# QML-Mod2-Classical Machine Learning

Riccardo Marega

March 2025

## Indice

# 1 Introduction to classical machine learning -22/03/2025

Despite maybe we were not aware but we've already trained models, in our life, with machine learning algorithms: an example is given by the tools asking the users to spot cars, traffic signals, etc. in some website before entering them. An other example is given by all the filters that one can apply before taking a picture when using social media.

**Artificial Intelligence vs Machine Learning vs Deep Learning**

- artificial intelligence: any technique that enables computers to mimic human intelligence. It includes machine learning
- machine learning: a subset of AI that includes techniques that enable machines to improve at tasks with experience. It includes deep learning
- deep learning: a subset of machine learning based on neural networks that permit a machine to train itself to perform tasks

Machine learning teaches computers to do what comes naturally to humans and animals: learning from experience. Machine learning algorithms use computational methods to "learn" information directly from data, without relying on a predetermined equation. These algorithms adaptively improve their performance as the number of samples available for learning increases.
Machine learning includes two kinds of modalities to train the algorithm: supervised and unsupervised learning. In the latter we don't give any label to the algorithm and we let it find itself the correct answer to the given problem.
The choice of using ML algorithms arise when:

- we can't rely on rule-based systems
- the rules depend on too many variables, and many of them overlap or need to be optimized
- scalability becomes an issue

An example of machine learning algorithm is to be found in surgical data science, where AI is at service of surgeons.
The choice of the best algorithm to apply has to be done considering the dimensions and the type of data one has to manage.

$$\text{machine learning} \begin{cases} \text{Unsupervised learning} \begin{cases} \text{Clustering} \end{cases} \\ \text{Supervised learning} \begin{cases} \text{Classification} \\ \text{Regression} \end{cases} \end{cases}$$

**Supervised learning**: the model is trained under the assumption that for each input the label in known.
**Unsupervised learning**: the model identifies meaningful patterns within the input data, which does not come with any label.
In supervised learning methods, the output is already known, and the goal of training is to map inputs into the corresponding outputs. To build a model, the machine learning is fed with a large amount of input data along with their corresponding labels. Supervised learning uses classification and regression techniques to develop predictive models. The difference between classification and regression is that the first one does a prediction of discrete outputs while the second one does a prediction of continuous outputs.
For the unsupervised learning case it's not necessary to supervise the model or provide labeled input data. The algorithm begins to learn from the data without guidance. The model uses unlabeled data to identify new information. Since there are no known output values to establish a logical relationship between input and output, specific techniques are used to extract rules, patterns, and grouping of similar types. Machine using unsupervised learning algorithms discover patterns to find meaningful insights. For example, the system can learn to distinguish between dogs and cats by understanding the features and characteristics of each animal.
Every machine learning workflow starts with three key questions:

- What type of data are we working with?
- What do we want to extract or learn from the data?
- What is the context or domain of application?

The main problem with unsupervised learning is that evaluating its performance is not always immediate. We choose unsupervised learning when we aim to understand: the distribution of data, the clustering of data based on similar characteristics and the dimensionality reduction (needed to obtain a more concise and efficient representation).
Note that one can combine supervised and unsupervised learning.

## 1.1 Input data

Standard algorithms of ML usually take as inputs features and characteristics extracted from the data set. These features are handcrafted or manually designed or selected. Features are relevant pieces of information that help solve the task at hand. **The quality of a machine learning model depends on its ability of extracting features.**
The data is divided in a training set (used exclusively during the training phase) and a validation set (used to monitor the process and optimize model trainings). An algorithm is subject to underfitting of the data set if it has a poor performance on the training set. This happens because the model fails to learn the relationship between input data an their corresponding labels. Otherwise it

is said being subject to overfitting if it works well with the data training but has poor results with data of the test set. This occurs because the model memorize the data it has seen and is unable to generalize to unseen examples.
For a fully functional training of the model we introduce a third data set called validation set. These data will be used exclusively during the training phase to monitor the learning process and optimize model training. This is called validation set because it serves to validate the results obtained in the training set. If the performance is poor, we may need to adjust the model's hyperparameters (in the case of ML) and retrain the model until the validation results are satisfactory. When implementing this third dataset the testing set will be exclusively used after training is complete to evaluate model's performance.

**Cross validation strategy:** one can repeat the whole training of the model swapping every time the data from one set to another (what was first, for example, in the training set now will go in the validation set).

**Generalization** In Machine Learning, generalization refers to a model's ability to perform well on unseen data, meaning data that was not used during training. A well-generalized model captures the underlying patterns in the data rather than memorizing specific examples, allowing it to make accurate predictions on new inputs. Poor generalization can lead to overfitting (where the model performs well on training data but poorly on new data) or underfitting (where the model fails to learn meaningful patterns from the training data).
When an algorithm makes consistent mistakes, we say it has a "bias".
A **bias** is a distortion of the training data which is propagated in the algorithm.
The goals related to the supervised trainings are a low error during training, validation and testing phases. The selection of the algorithm is based on:

- velocity of the training
- storage capacity
- accuracy on new data prediction
- transparency and interpretability

## 1.2 Machine Learning examples

- **Logistic regression**: the model is trained to predict the probability of a binary choice. Due to its simplicity, logistic regression is commonly used as a starting point for binary classification problems. This algorithm is normally used when the data can be clearly separated by

a single decision boundary and in general as baseline to evaluate more complex classification methods.

- **k nearest neighbor (kNN)**: is a pattern recognition algorithm based on distance of the data. If a subject A has, for example, characteristic close to the one of a subject B, then probably they are associated to the same category. This kind of algorithm is normally used: for establishing baseline learning rules, when memory usage is not a major problem and when prediction speed of the trained model is not a major concern.

A **decision tree** is a supervised learning algorithm used for both classification and regression tasks. It has a hierarchical tree structure composed of a root node, branches, internal nodes and leaf nodes. A decision tree starts with the root node, which has no incoming branches. The outgoing branches from the root lead to internal nodes, also known as decisions nodes. Both root and internal nodes perform evaluations to split the data into homogeneous subsets, which are represented by leaf nodes or terminal nodes. Leaf nodes represent all possible outcomes (either continuous or discrete) within the dataset.
ML learning models in training phase learn a set of rules which depends both on the data set and a fixed combination of hyperparameters. The automatic learning of a model is not a single process, indeed it is necessary to experiment different models fixing different values for the hyperparameters.

### 1.2.1 Code overview

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(n_estimators=?, max_depth=?)
```

Imports the RandomForestClassifier model from the scikit-learn library. This model is a supervised learning algorithm based on decision trees. Creates an instance of the Random Forest classifier with two key parameters:

- `n_estimators=?`: defines the number of trees in the forest (typically between 10 and 1000).
- `max_depth=?`: sets the maximum depth of the trees (can be `None` to allow them to grow until all leaves are pure).

## 1.3 Cross validation and hyperparameter tuning

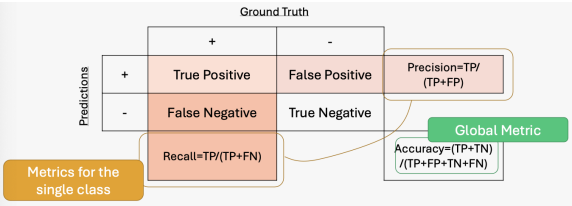
We divide the training and the validation set in multiple sets so that the model can be trained with all the available data. The model's performance for the same model with different hyperparameters combinations strongly depends on the specific data split. Each model is trained and evaluated only once, so its performance is tied to that single evaluation. But this raises an issue: we might get very different results when training and validating on different subsets of the same data. What

if we could split the training and validation sets multiple times, each time using different subsets of data, train and evaluate our models repeatedly, and observe model performance across several rounds of evaluation? → **k-fold Cross Validation**.

**Cross validation** is statistical method used to estimate the ability of different models of performing automatic learning. This procedure is done by defining a parameter k which represents the number of sets in which every set is subdivided. In this approach the dataset is randomly divided into k groups of roughly equal size. The model is trained on k-1 folds and validated on remaining fold. This process is repeated k times, each time using different fold as the validation set.

We can perform cross validation for hyperparameter tuning in either inner loops or outer loops

### 1.3.1 Evaluation metrics for classification




**Accuracy:** $\frac{TP+TN}{TP+FP+FN+TN}$

**Recall (Sensitivity/True positive rate):** $\frac{TP}{TP+FN}$

**Precision:** $\frac{TP}{TP+FP}$

**Specificity:** $\frac{TN}{TN+FP}$

**F1 score:** $2 \times \frac{Precision \times Recall}{Precision+Recall}$

**ROC Curve & AUC (Area under the curve)**

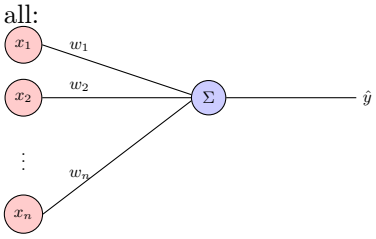
- **ROC curve:** Plot true positive rate (recall) vs. False Positive rate (FPR=$\frac{FP}{FP+TN}$) at various threshold.
- **AUC (Area Under Curve)** Probability the classifier ranks a randomly chosen positive higher than a randomly chosen negative (AUC = 0.5: random , AUC = 1: perfect).

# 2 Introduction to deep learning -28/03/2025

## 2.1 Training an artificial neural network

**Modeling problems** the first example is defining the position of a car whose position is given exactly by $d(t) = d_0 + vt$. This problem is fully solvable. That is not always the case; indeed, a problem could be characterized by a large number of variables. The problem with working with large number of variables is that not always every variable has the same importance as the others. The problem, can be reduced to: $y = \alpha x_1 + \beta x_2 + \gamma x_3$.
**Machine learning** allows to solve complex problems in which we can easily tell what variables are involved.
A typical problem that could be modeled is image recognition.
In ML we have a phase that does not appear in deep learning which is feature education: in traditional ML, model require pre-computed features that are manually designed based on domain knowledge. This feature engineering requires human expertise to identify the most relevant attributes for a given task.
Deep learning eliminates the need for manual feature extraction by learning hierarchical representations directly from raw data. Using AI networks, DL can automatically detect patterns at multiple levels of abstraction.

**Artificial Neural Networks** Biologically, neurons are unique cells that can communicate with one another, thus propagating information.
An artificial neuron receives n inputs $(x_1, ..., x_n)$, each scaled by a factor (weights) and sums them all:



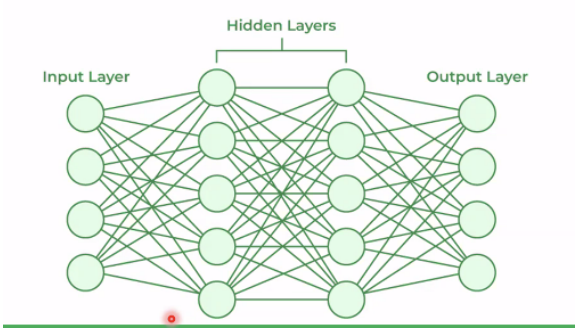

To map all the space is necessary to add a factor b called bias, such that:

$$y = \sum_{i=1}^{n} w_i \times x_i + b = w_1 \times x_1 + ... + w_n \times x_n + b.$$

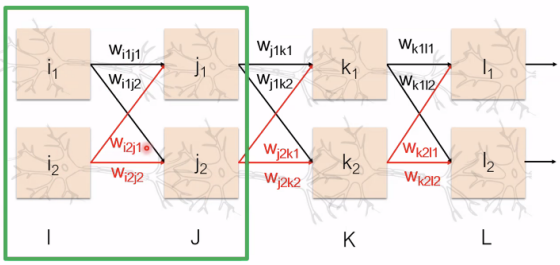
Weights and bias are the parameters of the neural network. Note that very few models are correctly modeled using linear combinations of the variables; indeed, we introduce an activate factor such that:

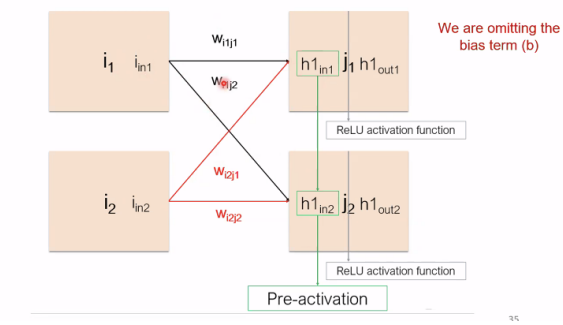
$$y = \sum_{i=1}^{n} g(w_i \times x_i + b) = g(w_1 \times x_1 + ... + w_n \times x_n + b).$$

Typical activation functions are: sigmoid, tanh, ReLU ($\max(0, x)$), Leaky ReLU ($\max(0.1x, x)$). The sigmoid and tanh functions limit the output, indeed, the output is always confined between 0 and 1.




This is what a simple neural network looks like: an input layer, a bunch of hidden layers, and an output layer. Each layer extracts information from the input and transmits it to the next one for further data processing.




$$\begin{bmatrix} i_{in1} i_{in2} \end{bmatrix} \times \begin{bmatrix} w_{i1j1} w_{i1j2} \\ w_{i2j1} w_{i2j2} \end{bmatrix} = \begin{bmatrix} h1_{in1} h1_{in2} \end{bmatrix}$$

$$\begin{bmatrix} h1_{out1} h1_{out2} \end{bmatrix} = ReLU(\begin{bmatrix} h1_{in1} h1_{in2} \end{bmatrix})$$

What we are building is a neural architecture. All these parameters will have to be adjusted. This process is exactly what is called "train of the neural network". The training process continues by making mistakes.

Let's consider the problem of predicting the cost of a house: y is the real cost and $\hat{y}$ is the predicted one. y is also called ground-truth value and is used to supervise the training. The idea under the training process lays in computing the error, and what we want indeed is to try to minimize this error. Error minimization is an optimization problem: I need a parameter configuration that explains the problem in the best possible way (i.e. I get as close as possible to the ground-truth value)

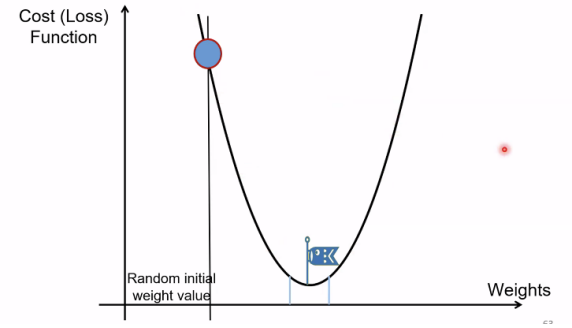
$$error = |y - \hat{y}|.$$

To optimize the parameters of a neural network we use the **gradient method**. What we always know when computing an algorithm is whether the error is increasing or decreasing.

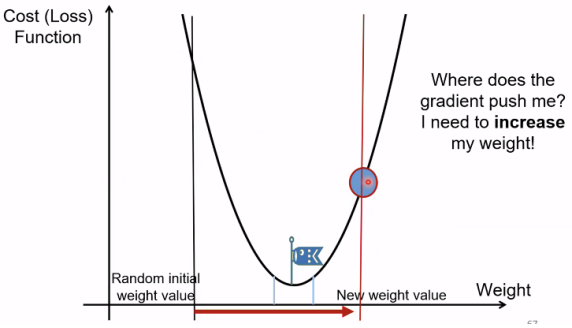
**Gradient descent** is the most used optimization technique to minimize the error. It means that, for each step, I compute the error and change the parameters in order to minimize it according to its dependence on each parameter.

$$error = f(w, b)$$

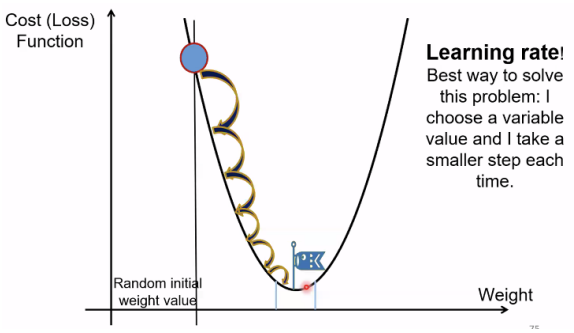
this function is called **cost function**. First order derivative (gradient) is a measure of dependence between each independent variable (w and b) and the dependent one (error).




Initializing the parameters randomly gives better results.



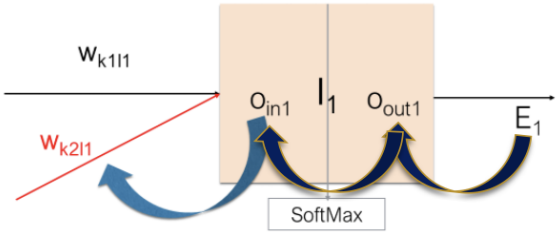
and so on till we reach the minimum.
In general the first step to reach the minimum is the bigger among all the steps that will be done. In this case we talk about learning rate. While "learning" we have to decrease the learning rate (i should take smaller steps) in order to reach the target.




Mathematically, training a neural network means to change its parameters N times in order to minimize the cost function (error). Each update can be expressed as

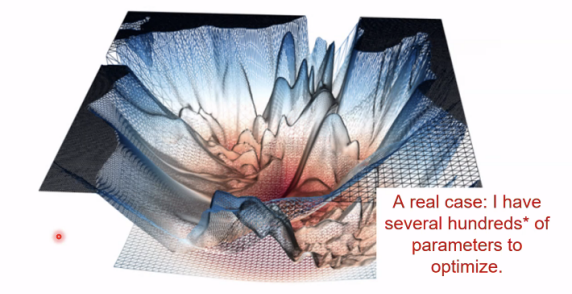
$$w_{t+1} = w_t - \eta \times \nabla error$$

where $\eta$ is the learning rate and $\nabla error$ is the gradient of the cost function with respect to w. A typical loss function is the MSE. The only constraint that we impose to the loss functions is to be differentiable everywhere. The error can be computed only from the output of the layers:




$$\frac{dE_1}{dw_{k1l1}} = \frac{dE_1}{dO_{out1}} \times \frac{dO_{out1}}{dO_{in1}} \times \frac{dO_{in1}}{dw_{ikl1}}$$

For a forward pass of information we have an error back-propagation. This kind of neural network is called fully connected.

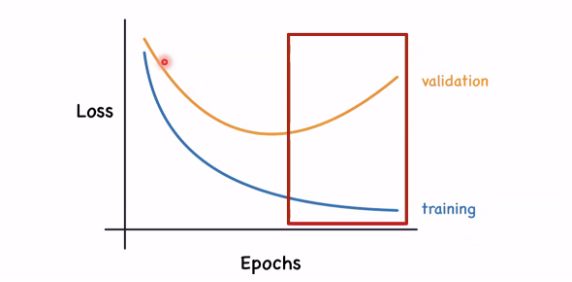



Usually, what we need is a dataset, i.e. a set of (x,y) samples where x is the input and y is the output. The dataset is divided into three sub-sets: a training set, a validation set and a test set. The training process involves parameters optimization on the same training samples several times, or epochs.

AI does not really "learn": it just finds patterns. It's like a student who only studies math by doing every exercise on the book but never reads a page of theory. How can we make sure that the student is learning and not just memorizing? During the test, the student has no way to learn new knowledge because it has no supervision (correct answers). It will be up to the teacher to evaluate the results from the test and assess the student's knowledge. Therefore, our neural network needs to perform well both on the training set and on the two test sets. What happens if it doesn't? The loss function on the training set has a very nice trend: as the network is iteratively optimized on the training sample, its error on these samples decreases. However, the loss function on the validation test doesn't seem as good: while the student improves with those "training" exercises, they make a lots of mistakes during the exam.

Two things are never to be expected:

- no error or 100% accuracy
- better results in validation




However, we should minimize the distance between training and validation performance: **overfitting**.

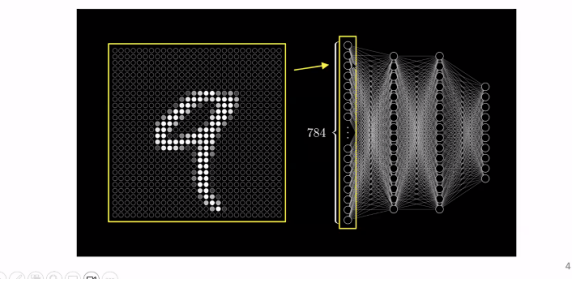
The easiest task for which ML and DL are used are regression and classification: regression is used to predict a number from a continuous set of numbers (e.g. prediction of the price of a house) while classification is used to predict a number from a discrete set of numbers.
The number of neurons in the final layer will depend on the task: for the regression case we need to predict one number → one neuron is needed, while in case of classification we need to label the inputs as one of the possible N classes → N neurons are needed. The optimal number of neurons in the hidden layers (as well as the number of the hidden layers) cannot be assessed a priori. The more the neurons and the layers, the more abstract information we can extract from the input data.

## 2.2 Artificial neural networks: tasks

**Check out the following site:** playground tensorflow

### 2.2.1 Binary classification

**Classification** is the most studied problem in DL. To classify means to assign a label to the input data among a closed set of labels.
The **MNIST** dataset is on of the most famous one: our network needs to classify the input image as one of the 10 possible labels (the digits 0-9).
How do we feed these images to the neural network?
Pixel by pixel: MNIST images are $28 \times 28 = 784$ pixels.




Pixels are integer values between 0 (black) and 255 (white) that are processed by neural network as numbers.
**Check out:** The stilwell brain
In the lecture of today we are going to classify whether a digit (lower than 10) is odd or even. Therefore, ours is a binary classification task. For classification, a perfect loss function is (binary) cross-entropy.

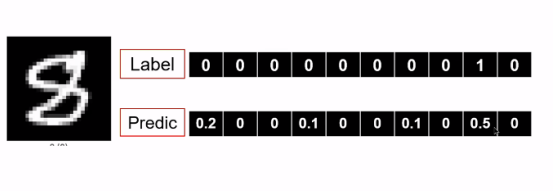



Figura 1: Note: 0.6 instead of 0.5

I want my prediction to be a probability distribution. The sum of all my elements needs to be 1, so that I can "compare" it with the label (also a probability distribution). In case of a odd vs even prediction the binary classification will give more precise results.

**Loss function: Binary cross enrtopy**

$$-\sum_{j=1}^{M} y_j \log(p(y_j))$$

$$-\sum_{i=1}^{N} y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))$$

**Final activation: softmax**

In order to obtain a probability distribution, I need to use the softmax activation function in the outer layer.

$$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} = \left[ \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \right] = \begin{bmatrix} 0.02 \\ 0.9 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

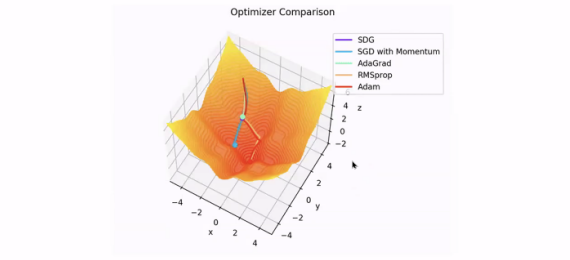
**Metric: accuracy**

$$\begin{bmatrix} \text{True positive (TP)} & \text{False Positive (FP)} \\ \text{False Negative (FN)} & \text{True Negative (TN)} \end{bmatrix}$$

where

- Recall = $\frac{\sum TP}{\sum TP+FN}$
- Precision = $\frac{\sum TP}{\sum TP+FP}$
- Accuracy = $\frac{\sum TP+TN}{\sum TP+FP+FN+TN}$

**Optimizer: Adam** (Adaptive moment estimator)



# 3 Advanced topics -29/03/2025

## 3.1 Convolutions from scratch

**Edge detection** is an old but gold problem in computer vision that involves detecting edges in an image to determinate object boundaries and thus separate the object of interest.
One of the most popular techniques for edge detection is the Conny Edge Detection algorithm.
An edge is a point of rapid change of intensity of the image function. The gradient points in the direction of the most rapid increase

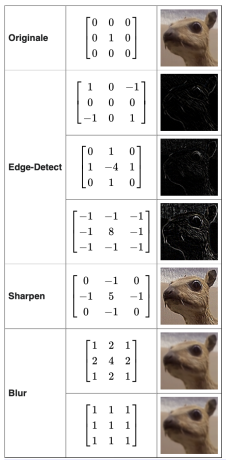
$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}.$$

Using filters (aka matrices) is possible defining the edges in an image. An example is given by:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

A **convolution** (of images) is simply an elementary multiplication of two matrices followed by a sum:

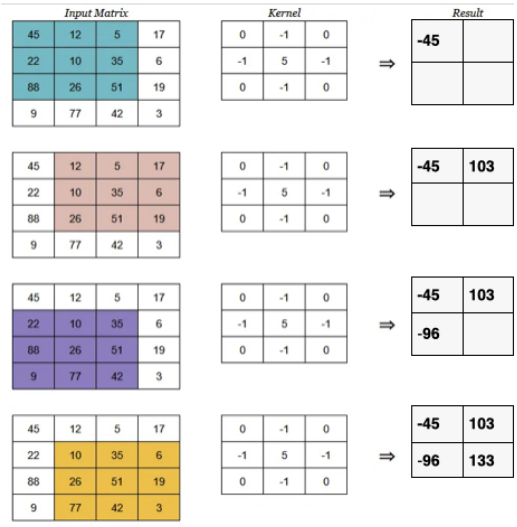
- take two matrices (which both have the same dimension)
- multiply them, element by element
- add up the elements




An image is just a multidimensional matrix, but unlike traditional matrices, images (RGB) can also have a depth. The kernel should be thought as a small matrix that is used for blurring, sharping,

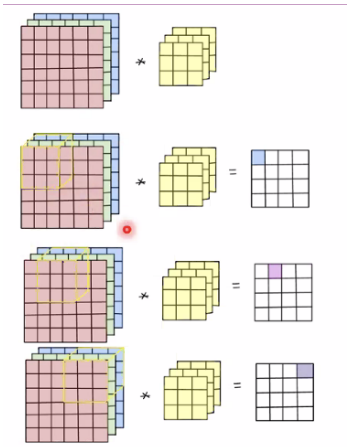
edge detection, and other image processing and functions. It is common to define the kernel by hand to achieve various image processing functions, edge detection: all these operations are hand-defined forms of kernels designed specifically to perform a particular function. A question then arises: is there a way to automatically learn these type of filters? And even use these filters for image classification and object detection? Of course there is: **CNN**.

**Kernel** Most of the kernels we usually see are $N \times N$ square matrices. We use an odd kernel dimension to ensure that there is a valid integer coordinate in the center of the image. In image processing, a convolution requires three components: an input image, a kernel matrix to apply to the input image and an output image to store the results of the input image convolved with the kernel.
The process of "sliding" a convolutional kernel over an image and storing the output decreases the spatial dimensions of the output data. This decrease in spatial dimension is simply a side effect of applying convolutions to images.




| | |
|---|---|
| 45*0 + 12*(-1) + 5*0 + 22*(-1) + 10*5 + 35*(-1)+ 88*0 + 26*(-1) + 51*0 | -45 |
| 12*0 + 5*(-1) + 17*0 + 10*(-1) + 35*5 + 6*(-1)+ 26*0 + 51*(-1) + 19*0 | 103 |
| .... | |
| .... | |

However, in most cases, we want the output image to be the same size as the input image. To ensure this, we apply padding. The fact that the output is smaller than the input does not seem to be a big problem: we did not lose much data because most of the important features are located in the central area of the input. The only case when losing this information is a real problem is when much information in concentrated on border of the image. Padding could be done using zero elements or just copies of the border.
For an image we have:



An RGB image is represented as a $6 \times 6 \times 3$ volume, where the 3 correspond to the 3 color channels (RGB). To detect edges or other features in this image, one could convolve the $6 \times 6 \times 3$ with a 3-D filter. Then also the filter itself will have 3 levels corresponding to the red, green and blue channels. So, the filter also has a height, a width and a number of channels. The number of channels in the image must match the number of channels in the filter.
In convolutional neural networks, convolutional layers are not only applied to input data, such as pixel values, but can also be applied to the output layers. The sequence of convolutional layers allows a hierarchical breakdown of the input. Consider that filters operating directly on the raw pixel values will learn to extract low level features from the starting image, such as lines. Filters operating on the output of the first convolutional layer can extract features that are combinations of lower level features, such as features that comprise multiple lines to extract shapes. This process continues until very deep layers extract faces, animals, houses and so on. The abstraction of characteristics to ever higher orders increases with network depth.
**Kernel size** The kernel size defines the convolution field of view.
**Padding** The padding defines how the edge of a sample is handled. A convolution with padding will keep the spatial dimensions of the output equal to those of the input, while consvolutions without padding will crop some of the edges if kernel il larger than 1.
**Stride** The stride defines the size of the kernel step when passing through the image. Although the default setting is usually 1, you can use a stride of 2 to downsample an image (this is used to improve the features of the network).

### 3.1.1 Types of convolutions

**Dilated/Atrous convolution** The atrous convolutions introduce another parameter called the rate of expansion. This parameter defines the distance between values of a kernel. This way you get

a wider field of view at the same computational cost. That type of convolution is use to understand large partial context.

**Spatial separable convolutions** A separable spatial convolution simply splits one kernel into smaller kernels. With fewer multiplications, the computational complexity decreases (the network has few parameters to learn).

**Depth-wise separable convolutions** We have an RGB input image (with 3 channels). After convolutions, a feature map can have more channels (as usually happens). Each channel can be as a particular interpretation of the image. Similarly to spatial separable convolution, a deep separable convolution divides a kernel into two separate kernels that perform two convolutions: the **deep convolution** and the **point convolution**.
**Width multiplier** allows scaling the network's width to control the number of parameters and computation.
**Resolution multiplier** enables adjusting the input image resolution to further reduce computational requirements.

## 3.2 Convolutional neural networks for image classification

The visual cortex has hierarchical structure: LGB (lateral geniculate body)-> simple cells -> complex cells -> lower order hypercomplex cells -> higher order hypercomplex cells
We can think that these complex cells are performing an aggregation of activations using functions such as the maximum, the sum. In this way, these cells can recognize edges and orientations.
Deep neural networks are normally organized in alternate repetitions of linear and non-linear operators. The reason for having multiple layers of this type is to build a hierarchical representation of the data.
The local pixels assemble to perform simple patterns like oriented edges. These borders are in tum combined to form patterns that are even more abstract. We can continue to build above these hierarchical representations until we arrive at the objects that we observe in the real world.
This compositional and hierarchical nature that we observe in the natural world is therefore not only the result of our visual perception, but it is something true in the physical world. At the lowest level of description, we have elementary particles, which are composed to form atoms, more atoms form molecules, and we continue to increase this process until we form materials, parts of objects and finally complete objects in the physical world.

## 3.3 Convolutional neural networks for classification task

### 3.3.1 Image classification

is the task of assigning to an input image a label belonging to pre-set set of categories.

**Pro convolutional neural networks** The 3D element that flows along the 3 image channels (RGB) is called convolutional kernel. The convolutional kernel slides over the image (in the case of the input layer) and extract features or features map. Unlike fully connected networks where the input was a carrier, CNNs operate on volumes (multi-channel image).
The pixels of the feature map with the same color are from the same kernel. Both types of networks learn by updating the weights which, in the case of fully connected networks, are the values of the connections whereas, in the case of convolutional neural networks, they are the values of the kernels and connections. In both types of networks, the neuron receives an input which is a combination of weighted inputs. This combination of weighted inputs represents the overall level of neuron excitation and is given as input to an activation function which produces some limited output.

**Other layers in CNNs: Pooling** Pooling provides a form of translation invariance: small shift in the input can lead to the same output. Statistics over neighboring features to reduce the size of the feature maps:

- separate the image into non-overlapping subimages

- select the maximum/average/... in each layer

### 3.3.2 Architectures

- **LeNet-5** 1998

| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 32x32 | - | - | - |
| 1 | Convolution | 6 | 28x28 | 5x5 | 1 | tanh |
| 2 | Average Pooling | 6 | 14x14 | 2x2 | 2 | tanh |
| 3 | Convolution | 16 | 10x10 | 5x5 | 1 | tanh |
| 4 | Average Pooling | 16 | 5x5 | 2x2 | 2 | tanh |
| 5 | Convolution | 120 | 1x1 | 5x5 | 1 | tanh |
| 6 | FC | - | 84 | - | - | tanh |
| Output | FC | - | 10 | - | - | softmax |

- **Alexnet** 2012

| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 227x227x3 | - | - | - |
| 1 | Convolution | 96 | 55 x 55 x 96 | 11x11 | 4 | relu |
| | Max Pooling | 96 | 27 x 27 x 96 | 3x3 | 2 | relu |
| 2 | Convolution | 256 | 27 x 27 x 256 | 5x5 | 1 | relu |
| | Max Pooling | 256 | 13 x 13 x 256 | 3x3 | 2 | relu |
| 3 | Convolution | 384 | 13 x 13 x 384 | 3x3 | 1 | relu |
| 4 | Convolution | 384 | 13 x 13 x 384 | 3x3 | 1 | relu |
| 5 | Convolution | 256 | 13 x 13 x 256 | 3x3 | 1 | relu |
| | Max Pooling | 256 | 6 x 6 x 256 | 3x3 | 2 | relu |
| 6 | FC | - | 9216 | - | - | relu |
| 7 | FC | - | 4096 | - | - | relu |
| 8 | FC | - | 4096 | - | - | relu |
| Output | FC | - | 1000 | - | - | Softmax |

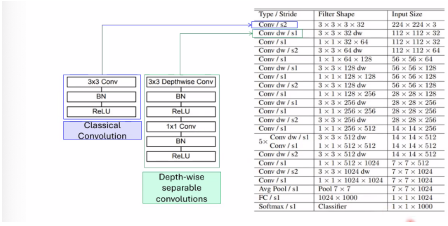
In deep learning application, the ReLU activation feature is among the most popular. ImageNet is a large dataset of multimedia data annotated manually and divided into 1000 categories.

- **Googlenet inception** 2014
  The inception module relies on several convolutions with reduced kernel size to drastically lower the number of parameters.

- **VGG16** 2014
  Characterized by a total of 16 layers with weights, that is, of parameters which are implemented.

- **ResNet50** 2014
  Particularity: skip connections.

- **MobileNet V1**
  MobileNet is designed to be a lightweight architecture optimized for mobile and embedded devices. It uses a new type of convolutional layer, known as Depthwise Separate convolution which comprises a depthwise convolution and a pointwise convolution.




| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | $3 \times 3 \times 3 \times 32$ | $224 \times 224 \times 3$ |
| Conv dw / s1 | $3 \times 3 \times 32$ dw | $112 \times 112 \times 32$ |
| Conv / s1 | $1 \times 1 \times 32 \times 64$ | $112 \times 112 \times 32$ |
| Conv dw / s2 | $3 \times 3 \times 64$ dw | $112 \times 112 \times 64$ |
| Conv / s1 | $1 \times 1 \times 64 \times 128$ | $56 \times 56 \times 64$ |
| Conv dw / s1 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 128$ | $56 \times 56 \times 128$ |
| Conv dw / s2 | $3 \times 3 \times 128$ dw | $56 \times 56 \times 128$ |
| Conv / s1 | $1 \times 1 \times 128 \times 256$ | $28 \times 28 \times 128$ |
| Conv dw / s1 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 256$ | $28 \times 28 \times 256$ |
| Conv dw / s2 | $3 \times 3 \times 256$ dw | $28 \times 28 \times 256$ |
| Conv / s1 | $1 \times 1 \times 256 \times 512$ | $14 \times 14 \times 256$ |
| $5\times$ Conv dw / s1 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| $5\times$ Conv / s1 | $1 \times 1 \times 512 \times 512$ | $14 \times 14 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 512$ dw | $14 \times 14 \times 512$ |
| Conv / s1 | $1 \times 1 \times 512 \times 1024$ | $7 \times 7 \times 512$ |
| Conv dw / s2 | $3 \times 3 \times 1024$ dw | $7 \times 7 \times 1024$ |
| Conv / s1 | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$ |
| Avg Pool / s1 | Pool $7 \times 7$ | $7 \times 7 \times 1024$ |
| FC / s1 | $1024 \times 1000$ | $1 \times 1 \times 1024$ |
| Softmax / s1 | Classifier | $1 \times 1 \times 1000$ |

*Batch normalization:*

$$\hat{x} = \frac{x - \mu}{\sigma},$$

where $\mu$ is the mean of x in mini-batch and $\sigma$ is the std of x in mini-batch.

- **DensNet**
  Densnet emphasizes strong feature reuse within the network through dense connections. Each

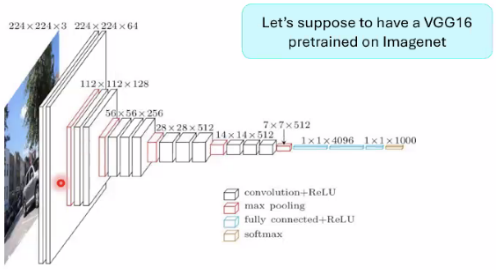
layer receives inputs from all preceding layers in a dense block. Dense connections include direct connections between layers within dense blocks, reducing the number of parameters and enabling better gradient flow. Growth rate controls the number of output feature maps produced by each layer within a dense block. Compound scaling: scaled the network's depth, width and resolution uniformly to find an optimal balance model size and accuracy.

## 3.4 Training a covolutional neural network

All previous CNNs have been trained on ImageNet to classify 1000 classes. This means that all CNN weights are available online. It's possible to use the pre-trained CNNs.

### 3.4.1 Transfer learning and fine tuning

It consists in taking the characteristics learned on a problem and exploiting them on a new similar problem. Transfer learning is usually used for tasks where the dataset has too little data to train a complete model from scratch. But how can we implement transfer learning?
What is done is adding new layers (trainable) of top of a set of layers previously frozen (so that the information contained in them is not destroyed during future training cycles) and then pass the new data set into the "new" network and record the output of one (or more) levels from the base model (this operation is called feature extraction). This result is used as input data for a new "smaller" model to be trained. The training of this model is called **fine-tuning**.
How many layers of the original model do I freeze? It depends on the task.




I can try to freeze all layers up to the last convolutional layer and train a mini-network composed of: convolutional block and 3 fully connected blocks
I could also think of not freezing any layer and initializing the net weights with the training weights on Imagenet.
Transfer learning is about "transferring" the representation learned during training of a CNN to another problem. For example, one can use pretrained CNN features to initialize the weights of a new CNN, developed for a different task. Fine tuning is about making fine adjustments. For

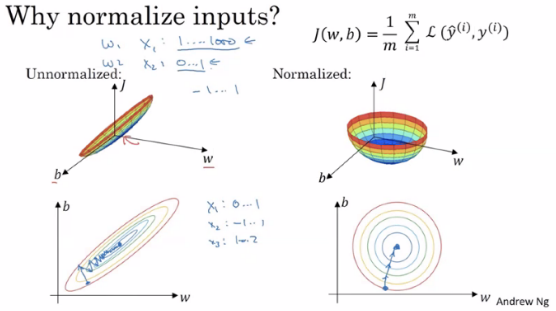
example, during transfer learning, you can unfreeze some of (or all) the pre-trained CNN latyers and let it adapt more to the task at hand.

## 3.5 Data preparation

With the hold-out cross validation, i have:

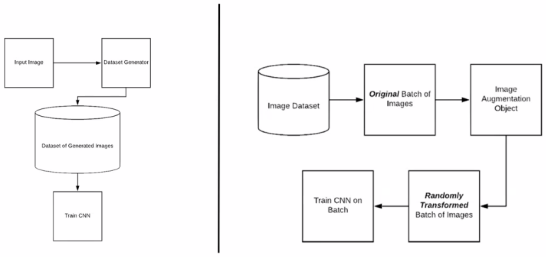
- 70% training set (development of the model)
- 20% validation set (tuning and selection of the model)
- 10% testing set (reporting of results)

This is often done, unless my data are small size. I can't build statistics on my results, if I have few testing samples.
The data needs to be normalized in order to obtain better results.




### 3.5.1 Data augmentation

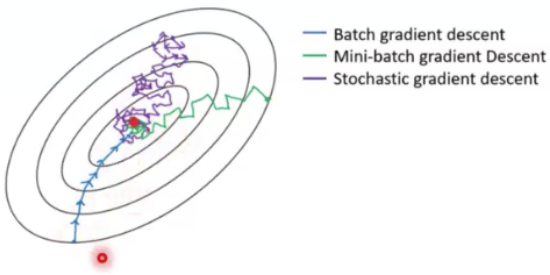
The aim is to increase artificially the dimension of the training set applying a series of transformations.



These transformations could be rotations, flips or a change of the brightness of the image itself. Be careful to consistency in the application of data augmentation techniques. I have to develop an algorithm for monitoring, for example, the movement of preterm infants. Suppose that the position and orientation of the room are fixed in relation to the child. A data augmentation that would be non sense to apply would be a rotation of the images.
Note that some bias of the original adding set persists.

## 3.6 Optimizers

**Tensor** A tensor is an N-dimensional array of data (non è vero :_( ).
**SGD (Stochastic gradient descent)** is an optimization algorithm used to train machine learning algorithms. The algorithm's task is to find a set of parameters within the model that minimizes the loss or error function. Optimization is a type of research precess and you can think of this research as learning. The optimization algorithm is called "gradient descent", where "gradient" refers to the calculation of an error gradient or slope of the error and "descent" refers to the movement along that slope towards a minimum level of error. The algorithm is iterative: **use the error to update the internal parameters of the model (back propagation)**.
A sample is individual data to which, in the case of supervised learning, a label is attached. This is used to compare the forecast and calculate an error. The batch size is a hyperparameter that defines the number of samples to be analyzed before updating the internal parameters of the model. A training dataset can be split into one or more batches. When the batch has sample size, the learning algorithm is called stochastic gradient descent. When the batch size is greater than a sample and less than the size of the training data set, the learning algorithm is called mini-batch gradient descent. The latter is the most common implementation used in the field of deep learning. Mini-batch gradient scent, the most common batch size are 32, 64 and 128.
In the batch gradient descent we use all the training data in a single iteration.




A training epoch means that the learning algorithm has made a pass through the training data

set.
**Momentum** is a technique used to help the optimizer go faster towards the optimal solution.

## 3.7 Loss function

For continuous outputs

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(t_i - p_i)^2$$

while for categorical output we'll implement accuracy (used only for evaluation) and cross entropy (used during training)

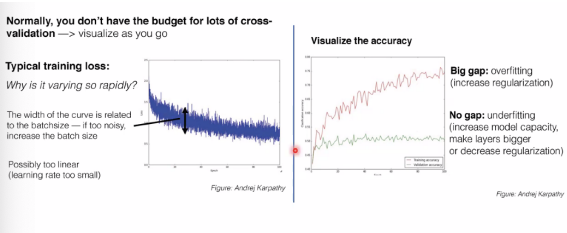
$$L_{CE} = -\sum_{i=1}^{c=2} t_i \log p_i = -t_1 \log(p_i) - (1 - t_1)\log(p_i).$$

The prediction is a probability vector: it represents the predicted probabilities of all classes with sum equal to 1. In a neural network, this prediction is usually made with the last layer activated by softmax function. We calculate the cross-entropy loss for the image. Loss is a measure of the performance of the model. The lower it is, the better. During learning, the model aims to achieve the lowest possible loss. The target represents the probability for all classes and is one-hot encoded vector, that is has 1 in a single position and 0 in all others. We'll start by calculating the loss for each class separately and then add it up. The loss for each class is calculated as follows:

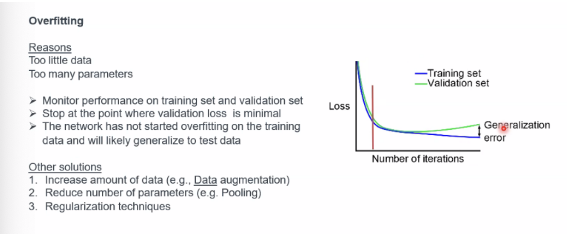
$$Loss = -p(X)\ln(q(X))$$

where the first term is the probability of class X in target while the second is the probability of class X in predictions. If the probability for the target is 0, the loss is 0.
The loss is 0 if the prediction is 1. The loss tends to infinity if the prediction is 0.
**Cross entropy** $= \sum_X p(X)\ln q(X)$. This is the cross entropy formula which can be used as a loss function for any two probability vectors. If we want to get the loss for our branch or for the whole set of data? You add up the losses of individual images. If our goal is one-hot encoded vector, we can actually forget the targets and predictions for all other classes and only calculate the loss for the hot class. Cross-entropy loss also works for distributions that are not one-hot vectors. In summary: **General formula used to calculate the loss between two probability vectors. The further we get from our target the more the error grows.**
**Binary classification** $= -[p(X)\ln q(X) + (1 - p(X))\ln(1 - q(X))]$ we use binary cross entropy, a specific case of cross entropy where our target is 0 or 1. It can be calculated with the cross entropy formula if you convert the target into a one-hot encoded vector, such as [0,1] or [1,0] and forecasts respectively. We can calculate it with the previous formula.
**Multi-label classification** $= \sum_x \text{Binary cross entropy}_x$ Our target can represent multiple classes (or even zero) at the same time. We calculate the binary cross entropy for each class separately

and the add it up to get the complete loss.




**Loss function- learning curves**




## 3.8 Metrics

The **F1 score** is a popular metric for evaluating the performance of a classification model. In the case of multi-class classification, for the calculation, for example, of the F1 score, averaging methods are used, which translate into a series of different averages (macro, weighted, micro) in the rating report.

$$\textbf{F1 score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

In order to assess the performance of a model in a comprehensive way, we should examine both recall and accuracy. The F1 is a useful metric that considers both.
Instead of having multiple F1 scores per class, it would be better to average for a single number that describes overall performance.
The weighted average F1 score takes into account the support of each class. The weight refers to the proportion of the support for each class in relation to the sum of all the support values.
Micro average F1 score calculates the global mean F1 score by counting the sums if the true positives, false negatives and false positives:

$$\textbf{Micro-average F1 score} = \frac{TP}{TP + 0.5(FP + FN)}.$$

The micro-average F1 score calculates the percentage of correctly ranked observations on all observations.

# 4 AI ETHICS: ethics of artificial intelligence -5/04/2025

This philosophical discipline discusses:

- AI impact on society,
- Values and principles of AI,
- Guide lines to ethically design AI systems.

**Ethics for AI**

- Foundation of ethics of AI
  - Unpacking moral concepts → diverse dimensions = diverse risks
  - Unveiling the values at the core of the ethical principles → **FRIA (EU AI Act)**
- Avoid:
  - Ethical principles as labels → checklist
  - Phenomena:
    - Ethics washing (and fake ethical self-regulation)
    - Ethics bashing (trivialization)
  - Criticism: ethics toothless and uselessness
  - Harm and reputation risk
- Ensure:
  - Adequate risk-based assessment (detection and management)
  - Fundamental rights and values impact assessment → **FRIA (EU AI Act)**
  - Develop actionable ethics by design criteria

**Benchmarking AI ethics frameworks and principles**
+200 ethical frameworks & guidelines:

- EU HLEGAI (2019)
- WHO (2021)
- UNESCO (2021)

**Benchmarking AI ethics frameworks and principles**

- Benevolence → benefit people/ minimize harm

- Autonomy → AI systems that respect/promote human autonomy
- Justice as fairness → fair AI systems, which do not discriminate
- Explicability → AI systems whose functioning and outputs are explicable
- Accountability & Responsability → Determine accountability mechanisms and harm redress
- Sustainability → (human and environmental) sustainability

**Explicability = understand how AIs achieve a certain goal**

- Black Box (Pasquale 2015)
- Transparency(?)
- Explicability ≠ Transparency
- Transparency = Accessibility
- Explicability = Intelligibility
- EXAI

**Accountability & Responsibility**

- Multi-agent-context = designers, company providers, users - who is responsible if something goes wrong?
- **Accountability mechanisms:**
  - Auditability:
    - Risk based assessment
    - Develop ex ante risk scenarios and guidelines for responsibility distribution (shared responsibility)
  - Harm redress:
    - Develop harm redress mechanisms (people affected negatively should be redressed)

**(environmental and human) Sustainability**

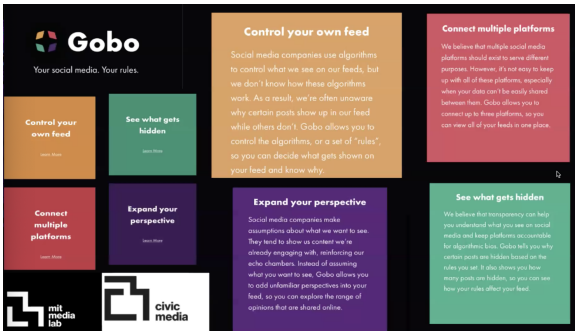
- Environmental sustainability:
  - Computational power for training models + water for cooling data centers, etc.
  - Carbon footprint → assess the environmental impact
  - Do we adopt technology that enable me to do the same task, but in a more sustainable way?

- Human sustainability
  - Impact of AI on human resources
  - Does this product affect employment? Does it create new or eliminate jobs? of whom? what kind of work? of whom?
  - Sustainability a 360° = impact assessment before implementing AI

**Human Autonomy as an ethical value**

- Respect the individuals as decision-makers
- To choose and act according to their own beliefs, interests, preferences
- Freedom of choice/ human agency
- Necessary conditions:
  - availability of alternative options
  - autonomy as rational deliberation or endorsement of what we choose

Algorithmic governance → pervasive SNS, IoT, etc.
Algorithms = information gatekeepers → choose what information of any kind we can see, what to show us, etc.
**Algorithmic choice architecture** = they decide what information (contact, opportunity, etc.) we can access to and influence our choices and actions.
**Our choice options are always reduced**




Autonomy as rational deliberation or endorsement of what we choose.

- Recommender system of highly personalized informational content
- Chosen on the basis of how we respond to the presentation of the sensitive information inferred/discovered on us (profiling: content-based or collaborative based)

- ML can infer choice-driving/eliciting elements or emotionally loaded information
- Use them to nudge our behavior toward pre-defined goals
- Rational deliberation bypassing = our choices/actions/behavior risk to be pre-determined by third parties

**ML and DL profiling ad personalization should be ethical:**

- We need them to navigate informational overload
- Limited time and cognitive resources
- Based on accurate profile of ourselves, disclosable to the user (do not use people as means: no exploitation of sensitive information).
- **Automation bias**
- Bias by proxy

**Problematic: increasing delegation to such systems tasks and decisions → biased/discriminate**
Technical solutions:

- Avoid/remove protected attributes
- Cases where protected characteristics are crucial to make fair decisions

**Parity models**, ensuring the measures of its predictive performance are equal across protected groups = algorithm performance is equal/ treatment is same for diverse groups.
Formal/procedural fairness (ensured by the law): formal equality of opportunity.

**Fairness as an ethical value**

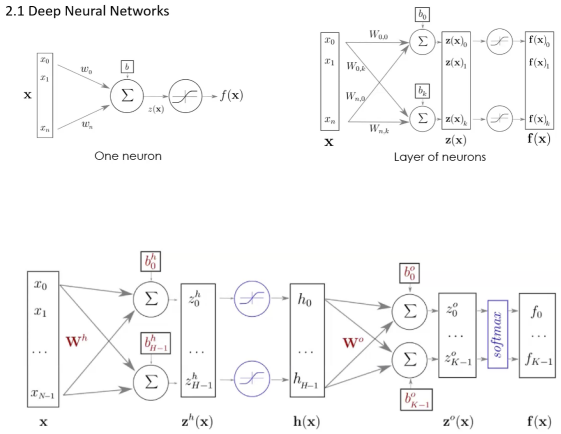
**Substantial fairness:** compensate inequalities

- Using AI to investigate sources of unfair bias in socio-relational conditions
- Design systems capable to compensate them
- Ethics by design (equality of opportunity as a criterion)

# 5 Deep generative models -11/04/2025

The aim is to understand the structure of DGM, use them, get a general knowledge of model training and as well of fine-tune issues.
In generative models data-driven methods are needed. A first example can be distribution modeling: **It's possible to generate new data given a specific distribution**. The goal is to take as input training samples from some distribution and learn a model that represents that distribution.

## 5.1 Preliminary concepts




As we saw we implement a loss function as

$$l(f(x^s;\theta),y^s) = nll(x^s,y^s;\theta) = -\log f(x^s;\theta)_{y^s}$$

and the training consists in finding the parameters (Wh, bh, W0, b0) which minimize negative log likelihood. We also saw how we can implement algorithms both supervised and unsupervised.

### 5.1.1 PDF and joint probability

Given $f(x) = \frac{dF(x)}{ddx}$, where $F(x) = P(a \leq x \leq b) = \int_a^b f(x)dx$, we have that f(x) is the PDF and F(x) is the CDF. While the PDF is a function of a single variable, the joint probability is a function of two or more variables. What we have for the joint probability is:

$$P(X_1 \in [a_1,b_1],...,X_k \in [a_k,b_k]) = \int_{a_1}^{b_1} ... \int_{a_k}^{b_k} f_X(x_1,...,x_k)dx_k...dx_1$$

### 5.1.2 Maximum likelihood estimation (MLE) & Maximum a Posteriori Estimation (MAP)

The PDF is our model, the question is: how do we estimate the parameters? If the PDF would be Gaussian we would use mean, std, etc. Otherwise we can implement MLE and MAP which are both methods able to estimate the parameters $\theta$ of a statistical model based ob observed data $\mathbf{X}$.

**MLE** For $\theta$ that **maximize** the likelihood $p(X|\theta)$:

$$\hat{\theta}_{MLE} = \arg\max_{\theta} p(X|\theta).$$

Are we making any hypothesis on the priori distribution on $\theta$? The answer is no. What we have is

$$\theta_{MLE} = \arg\max_{\theta} p(X|\theta) = \arg\max_{\theta} \prod_i p(x_i|\theta).$$

or also, using the logarithm

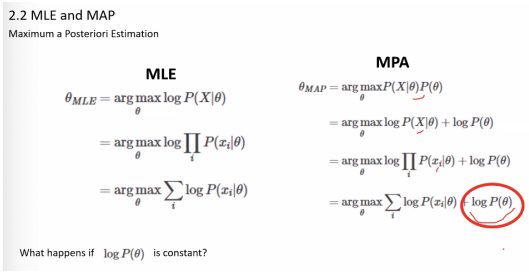
$$\theta_{MLE} = arg\max_{\theta} \log P(X|\theta) = \arg\max_{\theta} \log \prod_i P(x_i|\theta) = \arg\max_{\theta} \sum_i \log P(x_i|\theta)$$

Now we maximize w.r.t. $\theta$ with our "favorite" optimization algorithm (Gradient descent, PSO, Grid search, etc...).

**MAP** From a bayesian perspective, includes a prior probability $p(\theta)$ that reflects our belief about parameter values.

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)} \propto P(X|\theta)P(\theta).$$

We compare these two methods:



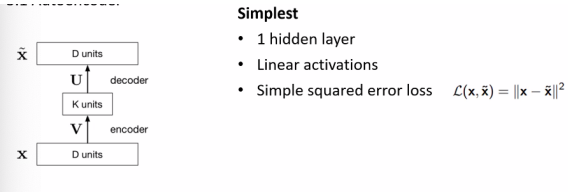

The logarithm $\log P(\theta)$ gives a constant contribution while $P(\theta)$ is constant. That means that MPA coincides with MLE.
In general MLE is simple to implement and is more prone to overfitting if there is limited data

and no regularization/prior is used. However, MAP can help reduce overfitting by imposing a prior (often act as a regularization term) and is useful when we have domain knowledge or want to encode certain parameter restrictions.
To calculate P(Y|X), the first estimate the prior probability P(Y) and the likelihood probability P(Y|X) from the data provided.

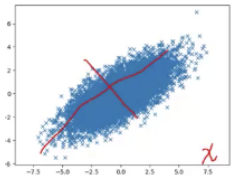
$$posterior = \frac{prior \times likelihood}{evidence} \rightarrow P(Y|X) = \frac{P(Y)P(X|Y)}{P(X)}.$$

## 5.2 Variational AutoEncoders (VAE)

An autoencoder is feed-forward neural net whose job is to take an input x and predict x. To make this non-trivial, we need to add a bottleneck layer whose dimension is much smaller than the input.




The simplest autoencoder is composed only of 1 layer, has liner activations and the loss function is simply $L(x, \tilde{x}) = ||x - \tilde{x}||^2$. This network computer $\tilde{x} = UV_x$, which is a linear function. If $k \geq D$, see can choose U and V such that UV is the identity. This isn't very interesting.
So a linear autoencoder with squared error loss is equal to the PCA (principal component analysis). Sample covariance of data → minimize covariance → diagonal matrix → eigenvalues.



The **Principal Component Analysis (PCA)** is a statistical technique used to reduce the dimensionality of a dataset while preserving most of the original variance. PCA finds a linear representation of the data in a lower-dimensional space, where the new variables (the *principal components*) are linear combinations of the original variables. The goal of PCA is to find the principal components that maximize the variance of the data along each axis, in order to obtain a more compact representation of the data.

Mathematically, the PCA process can be formulated as follows:

1. **Centering the data**: Subtract the mean of each variable from the data:

$$X' = X - \mu$$

where $X \in \mathbb{R}^{n \times d}$ is the data matrix (with $n$ observations and $d$ variables), and $\mu$ is the mean vector of each column of $X$.

2. **Calculating the covariance matrix**:

$$S = \frac{1}{n} X'^T X'$$

where $S \in \mathbb{R}^{d \times d}$ is the covariance matrix of the centered data.

3. **Eigenvectors and eigenvalues**: Solve the eigenvalue problem to obtain the **principal components** (eigenvectors) and the variance (eigenvalues):

$$Sw = \lambda w$$

where $w$ is an eigenvector and $\lambda$ is the associated eigenvalue.

4. **Projection onto the first $k$ principal components**: The original data is projected onto the principal component space, reducing the dimensionality from $d$ to $k$:

$$Z = X' W_k$$

where $W_k \in \mathbb{R}^{d \times k}$ is the matrix containing the first $k$ eigenvectors (principal components), and $Z \in \mathbb{R}^{n \times k}$ is the matrix of the projected data.

Main limitation: autoencoders and deep autoencoders do not define a distribution → they do not generate anything (they reproduce).

The encoder in generative models is the component that maps the data into a latent space, which becomes the foundation for generating new instances of data through the decoder or the generator. In models like the VAE, the encoder not only reduces the dimensionality but also transforms the input into a probability distribution that allows generating new data consistent with the original distribution.

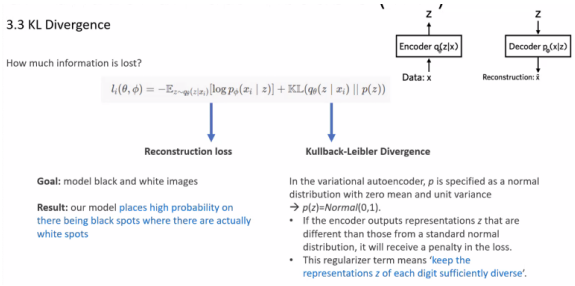
The loss function is a negative log-likelihood with a regularization term (for each $i^{th}$ sample). In the formula of the log-likelihood, E is the expectation taken w.r.t. to the encoder's distribution over the representations → encourages to learn the data.

- **Encoder:**
  - Input x, hidden output z, recostruction x
  - Encoder is NN with weights/biases $\theta$
  - Input 28 x 28 pixel photo → 784- dimensional representation
  - Hidden size len(z) is much less than 784

- Efficient compression with a probability density $q_\theta(z|x)$
- From this distribution, you can sample noisy values of z!

- **Decoder:**
  - Input x, hidden output z, reconstruction x
  - Decoder is NN with weights/bias $\Phi$
  - Its input is z and has decoding function $p_\Phi(x|z)$

In the example we were following before, photo are black and white and represent each pixel as 0 or 1. The probability distribution of a single pixel can be then represented using a PDF (i.e., Bernoulli distribution in this case). The decoder gets as input the latent representation of a digit z and outputs 784 Bernoulli Parameters (one for each of 784 pixels in the image).




We now move to the **training** part: that is done using backpropagation.

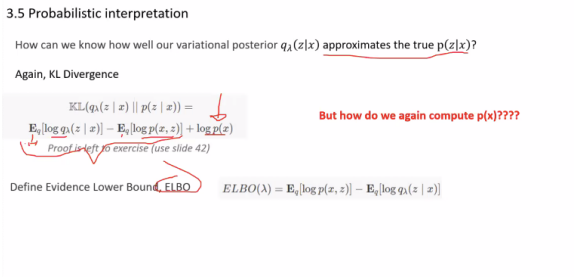
**Probabilistic interpretation** A variational autoencoder contains a specific probability model of a data x and latent variables z. We can write the joint probability of the model as:

$$P(x, z) = P(x|z)P(z)$$

The generative process:
for each datapoint i

- Draw $z_i \sim p(z) \rightarrow$ draw from a prior
- Draw $x_i \sim p(x|z) \rightarrow$ draw form a probability (Bernoulli distributed for b/w images)

The aim is to infer good values of z given z, or to calculate P(z|x). To calculate the latter we need P(x), but how can I get this probability? That is the prior probability of the data. We can marginalize the latent variables: $P(x) = \int P(x|z)P(z)dz$. However this integral takes time. We know that $q_\lambda(z|x)$ (from the encoder) has a parameter $\lambda$ which indexes the family distribution. If q were gaussian it would be mean and variance of each data point $\lambda_i = (\mu_{x_i}, \sigma_{x_i})$. We call from now on q: Variational Posterior.



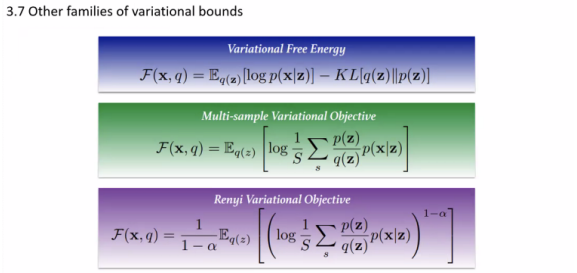

$$\log P(x) = ELBO(\lambda) = KL(q_\lambda(z|x)||p(z|x)).$$

So if we know that $KL \geq 0$, instead that minimizing KL we can maximize ELBO $\rightarrow$ avoided P(x). To sum up, we recall that

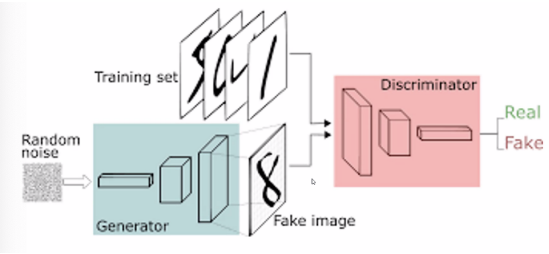
$$ELBO_i(\lambda) = Eq_\lambda(z|x_i)[\log p(x_i|z)] - KL(q_\lambda(z|x_i)||p(z)).$$

Let's remember now NN $\rightarrow$ the final step is:

- Parametrize the approximate posterior $q_\theta(z|x, \lambda)$ with an inference network (encoder)
- Parametrize the likelihood p(x|z) with a generative network (decoder) that takes latent variables z and outputs parameters to the data distribution $p_\theta(x|z)$
- $ELBO_i(\theta, \phi) = -l_i(\theta, \phi)$
- The inference and generative networks have model parameters $(\theta, \phi)$, while $\lambda$ represents variational parameters



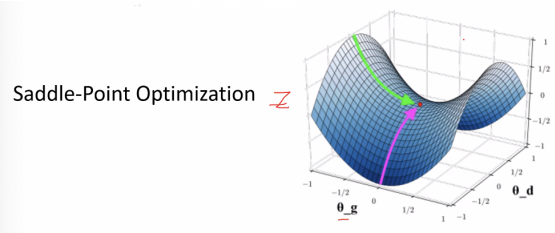
## 5.3 Generative Adversarial Networks (GAN)



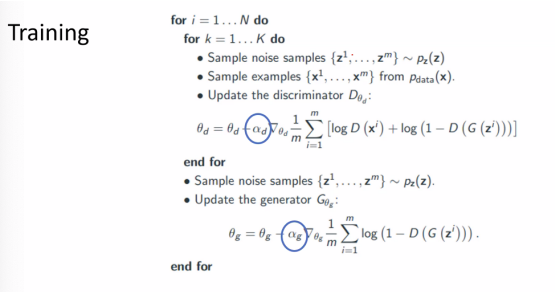

We start by generating a batch of samples (GAN) and then we update the weights based on the results.
Let z and p(z) be a simple base distribution and its probability. The generator G(z) is a deep neural network. D is the manifold and the discriminator D(x) is also a deep neural network. The parameters optimization is given by:

$$\min_{\theta_d} \max_{\theta_d} \underbrace{E_{p_{data}}[\log D_{\theta_d}(x)]}_{\text{likelihood of true data}} + \underbrace{E_{p_z(z)}[\log(1 - D_{\theta_d}(G_{\theta_d}))]}_{\text{likelihood of generated data}}$$

where the first term is the likelihood of true data and the second term is the likelihood of generated data.




An example of training is given:




For vanishing gradient, if D is close to perfect, min-max saturates →

- possibilities for non-saturating objectives;
- restrict capability of D (structurally);
- Balance learning $D_{\theta_d}$ and $G_{v\theta_d}$.

The **mode-collapse** happens when the generator focuses on producing a limited set of data patterns that deceive the discriminator. What happens is that it becomes fixated on a few dominant modes in the training data and fails to capture the full diversity of the data distribution.
**Wasserstein GANs** $\rightarrow$ use Wasserstein distance instead than cross-entropy. The Wasserstein-1 distance (or Earth-Mover's distance) is continuous and differentiable almost everywhere with respect to the generator's parameters. Even when generated data has little overlap with the real distribution $\rightarrow$ W-distance provides non-zero gradients $\rightarrow$ Always feedback $\rightarrow$ Steers away from vanishing gradients and thus form $\rightarrow$ mode collapse.

The Wasserstein-1 distance between two probability distributions $P_r$ (real data) and $P_g$ (generated data) is defined as:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

Here, $\Pi(P_r, P_g)$ represents the set of all joint distributions $\gamma(x, y)$ whose marginals are $P_r$ and $P_g$, respectively. Intuitively, $W(P_r, P_g)$ measures the minimum "cost" of transporting mass to change one distribution into the other.

**Kantorovich-Rubinstein Duality**

Directly computing the infimum in the above formulation is intractable in high dimensions. The Kantorovich-Rubinstein duality offers an alternative representation:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \left(\mathbb{E}_{x\sim P_r}[f(x)] - \mathbb{E}_{x\sim P_g}[f(x)]\right)$$

In this formulation:

- $f$ is any function that is **1-Lipschitz** (i.e., $|f(x_1) - f(x_2)| \leq |x_1 - x_2|$ for all $x_1, x_2$).
- The supremum is taken over all such functions $f$.

This dual form transforms the original problem into finding a function $f$ (often referred to as the *critic* in the GAN framework) that maximizes the difference in expectations between the real and generated distributions, under the constraint that $f$ remains 1-Lipschitz.

A critical challenge in implementing WGANs is ensuring that the critic remains 1-Lip. Two common methods are used:

- Weight clipping: Constraining the weights of the critic network to lie within a compact space. This is a crude method that can sometimes limit the capacity of the critic.
- Gradient Penalty (WGAN-GP): A more robust technique where an additional penalty term is added to the critic's loss function:

$$\lambda E_{\tilde{x}\sim P_{\hat{x}}}[(||\nabla_{\hat{x}} f(\hat{x})||_2 - 1)^2]$$

Here, $\hat{x}$ is sampled from a distribution defined along straight lines between real and generated samples, and $\lambda$ is a hyperparameter controlling the strenght of the penalty.

**Non-convergence** $D_{\theta_d}$ and $G_{\theta_d}$ are a saddle, that gives a more common non-convergence → saddle is complex to find. Training updates indeed undo each other. To solve this problem one can use:

- Soft and Noisy labels
- DCGAN/Hybrid models
- Stability tricks from RL
- ADAM optimizer

## 5.4 Diffusion Models (DM)

Based of the diffusion process: two steps (forward & backward). Add gradually to $X_0$ small amounts of Gaussian noise. Proceed as in Markov Chain (t+1 depends only on t).

$$q(x_t|x_t - 1) = N(x_t; \sqrt{(1-\beta_t)}x_{\{t-1\}}, \beta_t I)$$

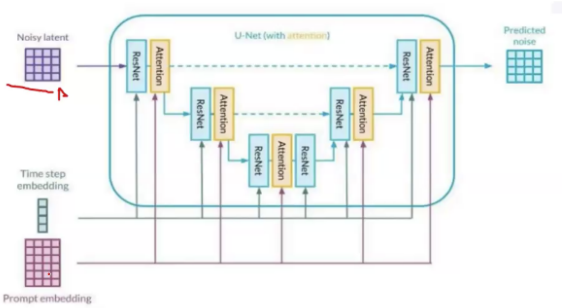
with N normal distribution and $\beta$ the noise schedule (i.e., how much noise is added).
So if N is a normal distribution, our objective is to predict mean and variance to predict the reverse process (invert forward) → generate data.

$$\text{Reverse} \rightarrow p$$

$$p(x_{t-1}|x_t) = N(x_{t-1}; \mu_\theta, (x_t, t), \Sigma_\theta(x_t, t)).$$

**We need to learn this distribution, how?**

- **Old attempts:** with math (Denoising Probabilistic Models)
- **New attempts:** with Unets (Den. Diffusion Prob. Models)




**Training:** Minimizing Variational Lower Bound (As VAE...)

$$E[-\log p_\theta(x_\theta)] \leq Eq[-\log \frac{p_{\theta(x_{0:T})}}{q(x_{1:T}|x_0)}]$$

The problem is: we have t/t-1 process, how much time will it take?
**Check out:** ControlNet