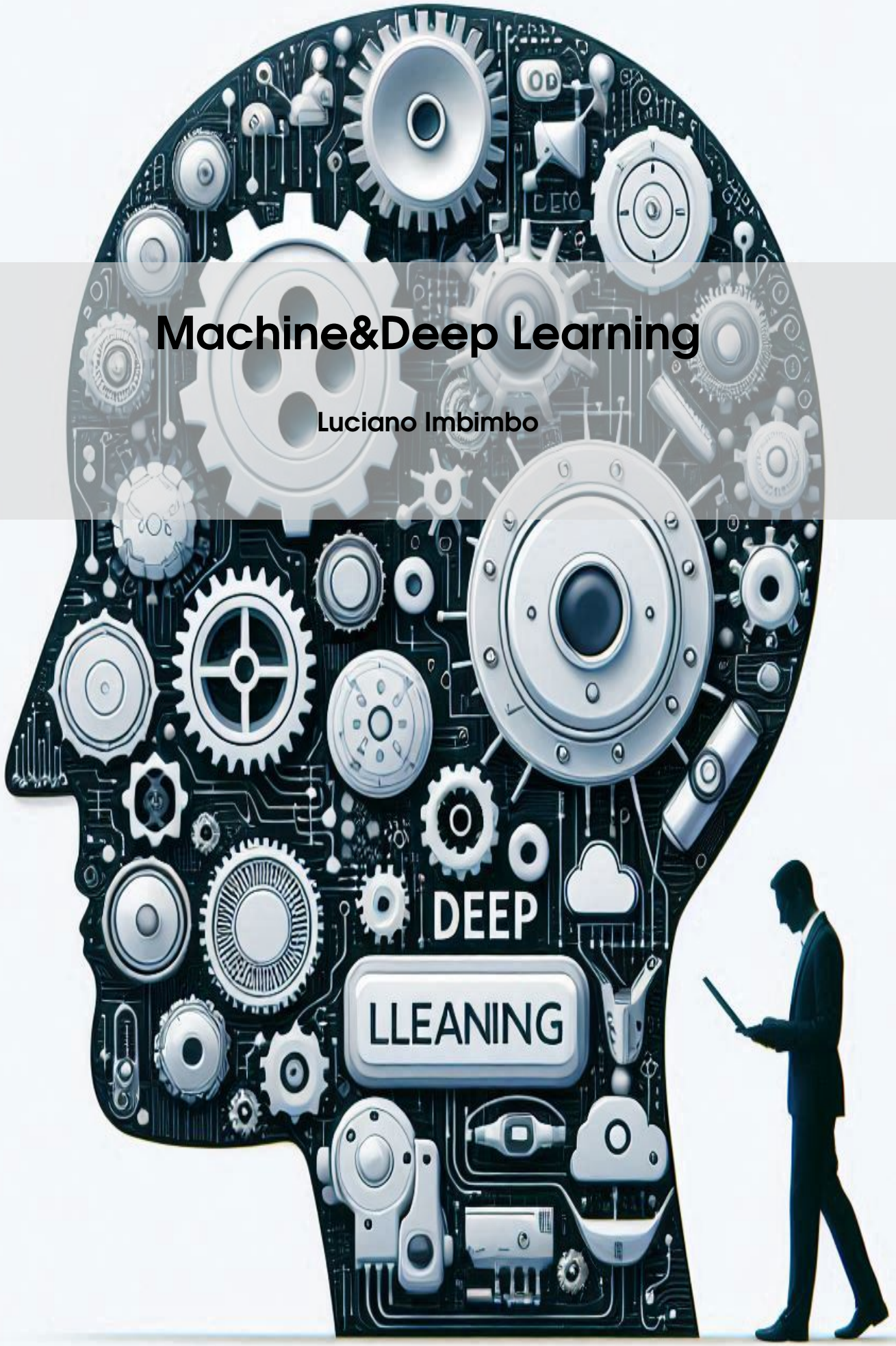
Machine&Deep Learning

Luciano Imbimbo



PUBLISHED BY RAFAEL BRITO GOMES

BOOK-WEBSITE.COM

Contents

Prerequisites

1. Linear Algebra

Linear algebra [3] is a branch of mathematics that is widely used throughout science and engineering. Therefore a good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms.

1.1 Vector and Matrix

The study of linear algebra involves several types of mathematical objects:

Definition 1.1.1 — Scalar. A scalar is just a single number.

Definition 1.1.2 — Vector. A vector is an array of numbers, where we can identify each individual number by its index in that ordering.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{1.1}$$

Definition 1.1.3 — Vector Space. The real vector space $\mathbb{R}^n$ is a set of vectors $\mathbf{x} = [x_1, \ldots, x_n]$ that satisfy the following properties:

- **Addition**: If $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$, then $x + y = [x_1 + y_1, \ldots, x_n + y_n] \in \mathbb{R}^n$
- **Scalar Product**: If $x \in \mathbb{R}^n$ and $a \in \mathbb{R}$, then $ax = [ax_1, \ldots, ax_n] \in \mathbb{R}^n$
- **Inner Product**: If $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$, then $xy = \sum_{i=1}^{n} x_i * y_i$

Definition 1.1.4 — Matrix. A matrix $\mathbf{X} \in \mathbb{R}^{nxm}$ is a rectangular array of elements $x_{ij} \in \mathbb{R}$,

$1 \leq i \leq n, 1 \leq j \leq m.$

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix} \tag{1.2}$$

1.2 Operations on matrix

A matrix $\mathbf{X} \in \mathbb{R}^{nxm}$ supports all the operations defined in real vector space.

Corollary 1.2.1 — Add matrices. We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements:

$$C = A + B \rightarrow C_{i,j} = A_{i,j} + B_{i,j} \tag{1.3}$$

Corollary 1.2.2 — Scalar Product. We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix:

$$D = aB + c \rightarrow D_{i,j} = a + B_{i,j} + c \tag{1.4}$$

Definition 1.2.1 — Transpose of a matrix. Another important operation on matrices is the **transpose**, that is the mirror image of the matrix across the main diagonal.

$$(A^T)_{i,j} = A_{j,i} \tag{1.5}$$

Definition 1.2.2 — Matrix product. We can multiply two matrices between them, this product is called **matrix** product and the result is a third matrix C. In order for this product to be defined, A must have the same number of columns as B has rows. If A is of shape m x n and B is of shape n x p , then C is of shape m x p:

$$C = AB \rightarrow C_{i,j} = \sum_k A_{i,k} B_{k,j} \tag{1.6}$$

In the context of deep learning, we also use the following conventional notation $C = A + b = A_{i,j} + b_j$ where the vector b is added to each row of the matrix. This shorthand eliminates the need to define a matrix with b copied into each row before doing the addition. This implicit copying of b to many locations is called **broadcasting**

1.3 Identity and Inverse Matrices

To describe matrix inversion, we first need to define the concept of an identity matrix.

Definition 1.3.1 — Identity matrix. An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix

$$\forall x \in \mathbb{R}^n, I_n x = x \tag{1.7}$$

The structure of the identity matrix is simple: all of the entries along the main diagonal are 1, while all of the other entries are zero.

Definition 1.3.2 — Inverse matrix. The **matrix** inverse of A is denoted as A^{-1}, and it is defined as the matrix such that

$$A^{-1}A = I_n \tag{1.8}$$

1.4 Norms

Sometimes we need to measure the size of a vector and to do that, in machine learning, we usually using a function called a **norm**.

Definition 1.4.1 — Norm. Formally, the L^p norm is given by (for $p \in \mathbb{R}, p \geq 1$)

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}} \tag{1.9}$$

Corollary 1.4.1 — Formal definition of norm. On an intuitive level, the norm of a vector x measures the distance from the origin to the point x; rigorously, a norm is any function f that satisfies the following properties

- $f(x) = 0 \rightarrow x = 0$
- $f(x+y) \leq f(x) + f(y)$ (**triangle inequality**)
- $\forall \alpha \in \mathbb{R}, f(\alpha x) = |\alpha| f(x)$

The L^2 **norm**, with p = 2, is known as the Euclidean norm. It is simply the Euclidean distance from the origin to the point identified by x and it is so frequently in machine learning that it is often denoted simply as $||x||$, with 2 the subscript omitted.
It is also common to measure the size of a vector using the **squared** L^2 **norm**, which can be calculated simply as $x^T x$. The squared L^2 norm is more convenient to work with mathematically and computationally than the L^2 norm itself.
In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases we use the L^1 **norm** because every time an element of x moves away from 0 by ε, the L^1 norm increases by ε.

1.5 Special Kinds of Matrices

Some special kinds of matrices and vectors are particularly useful.

Definition 1.5.1 — Diagonal matrix. Diagonal matrix consist mostly of zeros and have non-zero entries only along the main diagonal.

$$D_{i,j} = 0, \forall i \neq j \tag{1.10}$$

Definition 1.5.2 — Symmetric matrix. A symmetric matrix is any matrix that is equal to its own transpose

$$A = A^T \tag{1.11}$$

Definition 1.5.3 — Orthogonal matrix. An orthogonal matrix is a square matrix whose rows are mutually orthonormal (orthogonal + unit norm) and whose columns are mutually orthonormal.

$$A^T A = AA^T = I \tag{1.12}$$

1.6 Eigendecomposition

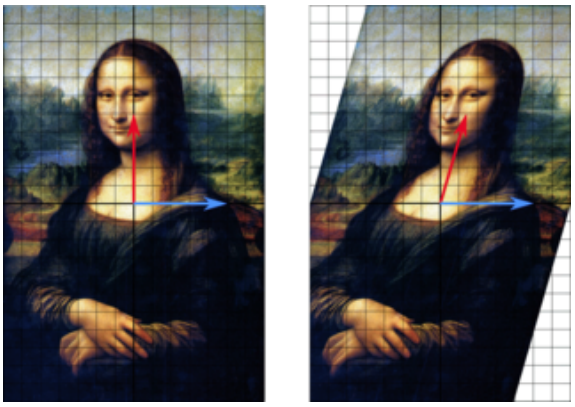
Many mathematical objects can be understood better by breaking them into constituent parts, or finding some properties of them that are universal, not caused by the way we choose to represent them. One of the most widely used kinds of matrix decomposition is called eigendecomposition, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

Definition 1.6.1 — Eigenvector. An **eigenvector** of a square matrix A is a non-zero vector v such that multiplication by alters only the scale of:

$$Av = \lambda v \tag{1.13}$$

The scalar λ is known as the **eigenvalue** corresponding to this eigenvector.

Therefore, an eigenvector of a matrix A is a vector x that is transformed by A into another vector, substantially equal to less than a scalar λ, called an eigenvalue. The example here, based on the Mona Lisa, provides a simple illustration. Each point on the painting can be represented as a vector pointing from the center of the painting to that point. The linear transformation in this example is called a shear mapping. Points in the top half are moved to the right, and points in the bottom half are moved to the left, proportional to how far they are from the horizontal axis that goes through the middle of the painting. The vectors pointing to each point in the original image are therefore tilted right or left, and made longer or shorter by the transformation. Points along the horizontal axis do not move at all when this transformation is applied. Therefore, any vector that points directly to the right or left with no vertical component is an eigenvector of this transformation, because the mapping does not change its direction. Moreover, these eigenvectors all have an eigenvalue equal to one, because the mapping does not change their length either.



1.7 Trace Operator

The trace operator is useful for a variety of reasons. Some operations that are difficult to specify without resorting to summation notation can be specified using matrix products and the trace operator.

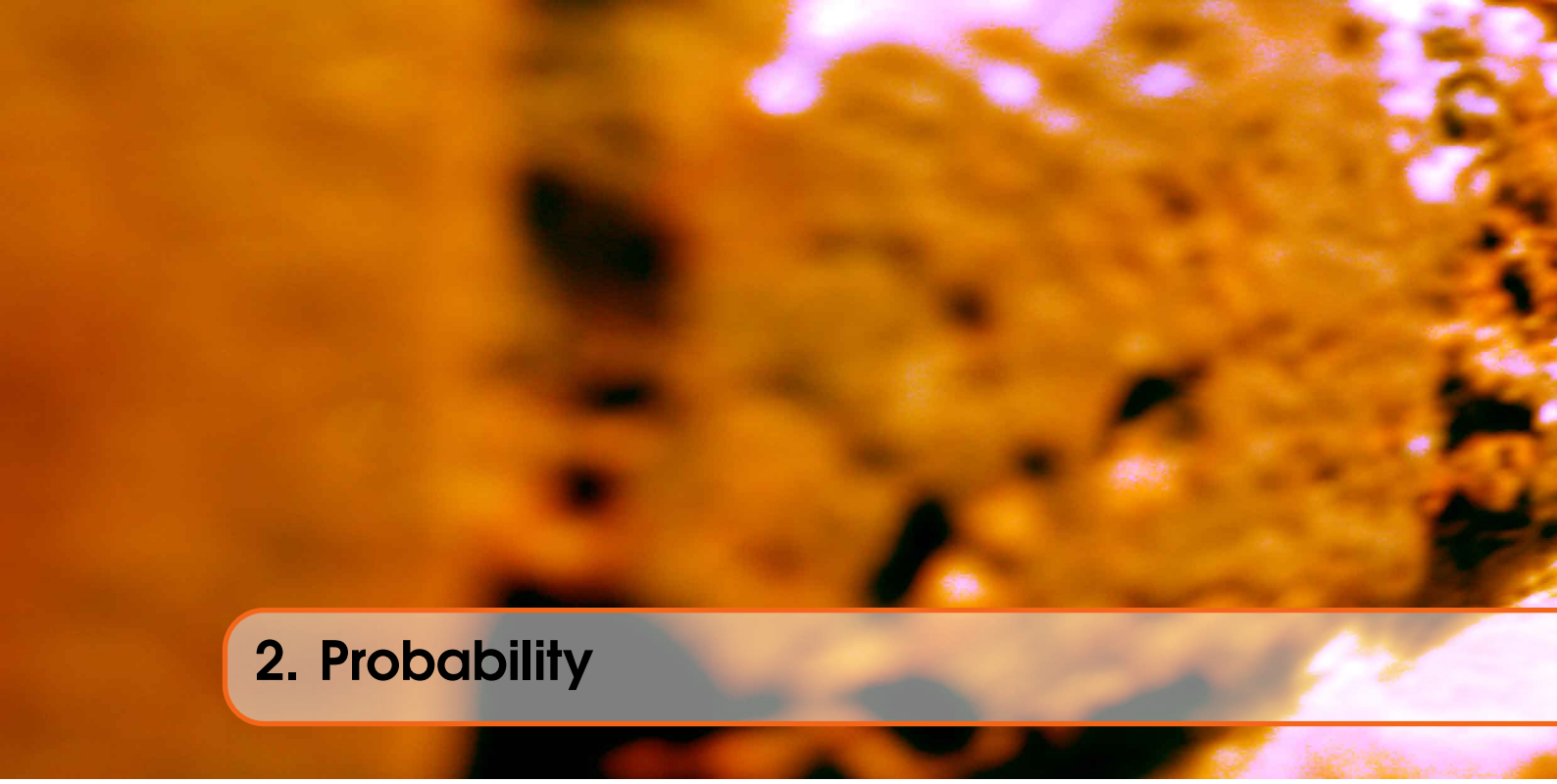
Definition 1.7.1 — Trace Operator. The trace operator gives the sum of all of the diagonal entries of a matrix

$$Tr(A) = \sum_i A_{i,i} \tag{1.14}$$

1.8 Determinant

The determinant of a square matrix, denoted det(A), is a function mapping matrices to real scalars. The determinant is equal to the product of all the eigenvalues of the matrix. The absolute value of the determinant can be thought of as a measure of how much multiplication by the matrix expands or contracts space.

- If the determinant is 0, then space is contracted completely along at least one dimension, causing it to lose all of its volume.
- If the determinant is 1, then the transformation preserves volume.



2. Probability

Probability theory is a mathematical framework for representing uncertain statements. It provides a means of quantifying uncertainty and axioms for deriving new uncertain statements. In artificial intelligence applications, we use probability theory in two major ways. First, the laws of probability tell us how AI systems should reason, so we design our algorithms to compute or approximate various expressions derived using probability theory. Second, we can use probability and statistics to theoretically analyze the behavior of proposed AI systems.

2.1 Random Variables

In many situations, it is desirable to assign a numerical value to each outcome of an experiment. Such an assignment is called a random variable.

Definition 2.1.1 — Random variable. A random variable X is defined by a function $f(X)$ that maps each element w of the sample space Ω to a value $f_X(w)$ in a set X called the range of the random variable.
For each $x \in X$ the event $X = x$ refers to the subset of the sample space $\{w|w \in \Omega, f_X(w) = x\}$.
For each $x \in X$ the probability $P(X = x) = P(\{w|w \in \Omega, f_X(w) = x\})$

Definition 2.1.2 — Discrete random variables. A random variable is discrete if its possible values form a **discrete** set. This means that if the possible values are arranged in order, there is a gap between each value and the next one. The set of possible values may be infinite; for example, the set of all integers and the set of all positive integers are both discrete sets.

Definition 2.1.3 — Continuous random variables. A random variable is **continuous** if its possible values are all the points between some two numbers (interval).

2.2 Probability Distributions

A probability distribution is a description of how likely a random variable or set of random variables is to take on each of its possible states.

Definition 2.2.1 — Probability Distribution. A probability distribution P over a sample space Ω is a mapping from subsets of Ω to the real numbers that satisfies the following conditions:

- **Non-negativity**: $P(\alpha) \geq 0, \forall \alpha \subseteq \Omega$
- **Normalization**: $P(\Omega) = 1$
- **Additivity**: $\forall \alpha, \beta \subseteq \Omega$ that are disjoint sets, $P(\alpha \bigcup \beta) = P(\alpha) + P(\beta)$

2.2.1 Discrete Variables and Probability Mass Functions

A probability distribution over discrete variables may be described using a **probability mass function (PMF)**, that typically is denote with a capital P.

Definition 2.2.2 — Discrete PMF. The probability mass function maps from a state of a random variable to the probability of that random variable taking on that state.

$$P(X = x) = P(x) \geq 0, \forall x \in X \ \ \& \ \ \sum_{x \in X} P(X = x) = 1 \tag{2.1}$$

With a probability of 1 indicating that x = x is certain and a probability of 0 indicating that x = x is impossible.
Probability mass functions can act on many variables at the same time. Such a probability distribution over many variables is known as a joint probability distribution.

Corollary 2.2.1 — Joint probability distribution. A joint probability distribution is a probability distribution defined over a collection of random variables $(X_1, ..., X_m)$ with ranges $X_1, ..., X_m$: $P(X_1 = x_1, \ldots, X_m = x_m)$ and denotes that $X_1 = x_1, \ldots, X_m = x_m$ simultaneously.

2.2.2 Continuous Variables and Probability Density Functions

When working with continuous random variables, we describe probability distributions using a probability density function (PDF) rather than a probability mass function.

Definition 2.2.3 — Continuous PDF. A probability density function (PDF), is a function p that satisfy the following properties.

$$p(X = x) \geq 0, \forall x \in X \ \ \& \ \ \int_X p(X = x)dx = 1 \tag{2.2}$$

Specifically, the probability that x lies in some set S is given by the integral of $p(x)$ over that set. In the univariate example, the probability that x lies in the interval $[a, b]$ is given $\int_{[a,b]} p(x)dx$

2.3 Marginal Probability

Sometimes we know the probability distribution over a set of variables and we want to know the probability distribution over just a subset of them.

Definition 2.3.1 — Marginal probability. The probability distribution over the subset is known as the marginal probability distribution.

$$\forall \in x, P(X = x) = \sum_y P(X = x, Y = y) \tag{2.3}$$

$$p(x) = \int p(x, y)dy \tag{2.4}$$

2.4 Conditional Probability

In many cases, we are interested in the probability of some event, given that some other event has happened. This is called a conditional probability.

Definition 2.4.1 — Conditional probability. We denote the conditional probability that $Y = y$ given $X = x$ as $P(Y = y|X = x)$. This conditional probability can be computed with the following formula

$$P(Y = y|X = x) = \frac{P(X = x, Y = y)}{P(X = x)} \tag{2.5}$$

2.5 Expectation, Variance and Covariance

Definition 2.5.1 — Expectation. The expectation or expected value of some function $f(x)$ with respect to a probability distribution $P(x)$ is the average or mean value that f takes on when x is drawn from P (it's not the most probable value but the most expected value).

$$\mathbb{E}[f(x)] = \sum_x P(x)f(x) \tag{2.6}$$

$$\mathbb{E}[f(x)] = \int p(x)f(x)dx \tag{2.7}$$

Expectation is equal to the mean only if the elements of distribution are equiprobable, that it means that $P(x)$ is an uniform distribution.

Definition 2.5.2 — Variance. The variance gives a measure of how much the values of a function of a random variable x vary as we sample different values of x from its probability distribution.

$$Var((f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{2.8}$$

Corollary 2.5.1 The square root of the variance is known as the **standard deviation**.

Definition 2.5.3 — Covariance. The covariance gives some sense of how much two values are linearly related to each other, as well as the scale of these variables

$$\text{Cov}(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])] \tag{2.9}$$

If the sign of the covariance is positive, then both variables tend to take on relatively high values simultaneously. If the sign of the covariance is negative, then one variable tends to take on a relatively high value at the times that the other takes on a relatively low value and vice versa.

Corollary 2.5.2 — Covariance matrix. The covariance matrix of a random vector $x \in R^n$ is an nn matrix, such that

$$\text{Cov}(\mathbf{x})_{i,j} = \text{Cov}\left(\text{x}_i, \text{x}_j\right). \tag{2.10}$$

2.6 Bayes' Rule

Definition 2.6.1 — Bayes' Rule. We often find ourselves in a situation where we know $P(y|x)$ and need to know $P(x|y)$. Fortunately, if we also know $P(x)$, we can compute the desired quantity using Bayes' rule

$$P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})} = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{\sum_x P(\mathrm{y}|\mathrm{x})P(\mathrm{x})} \tag{2.11}$$

2.7 Common Probability Distributions

Several simple probability distributions are useful in many contexts in machine learning.

2.7.1 Bernoulli Distribution

Suppose that a trial, or an experiment, whose outcome can be classified as either a "success" or as a "failure" is performed. If we let X = 1 when the outcome is a success and X = 0 when it is a failure, then the probability mass function of X is given by

$$P\{X = 0\} = 1 - p \tag{2.12}$$
$$P\{X = 1\} = p \tag{2.13}$$

Definition 2.7.1 — Bernoulli. A random variable X is said to be a Bernoulli random variable if its probability mass function is given by $P\{X = 0\} = 1 - p \;\; \& \;\; P\{X = 1\} = p$ for some $p \in (0, 1)$, where is the probability of "success".

$$E_x[X] = p \tag{2.14}$$
$$\mathrm{Var}_\mathrm{x}(\mathrm{x}) = p(1 - p) \tag{2.15}$$

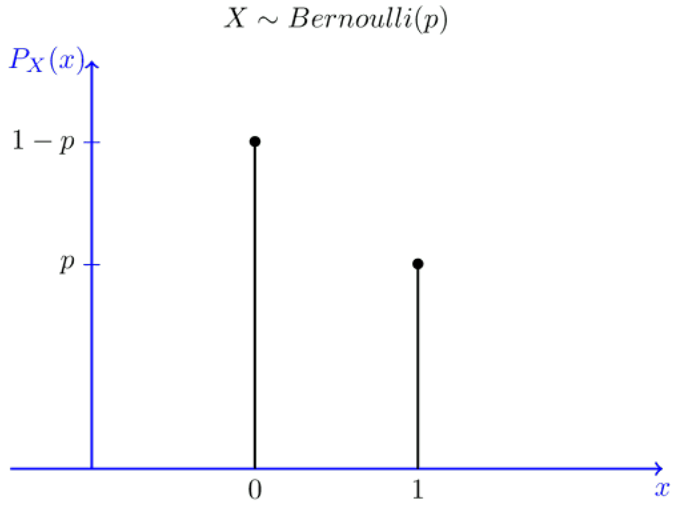


Figure 2.1: Bernoulli Distribution

2.7.2 Binomial Distribution

Suppose now that n independent trials, each of which results in a "success" with probability p and in a "failure" with probability $1 - p$, are to be performed.

Definition 2.7.2 — Binomial. If X represents the number of successes that occur in the n trials, then X is said to be a binomial random variable with parameters (n, p)

$$P\{X=i\} = \binom{n}{i} p^i (1-p)^{n-i}, \quad i = 0,1,\ldots,n \tag{2.16}$$

$$E[X] = \sum_{i=1}^{n} E[X_i] = np \tag{2.17}$$

$$\mathrm{Var}(X) = \sum_{i=1}^{n} \mathrm{Var}(X_i) = np(1-p) \tag{2.18}$$

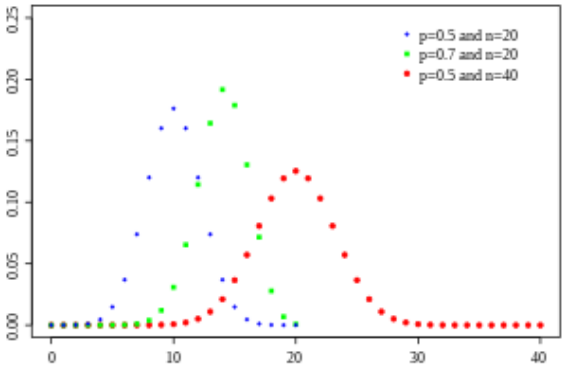


Figure 2.2: Binomial Distribution

2.7.3 Multinomial Distribution

The multinomial distribution is a generalization of the binomial distribution. In the multinomial distribution, the number of possible outcomes on any one given is allowed to be greater than 2. Therefore, suppose that there are n independent trials and each trials result in one of k mutually exclusive outcomes; on any single trial these k outcomes occur with probabilities $p_1,\ldots,p_k$ (sum of p_i is equal to 1).

Definition 2.7.3 — Multinomial. Let the random variable X_i represent the number of occurrences of outcome i.

$$P(X_1 = x_1,\ldots,X_k = x_k) = \frac{n!}{x_1!\cdots x_k!} p_1^{x_1}\cdots p_k^{x_k} \tag{2.19}$$

$$E[X_i] = np_i \tag{2.20}$$

$$\mathrm{Var}(X_i) = np_i(1-p_i) \tag{2.21}$$

Corollary 2.7.1 $p_1\cdots p_k$ is the probability of any one specific ordering that multiply for the number of possible ordering $x_1!\cdots x_k!$ give us the probability that $P(X_1 = x_1,\ldots,X_k = x_k)$ happening.

2.7.4 Gaussian or Normal Distribution

Normal distributions are a sensible choice for many applications. In the absence of prior knowledge about what form a distribution over the real numbers should take, the normal distribution is a good default choice for two major reasons. First, many distributions we wish to model are

truly close to being normal distributions. The central limit theorem shows that the sum of many independent random variables is approximately normally distributed. This means that in practice, many complicated systems can be modeled successfully as normally distributed noise. Second, out of all possible probability distributions with the same variance, the normal distribution encodes the maximum amount of uncertainty over the real numbers.

Definition 2.7.4 — Gaussian. The most commonly used distribution over real numbers is the normal distribution, also known as the Gaussian distribution.

$$N(x : \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{2.22}$$

$$E[X] = \mu \tag{2.23}$$

$$Var(X) = \sigma^2 \tag{2.24}$$

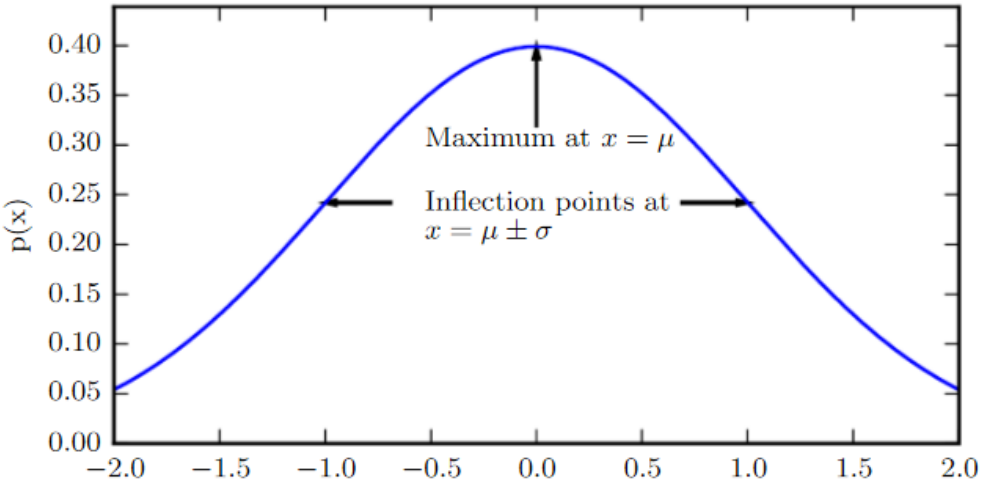


Figure 2.3: Normal Distribution

II Machine Learning

3. Introduction

3.1 What is Machine learning?

Before [1] to talk of Machine Learning we have to define the word "learning". So what is Learning?

Definition 3.1.1 — Learning. It's a process about humans and not machine, that during the history had had a different definitions:

- **Behaviorism, 1900-1950**
  Learning is a long-term change in **behavior** due to experience

- **Cognitivism, 1920**
  Learning is an internal **mental process** that integrates new information, in order to change previuos rules, into established mental frameworks and updates those frameworks over time.

- **Connectionism, 1949**
  Learning is a **physical process** in which neurons join by developing the synapses between them.

Defined learning can we explain the meaning of ML. Machine Learning is a field of study that gives computers the ability to learn without being explicity programmed.

Definition 3.1.2 — Machine Learning. A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

So Machine Learning is based on performing certain tasks T, that can be of two type:

1. **Supervised Task**: the goal is, given some point, draw a curve that can approximate them, in order to predict value of new point
2. **Unsupervised Task**: the goal is putting together data basing on some informations

Learning is accomplished by using an algorithm to adapt the parameters in a mathematical or

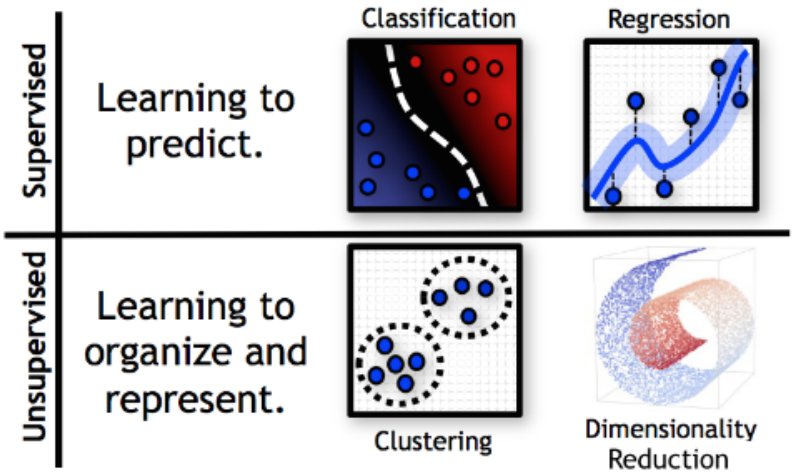


Figure 3.1: Types of machine learning tasks

statistical model given training data.

4. Supervised Learning Theory

4.1 Supervised Learning

Learning is based on the data, that have different forms depend on the type of tasks to perform. In the supervised learning data is defined in the following way.

Definition 4.1.1 — Data. A set of data records X $\in X^k$ also called **example**, instances or cases are described by

- k **features** or attributes: $x = \{x_1, x_2, \ldots, x_k\}$
- c $\in C$ **label**: each example is labelled with a predefined class target c

Therefore, when the labels are present in the problems, we talk about supervised learning and depending on the type of labels we can classify the tasks in the following way.

Definition 4.1.2 — Binary Classification. If labels can only take two values, we talk about binary classification.

$$D = \{(x_i, y_i), i = 1, \ldots, n\}, x_i \in X, y_i \in \{-1, 1\} \tag{4.1}$$

Definition 4.1.3 — Multiclass classification. If labels can take more than 2 values, we talk about multiclass classification.

$$D = \{(x_i, y_i), i = 1, \ldots, n\}, x_i \in X, y_i \in \{1, 2, \ldots, I\} \tag{4.2}$$

Definition 4.1.4 — Regression. If labels are real numbers, we talk about regression

$$D = \{(x_i, y_i), i = 1, \ldots, n\}, x_i \in X, y_i \in \mathbb{R} \tag{4.3}$$

The goal, in supervised learning, is to learn a classification model from the data that can be used to predict the classes of new examples. In order to do that we have to do a two step process:

1. Training: Learn a model using the training data
2. Testing or Inference: Test the model using unseen test data to measure the performance

4.1.1 Training

An example of dataset, sample by an unknown probability distribution, is given to function model that give as outcomes a specific label. This result is evaluated by the **loss function**, a function that measure the error(distance) between the result of function model and the label of example in dataset. After that, the result of loss function is given to optimizer that try to adjust the parameters of learning model in order to improve it (minimize the error). Therefore, we can define training as a try-and-error loop that ends when performance no longer improves.

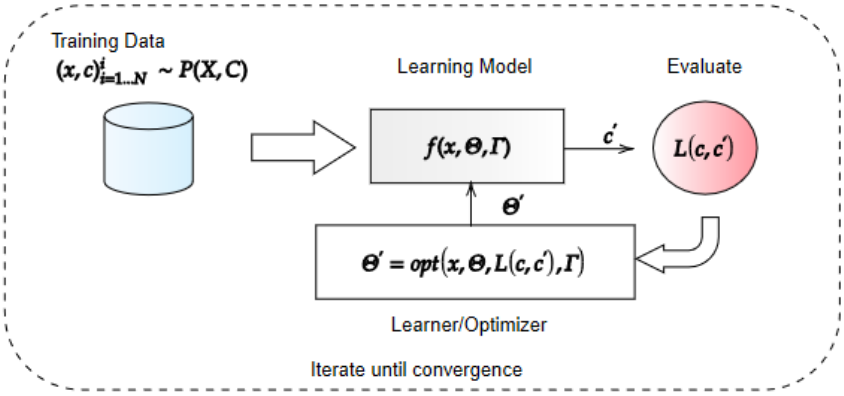


Figure 4.1: Training pipeline

Definition 4.1.5 Given a Dataset D, a class task C and a performance function L a computer system is said to learn the task C from data D if after osberving D performance on L improves at a level that is superior with respect to random guessing.

But, learning is possible only when training and inference are performed from data that come from the same distribution $x \sim P(X)$.
What does it mean same distribution? It means that we can take training set and test set, mix them in one dataset and then separate them again and the performance still the same (mean and variance are the same for all the subset of sampled elements).
That is said that samples are independently and identically distributed (i.i.d) from training and testing.

Definition 4.1.6 — i.i.d.. In probability theory and statistics, a collection of random variables is independent and identically distributed if each random variable has the same probability distribution as the others and all are mutually independent.

Independent and identically distributed random variables are often used as an assumption, which tends to simplify the underlying mathematics.
To understand it we can use the concept of **classifier generalization**.

Definition 4.1.7 — Generalization error. Given a classification function $f \in F$ a class task $c \in C$ and a data distribution D the generalization error or **risk** of the classifier is

$$R(f) = P_{x\sim D}[f(x) \neq c(x)] = \mathbb{E}_{x\sim D}[1_{f(x)\neq c(x)}] \tag{4.4}$$

It is the expectation (mean value) of the times that the function f produces an error on prediction.

Corollary 4.1.1 — Indicator function. In mathematics, an indicator function or a characteristic function of a subset of a set is a function that maps elements of the subset to one, and all other elements to zero.

$$1_A = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases} \tag{4.5}$$

This value is no measurable because is valuated on unknown data distribution D that represent all possible data in specific field (ex. all data in the universe).
This is the reason why we compute another measure called **empirical risk.**

Definition 4.1.8 — Empirical risk. The empirical risk given a set of m elements drawn by the distribution D, namely sample or samples, S = $(x_1, x_2, \ldots, x_m)$ (ex. photos on my PC) is

$$\hat{R}_S(f) = \frac{1}{m} \sum_{i=1}^{m} 1_{f(x) \neq c(x)} \tag{4.6}$$

Generally $R(f) \neq S(f)$ and this difference is the **generalization gap**. It tells the difference in perfomance from training and testing. Therefore, to have minimal difference, in performance, between training and test I would like to have the 2 error equals as possible.
Theorically, the gap is zero when i.i.d assumpition is verified.

Theorem 4.1.2 — i.i.d case. If S is drawn i.i.d from D we have

$$\mathbb{E}_{S \sim D^m}[\hat{R}_S(f)] = \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{S \sim D^m}[1_{f(x) \neq c(x)}] = \mathbb{E}_{S \sim D^m}[1_{f(x) \neq c(x)}] \tag{4.7}$$

I take m subsets S from D, all equidistributed, and I compute the average error for each S. Using the fact that is samples are i.i.d. the expectation assumes the same value on every sample set, so the error will be equals to the expectation on single S.

$$\mathbb{E}_{S \sim D^m}[\hat{R}_S(f)] = \mathbb{E}_{S \sim D^m}[1_{f(x) \neq c(x)}] = \mathbb{E}_{S \sim D^m}[1_{f(x) \neq c(x)}] = R(f) \tag{4.8}$$

In other words, if i.i.d. assumption is verified compute the error on S it will mean compute the error on all distribution D.

Under the i.i.d. assumption the error you get on training will almost be equal or very close to the error you will observe during test. Obviously, i.i.d. is not always verified because is not possible sample in uniform way from D.

4.1.2 Testing

Test process is similar to train process, but we haven't the optimization process:

1. Annotated samples, from test data (unknown probability distribution), are given as input to the prediction function.
2. The prediction function compute the prediction, using the last optimized parameters.
3. The predicted class is evaluate with true class.

4.1.3 Model Evaluation

How can we evaluate the prediction function?
The available dataset D, especially when D is large, is partitioned into two disjoint sets (70-30)

- **Training set Tr** for learning a model
- **Test set or holdout set Te** for testing the model

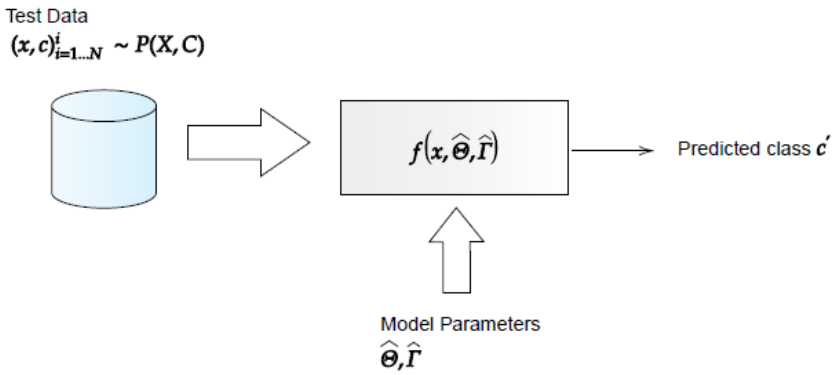


Figure 4.2: Inference pipeline

Obviously, training set should not be used in testing and the test set should not be used in learning. Furthermore, all classifers have additional parameters called **hyperparameters** that are not learnt but selected by design before training. Therefore it exists another set calles **validation set**, used for tuning hyperparameters and select the best model.

Depends on the size of dataset D, we have 2 way to do the learning process:

- **Train-Test-Validation**, used when the D is large (order of elements doesn't matter)
  1. Given a data set D, we randomly partition the data cases into a training set (Tr), a validation set (V), and a test set (Te). Typical splits are 60/20/20, 80/10/10, etc.
  2. Models M_i are learned on Tr for each choice of hyperparameters H_i
  3. The validation error V al_i of each model M_i is evaluated on V .
  4. The hyperparameters H_* with the lowest value of V al_i are selected and the classifier is re-trained using these hyperparameters on Tr + V , yielding a final model M_*
  5. Generalization performance is estimated by evaluating error/accuracy of $M*$ on the test data Te.
- **Cross-Validation and Test**, used for tuning hyperparameters and when dataset is not enough large.
  1. Randomly partition D into a set of K blocks $B_1, \ldots, B_K$.
  2. For each crossvalidation fold $k = 1, \ldots, K$:
     - Let Te = Bk and Tr = $D - B_k$ (the remaining K - 1 blocks).
     - Learn model M_k on Tr and test on B_k
  3. The procedures provide K accuracies.
  4. The final estimation of model performances is the average of the K accuracies
  5. 10-fold or 5-fold are typically used as a common numbers.

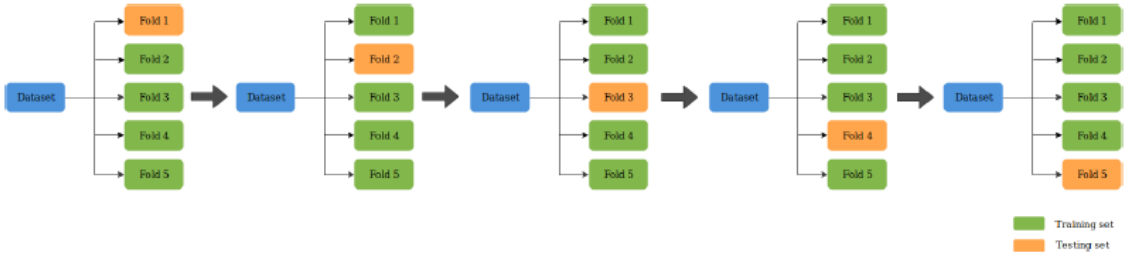


Figure 4.3: 5-Fold Cross-Validation and Test

4.1.4 Performance

In order to evaluate the performance of a model we have to define some measures:

- **Accuracy** = Number of correct classification / Total Number of cases
- Efficiency: time to construct the model and to use it.
- Robustness: handling noise and missing values
- Scalability: efficiency in disk-resident databases
- Interpretability: understandable and insight provided by the model
- Compactness of the model: size of the tree, or the number of rules.

The most important measure is accuracy but obviously is not the only one. In particular, accuracy is not suitable in some applications for example in context with rare events or with not equidistributed class.

For example we take a model that must recognize if there is a fire or not. We can define the following events.

- TP = when the model predict that there is a fire and the fire exists.
- FP = when the model predict that there is a fire but the fire doesn't exist.
- TN = when the model predict that there isn't a fire and the fire doesn't exist.
- FN = when the model predict that there isn't a fire but the fire exists.

		Truth: Positive	Truth: Negative	
Test	Positive	True Positive	False Positive	Total Testing Positive
	Negative	False Negative	True Negative	Total Testing Negative
		Total True Positive	Total True Negative	

Figure 4.4: Confusion Matrix

Definition 4.1.9 — Precision. The number of correctly classified positive examples divided by the total number of examples that are classified as positive.

$$Precision = TP/(TP+FP) \tag{4.9}$$

It represent the certainty of model but it doesn't count about false negative.

Definition 4.1.10 — Recall. The number of correctly classified positive examples divided by the total number of actual positive examples in the test set.

$$Recall = TP/(TP+FN) \tag{4.10}$$

In our example recall equals to 1 it means that every time that the model predict a fire, the fire really exists.

Definition 4.1.11 — F1-score. It is the harmonic mean of precision and recall.

$$\frac{1}{F1} = \frac{1}{p} + \frac{1}{r} \tag{4.11}$$

Why harmonic mean and not just an arithmetic mean? Because harmonic mean is always closer to lowest value.

Typically, the output of the system is a score and to obtain a category we need a threshold. Varying

threshold varies the events TP,FP,TN,FN and precision and recall accordingly. To understand what is the value threshold (working point) is used the **ROC** curve that measure the correlation between precision and recall.

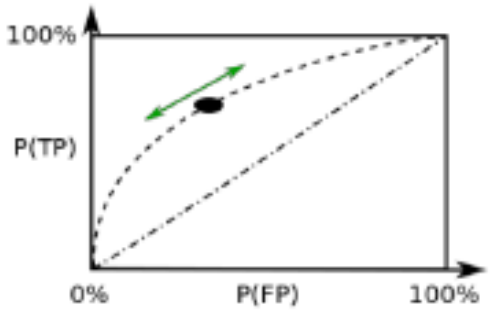


Figure 4.5: ROC curve

For these reason precision and recall are not absolute and they depend on threshold and application. All we say is true for discrete classes but the majority of classifier gives as output a score or a probability distribution. How we can proceed?
We have to define some tool to evaluate the performance also with continuous classes.

1. Thresholding the probability and using discrete class values
2. Using distance between distributions P(x) = true discrete distribution (discrete) and Q(x) = estimated distribution

Definition 4.1.12 — Bhattacharrya coefficient. It is an approximate measurement of the amount of overlap between two statistical samples.

$$BC(P,Q) = \int_x P(x)Q(x)dx \tag{4.12}$$

Obviously, in the discrete world we will have to sum. It is not really a difference but is more a coefficient that remain high if the values of two distributions are high (high level of overlap)

Definition 4.1.13 — Kullback-Leibler Divergence. It is a non symmetric ($KL(P|Q) \neq KL(Q|P)$) measure of lost information when distribution Q is used to approximate distribution P

$$KL(P|Q) = \int_x P(x) \log \frac{P(x)}{Q(x)} dx \tag{4.13}$$

Obviously, when $Q(X) = P(X)$ the $KL = 0$ and there isn't lose of information.

Definition 4.1.14 — Cross Entropy. It is an information theory measure that evaluates the average number of bits for discovering a certain value by a distribution q while the original one was p (entropy $H(P)$ measure the difficulty of problem).

$$H(P,Q) = H(P) + KL(P|Q) = -\int_x P(x)\log P(x) + \int_x P(x) \log \frac{P(x)}{Q(x)} dx = \tag{4.14}$$

$$= -\int_x P(x)\log P(x) + \int_x P(x)\log P(x)dx - \int_x P(x)\log Q(x) = \tag{4.15}$$

$$= -\int_x P(x)\log Q(x) \tag{4.16}$$

To compute cross entropy we can also use the theorem of Kraft-McMillan.

Theorem 4.1.3 — Kraft-McMillan theorem. A value x_i can be identified by l_i bits with probability $Q(x_i) = 2^{-l_i}$

For example with 1 bit the probability is 0.5 because we have two values 1 and 0. Computing the Expectation of l respect to P(x) we have:

$$\mathbb{E}_p[l] = \sum_x P(x) \log_2 \frac{1}{Q(x)} = \sum_x P(x) \log_2 Q(x)^{-1} \tag{4.17}$$

$$= -\sum_x P(x) \log_2 Q(x) \tag{4.18}$$

That is the cross entropy for discrete case, that can be used as a measure of error.

Corollary 4.1.4 Consider P a discrete distribution with k possible values and the problem being a k-class classification.

- $P_i(x)$ = 1 if x belongs to class i (use as selector).
- $Q_i(x)$ is the probability score that the classifier attributes to class i for element x.

The Expectation of the discrete Cross Entropy over the entire dataset D of N equiprobable elements is:

$$\mathbb{E}_p[H(P,Q)] = \frac{1}{N} \sum_{x \in D} \left(-\sum_x P(x) \log_2 Q(x) \right) \tag{4.19}$$

In binary case the binary cross entropy simplifies to:

$$\frac{1}{N} \sum_{x \in D} \left(-\sum_x P(x) \log_2 Q(x) \right) = -\frac{1}{N} \sum_{x \in D} [P_0(x) \log Q_0(x) + (1 - P_0(x) \log(1 - Q_0(x))] \tag{4.20}$$

5. Probability Classifier

Up to this point, we have not made any assumptions about the classification functin f. However, if we assume that it outputs the probability of an object belonging to a particular class, we would then be dealing with a probabilistic classifier. Suppose we know that the true probability of seeing a data case that belongs to class c is $\pi_c = P(Y = c)$ and the true probability density of seeing a data vector $x \in R^D$ that belongs to class c $\phi_c(\mathbf{x}) = p(X = \mathbf{x}|Y = c)$.
We are interested in compute the probability that an example x belongs to class c and to do that we have to define some probabilities.

Definition 5.0.1 — Likelyhood. Likelyhood function measures given the class c how likely is to observe x. In other words, it measures, fixing a certain class c, how what I am seeing (x) is consistent with the class.

$$P(X = \mathbf{x}|Y = c) \tag{5.1}$$

Definition 5.0.2 — Prior. Prior probability function measures without any observation x how likely is class c.

$$P(Y = c) \tag{5.2}$$

Given the likelyhood function and the Prior probability by applying the Bayes rule it is possible to obtain the **A-Posteriori** probability, the probability that an example x belongs to class c after observing the data.

$$P(Y = c|X = \mathbf{x}) = \frac{p(X = \mathbf{x} \cap Y = c)}{p(X = \mathbf{x})} = \frac{p(X = \mathbf{x}, Y = c)}{p(X = \mathbf{x})} \tag{5.3}$$

$$= \frac{p(X = \mathbf{x}|Y = c) \cdot P(Y = c)}{p(X = \mathbf{x})} \tag{5.4}$$

But we can rewrite total probability $P(X = x)$ in the following way:

$$p(X = x) = \sum_{c' \in \mathscr{Y}} p(X = \mathbf{x}|Y = c')P(Y = c') \tag{5.5}$$

$$= \sum_{c' \in \mathscr{Y}} p(X = \mathbf{x} \cap Y = c') \tag{5.6}$$

$$= \sum_{c' \in \mathscr{Y}} p(X = \mathbf{x}, Y = c') \tag{5.7}$$

Where $\mathscr{Y}$ is the set of all possible classes. So the final equation is the following one:

$$P(Y = c|X = \mathbf{x}) = \frac{p(X = \mathbf{x}|Y = c) \cdot P(Y = c)}{\sum_{c' \in \mathscr{Y}} p(X = \mathbf{x}, Y = c')} = \frac{\phi_c(\mathbf{x}) \cdot \pi_c}{\sum_{c' \in \mathscr{Y}} \phi_{c'}(\mathbf{x})\pi_{c'}} \tag{5.8}$$

We can arrive at the same result also by recalling the definition of the marginal PMF (Probability Mass Function) of X:

$$p_X(x) = P(X = x) = \sum_y p_{XY}(x, y) \tag{5.9}$$

Corollary 5.0.1 — Important Note. The likelihood is a probability density function (note the lowercase p) for which a consideration must be made.

In the discrete case, this density function is an actual probability indeed:

$$p_X(x) = P(X = x) \in [0, 1] \tag{5.10}$$

$$\sum_{x \in Val(X)} p_X(x) = 1 \tag{5.11}$$

In the continuous case, however, it is not a probability but simply a non-negative function (density probability)

$$f_X(x) \geq 0 \tag{5.12}$$

$$\int_{-\infty}^{+\infty} f_X(x)dx = 1 \tag{5.13}$$

Where to obtain the probability will have to integrate over a certain interval:

$$P(a \leq X \leq b) = \int_a^b f_X(x)dx, \quad \forall a \leq b \tag{5.14}$$

Corollary 5.0.2 It doesn't make sense, however, to calculate $P(X = a)$ from a density as I would always get 0 $\forall a$:

$$P(a \leq X \leq a) = \int_a^a f_X(x)dx = F_X(a) - F_X(a) = 0 \tag{5.15}$$

And what can we do if we have a continuous random variable and a discrete one? There exist mixed probability distributions where I have one discrete random variable and one continuous. For these, Bayes' formula is also defined as follows:

$$p_{Y|X}(y|x) = \frac{f_{X|Y}(x|y) \cdot p_Y(y)}{f_X(x)} \tag{5.16}$$

The result is a probability (indeed indicated with lowercase p) because the density $f_{X|Y}(x|y)$ in the numerator, which is a number that can be greater than 1, is normalized by the $f_X(x)$ present in the denominator, bringing the result back to being a number between 0 and 1, i.e., a probability.

How can we classify with this probabilities? Given a labeled dataset $\mathscr{D} = (\mathbf{x}_i, y_i)_{i=1,\dots,N}$, k classes $c_i | i = 1,\dots,k$ and a generic test element $\hat{\mathbf{x}}$, there are two classification strategies:

- **Maximum Likelihood**
  1. Learn the Likelihood $p(\mathbf{x}|c_i)$ from the dataset for each class c_i
  2. Recalculate with $\hat{\mathbf{x}}$ for each of the k Likelihoods
  3. $\hat{c} = argmax_c\ p(\hat{\mathbf{x}}|c)$
- **Maximum A-Posteriori**
  1. Learn the Likelihood $p(\mathbf{x}|c_i)$ from the dataset for each class c_i
  2. Also learn the Prior $P(c_i)$ from the dataset for each class c_i
  3. Apply Bayes' rule, finding

$$p(c_i|\hat{\mathbf{x}}) = \frac{p(\hat{\mathbf{x}}|c_i)P(c_i)}{\sum_{i=1}^{k} p(\hat{\mathbf{x}}|c_i)P(c_i)} \tag{5.17}$$

  4. $\hat{c} = argmax_c\ p(c_i|\hat{\mathbf{x}})$

In reality, when applying Bayes' formula, we can avoid calculating the denominator as it is a normalizer equal for all posteriors. Therefore, we only calculate:

$$\frac{\phi_{c_i}(\mathbf{x}) \cdot \pi_{c_i}}{\cancel{\sum_{i=0}^{k} \phi_{c_i}(\mathbf{x})\pi_{c_i}}} = \phi_{c_i}(\mathbf{x}) \cdot \pi_{c_i} \tag{5.18}$$

where $\phi_{c_i}(\mathbf{x})$ represents the likelihood and π_{c_i} represents the prior probability.

In general, the posterior or Bayesian classifier is more robust because it also takes into account the prior probabilities. However, it has the drawback of penalizing less probable classes. Indeed, if we have a rare class for which the prior has a low value, even if the element we want to classify has a high Likelihood (i.e., it is very likely to belong to that rare class), the posterior will end up having a low value because in the formula, Prior and Likelihood are multiplied together. So A-posterior classifier is usually really reliable when the classes are equiprobable.

5.1 Bayes Optimal Classifier

As we said before, the Bayes classification function is whose maximize the a-posteriori probability.

$$f_B(\mathbf{x}) = \underset{c \in \mathscr{Y}}{\text{argmax}}\ P(Y = c|X = \mathbf{x}) = \underset{c \in \mathscr{Y}}{\text{argmax}}\ \phi_c(\mathbf{x}) \cdot \pi_c \tag{5.19}$$

It is considered optimal as it minimizes the expected error among all classifiers:

$$1 - E_{P(X=\mathbf{x})}[\max_{c \in \mathscr{Y}} P(Y = c|X = \mathbf{x})] \tag{5.20}$$

We have some practical problems with this classifier:

- $P(X = \mathbf{x})$ **is not finite**, and this error depends on the functions we have learned, which could be incorrect.
- A-prior is easy to compute (for class c is the number of instance labelled with c divide by the total number of instance) but calculating the **likelihood is very complex** because x is usually x is a vector or a complex feature)

Since the likelihood is a probability density, this is a density estimation problem.

5.1.1 Density Estimation

Among the various densities we could estimate, the first that comes to mind might be the joint density. Indeed, from $p(\mathbf{x}, c)$ we could calculate the conditional density and the prior density, as $p(\mathbf{x}, c) = p(\mathbf{x}|c)p(c)$.
The problem is that $p(\mathbf{x}, c)$ is difficult to estimate because we need to create the so-called joint density (JD) table, which accounts for all possible combinations of our variables. This cannot be created if the values that our random variables can take are not countable or if they are countable but in large numbers; creating it is too expensive.

JD

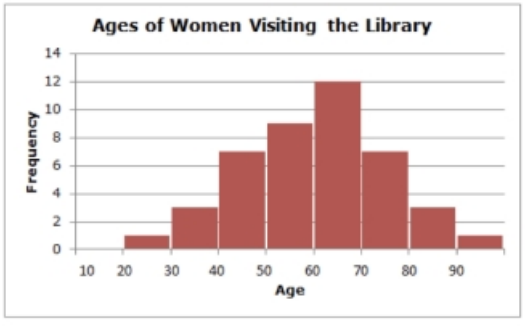
Gender	Hours worked	Wealth	Prob
Female	< 40	Poor	0,25
		Rich	0,024
	>40	Poor	0,042
		Rich	0,01
Male	< 40	Poor	0,33
		Rich	0,09
	>40	Poor	0,134
		Rich	0,12

Moreover, if a combination never appears in our dataset, we must put zero probability in the table, not reflecting reality. In fact, our dataset is only a small observation of reality, so it might not cover all cases. By doing so, the model learns an incorrect or incomplete observation and is said to be subject to **overfitting**.

Definition 5.1.1 — Overfitting. The model has adapted to the specific dataset and is unable to generalize.

As an alternative to joint density, conditional density can be calculated, and there are two ways:

- **Non-parametric estimation**: We construct a histogram (discrete case) where each column represents the number of times the variable x has taken a certain value. It has manageability problems if, for example, the number of values that x can take is high, it is subject to overfitting unlike the parametric one (which interpolates where there is no data), and depending on how we discretize the x-axis, the distribution could change.



- **Parametric estimation**: Instead of calculating it, we fix a known density function, parameterized by parameters Θ, $p(\mathbf{x}; \Theta)$ that could well represent our data and learn its optimal parameters on the dataset by maximizing its likelihood:

$$\hat{\Theta} = argmax_{\Theta} \underbrace{\prod_{\mathbf{x}_i \in \mathscr{D}} p(\mathbf{x}_i; \Theta)}_{\text{Likelihood of the entire dataset}} = argmax_{\Theta} \underbrace{\sum_{\mathbf{x}_i \in \mathscr{D}} \log\, p(\mathbf{x}_i; \Theta)}_{\text{Log Likelihood of the entire dataset}} \tag{5.21}$$

The use of logarithm is necessary because very often probability densities have values between 0 and 1, and multiplying by such a value is equivalent to dividing, so we lose the meaning of the product, making it unstable. By applying the logarithm, due to its properties, the product becomes a sum, obtaining the Log Likelihood which no longer presents instability problems. Moreover, the logarithm is a monotonic function (the order of elements is maintained), so doing maximum likelihood is equivalent to doing maximum log likelihood.

An example of a density function could be the 1D Gaussian where $\Theta = \mu, \sigma^2$:

$$f(\mathbf{x}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\mathbf{x}-\mu)^2}{2\sigma^2}} \tag{5.22}$$

This is the case where $\mathbf{x}$ is one-dimensional, but in general, being a vector, it could have more features. In that case, we will need to use the multivariate Gaussian.

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} e^{\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)} \tag{5.23}$$

With the 1D Gaussian the log likelihood on the dataset is the following:

$$L(\mu, \sigma^2; \mathbf{x}) = \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\mathbf{x}_i-\mu)^2}{2\sigma^2}}\right) \tag{5.24}$$

Since we want to maximize this quantity, we need to calculate its derivatives with respect to the parameters and set them equal to zero, thus finding the parameters that maximize it.

- First, let's rewrite the log likelihood formula as follows.

$$L(\mu, \sigma^2; \mathbf{x}) = \sum_{i=1}^{n} \left(\ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \ln\left(e^{-\frac{(\mathbf{x}_i-\mu)^2}{2\sigma^2}}\right)\right) \tag{5.25}$$

$$= \sum_{i=1}^{n} \left(\ln\left(2\pi\sigma^2\right)^{-1/2} - \frac{(\mathbf{x}_i-\mu)^2}{2\sigma^2}\right) \tag{5.26}$$

$$= \sum_{i=1}^{n} \left(-\frac{1}{2}\ln\left(2\pi\sigma^2\right) - \frac{(\mathbf{x}_i-\mu)^2}{2\sigma^2}\right) \tag{5.27}$$

$$= -\frac{1}{2}\sum_{i=1}^{n} \left(\ln\left(2\pi\sigma^2\right) + \frac{(\mathbf{x}_i-\mu)^2}{\sigma^2}\right) \tag{5.28}$$

- Proceed with the calculation of the first partial derivative

$$\frac{\partial L(\mu, \sigma^2; \mathbf{x})}{\partial \mu} = -\frac{1}{2}\sum_{i=1}^{n} \left(0 + \frac{1}{\sigma^2} \cdot 2(\mathbf{x}_i-\mu)(-1)\right) = 0 \tag{5.29}$$

$$\sum_{i=1}^{n} (\mathbf{x}_i - \mu) = 0 \tag{5.30}$$

$$\sum_{i=1}^{n} \mathbf{x}_i = \sum_{i=1}^{n} \mu \tag{5.31}$$

$$\sum_{i=1}^{n} \mathbf{x}_i = n\mu \tag{5.32}$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \tag{5.33}$$

$$\tag{5.34}$$

- Proceed with the calculation of the second partial derivative.

$$\frac{\partial L(\mu,\sigma^2;\mathbf{x})}{\partial \sigma^2} = -\frac{1}{2}\sum_{i=1}^{n}\left(\frac{\cancel{2\pi}}{\cancel{2\pi}\sigma^2} + (\mathbf{x}_i-\mu)^2\cdot\left(-\frac{1}{\sigma^4}\right)\right) = 0 \tag{5.35}$$

$$\sum_{i=1}^{n}\left(\frac{(\mathbf{x}_i-\mu)^2}{\sigma^4} - \frac{1}{\sigma^2}\right) = 0 \tag{5.36}$$

$$\sum_{i=1}^{n}\left(\frac{(\mathbf{x}_i-\mu)^2-\sigma^2}{\sigma^4}\right) = 0 \tag{5.37}$$

$$\sum_{i=1}^{n}(\mathbf{x}_i-\mu)^2 = \sum_{i=1}^{n}\sigma^2 \tag{5.38}$$

$$\sum_{i=1}^{n}(\mathbf{x}_i-\mu)^2 = n\sigma^2 \tag{5.39}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i-\mu)^2 \tag{5.40}$$

$$\tag{5.41}$$

However, when the cardinality of the number of features is very high, the overfitting problem remains because it will be very difficult to have all possible combinations, especially if the dataset is small. To solve this problem, naive density estimators are introduced.

Naive Density Estimator

This estimator assumes all features are independent of each other. In this way estimator doesn't overfit because it doesn't depend on all feature configurations but only on individual ones.

$$P(x_1,\ldots,x_n|c) = \prod_{i=1}^{k} P(x_i|c) \tag{5.42}$$

5.2 Naive Bayes Classifier

We have thus obtained an approximation of the optimal Bayes classifier called the Naive Bayes classifier, which uses a simplified version of the likelihood (**Note the transition from vector x to feature x_d!**)

$$\phi_c(\mathbf{x}) = p(X=\mathbf{x}|Y=c) = \underbrace{\prod_{d=1}^{k} p(X_d=x_d|Y=c)}_{\text{Naive DE Assumption}} = \prod_{d=1}^{k}\phi_{cd}(x_d) \tag{5.43}$$

The classification function will thus have the following form.

$$f_{NB}(\mathbf{x}) = \underset{c\in\mathcal{Y}}{\operatorname{argmax}}\ \pi_c \prod_{d=1}^{k}\phi_{cd}(x_d) \tag{5.44}$$

However, in this case too, I will somehow have to approximate the likelihoods with a known distribution.

- **Gaussian**, for real values:

$$\phi_{cd}(x_d) = \mathcal{N}(x_d;\mu_{dc},\sigma_{dc}^2) = \frac{1}{\sqrt{2\pi}\sigma_{dc}} e^{-\frac{(x_d-\mu_{dc})^2}{2\sigma_{dc}^2}} \tag{5.45}$$

Without the naive assumption, I would have had to calculate the multivariate Gaussian.

- **Bernoulli**, for binary values:

$$\phi_{cd}(x_d) = \theta_{dc}^{x_d}(1-\theta_{dc})^{(1-x_d)} \tag{5.46}$$

- **Categorical**, for categorical values:

$$\phi_{cd}(x_d) = \prod_{v \in \mathscr{X}_d} \theta_{vdc}^{[x_d=v]} \tag{5.47}$$

As seen previously, prior probabilities π_c and the parameters of the marginal class conditional distributions $\phi_{cd}(x_d)$ are learned on the dataset by maximizing the likelihood. The prior probability doesn't depend on x and is calculated as the average of correct predictions:

$$\pi_c = \frac{1}{n}\sum_{i=1}^{n}[y_i = c] \tag{5.48}$$

Regarding the parameters, for example of the Gaussian, it is calculated as we have already demonstrated. But now we move from parameters calculated with respect to the entire feature vector to the version for individual features and we want to calculate the optimal parameters, so we consider only the features whose classes are correct:

$$\mu = \frac{\sum_{i=1}^{n}[y_i = c]x_{di}}{\sum_{i=1}^{n}[y_i = c]}\ ,\ \sigma^2 = \frac{\sum_{i=1}^{n}[y_i = c](x_{di}-\mu_{dc})^2}{\sum_{i=1}^{n}[y_i = c]} \tag{5.49}$$

Exercise 5.1 Let's see the calculation for the Bernoulli distribution.

$$L(\theta_{dc};\mathbf{x}) = \sum_{i=1}^{n} \log\left(\theta_{dc}^{x_{di}}(1-\theta_{dc})^{(1-x_{di})}\right) \tag{5.50}$$

$$= \sum_{i=1}^{n} \left(\log \theta_{dc}^{x_{di}} + \log(1-\theta_{dc})^{(1-x_{di})}\right) \tag{5.51}$$

$$= \sum_{i=1}^{n} \left(x_{di}\log\theta_{dc} + (1-x_{di})\log(1-\theta_{dc})\right) \tag{5.52}$$

$$= \log\theta_{dc}\sum_{i=1}^{n} x_{di} + \log(1-\theta_{dc})\sum_{i=1}^{n}(1-x_{di}) \tag{5.53}$$

$$= \log\theta_{dc}\sum_{i=1}^{n} x_{di} + \log(1-\theta_{dc})\left(\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} x_{di}\right) \tag{5.54}$$

$$= \log\theta_{dc}\sum_{i=1}^{n} x_{di} + \log(1-\theta_{dc})\left(n - \sum_{i=1}^{n} x_{di}\right) \tag{5.55}$$

$$\text{Let} \sum_{i=1}^{n} x_{di} = Y \tag{5.56}$$

$$= Y\log\theta_{dc} + (n-Y)\log(1-\theta_{dc}) \tag{5.57}$$

$$\tag{5.58}$$

$$\frac{dL(\theta_{dc};\mathbf{x})}{d\theta_{dc}} = \frac{Y}{\theta_{dc}} + \frac{n-Y}{1-\theta_{dc}}(-1) = 0 \tag{5.59}$$

$$= \frac{Y}{\theta_{dc}} + \frac{Y-n}{1-\theta_{dc}} = 0 \tag{5.60}$$

$$= \frac{Y(1-\theta_{dc}) + (Y-n)\theta_{dc}}{\theta_{dc}(1-\theta_{dc})} = 0 \tag{5.61}$$

$$= Y - \cancel{\theta_{dc}Y} + \cancel{\theta_{dc}Y} - \theta_{dc}n = 0 \tag{5.62}$$

$$= n\theta_{dc} = Y \tag{5.63}$$

$$= \theta_{dc} = \frac{Y}{n} = \frac{1}{n}\sum_{i=1}^{n} x_{di} \tag{5.64}$$

We want to calculate the optimal parameters, so we consider only the features whose classes are correct.

$$\theta_{dc} = \frac{\sum_{i=1}^{n}[y_i = c]x_{di}}{\sum_{i=1}^{n}[y_i = c]} \tag{5.65}$$

■

Exercise 5.2 Let's see the calculation for the categorical distribution.

$$L(\theta_{vdc};\mathbf{x}) = \sum_{i=1}^{n}\left(\log\left(\prod_{v\in\mathscr{X}}\theta_{vdc}^{[x_{di}=v]}\right)\right) \tag{5.66}$$

$$= \sum_{i=1}^{n}\sum_{v\in\mathscr{X}}\log\theta_{vdc}^{[x_{di}=v]} \tag{5.67}$$

$$= \sum_{i=1}^{n}\sum_{v\in\mathscr{X}}[x_{di}=v]\log\theta_{vdc} \tag{5.68}$$

$$\tag{5.69}$$

$$\frac{dL(\theta_{vdc};\mathbf{x})}{d\theta_{vdc}} = \sum_{i=1}^{n}\sum_{v\in\mathscr{X}}\frac{[x_{di}=v]}{\theta_{vdc}} = 0 \tag{5.70}$$

$$\tag{5.71}$$

Unfortunately, this calculation turns out to be very complicated because it must take into account the constraint.

$$\sum_{i=1}^{n}\theta_{vdc} = 1 \tag{5.72}$$

So it cannot be solved analytically, but we'll need to use (iterative) numerical optimization methods (such as gradient ascent since it's a maximization problem). We'll settle for the final solution:

$$\theta_{vdc} = \frac{\sum_{i=1}^{n}[y_i=c][x_{di}=v]}{\sum_{i=1}^{n}[y_i=c]} \tag{5.73}$$

■

5.2.1 Geometric Interpretation

Let's now try to interpret a classifier from a geometric point of view, and to do this, let's consider the Gaussian distribution:

$$f_{NB}(\mathbf{x}) = \underset{c\in\mathscr{Y}}{argmax}\ \pi_c\prod_{d=1}^{D}\phi_{cd}(x_d) \tag{5.74}$$

$$= \underset{c\in\mathscr{Y}}{argmax}\ \log(\pi_c) + \sum_{d=1}^{D}\log\ \phi_{cd}(x_d) \tag{5.75}$$

$$= \underset{c\in\mathscr{Y}}{argmax}\ \log(\pi_c) + \sum_{d=1}^{D}\left(-\frac{1}{2}\log\left(2\pi\sigma_{cd}^2\right) - \frac{(x_d-\mu_{cd})^2}{2\sigma_{cd}^2}\right) \tag{5.76}$$

$$\tag{5.77}$$

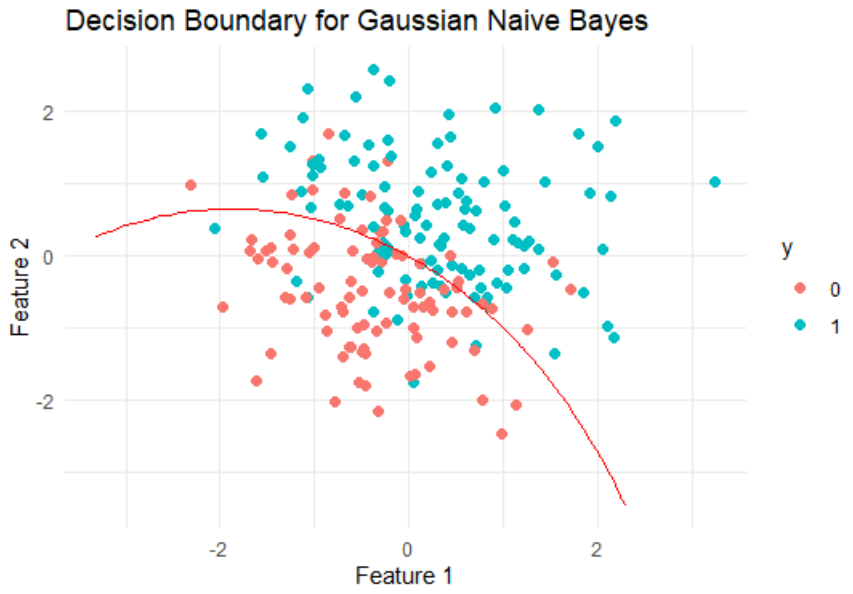
I applied the logarithm to the entire function to remove the product for the reasons already seen, and I can do this because it's a monotonic function, so the order of elements is maintained. The decision boundary is the border that separates two different classes and has, for example, the equation (score

of class 0 equals to score of class 1).

$$\log(\pi_0) + \sum_{d=1}^{D} \left(-\frac{1}{2}\log\left(2\pi\sigma_{d0}^2\right) - \frac{(x_d - \mu_{d0})^2}{2\sigma_{d0}^2} \right) + \tag{5.78}$$

$$= \log(\pi_1) + \sum_{d=1}^{D} \left(-\frac{1}{2}\log\left(2\pi\sigma_{d1}^2\right) - \frac{(x_d - \mu_{d1})^2}{2\sigma_{d1}^2} \right) \tag{5.79}$$

It's easy to notice that it's a quadratic function of x of the type $\sum_{d=1}^{D}(a_d x_d^2 + b_d x_d) + c = 0$, so the boundaries will be parabolas or hyperbolas.
In case of multiclass classification the border is quadratic two by two.



5.2.2 Trade-offs of the Naive Bayes Classifier

- Speed: both learning and classification have low computational complexity.
- Space: $O(DC)$ parameters (each parameter for each class takes D different values).
- Interpretability: good because the $\phi_c(\mathbf{x})$ are means conditioned on the class.
- Accuracy: low due to the naive assumption that almost never represents reality.
- Data: if scarce, care must be taken in estimating parameters.

5.3 Linear Discriminant Analysys

Linear Discriminant Analysis (LDA) is another approximation of the Bayes Optimal classifier for real-valued data. Unlike Naive Bayes, which uses independent Gaussian distributions, LDA assumes the distributions are multivariate Gaussian with a shared covariance matrix.

$$\phi_c(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu_c, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_c)^T \Sigma^{-1} (\mathbf{x} - \mu_c) \right) \tag{5.80}$$

Corollary 5.3.1 — Covariance matrix. In probability theory and statistics, a covariance matrix is a square matrix giving the covariance between each pair of elements of a given random vector.

$$\begin{pmatrix} \text{Var}(x_1) & \cdots & \text{Cov}(x_1, x_n) \\ \vdots & \ddots & \vdots \\ \text{Cov}(x_n, x_1) & \cdots & \text{Var}(x_n) \end{pmatrix} \tag{5.81}$$

Intuitively, the covariance matrix generalizes the notion of variance to multiple dimensions.

The classification function is the same as the Optimal Bayes classifier with a different $\phi_c(\mathbf{x})$

$$f_{LDA}(\mathbf{x}) = \underset{c \in \mathscr{Y}}{\text{argmax}}\ \phi_c(\mathbf{x}) \cdot \pi_c \tag{5.82}$$

In this case as well, maximum likelihood is used to learn the optimal parameters:

$$\pi_c = \frac{1}{n} \sum_{i=1}^{n} [y_i = c] \tag{5.83}$$

Let's calculate μ_c:

$$L(\mu_c, \Sigma; \mathbf{x}) = \sum_{i=1}^{n} \left(\log \left((2\pi)^d |\Sigma| \right)^{-1/2} - \frac{1}{2} (\mathbf{x}_i - \mu_c)^T \Sigma^{-1} (\mathbf{x}_i - \mu_c) \right) \tag{5.84}$$

$$= \sum_{i=1}^{d} \left(-\frac{1}{2} \log \left((2\pi)^d |\Sigma| \right) - \frac{1}{2} (\mathbf{x}_i - \mu_c)^T \Sigma^{-1} (\mathbf{x}_i - \mu_c) \right) \tag{5.85}$$

$$= \frac{1}{2} \sum_{i=1}^{d} \left(-\log \left((2\pi)^d |\Sigma| \right) - (\mathbf{x}_i - \mu_c)^T \Sigma^{-1} (\mathbf{x}_i - \mu_c) \right) \tag{5.86}$$

$$\tag{5.87}$$

Corollary 5.3.2 — Property matrix derivatives. One of the most important property is the following one.

$$\frac{d\mathbf{x}^T B \mathbf{x}}{dx} = B\mathbf{x} + B^T \mathbf{x} \tag{5.88}$$

$$\frac{\partial L(\mu_c,\Sigma;\mathbf{x})}{\partial \mu_c} = -\frac{1}{2}\sum_{i=1}^{n}\left(\Sigma^{-1}(\mathbf{x}_i-\mu_c)+\Sigma^{-T}(\mathbf{x}_i-\mu_c)\right)=0 \tag{5.89}$$

$$= -\frac{1}{2}\sum_{i=1}^{n}\left(\Sigma^{-1}(\mathbf{x}_i-\mu_c)+\Sigma^{-1}(\mathbf{x}_i-\mu_c)\right)=0 \tag{5.90}$$

$$= -\frac{1}{\cancel{2}}\sum_{i=1}^{n}\left(\cancel{2}\Sigma^{-1}(\mathbf{x}_i-\mu_c)\right)=0 \tag{5.91}$$

$$= -\Sigma^{-1}\sum_{i=1}^{n}(\mathbf{x}_i-\mu_c)=0 \tag{5.92}$$

$$= \sum_{i=1}^{n}(\mathbf{x}_i-\mu_c)=0 \tag{5.93}$$

$$= \sum_{i=1}^{n}\mathbf{x}_i = \sum_{i=1}^{n}\mu_c \tag{5.94}$$

$$= n\mu_c = \sum_{i=1}^{n}\mathbf{x}_i \tag{5.95}$$

$$= \mu_c = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i \tag{5.96}$$

As always, to calculate the optimal parameters we only consider the features whose classes are correct.

$$\mu_c = \frac{\sum_{i=1}^{n}[y_i=c]\mathbf{x}_i}{\sum_{i=1}^{n}[y_i=c]} \tag{5.97}$$

Now let's calculate Σ. To do this, we modify the log likelihood by exploiting a property of matrices.

Corollary 5.3.3 The trace of a scalar can be seen as the trace of a 1×1 matrix, i.e., the sum of its diagonal elements, which is the scalar itself.

We observe that $(\mathbf{x}_i-\mu_c)^T\Sigma^{-1}(\mathbf{x}_i-\mu_c)$ is a scalar, so we can replace it with $tr\left((\mathbf{x}_i-\mu_c)^T\Sigma^{-1}(\mathbf{x}_i-\mu_c)\right)$. Thus, we obtain:

$$L(\mu_c,\Sigma;\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\left(-\log\left((2\pi)^d|\Sigma|\right)-tr\left((\mathbf{x}_i-\mu_c)^T\Sigma^{-1}(\mathbf{x}_i-\mu_c)\right)\right) \tag{5.98}$$

The trace is invariant under cyclic permutations, so $tr(ABC)=tr(CAB)=tr(BCA)$ and we can move Σ^{-1} as I please, which I couldn't do before because matrix multiplication is not commutative.

$$L(\mu_c,\Sigma;\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\left(-\log\left((2\pi)^d|\Sigma|\right)-tr\left((\mathbf{x}_i-\mu_c)^T(\mathbf{x}_i-\mu_c)\Sigma^{-1}\right)\right) \tag{5.99}$$

$$= \frac{1}{2}\sum_{i=1}^{n}\left(-d\log 2\pi-\log(|\Sigma|)-tr\left((\mathbf{x}_i-\mu_c)^T(\mathbf{x}_i-\mu_c)\Sigma^{-1}\right)\right) \tag{5.100}$$

$$= -\frac{n}{2}\log(|\Sigma|)-\frac{nd}{2}\log 2\pi-\frac{1}{2}\sum_{i=1}^{n}\left(tr\left((\mathbf{x}_i-\mu_c)^T(\mathbf{x}_i-\mu_c)\Sigma^{-1}\right)\right) \tag{5.101}$$

$$= \frac{n}{2}\log(|\Sigma|^{-1})-\frac{nd}{2}\log 2\pi-\frac{1}{2}\sum_{i=1}^{n}\left(tr\left((\mathbf{x}_i-\mu_c)^T(\mathbf{x}_i-\mu_c)\Sigma^{-1}\right)\right) \tag{5.102}$$

$$= \frac{n}{2}\log(|\Sigma^{-1}|)-\frac{nd}{2}\log 2\pi-\frac{1}{2}\sum_{i=1}^{n}\left(tr\left((\mathbf{x}_i-\mu_c)^T(\mathbf{x}_i-\mu_c)\Sigma^{-1}\right)\right) \tag{5.103}$$

Corollary 5.3.4 We will also use the following matrix properties.

$$\frac{d|A|}{dA} = |A|A^{-T} \tag{5.104}$$
$$\frac{dtr(AB)}{dA} = \frac{dtr(BA)}{dA} = B^T \tag{5.105}$$
$$\frac{d(x^TAx)}{dA} = \frac{dtr(x^TAx)}{dA} = \frac{dtr(xx^TA)}{dA} = (xx^T)^T = xx^T \tag{5.106}$$

We can proceed computing the other derivative.

$$\frac{\partial L(\mu_c, \Sigma; \mathbf{x})}{\partial \Sigma} = \frac{n}{2}\frac{1}{\cancel{|\Sigma^{-1}|}}\cancel{|\Sigma^{-1}|}\Sigma^T - \frac{1}{2}\sum_{i=1}^{n}\left((\mathbf{x}_i - \mu_c)(\mathbf{x}_i - \mu_c)^T\right) = 0 \tag{5.107}$$
$$n\Sigma - \sum_{i=1}^{n}\left((\mathbf{x}_i - \mu_c)(\mathbf{x}_i - \mu_c)^T\right) = 0 \tag{5.108}$$
$$n\Sigma = \sum_{i=1}^{n}\left((\mathbf{x}_i - \mu_c)(\mathbf{x}_i - \mu_c)^T\right) \tag{5.109}$$
$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\left((\mathbf{x}_i - \mu_c)(\mathbf{x}_i - \mu_c)^T\right) \tag{5.110}$$
$$\tag{5.111}$$

Which can be rewritten as:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \mu_{y_i})(\mathbf{x}_i - \mu_{y_i})^T \tag{5.112}$$

Geometric interpretation

From a geometric point of view, the decision boundary turns out to be:

$$\log(\pi_0) - \frac{1}{\cancel{2}}\cancel{\log((2\pi)^d|\Sigma|)} - \frac{1}{2}(\mathbf{x} - \mu_0)^T\Sigma^{-1}(\mathbf{x} - \mu_0) = \tag{5.113}$$
$$\log(\pi_1) - \frac{1}{\cancel{2}}\cancel{\log((2\pi)^d|\Sigma|)} - \frac{1}{2}(\mathbf{x} - \mu_1)^T\Sigma^{-1}(\mathbf{x} - \mu_1) \tag{5.114}$$

But given that:

$$(\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu) = \mathbf{x}^T\Sigma^{-1}\mathbf{x} - \mathbf{x}^T\Sigma^{-1}\mu - \mu^T\Sigma^{-1}\mathbf{x} + \mu^T\Sigma^{-1}\mu \tag{5.115}$$
$$\mathbf{x}^T\Sigma^{-1}\mu = \mu^T\Sigma^{-1}\mathbf{x} \tag{5.116}$$
$$(\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu) = \mathbf{x}^T\Sigma^{-1}\mathbf{x} + \mu^T\Sigma^{-1}\mu - 2\mu^T\Sigma^{-1}\mathbf{x} \tag{5.117}$$

The final equation is the following:

$$\log(\pi_0) - \frac{1}{2}(\cancel{\mathbf{x}^T\Sigma^{-1}\mathbf{x}} + \mu_0^T\Sigma^{-1}\mu_0 - 2\mu_0^T\Sigma^{-1}\mathbf{x}) = \log(\pi_1) - \frac{1}{2}(\cancel{\mathbf{x}^T\Sigma^{-1}\mathbf{x}} + \mu_1^T\Sigma^{-1}\mu_1 - 2\mu_1^T\Sigma^{-1}\mathbf{x}) \tag{5.118}$$

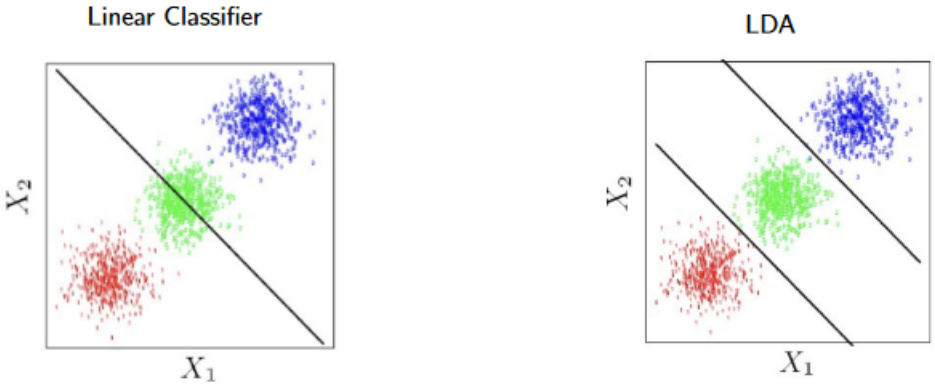
And thanks to the shared covariance matrix, many terms cancel out, we arrive to the equation of decision boundary.

$$\underbrace{\left(\log(\pi_0) - \frac{1}{2}\mu_0^T\Sigma^{-1}\mu_0 + \mu_0^T\Sigma^{-1}\mathbf{x}\right)}_{\text{Boundary for class 0}} - \underbrace{\left(\log(\pi_1) - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \mu_1^T\Sigma^{-1}\mathbf{x}\right)}_{\text{Boundary for class 1}} = 0 \tag{5.119}$$

This demonstrates that the decision boundary of LDA is linear with respect to **x**, therefore LDA is a linear classifier.

Summarizing the characteristics of LDA:

- Unlike Naive Bayes, which considered features independent, here we consider their relationships thanks to the covariance matrix.
- We have a likelihood for each class, so for each class we will have a decision boundary that separates it from the rest of the classes, thus avoiding a phenomenon called Masking, where a linear classifier could misclassify in the presence of three or more classes, for example when a third class is cut exactly in half by the decision boundary.



Trade-offs of LDA:

- Speed: slower by a factor of D compared to Naive Bayes in both training and classification due to quadratic dependence on D.
- Space: $O(D^2 + DC)$.
- Interpretability: good because μ_c are means conditioned on the class.
- Accuracy: low due to the assumptions it makes (shared covariance matrix) which will almost never be correct for real problems. What performs well is the linear decision boundary.
- Data: it needs more data than NB because it has to estimate $O(D^2)$ parameters for the covariance matrix.

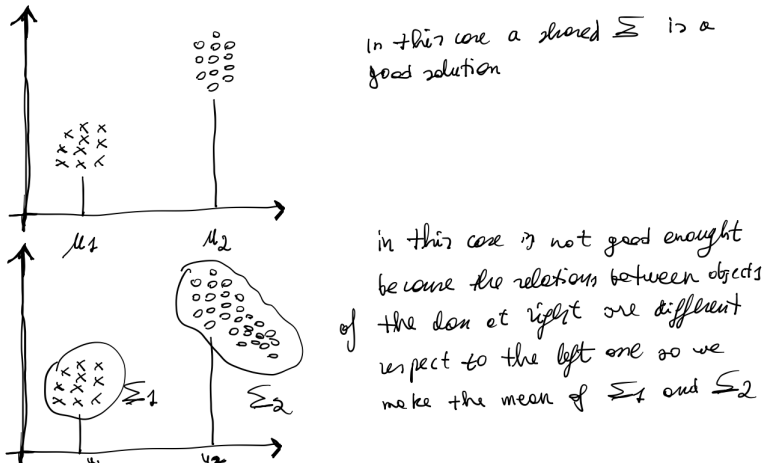


Figure 5.1: When is shared matrix a problem?

5.4 Quadratic Discriminant Analysys

Quadratic Discriminant Analysis (QDA) does not use a shared covariance matrix but has one for each class. From a geometric point of view:

$$\log(\pi_0) - \frac{1}{2}\log((2\pi)^d|\Sigma_0|) - \frac{1}{2}(\mathbf{x}-\mu_0)^T\Sigma_0^{-1}(\mathbf{x}-\mu_0) = \tag{5.120}$$

$$= \log(\pi_1) - \frac{1}{2}\log((2\pi)^d|\Sigma_1|) - \frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma_1^{-1}(\mathbf{x}-\mu_1) \tag{5.121}$$

Unlike LDA, in QDA the decision boundary is quadratic with respect to $\mathbf{x}$, leading to a result similar to the Optimal Bayes classifier when using multivariate Gaussian distributions. Therefore, the boundaries will be parabolas or hyperbolas. In contrast to LDA, QDA has $O(D^2C)$ parameters (where each class has D^2 different values). This method remains effective only when $D << N$.

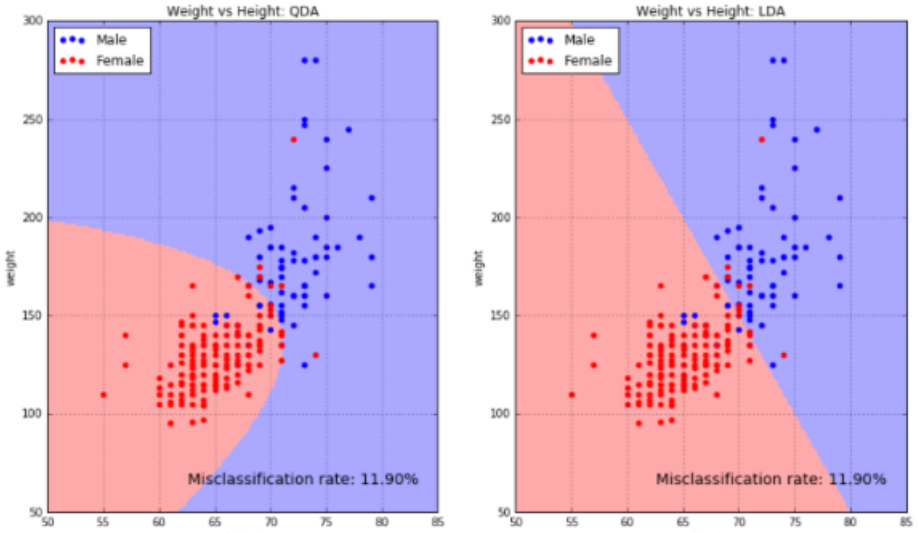


Figure 5.2: QDA vs LDA

5.5 Generative vs Discriminative Classifiers

A **generative** classifier explicitly models the joint distribution $P(X,Y)$. It's called generative because having modeled a probability distribution, we can refer to it to generate new data $\mathbf{x}$ given a certain class c using $p(\mathbf{x}|y=c)$.
LDA, Optimal Bayes, and Naive Bayes are generative classifiers as they try to model:

$$\phi_c(\mathbf{x}) \cdot \pi_c = p(X=\mathbf{x}|Y=c) \cdot P(Y=c) = p(X=\mathbf{x} \cap Y=c) = P(X,Y) \tag{5.122}$$

Using $\phi_c(\mathbf{x})$ or marginalizing $P(X,Y)$ over y, I can generate new data.
On the other hand, a discriminative classifier directly estimates the posterior probability $P(Y|X)$, ignoring the distribution of $\mathbf{x}$ and focusing only on the classes $y=c$.

5.6 Logistic Regression

Logistic Regression is a **probabilistic discriminative classifier**.

- Probabilistic because it give as output a probability that it is computed directly without the Bayes Rule.
- Discriminative because it doesn't model directly the distribution (we can't sample x) but it gives in output a score/probability of element x to belong to a certain classs

5.6.1 Binary Logistic Regression

In the binary case, it directly models the posterior probability in the following way.

$$P(Y=0|x) = \frac{e^{\mathbf{w}^T\mathbf{x}+b}}{1+e^{\mathbf{w}^T\mathbf{x}+b}} \tag{5.123}$$

$$P(Y=1|x) = \frac{1}{1+e^{\mathbf{w}^T\mathbf{x}+b}} \tag{5.124}$$

Remember that are probabilities so $P(Y=0|x)+P(Y=1|x)=1$. So given the score of beloging to class 0 we can simply compute the other score in the following way.

$$P(Y=0|x)+P(Y=1|x)=1 \tag{5.125}$$

$$P(Y=1|x)=1-P(Y=0|x) \tag{5.126}$$

$$P(Y=1|x)=1-\frac{e^{\mathbf{w}^T\mathbf{x}+b}}{1+e^{\mathbf{w}^T\mathbf{x}+b}} \tag{5.127}$$

$$P(Y=1|x)=\frac{1+\cancel{e^{\mathbf{w}^T\mathbf{x}+b}}-\cancel{e^{\mathbf{w}^T\mathbf{x}+b}}}{1+e^{\mathbf{w}^T\mathbf{x}+b}} \tag{5.128}$$

$$P(Y=1|x)=\frac{1}{1+e^{\mathbf{w}^T\mathbf{x}+b}} \tag{5.129}$$

As seen so far, we calculate the decision boundary as the set of points where $P(Y=0|x)=P(Y=1|x)$. Let's understand why is a linear decision boundary (feature of equation are all linear)

$$P(Y=0|x)=P(Y=1|x) \tag{5.130}$$

$$P(Y=0|x)-P(Y=1|x)=0 \tag{5.131}$$

$$logP(Y=0|x)-logP(Y=1|x)=0 \tag{5.132}$$

$$log\frac{P(Y=0|x)}{P(Y=1|x)}=0 \tag{5.133}$$

$$log(\frac{e^{\mathbf{w}^T\mathbf{x}+b}}{\cancel{1+e^{\mathbf{w}^T\mathbf{x}+b}}}\cdot\frac{\cancel{1+e^{\mathbf{w}^T\mathbf{x}+b}}}{1})=0 \tag{5.134}$$

$$\mathbf{w}^T\mathbf{x}+b=0 \tag{5.135}$$

$w^T\mathbf{x}+b=0$ is a linear equation so, as can be intuited, the decision boundary will be linear (a straight line).
The classification function is the following:

$$f_{LR}(\mathbf{x}) = \underset{c\in\mathscr{Y}}{argmax} P(Y=c|\mathbf{x}) \tag{5.136}$$

Therefore, to evaluate a point, I insert its coordinates into the formula $w^T\mathbf{x}+b=0$, and if it results in a positive value, it will belong to class 0; otherwise, it will belong to the class 1.

- If $P(Y=0|x)>P(Y=1|x)\rightarrow$ log ratio greater than $1\rightarrow w^T\mathbf{x}+b>0\rightarrow$ point over the line $\rightarrow$ class 0
- If $P(Y=0|x)<P(Y=1|x)\rightarrow$ log ratio less than $1\rightarrow w^T\mathbf{x}+b<0\rightarrow$ point above the line $\rightarrow$ class 1

In this formula, w (weights) and b (bias) are parameters that need to be learned during training. However, this formula provides me a score and not a probability. To convert to a probability, we use the **sigmoid** or logistic function, which is a continuous version of a threshold.

$$f(x)=\frac{1}{1+e^{-x}} \tag{5.137}$$

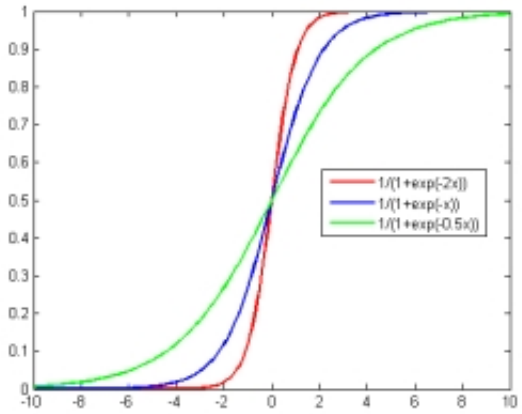


Figure 5.3: Logistic functions

5.6.2 Multiclass Logistic Regression

Logistic regression can also be extended to the multiclass case with K classes:

$$P(Y=c|x) = \frac{e^{\mathbf{w}_c^T \mathbf{x}+b_c}}{1+\sum_{l=1}^{K-1} e^{\mathbf{w}_l^T \mathbf{x}+b_l}} \tag{5.138}$$

$$P(Y=K|x) = \frac{1}{1+\sum_{l=1}^{K-1} e^{\mathbf{w}_l^T \mathbf{x}+b_l}} \tag{5.139}$$

The equation of score of the last class is different only for an implementation choice. Indeed, I need to compute only score of k-1 class because the last can be retrieve cause to the fact that probabilities always sum to 1 (REMEMBER THAT IS AN IMPLEMENTATION CHOICE, WE CAN ALSO HAVE THE LAST CLASS EQUALS TO THE OTHERS). In this way we save a w^T and b.
The classification function remains the same:

$$f_{LR}(\mathbf{x}) = \underset{c\in\mathscr{Y}}{argmax} P(Y=c|\mathbf{x}) \tag{5.140}$$

The decision boundary is piecewise linear (linear between pairs of classes).

5.6.3 Learning Optimal Parameters

In this case as well, I need to find the parameters $\theta = \{(\mathbf{w}_c, b_c), c \in \mathscr{Y}\}$ that maximize the conditional log-likelihood of the labels given the dataset $D = \{(\mathbf{x}_i, y_i), i = 1:n\}$:

$$\theta^* = \underset{\theta}{argmax} L(\theta|D) = \underset{\theta}{argmax} \sum_{i=1}^{n} logP(Y=y_i|X=\mathbf{x}_i) \tag{5.141}$$

This equation want to maximize score for the right class y_i for element x_i (only this and not also the score for wrong classes because they sum to 1).
However, $L(\theta|D)$ cannot be maximized analytically, but only using numerical optimization methods (of iterative type).

Exercise 5.3 — Binary case. However, for the binary case we can see some steps. Given 2 classes y 2 0, 1 and a dataset D by absorbing the bias:

$$P(Y=1|x) = \frac{e^{\mathbf{w}^T \mathbf{x}}}{1+e^{\mathbf{w}^T \mathbf{x}}} \qquad P(Y=0|x) = \frac{1}{1+e^{\mathbf{w}^T \mathbf{x}}} \tag{5.142}$$

Being the binary case, the likelihood can be modeled with a Bernoulli distribution as seen previously:

$$L(\theta|D) = \sum_{i=1}^{n} \Big(log\big(P(Y=1|\mathbf{x}_i)^{y_i}(1-P(Y=1|\mathbf{x}_i))^{(1-y_i)}\big)\Big) \tag{5.143}$$

$$= \sum_{i=1}^{n} \Big(y_i log P(Y=1|\mathbf{x}_i) + (1-y_i)(log P(Y=0|\mathbf{x}_i))\Big) \tag{5.144}$$

$$\tag{5.145}$$

Note how this term is equal to the negative binary cross-entropy, where y plays the role of a one-hot vector, i.e., a selector (vector of all zeros except for one one).

$$L(\theta|D) = \sum_{i=1}^{n} \Big(y_i log\big(\frac{e^{\mathbf{w}^T\mathbf{x}_i}}{1+e^{\mathbf{w}^T\mathbf{x}_i}}\big) + (1-y_i) log\big(\frac{1}{1+e^{\mathbf{w}^T\mathbf{x}}}\big)\Big) \tag{5.146}$$

$$= \sum_{i=1}^{n} \Big(y_i\big(\mathbf{l}og(e^{w^T\mathbf{x}_i} - log(1+e^{\mathbf{w}^T\mathbf{x}_i})\big) + (1-y_i)\big(log(1) - log(1+e^{\mathbf{w}^T\mathbf{x}_i})\big)\Big) \tag{5.147}$$

$$= \sum_{i=1}^{n} \Big(y_i\big(\mathbf{w}^T\mathbf{x}_i - log(1+e^{\mathbf{w}^T\mathbf{x}_i})\big) + (1-y_i)\big(0 - log(1+e^{\mathbf{w}^T\mathbf{x}_i})\big)\Big) \tag{5.148}$$

$$= \sum_{i=1}^{n} \Big(y_i\mathbf{w}^T\mathbf{x}_i - \cancel{y_i log(1+e^{\mathbf{w}^T\mathbf{x}_i})} - log(1+e^{\mathbf{w}^T\mathbf{x}_i}) + \cancel{y_i log(1+e^{\mathbf{w}^T\mathbf{x}_i})}\Big) \tag{5.149}$$

$$= \sum_{i=1}^{n} \Big(y_i\mathbf{w}^T\mathbf{x}_i - log(1+e^{\mathbf{w}^T\mathbf{x}_i})\Big) \tag{5.150}$$

In the binary case, therefore, maximizing the likelihood is equivalent to minimizing the negative cross-entropy. But as mentioned, this cannot be done analytically, so iterative methods like gradient descent must be used, which calculates the following derivatives for each x_i:

$$\frac{\partial L(\mathbf{x},\mathbf{w})}{\partial w_k} = y_i x_{ik} - \frac{1}{1+e^{\mathbf{w}^T\mathbf{x}_i}} \cdot e^{\mathbf{w}^T\mathbf{x}_i} \cdot x_{ik} \tag{5.151}$$

$$= \Big(y_i - \frac{e^{\mathbf{w}^T\mathbf{x}_i}}{1+e^{\mathbf{w}^T\mathbf{x}_i}}\Big) x_{ik} \tag{5.152}$$

Note: We find both $\mathbf{x}_i$ and x_{ik}. The first is the i-th vector of my dataset and remains because the calculation $\mathbf{w}^T\mathbf{x}_i$ must be done between all parameters and all vectors of my dataset and returns a scalar. Instead, x_{ik} (which came out by deriving the first term of the summation and deriving the exponent of the exponential) is the k-th feature of my feature vector $\mathbf{x}_i$. ■

Let's conclude this section with the trade-offs of logistic regression:

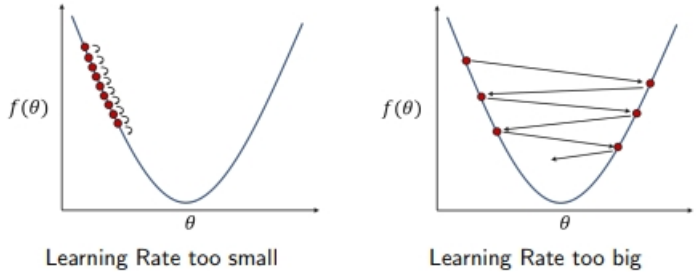
- Speed: faster than NB and LDA at classification time but slower during training due to gradient descent.
- Space: $O(DC)$ parameters (each parameter for each class takes D different values). It's equal to NB but better than LDA when $C << D$
- Interpretability: good because the importance of feature x_d is represented by its respective weight w_{dc}
- Accuracy: with high dimensions and few data, it's better than LDA. With low dimensions, lots of data, and non-linear decision boundaries, K-Nearest-Neighbors is better.

5.7 Gradient Descent

It is an iterative optimization algorithm used to find the minimum of a function by "descending" in small steps along the direction of the negative gradient of the function at a certain point. If we consider a function $f(\theta)$, the gradient descent update is expressed as:

$$\theta_j = \theta_j - \alpha \frac{\partial f(\theta)}{\partial \theta_j} \quad \forall \theta_j \tag{5.153}$$

where $\alpha \in (0,1)$ is called the learning rate and is a hyperparameter that controls the step size, i.e., it indicates how much I follow and trust the information coming from the derivative. It's essential to choose the correct learning rate. Indeed, with a too small step, convergence is reached too slowly, while with a too large step, I might go beyond the target or even diverge. In practice, the learning rate is like the accelerator of a car that follows the direction of the derivative of f.



The algorithm stops when the derivative is equal to zero, i.e., I have found a change in concavity. If the function is convex, gradient descent can find the global minimum; if it's not, it can at most converge to a local minimum.

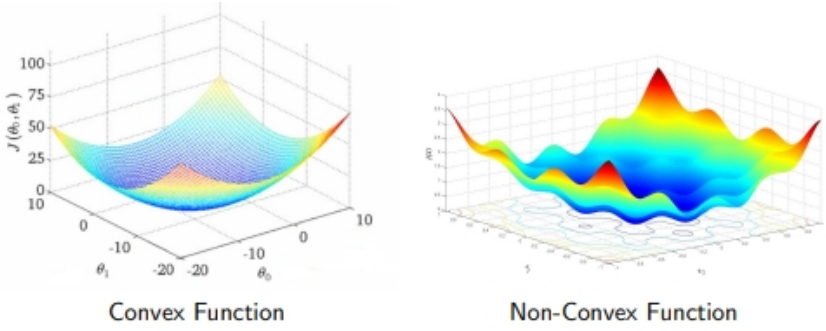


Figure 5.4: Convex and non-convex functions

Gradient descent is often used in machine learning to minimize a cost function, also called objective or loss function, indicated by $L(\cdot)$ or $J(\cdot)$. The cost function depends on the model parameters and is used to evaluate its performance. So in this context, minimizing the cost is equivalent to maximizing the effectiveness of the model.

There are different strategies for gradient descent:

- **Batch gradient descent**: to perform a single update step, I analyze the entire training dataset
- **Mini-batch stochastic gradient descent**: I analyze only a small subset
- **Stochastic gradient descent**: I analyze only a small subset taken randomly

Let's see the algorithm of Stochastic gradient descent for binary logistic regression:

Algorithm 1 Stochastic gradient descent

Require: dataset D of N elements, $y \in \{0,1\}, \alpha$

Ensure: vector of parameters $\mathbf{w}$

1: Initialize $\mathbf{w}$ randomly
2: **while** convergence reached **or** maximum iteration reached **do**
3: Take a sample $(\mathbf{x}_i, y_i)$ randomly
4: Calculate derivatives $\frac{\partial L(\mathbf{x}_i, \mathbf{w})}{\partial \mathbf{w}} = \left\{ \frac{\partial L(\mathbf{x}_i, \mathbf{w})}{\partial w_k} \right\}_{k=1,\ldots,K}$
5: Use the formula $\frac{\partial L(\mathbf{x}_i, \mathbf{w})}{\partial w_k} = \left(y_i - \frac{e^{\mathbf{w}^T \mathbf{x}_i}}{1 + e^{\mathbf{w}^T \mathbf{x}_i}} \right) x_{ik}$
6: Apply the update rule $\mathbf{w}^{new} = \mathbf{w} - \alpha \frac{\partial L(\mathbf{x}_i, \mathbf{w})}{\partial w}$
7: $\mathbf{w} = \mathbf{w}^{new}$
8: **end while**

6. Support Vector Machines

Support Vector Machines (SVM) are discriminative classifier used in various case from binary classification to regression. Let's go more in detail.

6.1 Linear SVM

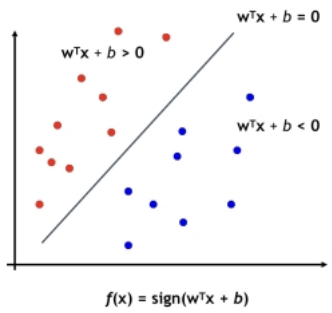
Consider a set of point linearly separable and a binary classification problem:

$$(x_i, y_i) \quad x_i \in \mathbf{R}^n, y_i \in \{-1, 1\} \quad with \quad i = 1, \ldots, N \tag{6.1}$$

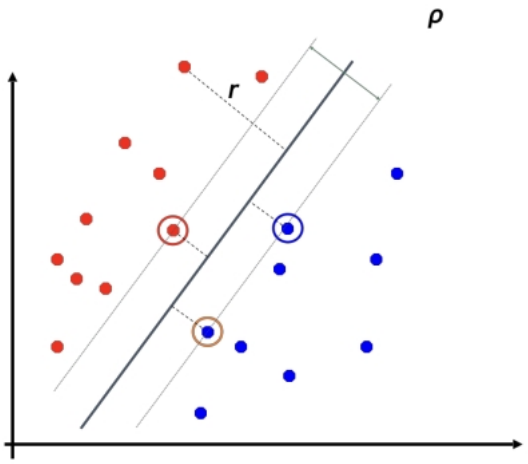
The goal is to find a separating hyperplane $w^T x + b$ able to classify in the right way points in the space. All the possible hyperplanes are those solve the follow inequalities:

$$y_i(w * x_i + b) \geq 0 \quad i = 1, \ldots, N \rightarrow \begin{cases} y_i = 1, & \text{if } w * x_i + b > 0 \\ y = -1, & \text{if } w * x_i + b > 0 \end{cases} \tag{6.2}$$

Instead of zero we can use a generic constant c. An useful implemented choice is to select c = 1 so, appropriately scaling the values of w^T and b the point closest to hyperplane solve the following equation $y_i(w * x_i + b) = 1$.
So to classify a new element we need just to compute a sign of $w^T x + b$.



Now the question is, between all the possible hyperplanes, which of the these is optimal one? SVM identify as optimal hyperplane the one that maximize the distance between the closest points of two classes and itself; this because greater is the margin and greater will be the distance between classes, reducing the chance of confusing (we aim at the most confident boundary). The distance between



x_i and the boundary $w^T x + b$ is the following (formula of distance between point and hyperplane):

$$r_i = \frac{w^T x_i + b}{||w||} \tag{6.3}$$

The objective is to maximize this value for all x_i. The width of the margin is denoted by ρ. Given a dataset $D = (x_i, y_i)_{i \in [1...N]}$ and $y_i \in \{+1, -1\}$, we want:

$$\begin{cases} w^T x_i + b \geq \rho/2 & \text{if and only if } y_i = +1 \\ w^T x_i + b < \rho/2 & \text{if and only if } y_i = -1 \end{cases} \tag{6.4}$$

In other words, we want all points above and below the line to be more than $\rho/2$ in absolute value (we don't want points inside the margin). This can all be compressed into a single formula:

$$y_i(w^T x_i + b) \geq \rho/2 \ , \ \forall x_i \in D \tag{6.5}$$

Consequently, absorbing $\rho/2$ into w:

$$r_s = \frac{w^T x_s + b}{\|w\|} = \frac{\rho}{2\|w\|} = \frac{1}{\|\hat{w}\|} \quad with \quad \hat{w} = \frac{w}{\rho/2} \tag{6.6}$$

Learning w or learning $\hat{w}$ is the same for us because we would divide all parameters by the same quantity $\rho/2$. The margin will therefore be twice r_s:

$$\rho = 2r_s = \frac{2}{\|\hat{w}\|} \tag{6.7}$$

Corollary 6.1.1 — SVM Maximization formula. The SVM problem formulation becomes:

$$\arg\max_{\hat{w},\hat{b}} \frac{2}{\|\hat{w}\|} \text{ such that} \tag{6.8}$$

$$y_i(\hat{w}^T x_i + \hat{b}) \geq 1 \ \forall (x_i, y_i) \in D \tag{6.9}$$

This last inequality derives from the absorption of $\rho/2$ into $\hat{w}$ and $\hat{b}$, indeed:

$$y_i(w^T x_i + b) \geq \rho/2 \tag{6.10}$$

$$y_i(\frac{w^T}{\rho/2} x_i + \frac{b}{\rho/2}) \geq 1 \tag{6.11}$$

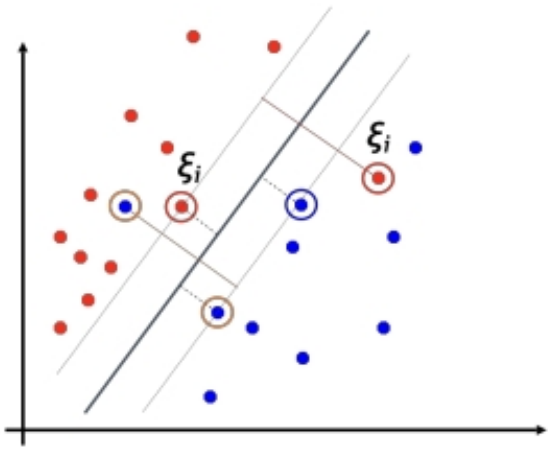
$$y_i(\hat{w}^T x_i + \hat{b}) \geq 1 \tag{6.12}$$

Corollary 6.1.2 — SVM Minimization formula. Maximize that distance is equal to minimize the norm, so we can rewrite the problem in the following way:

$$\arg\min_{w,b} \|w\|^2 \text{ such that} \tag{6.13}$$

$$y_i(w^T x_i + b) \geq 1 \; \forall (x_i, y_i) \in D \tag{6.14}$$

For simplicity, we will return to denoting $\hat{w}$ and $\hat{b}$ as w and b.
Defined the problem is easy to notice that SVM is a method of constrained optimization due to the fact that we have complex and difficult constraint and why we need to redefine the problem in a different way in order to resolve it.
Before that we extend the formulation to all type of sets (non linearly separable also). To do that we need to correct the constraint in order to make them less rigid and to do that we introduce the following slack variables $\xi = (\xi_i, \ldots, \xi_n), \xi_i \geq 0$, indicating the degree of violation of the constraint.



From the image above we can see that the methodology depends only on some example of the training set. Indeed, the definition of hyperplane depends only on the examples inside the margin (that solve $y_i(w * x_i + b) \leq 1$, that are called **Support Vector(SV)**. Then SV can be classified in two subcategories:

- **Bound Support Vector**: all the examples x_i such that $y_i(w * x_i + b) < 1$
- **Non-Bound Support Vector**: all the examples x_i such that $y_i(w * x_i + b) = 1$

So SVM for classification, find the most difficult example to classify (SV) and consider only those in the definition of hyperplane.

Corollary 6.1.3 — SVM General formula.

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i \quad \text{such that} \tag{6.15}$$

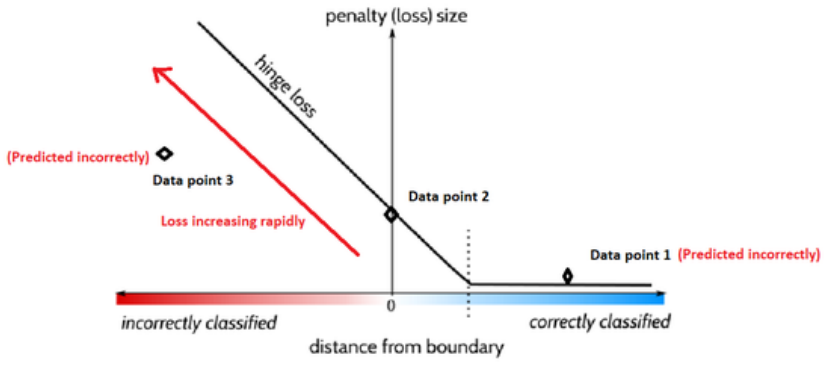
$$y_i(w^T x_i + b) \geq 1 - \xi_i \tag{6.16}$$

We introduce a new term in the objective function $C\sum_{i=1}^{N}\xi_i$ to the aim of minimize the number of points classified in the wrong way, so that fall in the margin (we want to maximize the number of $x_i = 0$. Hyperparametrs C is a trade-off between how much I want to generalize versus how much I want to be faithful to my dataset.

- With a high C, the second term (total penalty) prevails, so to minimize, I will have to focus on not violating the constraints (i.e., minimizing the translations introduced by my slack variables). Then makes the model good on my specific dataset, as I'm giving importance to the constraints that are tied to my data.
- With a low C, the first term will prevail, so I focus on maximizing the margin, and in doing so, I can also have large slack variables that could move points already outside the margin just to then maximize the latter even more and generalize better. This is because the sum of the penalties is multiplied by a low value and therefore gives me this freedom.

As loss functions we use **soft margin** loss function:

$$V(y, f(x)) = max(0, 1 - yf(x)) = \begin{cases} 0 & if \ \ yf(x) \geq 1 \\ 1 - yf(x) & if \ \ yf(x) \leq 1 \end{cases} \tag{6.17}$$



6.1.1 Constrained Optimization

Our problem is a constrained optimization issue, and in this type of problem, we need to integrate the constraints into the function we wish to optimize. For instance, consider a minimization problem of this nature:

$$\min f(x) \text{ subject to constraints} \tag{6.18}$$

$$h_k(x) \geq 0 \, k \in \{1, \ldots, K\} \tag{6.19}$$

To incorporate the constraints h_k into f we can use Lagrange function.

Corollary 6.1.4 — Langrange Function.

$$L(x,\alpha) = f(x) - \sum_{i=1}^{K} \alpha_i h_i(x) \tag{6.20}$$

$$\text{with } \alpha_i \geq 0 \text{ called Lagrange multipliers.} \tag{6.21}$$

Lagrange multipliers are vectors that connect gradients of constraints with the gradient of objective function (in points of local minimum these two gradients are proportional between them). We then move to a constraint in the form $h_k(x) \geq 0$:

$$y_i(w^T x_i + b) \geq 1 \tag{6.22}$$

$$y_i(w^T x_i + b) - 1 \geq 0 \tag{6.23}$$

We have two cases:

- $h_i(x_i) > 0$ then $\alpha_i = 0$ (inactive constraint)
- $h_i(x_i) = 0$ then $\alpha_i > 0$ (active constraint)

The constraint is inactive (we don't need it, so $\alpha_i = 0$) if the point x_i is in an allowed zone. Instead, it is active only if the point in question is an SV, so this time the value of $\alpha_i > 0$ interests us to calculate the separation line.
Being a problem of minimizing a function, we need to calculate its derivative and set it equal to zero. Returning to the previous cases, we will have:

- For inactive constraints

$$\nabla L(x,\alpha) = \nabla f(x) \tag{6.24}$$

- For active constraints

$$\nabla L(x,\alpha) = \nabla f(x) - \alpha_i \frac{dh_i(x)}{dx} \tag{6.25}$$

Therefore:

- If $\alpha_i = 0 \; \forall i$ the solution is:

$$x^* | \nabla f(x) = 0 \tag{6.26}$$

- If $\exists \alpha_i \neq 0$ the solution is:

$$x^* | \nabla f(x) - \alpha^T \nabla h(x) = 0 \tag{6.27}$$

$$x^* | \nabla f(x) - \sum_{i=1}^{K} \alpha_i \frac{dh_i(x)}{dx} = 0 \tag{6.28}$$

Karush-Kuhn-Tucker Conditions

The KKT conditions bring together everything we've seen so far and are necessary but not sufficient conditions for the existence of a unique solution x^*:

$$\nabla f(x^*) - \alpha^T \nabla h(x^*) = 0 \tag{6.29}$$

$$h(x^*) \geq 0 \text{ compliance with constraints} \tag{6.30}$$

$$\alpha_i \geq 0 \text{ non-negativity of multipliers} \tag{6.31}$$

$$\alpha^T h(x) = 0 \tag{6.32}$$

The last condition, $\alpha^T h(x) = 0$, is known as the **complementarity** condition. It means that we can never simultaneously have $\alpha_i^* \neq 0$ and $h_i(x^*) \neq 0$, as the multiplier of an inactive constraint must be zero.
In other words:

- If $h_i(x^*) > 0$ (inactive constraint), then $\alpha_i^* = 0$
- If $\alpha_i^* > 0$, then $h_i(x^*) = 0$ (active constraint)

The KKT become necessary and sufficient conditions for the existence of a unique solution x^* if:

- h are affine functions (essentially linear)
- f is a convex function
- f and h are smooth functions (infinitely differentiable and the derivatives are continuous)

Regarding the constraints h, they verify both points as they are simple scalar products, so they are both affine and smooth. As for f, since we have squared the norm of w, it also verifies both points as it is both convex and smooth.

Wolfe's Dual Formulation

Another tool to resolve constrained optimization is use Wolfe's dual formulation.

Theorem 6.1.5 — Wolfe Formulation. Having a primal minimization problema as the following one.

$$\min \ f(x) \tag{6.33}$$
$$\text{subject to } \ g(x) \leq 0 \tag{6.34}$$
$$\text{with } \ f, g \ \ \textit{convex function} \tag{6.35}$$

We can rewrite it with the use of Lagrange function in the following way

$$\max \ L(x, \mu) \tag{6.36}$$
$$\text{subject to } \ \nabla_x L(x, \mu) = 0 \ \ \mu \geq 0 \tag{6.37}$$
$$\tag{6.38}$$

Both problems have, in the optimal solution (x_i^*, μ_i^*), the same value of objective function, with optimal solution that solve KKT condition of primal formulation.
Now returning to our initial problem:

$$\min_{w,b} \|w\|^2 \text{ such that} \tag{6.39}$$
$$y_i(w^T x_i + b) - 1 \geq 0 \, \forall (x_i, y_i) \in D \tag{6.40}$$

Let's rewrite it in terms of Lagrangian considering a dataset D of N elements:

$$\min_{w,b,\alpha} L(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N} \alpha_i \underbrace{(y_i(w^T x_i + b) - 1)}_{\text{constraint } h_i(x)} \text{ such that (KKT):} \tag{6.41}$$
$$y_i(w^T x_i + b) - 1 \geq 0 \tag{6.42}$$
$$\alpha_i \geq 0 \tag{6.43}$$
$$\alpha_i(y_i(w^T x_i + b) - 1) = 0 \tag{6.44}$$

A $\frac{1}{2}$ appears before $\|w\|^2$ because it will simplify the calculation of its derivative. To find the parameters that minimize the Lagrangian, we need to calculate the derivatives and set them equal to zero.

$$\frac{\partial L(w, b, \alpha)}{\partial b} = \sum_{i=1}^{N} \alpha_i y_i = 0 \tag{6.45}$$
$$\frac{\partial L(w, b, \alpha)}{\partial w} = w - \sum_{i=1}^{N} \alpha_i y_i x_i = 0 \tag{6.46}$$

Consequently:

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i \tag{6.47}$$

This derivative (which is then the relationship between the dual and the primal) is very important because it relates the slope of our original line w to the Lagrange multipliers α. It underlines how the points that satisfy the constraint (are far from the separation surface) do not influence the calculation of w ($\alpha = 0$) while those that are needed are those with $\alpha > 0$, i.e., the SVs. Moreover, substituting these first two derivatives into the Lagrangian, we get something that depends only on α, which will be useful for calculating the last derivative (with respect to α indeed):

$$L(w,b,\alpha) = \frac{1}{2}\Big\| \sum_{i=1}^{N} \alpha_i y_i x_i \Big\|^2 - \sum_{i=1}^{N} \alpha_i \Big(y_i \big((\sum_{j=1}^{N} \alpha_j y_j x_j) x_i + b \big) - 1 \Big) \tag{6.48}$$

$$= \frac{1}{2}\Big(\sum_{i=1}^{N} \alpha_i y_i x_i \Big) \Big(\sum_{j=1}^{N} \alpha_j y_j x_j \Big) - \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) - \sum_{i=1}^{N} \alpha_i y_i b + \sum_{i=1}^{N} \alpha_i \tag{6.49}$$

$$= \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) - \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) - b \underbrace{\sum_{i=1}^{N} \alpha_i y_i}_{0} + \sum_{i=1}^{N} \alpha_i \tag{6.50}$$

$$= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \tag{6.51}$$

Now applying the Wolfe's dual theorem, we arrive to the final formulation:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \text{ such that} \tag{6.52}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{6.53}$$

To understand why this is the dual problem, consider that the initial problem was to find the minimum of a parabola with the concavity facing upwards, while the dual is represented by a parabola with concavity downwards, so we need to find the maximum. If the KKT conditions are satisfied, the maximum of the dual equals the minimum of the primal, and these two points coincide and represent the same solution. Wolfe's formulation will be solved by calculating the derivative with respect to α and setting it equal to zero.

SVM Classification Function Forms

In summary, the SVM classification function can be written in two forms:

- Primal:

$$f(x) = w^T x \tag{6.54}$$

  In training, we learn w and b (which we have absorbed into w by adding a dimension). We classify the new data point x by substituting its value into the equation. Depending on whether the result is positive or negative, we assign it to one class or the other. This version is the most efficient because it is a simple row-by-column product.

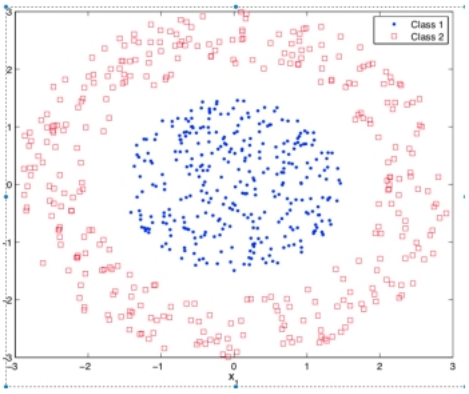
- Dual:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i x_i^T x \tag{6.55}$$

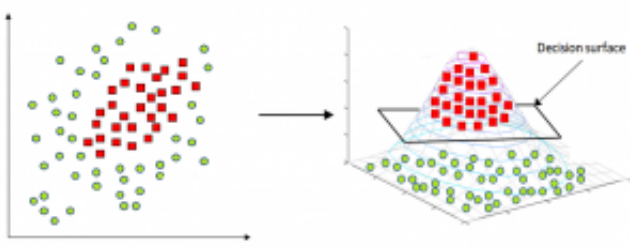
Where the first three terms of the summation (α_i, y_i, and x_i) come from the training dataset, while x is the new data point we want to classify. We classify the new data point x by substituting its value into the equation. Depending on whether the result is positive or negative, we assign it to one class or the other. This version is less efficient (because we need to store all the x_i, their corresponding α_i, and y_i) but it is the most commonly used. The reason for its popularity is related to the presence of a scalar product, which is useful for adding features to the SVM.

6.2 Non-Linear SVM

If the data are not linearly separable, meaning no straight line can separate two distinct classes of points, we need to change the classification space to one where the boundary between the two classes is linear. To do this, we use a non-linear function of the type:



$$\phi : x \rightarrow \phi(x) \tag{6.56}$$



Recalling the formulas seen previously:

$$L(\alpha) = \max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \tag{6.57}$$

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i (x_i^T x) \tag{6.58}$$

Since x appears only in a dot product, if we define the following quantity called the Kernel function:

$$\phi(x_i)^T \phi(x_j) = K(x_i, x_j) \tag{6.59}$$

The kernel function, given two points, returns the value of the dot product of the two points performed in the transformed space. Applying the mapping, we get:

$$L(\alpha) = \max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{6.60}$$

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i K(x_i, x) \tag{6.61}$$

The kernel function implicitly maps our data into a higher-dimensional space without the need to explicitly calculate $\phi(x)$. This is called the kernel trick and allows us to calculate the dot product between two data points in the transformed space without knowing their coordinates in this new space.

Theorem 6.2.1 — Mercer. Every symmetric positive semidefinite function is a kernel.

In our case, the matrix associated with this function will therefore be a Gram matrix.

Corollary 6.2.2 — Gram matrix. he Gram matrix of a set of vectors in a vector space equipped with a dot product is a matrix whose elements are the dot products between the vectors.

$$G(x_1, \ldots, x_n) = \begin{pmatrix} K(x_1, x_1) & \ldots & K(x_1, x_n) \\ \vdots & \ddots & \vdots \\ K(x_n, x_1) & \ldots & K(x_n, x_n) \end{pmatrix} \tag{6.62}$$

Note that the kernel trick can only be done between two vectors belonging to the same vector space, which is why we previously said that the dot product of the dual form $x^T x$ would be useful in adding features to SVM. I could not have applied the kernel trick in the primal form because I had a dot product between elements belonging to different vector spaces $w^T x$.
I can therefore choose any function as long as it is symmetric and positive semidefinite. Popular kernels are the followings:

- Linear, i.e., ϕ = identity function.

$$K(x, x_i) = x^T x_i \tag{6.63}$$

- Polynomial, maps to $\binom{d+p}{d}$ dimensions where every term of of polynom become a features(p hyperparameter).

$$K(x, x_i) = (1 + x^T x_i)^p \tag{6.64}$$

- Gaussian or RBF (Radial Basis Function), maps to an infinite-dimensional space (infinite sum = infinite features).

$$K(x, x_i) = \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2}\right) \tag{6.65}$$

  In an infinite-dimensional space, I will certainly have a linear function that separates my points. Through σ, I decide the bandwidth of the kernel, and if this is small, I will be more sensitive to small differences (microscopic part), while if it is large, only to large ones (macroscopic part).

Exercise 6.1 — Gaussian Kernel proof (not required). The Gaussian kernel is the normalized version of the kernel

$$\exp\left(\frac{x^T x_i}{\sigma^2}\right) \tag{6.66}$$

To prove this, we first need to define the normalized kernel:

$$K'(x_1,x_2) = \begin{cases} 0 & \text{if } K(x_1,x_1) = 0 \vee K(x_2,x_2) = 0 \\ \frac{K(x_1,x_2)}{\sqrt{K(x_1,x_1)K(x_2,x_2)}} & \text{otherwise} \end{cases}, \forall x_1,x_2 \in D \tag{6.67}$$

Therefore, in the case of $\exp\left(\frac{x^T x_i}{\sigma^2}\right)$:

$$K'(x,x_i) = \frac{\exp\left(\frac{x^T x_i}{\sigma^2}\right)}{\sqrt{\exp\left(\frac{x^T x}{\sigma^2}\right)\exp\left(\frac{x_i^T x_i}{\sigma^2}\right)}} \tag{6.68}$$

$$= \frac{\exp\left(\frac{x^T x_i}{\sigma^2}\right)}{\exp\left(\frac{x^T x}{2\sigma^2}\right)\exp\left(\frac{x_i^T x_i}{2\sigma^2}\right)} \tag{6.69}$$

$$= \frac{\exp\left(\frac{x^T x_i}{\sigma^2}\right)}{\exp\left(\frac{x^T x}{2\sigma^2} + \frac{x_i^T x_i}{2\sigma^2}\right)} \tag{6.70}$$

$$= \frac{\exp\left(\frac{x^T x_i}{\sigma^2}\right)}{\exp\left(\frac{\|x\|^2 + \|x_i\|^2}{2\sigma^2}\right)} \tag{6.71}$$

$$= \exp\left(\frac{x^T x_i}{\sigma^2} - \frac{\|x\|^2 + \|x_i\|^2}{2\sigma^2}\right) \tag{6.72}$$

$$= \exp\left(\frac{2x^T x_i - \|x\|^2 - \|x_i\|^2}{2\sigma^2}\right) \tag{6.73}$$

$$= \exp\left(-\frac{\|x\|^2 - 2x^T x_i + \|x_i\|^2}{2\sigma^2}\right) \tag{6.74}$$

$$= \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2}\right) \tag{6.75}$$

$$\tag{6.76}$$

■

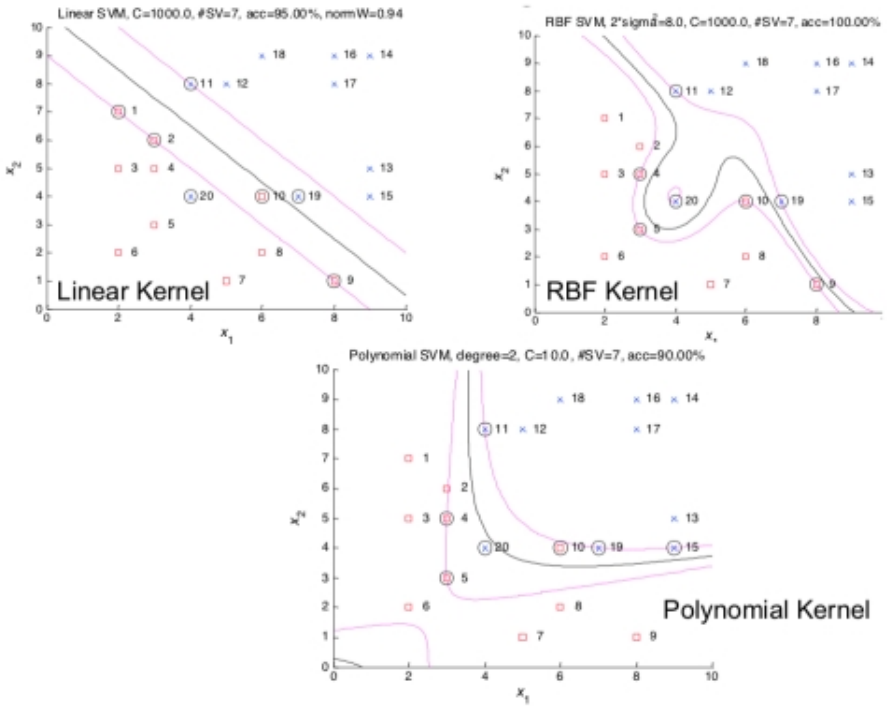
Exercise 6.2 — Gaussian kernel has infinite dimensions (not required). To prove that the target space of a Gaussian kernel has infinite dimensions, we use the relation:

$$e^x = \sum_{i=1}^{\infty} \frac{x^n}{n!} \tag{6.77}$$

It follows that:

$$\exp\left(\frac{x^T x_i}{\sigma^2}\right) = \sum_{i=1}^{\infty} \frac{(x^T x)^n}{\sigma^{2n} n!} \tag{6.78}$$

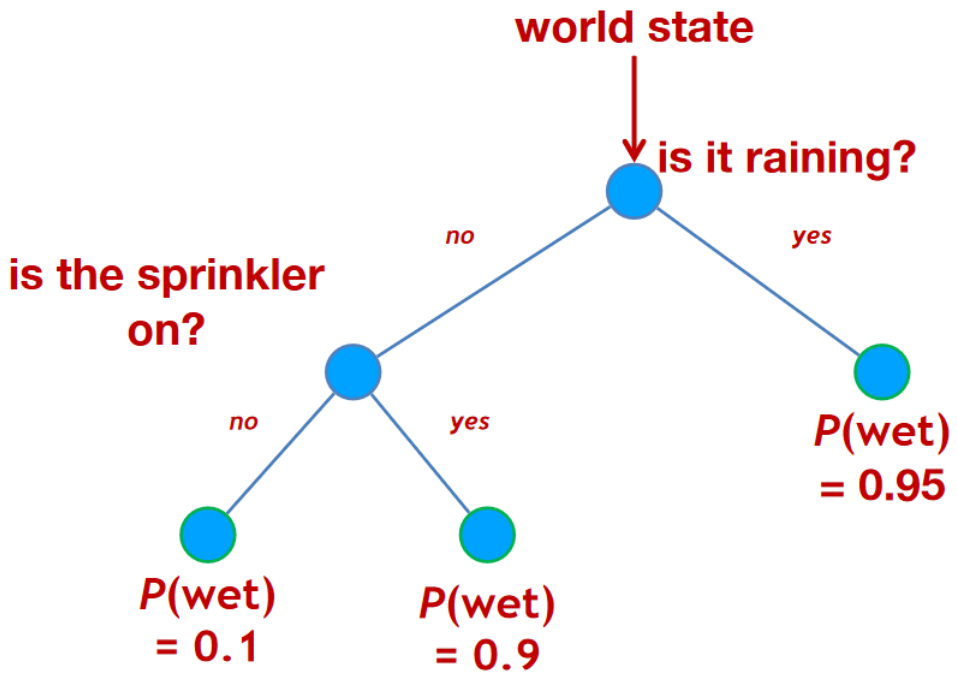
This shows how the Gaussian kernel is nothing more than a linear combination of polynomial kernels of increasing degree (up to ∞), so the dimension is $\binom{d+\infty}{d} = \infty$ ■



In conclusion, kernels can only be used in the dual version of SVM because we work in a transformed space whose mapping function we do not know, and because they are based on the dot product $x^T x$, which is present only in the dual form. One drawback is that every time we need to classify a new point, we have to calculate a kernel for each support vector. A final drawback is that being able to make even very complex separation surfaces (in the starting space), we are sensitive to the specificity of our dataset, "learning it by heart" (overfitting).

7. Decision Trees

Decision Trees are diagrams that represent a complex problem as sequences of binary decisions and are particularly formed by various nodes, the first of which is called the root and represents the observation of the problem. Each node poses a question and allows a binary split. The objective is to have a model that, through the questions, partitions the data, allowing us to find the solution. The interesting thing about this type of model is that they are completely interpretable: if you arrive at a particular leaf, by tracing the path back to the root (backtracking), it is possible to reconstruct the reasoning made by the classifier to establish how that particular solution was reached.



Going into more detail, in this model we have the following variables:

- **Data**, represented by feature vectors

$$\mathbf{v} \in \mathbb{R}^n \tag{7.1}$$

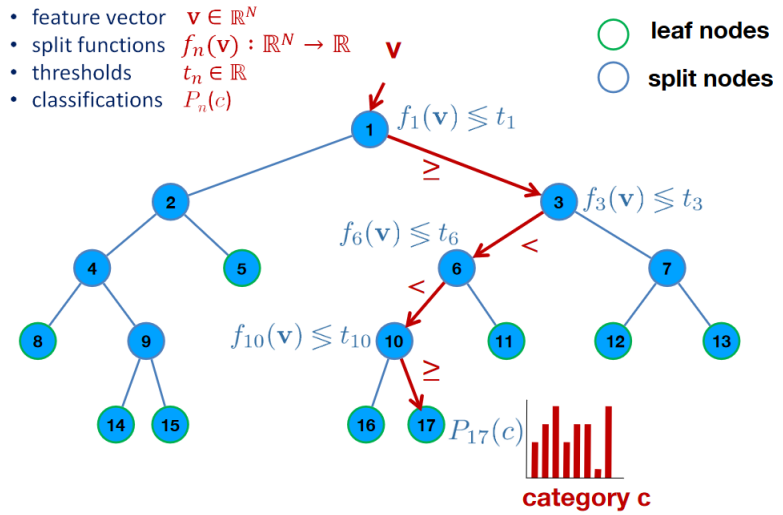
- **Questions**, represented by split functions

$$f_n(v) : \mathbb{R}^n \to \mathbb{R} \tag{7.2}$$

- **Thresholds**, the equivalent of "yes or no" is represented by thresholds

$$t_n \in \mathbb{R} \tag{7.3}$$
$$f_n(x) \lesseqgtr t_n \tag{7.4}$$

In the leaves, there will be a histogram where on the x-axis we have all possible classes and on the y-axis we have the number of training elements that have followed that particular path ending up in that leaf with that class value.



This information is used at the test level, establishing the probability of the class given the element.

$$p(y|x) \tag{7.5}$$

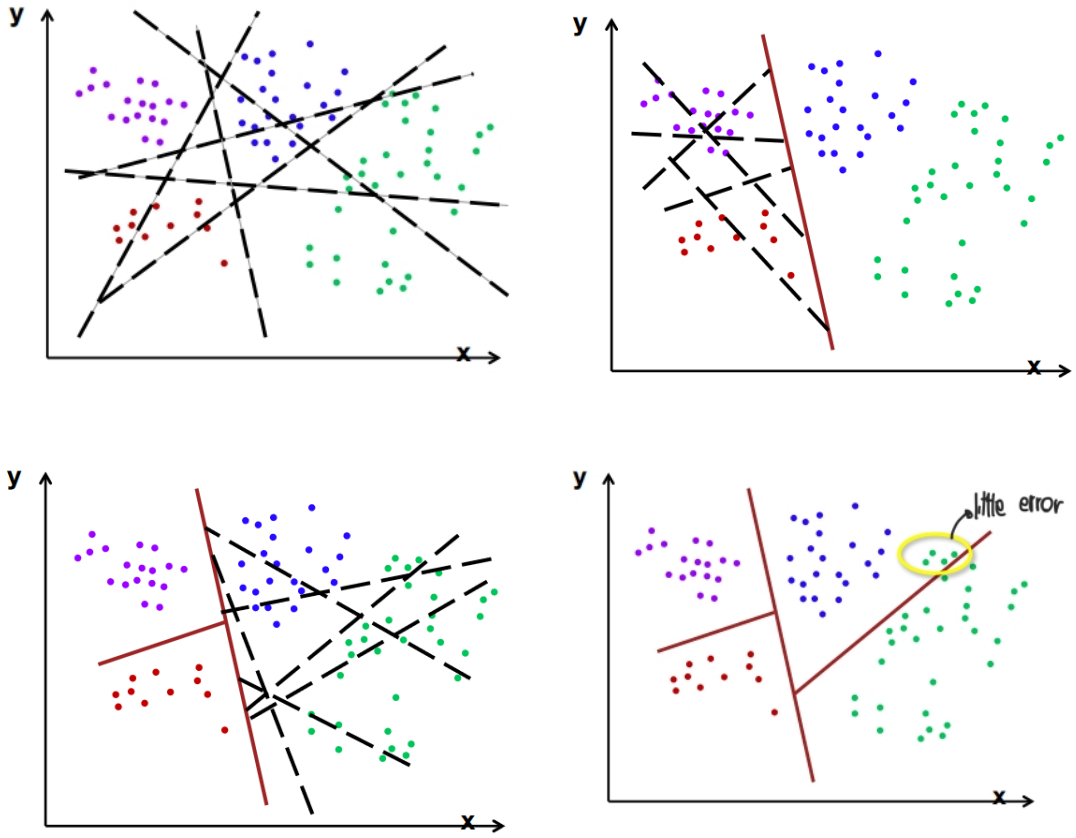
Let's see the recursive function that implements this:

```
double[] ClassifyDT(node,v){
    if (node.IsSplitNode()){
        if(node.f(v) >= node.t){
            return ClassifyDT(node.right,v)
        }
        else{
            return ClassifyDT(node.left,v)
        }
    }
    else{
        return node.P
    }
}
```

In the training phase, we need to learn the decision functions and thresholds which are the parameters. There are two ways to model the decision function f:

- All feature based, takes as input the entire feature vector, performs a calculation with $w^T \mathbf{v}$ and returns a scalar to compare with the threshold.
- **One feature based**, takes as input only one feature of the feature vector and compares it with the threshold.

Decision functions are typically linear and the first approach is to recursively try different lines and keep the one that best separates the data by maximizing the information gain.



It can happen that some points are on the wrong side of the line but this is not a problem because in the leaf I have the histogram that counts how many elements that follow a specific path of the tree belong to each class. For example, if I end up in the leaf in the upper right, I will have a high probability of belonging to the blue class and a very low probability of belonging to the green one. Let I_n be the set of indices relating to the labeled examples $(\mathbf{v}_i, I_i)$ that end up in node n, recursively at each node the data are separated in this way:

- Left split

$$I_l = \{i \in I_n | f(\mathbf{v}_i) < t\} \tag{7.6}$$

- Right split

$$I_r = I_n \backslash I_l \tag{7.7}$$

The thresholds are calculated in the range

$$t \in (min_i f(\mathbf{v}_i), max_i(\mathbf{v}_i)) \tag{7.8}$$

We then choose f and t that maximize the information gain (Maximizing the information gain therefore means minimizing the entropy of a split)

$$\Delta E = -\frac{|I_l|}{|I_n|}E(I_l) - \frac{|I_r|}{|I_n|}E(I_r) \tag{7.9}$$

Where E is the entropy of the labels, i.e., the uncertainty (diversity) calculated on the histogram built on the basis of the labels I. The entropy is calculated as follows:

$$E = -\sum_{i}^{nlabel} h_i log_2(h_i) \tag{7.10}$$

where h_i is the value of the histogram. E has a high value if all elements of the histogram have the same value (maximum disorder), i.e., the same probability, while it is 0 if only one element has probability 1 and all others zero (maximum purity = optimal split)

Speaking of implementation details, we can say that the number of features and thresholds can be the followings:

- only one (extremely randomized)
- few (fast training but can underfit)
- many (slow training and can overfit)

When to stop expanding the tree?

- manually setting a maximum depth
- fixing a minimum entropy gain beyond which to stop (under a certain threshold probably we will split instances from the same class)
- pruning, i.e., pruning branches that will certainly not do better than the upper ones (that don't change accuracy of classifier)

To sum up, characteristics of binary decision tree are the followings:

1. Fast greedy training phase
2. Fast test phase (just choose one feature and compare its values with thresholds)
3. Attention must be paid to the choice of parameters (maximum depth and number of features and thresholds)
4. Tends to overfit, especially for example when there are too many splits (extreme case: one leaf for each instance)

8. Ensemble Methods

Ensemble methods are techniques for assembling or combining multiple classifiers to achieve better performance compared to a single classifier. It is a set of models (which could be different versions of the same model or even different models) trained to solve the same task. The final output will be the weighted average or majority vote of the predictions of the individual models. Results in practice demonstrate that an ensemble almost always performs better than the individual models taken separately.

Suppose we have a set of binary classification functions $f_k(x)$, $k = 1, \ldots, K$ and that on average these have the same expected value of error (better than random choice 0.5)

$$\varepsilon = \mathbb{E}_{p(x,y)}[y \neq f_k(x)] < 0.5 \tag{8.1}$$

Let's also assume that the errors are independent, meaning there are no relationships between the errors of two different classifiers, i.e., if one classifier makes an error, the others will not make that same error (fundamental assumption of ensemble). The intuition is that the majority of the K classifiers in the ensemble will be correct on many examples where any individual classifier makes an error. And so a simple majority vote can significantly improve classification performance by decreasing variance in this setting. Let's prove this last statement.

Suppose we have K binary classifiers with accuracy $\alpha > 0.5$. The ensemble makes an error with majority voting when more than half of the classifiers $(\frac{K}{2} + 1)$ make an error. The error follows the cumulative binomial distribution:

$$P(x \leq k) = \sum_{i=0}^{k} \binom{K}{i} \alpha^i (1-\alpha)^{(K-i)} \tag{8.2}$$

This is the probability of having a certain number of events less than or equal to k out of K trials. In our case, it is the probability of having the number of correct classifications less than or equal to a certain number k.

So if I have 5 classifiers with accuracy 0.75, the probability that the ensemble is wrong is:

$$P(x \leq 2) = \sum_{i=0}^{k} \binom{5}{i} 0.75^i (0.25)^{(5-i)} = 0.105 \tag{8.3}$$

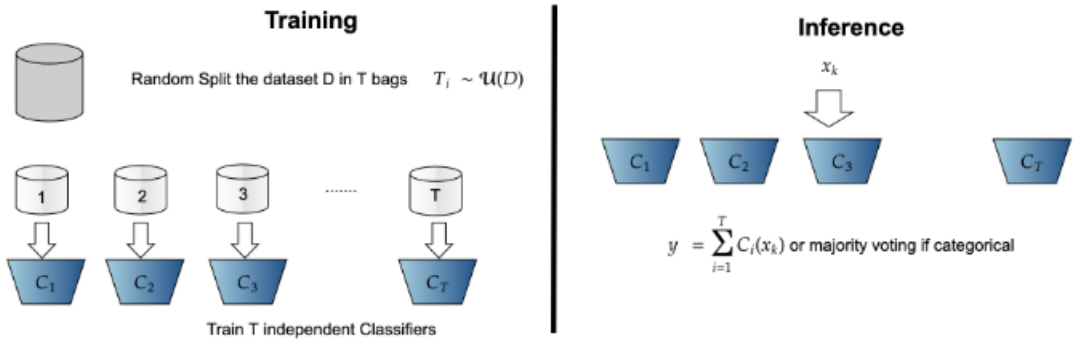
The error is indeed the probability of having the number of correct classifiers less than or equal to half of the classifiers (rounding down to the nearest integer). The accuracy of the ensemble will be

$$1 - P(x \leq 2) = 0.895 > 0.75 \tag{8.4}$$

The problem lies in having independence between errors. To guarantee this, I should train the different instances of classifiers of the same type with independent training sets, thus obtaining different classification functions. The problem is that it is too costly to have such a quantity of training sets. There are different techniques to overcome this problem, let's go more in detail.

8.1 Bagging

This method, also called Bootstrap aggregation (or **B**ootstrap **AGG**regat**ING**), consists of starting from a single training set Tr and forming from it K subsets $Tr_1, \ldots, Tr_K$ called **bags**, consisting of elements randomly sampled according to a uniform distribution (each element has the same probability of being taken) from the initial set with the possibility of repetition.
On each of these sets, an instance of the same type of classifier will be trained, obtaining K different classification functions. The errors will not be totally independent because the datasets are not totally independent, in fact there may be elements present in more than one group. However, the composition of each individual dataset has no statistical correlation with the others, as the probability of an element ending up in a certain subgroup is always the same.
In the inference phase, majority voting is used.



Computational performance wil be worse due to the training phase but at the end will have better accuracy. Let's compare the two different cases, namely not doing ensemble and doing ensemble, particularly bagging.
Suppose initially we don't do ensemble, in this case each single classifier learns a classification function.

$$y = f_i(x) + e_i \tag{8.5}$$

where e_i is the error of the i-th classifier. The average error of the single classifier over the entire training set is $\mathbb{E}_D[e_i]$. The average error of M classifiers taken individually on the dataset D is:

$$\mathscr{E}_{AV} = \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}_D[e_i] \tag{8.6}$$

It's an average of the average errors of the individual classifiers on the dataset, i.e., an average of the accuracies of the individual classifiers (in this section accuracy and error will be used interchangeably knowing that $acc = 1 - err$).
This quantity does not depend on which element I'm wrong about, i.e., I'm not taking into account that for a specific element of the dataset maybe more than half of the M classifiers classified well.
Now suppose we use bagging, in this case the classification function will be (majority vote)

$$y_{bagged} = \frac{1}{M}\sum_{i=1}^{M} f_i(x) + e_i(x) \tag{8.7}$$

where $e_i(x)$ is the error of the i-th classifier committed on the single element x. The average error of the ensemble of M classifiers on the dataset D using bagging is:

$$\mathscr{E} = \mathbb{E}_D\Big[\frac{1}{M}\sum_{i=1}^{M} e_i(x)\Big] \tag{8.8}$$

The argument of the expected value is the error committed by the ensemble which this time is calculated as the average of the errors of the individual classifiers on the specific element x.
So in this case I do point by point the average error of the various classifiers and then I do the average for all the points of the dataset, while for the previous case I did for all the points the average error of the single classifier and then I did the average for all the classifiers.
The average error for a point of all my classifiers is always less than or equal to the error of a classifier on all points because we made the assumption that the errors are uncorrelated so if a classifier makes a mistake on a point and has a high error the others will do it well (being the classifiers uncorrelated so they make mistakes on different points).
Let's compare the two cases better.
For $\mathscr{E}_{AV}$ I'm averaging the accuracies of the single classifier while for $\mathscr{E}$ I'm averaging the average of the accuracies point by point.
So in bagging if a classifier gets a point wrong for me but the others do it right (uncorrelated classifiers) that error weighs $1/M$ and if M is large it weighs little.
If instead I don't do bagging and the single classifiers get 65% of the points wrong for me, I don't care that I get different points wrong because I calculated the expectation over the whole dataset and the average of 1000 classifiers with 65% still makes 65%.
For $\mathscr{E}$ I focus on the single element x and if one classifier gets it wrong but all the other 9 get it right, it will have a weight of 1/10.
For $\mathscr{E}_{AV}$ the error e_i always weighs 1 because I don't take into account what the other classifiers in the ensemble said for the specific example.
In general, therefore, it always holds that

$$\mathscr{E} \leq \mathscr{E}_{AV} \tag{8.9}$$

If and only if all classifiers make mistakes in the same way on the same example (i.e., if the classifiers are not independent) then $\mathscr{E}_{AV} = \mathscr{E}$

8.2 Boosting

Another common technique is **Boosting**, considered to be one of the most significant developments in machine learning. This technique consists of having the different classifiers in series rather than in parallel as seen so far. So each classifier has visibility only of what happened to the previous classifier.

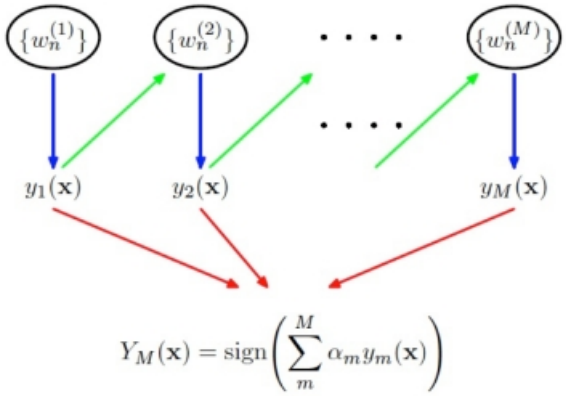
More in detail, starting from a dataset D, I sequentially train each classifier focusing on the errors made by the previous one, that is, giving them more importance (weight).
Initially, each element x_k of the dataset is assigned equal weight $w_k = \frac{1}{N}$, then for each classifier $i = 1, \ldots, M$:

- Sample the elements of a new dataset D^i from D using the weights $\{w_k^i\}_{k=1}^N$ as the probability of sampling the element x_k.
- Train each classifier f_i on D^i, measure and save its accuracy α_i (also called reputation or confidence)
- If x_k has not been correctly classified, increase its weight and renormalize all weights. In this way, in the next phase, i.e., for the next classifier, when sampling elements from D, the probability of picking elements that I got wrong in the previous phase is higher, and therefore the classifier will focus on the elements that were wrong by the classifiers of the previous phases.

The decision rule will therefore be

$$y(\mathbf{x}_{new}) = sign(\sum_{i=0}^{M} \alpha_i f_i(\mathbf{x}_{new})) \tag{8.10}$$

If the result of the sum is positive, it means that I had a majority of $+1$, otherwise of -1, so the sign gives me the correct class.



For boosting, therefore, we will have as hyperparameters:

- M, number of classifiers
- Types of classifier

However, a chain that is too long can overfit because in the long run I will only give importance to errors and therefore the final classifier will be too specialized on my specific dataset.

How do you implement weighted sampling of elements from the dataset in reality?

A first solution that might come to mind is to transform the probability of the element into an integer and repeat the element as many times as that integer in the dataset. For example, if an element has a probability of 0.3, I multiply that probability by 10 and get 3, so I will have to repeat it 3 times in the dataset. However, this method presents various problems, for example, if the elements have very similar probabilities between them and therefore many digits after the decimal point will also be significant, at that point it will be difficult to represent my dataset. For example, if I have an element with probability 0.12543, making it an integer would mean multiplying it by 10000 and I should repeat it in the dataset 12543 times, which is definitely too many.

An excellent alternative is the **cumulative density function (cdf) algorithm**:

Algorithm 2 CDF

Require: discrete distribution $H(x)$ with $x = \{1,\dots,N\}$ where $H(0) = 0$, N is the number of unique elements (not repeated) and K is the total number of elements I want to sample (repetition is allowed)

Ensure: K indices

Step 1: construct the cdf table T

$$T_x = \Big[\sum_{i=0}^{x-1} H(i), \sum_{i=0}^{x} H(i)\Big] \forall x$$

Step 2: find the indices

while number of iterations = K **do**

Choose a random number $n \in [0,1]$

Find $k \mid T_k[0] \leq n \leq T_k[1]$

Output k

end while

Let's see a practical example:

$$x = \{1,\dots,5\},\ D = (x_1,\dots,x_5),\ H(x) = \Big\{\frac{1}{5},\frac{1}{5},\frac{1}{5},\frac{1}{5},\frac{1}{5}\Big\} \tag{8.11}$$

Step 1: construct the table

	$T_x[0]$	$T_x[1]$
T_1	$H(0)$	$H(0)+H(1)$
T_2	$H(0)+H(1)$	$H(0)+H(1)+H(2)$
T_3	$H(0)+H(1)+H(2)$	$H(0)+H(1)+H(2)+H(3)$
T_4	$H(0)+H(1)+H(2)+H(3)$	$H(0)+H(1)+H(2)+H(3)+H(4)$
T_5	$H(0)+H(1)+H(2)+H(3)+H(4)$	$H(0)+H(1)+H(2)+H(3)+H(4)+H(5)$

	$T_x[0]$	$T_x[1]$
T_1	0	1/5
T_2	1/5	2/5
T_3	2/5	3/5
T_4	3/5	4/5
T_5	4/5	5/5

Step 2: find K indices that allow me to extract K samples.
So I draw K random numbers $n \in [0,1]$ and I have to find the row k such that my number n is between the value of the first column and the value of the second column relative to that row.

- For example, if I get $n = 0.39$, since $1/5 < 0.39 < 2/5$, I find $k = 2$.
- For example, if I get $n = 0.42$, since $2/5 < 0.42 < 3/5$, I find $k = 3$.

I sampled element 2 (x_2) and element 3 (x_3) from my dataset.
Suppose then that my classifier makes a mistake on x_1, then its weight must be increased as well as its probability of being extracted in subsequent datasets. So for example its probability could become $1/5+1 = 6/5$. Then normalize the probabilities because they must always sum to 1: so I calculate the current total probability which will be $6/5+1/5+1/5+1/5+1/5 = 2$ and divide all the probabilities by it obtaining:

$$H(x) = \Big\{\frac{6}{10},\frac{1}{10},\frac{1}{10},\frac{1}{10},\frac{1}{10}\Big\} \tag{8.12}$$

At this point I construct the new cdf:

	$T_x[0]$	$T_x[1]$
T_1	0	6/10
T_2	6/10	7/10
T_3	7/10	8/10
T_4	8/10	9/10
T_5	9/10	1

As can be easily understood, now it is much more likely that the extracted number will end up in the first row because I have a much larger interval than that of the others and in my dataset x_1 will appear a number of times proportional to this probability. In this way next classifiers can focus on mistakes made by the previous one.
The main roblem with this method is unfortunately the slowness of sampling because I have to remake all the tables each time.

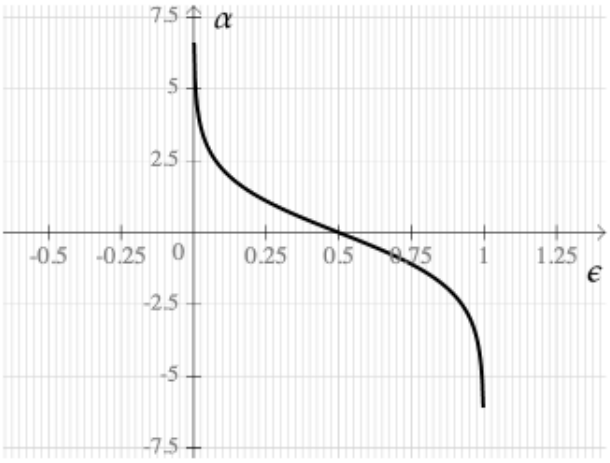
8.2.1 Adaboost

To solve the problem described above another algorithm, called **Adaboost**, was introduced.
Adaboost tries to circumvent the sampling problem, in particular instead of creating new datasets with a number of samples proportional to their weight, it interprets the latter as the importance I give to the example in calculating the error function of the single classifier. While before the weight was not used for any calculation except as a probability of sampling my data, now it is used directly in the error formula of the k-th classifier:

$$\varepsilon_k = \sum_{i=1}^{N} w_k^i 1_{(} f(x^i) \neq y^i) \tag{8.13}$$

This will then be used to calculate the accuracy (or confidence, trust) of the classifier and to do this a sigmoid function is used:

$$\alpha_k = log\Big(\frac{1-\varepsilon_k}{\varepsilon_k}\Big) \tag{8.14}$$



As can be seen from the graph, each classifier at each stage must have $1-\varepsilon_k > 0.5$ otherwise it would be like guessing randomly and in fact from the graph we can see how in this case we would have negative accuracy. If this were not the case, the chain would be restarted from the beginning.

In generic boosting, accuracy was used in the classification function but it was calculated simply on the error without using the weights which as mentioned were used only to sample the various elements with more or less probability. In Adaboost, accuracy is instead calculated taking into account also the weights of the individual elements.
In turn, the accuracy will be used to calculate the weight of the various elements in the next stage:

$$w_{k+1} = \begin{cases} w_k e^{-\alpha_k} & \text{if } f_k(x^i) = y^i \\ w_k e^{\alpha_k} & \text{if } f_k(x^i) \neq y^i \end{cases} \tag{8.15}$$

That is, after calculating the accuracy of the classifier, if for example this were high and I classified the example well, then its weight in the next stage is divided by e^{α_k} so it will have less weight. If instead I didn't get it right, I multiply by e^{α_k} so it will have more weight.
The final decision rule is as follows:

$$y^i = sign(\sum_{k=0}^{M} \alpha_k f_k(x^i)) \tag{8.16}$$

It's the same as before but as mentioned several times, accuracy is calculated taking into account also the weights of the individual elements. The real advantages of boosting is that we can achieve really good performance using classifiers and models that are very often **decision stumps**, so models very simple and weak that:

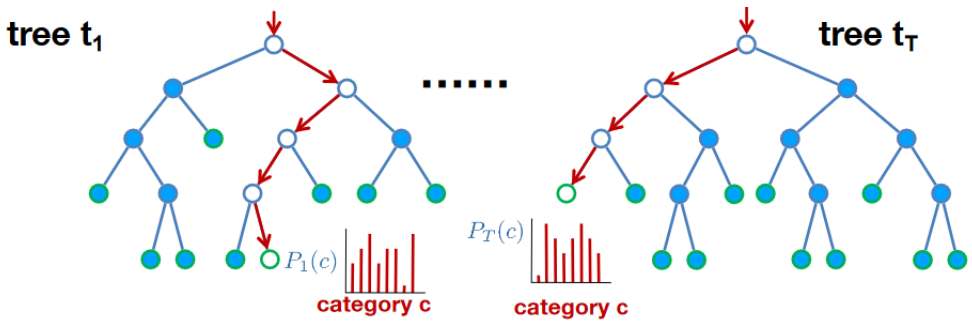
- Randomly choose a feature (which would be a dimension, therefore an axis)
- Randomly choose a threshold
- Classify all points beyond the threshold (along that particular axis) as +1 and the others as -1 (or vice versa)
- Calculate ε_j on D_j
- Repeat all this until $\varepsilon_j < 0.5$

8.3 Random forest

Random forest is an ensemble method based on bagging, founded on the concept of binary decision trees. By putting together multiple decision trees, we create a forest of trees (random forest) which is nothing more than the bagging technique applied to decision trees. We therefore divide the training set into T subsets as independent as possible and train each tree on a subset (in this way we have different shapes and so less possible errors).
The classification function will be:

$$P(c|\mathbf{v}) = \frac{1}{T} \sum_{t=1}^{T} P_t(c|\mathbf{v}) \tag{8.17}$$

So each element passes through all the trees of the forest and each leaf will give me a histogram.



At this point, I sum the histograms and renormalize obtaining the distribution of the forest. I choose randomly the elements that end up in each bag, as they are sampled uniformly, and I choose features and thresholds randomly. Proceeding in this way, the trees of the forest should not be equal. Keeping the trees as different as possible allows to decorrelate the errors, and to have all the benefits of bagging. The trees could also be trained for other tasks such as regression (regression trees) and clustering (clustering trees), what changes is the objective function (for more details see the slides and their references).

9. Unsupervised Learning

We talk about **unsupervised learning** when we have only the examples without associated labels. Unlike supervised learning, unsupervised machine learning models are given unlabeled data and allowed to discover patterns and insights without any explicit guidance or instruction. The two main techniques of unsupervised learning are the following ones:

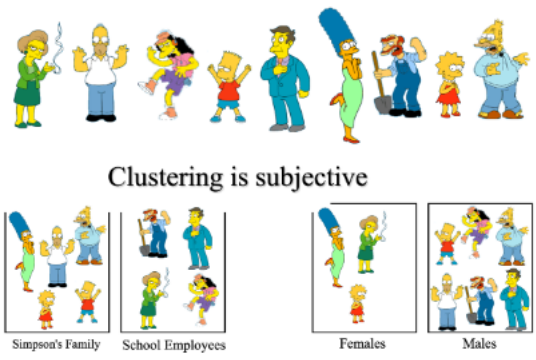
- **Novelty detection**: find a model to determine if a new example is different from the examples of the training set (problem of fake banknotes)
- **Clustering**: find the classes in which the examples of the training set can be grouped

9.1 Clustering

It involves grouping or creating groups (or classes) of data that are similar and in particular must have two characteristics:

- **High intra-class similarity:** In the single group, the elements must be very similar
- **Low inter-class similarity:** Between different groups, the elements must be very different.

The problem is that similarity is not a univocal concept. Essentially, given a set of elements, regardless of how you choose to describe them in terms of features, there is no univocal way to group them, so it is a subjective problem (depends on the feature of interest).



Similarity

Similarity is a complicated concept as it is closely related to the concept of semantics (i.e., meaning) and consequently is difficult to quantify.

Figure 9.1: Chromatic similarity but not semantic

One way to do this is to use a (mathematical) measure, particularly a distance.

Definition 9.1.1 — Measure. In mathematical analysis, a measure is a function that assigns a real number to a subset of a given set to quantify the notion of its extension. In particular, lengths are assigned to curve segments, areas to surfaces, volumes to three-dimensional figures, and probabilities to events[a].

[a] https://en.wikipedia.org/wiki/Measure_(mathematics)

Corollary 9.1.1 — Distance. A distance[a] d on a set X is a measure that satisfies the following properties for every choice of $a, b, c \in X$:

$$d : X \times X \to \mathbb{R} \tag{9.1}$$

- $d(a,b) \geq 0$ (Non-negativity)
- $d(a,b) = 0 \iff a = b$ (Identity of indiscernibles)
- $d(a,b) = d(b,a)$ (Symmetry)
- $d(a,b) \leq d(a,c) + d(c,b)$ (Triangle inequality)

[a] https://en.wikipedia.org/wiki/Metric_(mathematics)

Remember that similarity and distance are different measure (max similarity = min distance).

Types of clustering and properties

Clustering can be classified in two main categories:

- Hierarchical
- Partitional

Hierarchical clustering is an iterative type of clustering in which elements are merged or split hierarchically two by two (thus building a tree of merges or splits). Instead, partitional clustering divides the entire set of elements into groups (partitions).

Whatever the clustering mode, the following properties are required:

- Time and space scalability (it must be applicable to large amounts of data without becoming computationally expensive)
- Ability to handle heterogeneous data (described with feature vectors of integers, booleans, etc.)
- Have minimal knowledge about the domain
- Ability to handle noise and outliers

- Independence from the order in which the data is presented
- Incorporate user-defined constraints
- Interpretability and usability

9.1.1 Hierarchical clustering

Hierarchical clustering is represented by a tree representation called a **dendrogram** where the dots represent the elements and the lines represent the relationships between the elements. So two elements joined by a line belong to the same group. From the figure below we can notice that at different heights of dendogram we have different type of partitions.

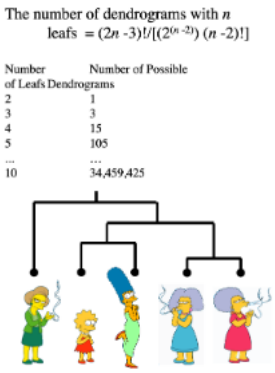


Figure 9.2: Dendrogram

Depending on the strategy chosen, clustering can be:

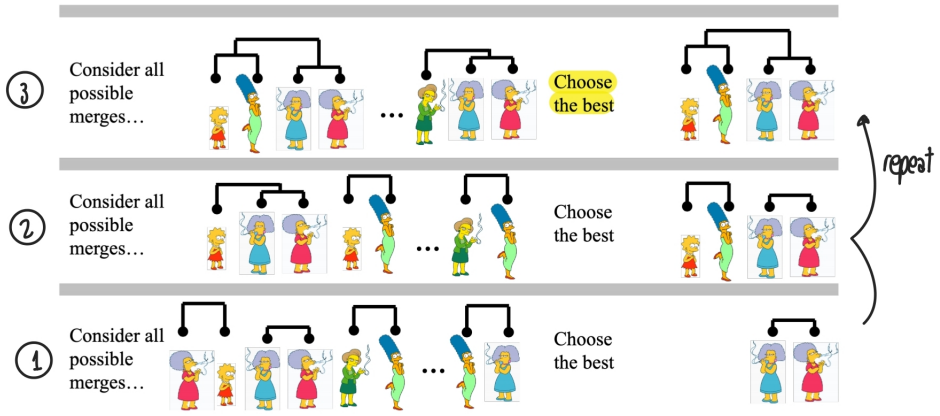
- Bottom-up (agglomerative)
- Top-down (divisive)

In the case of agglomerative clustering, initially all elements are considered disjoint, i.e., all belonging to different clusters (each item in its own cluster), and merge operations are performed, that is, union of two elements to create groups, and possibly merge groups with each other.
Instead the divisive one doesn't start from the leaves of the dendrogram, but from the root where all elements are together, and from that group all ways of dividing the cluster into two are considered and the best is taken and then recursively repeated for both subclusters just obtained.
To choose which elements to take or remove, a distance matrix is constructed which will be a symmetrical triangular matrix of the type:

$$\begin{array}{cc} elem & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & d(1,2) & d(1,3) \\ d(2,1) & 0 & d(2,3) \\ d(3,1) & d(3,2) & 0 \end{pmatrix} \end{array} \tag{9.2}$$

In the bottom-up case, among all pairs, I always take the one with minimum distance until all clusters are merged together. The problem remains of calculating the element-group and group-group distance. There are different strategies to do this:

- **Single linkage (nearest neighbor):** The group-group or element-group distance is that between the two closest elements. So it is the minimum of the distance between all elements belonging to group A against all elements belonging to group B. This strategy maximizes the agglomerativeness of clusters but also the variance. I will therefore have more elongated clusters.



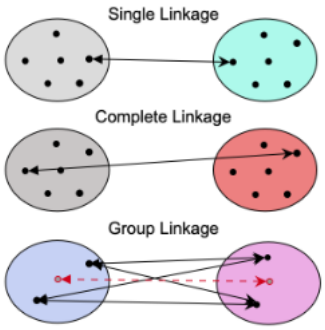
- **Complete linkage (furthest neighbor):** The distance is that between the farthest elements, consequently I will have more compact clusters and in fact it minimizes the variance.
- **Group average linkage:** The distance is the average of the distances between the elements belonging to A, against the elements belonging to B.
- **Ward's Linkage:** we want to minimize the variance indeed by minimizing it we maximize the similarity

$$\frac{mean}{variance} \tag{9.3}$$

and minimize the distance

$$mean \cdot variance \tag{9.4}$$

It takes into account how much the distances within the group have varied, this is because you could have some very close elements that somehow shift the average towards a low value, while other elements are also very far away.



At the end, we stop aggregating when the group-group or element-group distance is above a certain threshold.
So essentially, hierarchical clustering is accompanied by two hyperparameters:

- Aggregation strategy
- Threshold beyond which to stop aggregating

Let's summarize some characteristics of hierarchical clustering:

- It is not necessary to establish at the beginning how many clusters to make but a threshold must be defined

- For some domains, the hierarchical structure makes it understandable to humans, cause of the dendogram (unlike partitional), but in general the interpretation of results is very subjective
- It doesn't scale well because each time the distances of all with all must be calculated: minimum time complexity $O(n^2)$ where n is the number of elements.
- It is not an optimal algorithm, a heuristic is used, and it is not said that it is the globally best partitioning but only locally.
- Deal really well with outliers

9.1.2 K-means (partitional clustering)

It partitions the entire set into a desired number K of non-overlapping groups (each element is in only one group). It is therefore necessary to know how many groups to make. The most famous partitional clustering algorithm is **K-means**.

Corollary 9.1.2 — K-means. Given a set of n points $(x_1,\dots,x_n)$ and a set of k clusters $S=(S_1,\dots,S_k)$ has as objective function:

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x\in S_i} \|x-\mu_i\|^2 \tag{9.5}$$

where $\mu_i = \frac{1}{|S_i|}\sum_{x\in S_i} x$ is the center of cluster S_i

So we want to minimize for all clusters the sum of the distances of the elements from their center, and we want to obtain the partitioning in which this quantity is minimum. In other words, I want to squeeze the cluster towards its center which is equivalent to minimizing the variance and thus increasing the intra-cluster similarity.
The objective function is indeed equivalent to the following

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} |S_i| Var S_i = \underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x,y\in S_i} \|x-y\|^2 \tag{9.6}$$

Exercise 9.1 — Proof. Proof of the variance decomposition formula, which is fundamental to understanding the k-means clustering algorithm. The formula states:

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x,y\in S_i} \|x-y\|^2 = \underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x\in S_i} \|x-\mu_i\|^2 \tag{9.7}$$

where $S=\{S_1,S_2,\dots,S_k\}$ is a partition of the data points into k clusters, and μ_i is the mean of cluster S_i.
Let's focus on the inner sum for a single cluster S_i:

$$\sum_{x,y\in S_i} \|x-y\|^2 \tag{9.8}$$

We can rewrite this as:

$$\sum_{x,y\in S_i} \|x-\mu_i+\mu_i-y\|^2 = \sum_{x,y\in S_i} (\|x-\mu_i\|^2 + \|\mu_i-y\|^2 + 2(x-\mu_i)\cdot(\mu_i-y)) \tag{9.9}$$

Now, let's consider the third term:

$$2 \sum_{x,y \in S_i} (x - \mu_i) \cdot (\mu_i - y) \tag{9.10}$$

1. First, we can rewrite this sum as:

$$2 \left[\sum_{x \in S_i} (x - \mu_i) \right] \cdot \left[\sum_{y \in S_i} (\mu_i - y) \right] \tag{9.11}$$

   This is possible because the x and y terms are independent of each other in the double sum.
2. Now, let's focus on the second part: $\sum_{y \in S_i} (\mu_i - y)$
   By definition, μ_i is the mean of cluster S_i. This means:

$$\mu_i = \frac{1}{|S_i|} \sum_{y \in S_i} y \tag{9.12}$$

3. Therefore:

$$\sum_{y \in S_i} (\mu_i - y) = \sum_{y \in S_i} \mu_i - \sum_{y \in S_i} y = \cancel{|S_i|} * \frac{1}{\cancel{|S_i|}} \sum_{y \in S_i} y - \sum_{y \in S_i} y = 0 \tag{9.13}$$

4. Since one part of our product is zero, the entire term becomes zero:

$$2 \left[\sum_{x \in S_i} (x - \mu_i) \right] \cdot 0 = 0 \tag{9.14}$$

This property, where the sum of differences from the mean equals zero, is a fundamental property of the arithmetic mean and is crucial in many statistical derivations, including this one for the variance decomposition formula.
After that our expression simplifies to:

$$\sum_{x,y \in S_i} (\|x - \mu_i\|^2 + \|\mu_i - y\|^2) \tag{9.15}$$

Let's break this down further:

1. First, let's separate the double sum:

$$\sum_{x \in S_i} \sum_{y \in S_i} (\|x - \mu_i\|^2 + \|\mu_i - y\|^2) \tag{9.16}$$

2. Now, we can split this into two sums:

$$\sum_{x \in S_i} \sum_{y \in S_i} \|x - \mu_i\|^2 + \sum_{x \in S_i} \sum_{y \in S_i} \|\mu_i - y\|^2 \tag{9.17}$$

3. Notice that in the first term, the inner sum $\sum_{y \in S_i}$ is just counting the number of points in S_i, which we call $|S_i|$:

$$|S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 + \sum_{x \in S_i} \sum_{y \in S_i} \|\mu_i - y\|^2 \tag{9.18}$$

4. Similarly, in the second term, the outer sum $\sum_{x \in S_i}$ is also just counting $|S_i|$:

$$|S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 + |S_i| \sum_{y \in S_i} \|\mu_i - y\|^2 \tag{9.19}$$

5. Now, notice that in the second term, we're summing over y, which is just a dummy variable. We can replace it with x:

$$|S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 + |S_i| \sum_{x \in S_i} \|\mu_i - y\|^2 \tag{9.20}$$

6. The expression $\|x - \mu_i\|^2$ is equal to $\|\mu_i - x\|^2$ (distance is symmetric), so these sums are identical:

$$|S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 + |S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{9.21}$$

7. Finally, we can factor out n_i and combine the sums (we arrive at the simplified form):

$$2|S_i| \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{9.22}$$

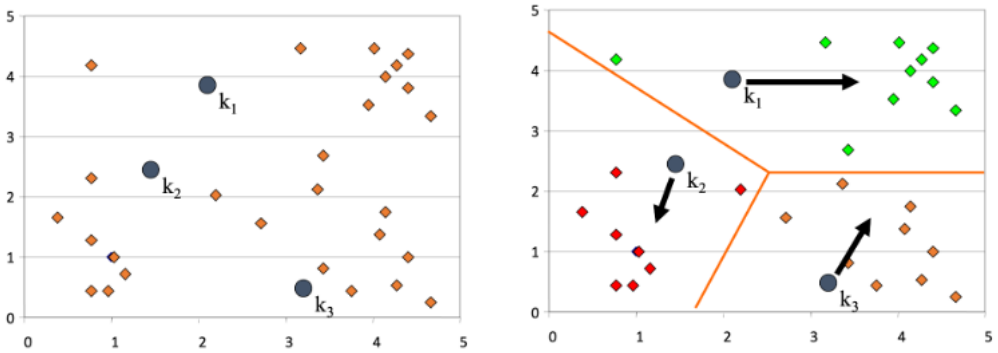
where $|S_i|$ is the number of points in cluster S_i.

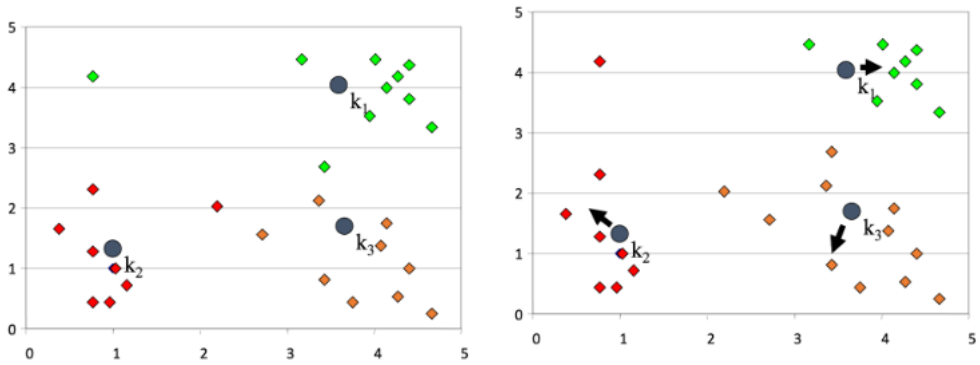
■

Here's the algorithm:

Algorithm 3 Naive k-means

```
Initialize k cluster centers randomly
while Centers not updated or maximum iteration reached do
    Assign each point to the nearest cluster                ▷ Assignment
    for each cluster do
        Calculate the mean of its points
        μ_{S_i}^{new} = Σ_{x_i ∈ S_i} x / |S_i|
    end for
    μ_{S_i} = μ_{S_i}^{new}                                 ▷ Update
end while
```





Let's summarize some characteristics of k-means:

- **Strenght:**
  - Relatively efficient: $O(tkn)$ where t is the number of iterations
  - Often ends up in a local optimum, but the global optimum can be found through other techniques such as simulated annealing (starting with different initializations and choosing the one that leads to the most voted solution) or genetic algorithms which consist of algorithms that allow evaluating different starting solutions (as if they were different biological individuals) and that by recombining them (analogously to sexual biological reproduction) and introducing elements of disorder (analogously to random genetic mutations) produce new solutions (new individuals) which are evaluated by choosing the best ones (environmental selection) in an attempt to converge towards "optimal" solutions[1].
- **Weakness:**
  - It can only be applied when the mean is defined. If one of the features is a boolean, or takes discrete values, k-means cannot be applied as is (in male-female mean doesn't have a semantic meaning).
  - I must specify the number of clusters k before executing it
  - It is not able to handle noise and outliers
  - Not suitable for clustering non-convex shapes (no compact data)

To overcome the problem of the mean, a variant is introduced that uses the PAM (Partitioning Around Medoids) method.

K-medoids

In K-means, when calculating the mean, new points are created, so the centers are never points that are in the dataset. If instead of creating new points, those already present in the dataset were used, there would be certainty that the incriminated variables have a value consistent with their domain. Therefore, PAM and, in particular, the K-medoids algorithm consists of calculating the new center not as an average, but as a representative object called a **medoid** which is the point among those assigned to the cluster whose sum of distances from all other points is lowest.

In other words, all points assigned to a cluster are taken, one point at a time is locked and its sum of distances from all others is calculated. In the end, the one with the lowest sum of distances is taken and this will become the new center. The assignment is redone and the position of the center is recalculated until convergence.

Also in this case, the initial medoids are set randomly.

[1] `https://en.wikipedia.org/wiki/Genetic_algorithm`

9.2 Spectral clustering

If the separation surfaces are not convex, the clustering types seen so far don't work because it might not be possible to calculate the distance between two points as Euclidean distance.

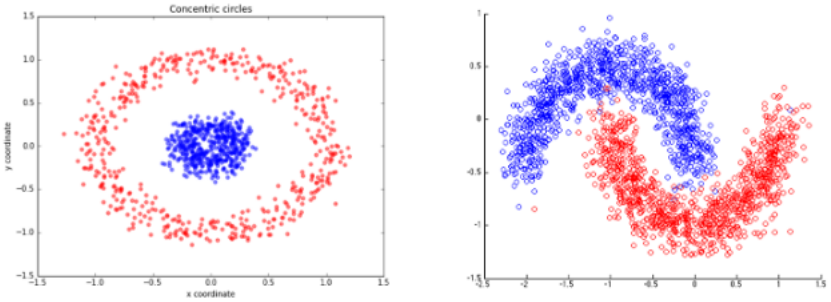


Figure 9.3: Non-convex surface

Therefore, we need to use another type of clustering, called **spectral clustering**, which assumes the points are distributed on a graph. The process of spectral clustering is composed by three main steps:

1. Construct a similarity graph (KNN) for all data points
2. Embed data points in a low-dimensional space, where the cluster are more obvious, with the use of eigenvectors of the graph Laplacian
3. A classical algorithm (k-means) is applied to partition the embedding

9.2.1 Graph Theory

To understand spectral clustering we have to do a short introduction to graph theory and understand how create the similarity graph. A graph is a pair of the type $G = (V, E)$ where:

- V is the set of Vertices
- E is the set of Edges

A graph can be categorize in different manner but the two most common are the followings:

- **Directed** or **Undirected**
- **Bipartite** (if there's the possibility to divide it into two disjoint groups of vertices without connections between them) or **Multipartite**

Both vertices and edges can be equipped with a scalar value called weight, indicated with $a_i > 0$ for vertices and with $w_{ij} > 0$ for directed edges where $i, j \in V$ and $(i, j) \in E$

Definition 9.2.1 — Vertex Field. We define the Vertex Field of a graph as an operator that goes from the set of vertices to real numbers:

$$f : V \to \mathbb{R} \tag{9.23}$$

Corollary 9.2.1 — Real Vertex function. The Vertex Field is also called Real Vertex function. Indeed, f is a function and in particular, it's a vector function (or vector of functions/coefficients):

$$f = (f_1, \cdots, f_n) \in \mathbb{R}^n \quad n = |V| \tag{9.24}$$

Corollary 9.2.2 — Hilbert space. If a scalar product is defined in a field, it is a Hilbert space.

The vertex field defines a Hilbert space as the scalar product between two fields is defined as

follows:

$$\langle f, g \rangle_{\mathscr{H}(V)} := \sum_{x_i \in V} f(x_i) g(x_i) \tag{9.25}$$

where $\mathscr{H}(V) := \{f : V \to \mathbb{R}\}$ is the Hilbert space defined on the vertex field.
Euclidean space is also a Hilbert space, but a Hilbert space is not necessarily Euclidean, because it may not be equipped with the concept of Euclidean distance (the triangle inequality might not hold).

Definition 9.2.2 — Edge Field. Similarly, we define the Edge Field of a graph (Edge Field/Real Edge function):

$$F : E \to \mathbb{R} \tag{9.26}$$
$$F = (F_1, \cdots, F_n) \in \mathscr{R}^n \quad n = |E| \tag{9.27}$$

$$\langle F, G \rangle_{\mathscr{H}(E)} := \sum_{(x_i, x_j) \in E} F(x_i, x_j) G(x_i, x_j) \tag{9.28}$$

where $\mathscr{H}(E) := \{F : E \to \mathbb{R}\}$ is the Hilbert space defined on the edge field.

Definition 9.2.3 — Weighted Difference. We define the **weighted difference** of f (or weighted derivative of a graph) along a directed edge $(x_i, x_j) \in E$ as follows:

$$\partial_{x_j} f(x_i) := \sqrt{w_{ij}} (f(x_j) - f(x_i)) \tag{9.29}$$

Definition 9.2.4 — Weighted Gradient. We define the **weighted gradient** operator of a graph as follows:

$$\nabla_w : \mathscr{H}(V) \to \mathscr{H}(E) \tag{9.30}$$
$$(\nabla_w f)(x_i, x_j) := \partial_{x_j} f(x_i) \tag{9.31}$$

Definition 9.2.5 — Adjoint operator. We define the **adjoint operator** to the weighted gradient as follows:

$$\nabla_w^* : \mathscr{H}(E) \to \mathscr{H}(V) \tag{9.32}$$
$$\langle \nabla_w f, G \rangle_{\mathscr{H}(E)} = \langle f, \nabla_w^* G \rangle_{\mathscr{H}(V)} \quad \forall f \in \mathscr{H}(V), G \in \mathscr{H}(E) \tag{9.33}$$

such that

$$(\nabla_w^* F)(x_i) := \frac{1}{2} \sum_{x_i \sim x_j} \sqrt{w_{ij}} (F(x_j, x_i) - F(x_i, x_j)) \tag{9.34}$$

Where $x_i \sim x_j$ indicates all vertices x_j connected to x_i.

Corollary 9.2.3 — Weighted Divergence. We define the **weighted divergence operator** as the negative adjoint operator of the gradient

$$div_w := -\nabla_w^* \tag{9.35}$$

The divergence at a vertex is essentially like measuring what's "flowing" in and out of that vertex through its edges, then finding the difference. For each edge connected to your vertex:

- If the edge is flowing OUT of the vertex, it contributes a positive value
- If the edge is flowing INTO the vertex, it contributes a negative value

The "net outflow" (divergence) is the sum of all these contributions

Definition 9.2.6 — Laplacian. The weighted Laplacian of a graph is defined as the divergence of the gradient. This operator maps functions from $\mathscr{H}(V)$ to $\mathscr{H}(V)$, where V is the vertex set:

$$\Delta_w : \mathscr{H}(V) \to \mathscr{H}(V) \qquad (\mathrm{div}_w(\nabla_w f))(x_i) = -(\nabla_w^*(\nabla_w f))(x_i) \tag{9.36}$$

where the negative sign appears from bringing the adjoint operator inside.
This expands to:

$$= -\frac{1}{2}\sum_{x_i \sim x_j} \sqrt{w_{ij}}((\nabla_w f)(x_j,x_i) - (\nabla_w f)(x_i,x_j)) \tag{9.37}$$

Here, the sum is over all adjacent vertices, and the factor of $\frac{1}{2}$ appears because each edge is counted twice in the summation (definition of adjoint operator). Then the terms are rearranged to:

$$= \frac{1}{2}\sum_{x_i \sim x_j} \sqrt{w_{ij}}((\nabla_w f)(x_i,x_j) - (\nabla_w f)(x_j,x_i)) \tag{9.38}$$

The gradient terms are then expanded:

$$= \frac{1}{2}\sum_{x_i \sim x_j} \sqrt{w_{ij}}(\sqrt{w_{ij}}(f(x_j) - f(x_i)) - (\sqrt{w_{ij}}(f(x_i) - f(x_j)))) \tag{9.39}$$

This shows how the gradient operates on the function values at vertices. Then the terms are simplified:

$$= \frac{1}{2}\sum_{x_i \sim x_j} w_{ij}(2f(x_i) - 2f(x_j)) \tag{9.40}$$

Note that the $\sqrt{w_{ij}}$ terms multiply together to give w_{ij}, and the factor of 2 comes from combining like terms. At the end, the final form of the weighted Laplacian is:

$$= \sum_{x_i \sim x_j} w_{ij}(f(x_i) - f(x_j)) =: (\Delta_w f)(x_i) \tag{9.41}$$

The Laplacian measures the difference between a function's value at a vertex and its weighted average at neighboring vertices. Instead, the negative sign from the adjoint operator determines the orientation of the differences. To summarize:

$$(\Delta_w f)(x_i) := \sum_{x_i \sim x_j} w_{ij}(f(x_j) - f(x_i)) \tag{9.42}$$

$$div(\nabla f) = \Delta f \tag{9.43}$$

$$\langle F, \nabla f\rangle_{\mathscr{H}(E)} = \langle \nabla^* F, f\rangle_{\mathscr{H}(V)} = \langle -divF, f\rangle_{\mathscr{H}(V)} \tag{9.44}$$

The Laplacian is like a **second-order derivative** because it tells me the smoothness of the graph, i.e., the absence of discontinuities between vertices.
The Laplacian can also be rewritten in matrix form where:

- $W : V \times V$ matrix of edge weights (Adjacency matrix)
- $D : V \times V$ (diagonal) degree matrix where $d_{i,i} = \sum_{\forall x_i \neq x_j} w_{ij}$

$$L = D - W \tag{9.45}$$

Which is equivalent to writing:

$$\Delta = D - W \tag{9.46}$$

Where:

$$W = \begin{pmatrix} 0 & w_{12} & \dots & w_{1n} \\ w_{21} & 0 & \dots & \vdots \\ \vdots & \dots & 0 & w_{n-1n} \\ w_{n1} & \dots & w_{nn-1} & 0 \end{pmatrix} \tag{9.47}$$

$$D = \begin{pmatrix} \sum_{j=1}^{n} w_{1j} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sum_{j=1}^{n} w_{nj} \end{pmatrix} \tag{9.48}$$

Observing how W and D are made, $L = D - W$ is symmetric and positive semidefinite. Consequently, it admits n real non-negative eigenvalues and eigenvectors that are orthonormal to each other (they may not be distinct).
From geometry, we recall that a certain ϕ is said to be an eigenvector of Δ if there exists an eigenvalue λ such that:

$$\Delta\phi = \lambda\phi \tag{9.49}$$

In general, having multiple eigenvectors and eigenvalues, we write:

$$\Delta\phi_k = \lambda_k\phi_k \quad k = 1,\dots,n \tag{9.50}$$

9.2.2 Partitioning using graphs

Let's now represent our data through a graph.
Given a dataset $D = \{x_i\}_{i=1}^{N}$ with $x_i \in \mathbb{R}^D$ we can construct a graph where

- Each vertex is a datapoint $V_i = x_i$
- The edge weights represent the similarity

$$w_{ij} = similarity(V_i, V_j) = similarity(x_i, x_j) \tag{9.51}$$

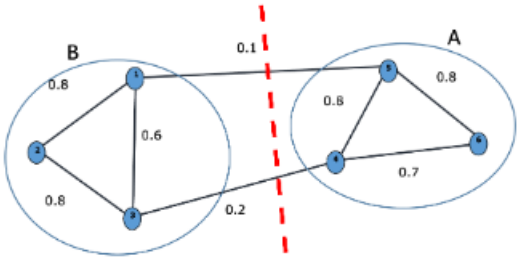
Once this is done, we can bipartition the graph by choosing two disjoint groups and cutting the edges that connect these. The value of the weights of these edges is called CUT:

$$CUT(A,B) = \sum_{i \in A, j \in B} w_{ij} \tag{9.52}$$

We thus fall into graph clustering which can have mainly two objective functions:

- **Minimum cut:** $mincut(A,B)$ set of edges to cut where the cost is minimum (low similarity). It doesn't work very well because at the end we have unbalanced cluster, especcialy if we have single nodes.
- **Normalized cut:** $minNcut(A,B) = \frac{cut(A,B)}{vol(A)} + \frac{cut(A,B)}{vol(B)}$ we want high volume (high similarity) and low cost cut at the same time.

Where the volume is the set of all edges entering a given set.



Dirichlet Energy and the mincut problem

Solving the mincut problem is equivalent to finding a discrete vector

$$p = \begin{cases} 1 & \text{if and only if} \quad V_i \in A \\ -1 & \text{if and only if} \quad V_i \in B \end{cases} \tag{9.53}$$

$$mincut(A,B) = \underset{p}{\operatorname{argmin}} \sum_{V} w_{ij}(p_i - p_j)^2 \tag{9.54}$$

Looking closer at p, we can notice that it is a field because it associates a real number to each vertex. However, this problem is NP-complete and can be simplified by relaxing p and using instead the field $f : V \to \mathbb{R}$.
The problem can thus be rewritten as follows:

$$mincut(A,B) = \underset{f}{\operatorname{argmin}} \sum_{V} w_{ij}(f_i - f_j)^2 = \underset{f}{\operatorname{argmin}} \sum_{V} \|\nabla_w f\|^2 = \underset{f}{\operatorname{argmin}} f^T L f \tag{9.55}$$

where $f^T L f$ is called **Dirichlet Energy** and indicates the smoothness of a function, i.e., the absence of discontinuities. Therefore, functions with high Dirichlet Energy are more jagged.

Exercise 9.2 Solving the mincut problem is equivalent to finding the continuous vertex field f that minimizes the Dirichlet Energy.

$$f'Lf = \sum_{V} w_{ij}(f_i - f_j)^2 \tag{9.56}$$

Given the definition of Laplacian, we can rewrite the first term in the following way:

$$f'Lf = f'Df - f'Wf \tag{9.57}$$

Then the matrix products are expanded:

- $f'Df$ becomes $\sum_{i=1}^{N} d_i f_i^2$ (diagonal matrix multiplication)
- $f'Wf$ becomes $\sum_{i,j=1}^{N} f_i f_j w_{ij}$ (weight matrix multiplication)

$$\sum_{i=1}^{N} d_i f_i^2 + \sum_{i,j=1}^{N} f_i f_j w_{ij} = \tag{9.58}$$

$$= \frac{1}{2}\sum_{i=1}^{N} d_i f_i^2 + \frac{1}{2}\sum_{i=1}^{N} d_i f_i^2 + \sum_{i,j=1}^{N} f_i f_j w_{ij} \tag{9.59}$$

Then I collect $\frac{1}{2}$:

$$\frac{1}{2}\left(\sum_{i=1}^{N} d_i f_i^2 - 2\sum_{i,j=1}^{N} f_i f_j w_{ij} + \sum_{j=1}^{N} d_j f_j^2\right) \tag{9.60}$$

The degree terms are expanded as sums of weights:

$$d_i f_i^2 = \sum_{j=1}^{N} w_{ij} f_i^2 \tag{9.61}$$

Replacing d we form the following quadratic expression:

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}(f_i^2 - 2f_i f_j + f_j^2) = \qquad \frac{1}{2}\sum_{i,j=1}^{N} w_{ij}(f_i - f_j)^2 \tag{9.62}$$

Minimizing this energy encourages similar values for strongly connected vertices. ■

All possible distinct solutions of the problem

$$\underset{f}{\operatorname{argmin}} \sum_{E} \|\nabla_w f\|^2 \tag{9.63}$$

are the eigenvectors of the Laplacian L ordered in ascending order with respect to their respective eigenvalues. So each eigenvector is associated with a Dirichlet energy which is its eigenvalue.
Before continuing, let's rewrite the problem in vector terms:

$$\min_{\phi_k} E_{Dir}(\phi_k) \text{ such that } \|\phi_k\| = 1, k = 1, \ldots, n \tag{9.64}$$

$$\phi_k \perp span\{\phi_1, \ldots, \phi_{k-1}\} \tag{9.65}$$

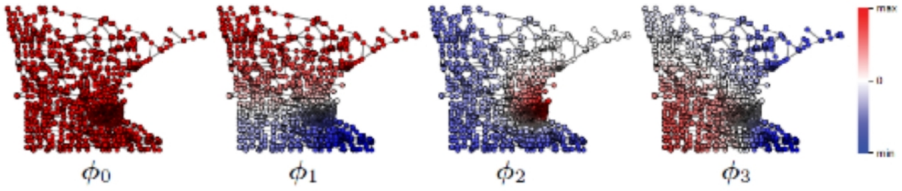
In other words, however I take an eigenvector, it must be orthogonal to the span generated by all the others, i.e., to the vector subspace generated by the others.

Theorem 9.2.4 — Rayleigh. The solution f^* of the mincut problem is the second smallest eigenvector of L and is called the Fiedler vector, while the value of the cut is the second smallest eigenvalue of L.

Why this? The set of eigenvalues always contains the null eigenvalues and the eigenvectors are sorted by their eigenvalues, so the fhe first eigenvalue of the Laplacian matrix is always $\lambda_1 = 0$ and the corresponding eigenvector is a constant vector (all entries are the same, typically all 1s). A constant vector can't help us partition the graph, because if all values are the same, we can't determine which vertices should be in different clusters (it provides no information about the graph's structure).
So the optimal solution is the smallest non-zero (feasible) solution that correspond to the second eigenvector of Laplacian matrix.

Explanation

Going back to the previous notation, one might wonder what f represents in practice? It's nothing than a vector that contains for each vertex a value that in some way allows me to evaluate some vertices as similar and others as different. In fact, the goal is to have partitions that are as homogeneous as possible from the point of view of field values (which represent the cluster to which a certain vertex belongs) and edge weights (which represent the similarity between the vertices that that edge connects). In other words, we are doing clustering, more in detail we are partitioning the graph in 2 clusters.



First Laplacian eigenvectors of a graph

How do we partition a graph into k clusters? There are mainly two approaches:

- **Recursive bi-partitioning (Hagen et al.,'91):** recursively applies the bipartition algorithm in a hierarchical and divisive manner. (inefficient and unstable)
- **Cluster multiple eigenvectors (Shi Malik,'00):** thanks to the eigenvectors, it constructs a new reduced space in which to do non-spectral clustering. (equivalent to doing PCA [see next section] and then k-means)

Clustering multiple eigenvectors

This second approach is divided into the following phases:

- Preprocessing dataset and calculation of the adjacency matrix W
- Calculation of the first k eigenvectors of L
- Construction of a reduced space by arranging the k eigenvectors by columns and thus obtaining an $n \times k$ matrix
- Application of k-means in this new reduced space of dimension $n \times k$ thus obtaining k clusters

The eigenvectors arranged by columns in this new matrix are the axes of a new reference system, therefore if I consider the rows, these represent the coordinates in this new space of a certain element of my dataset (features).
This is because the eigenvectors act as an ordered orthonormal basis for the new space, obtained by minimizing the Dirichlet Energy.

■ **Example 9.1** To give an example, image to have a dataset of 4 instances with a number n of features. We want to reduce the dimensions of these instances to 3 (it means k=3). We build the similarity graph and the matrices D and W; then we compute the Laplacian Matrix with the formula seen before $L = W - D$. We find the eigenvectors and eigenvalues of Laplacian and we take apart the first three. The new "dataset" will have as features (columns) the first three eigenvectors of Laplacian associated with the three smallest eigenvalues.
The goal of all this is to move from the initial space (Hilbert but not Euclidean) to a reduced but Euclidean one that allows me to do k-means as now the concept of Euclidean distance exists. Moreover, since this new space is constructed by minimizing the Dirichlet Energy, here I will have maximized the inter-class distance and minimized the intra-class one, partitioning the graph into subgroups as homogeneous as possible from the point of view of field values and weights (which as said represent the similarity between vertices). ■

Selecting the number of clusters

Unlike k-means where there is no rule for deciding k, in this case I can choose it using the **Eigengap Heuristic** which allows to find the k that makes the clustering as stable as possible. To find this value of k, just maximize the eigengap which is defined as follows:

$$\Delta_k = |\lambda_k - \lambda_{k-1}| \tag{9.66}$$

where λ_k indicates the k-th eigenvalue of the Laplacian.

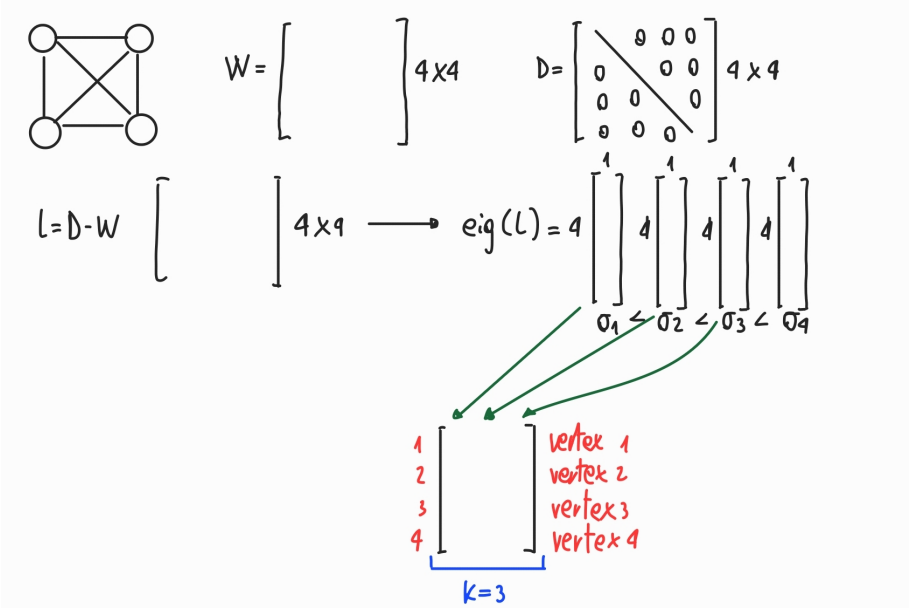


Figure 9.4: Example of clustering multiple eigenvectors

9.3 Dimensionality reduction

Any datapoint is represented by a certain number of features, and can therefore be described by its feature vector, let's say d-dimensional. If d is very high, looking for similarities becomes a problem known as the **"curse of dimensionality"** which refers to negative phenomena that arise when trying to analyze and organize data with many features (more features than datapoints, for example). Dimensionality reduction methods attempt to reduce the number of features used to represent an element, without losing descriptive capacity. In other words, we project the d-dimensional space into a new k-dimensional space such that:

- $k << d$
- Distances are preserved as much as possible

And then we solve the problem in this lower-dimensional space.

9.3.1 Multi-Dimensional Scaling

Multi-Dimensional Scaling (MDS) is a technique that consists of mapping elements in the reduced space by trying to minimize a quantity called stress:

$$stress = \sqrt{\frac{\sum_{i,j}(\hat{d}_{ij} - d_{ij})^2}{\sum_{i,j} d_{ij}^2}}, d_{ij} = |o_j - o_i| \text{ and } \hat{d}_{ij} = |\hat{o}_j - \hat{o}_i| \tag{9.67}$$

where d_{ij} is the distance in the original space and $\hat{d}_{ij} = |\hat{o}_j - \hat{o}_i|$ is the distance in the new space. So stress measures how much distances vary in the original space and in the transformed space. So the stress for each pair i and j within the dataset is the difference between the distance in the transformed space (indicated by variables with a hat) and in the original one, divided by a normalization constant.

This means that we want to have a new space where the distances between points are preserved. This is a problem that is not solved exactly, but is solved heuristically with the steepest descent algorithm, which initially assigns random coordinates to all points, evaluates the stress, takes a point randomly and moves it in the direction where stress decreases, and proceeds in the same way

for other points, without, however, having the guarantee that moving one point will not negatively affect all the others. This is an algorithm that continues as long as it can decrease stress by moving one point at a time. It's a poor algorithm given its iterative nature, and above all, there is no guarantee of convergence.
Furthermore, MDS does not work on features but only on distances between datapoints.

9.3.2 Embeddings

This technique consists of creating an embedding, that is, a new space embedded in the original one. More in detail, embedding is the procedure of incorporating one space within another, representing it with fewer features, in which we want some property to be preserved.

Definition 9.3.1 — Embedding. Given a distance matrix D, embed the objects in a k-dimensional vector space using a function F such that

$$D'(F(i),F(j)) \tag{9.68}$$

is as similar as possible to

$$D(i,j) \tag{9.69}$$

Actually exist two of mapping:

- **Isometric** if $D'(F(i),F(j)) = D(i,j)$ (distance still the same)
- **Contractive** if $D'(F(i),F(j)) \leq D(i,j)$

Depending on how we deform the space, there are two different types of embedding:

- **Linear**: datapoints projected by a linear transformation as in PCA
- **Non-linear**: datapoints projected non-linearly as in Local Linear Embedding, Laplacian eigenmap, Laplacian ISOMAP and t-SNE

Principal Component Analysis

Principal Component Analysis (PCA) is a linear transformation that project dataset in a new low-feature space. The algorithm is usually composed by these steps:

1. Create the dataset matrix X of dimension $N \times d$ where each row represents a data point.
2. Subtract the column mean from each row, feature by feature (it means center every points)
3. Calculate the covariance matrix Σ of X to understand the correlation between feature
   - On the diagonal is the variance, in the rest is the covariance
   - The variance is $(x-\mu)^2$ for 1D values while $(x-\mu)^T(x-\mu)$ for vectors. The covariance is $c_{ij} = \frac{1}{N}\sum_D (x_i-\mu_i)^T(x_j-\mu_j)$ That is, sum over the entire dataset D of i-th feature minus its mean multiplied by the j-th feature minus its mean. All divided by N. Having already subtracted the mean, I can calculate only $x^T x$ and $x_i^T x_j$
   - The resulting matrix will be $d \times d$, therefore square symmetric and with positive values along the diagonal; it will be also positive semi-definite, so it admits d real non-negative eigenvectors and eigenvalues.
4. Calculate eigenvectors and eigenvalues of Σ and order them in descending order with respect to the eigenvalues (connect with variance)
5. The principal components will be the M eigenvectors with the largest eigenvalues (the first M) where M is a hyperparameter that represents the dimension of the arrival space.

As said before, the covariance matrix is used to understand which features are correlated: the greater the covariance between two features, the more one varies as the other varies. The covariance is 0 if one feature never changes, in this case one of the two features is useless and should not be considered, because it is not correlated with the other.
The features that have greater covariance are those that interest us most because they spread out

more, so I can divide and classify them better. PCA therefore looks for the axes along which the features spread out the most (feature with high variance and low similarity), and these are the eigenvectors with the largest eigenvalues of the covariance matrix. I arrange these eigenvectors in columns as seen in spectral clustering, obtaining a $d \times M$ matrix. These eigenvectors will then be the set of axes of my new reduced space.
Since the operation I have done is linear (row by column product), this algorithm is limited to only rotating the original set of axes along the direction where I have the maximum possible variance for the first axis and minimum possible in the last axis (they are ordered in descending order).
The $d \times M$ matrix is also called the **projection matrix** P. In fact, if I take one of my data points in the original feature space, which will have dimensions $d \times 1$, and calculate $P^T x$, I get a vector $M \times 1$ which is the new representation of my point in the transformed space with M features instead of d. By performing this procedure for all points in the dataset, we obtain a new dataset that has N rows and M columns.

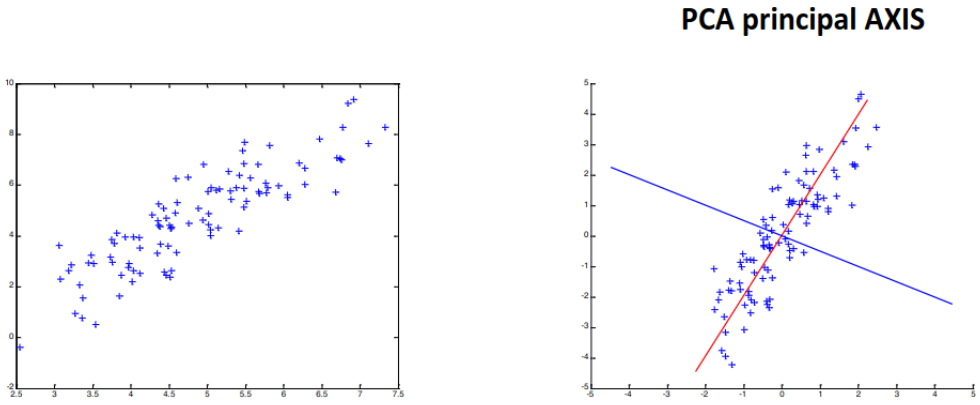


Figure 9.5: Geometrically, PCA rotates axis in a new position

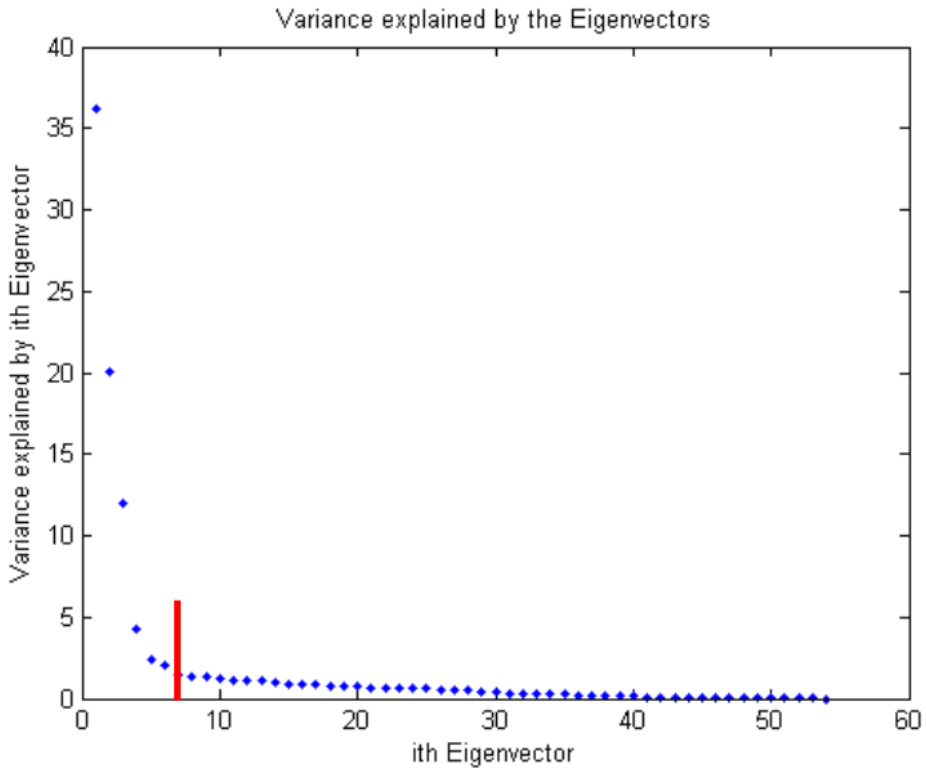
To choose the number of features to keep, I can normalize the eigenvalues (which correspond to the variance), i.e., I take each single eigenvalue and divide it by the sum of ALL the eigenvalues. Then, as I add eigenvectors to keep, these add up to the previous ones, covering more variance.
The goal is to take enough eigenvectors to cover 80-90% of the variance.

■ **Example 9.2** Given the following eigenvalues:

- Eigenvalue 1 = 36
- Eigenvalue 2 = 20
- Eigenvalue 3 = 13
- Eigenvalue 4 = 5
- Eigenvalue 5 = 3
- Eigenvalue 6 = 2
- Eigenvalue 7 = 1
- All the remaining small eigenvalues sum to 20 (their number doesn't matter to me)

The total sum of eigenvalues is equal to 100. Normalizing means, for example, dividing the first eigenvalue by the sum of the eigenvalues, thus obtaining 0.36, which is the percentage of variance I would cover if I decided to have only 1 dimension.
In this example, to cover 80% of the variance, it's enough to keep the first 7 eigenvectors. In fact, we get

$$\frac{36+20+13+5+3+2+1}{36+20+13+5+3+2+1+20} = 0.8 = 80\% \tag{9.70}$$



■

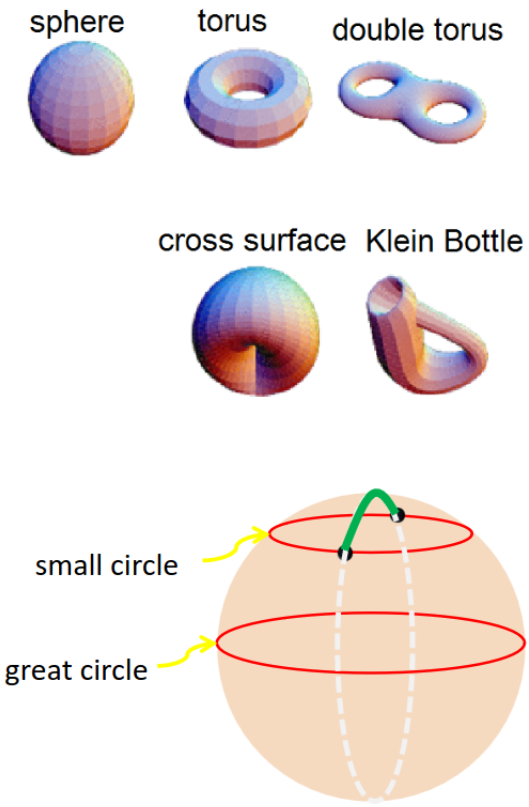
The problem with PCA is that it scales very poorly due to the covariance matrix, which is d^2 dimensional, and for example, if my data were images with a high number of pixels, PCA would become unsustainable. Essentially, it's very inefficient to apply PCA in spaces where the number of features is greater than the number of elements. One solution is to use **Singular Value Decomposition (SVD)**
Another limitation is that we are simply looking at our points from another perspective and we are not really changing space, so if in the original one I have overlapping points, I will also have them in the arrival one. To overcome this problem, non-linear transformations must be used.

Laplacian eigenmap

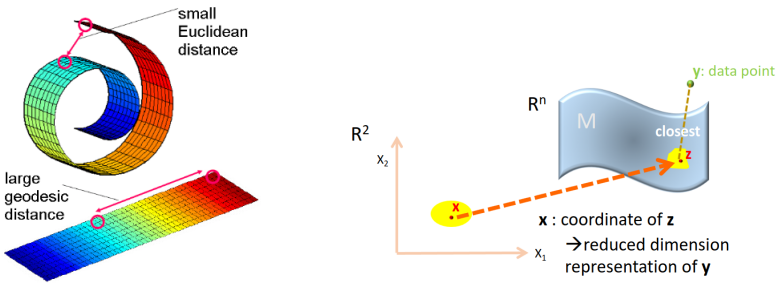
Non-linear embeddings construct a non-linear transformation between the starting space and the arrival space in order to preserve distances. In some way, it is an isometric transformation (not at the level of the overall space, but at the local level) and tries to obtain a new space that is locally Euclidean, in which distances are preserved. We have already seen a non-linear transformation in a locally Euclidean space where distances are preserved: spectral clustering.

Definition 9.3.2 — Manifold. A manifold is a locally Euclidean topological space (given a point, I can approximate the surface of its neighborhood as a Euclidean plane). In general, an object that is flat on a small scale is a manifold.

When you have many features, you can approximate their feature space (whose geometry is unknown) with a manifold that in general could be a subspace of $\mathbb{R}^n$.
The problem is that on the manifold, distances are not Euclidean. For example, if I wanted to calculate the distance between two opposite points on a sphere, I can't draw a straight line but must follow the curvature of the sphere. This distance is called the **geodesic distance**, the shortest curve connecting two points of a manifold.
If I take a point of this manifold, however, I can consider its neighborhood as a Euclidean plane.



By doing so, I can tessellate the space into many small Euclidean planes, and through embedding, I therefore want to project my points and their neighborhood (tile) into a reduced subspace $\mathbb{R}^k$, $k << n$ preserving the Euclidean distances between them and their neighbors.



The two main algorithms that use this approach are **Locally Linear Embedding (LLE)** and **Laplacian Eigenmaps**. They are both based on graphs and consist of 3 steps:

1. Find the k nearest neighbors.
2. Estimate the local properties of the manifold by observing the neighbors (for example, by constructing an adjacency matrix)
3. Find a global embedding that preserves these properties

To find this embedding, we need to calculate the unnormalized Laplacian of the graph $L = D - A$ where A is the adjacency matrix (which in the section on spectral clustering was indicated as W). The adjacency or weight matrix is a matrix that has all 0s on the diagonal and in the rest contains 1 if two points are close, 0 otherwise. Instead, the degree matrix D, on the other hand, is symmetric

as it contains all zeros except on the diagonal where the sum of the values of the adjacency matrix on the respective row is found.
The resulting Laplacian is therefore correct if doing the sums by row always gives zero.

So given the points $x_i \in M$ where M is the manifold found in $\mathbb{R}^l$, the goal is to find the points $y_i \in \mathbb{R}^m$ ($m << l$) that represent the original ones in the arrival space.
Laplacian eigenmap thus approximates the manifold with an adjacency graph that is based on an adjacency matrix that can be:

- **simple minded**, 1 if adjacent (distance less than a certain threshold) 0 otherwise (as seen so far)
- **heat kernel**

$$A_{ij} = e^{-\frac{\|x_i - x_j\|^2}{t}} \tag{9.71}$$

  where t is a real number (hyperparameter) called temperature that regulates the sensitivity of the distance. In particular, it amplify differences if it is less than 1 while make all distance equals if it is greater than 1.

If I use the heat kernel to understand which points to consider, I set a threshold that is related to the distance between the points. Being a negative exponential, I will have to take the points with values above my threshold. Alternatively, the number of neighbors is decided and for each point that number of neighbors is taken in ascending order of distance.
In this way, I make the structure of a Euclidean tile emerge.

If we call m the dimension of the reduced space, just as seen in spectral clustering, the goal is to find the first m eigenvectors that minimize the Dirichlet energy (represented by the respective eigenvalues of the Laplacian) and consider them as axes of my new reduced space.

Corollary 9.3.1 — Trace. The trace of a matrix is the sum of its eigenvalues (counted with multiplicities)

If I arrange the eigenvectors by columns, I get a matrix Y of dimension $n \times m$ and the problem can be written as seen for spectral clustering (with Y instead of Φ):

$$\underset{Y \in \mathbb{R}^{n \times m}}{argmin}\, tr(Y^T L Y) \text{ such that } Y^T Y = I \tag{9.72}$$

To sum up:

- It is a computationally efficient non-linear approach to dimensionality reduction
- It preserves only the local properties of my neighbors
- If you need global consistency look at **ISOMAP** methos that use geodesic distances

In conclusion, when I have a high number of features, it is better to apply a non-linear dimensionality reduction because with a high number it is easy for the space not to be Euclidean, while if I have a low number, the linear one might be fine. As for spectral clustering, the variables that I throw away by reducing dimensions are the noisiest ones.

T-SNE (asked only qualitatively)

The **t-distributed Stochastic Neighbor Embedding** (t-SNE) method allows creating and visualizing 2D (or 3D) embeddings. It is typically used to visualize (and not to classify) what is close and what is not in the feature space. An example is the "proximity" between Netflix movies.
The idea consists of finding a low-dimensional space, where the probability of finding point j in the neighborhood of point i considering k neighbors must be preserved. In other words, I want to preserve the density distribution of my points:

$$p_{j|i} = \frac{exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k \neq i} exp(-\|x_i - x_k\|^2/2\sigma_i^2)} \tag{9.73}$$

Where σ_i is a hyperparameter and thanks to the denominator I normalize in a neighborhood of k neighbors.
In other words, each point j is attributed the probability of being in the neighborhood of i proportionally to its distance from the point that is in the center of the considered area (the denominator does a sort of average that is equivalent to the center of the neighborhood of the k neighbors I'm considering).
In the destination space, where points are represented with other coordinates that we call y, the same quantity is constructed for the pair (j,i).

$$q_{j|i} = \frac{exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} exp(-\|y_i - y_k\|^2)} \tag{9.74}$$

The two probabilities must coincide, and this does not mean that point j in the destination space must be in the same position as in the original space, but it means that the distance is preserved even if it can be found in any position within point i. To do this, a measure between probability densities to be minimized is constructed.
To measure the distance between two probability distributions, we use the KL divergence. The objective is to make the two distributions be as close as possible, so we want to minimize:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} log \frac{p_{j|i}}{q_{j|i}} \tag{9.75}$$

Corollary 9.3.2 — KL divergence. In mathematical statistics, the Kullback–Leibler (KL) divergence is a type of statistical distance: a measure of how one reference probability distribution P is different from a second probability distribution Q.

To find the minimum of this quantity, we need use the gradient descent method and so compute the gradient of C, which is proven to be:

$$\frac{\partial C}{\partial y_i} = \sum_{j \neq i} \left(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j} \right) \tag{9.76}$$

At this point, the algorithm consists of:

1. Position $y_i, i = 1, \ldots, n$ randomly
2. Move these points in the direction of the gradient
3. In particular, for all points, perform this calculation:

$$Y^{(T)} = Y^{(T-1)} + \eta \frac{\partial C}{\partial y_i} + \beta(t)(Y^{(T-1)} - Y^{(T-2)}) \tag{9.77}$$

The term after $\beta(t)$ is called **momentum** and specifies that the difference between iteration (T-2) and (T-1) should be as low as possible (the points should not be moved too much) We proceed until the points are no longer moved (gradient zero) point i can no longer be moved, or c no longer decreases by moving y.
An interesting fact about this problem turns is that is similar to the N-body problem, i.e., the problem of estimating the attraction that celestial bodies have on another.

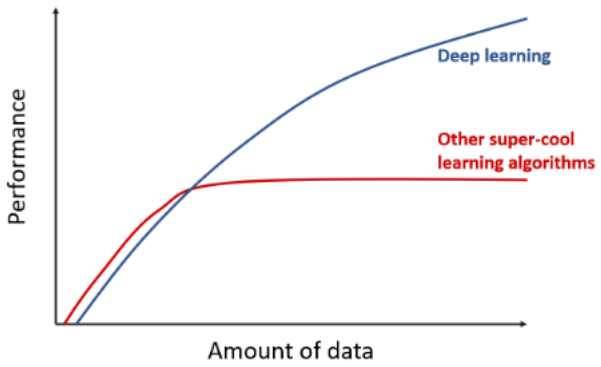
III Deep Learning

10. Deep Neural Networks

Deep learning is a subset of machine learning, primarily based on neural networks. These types of techniques are not new (their first theories date back to the 1940s) but, recently, went because its performances become very good, and the reasons are the followings

- Large amounts of labeled (and especially unlabeled) data
- Adequate computational power to train NN
- Dedicated hardware (GPUs, possibly distributed)

Why are neural networks so cool? In traditional machine learning, after a certain point, performance does not improve with increasing training data. This because the initial phase of feature extraction is not connected to the optimization of the ML model parameters but is done beforehand. The features are fed into the model, which must do the best it can with these features (feature fixed). Instead, in deep learning the feature extraction and parameter learning parts are not decoupled. By coupling the two phases, as training data is provided to the system, the classifier doesn't change, but the features do. The deep learning model extracts increasingly better features. This is what allows a model of this type to continuously improve.



Generally speaking, most ML methods depend on features called handcrafted (manually extracted).

Such features have the limitation of making sense to those who extracted them, but they may not be the best features for solving the specific problem. Deep learning allows extracting the best representation directly from the data (a representation that is cascaded with the learning process of the classifier parameters, so in some way the two parts can help each other, while in traditional ML they are in series).

Let's imagine we have a training dataset D of N images and we want to classify them among K distinct classes. The training set will therefore be formed by pairs (x_i, y_i) where:

- $x_i \in \mathbb{R}^D$, $i = 1, \ldots, N$
- $y_i \in \{1, \ldots, K\}$

Our goal is to find the function $f : \mathbb{R}^D \to \mathbb{R}^K$ that maps the images to scores for the different classes.

■ **Example 10.1** To give some real numbers, let's take as an example the CIFAR-10 dataset which has $N = 10000$ images formed by 32 x 32 RGB pixels each and each belonging to one of K = 10 classes. Consequently:

- $x_i \in \mathbb{R}^{3072}$ where 3072 = 32 x 32 x 3
- $f(x_i, W, b) = Wx_i + b$ where $W \in \mathbb{R}^{10 \times 3072}$ is the weight matrix and $b \in \mathbb{R}^{10}$ is the bias vector

■

The goal is to find the parameters that, given a new input image, I obtain an output score of the correct class higher than that of the other classes. The problem is that I have scores as output, so real numbers $\in (-\infty, +\infty)$ which are difficult to compare. So, as what happen in logistic regression, have to switch to probabilities.
To do this, imagining a two-class problem, we can add the sigmoid (which is a non-linearity) to our function obtaining a logistic regression classifier:

$$f(x_i, W, b) = \sigma(Wx_i + b) \tag{10.1}$$

where

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{10.2}$$

As we know, this can be trained by optimizing the binary cross entropy loss:

$$L(W, b) = -\frac{1}{m} \sum_{i=1}^{m} (y_i log(f_{x_i}) + (1 - y_i) log(1 - f_{x_i})) \tag{10.3}$$

Where m is the number of examples in the training set and f_{x_i} is an abuse of notation denoting $f(x_i, W, b)$.
However, we normally have more than two classes, so an alternative called **Softmax** is introduced.

Corollary 10.0.1 — Softmax. The Softmax is an operator that generalizes logistic regression in a multi-class context whose function is:

$$softmax_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{10.4}$$

for class j

This takes a generic vector z that can take any value and transforms it into a vector of values between 0 and 1. In our case, it transforms scores (unnormalized probabilities $f(x_i, W) = Wx_i$) into probabilities.

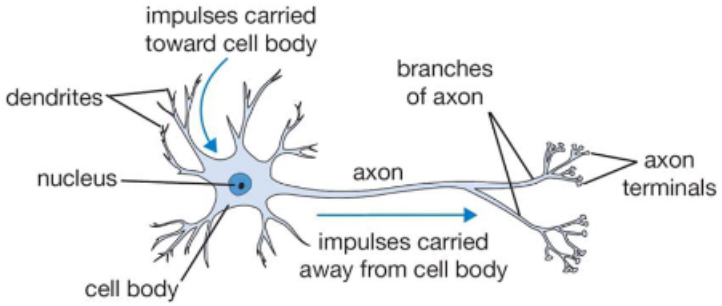
To train this classifier we use the cross-entropy loss:

$$L = -\sum_i y_i^{true} log(y_i^{pred}) \tag{10.5}$$

where y_i^{true} is the ground truth distribution while y_i^{pred} is the prediction distribution. In practice, y_i^{true} is a one-hot vector that selects the correct label.

10.1 Neuron and Perceptron

A neuron is the basic computational unit of our brain and follows the following "rule": it receives input signals from its dendrites that arrive at its nucleus which evaluates the "strength" of the combination of input signals and if it exceeds a certain threshold "fires" an output signal along its axon which can eventually branch out to connect to the various dendrites of other neurons through synapses (axon-dendrite connection sites).



More formally, we can model a single neuron as follows:

- From the axon of another neuron comes an input signal

$$x_i \tag{10.6}$$

- Associated with the i-th synapse there is a weight

$$w_i \tag{10.7}$$

- Along the i-th dendrite we will therefore have obtained the product

$$w_i x_i \tag{10.8}$$

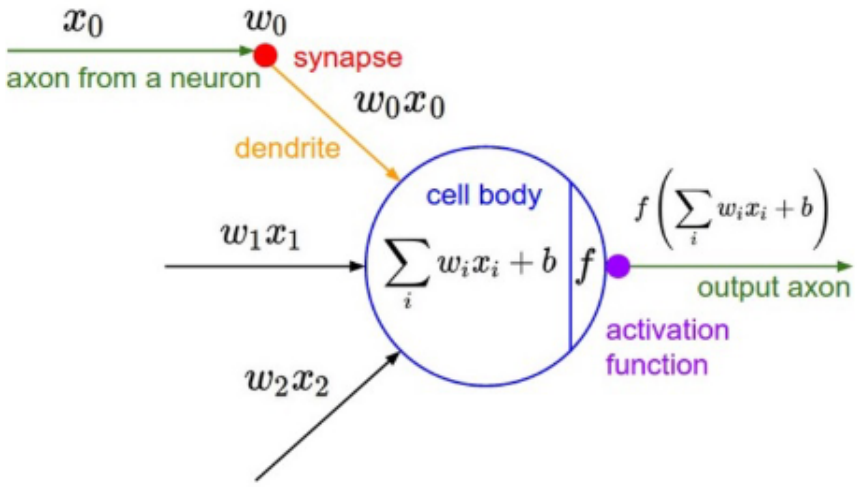
- All dendrites end in the nucleus where their linear combination is calculated

$$\sum_i w_i x_i + b \tag{10.9}$$

- Along the output axon we will have the result of a so-called non-linear activation function calculated on the combination:

$$f(\sum_i w_i x_i + b) \tag{10.10}$$

  This is nothing than a threshold function like logistic regression (binary case) or Softmax (multiclass case).

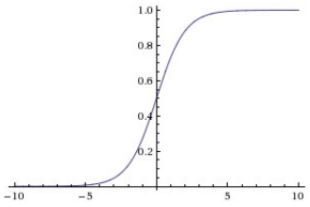


Activation Functions

As mentioned, these are non-linear functions calculated on the output of each neuron, let's see the most famous ones:

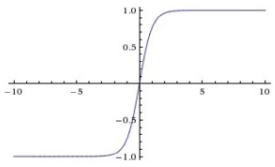
- **Sigmoid** which has the form

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{10.11}$$



  It has two main drawbacks:
  - Saturates and kills the gradient: from a certain input value onwards it always returns the same value in output and the gradient is always zero. Cause of this we can't compare scores because probability, whatever the score, is always the same.
  - The output is not centered with respect to zero
- **Tanh** (hyperbolic tangent) which has the form

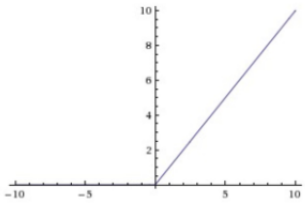
$$tanh(x) = 2\sigma(x) - 1 \tag{10.12}$$



It's a sigmoid centered at zero consequently it takes values in the range $[-1,+1]$ despite this it can still saturate and kill the gradient.

- **ReLU** (Rectified Linear Unit) which has the form

$$ReLU(x) = max(0,x) \tag{10.13}$$



As output it doesn't give a probability but returns the score itself only if it's positive (lose information on negative values). Therefore it applies only to non-terminal neurons of a neural network (we'll see what it is).

Why using non-linaer activations at all?. The reason is that the composition of linear functions is a linear function and therefore without non-linearity can be reduce to neural network formed by only one layer. Indeed given

$$f(x) = \phi(W_2\phi(W_1x)) \tag{10.14}$$

if I got rid of the non-linear activations ϕ I would get:

$$f(x) = W_2W_1x = Wx \tag{10.15}$$

where

$$W = W_2W_1 \tag{10.16}$$

that is is clearly linear.
So without a non-linear activation function all the network became a one layer network, with less parameters and identical to logistic regression, that means it can create only linear boundaries (not efficient).

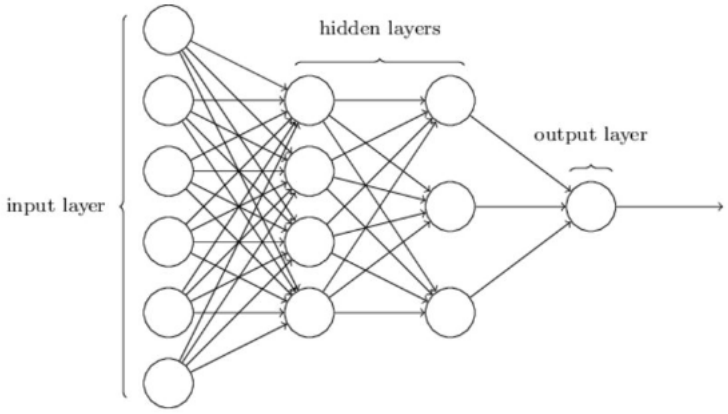
10.2 Neural Networks

We have already mentioned the concept of neural networks several times without ever explaining their meaning. This is nothing more than a set of neurons connected to each other in the form of a graph. In particular, we have:

- Input layer that has as many neurons as features
- Hidden layers, each with a number of neurons that is a hyperparameter
- Output layer that has as many neurons as there are classes

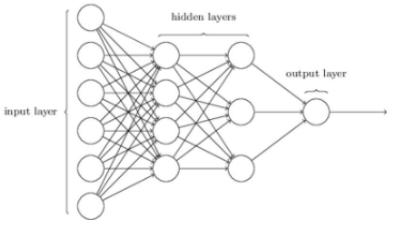
To calculate the number of parameters related to a layer i, I must count how many neurons there are in the previous layer $i-1$ and consider that in each neuron of my layer I will have as many weights as there are neurons in the previous layer and only one bias. To find the number of parameters of the entire layer, I will therefore have to calculate:

$$\#Parameters_i = \#Neurons_i \cdot (\#Neurons_{i-1} + 1) \tag{10.17}$$



To find the total number of parameters of the network, just sum the parameters of all layers from the second onwards because the input one has no previous layers.
Doing a dimensional analysis of the weight matrices, we notice that each matrix has as number of rows the number of neurons of the next layer and as number of columns the number of neurons of its layer.

- ϕ is the activation function
- $\mathbf{x} \in \mathbb{R}^6$ is the input
- $\mathbf{W_1} \in \mathbb{R}^{4\times6}$ are the weights of first layer
- $\mathbf{W_2} \in \mathbb{R}^{3\times4}$ are the weights of second layer
- $\mathbf{W_3} \in \mathbb{R}^{1\times3}$ are the weights of third layer



Notice that to ease the notation biases have been incorporated into weight matrices W.

The process of calculating the output of the network given its input is called **forward propagation**, whose formula is of the type:

$$out = \phi(W_3\phi(W_2\phi(W_1x))) \tag{10.18}$$

Representational Power

Theorem 10.2.1 — Universal approximator. It has been demonstrated that any function $f(x)$ can be approximated with arbitrary precision ε by a neural network $g(x)$ with at least 1 hidden layer for which:

$$\forall x,\ |f(x) - g(x)| < \varepsilon \tag{10.19}$$

This is why this type of neural networks are called **universal approximators**. In practice, however, this is not true as because in computer numbers are discrete; so more hidden layers you use less will be the difference between two function.

Parameters and Hyperparameters

As anticipated, the hyperparameters of a neural network are the number of layers and the number of neurons for each layer.
We said that the more layers I have, the more the capacity of the network will increase and I will be able to better represent any function, but by doing so, the tendency to overfit also increases if

regularization techniques that we will see are not properly used. In practice, usually networs are made as big as computational budget allows.

Objective Function

Generally speakng, the objective function or loss function measures the quality of the function I want to represent:

$$L(\theta;x,y) = \frac{1}{N}\sum_i L_i(\theta;x_i,y_i) + \lambda\Omega(\theta) \tag{10.20}$$

where N is the number of training examples, θ is the set of network parameters and λ weighs the strength of regularization. The formula can be divided in two parts:

- **Data term**

$$\frac{1}{N}\sum_i L_i(\theta;x_i,y_i) \tag{10.21}$$

  Measures the goodness of the model's predictions with respect to the labels
- Explicit **Regularization** term that contains complexity of parameters choosen.

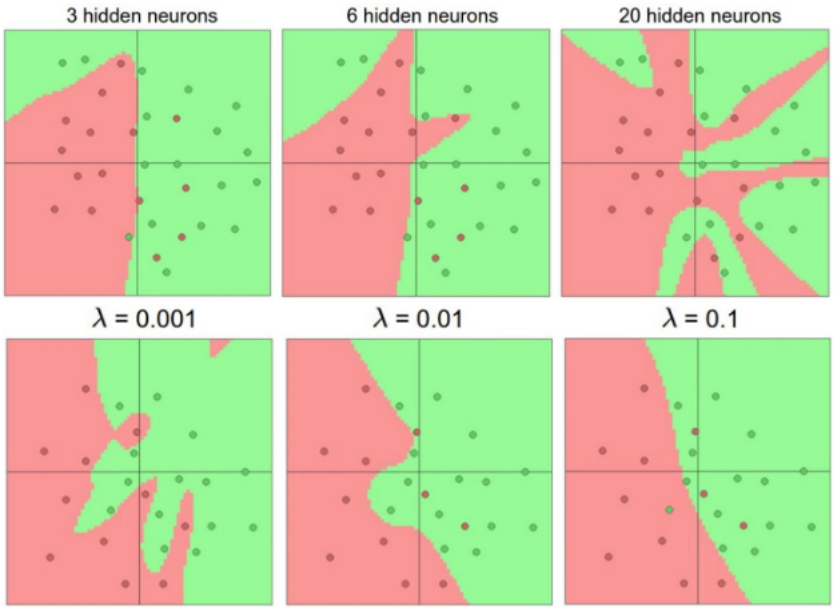
$$\lambda\Omega(\theta) \tag{10.22}$$

  Depends only on the parameters and tries to mitigate the risk of overfitting, in fact the larger λ is, the more the separation surface moves towards generalization and away from overfitting.
  1. L^2 norm, continuous version of the modulus operator that gives up the concept of hard sparsity, so it's more permissive.

$$\Omega(\theta) = \sum \theta^2 \tag{10.23}$$

  2. L^1 norm, introduce sparsity (a lot of zeros) and its derivative is not continuous

$$\Omega(\theta) = \sum |\theta| \tag{10.24}$$



The type of Loss depends on the task we want to solve:

- Classification
  - **Hinge loss**

$$L_i = \sum_{j \neq y_i} max(0, f_j - f_{y_i} + 1) \tag{10.25}$$

    Where f_{y_i} is the score of the correct class and f_j is that of an incorrect class. I will have an output different from 0 only if these are distant in modulus more than 1.
  - **Softmax loss**

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{10.26}$$

    It is the cross entropy loss calculated after applying a Softmax and is the score of the correct class divided by the sum of all the others except the correct one.
- Regression
  - **Mean squared error** (MSE)

$$L_i = \|f - y_i\|_2^2 \tag{10.27}$$

    I learn the function thanks to the input-output ratio

So, the optimal parameters we want to find in the training phase are those minimize the value of loss function:

$$\theta^* = argmin_\theta \left(\frac{1}{N}\sum_i L(\theta; x_i, y_i) + \lambda \Omega(\theta)\right) \tag{10.28}$$

This gradient cannot be calculated analytically, so I will have to use a gradient descent algorithm. How can we compute in automatic way the gradient? We have to use an algorithm called **backpropagation**

10.3 Backpropagation

The procedure of this algorithm is based on the chain rule of calculus:

$$\frac{df(g(x))}{dx} = \frac{df(x)}{dg(x)}\frac{dg(x)}{x} \tag{10.29}$$

This algorithm proceeds backwards with respect to the flow of computations performed by the Loss and at each step, locally, each neuron update its set of parameters. For these reasons backpropagation is a local process where each neuron is completely unaware of the topology of the entire network in which it is embedded.
The whole thing is based on the following assumption, each neuron is able to perform two derivatives:

- $\frac{\partial h_{i+1}}{\partial W_{i+1}}$ derivative of its output with respect to its weights
- $\frac{\partial h_{i+1}}{\partial h_i}$ derivative of its output with respect to its inputs

After that each neuron updates its parameters with the following formula (gradient descent):

$$W_i^{new} = W_i^{old} - \eta \frac{\partial L}{\partial W_i^{old}} \tag{10.30}$$

■ **Example 10.2** Let's see backpropagation in practice with an example: consider a network with three layer and one neuron for each layer (simple use case). To compute backpropagation I have to to perform the following calculations:

$$W_3^{new} = W_3^{old} - \eta \frac{\partial L}{\partial W_3^{old}} \tag{10.31}$$

$$W_2^{new} = W_2^{old} - \eta \frac{\partial L}{\partial W_2^{old}} \tag{10.32}$$

$$W_1^{new} = W_1^{old} - \eta \frac{\partial L}{\partial W_1^{old}} \tag{10.33}$$

$$\tag{10.34}$$

The unknown parameters, here, are the partial derivative respect weights:

$$\frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial W_1} \tag{10.35}$$

Let's see how compute these values:

1. The first derivative to compute refer to last neuron (we are going in back direction)

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial h_3} \frac{\partial h_3}{\partial W_3} \tag{10.36}$$

   The Loss directly depends on h_3 and w_3 so I can calculate the first derivative and the second derivative easily (hypothesis). I backpropagate $\frac{\partial L}{\partial h_3}$ because as we will see it will be needed by the neuron in the previous layer to do its calculations.

2. Second layer

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial W_2} \tag{10.37}$$

   The Loss does not directly depend on h_2 so I have to apply the chain rule:

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial h_3} \frac{\partial h_3}{\partial h_2} \tag{10.38}$$

   So I get:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial W_2} \tag{10.39}$$

   The first derivative had been backpropagated so I don't even have to recalculate it, I can calculate the second and third because a neuron must know how to do it by hypothesis. I backpropagate $\frac{\partial L}{\partial h_2}$

3. The last calculation

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial W_1} \tag{10.40}$$

   The Loss does not directly depend on h_1 so I have to apply the chain rule:

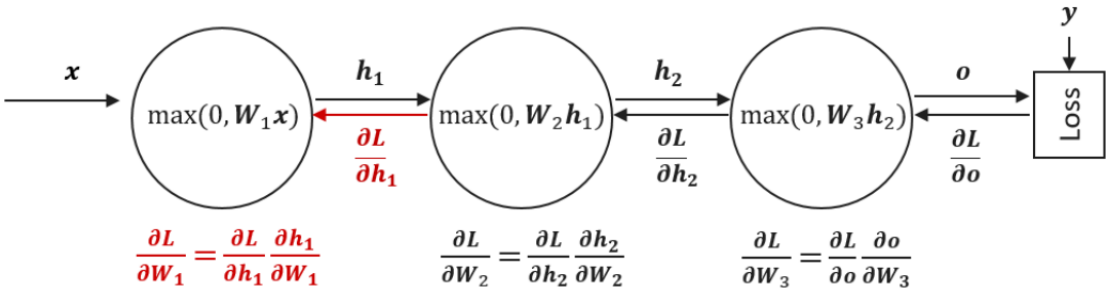
$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial h_1} = \frac{\partial L}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \tag{10.41}$$

So I get:

$$\frac{\partial L}{\partial W_1} = \underbrace{\frac{\partial L}{\partial h_3}\frac{\partial h_3}{\partial h_2}}_{\frac{\partial L}{\partial h_2}}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_1} \tag{10.42}$$

I don't have to calculate the first two derivatives being $\frac{\partial L}{\partial h_2}$ which was backpropagated to this layer, while I can calculate the last two by hypothesis.

■



Pay attentation to regularization factor in this example because it can be easily computed due to the fact that depends only on the local weight.

Weight Initialization

Initializing weights randomly has proven to be very important for breaking symmetries. If all weights were initialized with the same values, each neuron would perform identical calculations, i.e., same output, same gradient, and thus same update steps. A common practice is therefore to initialize weights with small random values (sampled from a uniform distribution) around zero and all biases to 0. One could then divide the value by the fan-in of the neuron, i.e., the number of inputs, to scale the sum of weight · input products.

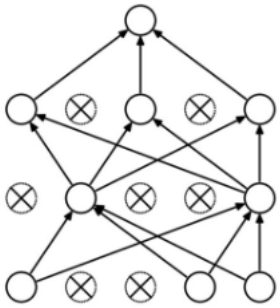
Another idea is to use a pre-trained networks to initialize randomly the initial weights.

10.3.1 Regularization

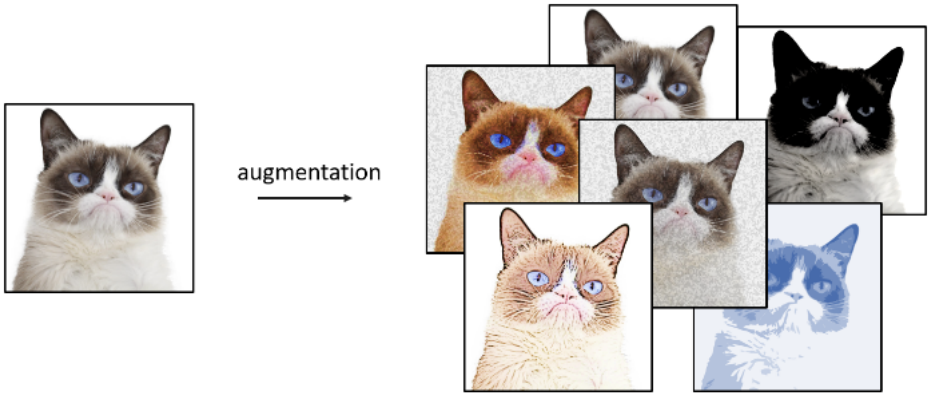
We know that neural networks, having high representative capacity, are prone to overfitting (they copy the input-output relationship directly into the neuron parameters without learning a true function, but rather an indexing that works like a memory). To avoid this, regularization techniques are introduced. There are two types: explicit and implicit. An explicit regularization technique is the explicit regularization term seen previously $\lambda\Omega(\theta)$
Let's now look at implicit techniques, i.e., techniques aimed at complicating the network to avoid overfitting:

- **Dropout:** during training, each neuron has a drop probability p, meaning it has a probability p of being set to zero (dropped). In this way we leverage chaos and diversity, preventing that a node use always the same value; so, at each step, we're working on a different sub-network, thus acting on a different portion of parameters. During testing, dropout is not applied, so it's like making an average prediction on a set of sub-networks.

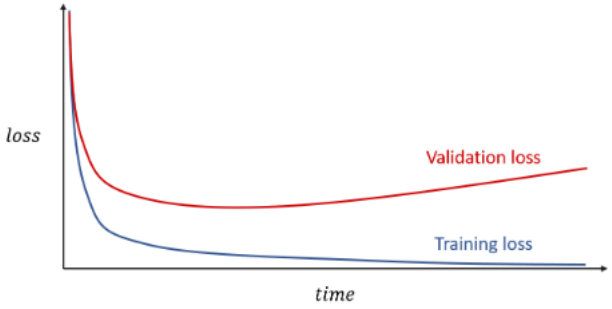
Usually it used at the end of the network because at the beginning the network have to learn the max "knowledge" (to prorogate "knowledge" to next layers). Dropout can replace the explicit regularization term or can be used alongside it.



- **Data augmentation:** consists of generating new training examples by applying transformations to my training inputs while preserving the label of the original example, of course. In this way, I have created a "fake" but larger dataset that will be difficult to memorize due to its size. The important thing in this case is to keep the sense of the dataset intact, i.e., I must not semantically ruin the input. It works well with images.



- **Early stopping:** the most common symptom of overfitting would be that while the Loss on the training set continues to decrease over time, the one on the validation set at some point stops decreasing and starts to increase. There is a technique called Early stopping that stops the training process when the validation loss stops decreasing and starts to increase. The number of training epochs in which the validation loss decreases or remains constant is called patience.



11. Convolutional Neural Networks

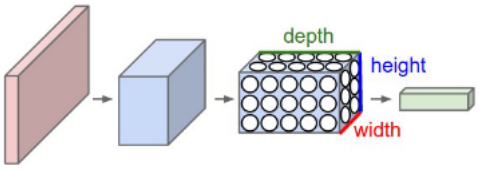
Convolutional Neuural Networks are very similar to ordinary Neural Networks.

- They are composed of neurons, each with weights and biases to learn
- Each neuron receives inputs, performs a dot product, and provides an output to which non-linearity is applied (otherwise the multi-layer network would collapse into a large overparameterized layer)
- The entire network represents a differentiable function

The difference with MLP is that this time the inputs are images (or generally data with spatial support) that have high dimensionality and required more computational power. If we used MLP with images, the number of parameters would be too high, which is why this architecture is introduced.

Given an image with a certain width, height, and depth (in this section, by depth we won't mean the depth of the neural network but the number of channels in an image, for example, if it's RGB, the depth is 3), the objective of this architecture is to manipulate images through differentiable operations to decrease width and height and increase the number of channels (which will have a different meaning) in order to extract a one-dimensional or two-dimensional feature vector.

Therefore, in CNNs, each neuron is organized in three dimensions (width W, height H, and depth C), and we will have various layers one after another, each taking a 3D volume as input and providing as output the result of a differentiable function (which may or may not have parameters) applied to the input that will thus be transformed.



11.1 Convolutional Layer

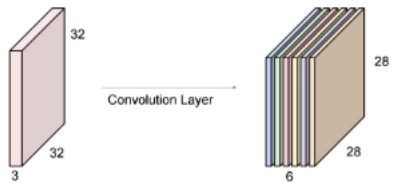
The main building block of CNN is the **convolutional layer**: each of these layers is equipped with a set of filters (or kernels) that will be learned.

Definition 11.1.1 — Kernel. A filter (or kernel) is a square-shaped object (like a grid) always having odd height and width to ensure there is always a center.

During the forward pass, each filter is convolved with the input volume producing a two-dimensional output with decreased height and width. For each filter, a two-dimensional output is produced, so having a set of filters, the final result will be obtained by stacking the individual two-dimensional outputs of each filter, obtaining a volume with decreased height and width and increased depth/channel (because we're stacking them), particularly equal to the number of filters in that layer.

Each convolutional layer has 4 fundamental hyperparameters:

- Number of **filters** N (number of output channels)
- **Kernel size** K (so we'll have a K x K filter)
- **Stride:** since a filter is smaller than an image, when performing the convolution operation, we overlap the filter with the first block of the image and gradually move it until covering the entire input image. The stride tells us how many pixels to move forward, and this number is responsible for reducing dimensions W and H; in fact, if we move 2 pixels at a time, we'll halve the dimensions.
- **Padding**: if a filter doesn't fit a finite number of times in the input image, we introduce a frame of artificial values around the image that allows us to apply the filter even near the edges where we previously exceeded bounds. There are various types of padding:
  - Zero padding: all values are zeros
  - Same padding: values are those of the last copied row
  - Mirror padding: values are copied by mirroring from our kernel



Result of $N = 6$ filters of kernel size $K = 5$x5 convolved on input image.

Convolution

The discrete convolution operation is a combination of sums and products between input image I and kernel K (defined respect to the center of the kernel):

$$F(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{11.1}$$

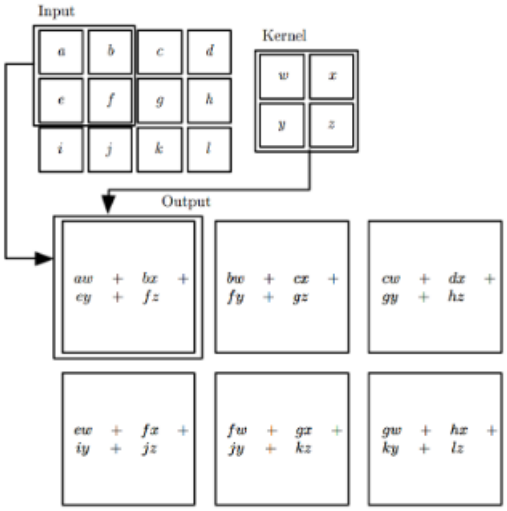
If we have K=3, in this case, indices m and n go from -1 to +1, and the final result is the product between each image value and its corresponding kernel value. The result is placed where the kernel origin is located, therefore at the center of the kernel.

Often, for computational reasons, it is implemented as a **cross-correlation** operation:

$$F(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{11.2}$$

If we have K=3, in this case, indices m and n go from 0 to 2, and the final result is the product between each image value and its corresponding kernel value. Instead, this time, the result is placed

in the top left where the kernel origin is located. This method is more used because, not having to manage negative coordinates, it can be implemented as a row-by-column product of the unrolled image and kernel.

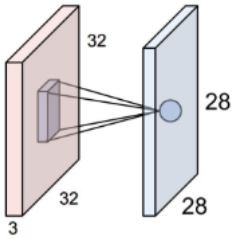


Receptive Field

We could say that a CNN neuron performs the same operation as a DNN neuron, namely:

$$\sum_i w_i x_i + b \tag{11.3}$$

However, each CNN neuron is only locally connected to the input volume. In particular, we know that each value of the output image from the convolution operation is calculated only on a small region of the input image. This region, called the **Receptive Field** of the neuron, summarize the information extract from the previous layer.
As the number of layers increases, the receptive field on the original image increases.



Each kernel of a layer extracts some small pattern (a microstructure like a line in a certain direction) and the layer of the next level can combine that pattern to extract increasingly complicated patterns like shapes or objects.
The number of parameters in a CNN doesn't depend on the image pixels but only on the kernel dimensions and the number of different kernels I decide to use.
Convolution is a translation-invariant operation, meaning the kernel responds in the same way wherever the pattern is, even though from a semantic point of view this is not always good (we wouldn't want to recognize figures in the sky as humans), this problem is solved by the receptive field which will eventually have dimensions equal to the entire initial image thus having global information about the semantics of the image.

We said that the initial image has length, width, and number of channels equal to its depth so it's a tensor, and so far we've seen that convolution acts on a single channel. I could think of applying the same kernel to all channels but this way we would only be able to identify patterns that are in the planar version of the tensor, i.e., patterns that don't involve more than one channel simultaneously. This is fine if I want to find a round shape for example, indeed I find it whether it's red, blue, or green. If instead I wanted to recognize faces that are pink round shapes, I should take into consideration multiple channels simultaneously.
So in reality when we specify a kernel, for example 3x3, it's not properly 3x3 but it's a 3x3xchannels tensor. So each channel has its own filter and the results are summed together. More formally, the channels are filtered by independent filters K^c and the results are summed together.
The cross-correlation is therefore calculated as follows:

$$F(i,j) = \sum_c \sum_m \sum_n I(i+m, j+n, c) K^c(m,n) \tag{11.4}$$

This could actually be intuited because at the beginning we had said that the results of individual kernels are stacked to form the output with increased channel number equal to the number of kernels.
Now we can also better understand the dimension of the generic receptive field which will be K x K x c where K and c are the kernel size and depth at the previous layer.
To calculate the number of learnable parameters, given an input image of dimensions H_1, W_1, C_1 and N filters $K \times K$:

$$tot\_learnable = N * K * K * C_1 + N \tag{11.5}$$

That is, each filter has $K * K * C + 1$ bias and I multiply everything by the number of filters N.

11.2 Pooling Layer

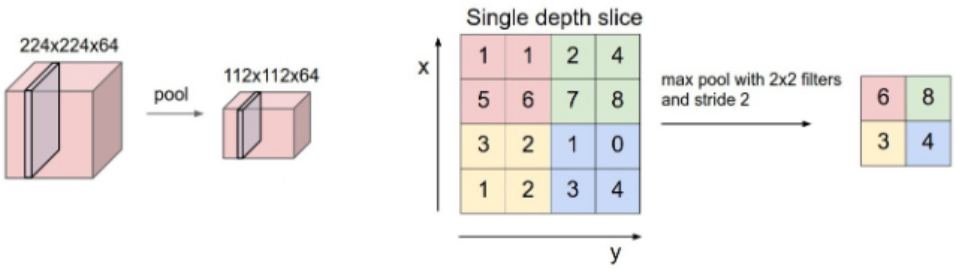
Its purpose is to reduce the W and H dimensions without increasing the number of channels and consists of applying a window of dimensions (Pool size) equal to K x K with a certain Pool Stride S. While sliding the window, I locally apply a function to only the pixels that are currently in the window. This pooling function can be:

- **Max** pooling:

$$h_i^n(x,y) = \max_{\bar{x},\bar{y} \in \mathbb{N}(x,y)} h_i^{n-1}(\bar{x},\bar{y}) \tag{11.6}$$

- **Average** pooling:

$$h_i^n(x,y) = \frac{1}{K} \sum_{\bar{x},\bar{y} \in \mathbb{N}(x,y)} h_i^{n-1}(\bar{x},\bar{y}) \tag{11.7}$$



We always try to have S = K to avoid overlapping windows when sliding.
The reasons for using pool layers are the following:

- Better localization of patterns we're looking for (for example, it removes what isn't the exact maximum, i.e., what doesn't perfectly overlap with our pattern, thus perfectly localizing)
- Reduced computational cost (less memory needed because we reduce image dimensions)
- Prevention of overfitting (because we don't copy all values, so we don't memorize)
- Increased receptive field for subsequent layers

Pooling should be avoided in cases of semantic segmentation because in that case, we don't want to reduce image dimensions, and even if we wanted to increase them again, we couldn't because, for example, the max is neither derivable nor invertible, so doing backpropagation we could at best have a "true" value and all others would have to be copied or set to zero, losing valuable information. This is why today it's preferred to do downsampling and upsampling using only stride (with S=2 indeed I halve the dimensions while with S=1/2 I double them) and not pooling, though losing its advantages like avoiding overfitting and better localizing filter responses.
Pooling is usually applied to the result of a convolution.

11.3 Activation Functions

After a convolutional layer (and possible pooling) there is an activation layer that applies a non-linear activation function like ReLU, sigmoid, or tanh. The best one is again ReLU because it doesn't saturate like the others and can introduce sparsity (if it receives a negative value it provides 0 so the neuron is off) and this is useful to avoid overfitting as already seen.
The reason for inserting such a non-linear function is the same as in DNNs, namely that the composition of linear functions is a linear function and therefore I could simplify the network to a logistic regression with a single layer.
Let's see how to calculate the output dimensions of a convolutional layer (not required for the exam). Given an input image of dimensions H_1, W_1, C_1 and N filters $K \times K$ with stride S and padding P, the output of the convolution between image and filter will give me a new image of dimensions:

$$H_2 = \frac{H_1 - K + 2P}{S+1} \tag{11.8}$$

$$W_2 = \frac{W_1 - K + 2P}{S+1} \tag{11.9}$$

$$C_2 = N \tag{11.10}$$

To calculate those of the image output from a pooling layer:

$$H_2 = \frac{H_1 - K}{S+1} \tag{11.11}$$

$$W_2 = \frac{W_1 - K}{S+1} \tag{11.12}$$

$$C_2 = C_1 \tag{11.13}$$

To calculate those of the image output from an activation layer:

$$H_2 = H_1 \tag{11.14}$$

$$W_2 = W_1 \tag{11.15}$$

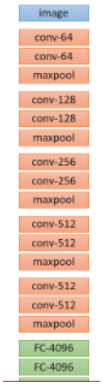
$$C_2 = C_1 \tag{11.16}$$

11.4 VGG

As an example, let's look at the high-performing **VGG network** developed for image recognition by the Oxford Vision Geometry Group in 2014. VGG is well-known for a variety of reasons:

- Performance of the network was great
- Pre-trained weights
- Architectural choices by the authors led to very neat network, used as guideline for a number of future works.

VGG architecture is composed by the following components:



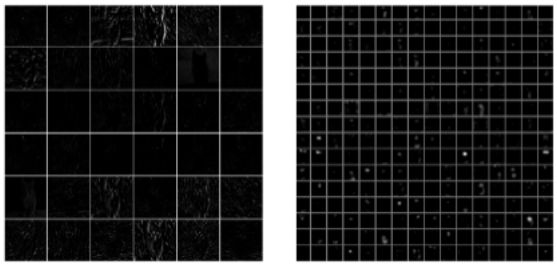
- Input: RGB images 224 x 224 preprocessed by subtracting the mean image
- **Convolutional filters**: K = 3 and S = 1, in total I have 10 convolutional layers with increasing number of kernels
- Spatial Pooling: max pooling performed with K = 2, S = 2 (halve the images), in total I have 5
- ReLU: follows each hidden layer
- **Fully connected layers**: these are MLPs and the first two layers (each followed by a ReLU) are formed by 4096 neurons while the last one is 1000 (as many as there are ImageNet classes) and is followed by a Softmax for classification

The orange blocks therefore are the convolutional part of feature extraction and serve only to transform the images (or structured data) into a feature vector that the green part will classify.

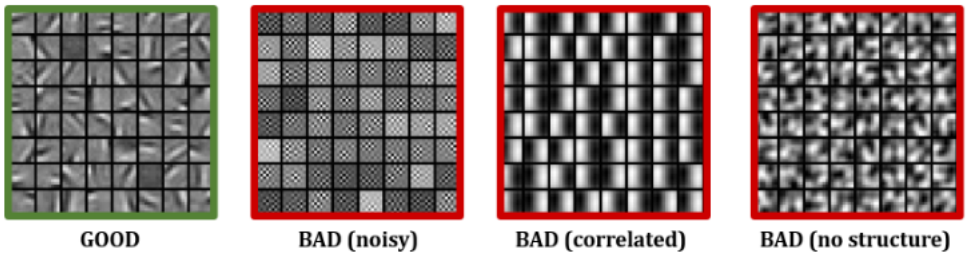
VGG has 138M parameters of which 100M are from the fully connected layers.

11.5 Interpretability

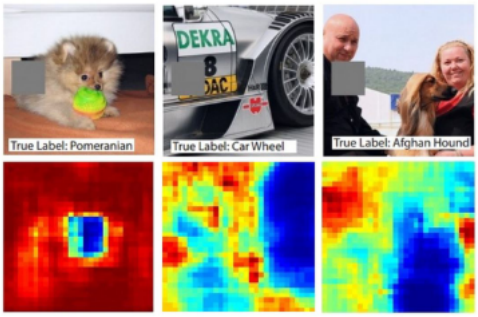
While linear models and decision trees are considered the "champions" of interpretability, CNNs have often been criticized for their lack of it as they are considered black boxes that give correct results without allowing us to know what happens inside. In reality, the formulas behind CNNs as we have seen are deterministic, the problem is that there are too many convolutions, activation maps, filters that are often operations or objects difficult to inspect (also because they often go beyond 3 dimensions) and moreover there is no semantics for intermediate features.
Despite this, various methods have been developed to see what a CNN learns.
One method allows **visualizing activations**, that is, during the forward pass I can see the filters that had a good correspondence with the image (light areas, there is coherence between filter and image) and dead filters (dark areas, no correspondence with my patterns).



Another method allows **visualizing weights**, that is, the patterns that are being learned. It's interpretable only at the first layer that works directly on images; on all subsequent ones having more channels I would have to inspect them in slices. It can be useful to see if the patterns are good, that is, if there are uncorrelated structures (patterns with different directions) and little noise. If there were no structures, there was noise, or the patterns were correlated, we would be looking at poor weights.



A final method we'll look at allows seeing which portion of the image contributed most to making a certain prediction. It consists of sliding a rectangle along the image and for each zone it covers, I look at the output value of the correct class and assign it to the output image then proceed to the next zone. In the end, I'll have an image with colors between red and blue. The red zones are those that even when covered give me high probability that the image belongs to the correct class (I'm not covering the target), the blue ones are those that give low probability, meaning I'm covering the target.

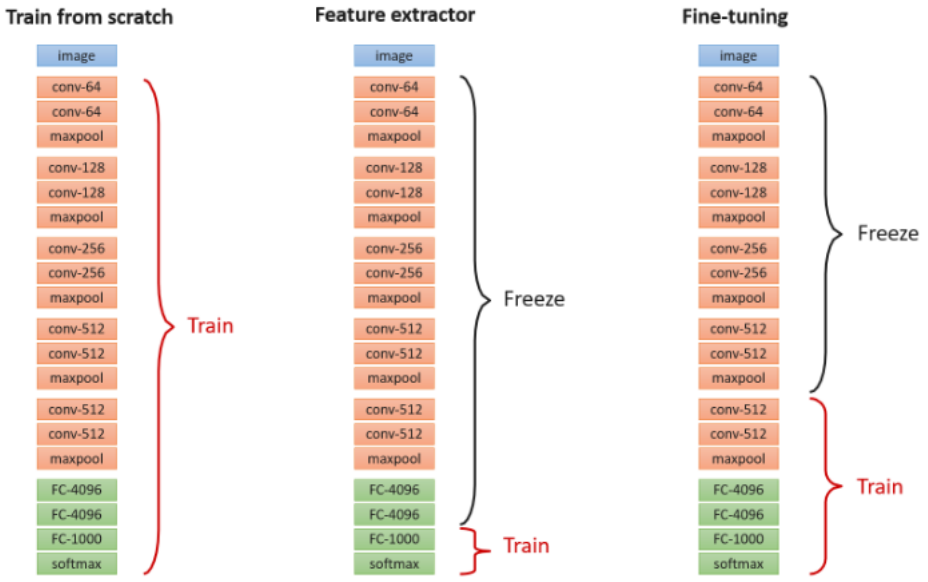


11.6 Transfer learning

For most applications, there is no reason to retrain an entire CNN from scratch. Instead, it makes sense to use the first layers of known convolutional networks (VGG or ResNet, pre-trained on ImageNet) as feature extractors because the filters they use are general and work on all images, and then possibly train only the subsequent layers. This technique is called **Transfer learning**. Expanding on this concept, depending on how much data I have, there are three possibilities:

- Lots of data: I can train a CNN from scratch. This method is called **Train from scratch**
- Medium amount: I use the pre-trained first convolutional layers of a known CNN and then train only the last convolutional layers and all classification layers. This method is called **Fine-tuning**
- Little data: I use all convolutional layers and all classification layers except the last one pre-trained from a known CNN and then train only the last classification layer. This method is called **feature extractor**.

The pre-trained layers are called "frozen".



Beyond dataset dimensions, the choice of method also depends on the similarity between my new data and the data used to pre-train the network. Indeed, if I had different domains, I wouldn't get great results. Therefore, in conclusion, I will have 4 possibilities:

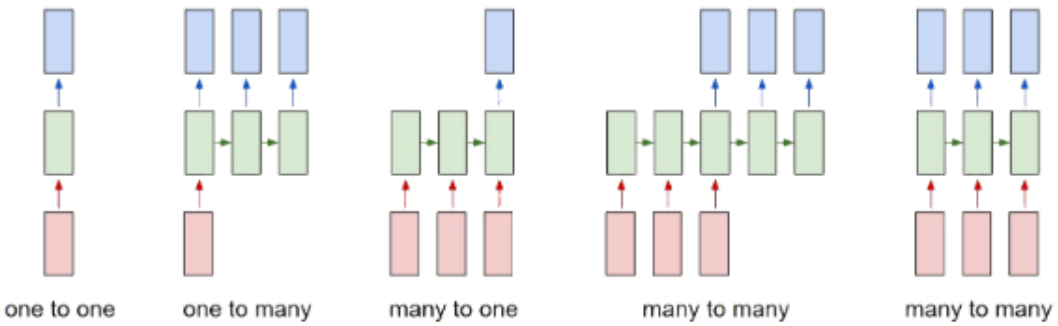
- Lots of very different data: Train from scratch (better if initialized with the first pre-trained weights because the first ones are generic)
- Lots of very similar data: Fine tuning freezing as little as possible (load the entire network but still allow the gradient to change the parameters)
- Little similar data: Feature extractor
- Little different data: Fine-tuning freezing as much as possible (train also the last convolutional layers in addition to the classification ones)

12. Recurrent Neural Networks

In feedforward neural networks, computation flows from input to output through intermediate layers. However, there exists a family of models that present feedback connections where the model's output is fed back into the model as input. These models are called **Recurrent Neural Networks** (RNN).

RNNs are specialized in processing sequential data (sequences of feature vectors ordered with respect to time) and can take both sequential and non-sequential data as input and provide both sequential and non-sequential data as output, thus various types of RNNs exist:

- One-to-one: non-recurrent network (purely forward)
- One-to-many: takes a single input and provides a sequence (used in text generation)
- Many-to-one: takes a sequence as input and provides a single value as output (used in time series prediction)
- Many-to-many: takes a sequence as input and provides another sequence as output (for example, input consists of observations up to the present moment and output is the prediction of the future, as in trajectory prediction, or Google Translator translations). Many-to-many architecture can be asynchronous or synchronous.

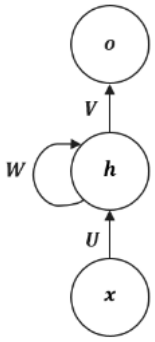


12.1 Vanilla RNN

Vanilla RNNs are provided with three sets of parameters:

- U: set of parameters that maps inputs to hidden states
- W: parameterizes transitions between different hidden states (from previous state to current)
- V: set of parameters that maps hidden states to outputs

The recurrent unit is shown in the figure below.



U, V, and W are parameter matrices that remain the same for all time steps.
The hidden state at the current time step t, $h^{(t)}$, is the representation of processed information over time that takes into account the input at the current time step, $x^{(t)}$, and the state at the previous time step $h^{(t-1)}$. This is what creates the recurrence.
The system dynamics can be represented as follows:

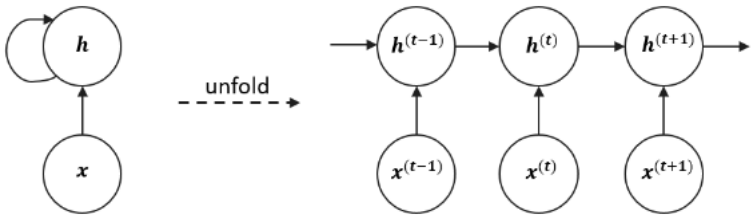
$$\begin{cases} h^{(t)} = \phi(Wh^{(t-1)} + Ux^{(t)}) \\ o^{(t)} = Vh^{(t)} \end{cases} \tag{12.1}$$

This structure is called a **Markovian** process and assumes that all information is contained in the state at the previous instant. In reality, this summary will have information loss due to the finite dimensions of h.

■ **Example 12.1** The state of a recurrent network is like arriving late to the cinema and asking your friend for a summary who, to be quick, will have to tell you only the salient information, losing some details. ■

Unrolling the graph as a function of time-steps gives us a repetitive sequential structure

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{12.2}$$



12.1.1 BackPropagation Through Time

The ability to unroll a recurrent graph into a directed acyclic graph (DAG) allows us to train RNNs in terms of standard backpropagation. Since going backward means going to previous time steps, this is called **BackPropagation Through Time** (BPTT).
Given a differentiable Loss L, to find the optimal parameters we must calculate the following derivatives:

- $\frac{\partial L}{\partial V} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial V}$, L depends only on o and o depends only on a single V.
- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h^{(3)}}\sum_{k=0}^{3}\left(\frac{\partial h^{(3)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial W}\right)$, L depends only on o but o depends on all W at all previous time steps.
- $\frac{\partial L}{\partial U} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h^{(3)}}\sum_{k=0}^{3}\left(\frac{\partial h^{(3)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial U}\right)$, L depends only on o but o depends on all U at all previous time steps.

Let's look at some of these calculations in detail.By the chain rule:

$$\frac{\partial h^{(3)}}{\partial h^{(1)}} = \frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial h^{(1)}} \tag{12.3}$$

Recalling that

$$h^{(t)} = \phi(Wh^{(t-1)} + Ux^{(t)}) \tag{12.4}$$

and ignoring ϕ for a moment, we can say that:

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} \cong W \tag{12.5}$$

Therefore

$$\frac{\partial h^{(3)}}{\partial h^{(1)}} = \frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial h^{(1)}} \cong W^2 \tag{12.6}$$

This means that the derivatives of the Loss with respect to U and W are increasingly large powers of W as we go back in time.
Depending on the values within matrix W, two problems could arise:

- **Vanishing Gradient:** if the values are $\in (0,1)$, these tend to zero very quickly as the power increases. This means that where I have higher powers, therefore at initial time steps, I will have null values that don't count in the summation, resulting in a gradient that vanishes over time, giving little weight to initial events.
- **Exploding Gradient:** if the values are > 1, I will have the opposite problem. This means that where I have higher powers, therefore at initial time steps, I will have high values, resulting in a gradient that explodes over time, giving too much weight to initial events.

It would be desirable for all events to have the same weight, which is why due to these two problems, Vanilla RNNs are no longer used, and more advanced architectures are used instead.

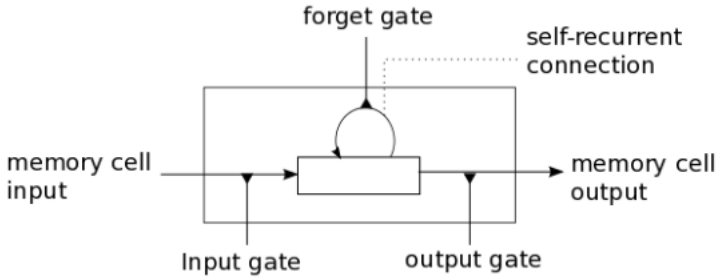
12.2 Advanced Recurrent Architectures

Advanced architectures, to avoid exploding or vanishing gradients, complicate state management. In **Long Short-Term Memory** (LSTM) and also in **Gated Recurrent Unit** (GRU), a trainable gating mechanism is introduced.

Definition 12.2.1 — Gate. A gate is a learnable switch that can control the flow of information to the memory cell and is implemented through a vector of parameters between 0 (completely closed gate) and 1 (completely open gate).

We have three different gates:

- Input gate
- Forget gate: allows resetting (forgetting what has happened up to that moment) in case of gradient exploding
- Output gate



Let's look at the LSTM model equations:

$$\begin{cases} i = \sigma(x^{(t)}U_i + s^{(t-1)}W_i) & \text{input gate} \\ f = \sigma(x^{(t)}U_f + s^{(t-1)}W_f) & \text{forget gate} \\ o = \sigma(x^{(t)}U_o + s^{(t-1)}W_o) & \text{output gate} \\ g = tanh(x^{(t)}U_g + s^{(t-1)}W_g) & \text{candidate state} \\ c^{(t)} = c^{(t-1)} \odot f + g \odot i & \text{memory cell at current time} \\ s^{(t)} = tanh(c^{(t)}) \odot o & \text{next state} \end{cases} \tag{12.7}$$

Where $c^{(t-1)}$ is the state at the previous time step and $\odot$ denotes element-wise multiplication (Hadamard product).

Explanation

The first three are the gate equations and are similar to the Vanilla case with the difference that each gate has its own set of parameters, as indicated by the indices on U and W.

The gates will be multiplied element-wise with other functions of the LSTM model, acting as regulators. Thanks to the sigmoid function, which returns a value between 0 and 1, they control how much input and output to pass through and how much memory from the previous step to overwrite (forget).

The fourth equation describes the candidate state and has a form similar to the vanilla case.

The last two equations update the memory cell c. In particular, in the first one, the forget gate controls how much memory from the previous step $c^{(t-1)}$ to keep, while the input gate controls the amount of the new candidate state to flow into memory. In the second one, the hidden output state is calculated, and specifically, the output gate regulates how much of the current cell memory to pass to subsequent layers.

13. Unsupervised DeepLearning

As previously mentioned, in unsupervised learning, we have data but no labels, so we must find correlations between the data without knowing the task in advance. This approach was introduced because data annotation is very expensive (both in terms of time and money) and is not always possible.

- It is often easier to obtain unlabeled data than labeled data; furthermore, unlabeled data is much more readily available and often free
- Frequent structures and hidden patterns are already present in data, regardless of presence or not of the supervision signal. Unsupervised learning involves learn interesting properties of data distributions (data have some regularity/pattern).
- It is a good strategies to pre-trained the networks
  - **Optimization:** unsupervised pre-training helps in initializing the parameters.
  - **Regularization:** unsupervised learning has been shown to be a good regularizer for supervised learning (generalize and try to reduce overfitting).

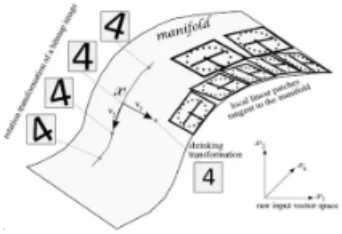
■ **Example 13.1** Let's take an example and consider MNIST, a database of 28 x 28 handwritten digit images. Assuming each pixel can take two values, the possible combinations will be:

$$2^{28\times 28} = 2^{784} = 1.01 * 10^2 36 \tag{13.1}$$

The problem is that most of these are invalid inputs that reduce space available and performance. ■

As for unsupervised machine learning, also here we make the **manifold assumption** that assumes that valid data lies on a manifold of much lower dimensions than the entire input space, meaning the actual degrees of freedom (i.e., the variability) of the data are much fewer.
When data lies on a manifold of lower dimensions than the input space, it becomes more natural to represent the data in terms of coordinates in the manifold rather than coordinates in $\mathbb{R}^n$.
A classic unsupervised learning task is to find the manifold that best represents my input data. We are looking for a representation with the following properties:

- Preserve as much information as possible
- Add constraints that make the representation simpler and more accessible than the original data (for example, fewer features)



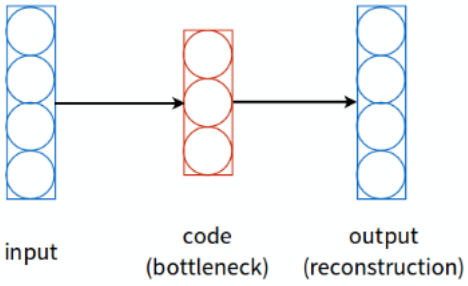
- Be task-specific or task-agnostic

Therefore, I can have various types of representations:

- **Lower-dimensional** representations: compress information
- **Sparse** representations: most features have zero value
- **Independent** representations: attempt to disentangle the underlying factors of variability in the data distribution so that the dimensions of the new representation are statistically independent. An example is PCA eigenvectors.

Dimension reduction techniques, for example, create a lower-dimensional space where elements have fewer or weaker dependencies compared to the original space. In doing so, I force data compression and implicitly identify and remove redundancies. An example is PCA, which learns a lower-dimensional representation whose elements have no linear correlations with each other.
Now suppose we want to implement PCA with a neural network. By relaxing some constraints, we get a network composed of:

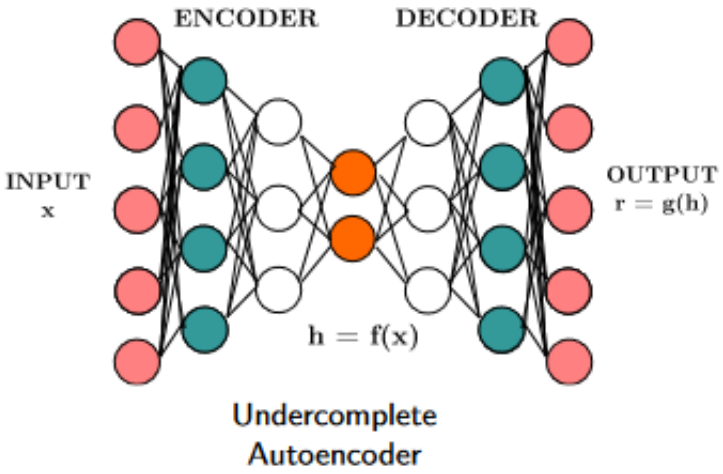
- An input layer
- A layer called code or bottleneck having fewer neurons than the input layer
- An output layer called reconstruction having the same number of neurons as the input and trying to reconstruct the input after it has been mapped to a lower number of neurons (like projection onto lower dimensions in PCA)



If the input and output layers are linear and we minimize the squared reconstruction error, we find that the M hidden units will create the same space that the M principal components would have created, with the only difference being that we haven't forced orthogonality between features, so they could have the same variance and represent the same informational content (at least in part).

13.1 Autoencoders

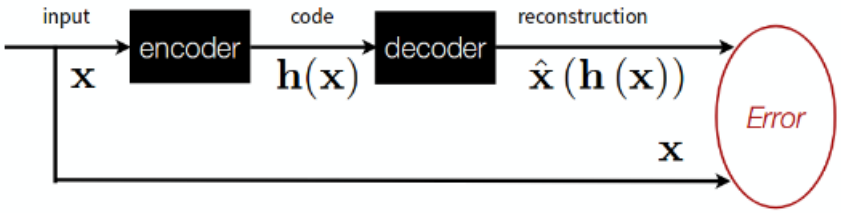
An **AutoEncoder** (AE) is a feed-forward neural network (convolutional or dense but not recurrent) trained to take the input, process it with an encoding network (encoder), obtain a reduced representation called code denoted as h=f(x) or more compactly h(x), and then have another decoding network (decoder) that can reconstruct the input (approximately) from the code.
The reconstruction is denoted as $r = g(h) = g(f(x))$ or $\hat{x}(h(x))$



The code must contain the most salient features of the input.
If I use non-linear layers in the encoder and decoder, I can also represent data lying on non-linear manifolds, and in particular:

- The encoder maps from the data space to the manifold space
- The decoder performs the inverse transformation

Finally, a Loss (such as MSE) will be calculated between all reconstructed values and original ones.



The biggest risk of autoencoders is learning too much and able to performs only the identity function. This means that the autoencoder learns the reconstruction function $g(f(x)) = x$ copying the input directly to the output, making the bottleneck and autoencoder uselees. This overfitting problem happens if the encoder or decoder has too high capacity or if the bottleneck has more neurons than the input.

13.1.1 Undercomplete Autoencoder

It has the constraint that the code must have smaller dimensions than the input, and this forces the AE to capture only the salient information of my data.
The learning process aims to penalizes incorrect representations, minimizing

$$L(x, g(f(x))) \tag{13.2}$$

It can be considered a non-linear generalization of PCA.
However, if encoder and decoder have too high capacity, we can run into indexing.
Regarding this discussion, we can say that:

- If a model has too few parameters, it will underfit the training set

- If a model has too many parameters, it will overfit the training set

How to select the right capacity of a machine learning model?

A Deep Learning principle recommends providing enough capacity to fit even complex functions while forcing the model to use the available power wisely. Indeed, the best performance comes from large and well-regularized models.

13.1.2 Regularized autoencoders

Regularized autoencoders try to follow the above principle avoiding learn the identity function. Use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. Because the model are forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data

Sparse Autoencoders

Autoencoder in which, in addition to calculating the reconstruction error in the Loss, a sparsity term (sparsity penalty) is added.

$$L_{SAE} = \underbrace{D(x, g(f(x)))}_{\text{reconstruction error}} + \underbrace{\lambda |h|}_{\text{sparsity penalty}} \tag{13.3}$$

Thanks to this term, I can build an overcomplete representation of the input data without the risk of learning the identity function because minimizing the Loss means trying to reconstruct as best as possible while having as many neurons turned off as possible in the bottleneck. Note that here we act on h, the output of neurons after the activation function, and not on weights as in DNNs.

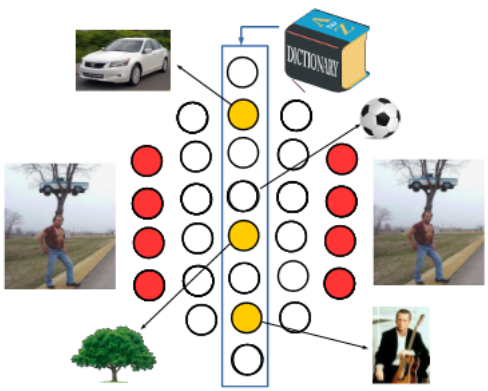
Sparse autoencoders are often used for classification and are particularly useful for building dictionaries (turning on the same neurons for the same category, neurons are not linked to semantics but only to the category). Indeed, they are able to extract the elementary and recurring concepts of the input.

To give a better idea of what it does, just think of a network that, given an input, finds a short list of terms (categories) that describe it taken from a dictionary.

The code has larger dimensions than the input precisely for this reason, because it acts as a dictionary, but thanks to the sparsity term, only the neurons deputized to classify something specific like a car or a tree are activated each time.

Since few neurons are activated for each category, these are forced to give an average response. Also, each neuron is linked to a group (cluster).

In the end, I won't represent my input image well from a semantic point of view but only its content in terms of categories.



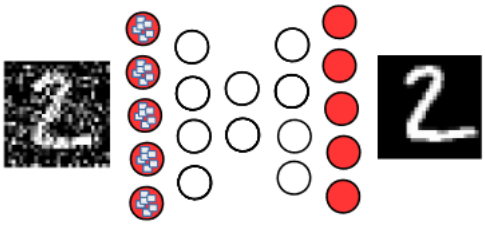
Denoising Autoencoders

To avoid learning the identity function and force the AE to find robust (important) features, the denoising autoencoder is trained to reconstruct the input x from a corrupted version $\hat{x}$.

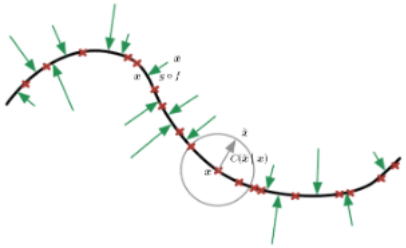
$$L_{DAE} = D(x, g(f(x))) \tag{13.4}$$

$$\hat{x} \sim \mathcal{N}(x, \sigma^2 I) \tag{13.5}$$

The corrupted input can be obtained by applying noise to x such as Gaussian white noise.
Therefore, this type of autoencoder will have to remove noise from the input.



Removing noise is useful for discovering important features because autoencoders focus on regular patterns that exist even in the presence of distortion. For example, in MNIST, it might recognize that there is always a black background and white figures.
Returning to the concept of manifold on which all valid data lies, corrupted data is slightly further from the manifold, and removing noise means bringing it back towards it, i.e., learning a vector field that brings all noisy data back to the manifold.



To do this, the DAE must have IMPLICITLY learned the structure of the manifold that represents the data, so learning this manifold means learning the structure of the probability distribution that generated my data $p_{data}(x)$.
I highlighted the word implicitly because, as mentioned, what a DAE actually learns is the vector field and not the manifold, and through it, the manifold is implicitly learned.

Contractive autoencoder

It's an AE that reconstructs the input from the code but makes the code insensitive to the input.

$$L_{CAE} = D(x, g(f(x))) + \lambda \left\| \frac{\partial f(x)}{\partial x} \right\|^2 \tag{13.6}$$

Also in this case, we have a regularization penalty that encourages the derivative of the code with respect to the input to be as low as possible. This forces the model to:

- Learn a robust function that doesn't change much when x changes slightly or is affected by small perturbations (if I memorized the code, it would change for every input)
- Reduce the number of effective degrees of freedom of the representation
- Represent only the variations necessary to reconstruct the training examples

Moreover, I focus on extracting features that reflect only the variations observed in the training set (there are indeed directions of variation for which the encoder should not be sensitive because they will never be observed).
The name contractive comes from the fact that the CAE deforms space, i.e., compresses the space around the manifold which is pushed (projected) towards it.
Since it is designed to resist input perturbations, in the end, I project a neighborhood of inputs into a smaller neighborhood of outputs. It is thanks to the contractive term that it is invariant to small perturbations.

13.2 Generative models

The autoencoders seen so far are discriminative, but there are also generative autoencoders capable of sampling new samples from a PDF to learn, unlike discriminative ones that only try to learn a function that discriminates between classes.
Generative modeling is therefore that area of Machine Learning that deals with finding distribution models $P(x)$ of data in a high-dimensional space. For example, when we talk about images, real images will have high probability and noisy images will have low probability. The goal is to generate new ones similar to my dataset but not exactly the same.
The DAE and CAE we said learn $P(x)$ only implicitly, and it would be too complicated to generate new data using these models. One solution can be to model the true distribution $P(x)$ with a simpler distribution.

13.2.1 Latent variables

To do this, observed variables can be related to support variables that simplify the problem, called **latent variables**. These model the causal factors that determine the final observation of my data (for example, in a dog image, the latent variables could be breed, fur color, etc.).

> **Corollary 13.2.1 — Latent variables.** In statistics, latent variables are variables that can only be inferred indirectly through a mathematical model from other observable variables that can be directly observed or measured.

Therefore, a constraint is added to the encoder that forces it to generate so-called latent vectors z (feature vectors that lie in the latent space, i.e., the manifold that I want to learn but don't yet know) that follow a known probability distribution like a Gaussian.
Now to generate a new sample, I just need to generate a new latent vector to pass to the decoder.
In particular, given z the set of latent variables (i.e., the code), the sampling procedure is as follows:

- Sample latent variables $z \sim p(z)$ which is the prior probability and we decide it (for example Gaussian)
- Sample new data $x \sim p(x|z)$ starting from latent vectors

The point is that we want to infer good values for the latent variables based on the observed data.
For inference, Bayes' rule is used:

$$p(z|x) = \frac{p(x,z)}{p(x)} = \frac{p(x|z)p(z)}{p(x)} \tag{13.7}$$

Normally we cannot access $p(x)$ because it is defined on all possible data. Therefore, I cannot access $p(z|x)$ either because it depends on:

$$p(x) = \int_z p(x,z)dz = \int_z p(x|z)p(z)dz \tag{13.8}$$

which is intractable.

13.2.2 Variational inference

However, there is a technique called **variational inference** which consists of approximating these complex distributions with very simple surrogate (auxiliary) distributions defined on auxiliary variables that will be the latent variables.
I approximate the posterior probability $p(z|x)$ with a known surrogate $q_\lambda(z|x)$ where λ are the parameters of the known distribution.
The goal is to make this surrogate distribution as similar as possible to the real one, minimizing their variation (that's why it's called variational inference) i.e., their "distance".

Exercise 13.1 We already know that the way to do this is to use the KL divergence. Remember that:

$$KL(q(z|x)||p(z|x)) = \int_x q(z|x) log \frac{q(z|x)}{p(z|x)} dx \tag{13.9}$$
$$= \int_x q(z|x) log q(z|x) dx - \int_x q(z|x) log p(z|x) dx \tag{13.10}$$
$$= \mathbb{E}_q(log q(z|x)) - E_q(log p(z|x)) \tag{13.11}$$
$$= \mathbb{E}_q(log q(z|x) - log p(z|x)) \tag{13.12}$$
$$= \mathbb{E}_q\left(log q(z|x) - log \frac{p(x,z)}{p(x)}\right) \tag{13.13}$$
$$= \mathbb{E}_q(log q(z|x) - log p(x,z) + log p(x)) \tag{13.14}$$
$$= \mathbb{E}_q(log q(z|x)) - \mathbb{E}_q(log p(x,z)) + \mathbb{E}_q(log p(x)) \tag{13.15}$$
$$= \mathbb{E}_q(log q(z|x)) - \mathbb{E}_q(log p(x,z)) + log p(x) \tag{13.16}$$

This quantity remains intractable due to $log p(x)$. ■

13.2.3 Variational approximation

As consequence, the **Evidence Lower BOund (ELBO)** is defined as follows:

$$ELBO(\lambda) = \mathbb{E}_q(log(p(x,z))) - \mathbb{E}_q(log(q(z|x))) \tag{13.17}$$

Observing better, we notice that:

$$KL(q(z|x)||p(z|x)) = \mathbb{E}_q(log q(z|x)) - \mathbb{E}_q(log p(x,z)) + log p(x) = -ELBO_\lambda + log p(x) \tag{13.18}$$

Which can be rewritten:

$$log(p(x)) = ELBO_\lambda + KL(q(z|x)||p(z|x)) \tag{13.19}$$

Since KL is always non-negative (due to Jensen's inequality), minimizing KL means maximizing ELBO which is tractable.

Exercise 13.2 Proof that minimizing KL means maximizing ELBO:

$$log(p(x)) = log \int_z p(x,z)dz \tag{13.20}$$
$$= log \int_z p(x,z)\frac{q(z|x)}{q(z|x)}dz \tag{13.21}$$
$$= log \int_z q(z|x)\frac{p(x,z)}{q(z|x)}dz \tag{13.22}$$
$$= log\mathbb{E}_q \frac{p(x,z)}{q(z|x)} \tag{13.23}$$
$$\geq \mathbb{E}_q log\frac{p(x,z)}{q(z|x)} \tag{13.24}$$
$$= \mathbb{E}_q logp(x,z) - \mathbb{E}_q logq(z|x) = ELBO_\lambda \tag{13.25}$$
$$\tag{13.26}$$
$$log(p(x)) \geq ELBO_\lambda \tag{13.27}$$

Jensen's inequality states that $\mathbb{E}f(x) \leq f(\mathbb{E}x)$ ■

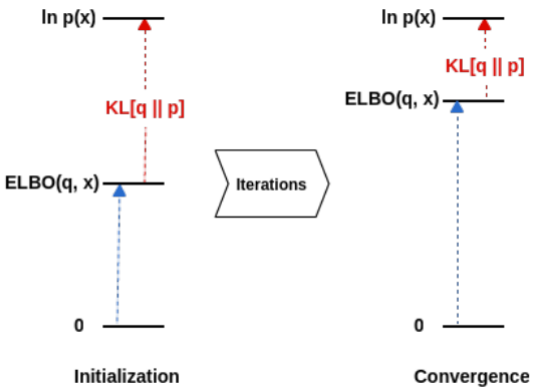


Figure 13.1: KL viewed as gap between $logp(x)$ and $ELBO$ - Visual representation showing how the Kullback-Leibler divergence represents the difference between the log probability and the Evidence Lower Bound

We can further rewrite ELBO as follows:

$$\begin{aligned}
ELBO_\lambda &= \mathbb{E}_q(log(p(x,z))) - \mathbb{E}_q(log(q(z|x))) \\
&= \mathbb{E}_q(log(p(x|z)p(z))) - \mathbb{E}_q(log(q(z|x))) \\
&= \mathbb{E}_q(logp(x|z)) + \mathbb{E}_q(logp(z)) - \mathbb{E}_q(log(q(z|x))) \\
&= \mathbb{E}_q(logp(x|z)) - (\mathbb{E}_q(log(q(z|x))) - \mathbb{E}_q(logp(z))) \\
&= \mathbb{E}_q(logp(x|z)) - \mathbb{E}_q(log(q(z|x)) - logp(z)) \\
&= \mathbb{E}_q(logp(x|z)) - \mathbb{E}_q(log\frac{q(z|x)}{p(z)}) \\
&= \mathbb{E}_q(logp(x|z)) - \int_x q(z|x)log\frac{q(z|x)}{p(z)}dx \\
&= \mathbb{E}_q(logp(x|z)) - KL(q(z|x)||p(z))
\end{aligned} \tag{13.28}$$

From here we can define ELBO for the single data point which we better indicate in the following

way:

$$ELBO = \mathbb{E}_q(logp_\phi(x_i|z)) - KL(q_\theta(z_i|x)||p(z)) \tag{13.29}$$

Where ϕ and θ are respectively the parameters of the decoder and encoder indeed $p_\phi(x|z)$ observes the code z and returns an x (decoding) while $q_\theta(z|x)$ does the opposite (encoding).
Note that both the decoder and encoder don't give me a value but a probability distribution respectively on x and on z.
The problem is that normally the decoder as Loss uses the difference between what is reconstructed and the input, that is the Mean Squared Error:

$$||x_i - \bar{x}||^2 \tag{13.30}$$

However, this is not a probability distribution, to make it such I use a trick:

$$p(x_i|z) = e^{-\frac{||x_i - \bar{x}||^2}{1}} = \mathcal{N}(x_i; \bar{x}, 1) \tag{13.31}$$

This quantity I can consider as a Gaussian centered in $\bar{x}$ and I have transformed the MSE into a distribution $p(x|z)$.
Moreover, I just need to do:

$$logp(x_i|z) = -||x_i - \bar{x}||^2 \tag{13.32}$$

to obtain the reconstruction error with negative sign.

13.2.4 Variational Autoencoders

Finally, we can define a **Variational Autoencoder** as an autoencoder that wants to maximize:

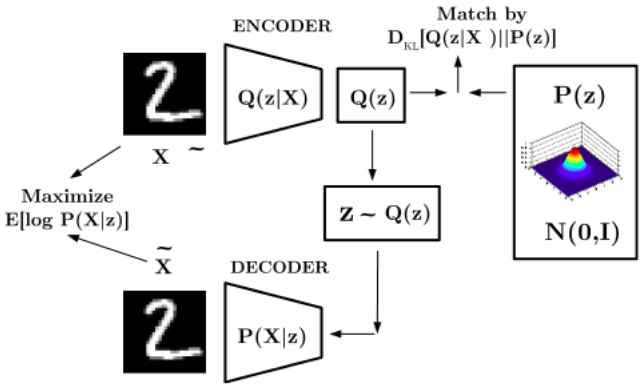
$$\underbrace{\mathbb{E}_{z\sim q(z|x)}[logp(x|z)]}_{\text{Reconstruction term}} - \lambda \underbrace{D_{KL}[q(z|x)||p(z)]}_{\text{Regularization term}} \tag{13.33}$$

The reconstruction term tells us how well we are reconstructing (and is the MSE), the second term wants the generated code to follow a distribution that is as close as possible to our prior.
Let's see specifically the training steps of a VAE:

- Provide input x
- The encoder realizes $q(z|x)$ in particular

$$x \to f_{ENC}(x, \theta) = [\mu(x), \sigma(x)^2] \tag{13.34}$$

  such that $q(z|x)$ is $\mathcal{N}(z; \mu(x), \sigma(x)^2)$
  In practice, the encoder outputs two vectors $\mu(x)$ and $\sigma(x)^2$ that represent the parameters of the estimated posterior distribution, modeled as Gaussian. The encoder thus projects the data into the space of latent variables.
- From $q(z|x)$ I construct $q(z)$ from which to sample z minimizing the regularization term $D_{KL}[q(z|x)||p(z)]$ i.e., trying to make the estimated posterior as similar as possible to the prior (which we decide e.g., Standard Gaussian $\mathcal{N}(0, I)$).
  Being both posterior and prior in Gaussian form, the divergence can be calculated through a closed-form equation.
- Sample a vector $z \sim q(z)$ and pass it to the decoder
- Decoder generates new data from latent variables through $g_{DEC}(z, \theta)$ which plays the role of $p(x|z)$

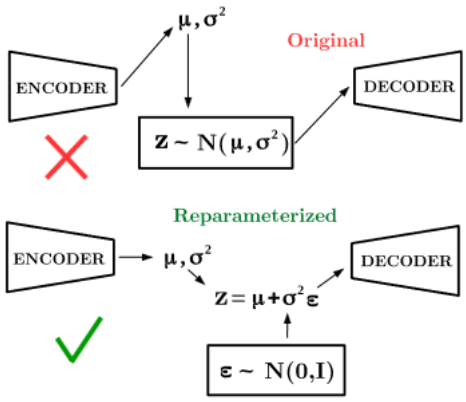


- Calculate $\mathbb{E}[logp(x|z)]$ = -MSE which I want to maximize (watch the minus sign)

However, sampling is not a derivable operation so it breaks the gradient chain. To solve this problem, a trick called **reparametrization** trick is used: I don't sample z directly from q(z) like for example $\mathcal{N}(\mu, \sigma^2)$ but instead I sample a number $\varepsilon \sim \mathcal{N}(0, I)$ and then calculate z as:

$$z = \mu + \varepsilon\sigma^2 \tag{13.35}$$

It doesn't break the gradient chain because it's an input of my network and is not a function of the variables I want to calculate the derivative of.



Exercise 13.3 — Calculation of KL between two Gaussians (not required). Given the following distribution:

$$p(x) = \mathcal{N}(\mu_1, \sigma_1^2) = \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} \quad q(x) = \mathcal{N}(\mu_2, \sigma_2^2) = \frac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \tag{13.36}$$

Compute the KL divergence.

$$KL(p(x)||q(x)) = \int p(x) log \frac{p(x)}{q(x)} dx = \int p(x) log p(x) dx - \int p(x) log q(x) dx \tag{13.37}$$

Properties useful

$$Var[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 \tag{13.38}$$
$$\sigma_1^2 = \mathbb{E}[x^2] - \mu_1^2 \tag{13.39}$$
$$\mathbb{E}[x^2] = \sigma_1^2 + \mu_1^2 \tag{13.40}$$

First term

$$\int p(x) log p(x) dx = \int p(x) log \left((2\pi\sigma_1^2)^{-1/2} exp\Big(- \frac{(x-\mu_1)^2}{2\sigma_1^2} \Big) \right) dx \tag{13.41}$$
$$= \int p(x) log (2\pi\sigma_1^2)^{-1/2} dx + \int p(x) exp\Big(- \frac{(x-\mu_1)^2}{2\sigma_1^2} \Big) dx \tag{13.42}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) \underbrace{\int p(x) dx}_{1} + \int p(x) \Big(- \frac{(x-\mu_1)^2}{2\sigma_1^2} \Big) dx \tag{13.43}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) - \frac{1}{2\sigma_1^2} \int p(x)(x-\mu_1)^2 dx \tag{13.44}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) - \frac{1}{2\sigma_1^2} \int p(x)(x^2 - 2\mu_1 x + \mu_1^2) dx \tag{13.45}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) - \frac{1}{2\sigma_1^2} \Big(\int p(x) x^2 dx - 2\mu_1 \int p(x) x dx + \mu_1^2 \int p(x) dx \Big) \tag{13.46}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) - \frac{1}{2\sigma_1^2} (\mathbb{E}[x^2] - 2\mu_1 \mathbb{E}[x] + \mu_1^2) \tag{13.47}$$
$$= -\frac{1}{2} log(2\pi\sigma_1^2) - \frac{1}{2\sigma_1^2} (\sigma_1^2 + \mu_1^2 - 2\mu_1^2 + \mu_1^2) \tag{13.48}$$
$$= -\frac{1}{2} (1 + log(2\pi\sigma_1^2)) \tag{13.49}$$

Second term

$$\int p(x)logq(x)dx = \int p(x)log\Big((2\pi\sigma_2^2)^{-1/2}exp\Big(-\frac{(x-\mu_2)^2}{2\sigma_2^2}\Big)\Big)dx \tag{13.50}$$

$$= \int p(x)log(2\pi\sigma_2^2)^{-1/2}dx + \int p(x)exp\Big(-\frac{(x-\mu_2)^2}{2\sigma_2^2}\Big)dx \tag{13.51}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2)\underbrace{\int p(x)dx}_{1} + \int p(x)\Big(-\frac{(x-\mu_2)^2}{2\sigma_2^2}\Big)dx \tag{13.52}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2}\int p(x)(x-\mu_2)^2dx \tag{13.53}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2}\int p(x)(x^2-2\mu_2x+\mu_2^2)dx \tag{13.54}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2}\Big(\int p(x)x^2dx - 2\mu_2\int p(x)xdx + \mu_2^2\int p(x)dx\Big) \tag{13.55}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2}(\mathbb{E}[x^2] - 2\mu_2\mathbb{E}[x] + \mu_2^2) \tag{13.56}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{1}{2\sigma_2^2}(\sigma_1^2 + \mu_1^2 - 2\mu_1\mu_2 + \mu_2^2) \tag{13.57}$$

$$= -\frac{1}{2}log(2\pi\sigma_2^2) - \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} \tag{13.58}$$

$$\tag{13.59}$$

At the end

$$KL(p(x)||q(x)) = \int p(x)logp(x)dx - \int p(x)logq(x)dx \tag{13.60}$$

$$= -\frac{1}{2}(1+log(2\pi\sigma_1^2)) + \frac{1}{2}log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} \tag{13.61}$$

$$= \frac{1}{2}log(2\pi) + \frac{1}{2}log(\sigma_2^2) - \frac{1}{2}log(2\pi) - \frac{1}{2}log(\sigma_1^2) - \frac{1}{2} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} \tag{13.62}$$

$$= log(\sigma_2) - log(\sigma_1) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \tag{13.63}$$

$$= log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \tag{13.64}$$

This demonstrate that KL can be expressed with a close equation. ■

13.2.5 Generative Adversarial Networks (only qualitatively asked)

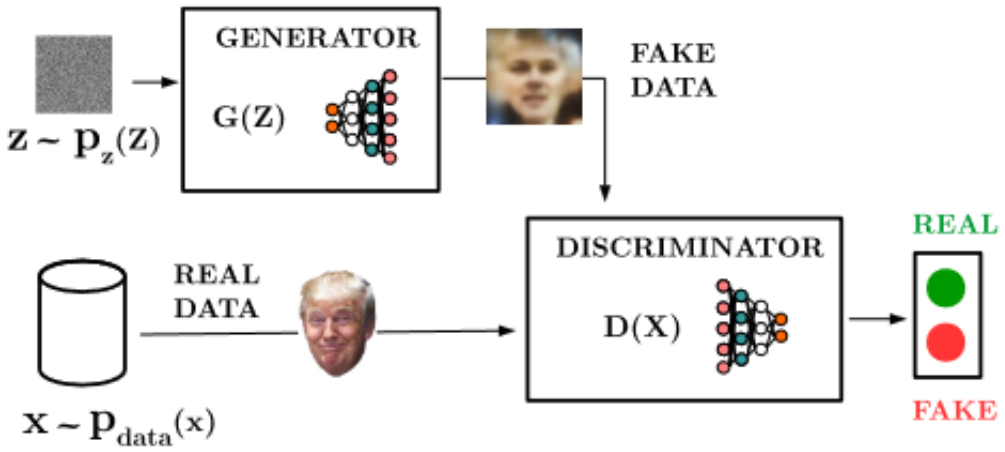
Generative Adversarial Networks (GANs) are another generative approach (purely generative), based on two networks competing with each other in a typical game theory scenario:

- **Generator** G(z): a network topologically similar to a decoder that, given input z sampled from a known probability distribution, tries to generate an object of the same dimension.
- **Discriminator** D(x): a network similar to a classifier that tells us whether certain data is real (taken from training data) or fake (generated by the generator)

It's as if the generator were a criminal creating counterfeit currency and the discriminator were the police trying to detect counterfeit coins. The competition between the two leads them to improve until we have criminals who generate counterfeit currency indistinguishable from real ones. As a side effect, criminals must have learned well how real coins are made to produce high-quality counterfeits.
I want both parties to be good because an inadequate discriminator wouldn't allow the generator to improve properly. But in the end, I want the generator to win the competition.
Let's look at the Loss:

$$\min_G \max_D V(G,D) = \mathop{\mathbb{E}}_{x\sim p_{data}} log\, D(x) + \mathop{\mathbb{E}}_{x\sim p_{model}} log(1-D(x)) \tag{13.65}$$

Where $D(x)$ is the probability that the discriminator predicts that data taken from the real dataset is genuine, while $1-D(x)$ is the probability that the discriminator predicts that generated data is fake. So with min, I want a generator so good that it minimizes the probability that the discriminator predicts that generated data is fake, while with max, I want a discriminator so good that it maximizes its probability of predicting that real dataset data is genuine.
However, I can't do these two things simultaneously, so I alternatively minimize the loss for the generator and maximize the loss for the discriminator. I will therefore have two different model update steps because the gradients will have different directions (one is a max so I'll do gradient ascent and one a min so I'll do gradient descent).



Let's look at the algorithm (not required for the exam):

Algorithm 4 Minibatch Stochastic gradient descent for GANs

for number of training iterations **do**

for k steps **do**

- Sample a minibatch of m noise samples (generated) $\{z^{(1)},\dots,z^{(m)}\}$ from the noise prior $p_g(z)$
- Sample a minibatch of m real samples $\{x^{(1)},\dots,x^{(m)}\}$ from the real data distribution $p_{data}(x)$
- Update the discriminator by performing gradient ascent:
  $\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[logD(x^{(i)})+log(1-D(G(z^{(i)})))\right]$

end for

- Sample a minibatch of m noise samples (generated) $\{z^{1},\dots,z^{m}\}$ from the noise prior $p_g(z)$
- Update the generator by performing gradient descent:
  $\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}log(1-D(G(z^{(i)})))$

end for

Pay attention to the fact that the first gradient is taken with respect to the discriminator parameters θ_d while the second with respect to the generator θ_g. So we can update the two gradient only one at time because they have different direction (min and max).
The generator must not become too good at the beginning, otherwise the discriminator will never improve. Implicitly within the network, I have learned the probability density p(x) from which the data was generated, and the discriminator is therefore a "probe" to understand if data is inside this or not; in fact, by continuously saying real/fake, I cover all its space.
This relates to Monte Carlo methods (such as MCMC, **Markov Chain Monte Carlo**) which are tools for calculating quantities using these "probes" that tell us whether a sample is inside or outside a desired curve. For example, if I fire cannonballs in the dark, depending on what noise I hear, I can understand if the ball ended up in the lake or not, and by firing many, I could approximate the area of this lake without knowing its shape (I'm in the dark). This process must be repeated many times and stochastically.
Let's conclude with the pros and cons of GANs.
Pros:

- They can leverage backpropagation
- The Loss is learned instead of manually selected
- They don't require MCMC methods

Cons:

- Difficult to train (minimax optimization tools are immature)
- Need to manually "babysit" during training (pay attention to eventually domination of one model)
- No quantitative metrics to evaluate and compare them with other models, only qualitative ones (wouldn't know how to measure the beauty of a generated image, no guarantee of quality, subjective judgement)

IV Reinforcement Learning

14. Reinforcement Learning

Reinforcement learning [2] is one of the three main paradigms of machine learning, along with supervised learning and unsupervised learning. The differences between reinforcement learning and other machine learning paradigms are the followings:

- There is no dataset but an environment
- There is no supervision but reward signals in favor of an agent based on its behavior within an environment
- Feedback (rewards) are not instantaneous (there is delay)
- Tasks have a strongly sequential nature, so actions are strongly correlated with time (therefore data is not i.i.d)
- Agents are **active** and by acting they modify the environment they are in

To understand better we have to give some definitions:

Definition 14.0.1 — Reward. A reward R_t is a feedback signal and in particular is a scalar number that indicates how well the agent is performing at time t.

Definition 14.0.2 — Episode. An episode is a sequence of steps that reach a conclusion (not always there is an end) which could be achieving a goal or failure. For example, an episode could be reaching the end of a Super Mario level with the maximum number of coins or in the shortest possible time.

The goal of agent is to maximize cumulative reward over and episode. To maximize total future reward, the agent will have to select which actions to take considering that these might have long-term consequences and that rewards might be delayed, so it might be better to sacrifice immediate rewards to have better rewards in the long term.

14.1 Agent&Enviroment

The agent is the subject that interacts with the environment during the episode. At each step t:

- Receives an observation O_t
- Receives a scalar reward R_t
- Executes an action A_t

Instead the environment:

- Receives an action A_t
- Emits an observation O_{t+1}
- Emits a scalar reward R_{t+1}

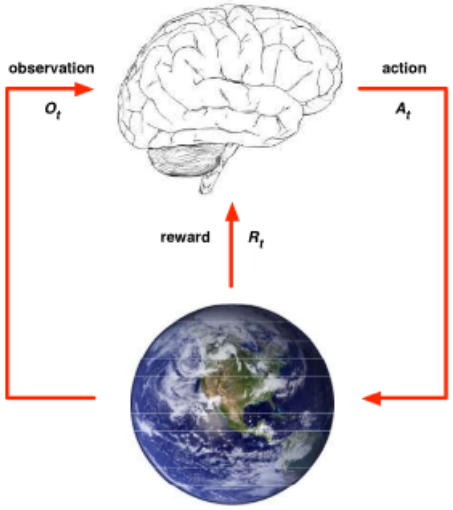


Figure 14.1: Agent and environment interaction diagram showing the cycle of observation, action, and reward between agent and environment

Definition 14.1.1 — History. An agent will therefore have a sequence of observations, rewards, and actions up to time t called history (or episode trajectory):

$$H_t = O_1, R_1, A_1, \ldots A_{t-1}, O_t, R_t \tag{14.1}$$

Definition 14.1.2 — State. The state of an agent S_t^a is the information used to undertake the next action. It is a function of history or a subset of it.

$$S_t^a = f(H_t) \tag{14.2}$$

It is therefore a more restricted, more manageable, and more usable form of history, for example, the last 3 observations, rewards, and actions or their average.

We speak of **full observability** when the agent can see the environment state directly while partial observability when it sees it indirectly (based on observations and rewards emitted by the environment).

An example of **partial observability** is an agent trying to guess what cards an opponent has in their hand while watching them play briscola, while an example of full observability is a game of briscola with open cards.

In this book we will focus on only partial observability because it is the most difficult and complex scenario.

Definition 14.1.3 — Policy. The policy defines the behavior of an agent and is a function that maps the state space to the action space.

Can be of two types:

- **Deterministic**

$$\pi(s) = a \tag{14.3}$$

  Given a state, I always take the same action.
- **Stochastic**

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{14.4}$$

  Given a state, I have a probability distribution over the actions I can take.

In general, the probabilistic policies are better because they allow better exploration of the state space thanks to the introduction of randomness in the process.

Definition 14.1.4 — Return. The return G_t is the sum of all discounted rewards from time t onwards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \tag{14.5}$$

The discount $\gamma \in [0,1]$, applied to future returns, weighs how important the reward is for me, after $k+1$ steps indeed in the sum I have $\gamma^k R_{t+k+1}$.
In other words, the discount weighs how much I care about receiving a reward ahead in time and if I have γ close to 0 I will have a myopic evaluation therefore a short-sighted agent behavior while with γ close to 1 I will have a far-sighted evaluation (long-term strategy).
With $\gamma = 1$ I will have a so-called **undiscounted** process and it's not good for example for processes that never end.

Definition 14.1.5 — Value function. The value function is a prediction of future reward (return) used to know how good it is to be in a certain state from the return point of view.

There exist two types:

- State-value function
- Action-value function

Corollary 14.1.1 — State-value function. The state-value function is the expectation of return starting from a state s following a policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{14.6}$$

It's a sort of average of the returns I have observed starting from state s following policy π. The policy indeed determines what action to take starting from a certain state and in response to the action I will receive a certain return.

Corollary 14.1.2 — Action-value function. The action-value function is the expectation of return starting from a state s, taking an action a and following a policy π.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{14.7}$$

In this case I specify the starting action as well as the state.

14.2 Bellman expectation equation

The Bellman expectation equation decomposes the state-value function into immediate reward R_{t+1} and discounted value of the state-value function for the next state $\gamma v_\pi(S_{t+1})$:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots)|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma\mathbb{E}_\pi[E_\pi[G_{t+1}|S_{t+1} = s']|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma\mathbb{E}_\pi[v_\pi(S_{t+1})|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{14.8}$$

We made use of the linearity of expected value and the Law of total expectation:

$$\mathbb{E}[Y|X] = \mathbb{E}[\mathbb{E}[Y|X,Z]|X] \tag{14.9}$$

where:

- $Y \to G_{t+1}$
- $X \to S_t = s$
- $Z \to S_{t+1} = s'$

Therefore:

$$\mathbb{E}_\pi[G_{t+1}|S_t = s] = \mathbb{E}_\pi[\mathbb{E}_\pi[G_{t+1}|S_t = s, S_{t+1} = s']|S_t = s] \tag{14.10}$$

But G_{t+1} depends only on S_{t+1} so we remove the condition $S_t = s$

$$\mathbb{E}_\pi[G_{t+1}|S_t = s] = \mathbb{E}_\pi[\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']|S_t = s] = \mathbb{E}_\pi[v_\pi(S_{t+1})|S_t = s] \tag{14.11}$$

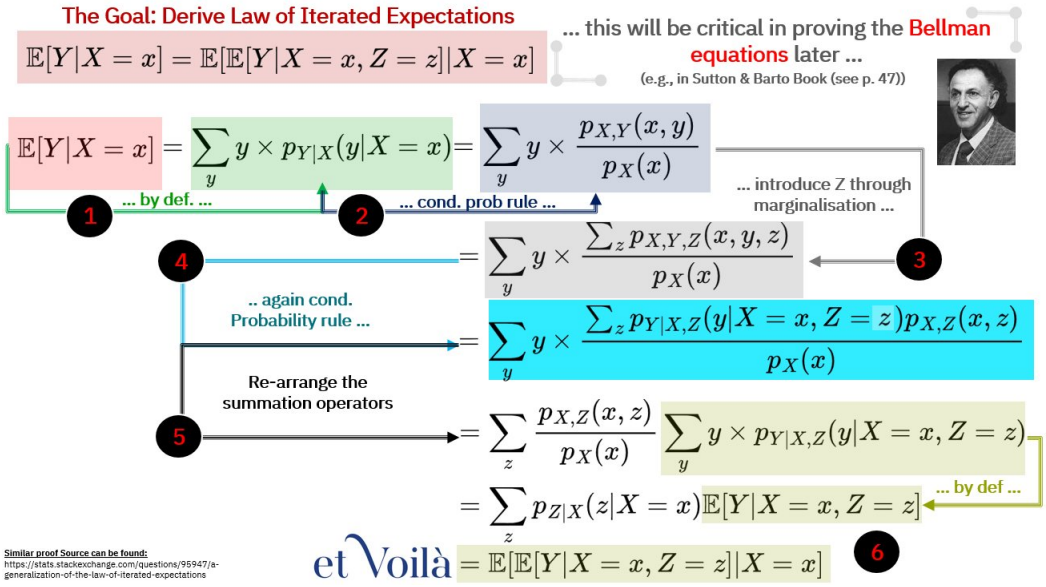


Figure 14.2: Demonstration of the law of total expectations showing key mathematical steps of the proof

Similarly, I can decompose the action-value function:

$$\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\dots|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}+\gamma(R_{t+2}+\gamma R_{t+3}+\dots)|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}+\gamma G_{t+1}|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}|S_t=s,A_t=a]+\gamma\mathbb{E}_\pi[G_{t+1}|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}|S_t=s,A_t=a]+\gamma\mathbb{E}_\pi[E_\pi[G_{t+1}|S_{t+1}=s',A_{t+1}=a']|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}|S_t=s,A_t=a]+\gamma\mathbb{E}_\pi[q_\pi(S_{t+1},A_{t+1})|S_t=s,A_t=a] \\
&= \mathbb{E}_\pi[R_{t+1}+\gamma q_\pi(S_{t+1},A_{t+1})|S_t=s,A_t=a]
\end{aligned} \tag{14.12}$$

Definition 14.2.1 — Model. Explicit representation of how the environment works that predicts what it will do at the next step:

- $\mathscr{P}$ predicts the next state

$$\mathscr{P}^a_{ss'} = P[S_{t+1}=s'|S_t=s,A_t=a] \tag{14.13}$$

  It's a state transition matrix that can help in decision making.
- $\mathscr{R}$ predicts the next immediate reward

$$\mathscr{R}^a_s = \mathbb{E}[R_{t+1}|S_t=s,A_t=a] \tag{14.14}$$

We won't use these concepts because we'll focus on model-free methods that don't explicitly build a model of the environment.

14.3 Model-free prediction

Also famous as policy evaluation, they are methods aimed at estimating a value function given a policy in an unobservable environment:

- **Monte Carlo learning**
- **Temporal Difference learning**

14.3.1 Monte Carlo learning

Monte Carlo methods are characterized by the following properties:

- Learn directly from complete episodes (that have an end) of their experience, so they can only be applied to episodic environments (there exists a terminal state)
- Are model-free: they don't need explicit knowledge of the environment
- Are based on a simple idea that value of a state = average of the return observed starting from that state

The objective therefore is to learn v_π from episodes of experience and under policy π. So I must wait for the end of the episode.

The policy evaluation of Monte Carlo methods is based on empirical average rather than expected value of the return. So until the episode ends, I remember all the states I've visited (actions don't matter to me) and the rewards I've obtained instant by instant. Every time I visit a new state s, I keep track of it by incrementing a counter and incrementing the total return:

$$N(s) \leftarrow N(s)+1 \tag{14.15}$$

$$S(s) \leftarrow S(s) + G_t \tag{14.16}$$

The value will be the total return I obtained passing through that state divided by the number of times I passed through it:

$$V(s) = \frac{S(s)}{N(s)} \tag{14.17}$$

By the law of large numbers, if I sample many episodes:

$$V(s) \rightarrow v_\pi(s) \text{ for } N(s) \rightarrow \infty \tag{14.18}$$

So this formula is used when I want to calculate V after many different episodes.
If we wanted to analyze the individual incremental updates of $V(s)$ within the same episode, then for each state S_t with return G_t:

$$N(S_t) \leftarrow N(S_t) + 1 \tag{14.19}$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{14.20}$$

That is, I add to the current value of the value function a quantity proportional to the error I'm making right now in evaluating state S_t. Indeed, G_t is the true return while $V(S_t)$ is what I expected to observe.
Often $\frac{1}{N(S_t)}$ is replaced with a fixed α for all states:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{14.21}$$

Which, if you notice, looks very similar to the learning rate and modulates how "strongly" I go in the direction of the error, that is, how quickly I move the value function in the direction of the true return G_t. It's also called **confidence**.
Being a moving average, I forget old episodes when the value function was still poorly estimated.

14.3.2 Temporal Difference learning

TD is another method that, instead of Montecarlo, can learn (update the value function) before the end of the episode (it's said that TD learns online). So the update rule is the following:

$$V(S_t) \leftarrow V(S_t) + \alpha(\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t)) \tag{14.22}$$

That is

$$V(S_t) \leftarrow V(S_t) + \alpha\delta_t \tag{14.23}$$

Where δ_t is called **TD error** and $R_{t+1} + \gamma V(S_{t+1})$ is the **TD target**
In this case, instead of waiting for the end of the episode and calculating the true return, I evaluate the estimate of the return in the next state (thanks to the Bellman Expectation Equation) and move in its direction. So the difference with MC are the followings:

- TD can learn from incomplete sequences in continuous environments (non-episodic i.e., without an end)
- The true return used in MC, G_t, is an unbiased estimate of the true state-value function $v_\pi(S_t)$
- The true TD Target used in TD, $R_{t+1} + \gamma v_\pi(S_{t+1})$, is an unbiased estimate of the true state-value function $v_\pi(S_t)$

- The estimated TD Target used in TD, $R_{t+1} + \gamma v(S_{t+1})$, is a biased estimate of the true state-value function $v_\pi(S_t)$
- MC is based on the final return which has high variance because it depends on many transitions, actions, rewards so to have a correct estimate I need many episodes
- The TD target depends on a single transition, a single action and a single reward so it has lower variance than the final return and learns in less time

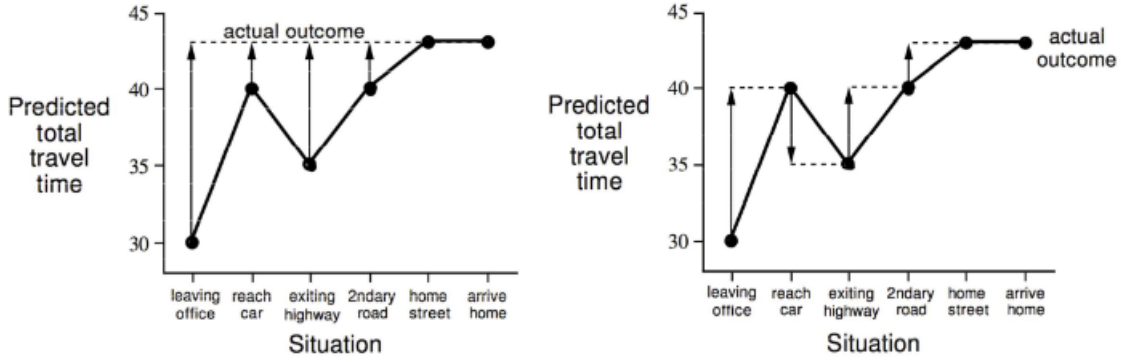


Figure 14.3: Comparison between Monte Carlo (MC) and Temporal Difference (TD) learning approaches showing their key differences in learning patterns

14.4 Model-free control

Methods that do not try to evaluate a policy but to find a good one in an unobservable environment (fixed action-value function)

- On-Policy Monte-Carlo Control
- On-Policy Temporal-Difference Control (SARSA)
- Off-Policy Learning (Q-learning)

Given a policy, we currently know two ways to evaluate it:

$$v_\pi(s) \tag{14.24}$$

$$q_\pi(s,a) \tag{14.25}$$

A policy can be improved starting from the value function by acting greedily:

$$\pi' = greedy(v_\pi) \tag{14.26}$$

$$\pi' = greedy(q_\pi) \tag{14.27}$$

These new greedy policies select among available actions the one that maximizes future rewards. Since the value function estimates future returns, acting greedily doesn't mean having a policy that only thinks about the short term.
In model-free contexts, we only have access to $Q(s,a)$ because to be greedy with respect to the state-value function, we need the model and particularly need to know which action leads to which state (transition probabilities).
However, we don't just want the agent to maximize rewards (be greedy), but we also want it to explore all possible states to cover the state space and be confident about its policy.

To force exploration, there's the **ε-greedy** policy where we choose a threshold $\varepsilon \in [0,1]$, then each time we sample a number between 0 and 1, and if it's less than ε, we choose a random action (explore), otherwise we choose the greedy policy.

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{m} + (1-\varepsilon) & \text{if } a^* = \underset{a \in A}{argmax}\, Q(s,a) \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases} \tag{14.28}$$

That is, given m actions each with non-zero probability, I have probability ε of taking one at random and probability $1-\varepsilon$ of taking a greedy one.

By changing ε, I can be more or less explorative of the state space.

For example, if I have two doors and I open the left one and get reward 0, and then open the right one and get reward +2, a completely greedy policy will continue to choose the right door until it gives me a value less than zero. Therefore, I might never open the left door again just because the first time I observed a zero, but that zero may not be representative of the average I would get by opening the left door many times. In fact, maybe opening it another time I could have gotten +20, which is much greater than +2.

Being greedy leads to excluding trajectories in the state space that could have much value, which is why the ε-greedy policy is introduced (completely greedy policies can collapse into wrong deterministic behavior).

For model-free control methods, we generally have two possibilities:

- On-policy learning: learns the best policy based on its own episodes
- Off-policy learning: learns the best policy based on episodes from another policy

14.4.1 Monte Carlo policy iteration

Iteratively repeats two steps:

- Policy evaluation
- Policy improvement

Starting from a random action-value function Q with a random policy π, I perform MC policy evaluation of Q with policy π and when it converges to the true value of the action-value function for that policy, I change policy in favor of an ε-greedy policy with respect to Q. Then I start evaluating the latter with MC policy evaluation and continue until I effectively converge to the optimal action-value function and policy q_*, π_*

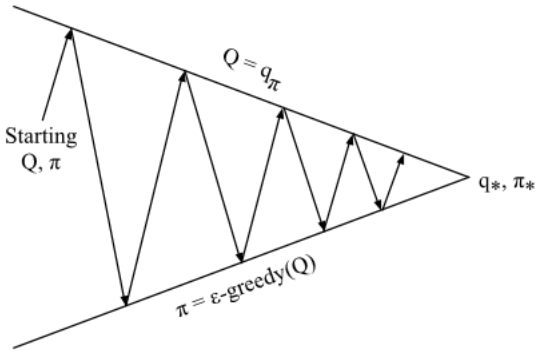


Figure 14.4: MC policy iteration process diagram

14.4.2 On policy Monte Carlo Control

I can avoid converging to the true value of the action-value function for that policy, changing to an ε-greedy policy with respect to Q at the end of every episode and still converge to the optimal action-value function and policy q_*, π_*.

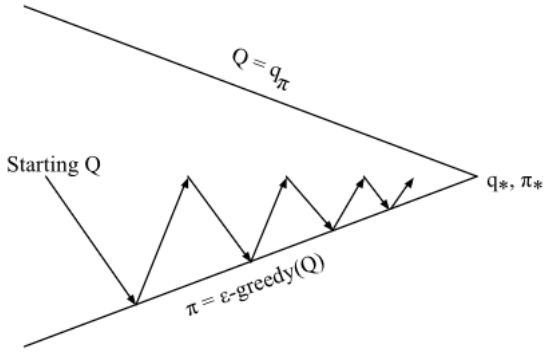


Figure 14.5: Monte Carlo control diagram

14.4.3 On policy Temporal Difference Control (SARSA)

We've already seen how TD has some advantages over MC (lower variance, it's online, and it's compatible with incomplete sequences), so we can think of doing TD policy evaluation of Q(s,a) instead of MC while maintaining the ε-greedy policy improvement.
This technique is also called **SARSA** because:

- We are in a state S
- We sample action A following our policy
- We observe a reward R
- We end up in state S'
- We sample an action A' to take at a later time (we ask ourselves what action we would take starting from S' without executing it immediately)

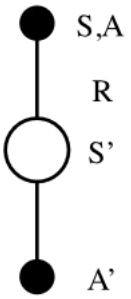


Figure 14.6: SARSA process diagram

The update rule is the following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma Q(s',a') - Q(s,a)) \tag{14.29}$$

It's the same as seen before for the state-value function in TD, meaning we move our Q towards the TD target which is the reward + the discounted version of Q in the next instant. We need to sample A' precisely because working on Q we need both the state and the next action.

This is called On-policy evaluation with SARSA. Here's the algorithm:

Algorithm 5 SARSA

Initialize $Q(s,a), \forall s \in S, a \in \mathscr{A}(s)$ and $Q(terminal,\cdot) = 0$ **for** each episode **do**
Initialize S Choose A from S using policy derived from Q (e.g., ε-greedy) **for** each step of the episode **do**
Take action A, observe R,S' Choose A' from S' using policy derived from Q (e.g., ε-greedy)
$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma Q(s',a') - Q(s,a))$ $S \leftarrow S', A \leftarrow A'$ **end for**
end for

14.4.4 Off-policy learning (Q-learning)

It consists of evaluating a policy $\pi(a,s)$ by calculating $v_\pi(s)$ or $q_\pi(s,a)$ following the behavior of another policy $\mu(a|s)$. Therefore, we sample episodes from this other behaviour policy:

$$\{S_1, A_1, R_1, \ldots, S_T\} \sim \mu \tag{14.30}$$

This opens the door to so-called imitation learning, as we can learn by observing how a human or another agent performs a task and imitate it, or we can reuse experience generated by past policies:

$$\pi_1, \ldots, \pi_{t-1} \tag{14.31}$$

We can also search for the optimal policy (purely greedy) by exploring the behaviour policy or learn various policies (not just the optimal one) from the behaviour policy.
We are interested in learning an optimal policy on the shoulders of another policy that explores the entire state space, and this is what happens in Q-learning.
It is a temporal difference algorithm, meaning that at each step we update our action-value function, and in particular, the next action $A_{t+1} \sim \mu(\cdot, S_t)$ is chosen from the behaviour policy, but we consider an alternative $A' \sim \pi(\cdot, S_t)$ sampled from the policy we want to optimize.
We update the action-value function with the following update rule:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1},A') - Q(S_t,A_t)) \tag{14.32}$$

The difference from SARSA is that in the latter, A' was always sampled from μ while here it's from π.
How to choose the two policies?
The target policy π is purely greedy with respect to Q(s,a):

$$\pi(S_{t+1}) = \underset{a'}{argmax}\, Q(S_{t+1},a') \tag{14.33}$$

Therefore, we always sample the action that maximizes our return.
The behaviour policy μ is ε-greedy with respect to Q(s,a), thus more explorative.
The Q-learning target becomes:

$$\begin{aligned} R_{t+1} + \gamma Q(S_{t+1},A') &= R_{t+1} + \gamma Q(S_{t+1}, \underset{a'}{argmax}\, Q(S_{t+1},a')) \\ &= R_{t+1} + \gamma \max_{a'} Q(S_{t+1},a') \end{aligned} \tag{14.34}$$

Here's the Q-learning algorithm:

Algorithm 6 Q-learning

Initialize $Q(s,a), \forall s \in S, a \in \mathscr{A}(s)$ and $Q(terminal,\cdot) = 0$ **for** each episode **do**
Initialize S **for** each step of the episode **do**
Choose A from S' using policy derived from Q (e.g., ε-greedy) Take action A, observe R,S'
$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma \max_{a'} Q(S_{t+1}, a') - Q(s,a))$ $S \leftarrow S'$ **end for**
end for

What changes compared to SARSA is that before, A and A' were chosen from the same policy, and moreover, A' is chosen in a purely greedy way as can be understood from the max in the update step, and after being used in the calculation, we forget about it. Therefore, A' is used only to calculate the target but not to move, meaning we move following an ε-greedy policy and are very explorative but meanwhile learn with respect to a purely greedy policy.

14.5 Function Approximation (not required)

Until now, state-value functions and action-value functions were seen as tables where we have different values for each state and for each state-action pair respectively. However, there are problems where the number of states and actions is very high or where the concept of state is continuous (for example, helicopter control) and even though it can be discretized, the number of possible configurations is too large to be represented in tabular form (we would need to visit every possible state and action). Therefore, instead of lookup tables, we can use function approximators parameterized by learnable parameters w:

$$\hat{v}(s,w) \approx v_\pi(s) \tag{14.35}$$

$$\hat{q}(s,a,w) \approx q_\pi(s,a) \tag{14.36}$$

This topic was not covered in this book.

V

Bibliography

Bibliography

References

[Cal23] Simone Calderara. *Slide of Machine and Deep Learning*. UniMORE, 2023 (cited on page 23).

[Flo24] Aldo Flotta. *Appunti di Machine e Deep Learning*. UniMORE, 2024 (cited on page 143).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cited on page 9).