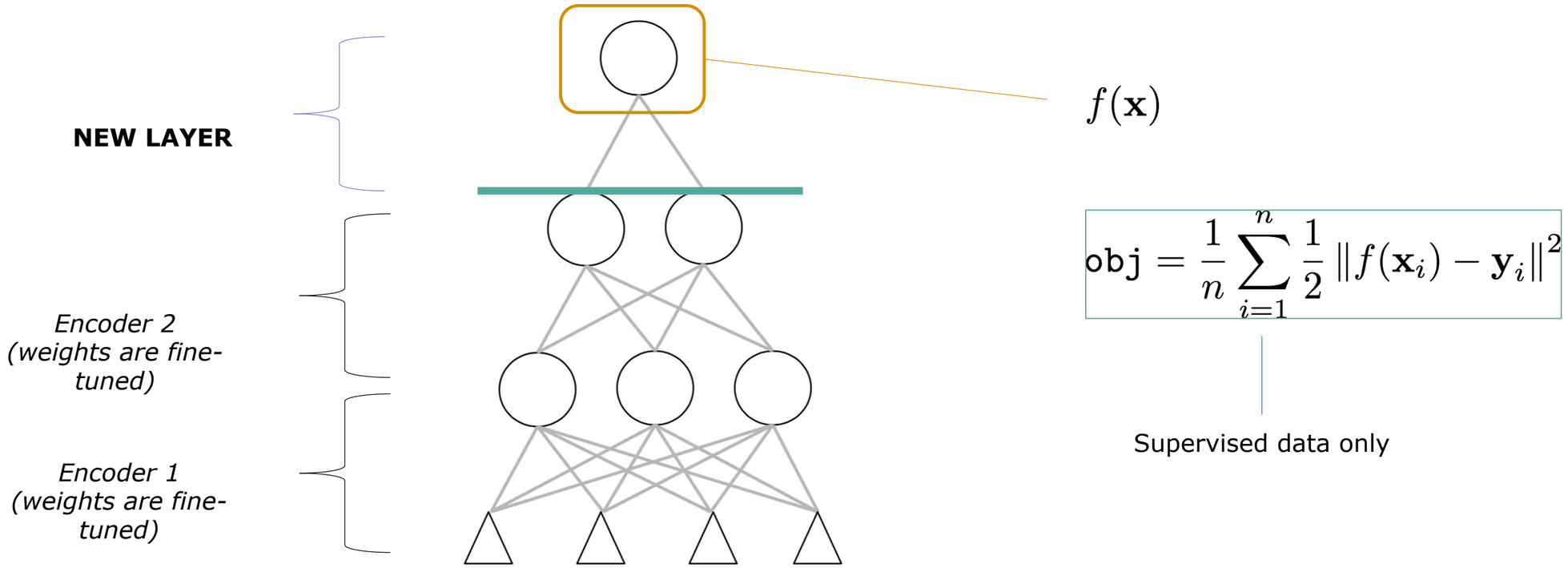
Layer-wise Training: Stacked Autoencoders (5)

- We can repeat the process and learn a new autoencoder that encodes/decodes the output of the just-learnt layer, progressively building (layer-wise) a deep architecture



Layer-wise Training: Stacked Autoencoders (6)

- Finally, the whole network can be **fine-tuned** using the available supervised data (*all the network weights are updated*)



Training Deep Networks: Weight Decay

- **Weight Decay**: a simple instance of L2-norm regularization
  - It improves the development of "regular/smoother" solutions, enhancing the generalization quality of the network and reducing overfitting

$$\texttt{obj} = \sum_j \texttt{loss}(x_i, y_i) + \sum_{h=1}^{\ell} \lambda_h \|\hat{\mathbf{w}}^h\|^2$$

This vector is flattened version of the weight matrix of layer h

  - The second element of the summation depends on a number of positive small scalar coefficients (lambdas), user-selected (it might involve only a sub-portion of the layers)
  - The norm is computed on the vector of all the weights in layer *h*
  - It favors solution with small weights

$$\frac{\partial \texttt{obj}}{\partial w_{ij}^h} = \sum_j \frac{\partial \texttt{loss}(x_i, y_i)}{\partial w_{ij}^h} + 2\lambda_h w_{ij}^h$$

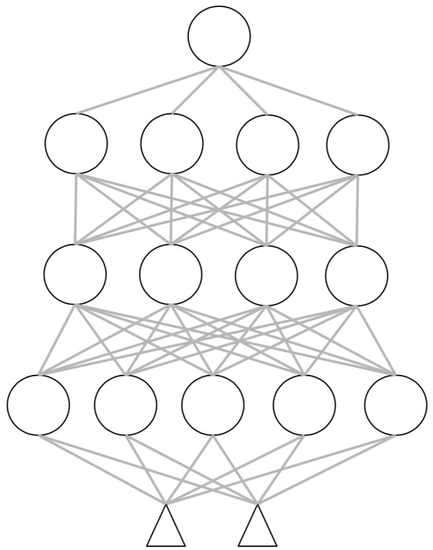
What we saw so far

New term

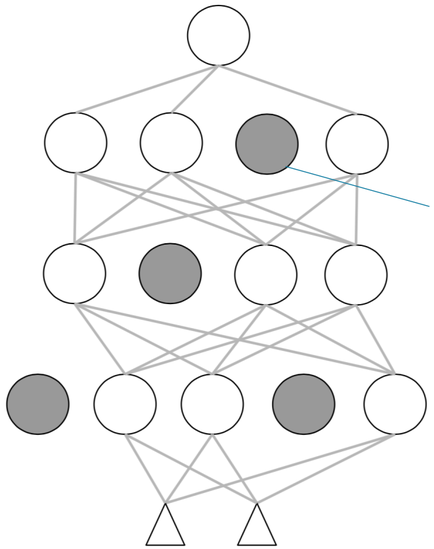
Training Deep Networks: Dropout

- **Dropout**: randomly drop units in training epochs (with a certain probability)
  - Dropped units are ignored (different units are dropped at different time instants)
  - Noisy training ➔ More robust layers
    - Layers are enforced to be robust to the dynamic dropping of the neurons that are inputs of the layer
    - It can be seen as a form of regularization
    - It reduces overfitting

Neural Net



Dropping some units

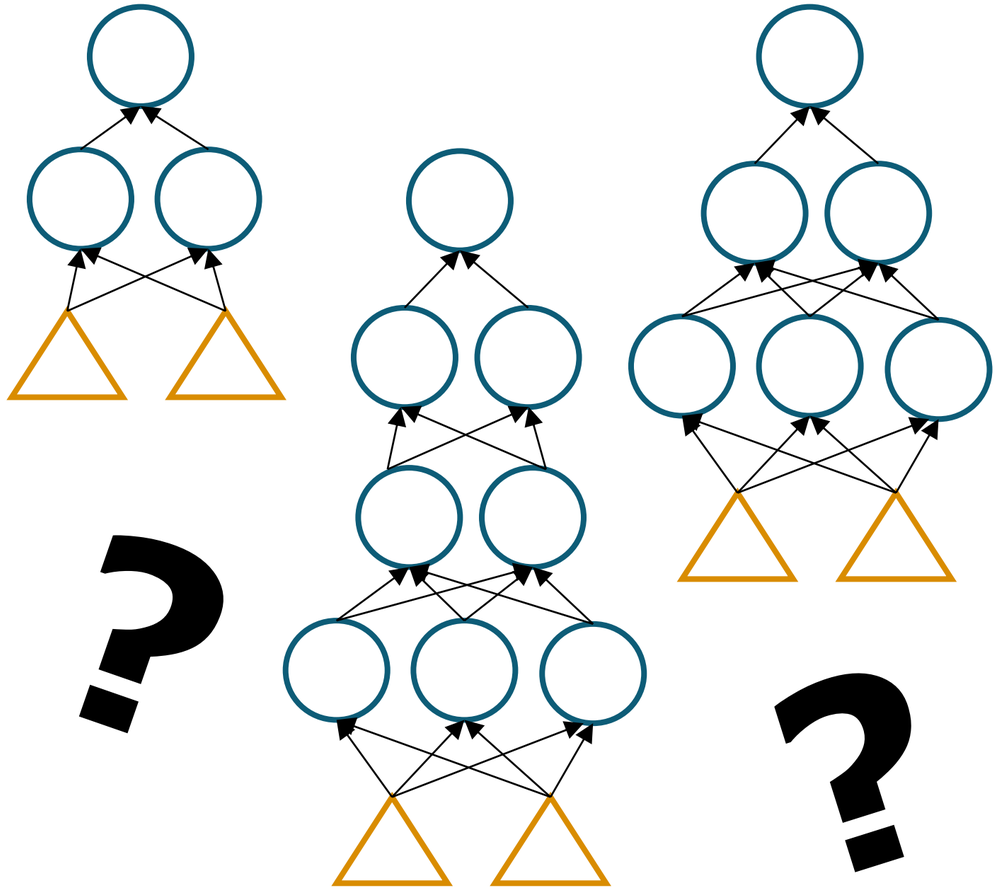


Selecting the Optimal Architecture

Stefano Melacci

Which Architecture?

- We are given a learning problem and we want to approach it with neural networks:
  - *How many layers do we need?*
  - *How many neurons on each layer?*
  - *Which activation functions?*
  - *What about the appropriate learning rate?*
  - *How many epochs should we run the training procedure?*
  - *Which initialization of the weights?*



Which Architecture? (2)

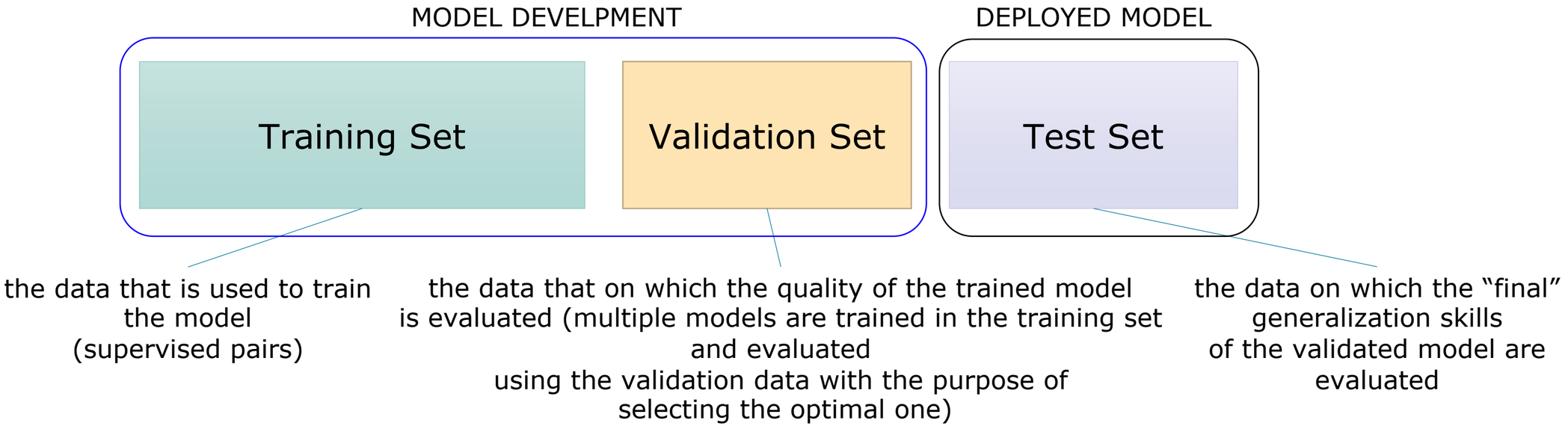
- Let us consider an award-winning network in the context of Computer Vision (object categorization)
  - The so-called ResNet model
  - **152 layers**
  - It is not a classical MLP as the ones we described in this course, but it is helpful to give an idea of how many layers researchers are experimenting in some Deep Learning problems
- Is this the best network to use?

*The the number of layers, and, more generally, the activation functions, learning rates, etc., are **task-dependent***

(we will generically talk about "network architecture", that is meant to include all these elements)

Data Splits

- The problem of determining the optimal architecture is usually solved by *training and evaluating multiple network architectures*, discovering the best one
  - Before diving into further details, it is important to remark the common data sets that are used when training and evaluating machine learning models



Data Splits (2)

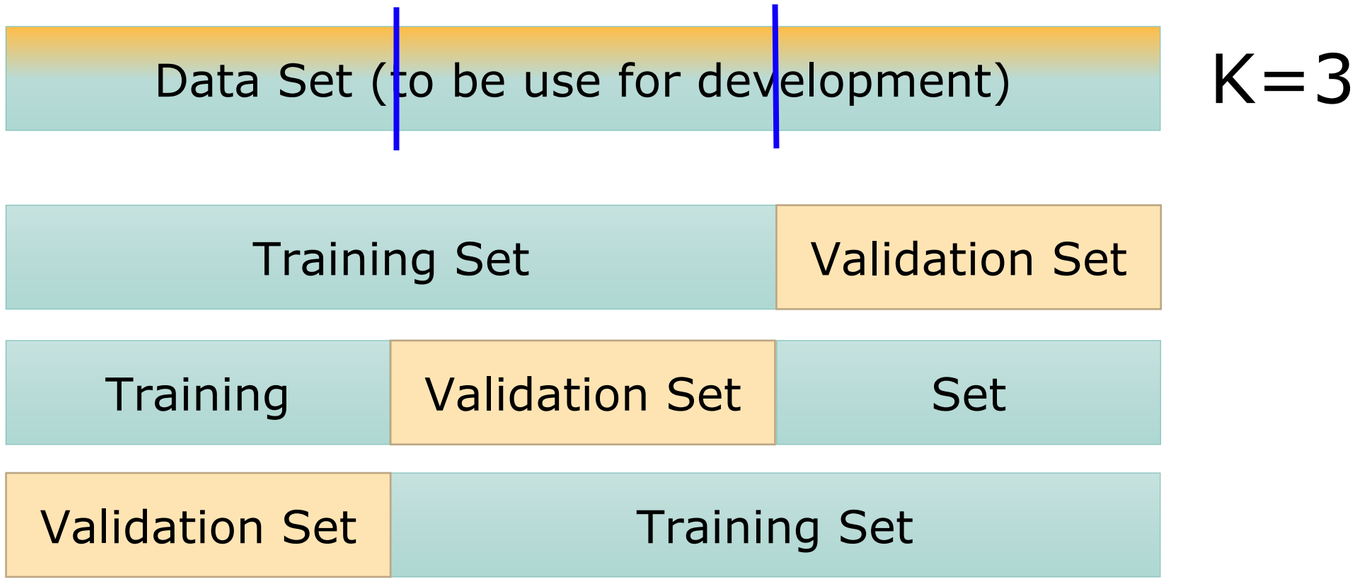
- When we are exposed to real-world problems, we do not have the use of training, validation, test set, but just a single dataset

- For example, a company might provide you a **dataset** that you are expected to use to develop you model, and they are keeping the **test data** as a private resource of the company, that the company will use to evaluate your work
  - Let's focus on such scenario…

We have to use these data to train the network(s) and find the best one

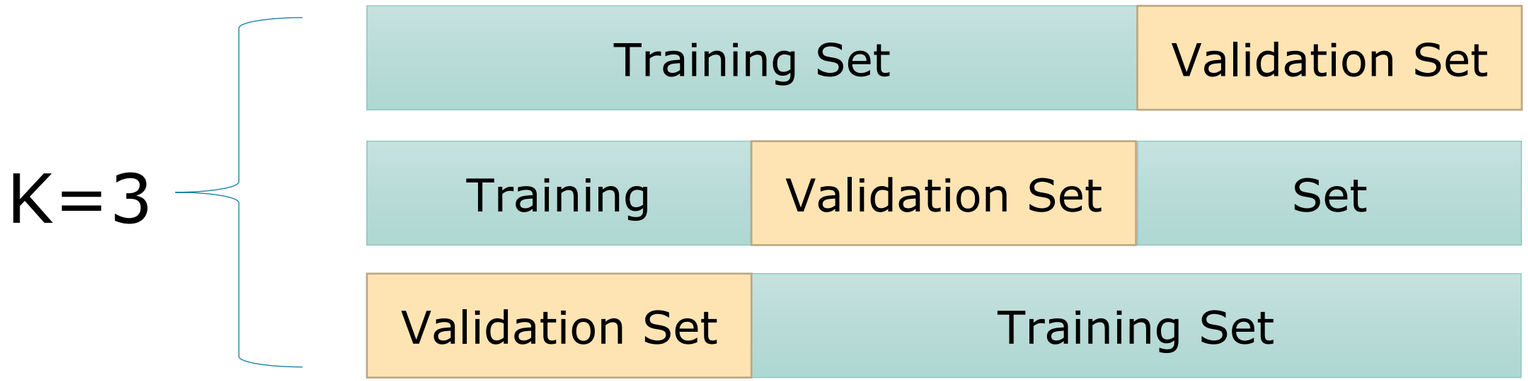
K-Fold Cross-Validation

- K-fold Cross-Validation is a very simple procedure that allows us to split the available data into multiple pairs *(training, validation set)*
  - The data is randomized and divided into *k* portions (**folds**) of equal size (balanced)
  - One fold is used as validation set, the other *k-1* are the training set



K-Fold Cross-Validation (2)

- For each neural architecture
  - We have *k* training sets and *k* validation sets
  - We train the model *k* times and we measure the performance (in the validation data) *k* times - then we can average the results to have a unique index that we can use to compare to different architectures



Measuring Performance

- We also have to define a way to measure the performance of the network
- In classification tasks, the most straightforward way is to use the **accuracy**
  - The number of right predictions divided by the total number of predictions that were computed

$$\texttt{accuracy} = \frac{\#\left[\texttt{prediction} = \texttt{class\_label}\right]}{\#\texttt{predictions}}$$

- It does not take into account the distribution of data over classes
  - For example, if your dataset has 90% of examples of class A and 10% of class B, a classifier that always outputs class A will get 0.9 of accuracy (90%)!
  - We can compute the accuracy on each class independently, and then average the results
    - Macro accuracy

Measuring Performance (2)

- Let's consider a binary classification problem (two classes, A and B)
- Let's describe the problem as the one of determining if an input example belongs to class A or not (then, of course, it will belong to class B)
  - Class A: *positive class*
  - Class B: *negative class*

- We can compute some useful measures, that are about the positive class:
  - TP: #**True Positives**
    - #Examples that were correctly classified as belonging to class A
  - FP: #**False Positives**
    - #Examples that were incorrectly classified as belonging to class A
  - TN: #**True Negatives**
    - #Examples that were correctly classified as not-belonging to class A
  - FN: #**False Negatives**
    - #Examples that were incorrectly classified as not-belonging to the class A

Measuring Performance (3)

- **Confusion Matrix**: the matrix that compares predictions and labels, counting the number of instances that falls in each of the 2x2=4 cases

Label	Prediction
A	A
A	B
A	B
B	B
B	B
B	A
B	B
B	A
B	A

PREDICTIONS

LABELS (SUPERVISIONS)

	A	B
A	1	2
B	3	3

PREDICTIONS

LABELS (SUPERVISIONS)

	A	B
A	TP	FN
B	FP	TN

Measuring Performance (4)

- We can use TP, FP, TN, FN to compute the following measures:

Already defined (right/true predictions on the total number of predictions)

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+TN+FN}$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

It measures "how good is the system when generating a positive decision" (if the system predicts the positive class for all the examples of that class, precision is 1)

$$\text{Recall} = \frac{TP}{TP+FN}$$

It is related to the tendency of the system of predicting the positive class (if the system ALWAYS predict such class, recall is 1)

$$\text{F-Measure (F1 Score)} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

It mixes precision and recall (it is in [0,1])

Measuring Performance (5)

- In the multi-class case it is straightforward to compute the **confusion matrix**, that will a *c-by-c* matrix, where *c* is the number of classes

PREDICTIONS

LABELS (SUPERVISIONS)	A	B	C
A	…	…	…
B	…	…	…
C	…	…	…

PREDICTIONS

LABELS (SUPERVISIONS)	A	B	C
A	TP_A	FN_A	
B	FP_A	TN_A	
C			

PREDICTIONS

LABELS (SUPERVISIONS)	A	B	C
A	TN_C		FP_C
B			
C	FN_C		TP_C

Measuring Performance (5)

- Precision, Recall, F1 can be computed *for the predictor of each class*
  - **Precision of class *i***: divide the element in position *(i,i)* by the sum of the *i-th* row
  - **Recall of class *i***: divide the element in position *(i,i)* by the sum of the *i-th* column

- It is pretty common to average the **F1**'s of the *c* classes in order to have a single, multi-class index, if needed
  - Macro averaging

PREDICTIONS

LABELS (SUPERVISIONS)	A	B	C
A	TP_A	FN_A	
B	FP_A	TN_A	
C			

Selecting the Best Architecture

- Finding the optimal neural architecture for the considered task requires several trainings
  - It is a **costly** procedure, frequently underestimated by people that approach Machine Learning for the first time

- We have to try a **large number** of configurations with significantly different properties

- Several possible strategies
  - "Random" search (evaluate a batch of different configurations)
  - **Grid search** (next slide)
  - A mixture of both of random and grid search (next slide)

Selecting the Best Architecture (2)

- Grid search
  - Define the parameters that will be involved in the search procedure
    - Examples: Learning rate, number of layers, activation functions, …
  - For each of them, define a list of different values to evaluate
    - Example: Learning rate = [0.01, 0.001, 0.000001]
  - Test all the possible combinations of grid values

- Grid search is time consuming and it quickly becomes unfeasible
  - Coarse-to-fine search: define coarse grids and find the best configuration, then define new grids composed of values that are close to the ones of such configuration (**fine tune**)

- Another possible strategy consists in trying randomly selected configurations
  - And then **fine tune** the best one

Selecting the Best Architecture (3)

- Multiple trainings must be executed on the same architecture, given **different initializations of the weights**
  - The network accuracy can be averaged over the multiple training instances
  - In some tasks, especially in the ones that exploit large-scale fully supervised data, the quality of the solution is pretty stable with respect to different initialization

Selecting the Best Architecture (4)

- Frequently underestimated constraints in the selection of the optimal architecture are
  - The hardware-spec of the target machine in which the model will be deployed
  - The hardware-spec of the machine that will be used to train the network
- It is useless to train a large model if we are going to deploy it in a machine were each prediction would take a long time or where the available memory is not enough to store the trained model
- It is also useless to select large/deep models that would make the training times prohibitive
  - Evaluate smaller models or on subsets of the training data
  - Then make a last, long-term, training attempt using the whole training data or increasing the number of units in the selected model